

libbats

User Manual

Little Green BATS

University of Groningen, the Netherlands
<http://www.littlegreenbats.nl>

Bold Hearts

University of Hertfordshire, UK
<http://homepages.stca.herts.ac.uk/~epics/boldhearts>

Contents

Introduction	v
1 SimSpark and RCSSServer3D	1
1.1 Basic Architecture	1
1.2 Protocol	2
2 Installation	5
2.1 RoboCup 3D Simspark Simulation Server	5
2.2 libbats	6
2.3 Testing	7
2.4 Troubleshooting	7
3 Running the Simulation	9
3.1 Starting	9
3.2 Controls	9
4 Quick Start	11
5 Main Modules	15
5.1 AgentSocketComm	16
5.2 Cochlea	18
5.3 Clock	18
5.4 AgentModel	19
5.5 WorldModel	19
5.6 Localizer	20
5.7 Cerebellum	21
5.8 HumanoidAgent	22
6 Utility Classes	23
6.1 Rf	23
6.2 Singleton	23
6.3 Distribution	24
6.4 Math	25
6.5 Conf	26
6.6 Types	28

Introduction

If you read this, you will probably already know about the RoboCup initiative: to progress robotics and artificial intelligence through competition, to such a level that in 2050 an artificial football team can beat the human world champions. Several leagues exist to focus on separate aspects of this problem, of which the 3D Soccer Simulation (Sub) League is one. The aim of this league is to develop team strategies and behaviors that are not (yet) feasible in the hardware leagues.

This is the manual of `libbats`, a library that can be used to get started in the RoboCup 3D Simulation. It was originally developed by the RoboCup 3D Simulation team the Little Green BATS in 2006. Currently it is being maintained and used in competitions by the BATS and by team Bold Hearts, both vice world champion teams (in 2007 and 2009 respectively). This library can freely be used, both as in free beer and free speech, to create your own (team of) RoboCup 3D simulation agent(s) and to do research and enter competitions with. It supplies some modules to get you started quickly, is generic enough to add any algorithm, behavior model or learning method and still gives you detailed control of the basics if required.

This manual is intended to give you an overview of the library and to get you familiar with the structure and use of its parts. You can have your own agent running within a few lines of code. More detailed information on all possibilities can be found in the documentation in the source code. If you want you can dig through all this code, but we suggest to install doxygen¹ instead. If you do this and follow the installation instructions in chapter 2, you will find this documentation nicely formatted with HTML in the `docs/html` directory.

The next chapter will give a brief discussion of the SimSpark/RCSSTServer3D simulation environment, which is used in the 3D Soccer Simulation, and the way an agent should interact with it. This chapter can in theory be skipped, since `libbats` offers abstractions and tools such that you don't have to worry about low level issues. However, a general understanding of these will probably be beneficial anyway.

After that we will give a tutorial of how to install the simulation environment and `libbats`, and some configuration options. Chapter 4 takes you through the steps of creating your first agent and shows how easy it is to set up a new team using `libbats`. The next chapters will go into the different parts and modules of `libbats` in more depth.

If something is still unclear, you found a bug or just have a question related to this library, do not hesitate to contact us. Good luck and happy coding!

¹<http://www.stack.nl/~dimitri/doxygen/>

1 SimSpark and RCSSServer3D

This chapter offers a brief introduction of the simulation architecture used in the Robocup 3D Soccer Simulation competitions. The aim of this is to give a global understanding of the inner workings of the system; some details will be skipped, and even then a large part of this information is not necessarily needed to be able to create a team using `libbats`. However, a more fundamental understanding can help a great deal. For a more in-depth treatment, see the SimSpark wiki at <http://simspark.sourceforge.net/wiki>.

1.1 Basic Architecture

The RoboCup 3D Soccer Simulation competitions use a unified simulation platform, SimSpark/RCSSServer3D, to supply the simulation of the soccer field, robot mechanics, game rules, etc. In 2006-2007 the first versions of SimSpark were developed, as a generic physics/robotics simulator, with a specific implementation for RoboCup 3D Soccer Simulation. Later on these two parts were explicitly separated: SimSpark now just includes the basic simulation engine, which takes care of the low level physics simulation and interfacing with agents, and exposes several templates to implement specific robot models, visualization systems and a plug-in system. The collection of implementations and plug-ins specific for the RoboCup 3D Simulation league is placed in the separate RCSSServer3D (RoboCup Soccer Simulation Server 3D) package. Together they form what is commonly referred to as '*the simulation server*', the '*the simulator*', or '*the server*'.

As said, the simulator takes care of all the world and game aspects. It also supplies a fixed physical robot model, which is the same for each team. The simulator determines which sensor information is available for each robot at each time step and executes control commands supplied by the robot's 'brain'. This brain is what a team, you, has to create in order to be able to participate in the RoboCup 3D Soccer Simulation.

The brain of a robot consists of a standalone program and is connected to its body through a TCP/IP connection; the simulator opens a port (by default 3100), to which your program can connect. After some handshaking messages (explained below), the server will send sensor information at each time step (by default every 0.02 seconds). In reaction to such a sensor message your agent should send a control message, before the end of the time step. This message contains control commands for the agent's controllers, in our case these are target angular velocities for the robot's joints. This perception-action loop is represented in Fig. 1.1. This figure also shows that the physics simulation and the visualization of the results of this simulation are separated; the server opens another port (by default 3200), to which a so called '*monitor*' can connect. Through

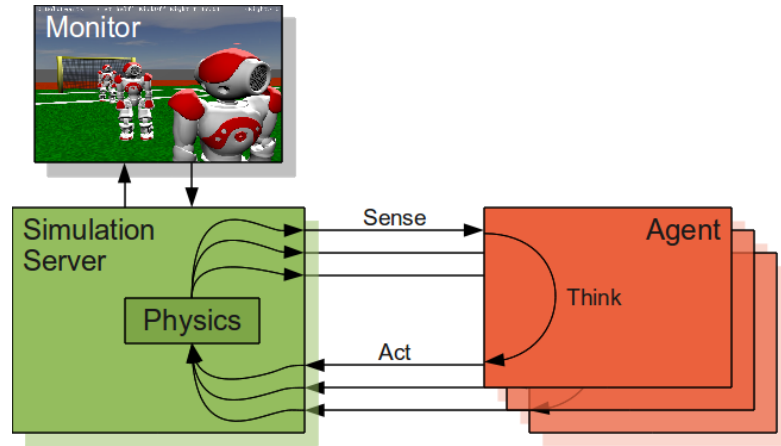


Figure 1.1: Interactions between simulator, monitor and agents. Each box represents a stand alone program, arrows between boxes depict the TCP/IP connections used for communication.

this connection the server sends all information needed by a monitor to make a graphical representation of the current state of the simulation. The monitor in its turn can send command to the server to control the simulation, which is done for instance when starting the game with a kick off.

The communication between the server and the agents and between the server and a monitor follows a Lisp-like protocol, in which messages contain *predicates* in the form of S-expressions. A predicate consists of a list of elements, the first of which is the name and the rest are its parameters. These parameters can again be predicates, resulting in a tree of predicates. As a simple example, the statement “the current game mode number is 6 is contained in the message (`playMode 6`). A more extensive example, taken from an actual simulation run, is given in Fig. 1.2. In the next section we will give an explanation of the different predicates.

1.2 Protocol

Here we will give a small example of a proper communication sequence between an agent and the simulator.

Connect Of course, first of all the agent should connect to the agent port of the server, by default port number 3100.

Create After connection, the agent should ask the simulator to create new robot model for it. This is done with a `scene` predicate, which takes the name of a Ruby Scene Graph (RSG) file that contains the model’s description as an argument:

```
(scene rsg/agent/nao/nao.rsg)
```



```

((time (now 70.32)) (GS (t 0.00) (pm BeforeKickOff)) (GYR (n torso) (rt -0.12
0.13 -0.01)) (ACC (n torso) (a -0.06 -0.06 9.81)) (HJ (n hj1) (ax 0.00)) (HJ (n
hj2) (ax 0.00)) (See (G2R (pol 17.64 -12.45 1.00)) (G1R (pol 17.30 -5.75 0.93))
(F1R (pol 17.50 10.54 -1.74)) (F2R (pol 19.35 -26.93 -1.46)) (B (pol 8.69
-18.65 -3.10)) (P (team Enemy) (id 2) (head (pol 15.79 -15.79 0.06)) (rlowerarm
(pol 15.75 -15.57 -0.84)) (llowerarm (pol 15.85 -15.84 -0.49)) (rfoot (pol
15.79 -15.37 -1.64)) (lfoot (pol 15.79 -16.08 -1.91)))) (HJ (n raj1) (ax
90.00)) (HJ (n raj2) (ax -64.10)) (HJ (n raj3) (ax -0.00)) (HJ (n raj4) (ax
-0.03)) (HJ (n laj1) (ax 90.00)) (HJ (n laj2) (ax 64.10)) (HJ (n laj3) (ax
-0.00)) (HJ (n laj4) (ax 0.04)) (HJ (n rlj1) (ax 0.01)) (HJ (n rlj2) (ax
-0.05)) (HJ (n rlj3) (ax 0.01)) (HJ (n rlj4) (ax -0.00)) (HJ (n rlj5) (ax
0.02)) (FRP (n rf) (c 0.02 -0.03 -0.01) (f 0.19 0.08 21.04)) (HJ (n rlj6) (ax
0.08)) (HJ (n llj1) (ax 0.00)) (HJ (n llj2) (ax 0.01)) (HJ (n llj3) (ax -0.00))
(HJ (n llj4) (ax -0.00)) (HJ (n llj5) (ax 0.03)) (FRP (n lf) (c -0.01 -0.03
-0.02) (f -0.02 0.11 24.22)) (HJ (n llj6) (ax 0.00)))

```

Figure 1.2: Example message sent by the server to an agent. Red: time, blue: game state, brown: gyroscopic sensor, green: accelerometer, purple: joint angles, orange: vision.

Initialize The server will now start sending messages to the agent. However, before doing something, the agent has to finalize its initialization. It has to tell the simulator from which team it is and what uniform number (*'unum'*) it wants, by sending an *init* predicate:

```
(init (unum 2)(teamname MyTeam))
```

If *unum 0* is sent, the simulator will appoint the next free uniform number to the agent. In response to this message, the simulator will reply with a message containing confirmation of the uniform number selected by the agent, or chosen by the simulator, and on which side of the field, left or right, the agent's team plays:

```

((time (now 5.80)) (GS (unum 2) (team left) (t 0.00) (pm BeforeKickOff))
(GYR (n torso) (rt -0.00 0.00 0.00)) (ACC (n torso) (a 0.00 0.00 9.81))
(HJ (n hj1) (ax -0.00))...

```

This completes the initial handshake sequence.

Sense From now on, the simulator sends messages containing time, game state and perceptor information at each time step. You have already seen an example of these in Fig. 1.2.

Act After each simulator message, the agent should reply with an action message. Such a message contains control commands for the agent's actuators, i.e. a target angular velocity for each joint¹. For instance, the following message tells the simulator to

¹In degrees per second. Note however that **libbats** internally uses radians per second (and radians for

1 *SimSpark and RCSSServer3D*

move the two head joints **he1** and **he2**, and the first joints of both arms, **lae1** and **rae2**:

```
(he1 11.759) (he2 -8.4038) (lae1 -18.987) (rae1 -18.985)
```

If for some joints no control is specified, the last target angular velocity that was sent is used by the simulator. Besides the joint actuators, there is a special 'beam' actuator, that can be used to quickly position an agent before kick-offs. It takes as arguments the x and y coordinates to beam to and the angle to face at. So, to beam to coordinates (-7, 1.5), while face angle 0, which is towards the opponent's goal, the agent should send the following message:

```
(beam -7 1.5 0)
```

angles in general).

2 Installation

This section describes the installation procedure for the RoboCup 3D Simspark simulation server and for the `libbats` library. These instructions have been tested and verified on Ubuntu 10.04, but may also work for other distributions. The RoboCup 3D Simspark simulation server is in active development, so the following instructions may not work as expected. Please refer to the end of this chapter for instructions on what to do if the installation instructions described here fails.

2.1 RoboCup 3D Simspark Simulation Server

Follow the following instructions to install Simspark and the Robocup 3D Simspark Simulation Server.

Repositories Make sure that the Universe and Multiverse repositories are enabled:

```
$ sudo gedit /etc/apt/sources.list
[ Follow instructions in the file to enable the Universe
and Multiverse repositories, save and exit ]
$ sudo apt-get update
```

Dependencies Install the necessary dependencies and tools:

```
$ sudo apt-get install build-essential libboost-dev \
> libboost-thread-dev libboost-serialization-dev \
> libboost-regex-dev > libsdl1.2-dev libfreetype6-dev \
> libdevil-dev libslang2-dev libjpeg62-dev libtiff4-dev \
> libpng12-dev ruby1.8 ruby1.8-dev
```

ODE Install the latest version of the Open Dynamics Engine (ODE):

1. Download the `ode-src-0.x.y.zip` (where 0.x.y is the latest version number, currently 0.11.1) source code package from:
`http://sourceforge.net/projects/opende/files/`
2. Unpack and navigate to the source code directory:

```
$ unzip ode-src-0.x.y.zip
$ cd ode-0.x.y
```
3. Configure, make and install:

2 Installation

```
$ ./configure --enable-double-precision --enable-release \  
> --disable-asserts --with-drawstuff=none --disable-demos  
$ make  
$ sudo make install
```

SimSpark Install the latest version of SimSpark:

1. Download the simspark-0.x.y (where 0.x.y is the latest version number, currently 0.2.1) source code package from:
<http://sourceforge.net/projects/simspark/files/>
2. Unpack and navigate to the source code directory:

```
$ tar xvzf simspark-0.x.y.tar.gz  
$ cd simspark-0.x.y
```
3. Configure, Build and Install the package:

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

RCSSServer3D Install the latest version of RCSSServer3D:

1. Download the rcssserver3d-0.x.y (where 0.x.y is the latest version number, currently 0.6.4) source code package from:
<http://sourceforge.net/projects/simspark/files/>
2. Unpack and navigate to the source code directory:

```
$ tar xvzf rcssserver3d-0.x.y.tar.gz  
$ cd rcssserver3d-0.x.y
```
3. Configure, Build and Install the package:

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

2.2 libbats

Dependencies Install the necessary dependencies:

```
$ sudo apt-get install libxml2-dev libsigc++-2.0-dev libeigen2-dev
```

libbats Install the latest version of libbats:

1. Download the libbats-x.y.tar.gz (where x.y is the latest version number, currently 2.0) source code package from:
<https://launchpad.net/littlegreenbats/+download>
2. Unpack and navigate to the source code directory:


```
$ tar xvzf libbats-x.y.tar.gz
$ cd libbats-x.y
```
3. Configure, make and install:


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

2.3 Testing

To test whether everything is working correctly, start the simulator with:

```
$ rcsoccersim3d
```

This should start the simulator and the monitor, showing a football pitch. Now start the example agent supplied with libbats; in the libbats source directory execute:

```
$ cd examples/helloworld
$ ./helloworld
```

If everything went well, an agent should appear, standing in the left side of the field, waving its arms.

2.4 Troubleshooting

1. To uninstall Simspark or RCSSServer3D, simply execute the following command within the respective build directory:

```
$ sudo make uninstall
```

2. If compilation fails due to missing dependencies, try looking up the required libraries using the package manager:

```
$ sudo apt-get install lib<your package>[press tab several times]
```

3. Please refer to the SimSpark wiki for more information on how to install (or use) simspark, e.g. when using a different operating system:
http://simspark.sourceforge.net/wiki/index.php/Main_Page.

3 Running the Simulation

3.1 Starting

To start the simulation server, run the following:

```
$ rcssserver3d
```

This will generate some output to the terminal, including a list of 'not found' and 'Unknow function' errors. These are actually more like warnings and can be ignored. Next, you will have to start a monitor to be able to see what is going on in the simulation. If you run the monitor on the same machine as the simulator, use:

```
$ rcssmonitor3d
```

If you run the monitor on a different machine, which can boost the speed of the simulator if you experience lag, use:

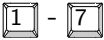











```
$ rcssmonitor3d --server 192.168.1.2
```

Of course, use the actual address of the machine you run the server on. You can connect as many monitors you want (and your machine(s) can handle).

For convenience, there is also a script that starts both the server and the monitor, and kills the server when you close the monitor:

```
$ rcsoccersim3d
```

3.2 Controls

	Move camera to default positions
 ,  ,  , 	Move camera horizontally
 , 	Move camera vertically
Click + drag	Turn camera
	Kick-off
	Drop ball
	Free kick for the left team
	Free kick for the right team
	Quit

4 Quick Start

This chapter is intended to show how easy it is to create an agent using `libbats`. By following these steps you will recreate the simple Hello World example agent that is supplied with the library. See the next chapters for more detailed information on the modules that are used, or when you only need a small part of the library, like communication with the simulation server.

The base of a `libbats` agent is the `HumanoidAgent` class. This class initializes all parts of the library and supplies a simple life cycle for your agent. So let's start by creating your own agent class by extending `HumanoidAgent`. Of course, we have to create a constructor and the `HumanoidAgent` class requires that your agent defines an `init()` and a `think()` method. Listing 4.1 shows what your header file may look like.

Now, what should these methods do?

MyAgent() The constructor should give some initialization information to the constructor of the base class `HumanoidAgent`. At least the name of your team should be supplied, but you could also set parameters such as the host address and port

Listing 4.1: `myagent.hh`

```
1  #ifndef INC_MYAGENT_HH_
2  #define INC_MYAGENT_HH_
3
4  #include <bats/HumanoidAgent/humanoidagent.hh>
5
6  /** My first agent */
7  class MyAgent : public bats::HumanoidAgent
8  {
9      /** Initialize agent */
10     virtual void init() {}
11
12     /** Think cycle */
13     virtual void think();
14
15 public:
16
17     /** The Constructor */
18     MyAgent()
19     : HumanoidAgent(std::string("MyTeam"))
20     {
21     }
22
23 };
24
25 #endif
```

4 Quick Start

number to connect to. See the following chapters and details in `HumanoidAgent`'s class documentation for more information on these parameters.

init() This method is called once after the agent is created, a connection to the simulator is established, and all parts of the library are initialized. You can use this to initialize your own things, like a formation module, movement generators, et cetera.

think() Here is where you put your agent's 'brain'. After the agent is started and initialized, this method is called at every think cycle, 50 times per second. When the **think()** method is called, new sensor information from the server is read and integrated in different modules, like the `AgentModel`, the `WorldModel` and the `Localizer` (more on these later). In this method your agent should decide what to do and make sure actions for the current think cycle are sent to the server.

At the moment we don't have our own fancy modules yet, so the constructor is empty, as well as the **init()** method. However, we do want our agent to do something cool, so we will fill the **think()** method as shown in listing 4.2 to make him wave his arms at us. Let's look at what all of this does.

lines 1-6 First include the header file of your agent class, here `helloworldagent.hh`, and the header files of the modules that are used. All `libbats` classes are in the `bats` namespace, so in line 6 we import this namespace so we don't have to type `bats::` all the time.

lines 10-12 Here references to the used modules are requested. Most modules are so called *singletons*, which means there is only one instance of each class. The `Clock` and `AgentModel` do what you probably already expect they do: they give the current time and a model of the agent's state. The `Cerebellum` is named after the part of your brain that handles control and coordination of your movements and is used to actually do stuff, as you will see later on.

line 14 Get the current time.

lines 17-21 We want our agent to wave his arms, by moving his shoulder joints. To do this it is useful to know the current state of these joints. This sounds like a job for the `AgentModel` and as you can see it is. The `Types` class defines all sorts of handy types used by several modules.

lines 24-28 Next we define the angles we want to move the joints to. Here a sinusoidal pattern is used to create a smooth, friendly waving behavior.

lines 31-33 The agent is controlled by setting the angular velocities of its joints, so here these are calculated based on the current and target angles and a gain factor.

lines 36-39 As mentioned earlier, the `Cerebellum` is used to act. It is fed with actions, in this case joint movements, but it also controls the other actuators like speech and beaming.

Listing 4.2: myagent.cc

```

1  #include "helloworldagent.hh"
2  #include <bats/Clock/clock.hh>
3  #include <bats/AgentModel/agentmodel.hh>
4  #include <bats/Cerebellum/cerebellum.hh>
5
6  using namespace bats;
7
8  void HelloWorldAgent::think()
9  {
10     Clock& clock = SClock::getInstance();
11     AgentModel& am = SAgentModel::getInstance();
12     Cerebellum& cer = SCerebellum::getInstance();
13
14     double t = clock.getTime();
15
16     // Get current joint angles
17     double angles[4];
18     angles[0] = am.getJoint(Types::LARM1)->angle.getMu()(0);
19     angles[1] = am.getJoint(Types::LARM2)->angle.getMu()(0);
20     angles[2] = am.getJoint(Types::RARM1)->angle.getMu()(0);
21     angles[3] = am.getJoint(Types::RARM2)->angle.getMu()(0);
22
23     // Calculate target joint angles
24     double targets[4];
25     targets[0] = 0.5 * M_PI;
26     targets[1] = 0.25 * M_PI * sin(t / 2.0 * 2 * M_PI) + 0.25 * M_PI;
27     targets[2] = 0.5 * M_PI;
28     targets[3] = -0.25 * M_PI * sin(t / 2.0 * 2 * M_PI) - 0.25 * M_PI;
29
30     // Determine angular velocities
31     double velocities[4];
32     for (unsigned i = 0; i < 4; ++i)
33         velocities[i] = 0.1 * (targets[i] - angles[i]);
34
35     // Add joint movement actions to the Cerebellum
36     cer.addAction(new Cerebellum::MoveJointAction(Types::LARM1, velocities[0]));
37     cer.addAction(new Cerebellum::MoveJointAction(Types::LARM2, velocities[1]));
38     cer.addAction(new Cerebellum::MoveJointAction(Types::RARM1, velocities[2]));
39     cer.addAction(new Cerebellum::MoveJointAction(Types::RARM2, velocities[3]));
40
41     // Tell the Cerebellum to send the actions to the simulator
42     cer.outputCommands(SAgentSocketComm::getInstance());
43 }

```

Listing 4.3: main.cc

```
1 #include "myagent.hh"
2
3 int main()
4 {
5     MyAgent agent;
6     agent.run();
7 }
```

line 42 When the `Cerebellum` has gathered all actions, it is time to send them to the simulation server. A `SocketComm`, in the form of the specialized `AgentSocketComm`, is needed for this, which handles the actual complicated communication through sockets.

And that's it! Almost. The only thing left to do now is to create an actual executable program that runs our agent. This is done by defining the standard `main()` method, creating an instance of the agent class and tell it to run, as shown in listing 4.3. Now compile these files with your favorite building method (ours is `ccbuild`¹), fire up `RCSSServer3d`, run the agent binary and wave back at your new friend!

¹<http://ccbuild.sourceforge.net/>

5 Main Modules

The BATS agent architecture consists of several parts and layers. This tutorial guides you through using these parts step by step. All lower layers are independent of the higher layers, so if you do not need all layers, you can skip the later sections. For instance, if you are only interested in an easy interface with the simulation server, reading section 5.1 will suffice. If you want to start quickly with a working agent, you can skip until section 5.8 for now. However, the agent template described there is based on the elements described in the sections before that, so be sure to read those too at some time, to fully understand how your agent works.

The main `libbats` modules and their relations are shown in Fig. 5.1. As you can see, they can be divided into several layers:

- 1: Low level communication** As discussed in chapter 1, communication between the simulator and an agent is done using an ASCII, S-expression based protocol through a TCP/IP connection. The `AgentSocketComm` module handles setting up this connection, reading and writing messages, and parsing these messages into and from more manageable data structures.
- 2: Input and output integration** In the second layer, input and output is handled at a slightly higher level. On the input side, the `Cochlea` extracts all data from the still text-based messages supplied by the `AgentSocketComm` and turns that data into readily usable binary values. The `Cerebellum` is used to gather control commands,

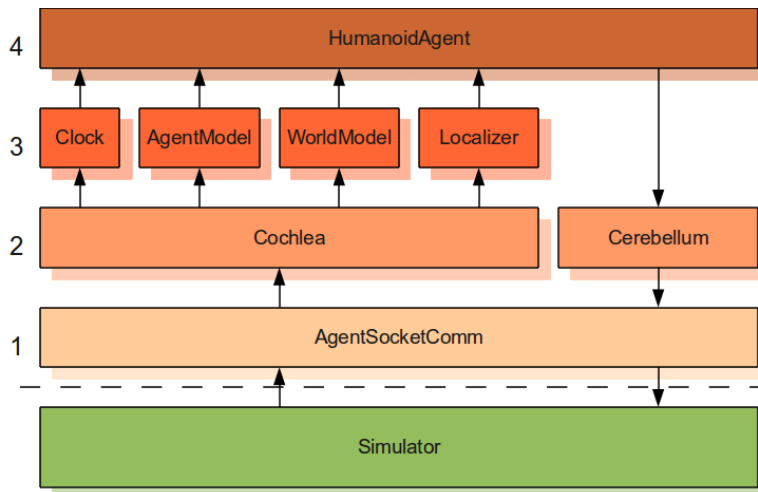


Figure 5.1: libbats modules

5 Main Modules

work out contradictions if necessary and turn them into the text based structures that the **AgentSocketComm** understands.

- 3: Models** The 4 modules in the third layer use the data from the **Cochlea** to update models of the current state of the world, i.e. the current time, the state of the agent's body, the state of the world and the game, and the location of all objects in the field.
- 4: Intelligence** Finally, at the highest level, the actual intelligence of the agent is implemented. An instance of **HumanoidAgent** has access to all information gathered in the different modules, decides upon actions based on this information, and submits these actions to the **Cerebellum**.

In the following sections we will describe in more detail how each module can be used. While doing so, we will encounter different helper and utility classes. Please refer to later chapters for more details on these. Also, again, for more information, make sure to read the Doxygen documentation contained in the source.

5.1 AgentSocketComm

As mentioned earlier, the lowest layer in the library manages the communication with the simulation server. This communication is done through TCP sockets and consists of S-expressions (predicates) that the agent and server send back and forth. To make sure you don't have to worry about what this stuff actually is and does, the library offers you the **AgentSocketComm**. This class handles the connection to the server and sending, receiving and parsing of messages. This section explains how to use this module. If you use the **HumanoidAgent** class, all this is done for you.

Before you can use **AgentSocketComm**, you have to supply a host name and a port number to connect to. When running the server on the same computer as your agent, with default settings, these are 'localhost' and '3100'. After this is done, the first thing to do is to open an actual connection by calling `connect()`:

```
AgentSocketComm& comm = SAgentSocketComm::getInstance();  
comm.initSocket("localhost", 3100);  
comm.connect();
```

Note that the **AgentSocketComm** is a singleton, refer to later chapters for information on what this is. The **AgentSocketComm** keeps two internal message queues, one for input and one for output. These queues are filled and emptied, respectively, when calling **AgentSocketComm**'s `update()` method. This call blocks until new data is received from the server:

```
comm.update();
```

SocketComm supplies several methods to place messages that should be sent to the server into the output queue. First of all, you can build your own predicate using the `Predicate` and/or `Parser` classes¹ and put it directly into the queue by calling the `send` method:

```
rf<Predicate> myPredicate = makeMyPredicate();
comm.send(myPredicate);
```

However, you can also leave the trouble of building the predicates to `AgentSocketComm` by using its `make*Message()` methods. And if you want it totally easy, use the `init()`, `beam()` and `move*()` methods, which not only build the predicates, but also place the messages directly into the queue for you.

The input queue holds the messages received from the server. To check whether there is a new message, you can call `hasNextMessage`. To extract the next message, you can use `nextMessage()`:

```
while (comm.hasNextMessage())
    rf<Predicate> message = comm.nextMessage();
```

To conclude and summarize this section, a typical way to have successful communication with the server is presented here:

```
// Get the AgentSocketComm, initialize and connect
AgentSocketComm& comm = SAgentSocketComm::getInstance();
comm.initSocket("localhost", 3100);
comm.connect();

// Create robot model
rf<Predicate> scene = new Predicate("scene");
scene->pushLeaf("rsg/agent/" + am.getRSG());
comm.send(scene);

// Wait for the first message from the server
comm.update();

// Identify yourself to the server
comm.init(0, "MyTeam");

// Main loop
while (true)
{
    comm.update();
    while (comm.hasNextMessage())
        handleMessage(comm.nextMessage());
}
```

¹This method is not described in this manual. Look at the documentation of the respective classes for more info

5.2 Cochlea

The `AgentSocketComm` parses the S-expressions that the agent receives from the server into a `Predicate` structure. However, to extract useful data you still have to dig through these structures. The `Cochlea` offers a layer over the `AgentSocketComm` to extract information from the predicates received from the server and present it in an easily accessible way.

Before using the `Cochlea` you have to initialize some parameters. You have to let it know what the name of your team is, so it can tell team mates and opponents apart:

```
Cochlea& cochlea = SCochlea::getInstance();
cochlea.setTeamName("MyTeam");
```

Next you have to set up translations for joint-angle sensors. The `Cochlea` uses internal names for these, that may not be the same as the names used in the messages sent by the server. These internal names are "head1", "head2", "larm1" to "larm4" for the left arm, "rarm1" to "rarm4" for the right and "lleg1" to "lleg6" and "rleg1" to "rleg6" for the legs. The Nao robot used by the server for instance uses names of the form "laj1" and "rlj3", so these have to be translated:

```
cochlea.setTranslation("laj1", "larm1");
cochlea.setTranslation("rlj3", "rleg3");
```

This way the `Cochlea` can handle different robot models. If you use the `AgentModel` module, this is done for you.

Now you can start using the `Cochlea` by calling `update()` every time a new message is received by the `AgentSocketComm`. This will integrate the information of the message, after which you can request data with the `getInfo()` method, or one of the methods for more complex data:

```
comm.update();
cochlea.update();
// Get the polar coordinates of the ball
Vector3D polarBallPos = cochlea.getInfo(
    Cochlea::iVisionBall);
```

5.3 Clock

The `Clock` is pretty straightforward: it tells the current time:

```
double t = SClock::getInstance().getTime()
```


5.4 AgentModel

The

AgentModel keeps a model of the agent's own state. It keeps track of joint angles and data of other sensors, as well as some higher level data, like the position of the Center Of Mass (COM). The **AgentModel** also needs some initialization before it can be used. You have to tell it the uniform number of the agent, after which you should call the `initBody()` method:

```
AgentModel& am = SAgentModel::getInstance();
am.setUnum(unum);
am.initBody();
```

This loads an XML configuration file that contains the names, sizes and weights of the agent's body parts and joints. It also uses this data to set up the translations for the **Cochlea** for you, so you don't have to do this by hand. If default settings are used, the default configuration file `conf.xml` is loaded, which is installed with the library and which imports the `nao.mdl.xml` file for each agent to get the description of the Nao robot model. For more information on loading XML configuration files and the robot model descriptions, see the documentation of the **Conf** class.

After initialization, the **AgentModel** is also ready to be updated and used:

```
comm.update();
cochlea.update();
am.update();

Vector3D com = am.getCOM();
```

5.5 WorldModel

The raw data offered by the **Cochlea** may not be directly usable and perhaps you want to have some information that is deduced from these facts. This is exactly what the **WorldModel** is for. It for instance gives the current game state and field size, but also higher level information like which team should take the kick-off, or if there is another player closer to the ball.

To start, the **WorldModel** also needs to know the name of your team for some of its capabilities:

```
WorldModel& wm = WorldModel::getInstance();
wm.setTeamName("MyTeam");
```

Next, the model has to be updated at every cycle. The **WorldModel** extracts data from the **Cochlea**, so make sure it is updated before updating the **WorldModel**:

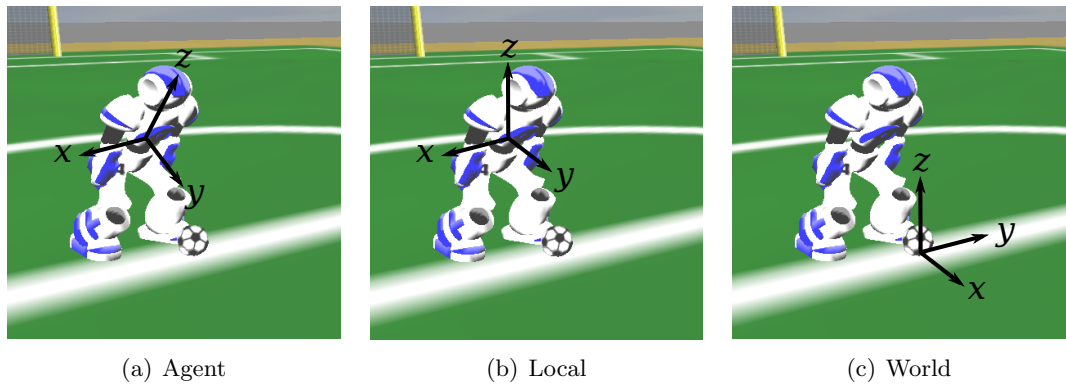


Figure 5.2: The coordinate systems used in `libbats`: (a) Agent ('raw') coordinates, (b) local coordinates, and (c) global coordinates.

```
comm.update();
cochlea.update();
wm.update();

bool shouldWeKickOff = wm.weGetKickOff();
```

5.6 Localizer

The `WorldModel` provides the state of the game, such as the game time, the gamestate, and the team name. However, often the agents will want to know where they are, where the ball is, or where their opponents are. This can be obtained through the `Localizer`. To be able to implement different localization methods, the `Localizer` is an abstract class from which all implementations are derived. One realization of this class is provided in `libbats`, called the `KalmanLocalizer`. As you might have guessed, this `Localizer` uses a Kalman filter to keep track of the locations of the player itself and of other objects. At the start of the program, we have to initialize the `Localizer` and tell it which implementation to use:

```
SLocalizer::initialize<KalmanLocalizer>();
```

Like the other models, the `update()` member function of the `Localizer` should be called each timestep to update the current location estimates with new data from the `Cochlea`. The other member functions of the `Localizer` can be used to obtain the current position of the agent itself, the ball, the other players, or objects such as the goal posts and corner flags.

Within `libbats`, several different coordinate systems are used (see Fig. 5.2:

Agent/raw coordinates The origin of this system is the center of the agent's torso. The positive x axis extends to the right, parallel to the line through the shoulders, the

positive y axis forward out of the torso, and the positive z-axis upwards, through the center of the head. This system is used by the **AgentModel** for the coordinates of body parts. Shown in Fig.5.2(a).

Local coordinates The origin of the local coordinate system is also the center of the agent's torso, but the positive z axis of this system always points upwards, perpendicular to the field. So, the x and y axes always lie parallel to the field, pointing right and forward respectively. This is one of the systems used by the **Localizer** and is the most intuitive to use to determine 'in front of me/to the side of me/above of me' relations. Shown in Fig.5.2(b).

World coordinates The world coordinate system is fully independent of the agent's location and orientation. Its origin is the center of the field, the positive x axis points to the center of the opponent's goal, the positive y-axis to the left when looking along the x axis, and the positive z axis points up, perpendicular to the field. This is the second system that is used by the **Localizer** and is the best one to use to determine global relations. Shown in Fig.5.2(c).

5.7 Cerebellum

The **Cochlea** takes the trouble of having to deal with raw predicates on the input side. On the output side the **Cerebellum** is there for you. It supplies more useful structures to define actions and the possibility to integrate actions from different sources in your agent. The **Cerebellum** defines the **Action** substructure and a few of its derivatives with which you can make new actions. At the moment there are 5 different action types available:

MoveJointAction Move a hinge joint or one axis of a universal joint.

MoveHingeJointAction Move a hinge joint.

MoveUniversalJointAction Move both axes of a universal joint.

BeamAction Beam to a certain position.

SayAction Shout something.

If you don't understand the difference between hinge and universal joints, don't worry. Just use **MoveJointActions** and let the **Cerebellum** handle the rest.

The **Cerebellum** again is a singleton that can be retrieved by calling **SCerebellum::getInstance()**, after which you can add actions to it. After all the sub parts have added their actions, you can call **outputCommands()** to send them through an **AgentSocketComm**:

```
Cerebellum& cer = SCerebellum::getInstance();
```

5 Main Modules

```
rf<MoveJointAction> action =  
    new MoveJointAction(Types::LLEG1, 0.1);  
cer.addAction(action);  
  
cer.outputCommands(comm);
```

When more than one action is supplied for a joint, the angular velocities will be added together before sending the actions to the server.

5.8 HumanoidAgent

Now you have learned about how to initialize, update and maintain the several models of the library, you will learn how to forget all that. The **HumanoidAgent** class does this all for you. It connects the **AgentSocketComm**, initializes the **Clock**, **AgentModel**, **WorldModel** and **Localizer** and updates all modules in the correct order at every cycle. See chapter 4 to learn how to use this class to set up your own agent.

6 Utility Classes

Next to the main modules described in the previous chapter, **libbats** offers a collection of utility classes. We will discuss these here.

6.1 Rf

When working with pointers and dynamic memory allocation, things could get messy. A memory leak is easily produced and premature deletion of data can cause crashes. The **libbats** library uses a garbage collection system to handle these problems, in the form of the **rf<T>** template class. This class is a wrapper around a pointer, keeps track of the number of times the pointer is used and deletes the data when this number is zero (i.e. it performs reference counting). Its behavior is similar to the smart pointers offered by the Boost library. An **rf** is mostly interchangeable with regular pointers; all operations you usually do directly on a pointer you can do on an **rf** and one can be assigned to the other:

```
rf<MyClass> myRf = new MyClass();
myRf->doSomething();
MyClass& myReference = *myRf;
```

It is recommended to use **rf**'s any time you want to use dynamically allocated memory. When you create your own class that you want to be used with **rf**, it is required that that class extends the **RefAble** class. This is needed for the reference counting to work:

```
#include <bats/RefAble/refable.hh>
class MyClass : public bats::RefAble
{
    // Member definitions
};
```

6.2 Singleton

Many modules of the library are so called singletons, so we will give a short introduction about what this is and how they are used. The singleton pattern is a design pattern that makes sure that there is no more than one instance of a certain class. For instance, there is only one Queen of The Netherlands, it would make no sense to create multiple instances. In the singleton design pattern, special measures are taken to prevent you from copying the single instance or creating new objects of the class. This pattern

is used in the library for modules for which it makes no sense and for which it could cause problems if there are multiple instances. This is for instance the case for modules keeping track of states, such as the `Localizer` and the `AgentModel`, and a module for maintaining the connection with the server.

In this library, singletons are implemented by the `Singleton<T>` template class. It offers the static `getInstance()` method to request a reference to the single instance of class `T`. For instance, the following shows how to get a reference to the `WorldModel`:

```
WorldModel& wm = Singleton<WorldModel>::getInstance();
```

For each singleton class of the library, a `typedef` is set for the `Singleton<T>` instantiation of that class, formed by prefixing the class name with a capital `S`. So another way to write the previous example would be:

```
WorldModel& wm = SWorldModel::getInstance();
```

If you want to use the `Singleton<T>` template to create singletons of your own classes, make sure it adheres to the following points:

- Your class must have a default constructor.
- Make all constructors, including the copy constructor, of your class private.
- Make the assign operator, `operator=`, private.
- Define `Singleton<T>`, instantiated with your own class, as friend of your class.

If you do this, your class definition will look like this:

```
class MySingleton
{
    friend class bats::Singleton<MySingleton>;
private:
    // Default constructor
    MySingleton() {}
    // Copy constructor, not implemented
    MySingleton(MySingleton const& other);
    // Assignment operator, not implemented
    MySingleton& operator=(MySingleton const& other);
public:
    // Some public stuff
};
// libbats style singleton typedef
typedef bats::Singleton<MySingleton> SMySingleton;
```

6.3 Distribution

Agents often have to work with uncertainty and probability distributions. For instance, vision data received from the simulator is limited and noisy, therefore location estimates

derived from this data are just that: estimates, with a certain variability. To deal with this, `libbats` offers the `Distribution` template. This template supports distributions over any number of dimensions, though most commonly 1D distributions, e.g. for joint angles, and 3D distributions, e.g. for locations, are used. Currently there is only a single implementation of this template, the `NormalDistribution`, however it is possible to create new distribution types, like Monte Carlo or histogram based representations.

The most common operation on a distribution is to get the mean value, or 'the most likely' value, which is done with the `getMu()` method. For instance, when asking the `Localizer` for a location, it returns a (rf to a) 3D distribution, so to get the most likely local location of the ball you use:

```
Vector3D ballLoc = localizer.getLocationLocal(Types::BALL)->getMu();
```

Other useful methods are `getSigma()`, to get the distribution's covariance matrix, and `draw()`, to draw a random value from the distribution:

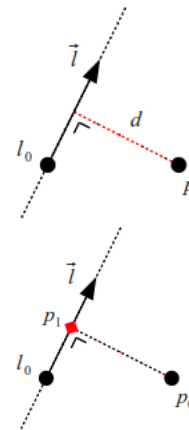
```
Matrix3d ballVariance = localizer.getLocationLocal(Types::BALL)->getSigma();
// 1 dimensional normal distribution
rf<Distribution> myDist = new NormalDistribution(1)
// Initialize distribution with mean 0 and variance 1
Vector1d mu = Vector1d::Zero();
Matrix1d sigma = Matrix1d::Ones();
myDist->init(mu, sigma);
// Draw a random value
Vector3d v = myDist->draw();
```

6.4 Math

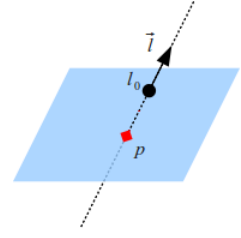
Several common mathematical problems that are useful in 3D Soccer Simulation are included in `libbats` as methods of the `Math` class. Some of these are pretty self-explanatory, for the rest see the following descriptions (and again the documentation in the code itself):

distanceLinePoint This method is used to calculate the distance between a line and a point. The line, dashed black in the example to the right, is defined by a point vector l_0 and a direction vector \vec{l} , the point p is also a point vector. The method returns the length of the red, dashed line d .

linePointClosestToPoint This method is used to determine the point on a line that is closest to a given point. The line, dashed black in the example to the right, is defined by a point vector l_0 and a direction vector \vec{l} , the point p_0 is also a point vector. The method returns the coordinates of point p_1 .



intersectVectorPlane Determine the coordinates of the intersection of a line with a plane. The line, dashed black in the example to the right, is defined by a point vector l_0 and a direction vector \vec{l} . The plane is defined by the vector $(a, b, c, d)^T$, such that $ax+by+cz = d$. In this representation, the vector $(a, b, c)^T$ is normal to the plane and d is the distance of the plane to the origin of the reference frame. The method returns the coordinates of point p .



6.5 Conf

The `libbats` library includes an XML based configuration module, realized by the `Conf` class. This module is for instance used by the `AgentModel` to load the specifications of the robot model, such as sizes, weights and joint names. It is also possible to add new configuration settings, which you can use in your own code.

As most modules, `Conf` is a singleton. When a reference to it is requested for the first time, it loads the default configuration file `conf.xml`, which is supplied and installed along with the library (it can be found in the `xml` directory and is usually installed in `/usr/local/share/libbats/`). Listing 6.1 shows the content of this file.

The configuration file has to contain at least the following elements:

Root The root node should be a `conf` element. (lines 2 and 19)

Player definition For each uniform number used, a `player` element should be added.

These elements should have an `id` attribute, which is the uniform number, and a

Listing 6.1: `conf.xml`

```

1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <conf xmlns:xi="http://www.w3.org/2003/XInclude">
3   <player-class id="nao">
4     <xi:include href="nao-mdl.xml"/>
5   </player-class>
6
7   <player id="0" class="nao" />
8   <player id="1" class="nao" />
9   <player id="2" class="nao" />
10  <player id="3" class="nao" />
11  <player id="4" class="nao" />
12  <player id="5" class="nao" />
13  <player id="6" class="nao" />
14  <player id="7" class="nao" />
15  <player id="8" class="nao" />
16  <player id="9" class="nao" />
17  <player id="10" class="nao" />
18  <player id="11" class="nao" />
19 </conf>

```


class attribute, which determines of which class a player is. (lines 7-18)

Classes Each player can be of a different class, e.g. keeper, defender or attacker, and each class can have different configuration settings. To achieve this, a **player-class** element should be added for each class, with an appropriate **id** attribute. A player class is given to a player by setting its **id** as value of the player's **class** attribute. (lines 3 and 5)

Model For each player class a model description should be given. The description for the Nao-based model used in the 3D Soccer Simulation competitions is supplied with the library in the file `nao.mdl.xml` and included in the default configuration. There is most likely no need for you to change this. (line 4)

Besides these necessary elements, there are a few optional configuration settings used by the library. Currently these only include field dimensions (length, width and goal width), and a setting whether or not a match is restarted and the teams switch sides at half-time, which is used to determine who gets the kick-off in the second half. These settings are global and the same for all players, so they are placed in a **parameters** element directly under the root node:

```
<conf xmlns:xi="http://www.w3.org/2003/XInclude">
  <parameters>
    <fieldlength>18</fieldlength>
    <fieldwidth>12</fieldwidth>
    <goalwidth>2.1</goalwidth>
    <halftimerestart>1</halftimerestart> <!-- 1 = true, 0 = false -->
  </parameters>

  ...
</conf>
```

If these parameters are missing, default values will be loaded (which are the values shown above).

It is easy to create your own configuration file. Just make a copy of the default XML file and make sure that this file is loaded by **Conf** at the beginning of execution:

```
Conf& conf = SConf::getInstance();
conf.setFile("myconf.xml");
```

The parameter given to **setFile** should be the path to your configuration file, relative to your agent's binary.

You can now add new parameters to the global **parameters** element, the value of which can be retrieved at runtime using **Conf**'s **getParam** method. This method requires a default value, to be able to determine the type of the parameter, and which is used when the parameter is not defined in the configuration file:

```
// Get the value of the formation parameter.
// e.g. <formation>2</formation>.
// This parameter should be defined in the configuration file,
```

```
// If it is not defined, the default value (here 1) will be used
int formation = conf.getParam("formation", 1);
```

It is also possible to set parameters specific to a player class. To do so, include a `parameters` element into the appropriate `player-class` element, having the same form as the global version. The value of a parameter for a certain player class can now be retrieved by passing the class's id along to `getParam`. The id of the player class of the current agent can be retrieved from the `AgentModel`:

```
// am is a reference to the AgentModel instance
int myformation = conf.getParam(am.getPlayerClass(), "formation", 1);
```

These methods are sufficient to set up a basic configuration. For more complex configuration, `Conf` offers the `selectXPath` method, with which you can select any part of the XML file using X-Path queries¹. This allows you to make your configuration as extensive as you want:

```
// Find out the unum of the team captain
string xpath = "/conf/player[class=/conf/player-class/parameters[captain=1]/../↵
    @id]/@id";
int unum = conf.selectXPath(xpath).front().getContent().c_str().atoi();
```

6.6 Types

The `Types` class holds many useful types and enumerations, used throughout the library. Here is a brief overview of these, but also make sure to look at the code documentation.

PlayMode This enumeration lists all the possible play modes. For modes of which there are two side variants, e.g. kick-off and free kick, there is also an 'us' and a 'them' version. It is recommended to use these instead of the left/right versions, and all libbats modules use the us/them versions. For instance, if the left team gets the kick-off, and our team plays on the right, `WorldModel::getPlayMode()` will return `Types::KICKOFF_THEM`.

Side A simple enumeration to discern left and right in a human-readable manner.

Joint An enumeration of all the agent's joints. This is for instance used to get the current angle of a joint from the `AgentModel`:

```
// Get angle of the first joint of the left arm, in radians
double angle = am.getJoint(Types::LARM1)->angle.getMu()(0);
```

BodyPart An enumeration of all the agent's body parts. This is for instance used to get the current location of a body part from the `AgentModel`:

¹See <http://www.w3.org/TR/xpath/> and <http://www.w3schools.com/xpath/>

```
// Get the position of the left foot, relative to the torso
Vector3d pos = am.getBodyPart(Types::LF00T)->transform.translation;
```

Object This enumeration lists all objects in the environment, such as the ball, players, opponents, and landmarks. For the latter there are left/right and us/them versions. Again, it is recommended to use the us/them variants:

```
// Get the location of the first post of the opponent goal,
// in local coordinates
Vector3d pos = localizer.getLocaltionLocal(Types::GOAL1THEM)->getMu();
```

