# Roombot modules - Kinematics Considerations for Moving Optimizations

Mikaël Mayer

January 9, 2009

**Abstract**

The roombots are modular robots being developed by the BIRG team, as the succession of many worldwide attempts to build robots that humans will be using in their tomorrow's house. The new possibilities that they are offering, such as these 6 degrees of freedom when combined in pairs make them good candidates to fulfill the promises that we are expecting from such domestic robots.

Self-reconfiguration is one of the main keywords for these robots. And robustness as well. But for the roombot modules, we still lacked a way to understand their moves good enough so that we would be able to give them complex and automatized tasks.

In this project, we will present the clarification of concepts such as the possible moves in space of the roombot modules, the resolution of inverse kinematics for a lot of potential configurations, and some experiments where we coordinate all previous results and interface them with Webots, a robot-simulation environment.

# Contents

# Acknowledgement

. . . Special thanks to :

My supervisor Alexander Spröwitz, who focused me on the real goals of my project, gave me a lot of information that I needed, encouraged me to do the best that I could,

Alessandro Crespi, who helped me a lot during the developement part and also for the configuration of my linux environment,

Alexandre Tuleu, who enhanced a lot my different C++ classes, who designed a program to create roombot worlds that I used for my final scene, and who directed me to useful books,

Jocelyne Lotfi, who, apart from helping me in my work at all scales, showed an exemplary patience while testing my roombot controllers on her own roombots,

Julia Jess, who directed me to good information sources,

Yvan Bourquin, who gave me some useful tips on Webots,

Auke Ijspeert, who supervised my project at high level,

and also thanks to all others who did deserve it and I did not mention here.

# Chapter 1

# State of the art

## 1.1 Context of the project

### 1.1.1 Biologically inspired robotics

When one considers for example the self-repair mechanism of the human body, they have no choice but to marvel at the complexity which makes this functionality feasible. Looking into the details, one sees the great amount of needed work : coagulating cells for blood injuries, the white blood cells to defend against external agents, stem cells to regenerate all kind of tissues ...

Such examples are part of the reason why in our biologically inspired robotic group [1] we want to reproduce as far as possible the best of the capabilities of the living beings for our robots. Such capabilities are "neural mechanisms underlying movement control" [1], "learning" [1], adaptability, self-repairability, sociability, and much more.

In general, existing robots are often specialized, so it becomes easy to observe, to model and to reproduce movements, to launch some learning capabilities – which in fact often means optimization in a multi-dimensional space of capabilities –, and to even explore some concepts of sociability like in competitive and cooperative games.

Robots are meant to help humans, like an extension of our capabilities. To check if a particular robot is working well, the basic choice is to design other independent robots able to cope with the problems of the previous robot, to solve them or to notify them to humans. But what if these last robots are not working well either ? We could create a chain of robots supervising on each other, but this is not "scalable".

### 1.1.2 The modular approach

The problem is not only a matter of reliability. If a robot has only one arm, whatever is the force of each module, it might not be able to lift up a cube which is on a slippery surface if the arms are too small (see Figure 1.1). In

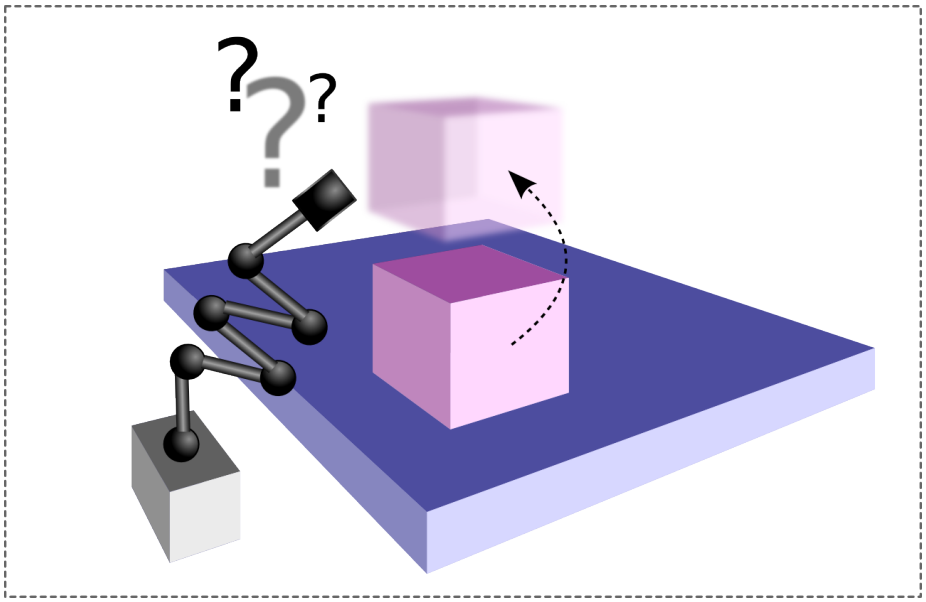

Figure 1.1: This one-arm robot cannot visibly lift the cube.

order to reach such a goal, it should itself put some of its degrees of freedom into another arm (see Figure 1.2). It means that it should be *self-reconfigurable*, i.e. each part of the robot should be able to change its position, in others words the robot should be *modular*.

### Properties of the modular approach

The characteristic of what can be called a *modular* robot is the following. It is build out of modules, which all share identical characteristics – like the set of spheres plus the forearms in the example of Figure 1.2 –, which are able to either work together or to act in totally independant ways.

The advantages provided by such modules are robustness against failure as each module is replaceable, adaptability to new situations, self-reconfiguration, multi-tasking and much more [22].

### History of modular robots

The first modular robot is believed to be CEBOT (CEllular roBOT), created by Toshio Fukuda in 1989 [8]. It freed the concept of the possibility of such modular robots. It was thus followed by no less than 30 different kind of modular robots in between. One can cite such robots as M-TRAN [18], the Dof box [5], and the YaMoR [17, 27], developed here by the BIRG.

All of these modular robots are somewhat unique, because they have differents shapes, different reconfiguration types, not necessarily the same number of

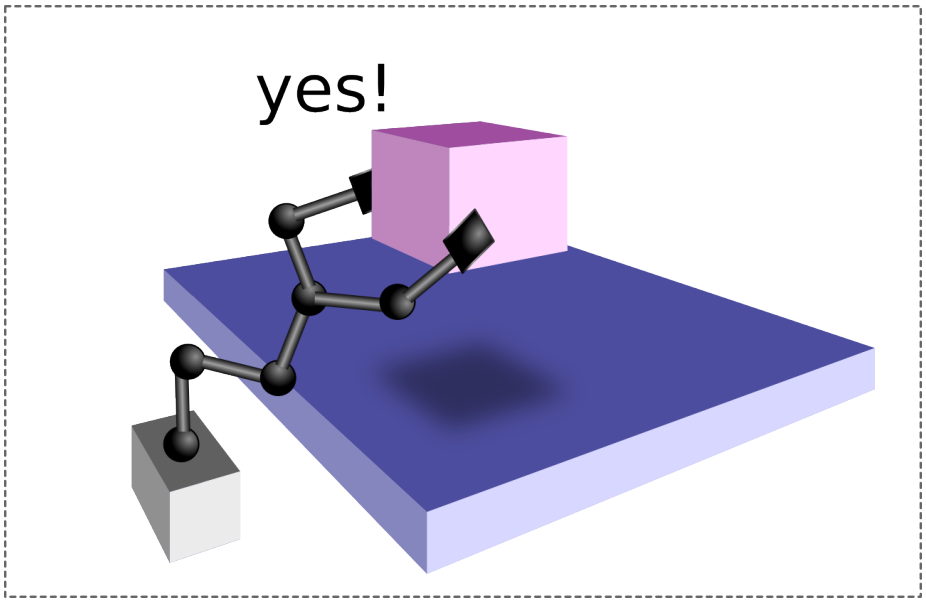


Figure 1.2: This reconfigurated two-arms robot can solve this problem.

degrees of freedom, weight, size, functionalities, etc. It shows that the research is still very active in this domain. In 20 years, it is likely that modular robots will have been commercialized, with applications such as Catoms to reproduce color and forms of given objects[3], space-exploration robots or even domestic modular "roombots", the type of modular robots which we are developing here.

### 1.1.3 Why roombots?

Roombots are being designed by the BIRG team [7]. Sample of the original idea are presended on figures 1.3 and 1.4. The name of roombot was first applied to the YaMoR modules and then the DOF-BOX, because their usage were meant to be in a domestic room.

But now the name fully applies to the module presented on Figure 2.1. One module is the reunion of two shapes, each one looking like the intersection of a sphere of 14cm diameter and a cube of 11cm side, with a separating plane along the median plan of the greatest diagonal of the cube.

It already looks promising, when one looks at the following expected behaviors:

- The edges of the module are soft, so it is harmless and solid.
- There are 3 continuous rotating degrees of freedom (DOF) per module, so with only two modules, we are reaching 6 DOF, the minimum number of DOF to make arbitrary positions possible [4].

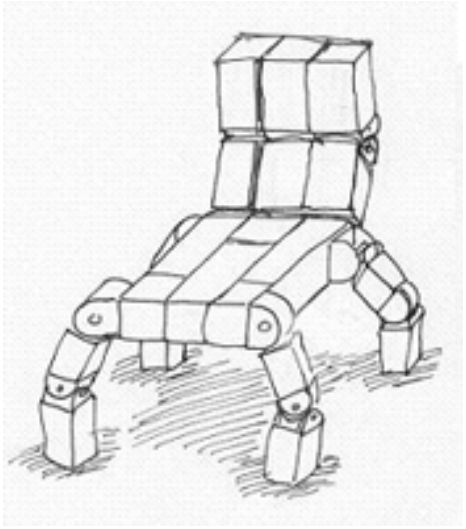

Figure 1.3: The original sketch of the roombots [7].

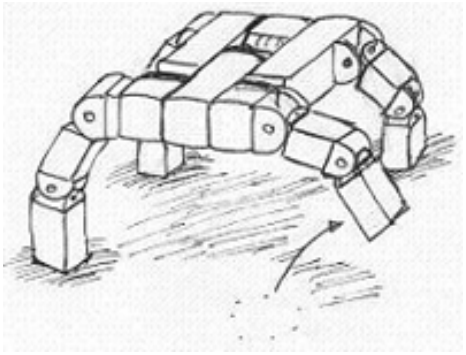

Figure 1.4: The roombots create a chair [7].

- No connectors are appearent, they are retractable, and on the 10 faces.
- One module can lift itself plus one other, because of their good torque.
- The position is controlled by position in order to be precise.

### 1.1.4 Complementary approaches to the roombots

Roombot reconfiguration is currently being handled by simplifying the space of roombots motions to 90° motions, to have better heuristics to resolve this NP-hard problem [13].

Early this year there has been some work on the locomotion of the roombots, by exploring the multi-dimensional space of roombots periodic movements and maximizing the average speed of given structures.[14]. The continuation of this highly bio-mimetic approach is about central pattern generators (CPG) [20], an approach which already showed good results [16, 15].

For the moment these approaches do not have a real kinematic model. It means that we are not fully using all their DOF (degrees of freedom) so we do not exactly know what are their power and their limitations.

Two roombot modules have together up to 6 DOF. This was exactly designed so that two roombot modules together could reach arbitrary positions and orientations. This is what motivated this project.

### 1.1.5 The initial goals of our project

Here are the objectives.

> “Aim of this project is to establish a forward kinematics representation for the RB modules, and to find appropriate solving strategies for the inverse kinematics. The kinematics framework can then be applied to several situations both in simulation using Webots [21] or the real modules. Task list:
>
> 1. An extensive literature research about Modular Robots, Self-Reconfiguration Modular Robots, and kinematics models. Special focus are the existing projects and publications of the RB modules.
> 2. Establish a forward kinematics model for the Roombots modules. Find ways to solve the inverse kinematics. Dynamics can be neglected for the beginning, however can be included.
> 3. Check for the reachable space of the RB modules, e.g. regarding two RB modules in series.
> 4. Discretize the search space by discretizing the joint angles of the RB modules. Check for simplification, possible configurations...

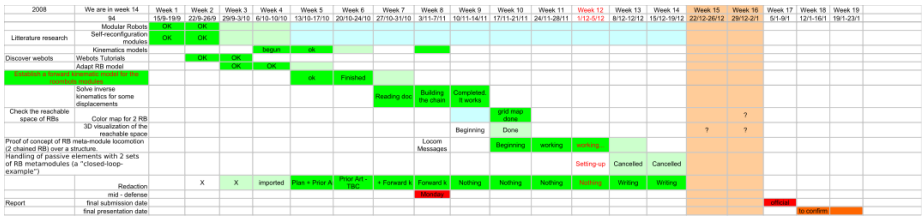

Figure 1.5: The timetable describing the objectives and the due dates.

5. Show a proof of concept of RB meta-module locomotion (2 RB modules in series) over a specifically selected passive structure.
6. Handling of passive elements with 2 sets of RB meta-modules (a "closed-loop-example")"

We will go through all these objectives, except the "handling of passive elements with 2 sets of RB meta-modules", the "closed-loop-example". Indeed, we discovered that we did not have the time to implement such great feature, but we worked in a way so that our successors would have a chance to do it.

### Working methods

First we designed a timeline in which we put deadlines for each of our objectives. After one revision, it gave the result on figure 1.5, which has been followed without problems.

Second we were writing a daily report for ourselves, in order to keep references to everything that we discovered and for further use.

And third we wrote weekly reports to state the progresses done on the project, so that we could have an objective view of it.

## 1.2 Theoretical background

### 1.2.1 Kinematics models

Let us simplify the notion of kinematic models to only rotations and translations, without taking into account velocities, forces, accelerations or dynamic properties. Let us keep translational joints aside as well. A kinematic model of a set of objects linked by some joints has then the following properties:

- First, for each independent moving object, we define a "frame", or referential, composed of an orientation and a position which are stuck to this moving part (see Figure 1.6).
- Second, for each *joint* between two moving objects, we define a parameter (which can also be a vector), and we define a "frame" within each moving

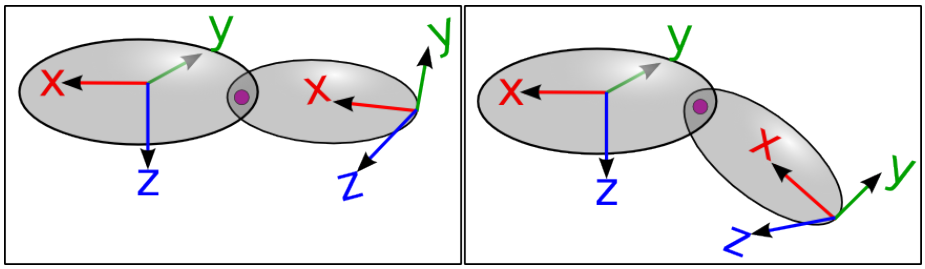


Figure 1.6: Referentials stuck to objects

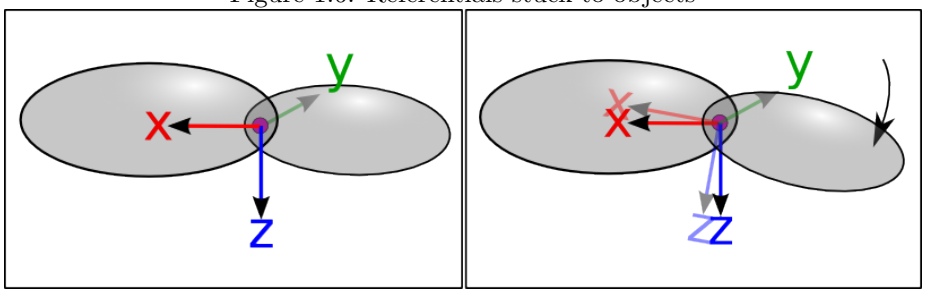


Figure 1.7: The joint frames coincides for a default parameter, and else only differ by a rotation.

part, so that they coincide for the default value of the parameter – typically zero – and else only differ by an orientation (see Figure 1.7)).

But these two properties are not sufficient to have a good model, because there are too many possibilities. Fortunately, people already worked in this domain and found some models in which all those referentials (frame of references) are described by a small number of parameters and are easy to locate and to guess.

Instead of considering our kinematic problem as a "graph" or a "tree" where the objects are the nodes and the links are the edges, we simplify it to a chain of "bones" linked by rotational joints around Z-axis, and we choose to take the D-H convention (Denavit and Hartenberg) [4] to place the referentials. Figure 1.8 explains the main idea and presents the four parameters used in the convention : $a_i$, $\alpha_i$, $d_i$, and $\theta_i$, as long with the corresponding referentials. In these notations, $R_0$ would be the first referential, and $R_n$ the last one.

If the same chain is considered in reverse order, we call it the *inverted chain*.

If we use a library such as KDL [19], there already exists some functions that take those parameters in argument and are building the corresponding chains.

Given these parameters, we can either compute the general form of the matrix [4], or give a factorized form (Figure 1.9). In fact, we can totally separate the static transformation given by the parameters $a_{i-1}$, $\alpha_{i-1}$, $d_i$ and the real rotation given by the parameter $\theta_i$.

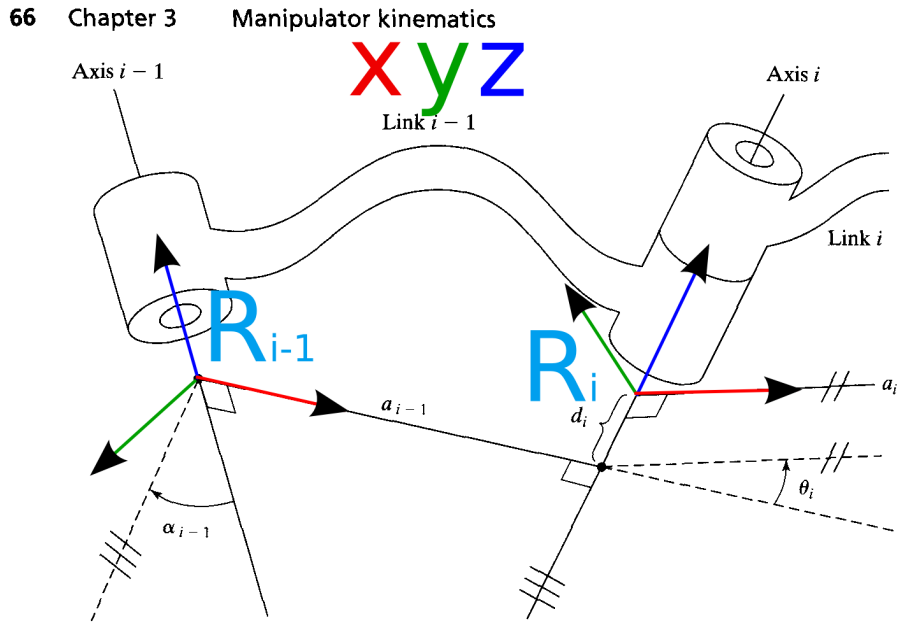


Figure 1.8: DH parameters (Denavit and Hartenberg)

As we wanted to have a complete control of the structure, we decided to use the factorized form in our implementation.

### 1.2.2 Forward and inverse kinematics

#### Forward kinematics

Computing forward kinematics means that given all the joint angles $\theta_0, \ldots \theta_{n-1}$ and the first referential $R_0$, we want to calculate the final position and orientation the frame $R_n$.

The solution is to multiply the matrices : $R_n = R_0 \times {}^0_1T \ldots \times {}^{n-1}_nT$ (see Figure 1.9 for the definition of T). We also only have to provide the $\theta_i$ at computation-time, all other parameters are already predefined if the configuration does not change, which is the case most of the time.

#### Inverse kinematics

Computing inverse kinematics means typically that given a desired final position $R'_n$ and the first referential $R_0$, we want to calculate the joint angles $\theta_0, \ldots \theta_{n-1}$ that minimizes the error between the actual final position $R_n$ and the desired final position $R'_n$.

There are as well several methods to solve these problems, but most of them works iteratively, because it generalizes well to bigger problems.

$$
{}^{i-1}_{i}T = \begin{pmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i s\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1}d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i c\alpha_{i-1} & c\alpha_{i-1} & cos\alpha_{i-1}d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

$$
= Screw_X(a_{i-1}, \alpha_{i-1}) \times Screw_Z(d_i, \theta_i)
$$

$$
= \begin{pmatrix} 1 & 0 & 0 & a_{i-1} \\ 0 & c\alpha_{i-1} & -s\alpha_{i-1} & 0 \\ 0 & s\alpha_{i-1} & c\alpha_{i-1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} c\theta_i & -s\theta_i & 0 & 0 \\ s\theta_i & c\theta_i & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

Figure 1.9: General form of the matrix obtained by DH parameters. $sX = sin(X)$ and $cX = cos(X)$

The basic algorithm used to solve IK for position is the Newton-Raphson algorithm [11], and proceeds as in algorithm 1.2.2.

**Algorithm 1** Computes the angles $\theta_0, \dots \theta_{n-1}$ to reach the final position.

$R_0 \leftarrow$ first referential
$\theta_0, \dots \theta_{n-1} \leftarrow$ initial angles
$R'_n \leftarrow$ desired final position
$R_n \leftarrow$ result from forward kinematics
**while** $|R'_n - R_n| < \epsilon$ **do**
  $J_F \leftarrow$ Jacobian of $R_n(\theta_0, \dots \theta_{n-1})$ at position $\theta_0, \dots \theta_{n-1}$ .
  $J_F 1 \leftarrow J_F^{-1}$ // Inverse of the jacobian
  $T \leftarrow R'_n \times R_n^{-1}$ // The twist between the two referentials
  $\Delta \leftarrow J_F 1 \times T$ // The difference in the angles which locally makes $R_n$ going towards $R'_n$
  $\theta_0, \dots \theta_{n-1} \leftarrow \theta_0, \dots \theta_{n-1} + \max \Delta, \eta$ // $\eta$ is a small parameter because the displacement is not linear else.
**end while**
**return** $\theta_0, \dots \theta_{n-1}$

As we could also see this problem like a minimization of the error function in multi-dimensional space, one could use also such methods to compute the IK. However, we did not have time to explore them, so we just mention some points about that in appendix A.

# Chapter 2

# Our approach

After this theoretical background, time has come to show that all these approaches can be used together on the roombots.

## 2.1 Designing the roombot kinematic model

The roombot module is presented on Figure 2.1. As said before, it has 3 rotation axis, and 10 connectors. As each connector of a roombot module can be connected to any connector of other roombot modules, we would need more than a simple kinematic chain. But we will first concentrate on the end-to-end main kinematic chain of a roombot module.

### 2.1.1 Model for submodules.

A roombot module is made of 4 submodules, each one having, to simplify but not that much, the shape of the half of an intersection between a cube and a sphere. So the main shape is in fact the cube.

We can then specify at least 5 different referentials for each submodule (Fig. 2.2). One is the "middle referential", named $M_i$ with $i \in 0, 1, 2, 3$, it is placed exactly at the center of the imaginary including cube. and its orientation depends on the following connector referential convention. The connectors' referentials are defined by $C_iX, C_iY, C_iZ$. Fig. 2.2 present those connectors for the first submodule of the chain.

The connector referential convention used is the following:

- All Z axis of the connectors referentials are pointing outwards the submodule.
- The connector referential name comes from the axis of $M_i$ which points to them
- The connector referential orientation is so that the X axis of each referential is pointing to the next connector in an anticlockwise manner when

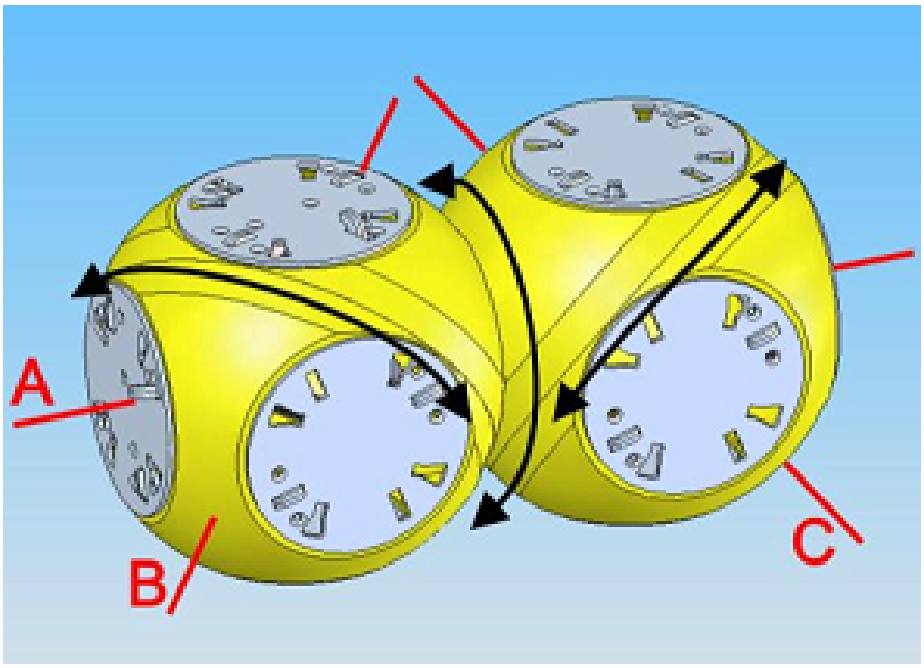


Figure 2.1: The roombot module.

looking at the three connectors at the same time. The Y axis of each connector referential is pointing to the previous connector.

Last but not least, there is a "base referential" for each one of the four submodules, named $B_i$. As the 4 submodules are connected by 3 motors, the $Z$ axis of $B_1$, $B_2$ and $B_3$ are the respective rotation axis of these motors. The sequence of referentials $B_0, B_1, B_2, B_3$ are defining the main kinematic chain of the roombot, on which the DH parameters and convention are going to be used.

### 2.1.2 Model for the submodules chain.

Once we have this convention for submodules, we can use the convention mentionned on page 11 to orientate the base referentials from one module to the other.

First, if there is no rotation, the first two referentials $B_0$ and $B_1$ are the same, except that the first one is still "virtually linked" to the first submodule and the second one to the second submodule (Figure 2.3).

The X axis of $B_0$ is oriented so that it is perpendicular to this rotation axis and to the next one (see Figure 1.8 page 11), and the Z axis of both referentials are pointed at the same direction as their respective rotation axis.

We can do the same work for the connection between the second submodule and the third submodule (Figure 2.4). One can remark that with this model,

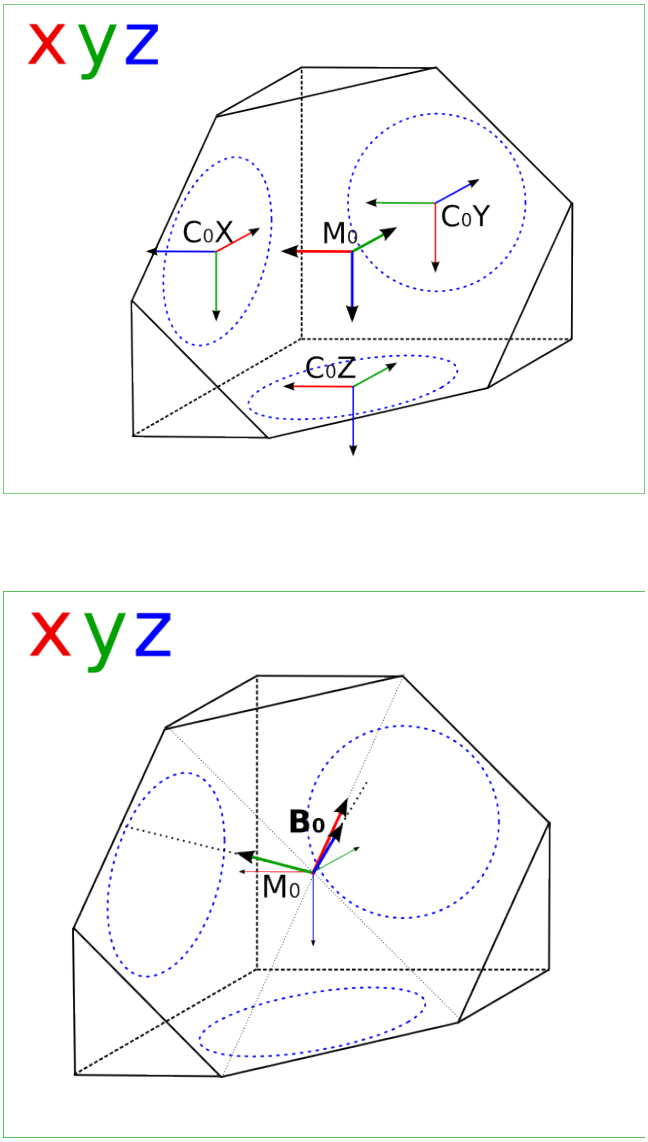


Figure 2.2: The roombot submodule and its referentials.

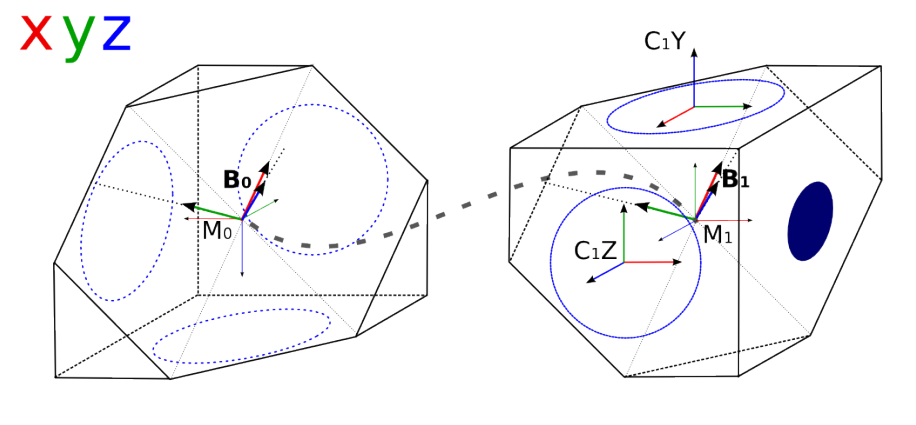


Figure 2.3: Connection from the first submodule to the second.

there is only one translation between the four base referentials, which is between $B_1$ and $B_2$.

As you can see, $C_1X$ and $C_2X$ do not exist, because there is a rotation joint at the same point.

Last we design the transformations from the third referential to the fourth and last one (Figure 2.5).

### 2.1.3 Connecting roombot modules together

One last convention that we had to develop was to describe the four types of connections when we have two roombot modules together. We designed four keywords: Parallel, Perpendicular, Shear Z and Shear S. They all correspond to some realities described in figure 2.6.

The existence of such convention make it very easy to not only name a configuration, just by looking at the closest point between the two "circles", but also to build a particular configuration given the keyword and the connectors. As all these situations are symmetric for both roombots, the keyword is also unique.

To name the configuration, this visual algorithm can be followed:

1. Look at the two circles of the servos in diagonal.
2. If the two circles are parallel, the connection type is parallel
3. If the two circles are "touching" each other, or in other words, their respective plans are orthogonal, the connection type is perpendicular
4. Else, put in front of your eyes the two points, one of each circle, which define the distance between the two circles.
5. If following the curve that you see, you see a 'Z', then it is Shear Z.

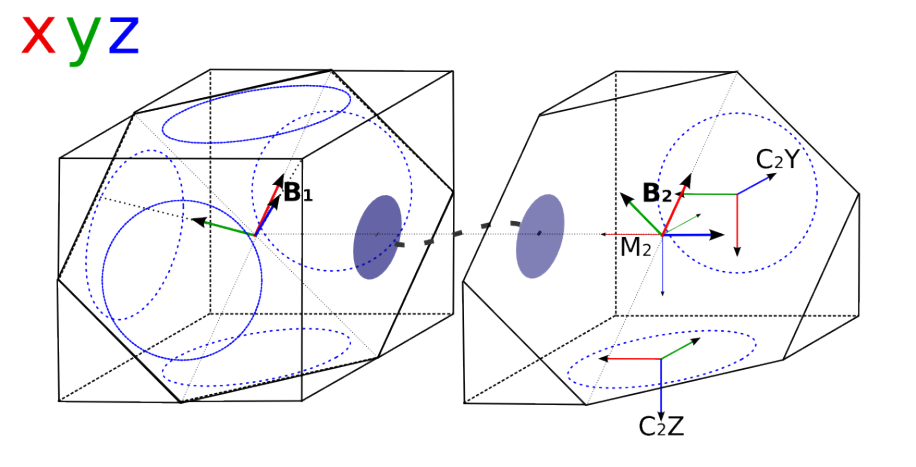


Figure 2.4: Connection from the second submodule to the third.

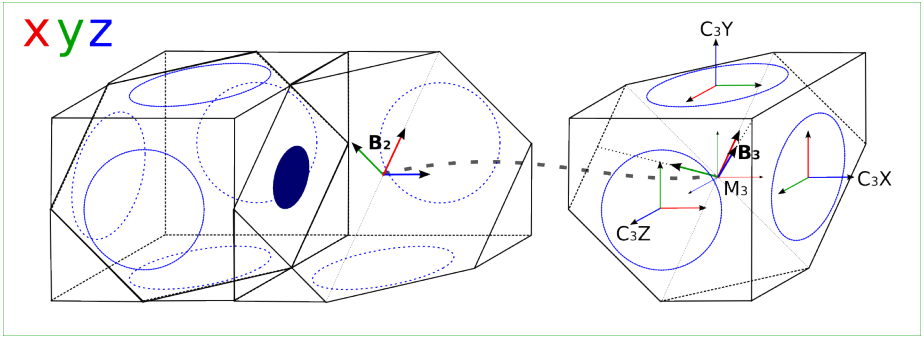


Figure 2.5: Connection from the third submodule to the fourth (last).

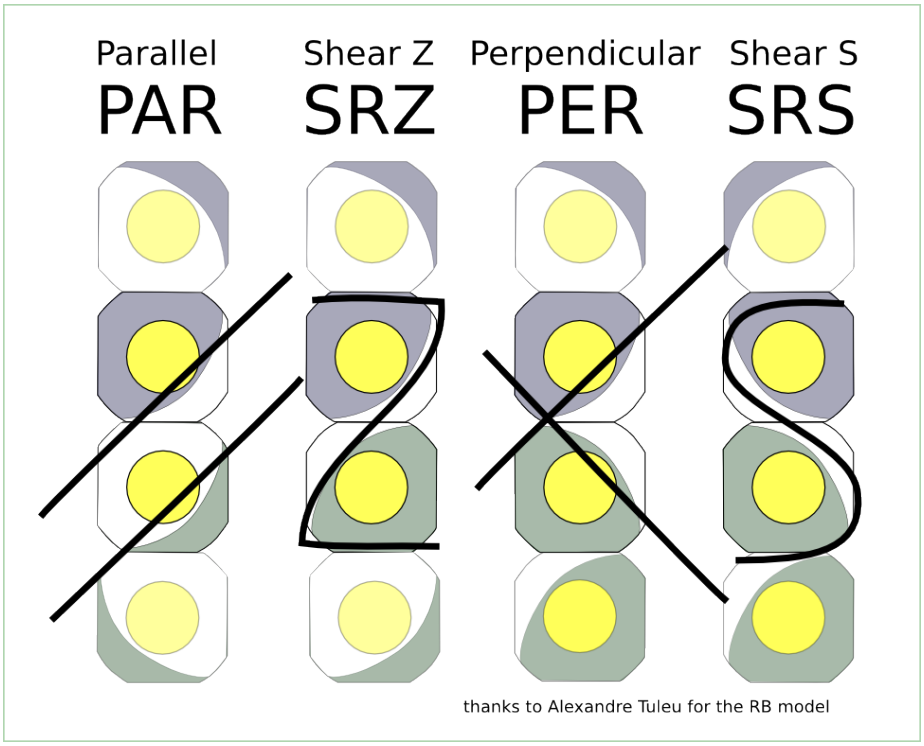


Figure 2.6: The four connection types of the roombot modules

6. Else you see a 'S' and it is Shear S.

# Chapter 3

# Implementation

We implemented all our models in C++, because it is object-oriented, compilable in assembly and thus powerful. We describe in this chapter how we dealt with the computation of kinematics, with the graph of connectors and associated transformation matrices, and also with the message class used by roombot controllers to talk to each other.

## 3.1 Forward and inverse kinematics

To implement forward and inverse kinematics, we first needed a way to represent transformations.

### 3.1.1 Implementation of referentials

At the beginning and for simplicity reasons, we design our own transformation class based on a representation containing a quaternion [24] plus a vector.

The advantages of quaternions are that the corresponding rotation is aways normalized, there is no gimbal lock effect[26] because of the continuous representation, and the multiplication cost less.

The counter-part is that we have to compute the corresponding rotation matrix[25] when we have to multiply it by a vector.

#### KDL and boost

KDL [19] also provides Frames, Rotations and Vectors, which are provided by boost. We later managed to integrate KDL to our code, which was difficult due to a lack of documentation.

We mainly used KDL capacities to solve the inverse kinematics. Forward kinematics were solved by both KDL and our quaternion representation.

## 3.2 Implementing the kinematic model of the roombot

Thanks to the Figure 3.1, we are able to see what the kinematic chain is from one connector to the other, like $C_0X$ to $C_3Y$, or from any known referential in the roombot to any other inside the same roombot.

To efficiently recover the succession of referentials, we designed two 2D arrays.

1. `Successor[REF1][REF2]` is the neighbour referential to REF1 that enables to get closer to REF2 in the graph. It is defined everywhere, except if REF1 = REF2
2. `Between[REF1][REF2]` is the transformation between two adjacent referentials in the graph, else it is null.

For example, the initialization might look like this:

```
static REFERENTIAL      Successor[REF_COUNT][REF_COUNT];
static Transformation* Between[REF_COUNT][REF_COUNT];

Between[C0X][M0] = CiX_Mi;
Between[M2][C2Z] = Mi_CiZ;
Between[M0][B0]  = M0_B0;
Between[B0][B1]  = NULL;
...
```

We remark that the transformation between B0 and B1 for example cannot be computed statically because it depends on angles provided at the run-time.

## 3.3 Useful C++ design pattern

We got some problems due to the C++ language itself. Contrary to java, if we want to manipulate abstract types, we have to use pointers with all the pain that it represents : Calling new, delete, forgetting to destroy objects, etc. But whatever we might do, abstract types cannot be set without pointers. And memory management is really not a good solution, not scalable, hardly extensible.

So we present here a model of a design pattern which provides the user all the power of manipulating abstract classes, but without the pointers. Here are the specifications.

- One abstract class m models an interface, like a “model”, a “type of”, or a “sort of”. Imagine, for example, a model of message, which already contains the ID of the sender and the receiver, but still needs a type and serialization function.

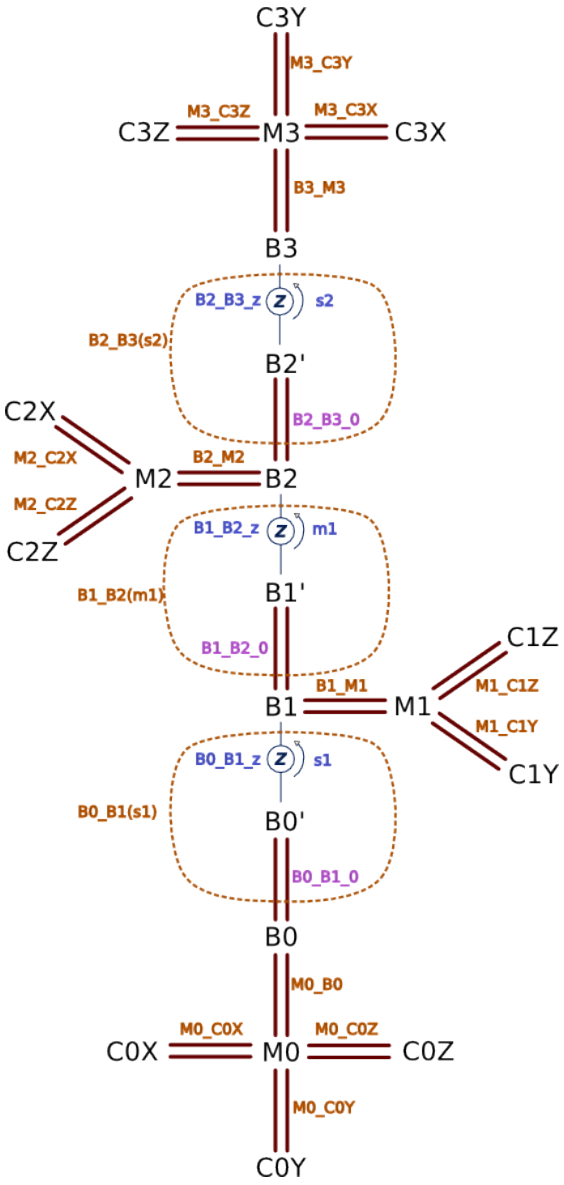


Figure 3.1: The graph of the transformations between referentials

- One class M containing a pointer to this abstract class m, and having the same public methods. It is a also "`friend`" of this abstract class.
- Each child class c has to register a `static m* create()` function of itself into the abstract class, with its corresponding type given by `int type()`
- Each child class c has a default `C cast(M)` function which takes an object of class M and if possible casts or converts it into its own type.
- Each child class c has a default `m* clone()` function which clones itself, and is called by the copy constructor of class M, so that two different objects of class M would never share the same internal class m.

Of course, this design pattern has its own pros and cons.
Pro:

- To design a new child class, one has to implement the pure virtual methods, so something cannot be forgotten nor hidden at run time.
- All non-abstract children can be created on the stack, without "new" pointers, so they can automatically be deleted by the compiler at the end of their scope.
- If there is at least one registered child, the class M can create a default object by looking at the registration array. If there are some good setters and getters, this can be a good solution to create an object without knowing what the implementation would even be.

Cons:

- The registration: If a child class does not register itself, they might be blocking functions to prevent the use of this child class.
- Depending on the compiler, the copy constructor might be used too much, so this might slow down the whole program. But this is usually optimized and not that harmful.

### 3.3.1 Example in communication : Class Message

We exactly used this design pattern in the Message class, as we needed a way such that the Message class does not depend on anything, the customized messages easily implement the missing methods, and receiving/sending messages is done in few lines of code.

```
class Message {
protected:
  InternalMessage* m;
public:
  /// ...
  Message& operator=(const Message &message) {
```

```
    if(m) delete m;
    m = message.m->clone();
    return *this;
  }

  unsigned char type() {
        if(m) return m->type();
        return -1;
  }

  int get_id_sender() {
        if(m) return m->get_id_sender();
        return -1;
  }
};

class InternalMessage {
private:
  int id_send;

public:
  int get_id_sender() {
    return id_send;
  }

  virtual Serializer&        serialize(Serializer& s) = 0;
};
```

And this is how we would implement a message which sends 42 to everyone by using the command `MessageInteger::send(my_id, 42);`

```
class MessageInteger: public InternalMessage {
private:
  int my_integer;

  MessagePositions(int id_send, int my_integer);
public:
  static InternalMessage* create() {
    return new MessageInteger();
  }
  static void send(int id_send, int my_integer);

  InternalMessage* clone();
  int type() { return 1; }

  virtual void serialize(Serializer& s) {
    s<<my_integer;
```

```
    }
};
```

More documentation is available in the source code of the project, with a complete example from scratch, and a lot of message examples developed for the roombots and for Webots.

One good point about the Message class is that we can specify how we emit and receive the messages, so we were able to inject one emitter/receiver for Webots and one for the Physic plugin, as they have two different interfaces.

### 3.3.2 Working with KDL

We had to get information about KDL[19] in order to integrate it to my existing kinematic structure.

KDL does not work with trees for the moment, but with chains. So for a given configuration, we had to build the corresponding chain.

Fortunately, my model authorizes inverted chains with the same angles as if the chain was not inverted, this is useful when the roombot moves by reaching a position locks one part, unlocks the other and then continue but in a reverse configuration. See the chair problem (page 41) for an example of such moves.

# Chapter 4

# Working with Webots

It is now time to present how the integration of these algorithms has been made in Webots [21], which is a real-time simulator of robots in a 3D graphical environment.

Most of the experiment have been done on a **meta-module**, which is composed of two roombot modules linked together on connectors $C_3X$ and $C_0X$ (see figure 4.1).

## 4.1 The roombot controllers

We consider two types of controllers that we used in Webots. The pure roombot controller, which the real roombots should have in the future, and the supervisor controller, which acts like a centralized coordination.

For the moment all the forward and inverse kinematic code has been put in the supervisor controller, because it was experimental. But a lot of the functionalities provided by the supervisor should finally appear in the roombots as soon as they would become autonomous.

The plan used to do all experiments was the following:

- The supervisor computes everything and sends orders to the roombots
- The roombots execute the orders and send back notifications.

This plan does not exclude the fact to have several supervisors, each one controlling different sets of modules, as you can see in the chair problem (page 41).

### 4.1.1 The reactive roombot controller

This controller is the one developed during this project. Its purpose is only to react to orders in a clever way.

So first, let us have a look at the list of tasks that our roombot controller should be able to do:

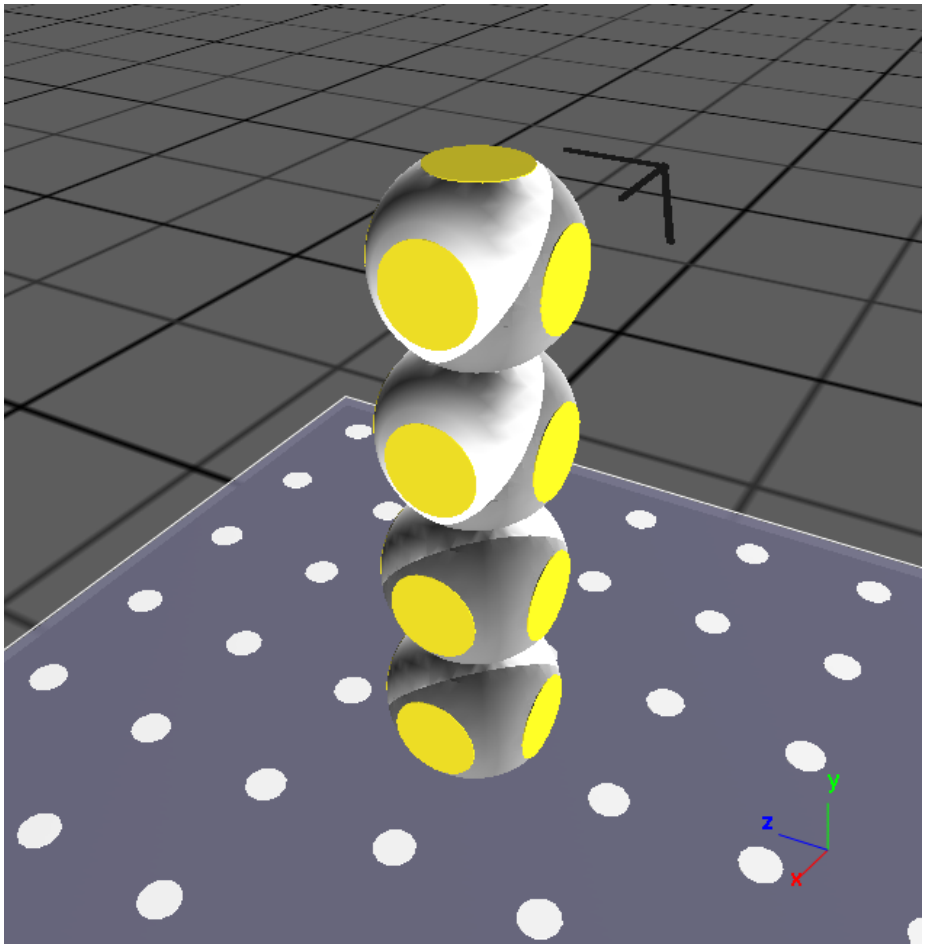


Figure 4.1: The roombot meta-module, composed of two roombot modules.

- Gives the servos positions to anyone who asks for it.
- Set up the positions given by anyone.
- Sends a message to tell that the move is finished.
- Locks or unlocks a connector.

One sees that these tasks arise some problems.

#### Problems while locking and unlocking connectors

First, what if we are asked to lock a connector but there is nothing to lock on? Or even worse, if we have to unlock one side of a roombot chain without being sure to be connected on the other side?

For this reactive controller, we decided not to cope with this problem, so it was the role of the supervisor to be sure of the orders it would send.

#### Problems while moving

The other problem is the set-up of servos positions, i.e. to move the roombot. The roombot can easily get stuck if it encounters an obstacle, and without internal expensive collision detection the supervisor cannot anticipate that.

As the roombot have this fantastic ability to make continuous 360° rotations around any servos, we decided to implement an bounding-trial algorithm. If one of the servos detects three times a change towards a bad direction (marked as red stars on the graph of Figure 4.2) or no movement at all without having reached the goal, it changes the goal modulo $2\pi$ to overcome the problem.

If it still does not work while changing once the goal, the roombot controller will change it up to three times, moving first forward, then backward, then forward.

If it still does not work, the roombot sends back a message with its current positions telling that it got stuck.

If it manages to reach the position, it sends back a message telling that it succeed.

### 4.1.2 The supervisor controller

The supervisor controller went through a lot of modifications over time, resulting in the current controller which sums up all possibilities that the supervisor is able to do.

Thus, the current supervisor has the following behaviour:

- It receives in argument the ID of two roombots making a meta-module.
- It receives in argument the way the roombots are assembled.
- It receives in argument a file containing instructions to execute.

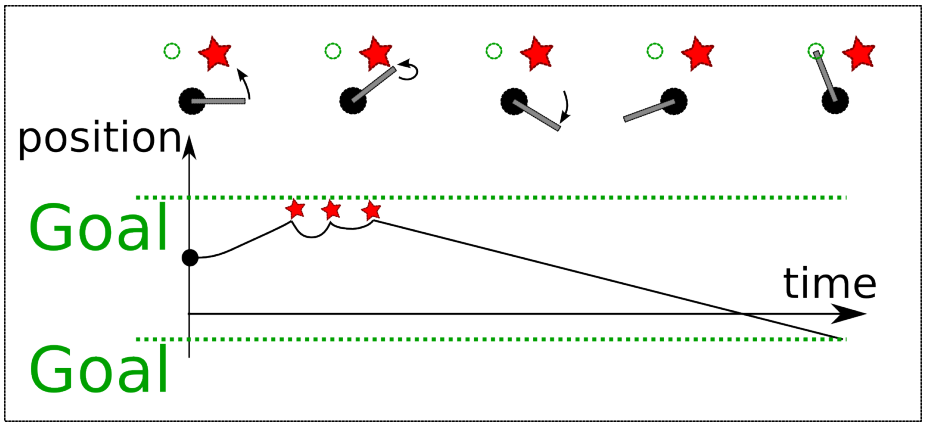


Figure 4.2: Movement of a roombot servo during time if it gets stuck while reaching a goal.

- It sends the orders following the instruction file.
- It may stops, following instructions, to wait for more instructions on the command line.

One sees that the most important step here is the set of instructions that the supervisor can parse and execute.

#### Supervisor capacities

The supervisor can first ask the meta-module to move, either with forward kinematics, giving explicitly the angles, or with inverse kinematics, giving the position to reach. It can ask them to detach or reattach, as we discussed in the previous section. For debugging purposes, it can draw a cube with axis at the extremes of the meta-module (if the kine-controller-plugin is activated), on-the-fly read commands on the command line, and it can wait some time.

One sees that a lot more could be done in this field, giving more high-level and/or implicit commands such as "assume that you are moving on a grid and move to this absolute position". For future projects, it is easily extensible or reimplementable.

See Appendix B for more details about the implementation of the commands.

# Chapter 5

# Experimentation and results

Now it is time to show up some of our main results that we obtained thanks to our work.

## 5.1 Moving and locking messages

First, we tested our class Message with a small example of reconfiguration, see Figure 5.1 for the screenshots of the video.

Some remarks about this video. In the initial position, the meta-module is somehow stuck, because the two roombots are connected from $C_2Y$ (first) to $C_0X$(second), so it looses one degree of freedom over its six. The purpose of the reconfiguration is to connect $C_3X$ of the first roombot module to $C_0X$ of the second.

At some point, we added a small movement of the second roombot module to allow the middle servo of the first roombot module to turn, because there would be some collisions else in reality.

## 5.2 Testing forward and inverse kinematics

### 5.2.1 Solving forward kinematics

On of our first results and proof that it was working was what we called the "consciousness" of the robot's position. The supervisor computes the position that the roombot meta-module should reach given some random angles, set-up a red-green-blue referential at this point, and then sends a message to set up the angles like it decided.

The coincidence of the last referential with the axis system showed that it was perfectly working in this case (see Figure 5.2)

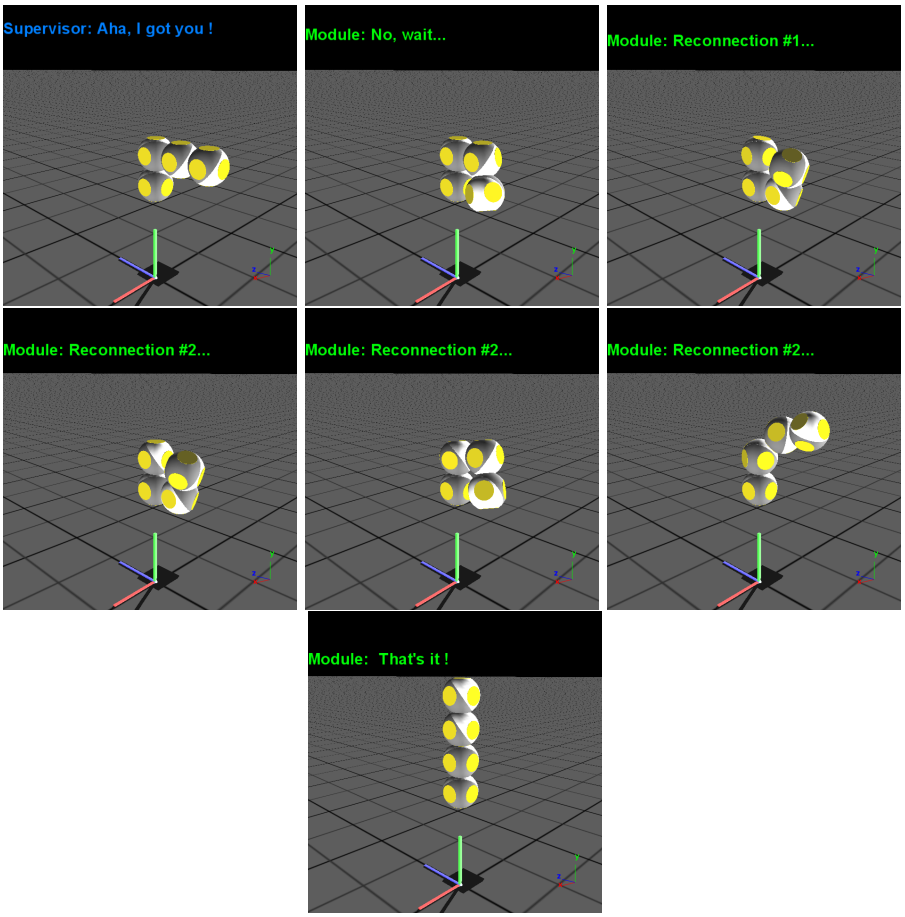


Figure 5.1: The test of move and lock messages.

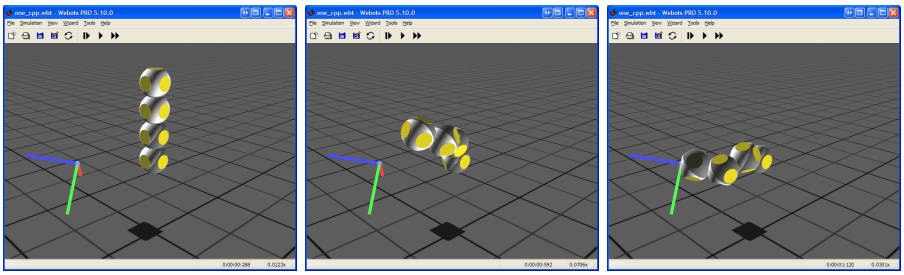

Figure 5.2: Computing internal and external forward kinematics.

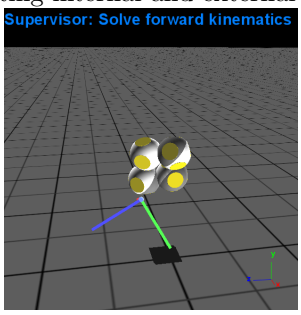


Figure 5.3: Blocked while solving forward kinematics.

However, for some angles, we got some internal collision (see Figure 5.3). It meant that it had the consciousness of the kinematic chain, but not of the shape of the roombots.

It looks hard to integrate an internal collision detector into the controller, but it could be treated in a possible future work.

### 5.2.2 Solving inverse kinematics

Once the kinematic model was working, and the integration with KDL done, we were able to specify absolute referentials to reach and the meta-module would align the last connector of the kinematic chain $C_3X$ to this referential. See Figure 5.4 for an example.

We encountered some problems with the KDL inverse kinematic solver, because the default algorithm was first not even able to get the best approximation of the inverse kinematics when there was no solution, and worse, it could find solution very different from the expected approximate solution (see Figure 5.5), which is sometimes very bad, especially when we take collision into account.

Of course, there has been some problems because of singularities, which are quite common in the domain of inverse kinematics [28].

The best method was to get some knowledge about how inverse kinematics was working [23, 6, 12], in order to manipulate the internal coefficients to make resolution smoother. Section 5.3.1 presents some methods used to improve the

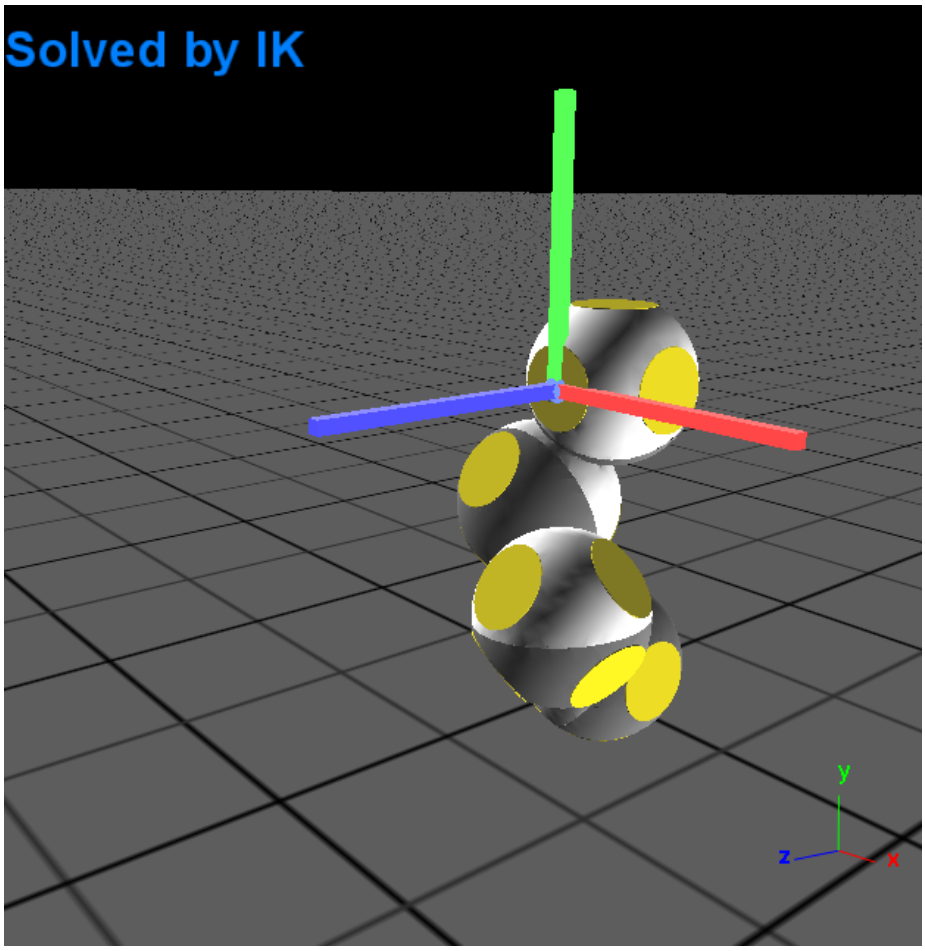


Figure 5.4: The roombot reaches an arbitrary referential.

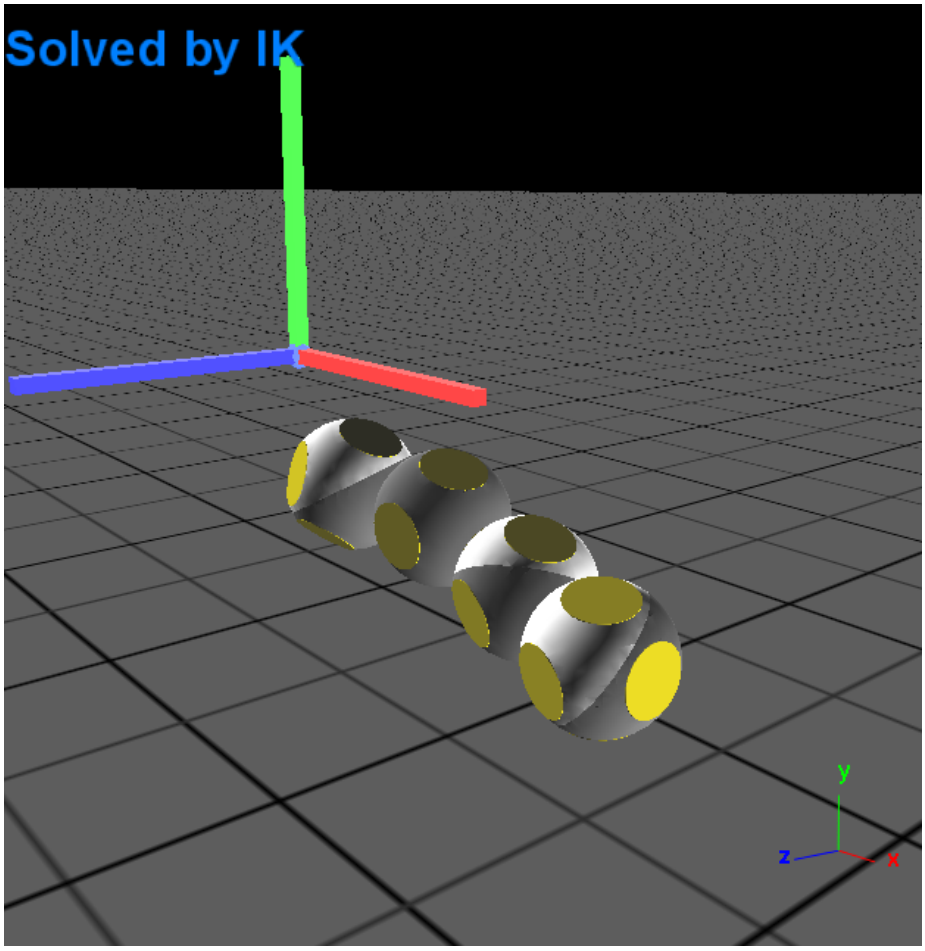


Figure 5.5: KDL default IK algorithm was not able to direct the roombot toward the unreachable goal.

algorithm.

## 5.3 Direct application of IK

Once we have the inverse kinematic algorithm, we were able to do most of the proof of concepts, like moving on the diagonals of a grid or compute the reachable space.

### 5.3.1 Reachable space and plan

First, we reached the most far points from the base of the roombot in all directions, locally oriented to Z. This result is displayed in figure 5.6.

Second, we reached the referentials on a discrete plan, as if the roombot meta-module wanted to move on it. For each position, we displayed a blue box if the position has been reached, with the corresponding X axis as a red stick. We tested four possible rotations per position, so we could have one to four red sticks on each blue box.

We modified the existing inverse kinematics algorithm in order to try to let it solve everything "more smoothly", by different ways, and we compared it to the solutions of the original algorithm (Figure 5.7) The point was that some solutions were too far away from the original position, so while trying to reach positions in a plan, the solution computed by KDL would collide the plan.

One big artifact of existing KDL algorithm, was the fact that the angles could diverge, and it was very common to get values such as 1204.18 radians, which was very bad and resulted to some solutions like in figure 5.5. So we tested following modifications, sometimes independently and other times merging them.

- Recomputing all angles modulo $2\pi$ and between $-\pi$ and $\pi$ at each iteration step.
- Reducing the greatest angle step.
- Applying only the biggest angle step (direction by direction)
- Applying different weights on the angles, decreasing, increasing.
- Discretizing difference angles to multiples of $\pi/6$ ($\Rightarrow$ rounding angle steps.)
- If the max is greater than a certain threshold (0.05), normalizing the angle step so that the max is less than a given value. (Figure 5.8)
- Increasing the tolerance of the final position from 0.01 to 0.05 (Figure 5.9)

We displayed the result of such changes on figures 5.8 and 5.9. In the final algorithm, we just kept the last two modifications, let us explain why.

One can remark that increasing the tolerance of the final position did not really increase the number of inverse kinematics solved (56 to 57), but it increased

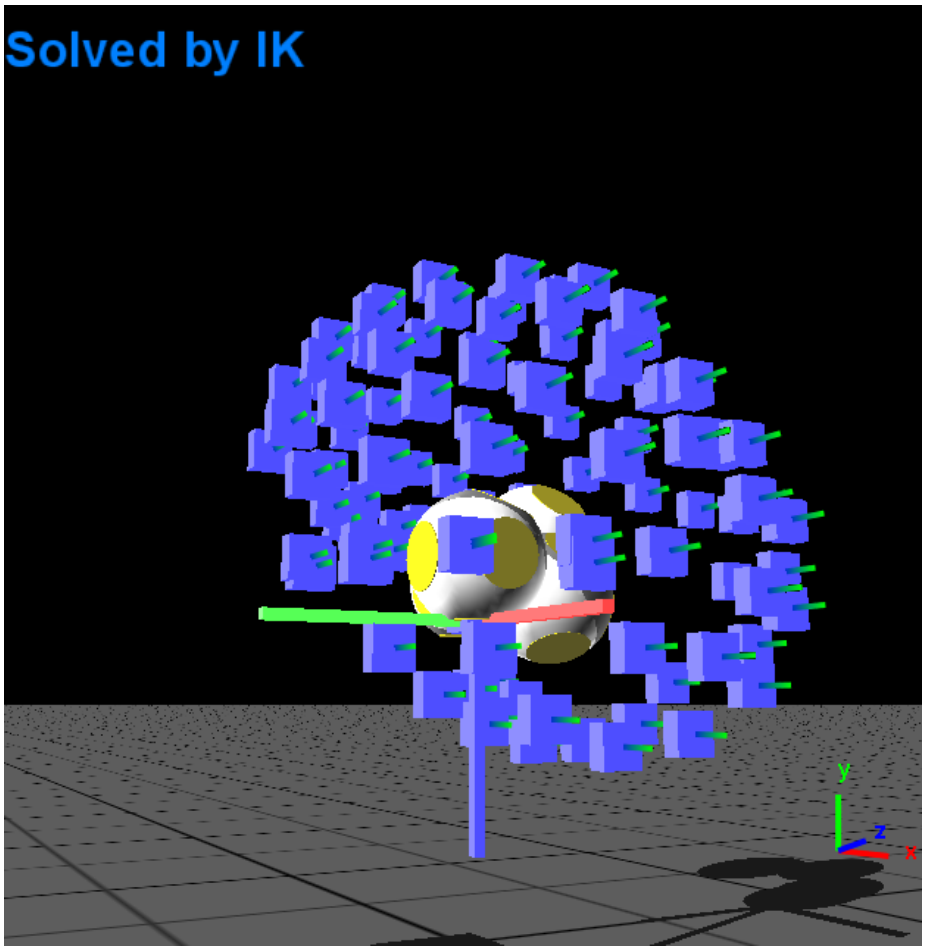


Figure 5.6: The reachable space solved by KDl, without checking self-intersections

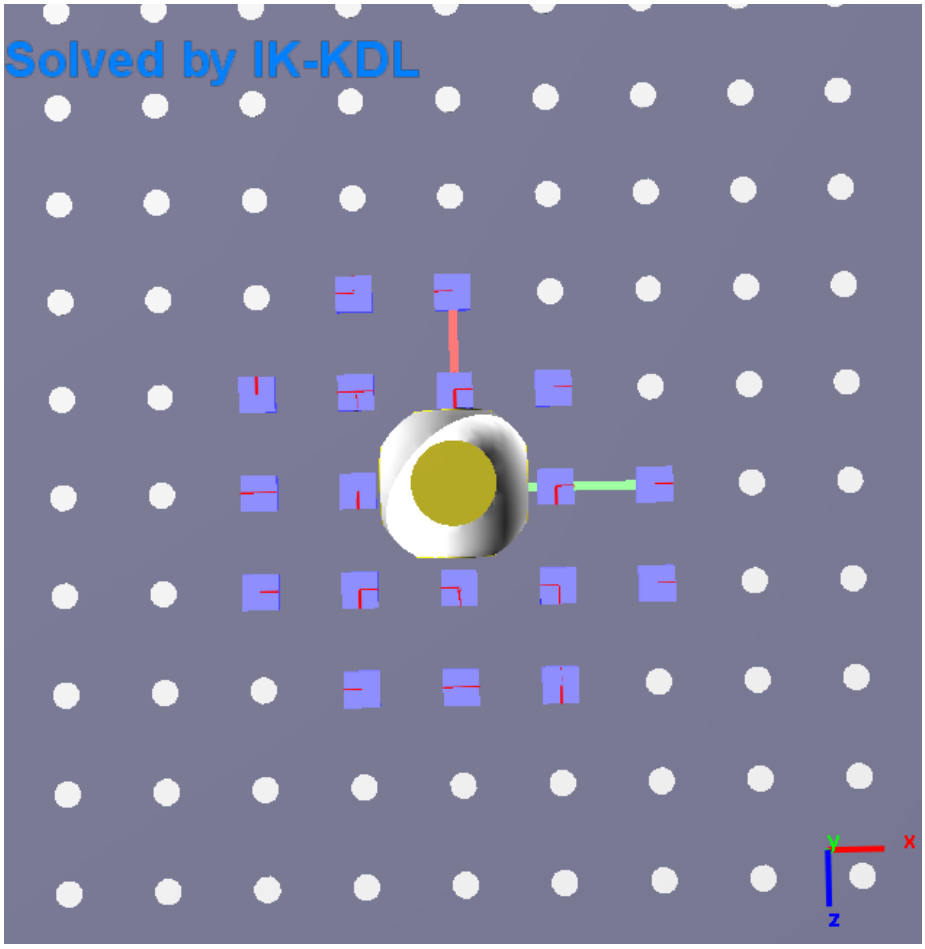


Figure 5.7: Reachable plan solved by KDL. 32 reached over 56 solved.

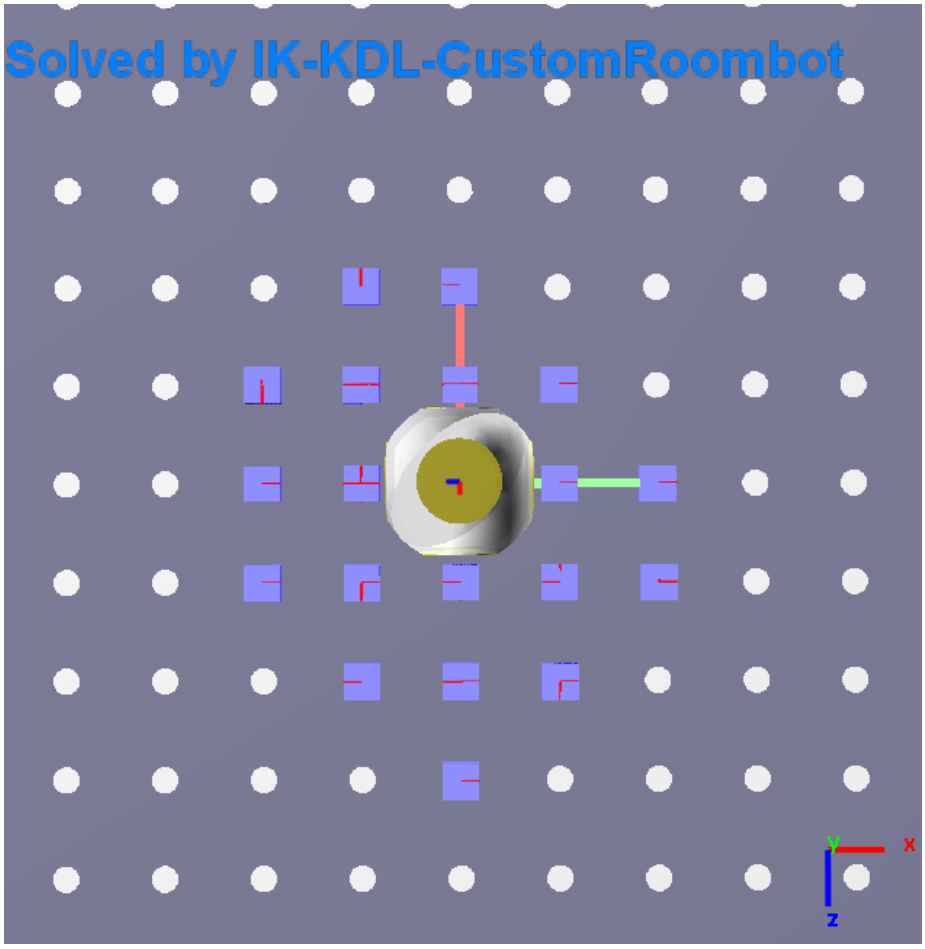


Figure 5.8: Reachable plan solved by our model. 37 reached over 56 solved.

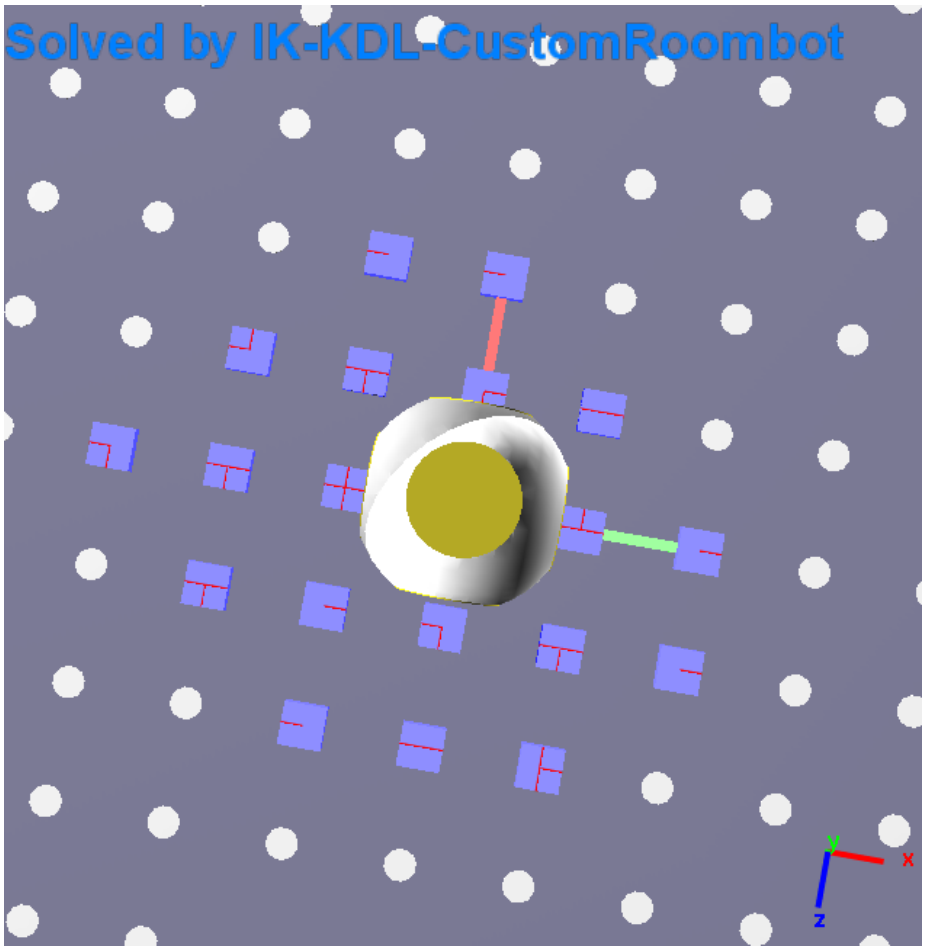


Figure 5.9: Reachable plan solved by our model with higher tolerance. 40 reached over 57 solved.

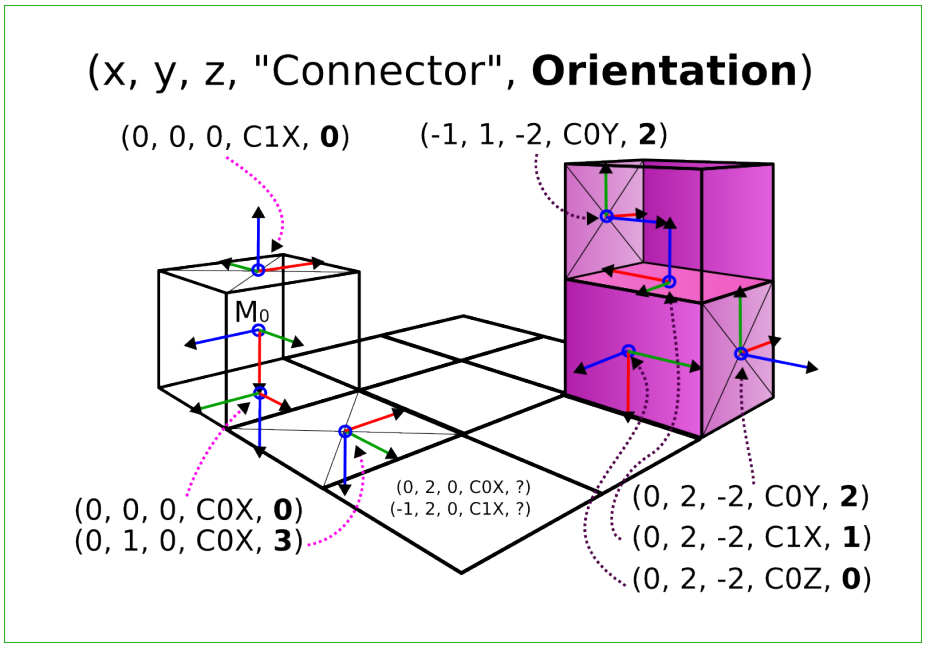


Figure 5.10: Description of all the referentials located on the faces of a grid.

the number of inverse kinematic reached in the simulation (32 to 40). However this is not an amelioration of the initial algorithm, it is only a parameter.

One can also remark that the main difference between our changes and the original algorithm is not in the number of positions reached – between 30 and 40 for all modifications mentioned above, but rather the set of positions. There are some positions that can be reached in some models and not in others.

So it looks like we should solve the IK with several different models and algorithms to be sure to have better chances to get at least one solution. This could be a future improvement.

### Moving inside a 3D grid

The figure 5.10 presents how we did to specify goals when we were moving on a grid.

So if the $M_0$ referential of figure 5.10 coincides with the $M_0$ referential of the first roombot module of a meta-module, the command

```
reachfw   0 1 0   1 0 0   3
```

will make the meta-module place the connector $C_3X$ of the second roombot module to coincide with the referential described by (0, 1, 0, C0X, 3) on figure 5.10. See Appendix B page 48 for the listing of all the commands.

We designed a static method GridPosition::getTransformationTo(0, 1, 0, 1, 0, 0, 3) to compute the corresponding transformations. It could also be

called by GridPosition::getTransformationTo(0, 1, 0, GridPosition::DIR_P_X, 3) because this is the connector referential computed from M0 by moving toward X (Plus X).

The purpose of a grid is to quickly be able to describe a position to reach with the minimum of parameters, knowing that the whole world is located on that grid.

### 5.3.2 The acrobat roombot

We also tested our kinematic chain models with three roombots, and we artificially increased the torque for the simulation, because the roombots are not able else to handle that many modules.

So the screenshots of the movie are displayed on figure 5.11, and let us give some comments on that.

One sees that once there is one roombot on one side and two roombots on the other side, the easiest way to reconnect them together is to give them back the angles when it first solved the inverse kinematics. This can be generalized if we ever wanted to make roombots help each other to move.

In the real video, there are two times in which one chain turns in the other direction, thanks to our controller. Else, it would have get stuck, because the closest goal is not always reachable (see section 4.1.1).

## 5.4 The chair problem

Our main goal (see section 1.1.5) was to make a proof of concept of the locomotion of a meta-module over a passive roombot structure.

Now that we have implemented all the required concepts, we combined them to make the video of the chair, which is quite complex even if it is short.

So let us recall the video : Two meta-modules start on the ground, on a grid of connectors. Both move toward the back of the stool (see figure 5.12) to form the back of the chair (see figure 5.13)

To achieve this result, we set up two command files containing line by lines the commands to achieve. A lot of the previously explained concepts have been re-used :

- Move on a grid: We used some commands to tell a position to the meta-module to solve the inverse kinematics.
- Move with angles: Sometimes and to optimize the movement, we gave instructions to move to absolute angles
- Move forward or backwards: Sometimes we have to compute the IK from $C_0X$ of the first to $C_3X$ of the second, sometimes it is reversed, and it does not cause any problem.
- Locking/unlocking: Once over two, we connected $C_0X$ of the first and disconnected $C_3X$ of the second

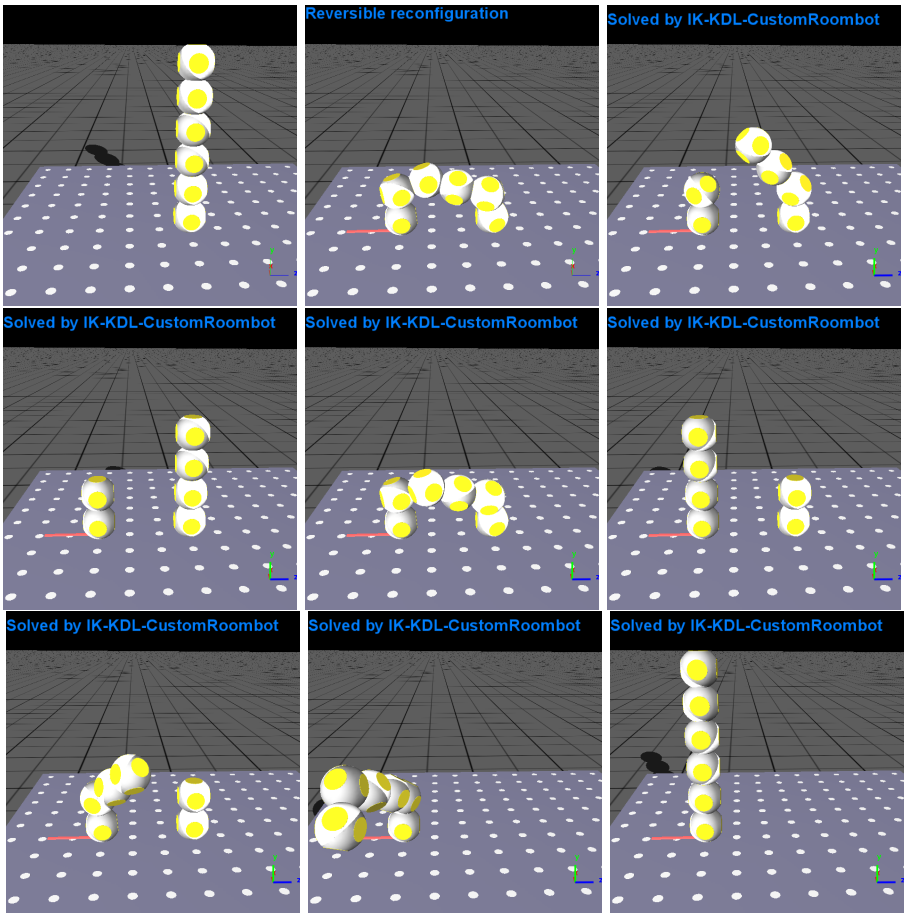


Figure 5.11: The acrobats, reconfiguring and exchanging modules.

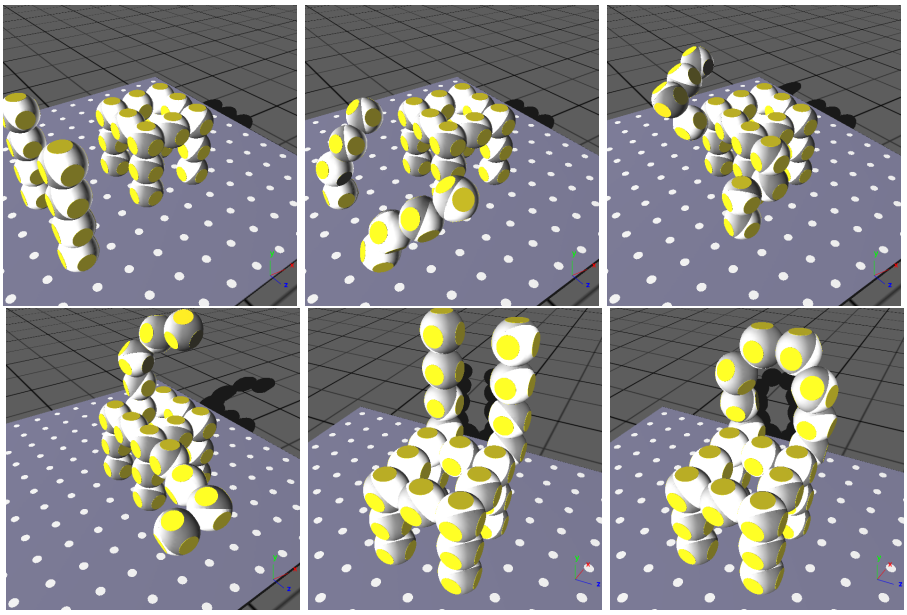

Figure 5.12: Locomotion of the meta-modules to make the back of the chair.

- Reactive roombot controller: Sometimes it changes direction because it cannot reach its goals in the direction it is moving to.

The results that we can see are as follow:

- The roombot meta-module is able to go through concave (beginning of the movie) as well as convex angles (when it reach a position in the back of the chair)
- If the position to reach is "easy" in a sense that one could turn the roombots by 90° rotations to reach the position, then it is often more adapted. In the future, a database with such precomputed movement would be good.
- Finally, it would be quite easy to give a goal position to a roombot meta-module, and knowing its environment, it could choose a best path.

Now that the infrastructure is working well, we think that they can be a lot of experiences that could be made on roombot meta-modules, like distributed goal-based dynamic reconfiguration.

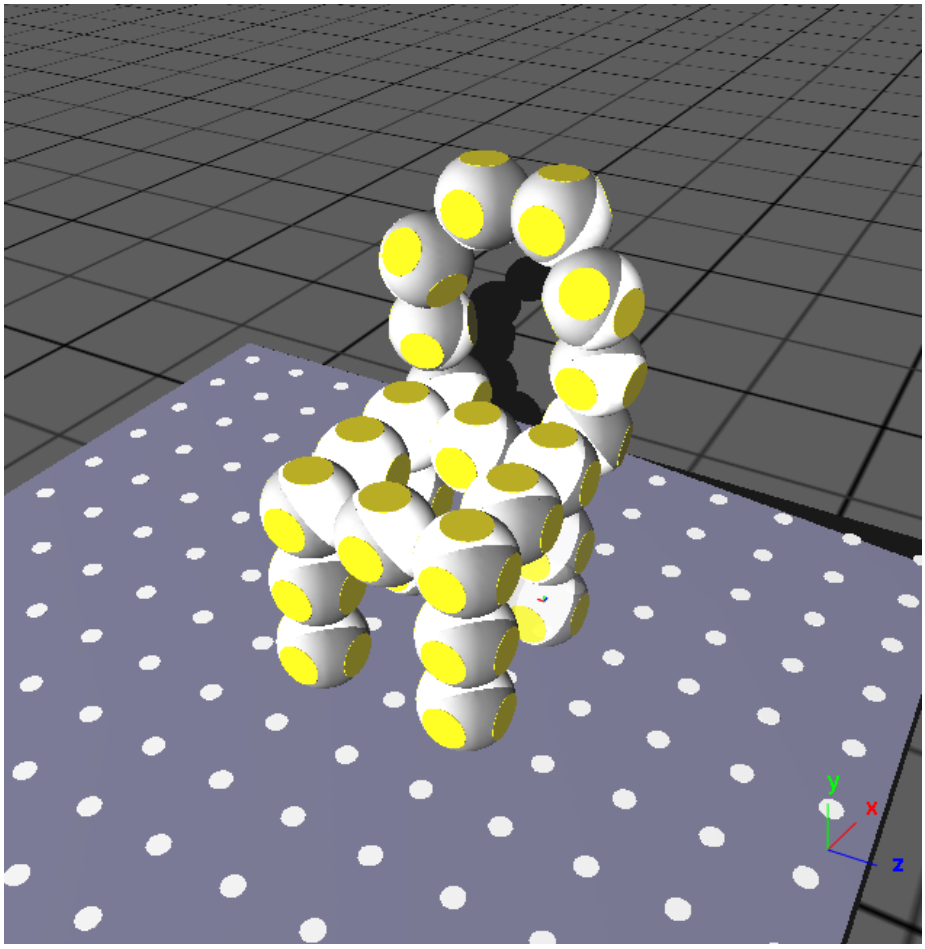


Figure 5.13: The resulting chair. The two meta-modules form the back.

# Conclusion

This project clarified the move possibilities of the roombots by establishing a kinematic model that can also be re-used for completely different future projects. It also presented some of the solutions to implement this model in an object-oriented way and to simulate it properly on Webots. Last we demonstrated the pertinence of this model by generating relevant examples such as the locomotion over the chair (see section 5.4).

We know that a perfect model does not exists, else we would not call it a model. Thus, even if the previous model has proven to be good, it still can be improved, e.g. by replacing chain structures by tree or graph structures, by distributing the kinematic computation along the roombot modules, or even by setting up autonomous agents inside each roombot, so that intelligent behaviors could emerge.

This work has been a step towards the goal of the main roombots projects, to create intelligent furniture which would be able to reconfigure by themselves.

# Appendix A

# Future improvements

During my project, I have found some problems or questions and I give here some way of thinking about them.

## A.1 Webots

What we are really missing in Webots, is an API to be able to dynamically modify the tree of a scene, with functions such as:

```
#include <webots/fictive_webots_api.hpp>

// Let's add two roombots from the webots scene.
int main() {
  Scene s;
  s.importFromWebotsFile("my_webot_scene.wbt");
  CustomRobotNode crn1("name_bubby");
  CustomRobotNode crn2("name_foo");
  crn1.setChild(crn2);
  crn1.setMaxForce(12);
  crn1.setRotation(0, 1, 0, 3.14);
  s.addNodeAtEnd(crn1);
  s.exportToWebotsFile("two_robots_more.wbt");
}
```

In our worldMaker, we are currently parsing the text file by hand and replacing the values with those that we need, which is not very extensible nor robust, because Webots does not store fields which have the default values.

In the same direction, it would be good if the existing function Supervisor::importNode would not crash the entire simulation – I mean, not unlock all the connectors –, because it is still buggy for the moment. This is the same when we sometimes want to modify some values in the tree while the simulation is running, for the simulation itself. Supervisor::deleteNode, Supervisor::modifyNode

could also be useful to have a total control on the simulation. But it might not be in the spirit of Webots.

It would also not be too difficult (something like 100 working hours) to be able to *modify the initial parameters in the tree for the next simulation.* It would mean to have a separate tree for the simulation and for the initial parameters, and as well not to restart the whole program when we are doing a reset.

## A.2 Multi-dimensional space exploring

Exploring a multi-dimensional space to find minimum could be rewritten as a problem of finding a global minimum on a space-filling curve. There has been plenty of mathematical work on them [10, 2], which could give a new approach.

Space-filling curves could also be a good straightforward way to explore a representative sample of space, if we quantize the curve, for example to compute reachable space.

# Appendix B

# Commands that the supervisor can parse

`movefw | movebw` $s1_1\ m1_1\ s2_1\ s1_2\ m1_2\ s2_2$
where $si_j$ are the desired angles divided by 2pi/3 (e.g. to have an angle $s1_2$ of $4\pi/3$ put 2 in the field) and $m1_j$ are the desired angles divided by $\pi/2$ (e.g. to have an angle $m1_1$ of $-3\pi/2$ put -3 in the field) if we use movebw, the angles are inverted.

`lock n_module CONNECTOR`
where `n_module` is either 0 or 1, depending on the module we want the connector to be locked, and CONNECTOR one of (C0X, C0Y, C0Z, C1Y, C1Z, C2Y, C2Z, C3X, C3Y, C3Z)

`unlock n_module CONNECTOR`
same commands as lock but unlocks the corresponding connector

`step TIME`
waits `TIME` milliseconds before continuing.

`reachfw Dx Dy Dz dirX dirY dirZ Orientation`
'Reaches a referential on the grid. See figure 5.10 page 40 for more explanations. If we add the first fields Dx Dy Dz, the referential will be translated by a vector Dx Dy Dz in the referential M0 If we add an Orientation (last field), the final referential will be turned by an angle of $Orientation * \pi/2$ around the Z-axis.

`reachbw Dx Dy Dz dirX dirY dirZ Orientation`
same as reachfw but starting from the M3 referential of the second module, thus reaches C3X, C3Y, C3Z, 'C2X', C2Y, C2Z. 'C2X' and 'C1X' do not really exist but they can be computed.

`dropBoxFw (resp. dropBoxBw)`
If the kine_controller_physics plugin is activated, puts a small box at the end of the 2nd module (resp at the beginning of the 1st module).

`cmd`
Enables the user to enter these commands at the command line, for testing purposes. Webots must have been run at the command line on linux or the

controller should have a command window on Windows (not tested.)

```
file
```

Re-enables the file parsing after testing on the command line.

# List of Figures

Most of the pictures in this report were made with Inkscape [9].

# Bibliography


[1] BIRG. http://birg.epfl.ch. Biologically Inspired Robotic Group.

[2] Alexander Bogomolny. Plane filling curves. http://www.cut-the-knot.org/do_you_know/hilbert.shtml.

[3] Seth Copen, Goldstein, and Todd Mowry. Claytronics: A scalable basis for future robots. Carnegie Mellon University.

[4] J. Craig. *Introduction to Robotics: Mechanics and Control*. Addison-Wesley Publishing Company, E.U.A., segunda edition, 1989.

[5] 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.

[6] Hugo Elias. Inverse kinematics - improved methods, 2004. http://freespace.virgin.net/hugo.elias/models/m_ik2.htm.

[7] BIRG EPFL. Roombots: Modular robotics for adaptive and self-organizing furniture. http://birg.epfl.ch/page65721.html.

[8] Toshio Fukuda, Martin Buss, Hidemi Hosokai, and Yoshio Kawauchi. Cell structured robotic system cebot - control, planning and communication methods. In Intelligent Autonomous Systems 2, An International Conference, pages 661–671, 1989.

[9] Inkscape. Open-source scalable vector graphic editor. http://www.inkscape.org.

[10] Aubrey Jaffer. Peano's algorithm. http://people.csail.mit.edu/jaffer/Geometry/PSFC.

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

[12] Lydia E. Kavraki. Protein inverse kinematics and the loop closure problem. http://cnx.org/content/m11613/latest/.

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

[14] Simon Lépine. Locomotion in modular robots: Yamor host 3 and roombots. http://birg.epfl.ch/webdav/site/birg/users/160150/public/report.pdf.

[15] Daniel Marbach. Evolution and online optimization of modular robot locomotion., 2006. http://birg.epfl.ch/page56514.html.

[16] Daniel Marbach and Auke J. Ijspeert. Online optimization of modular robot locomotion., 2005. http://birg2.epfl.ch/publications/fulltext/marbach05.pdf.

[17] Jerome Maye. Control of locomotion in modular robotics. http://birg.epfl.ch/page65464.html.

[18] Satoshi Murata and Haruhisa Kurokawa. Self-reconfigurable robotics. http://www.mrt.dis.titech.ac.jp/ICRA-tutorial.html.

[19] Orocos. Kdl tutorials. http://eris.liralab.it/wiki/KDL-simple.

[20] Alexandre Tuleu. Roombots – central pattern generators, symmetries and online learning, 2009.

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

[22] Wikipedia. Self-reconfiguring modular robotics. http://en.wikipedia.org/wiki/Self-Reconfiguring_Modular_Robotics.

[23] Wikipedia. Wikipedia inverse kinematics. http://en.wikipedia.org/wiki/Inverse_kinematics.

[24] Wikipedia. Wikipedia quaternions and spatial rotation. http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation.

[25] Wikipedia. Wikipedia rotation matrix. http://en.wikipedia.org/wiki/Rotation_matrix.

[26] Wikipedia. Wikipedia spherical linear interpolation. http://en.wikipedia.org/wiki/Slerp.

[27] Michel Yerly. Yamor lifelong learning., 2007. http://birg.epfl.ch/page65665.html.

[28] Sadi Yigit, Catherina Burghart, and Heinz Woern. Avoiding singularities of inverse kinematics for a redundant robot arm for safe human robot co-operation., 2003. http://www.sfb588.uni-karlsruhe.de/publikationen/2003/130%20YigitK2_2003.pdf.