libbats
User Manual

Little Green BATS
University of Groningen, the Netherlands

Bold Hearts
University of Hertfordshire, UK
http://homepages.herts.ac.uk/~epics/boldhearts

v. 3.0

Contents

Introduction

If you read this, you will probably already know about the RoboCup initiaive: 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 seperate 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 doxygen [1] 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/RCSSServer3D 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 3 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, preferably through `libbats` ' Launchpad page[2]. Good luck and happy coding!

[1] `http://www.stack.nl/~dimitri/doxygen/`
[2] `http://launchpad.net/littlegreenbats`

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 to see what is going on. 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 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 agent an all teams. 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, i.e. 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

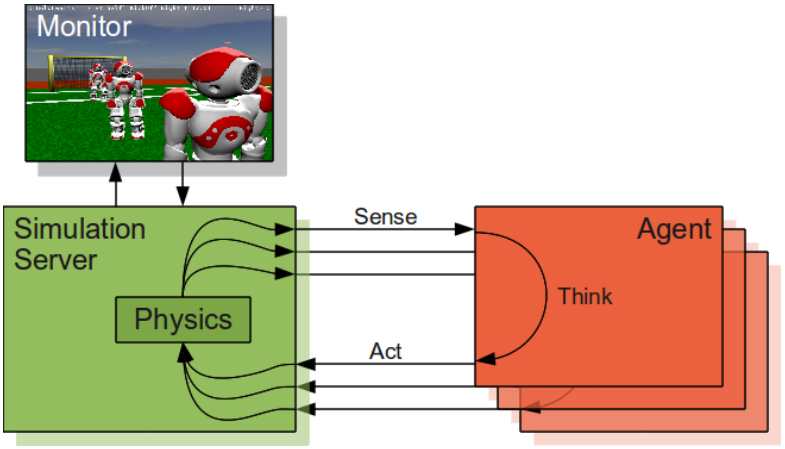


Figure 1.1: Interactions between simulator, monitor and agents. Each box represents a stand-alone program, arrows between boxes depict the TCP/IP connections used for communication.

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 commands 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 a 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. Currently, the 3D Simulation League uses a model based on the Nao robot:

```
(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 anything else, 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. These senses consist of the state of several internal sensors, namely a gyroscopic and an accelerometric sensor measuring the agen's torso angular velocity and linear acceleration respectively, and two pressure sensors on the feet, and of visual data. This visual data contains the coordinates of objects in the field of view of the agent, i.e. the ball, several body parts of other agents, corner flags and goal posts, and lines. The field of view of an agent is by default 120 degrees both

horizontally as vertically.

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 joint. For instance, the following message tells the simulator to move the two head joints `he1` and `he2`, and the first joints of both arms, `lae1` and `rae2`[1]:

```
(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)
```

[1] SimSpark's communication protocol measures angles in degrees. Note however that `libbats` internally uses radians, (and radians per second for angular velocities).

2 Installation

This section describes the installation procedure for the RoboCup 3D Simspark simulation server and for the `libbats` library.

2.1 RoboCup 3D Simspark Simulation Server

For the latest installation instructions of the simulator, see the SimSpark project's wiki:
`http://simspark.sourceforge.net/wiki`
There you will find instructions for installation on various Linux distributions, as well as for Windows and MacOS.

2.2 libbats

Dependencies Install the necessary dependencies. For Ubuntu, and possibly other Debian based distributions, run:

```
$ sudo apt-get install libxml2-dev libsigc++-2.0-dev libgtkmm-2.4-dev
```

On Arch, run as root:

```
$ pacman -S libxml2 libsigc++ gtkmm
```

You can leave out `gtkmm` if you don't want to build the GTK based debugger implementation.

libbats Install the latest version of `libbats`:

1. Download the latest source code package release from:
   `https://github.com/sgvandijk/libbats/releases`
2. Unpack and navigate to the source code directory:
   ```
   $ tar xvzf libbats-x.y.z.tar.gz
   $ cd libbats-x.y.z
   ```
3. Use CMake to configure, then make and install:
   ```
   $ mkdir build && cd build
   $ cmake ..
   $ make
   $ sudo make install
   ```

2.3 RoboViz

RCSSServer3D comes with a monitor, RCSSMonitor3D, however it is rather basic, both graphically as functionally. We suggest using RoboViz instead, which offers much better visualization, and an advanced debugging interface that is supported by `libbats` . Installation instructions for RoboViz can be found at:
`https://sites.google.com/site/umroboviz/`

2.4 Testing

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

```
$ rcssserver3d
```

This command should give you a greeting saying something similar to the following:

```
rcssserver3d (formerly simspark), a monolithic simulator 0.6.5
Copyright (C) 2004 Markus Rollmann,
Universitt Koblenz.
Copyright (C) 2004-2009, The RoboCup Soccer Server Maintenance Group.

Type '--help' for further information
```

plus some initialization output. This output will contain some messages such as '`ERROR: cannot find TextureServer`' and '`ERROR: no FPSController found at '/usr/scene/camera/physics/controller'`'; you can ignore these messages.

Next start RoboViz. Change to the directory containing RoboViz' binary and run:

```
./roboviz.sh
```

After some time in which RoboViz connects to the server and initializes its graphics a green field will appear with some lines and two goals. Ta-da, you are successfully running the simulator! If RoboViz reports 'Disconnected', you are not and something went wrong with installing the simulator; see below for some troubleshooting tips.

Finally start the example agent supplied with `libbats` ; in the `libbats` build 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. You can also try another, more advanced example agent; again, in the `libbats` build directory, run:

```
$ cd examples/dribble
$ ./dribble
```

This should start an agent which, when you start the game, walks to the ball and dribbles it over to the opponent's goal.

2.5 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, as reported when running `cmake` for the simulator or for `libbats` , try looking up the required libraries using your distributions's package manager. For Ubuntu:

   ```
   $ sudo apt-cache search <missing package>
   ```

   For Arch:

   ```
   $ pacman -Ss <missing package>
   ```

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`.

4. Search through the sserver-three-d mailing list to see if there is a solution to your problem:
   `http://sourceforge.net/search/?type_of_search=mlists&group_id=24184`
   otherwise post it to the list (see `http://sourceforge.net/mail/?group_id=24184`).

5. For problems with `libbats` , see the GitHub Issues section at:
   `https://github.com/sgvandijk/libbats/issues`.

3 Quick Start

This chapter is intended to quickly get you started with creating 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.

3.1 Setting Up

First, we will set up the basic project structure for coding and compiling your agent.

- Create a directory that will hold all files, called say '`myagent`. We will refer to this as the source directory.
- Create a directory in this new directory, called '`cmake`', and copy the following files from the `libbats` source directory into it: `FindEigen3.cmake`,`FindSigC++.cmake`, and `LibFindMacros.cmake`.
- Create a file called `CMakeLists.txt` in your source directory and fill it with the content of listing 3.1. This will do the following:

  lines 1-2 Give some header info, such as CMake version and your project name.

  lines 4-8 Use the files copied in the previous step to find and comfigure required libraries. If you didn't build libbats with GTK debugger support, remove line 8, as well as lines 17 and 29.

  line 10 We need to tell the compiler that `libbats` extensively uses C++11 (the 0x standard is used, because older compilers don't support more than that).

  lines 12-18 Tell CMake where to find all library headers.

  lines 20-23 List all source files belonging to your agent that need to be compiled.

  lines 25-30 Tell CMake which libraries to link to.

- Copy the '`xml`' directory fully from the `libbats` source directory to your own source directory.

Listing 3.1: `CMakeLists.txt`

```
cmake_minimum_required (VERSION 2.6)
project (myagent)

set(CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/cmake/)
find_package(Eigen3 REQUIRED)
find_package(LibXml2 REQUIRED)
find_package(SigC++ REQUIRED)
PKG_CHECK_MODULES(GTKMM gtkmm-2.4)

set(CMAKE_CXX_FLAGS "-std=c++0x")

include_directories(
  ${EIGEN3_INCLUDE_DIR}
  ${LIBXML2_INCLUDE_DIR}
  ${SigC++_INCLUDE_DIRS}
  ${LIBXMLXX_INCLUDE_DIRS}
  ${GTKMM_INCLUDE_DIRS}
)

add_executable(myagent
myagent.cc
main.cc
)

target_link_libraries(myagent
  bats
  ${LIBXML2_LIBRARIES}
  ${SigC++_LIBRARIES}
  ${GTKMM_LIBRARIES}
)
```

Listing 3.2: `myagent.hh`

```
1  #ifndef MYAGENT_HH
2  #define MYAGENT_HH
3
4  #include <libbats/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     : bats::HumanoidAgent("MyTeam", "xml/conf.xml")
20   {  }
21 };
22
23 #endif
```

3.2 Coding Your Agent

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 3.2 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 some parameters such as the host address and port number to connect to. In this case, the path to a custom XML configuration file is given. 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. However, we do want our agent to do something cool, so we will fill the `think()` method as shown in listing 3.3 to make him wave his arms at us. Let's look at what all of this does.

lines 1-9 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. The `std` namespace is also imported for convenience.

lines 11-17 This is the implementation of the `init` method, which is called once at start-up. Here you should initialize all your modules. At the very least, you must tell `libbats` which classes of `Localizer` and `Debugger` you want to use.

lines 21-23 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 25 Get the current time.

lines 28-32 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 35-39 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 42-44 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 47-50 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.

line 53 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.

Listing 3.3: `myagent.cc`

```
#include "myagent.hh"
#include <libbats/Clock/clock.hh>
#include <libbats/AgentModel/agentmodel.hh>
#include <libbats/Cerebellum/cerebellum.hh>
#include <libbats/Localizer/KalmanLocalizer/kalmanlocalizer.hh>
#include <libbats/Debugger/RoboVizDebugger/robovizdebugger.hh>

using namespace bats;
using namespace std;

void MyAgent::init()
{
  // You must tell libbats which flavor localizer
  // and debugger you are using
  SLocalizer::initialize<KalmanLocalizer>();
  SDebugger::initialize<RoboVizDebugger>();
}

void MyAgent::think()
{
  Clock& clock = SClock::getInstance();
  AgentModel& am = SAgentModel::getInstance();
  Cerebellum& cer = SCerebellum::getInstance();

  double t = clock.getTime();

  // Get current joint angles
  double angles[4];
  angles[0] = am.getJoint(Types::LARM1)->angle->getMu()(0);
  angles[1] = am.getJoint(Types::LARM2)->angle->getMu()(0);
  angles[2] = am.getJoint(Types::RARM1)->angle->getMu()(0);
  angles[3] = am.getJoint(Types::RARM2)->angle->getMu()(0);

  // Calculate target joint angles
  double targets[4];
  targets[0] = 0.5 * M_PI;
  targets[1] = 0.25 * M_PI * sin(t / 2.0 * 2 * M_PI) + 0.25 * M_PI;
  targets[2] = 0.5 * M_PI;
  targets[3] = -0.25 * M_PI * sin(t / 2.0 * 2 * M_PI) - 0.25 * M_PI;

  // Determine angular velocities
  double velocities[4];
  for (unsigned i = 0; i < 4; ++i)
    velocities[i] = 0.1 * (targets[i] - angles[i]);

  // Add joint movement actions to the Cerebellum
  cer.addAction(make_shared<MoveJointAction>(Types::LARM1, velocities[0]));
  cer.addAction(make_shared<MoveJointAction>(Types::LARM2, velocities[1]));
  cer.addAction(make_shared<MoveJointAction>(Types::RARM1, velocities[2]));
  cer.addAction(make_shared<MoveJointAction>(Types::RARM2, velocities[3]));

  // Tell the Cerebellum to send the actions to the simulator
  cer.outputCommands(SAgentSocketComm::getInstance());
}
```

Listing 3.4: `main.cc`

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

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 3.4. Now, configure and compile your agent, by running '`cmake . && make`' in your source directory, fire up RCSSServer3d and a monitor, run the '`myagent`' binary and wave back at your new friend!

4 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 4.1 will suffice. If you want to start quickly with a working agent, you can skip until section 4.8 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. 4.1. 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 managable 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,

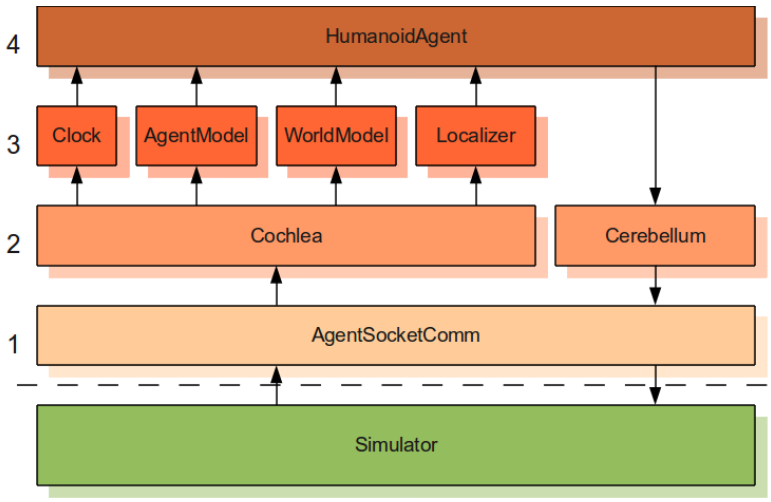


Figure 4.1: `libbats` modules

work out contradictions if necesary 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.

4.1 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` classes [1] 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());
}
```

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

4.2 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);
```

4.3 Clock

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

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

4.4 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();
```

4.5 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`:

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

(a) Agent (b) Local (c) World

Figure 4.2: The coordinate systems used in `libbats`: (a) Agent ('raw') coordinates, (b) local coordinates, and (c) global coordinates.

```
wm.update();

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

4.6 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. 4.2:

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.4.2(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.4.2(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.4.2(c).

4.7 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();

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.

4.8 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 3 to learn how to use this class to set up your own agent.

4.9 XML Configuration

The `libbats` library comes with an XML based configuration framework, to make it easier to set parameters at runtime without having to recompile, either globally, for a specific module, or for specific agents. The library comes with a minimal configuration file, at '`xml/conf.xml`' that is loaded by default, unless you pass a different file path to the `HumanoidAgent` constructor. You should base your own configuration file on this one, as it contains the minimally necessary items:

- A correct XML header, and a root 'conf' element that includes that XInclude namespace.
- The inclusion of the XML file describing the robot model.
- A 'player-class' element for each possible player class, where by default each class index corresponds to a uniform number.

This is sufficient configuration for `libbats` internally. The following sections describe which classes to use and how to format your XML file to use the configuration framework in your own code.

4.9.1 `Conf`

The direct interface to the configuration file is provided by the `Conf` class. It is again a singleton, to get a reference to it use:

```
Conf& conf = Conf::getInstance();
```

It is used to parse the XML file (which is done for you if you use the `HumanoidAgent` class), and gives direct access to it through XPath expressions. However, usually you want to use the higher level `getParam` methods on it to get configuration parameters. There are two variants, both templated:

`T getParam(string name, T def)` Use this to get global parameter values, that are defined in a 'parameters' element directly under the 'conf' root element. The first argument gives the name of the parameter to look for, the second gives a default value to return if the parameter is not found in the configuration file. Note that the type of this second argument also determines the type of the return value. Consider the following statements, given the the example configuration file in listing 4.1:

```
double foo = conf.getParam("foo", 2.0);
string bar = conf.getParam("bar", string("unknown"));
```

After this, `foo == 1.0`, as defined in the XML file, and `bar == "unknown"`, because there is no global parameter 'bar' set in configuration. Note that to read a string parameter you have to explicitly pass a `std::string` value as second argument.

Within the library a few parameters are read with this method. Currently these only consist of field dimensions, and whether the server is restarted for half time or not. To override the default values for these, use the following format:

```
<conf xmlns:xi="http://www.w3.org/2003/XInclude">
  <parameters>
    <fieldlength>30</fieldlength>
    <fieldwidth>20</fieldwidth>
    <goalwidth>2.1</goalwidth>
    <goalheight>0.8</goalheight>
    <penaltyxlength>1.8</penaltyxlength>
    <penaltyylength>3.9</penaltyylength>
    <numberofplayers>11</numberofplayers>
    <halftimerestart>1</halftimerestart> <!-- 1 = true, 0 = false -->
  </parameters>

  ...
```

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

`T getParam(unsigned playerClassIdx, string name, T def)` The second variant takes a player class index as an additional parameter, to fetch parameters specific to a certain player. Again given listing 4.1, consider the following statements:

```
string bar1 = conf.getParam(1, "bar", string("unknown"));
string bar2 = conf.getParam(2, "bar", string("unknown"));
string bar3 = conf.getParam(3, "bar", string("unknown"));
```

The values of `bar1`, `bar2`, and `bar3` will be "attacker", "defender", and "unknown", respectively.

What you use as an index is completely up to you, but a sensible choice is the uniform number, which you can get from the `AgentModel`. The library also provides

a simple framework to start you off with more dynamic player class selection, through the interface supplied by the `PCSelector` class. You can inherit from this class to create your own method of selecting the role of a player, then choose it when initializing the `PCSelector` singleton. Currently only one implementation is included in `libbats` , the `UnumPCSelector`, which simply returns the uniform number. It can be initialized and used as follows:

```
SPCSelector::initialize<UnumPCSelector>();
...
conf.getParam(SPCSelector::getInstance().getPlayerClass(), "foo", 2.0);
```

4.9.2 `Configurable`

Beyond direct access through the `Conf` class, you can make your own classes configurable by having them extend the.. `Configurable` class. When doing so, your class' constructor must pass two things to the constructor of `Configurable`: a tag name and an ID string. These are then used to find the value of parameters when calling the `getConfParam` method.

Listing 4.2 gives a usage example, where the parameter for the first instance is fetched from the configuration file, whereas the second resorts to the default value.

Listing 4.1: `conf.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<conf xmlns:xi="http://www.w3.org/2003/XInclude">
  <xi:include href="nao_mdl.xml"/>

  <parameters>
    <foo>1</foo>
  </parameters>

  <myclass id="inst1">
    <foo>10</foo>
  </myclass>

  <player-class index="0"/>

  <player-class index="1">
    <parameters>
      <bar>attacker</bar>
    </parameters>
  </player-class>

  <player-class index="2">
    <parameters>
      <bar>defender</bar>
    </parameters>
  </player-class>

  <player-class index="3"/>

  <player-class index="4"/>

  <player-class index="5"/>

  <player-class index="6"/>

  <player-class index="7"/>

  <player-class index="8"/>

  <player-class index="9"/>

  <player-class index="10"/>

  <player-class index="11"/>
</conf>
```

Listing 4.2: `Configurable` example

```
#include <libbats/Configurable/configurable.hh>

class MyClass : public bats::Configurable
{
public:
  MyClass(std::string const& id) : bats::Configurable("myclass", id)
  {
    d_foo = getConfParam("foo", 1.0);
  }

  double getFoo() const { return d_foo; }
private:
  double d_foo;
};
...
MyClass inst1("inst1");
MyClass inst2("inst2");

cout << inst1.getFoo() << endl; // Output: 10
cout << inst2.getFoo() << endl; // Output: 1
```

5 Movement

In Chap. 4 we showed how to move your agent through the `Cerebellum`, which you need to provide joint velocities. `libbats` provides an interface to create special modules to determine these velocities and create coordinated movements, in the form of the `JointController` class, with a few specific implementations. We will discuss these here.

5.1 `JointController`

The `JointController` class is a very basic interface that provides two-step semantics for generating angular velocities for joints: first, the controller is run by calling the virtual `run()` method, which should fill the `d_jointVelocities` vector. Secondly, this vector is requested through the `getJointVelocities()` method, which can then be used to build actions and feed these to the Cerebellum, as discussed before. So, specific implementations must implement the `run()` method, which then should be called at every think cycle. The `run()` method takes a pointer to an optional parameter object that is specific to the controller implementation. For its use, also have a look at the `DribbleAgent` example that comes with `libbats` . The next sections discuss the implementations already provided.

5.2 `MotionSequencePlayer`

The first implementation is used to play pre-determined, fixed motion sequences. Such a sequence is captured by an instance of the `MotionSequence` class, and consists of the following:

- A list of sequences for each joint. Such a joint sequence is defined by a list of key-frames, which are pairs of time stamps and angles, and as such prescribe the position of a joint at the given time. The target positions of the joints for time-steps in between key-frames are determined by linear interpolation of surrounding key-frames.

- A time length, determining the duration of the sequence.

One could construct such a sequence in code, by creating an instance of `MotionSequence`, fill its `jointSequences` and `length` members, and then pass it to a `MotionSequencePlayer`'s `setSequence` method. It is however also possible to define the sequence in the XML

configuration file, and have the `MotionSequencePlayer` load it from there (possible extending from `JontController`, which in turn inherits from `Configurable`). An example motion sequence in XML looks as such:

```
<!DOCTYPE conf SYSTEM "conf.dtd">
<conf xmlns:xi="http://www.w3.org/2003/XInclude">
  ...
  <motionsequenceplayer id="playerid">
    <sequence>
      &larm1;: 0 0, 1 90, 2 -90;
      &lleg4;: 0 0, 1.5 -90, 3.5 0;
      &end;: 4;
    </sequence>
    <issymmetric>1</issymmetric>
  </motionsequenceplayer>
  ...
</conf>
```

Note that the `notionsequenceplayer` is a direct child of the root `conf` tag, and that the "`conf.dtd`" definition is loaded, to provide the joint entities to make selecting the right joint easier. Each joint sequence is simply a comma-separated list of '`timestamp angle`' entries, where the time is in seconds and the angle is in degrees.

The `issymmetric` element is provided to make it easier to make laterally symmetric sequences: if set to 1, all sequences defined for joints on the left side are copied to the matching joint on the right side, transforming the angle values where needed to create symmetric movement.

A inal note: a sequence player has no run-time parameters, so its `run()` method can take a zero pointer. Again, for example sequences in XML and their usage, look at the DribbleAgent example, which uses such sequences to create getting-up behaviors.

5.3 `GaitGenerator`

One of the biggest challenges of creating a team of humanoid robot footballers is to get them to walk in a fast, stable and reliable manner. To help with this, `libbats` offer the `GaitGenerator` class, to build walking modules on. On its own it is just an extension of a JointController that takes a vector of doubles as run-time parameters.

More interesting is the gait generator implementation that comes with `libbats` : the `IKGaitGenerator`. Here, 'IK' stands for *Inverse Kinematics*, which basically indicates that the joint velocities are determined from a desired path of the feet in Euclidean space. In this case, this path is a semi-ellipse, as visualized in Fig. 5.1.

This figure show some of the parameters that determine the actual resulting path. These parameters can be configured in the XML configuration file; the full list and the XML element names to be used can be found in Tab. 5.1. Again, for a working example, look at the `DribbleAgent` example agent code, and it's XML configuration file.

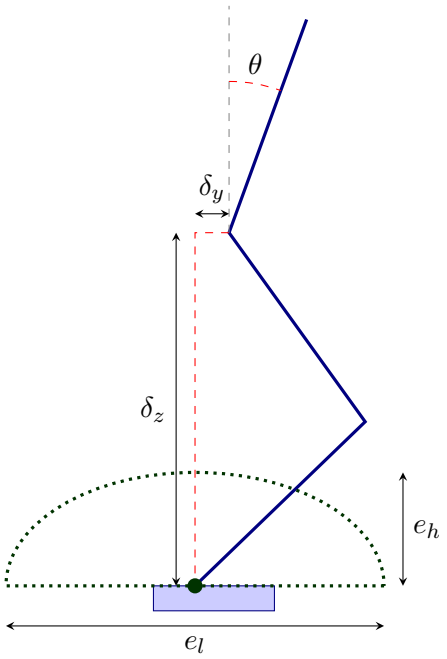


Figure 5.1: Visualization of the IK gait and its parameters. joint. The green dotted line shows the path that the feet follows (not that actually the path is defined in coordinates of the ankle joint). This path is characterized by the `ellipseheight` (e_h) and `ellipselength` (e_l) parameters. The position of the pathe relative to the agent's hip is determined by the `offsety` and `offsetz` parameters (δ_y and δ_z respectively). Finally, the `leanangle` (θ) parameter determines how far the torso should additionally lean forward.

Param	XML	Description
T	`period`	Time to complete 2 steps: one with each leg (seconds).
τ	`startuptime`	Time to reach full speed/step sizes from standstill (seconds).
e_l	`ellipselength`	Maximum ellipse length/step size (meters).
e_h	`ellipseheight`	Maximum ellipse/step height (meters).
e_w	`sidewidth`	Maximum sideways step size (meters).
d_y	`offsety`	Offset of ellipse center from hip along y-axis (meters).
d_z	`offsetz`	Offset of ellipse center from hip along z-axis (meters).
α_{max}	`maxturnangle`	Maximum angle for `LEG1` joint, used to turn (degrees).
θ	`leanangle`	Angle of torso relative to vertical (degrees).

Table 5.1: List of `IKGaitGenerator` configurable parameters

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 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 class, make sure it adhears to the following points:

- Your class must have a default constructor.
- Make all constructors, including the copy constructor, of your class private.
- Make the assignment 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;
```

6.2 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();
```

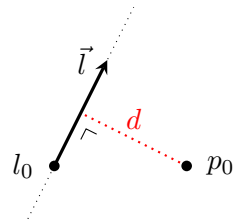
Other useful methods are `getSigma()`, to get the distribution's covariance matrix, and `draw()`, to draw a random value from the distribution:

```
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();
```

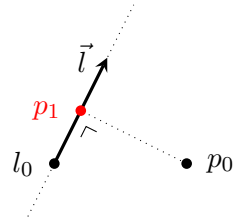
6.3 Math

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 selfexplenatory, for the rest see the following descriptions (and again the documentation in the code itself):

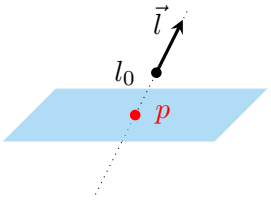
distanceLinePoint This method is used to calculate the distance between a line and a point. The line, dashed black in the example to the right, is defined by a point vector l_0 and a direction vector $\vec{l}$, the point p is also a point vector. The method returns the length of the red, dotted line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



6.4 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 list 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`:

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