B-Human

Team Report and Code Release 2015

Thomas Röfer[1,2], Tim Laue[2]
Jesse Richter-Klug[2], Maik Schünemann[2],Jonas Stiensmeier[2]
Andreas Stolpmann[2], Alexander Stöwing[2], Felix Thielke[2]

[1] Deutsches Forschungszentrum für Künstliche Intelligenz,
Enrique-Schmidt-Str. 5, 28359 Bremen, Germany
[2] Universität Bremen, Fachbereich 3, Postfach 330440, 28334 Bremen, Germany

Revision: March 9, 2016

Contents

Chapter 1

Introduction

1.1 About Us

B-Human is a joint RoboCup team of the University of Bremen and the German Research Center for Artificial Intelligence (DFKI). The team was founded in 2006 as a team in the Humanoid League, but switched to participating in the Standard Platform League in 2009. Since then, we participated in seven RoboCup German Open competitions and in seven RoboCups. We always won the German Open and became world champion four times.

This year, we came second by only losing the final to the world champion *UNSW Australia*. Beside this, we reached also the second place in the *Drop-In Competition*. In the *Technical Challenge* we became first by reaching the second place in each single contest, which were the *Corner Kicks Challenge*, the *Many Carpets Challenge*, and the *Realistic Ball Challenge*.

The current team consists of the following persons:

Team Leaders / Staff: Tim Laue, Thomas Röfer.

Students: Arne Böckmann, Patrick Glaser, Vitali Gutsch, Florian Maaß, Jesse Richter-Klug, Maik Schünemann, Jonas Stiensmeier, Alexander Stöwing, Andreas Stolpmann, Felix Thielke, Alexis Tsogias, Jan-Bernd Vosteen

Associated Researchers: Udo Frese, Judith Müller, Dennis Schüthe, Felix Wenk

1.2 About this Document

In this document, we give a short overview of the changes that we made in our system since last year and provide descriptions of the approaches used in the technical challenges. The most comprehensive reference to our system is still the team report of 2013 [12].

Chapter 2 gives a short introduction on how to build our 2015 code, including the software required to do so, and how to run the NAO with our system. In Chapter 3, we summarize the changes we made since 2014 that are not described elsewhere in this document. Chapter 4 describes our contributions to this year's technical challenges. Chapter 5 gives an overview of the Team Communication Monitor. Finally, Chapter 6 lists the software developed by others that we include in our code release as well as the people and organizations that sponsored us.

Figure 1.1: The team members of B-Human 2015

Chapter 2

Getting Started

The goal of this chapter is to give an overview of the code release package and to give instructions on how to enliven a NAO with our code. For the latter, several steps are necessary: downloading the source code, compiling the code using Visual Studio, Xcode, or make on Linux, setting up the NAO, copying the files to the robot and starting the software. In addition, all calibration procedures are described here.

2.1 Download

The code release can be downloaded from GitHub at `https://github.com/bhuman`. Store the code release in a folder of your liking. After the download is finished, the chosen folder should contain several subdirectories which are described below.

Build is the target directory for generated binaries and for temporary files created during the compilation of the source code. It is initially missing and will be created by the build system.

Config contains configuration files used to configure the B-Human software. A brief overview of the organization of the configuration files can be found in Sect. 2.9.

Install contains all files needed to set up B-Human on a NAO.

Make Contains Makefiles, other files needed to compile the code, the *Copyfiles* tool, and a script to download log files from a NAO. In addition there are *generate* scripts that create the project files for Xcode, Visual Studio, CodeLite, and NetBeans.

Src contains the source code of the B-Human software including the B-Human User Shell [12, Chapter 8.2]

Util contains auxiliary and third party libraries (cf. Sect. 6) as well as our tools, e. g. SimRobot [12, Chapter 8.1]

2.2 Components and Configurations

The B-Human software is usable on Windows, Linux, and OS X. It consists of two shared libraries for NAOqi running on the real robot, an additional executable for the robot, the same software running in our simulator SimRobot (without NAOqi), as well as some libraries and tools. Therefore, the software is separated into the following components:

bush is a tool to deploy and manage multiple robots at the same time [12, Chapter 8.2].

Controller is a static library that contains NAO-specific extensions of the simulator, the interface to the robot code framework, and it is also required for controlling and high level debugging of code that runs on a NAO.

copyfiles is a tool for copying compiled code to the robot. For a more detailed explanation see Sect. 2.5. In the Xcode project, this is called *Deploy*.

libbhuman is the shared library used by the B-Human executable to interact with NAOqi.

libgamectrl is a shared NAOqi library that communicates with the GameController. Additionally, it implements the official button interface and sets the LEDs as specified in the rules. More information can be found in our 2013 code release [12, Chapter 3.1].

libqxt is a static library that provides an additional widget for Qt on Windows and Linux. On OS X, the same source files are simply part of the library *Controller*.

Nao is the B-Human executable for the NAO. It depends on *SpecialActions*, *libbhuman*, and *libgamectrl*.

qtpropertybrowser is a static library that implements a property browser in Qt.

SimRobot is the simulator executable for running and controlling the B-Human robot code. It dynamically links against the components *SimRobotCore2*, *SimRobotEditor*, *SimRobotHelp*, *SimulatedNao*, and some third-party libraries. It also depends on the component *SpecialActions*, the result of which is loaded by the robot code. SimRobot is compilable in *Release*, *Develop*, and *Debug* configurations. All these configurations contain debug code, but *Release* performs some optimizations and strips debug symbols (Linux and OS X). *Develop* produces debuggable robot code while linking against non-debuggable but faster *Release* libraries.

SimRobotCore2 is a shared library that contains the simulation engine of SimRobot.

SimRobotEditor is a shared library that contains the editor widget of the simulator.

SimRobotHelp is a shared library that contains the help widget of the simulator.

SimulatedNao is a shared library containing the B-Human code for the simulator. It depends on *Controller*, *qtpropertybrowser*, and *libqxt*. It is statically linked against them.

All components can be built in the three configurations *Release*, *Develop*, and *Debug*. *Release* is meant for "game code" and thus enables the highest optimizations; *Debug* provides full debugging support and no optimization. *Develop* is a special case. It generates executables with some debugging support for the components *Nao* and *SimulatedNao* (see the table below for more specific information). For all other components it is identical to *Release*.

The different configurations for *Nao* and *SimulatedNao* can be looked up here:

	without assertions (NDEBUG)	debug symbols (compiler flags)	debug libs[1] (_DEBUG, compiler flags)	optimizations (compiler flags)
Release				
Nao	✓	×	×	✓
SimulatedNao	✓	×	×	✓
Develop				
Nao	×	×	×	✓
SimulatedNao	×	✓	×	×
Debug				
Nao	×	✓	✓	×
SimulatedNao	×	✓	✓	×

[1] - on Windows - `http://msdn.microsoft.com/en-us/library/0b98s6w8(v=vs.140).aspx`

2.3 Building the Code

2.3.1 Project Generation

The scripts *generate* (or *generate.cmd* on Windows) in the *Make/<OS/IDE>* directories generate the platform or IDE specific files that are needed to compile the components. The script collects all the source files, headers, and other resources if needed and packs them into a solution matching your system (i. e. Visual Studio projects and a solution file for Windows, a CodeLite project for Linux, and an Xcode project for OS X). It has to be called before any IDE can be opened or any build process can be started and it has to be called again whenever files are added or removed from the project. On Linux, the *generate* script is needed when working with CodeLite or NetBeans. Building the code from the command line, via the provided Makefile, works without calling *generate* on Linux.

2.3.2 Visual Studio on Windows

2.3.2.1 Required Software

- Windows 7 64 bit or later
- Visual Studio 2015[1] or later
- Cygwin x86 / x64 (available at `http://www.cygwin.com`) with the additional packages *rsync*, *openssh*, *ccache*, and *clang*. Let the installer add an icon to the start menu (the *Cygwin Terminal*). Add the ...\cygwin64\bin directory to the beginning of the PATH environment variable (before the Windows system directory, since there are some commands that have the same names but work differently). Make sure to start the *Cygwin Terminal* at least once, since it will create a home directory.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the NAOqi *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-linux32.tar.gz*),

[1] Visual Studio 2015 Community Edition with only the "Common Tools for Visual C++ 2015" installed is sufficient.

the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `https://community.aldebaran.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

2.3.2.2 Compiling

Generate the Visual Studio project files using the script *Make/VS2015/generate.cmd* and open the solution *Make/VS2015/B-Human.sln* in Visual Studio. Visual Studio will then list all the components (cf. Sect. 2.2) of the software in the "Solution Explorer". Select the desired configuration (cf. Sect. 2.2, *Develop* would be a good choice for starters) and build the desired project: *SimRobot* compiles every project used by the simulator, *Nao* compiles every project used for working with a real NAO, and *Utils/bush* compiles the B-Human User Shell (cf. [12, Chapter 8.2]). You may select *SimRobot* or *Utils/bush* as "StartUp Project".

2.3.3 Xcode on OS X

2.3.3.1 Required Software

The following components are required:

- OS X 10.11 or later
- Xcode 7.1 or later
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the NAOqi *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-linux32.tar.gz*), the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `https://community.aldebaran.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot. Also note that *installAlcommon* expects the extension *.tar.gz*. If the NAOqi archive was partially unpacked after the download, e. g., by Safari, repack it again before executing the script.

2.3.3.2 Compiling

Generate the Xcode project by executing *Make/OSX/generate*. Open the Xcode project *Make/OSX/B-Human.xcodeproj*. A number of schemes (selectable in the toolbar) allow building SimRobot in the configurations *Debug*, *Develop*, and *Release*, as well as the code for the NAO[2] in all three configurations (cf. Sect. 2.2). For both targets, *Develop* is a good choice. In addition, the B-Human User Shell *bush* can be built. It is advisable to delete all the schemes that are automatically created by Xcode, i. e. all non-shared ones.

When building for the NAO, a successful build will open a dialog to deploy the code to a robot (using the *copyfiles* script, cf. Sect. 2.5).[3] If the *login* script was used before to login to a NAO, the IP address used will be provided as default. In addition, the option `-r` is provided by default, which will restart the software on the NAO after it was deployed. Both the IP address selected

[2]Note that the cross compiler actually builds code for Linux, although the scheme says "My Mac".

[3]Before you can do that, you have to setup the NAO first (cf. Sect. 2.4).

and the options specified are remembered for the next use of the deploy dialog. The IP address is stored in the file *Config/Scenes/connect.con* that is also written by the *login* script and used by the *RemoteRobot* simulator scene. The options are stored in *Make/OSX/copyfiles-options.txt*. A special option is `-a`: If it is specified, the deploy dialog is not shown anymore in the future. Instead, the previous settings will be reused, i. e. building the code will automatically deploy it without any questions asked. To get the dialog back, hold down the shift key at the time the dialog would normally appear.

2.3.4 Linux

The following has been tested and works on Ubuntu 14.04 64-bit. It should also work on other Linux distributions (as long as they are 64-bit); however, different or additional packages may be needed.

2.3.4.1 Required Software

Requirements (listed by common package names) for Ubuntu 14.04:

- clang $\geq$ 3.4
- qt4-dev-tools
- libglew-dev
- libxml2-dev
- graphviz – Optional, for generating module graphs and the behavior graph.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the NAOqi *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-linux32.tar.gz*), the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `https://community.aldebaran.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

On Ubuntu 14.04, you can execute the following command to install all requirements except for *alcommon*:

```
sudo apt-get install qt4-dev-tools libglew-dev libxml2-dev clang graphviz
```

2.3.4.2 Compiling

To compile one of the components described in Section 2.2 (except *Copyfiles*), simply select *Make/Linux* as the current working directory and type:

```
make <component> [CONFIG=<configuration>]
```

To clean up the whole solution, use:

```
make clean [CONFIG=<configuration>]
```

As an alternative, there is also support for the integrated development environments NetBeans and CodeLite that work similar to Visual Studio for Windows (cf. Sect. 2.3.2.2).

To use CodeLite, execute *Make/LinuxCodeLite/generate* and open the *B-Human.workspace* afterwards. Note that CodeLite 5 or later is required to open the workspace generated. Older versions might crash. For the NetBeans project files execute *Make/NetBeans/generate*.

2.4 Setting Up the NAO

2.4.1 Requirements

First of all, download the the atom system image, e. g. version 2.14 (*opennao-atom-system-image-2.1.4.13.opn*), and the *Flasher*, e. g. version 2.1.0, for your operating system from the download area of `https://community.aldebaran.com` (account required). In order to flash the robot, you need a USB flash drive having at least 2 GB space and a network cable.

To use the scripts in the directory *Install*, the following tools are required[4]:

sed, *rsync*.

Each script will check its requirements and will terminate with an error message if a required tool is not found.

The commands in this chapter are shell commands. They should be executed inside a Unix shell. On Windows, you *must* use the *Cygwin Terminal* to execute the commands. All shell commands should be executed from the *Install* directory.

2.4.2 Creating robot configuration files for a NAO

Before you start to set up the NAO, you need to create configuration files for each robot you want to set up. To create the configuration files, run *createRobot* followed by *addRobotIds* in the *Install* directory. The first script expects a team id, a robot id and a robot name. The team id is usually equal to your team number configured in *Config/settings.cfg*, but you can use any number between 1 and 254. The given team id is used as third part of the IPv4 address of the robot on both interfaces LAN and WLAN. All robots playing in the same team need the same team id to be able to communicate with each other. The robot id is the last part of the IP address and must be unique for each team id. The robot name identifies the robot and is used in the system to load robot specific configurations. Furthermore, it is used as the host name of the NAO operating system. The second file creates a table associating the *headId* and *bodyId* of each NAO to the name used by *createRobot*. These ids are the serial-numbers Aldebaran Robotics uses for the NAO. Apart from the name this script expects either those ids, typed in manually, or the current ip-address of the NAO, in which case the ids will be loaded from the robot.

Before creating your first robot configuration, check whether the network configuration template files *wireless* and *wired* in *Install/Network* and *default* from *Install/Network/Profiles* match the requirements of your local network configuration.

Here is an example for creating a new set of configuration files for a robot named Penny in team three, which will get the IP xxx.xxx.3.25:

```
cd Install
```

[4]In the unlikely case that they are missing in a Linux distribution, execute *sudo apt-get install sed scp*. On Windows and OS X, they are already installed at this point.

```
./createRobot -t 3 -r 25 Penny
./addRobotIds -ids ALDxxxxxxxxxxxx ALDxxxxxxxxxxxx Penny
```

If the robot is already running (cf. Sect. 2.4.4) the script can use its ip (e. g. 169.254.54.28) to extract the ids from the robot directly:

```
./addRobotIds -ip 169.254.54.28 Penny
```

Help for both scripts is available using the option *-h*. Running *createRobot* creates all needed files to install the robot. This script also creates a directory with the robot's name in *Config/Robots*. *addRobotIds* will store the table in *Config/Robots/robots.cfg*.

Note: When upgrading from an older B-Human code release running *createRobot* is not necessary. Nevertheless, the script *addRobotIds* has to be executed!

2.4.3 Managing Wireless Configurations

All wireless configurations are stored in *Install/Network/Profiles*. Additional configurations must be placed here and will be installed alongside the *default* configuration. After the setup is completed, the NAO will always load the *default* configuration, when booting the operating system.

You can later switch between different configurations by calling the script *setprofile* on the NAO, which overwrites the *default* configuration.

```
setprofile SPL_A
setprofile Home
```

Another way to switch between different configurations is by using the tools *copyfiles* (cf. Sect. 2.5) or *bush* (cf. [12, Chapter 8.2]).

2.4.4 Setup

After the robot specific configuration files were created (cf. Sect. 2.4.2 and Sect. 2.4.3), plug in your USB flash drive and start the *NAO flasher tool*[5]. Select the *opennao-atom-system-image-2.1.4.13.opn* and your USB flash drive. Enable "Factory reset" and click on the write button.

After the USB flash drive has been flashed, plug it into the NAO and press the chest button for about 5 seconds. Afterwards, the NAO will automatically install NAO OS and reboot. While installing the basic operating system, connect your computer to the robot using the network cable and configure your network for DHCP. Once the reboot is finished, the NAO will do its usual Aldebaran wake up procedure. Now the NAO will say its current IP address by pressing the chest button.

Afterwards the script *installRobot* has to be executed in oder to prepare the robot for the B-Human software. This script only expects the current IP address of the robot. For example run:

```
./installRobot 169.254.54.28
```

Now you can use *copyfiles* (cf. Sect. 2.5) or *bush* (cf. [12, Chapter 8.2]) to copy compiled code and configuration files to the NAO.

[5]On Linux and OS X you have to start the flasher with root permissions. Usually you can do this with sudo ./flasher

2.5 Copying the Compiled Code

The tool *copyfiles* is used to copy compiled code and configuration files to the NAO. Although *copyfiles* allows specifying the team number, it is usually better to configure the team number and the UDP port used for team communication permanently in the file *Config/settings.cfg*.

On Windows as well as on OS X, you can use your IDE to use *copyfiles*. In Visual Studio, you can run the script by "building" the tool *copyfiles*, which can be built in all configurations. If the code is not up-to-date in the desired configuration, it will be built. After a successful build, you will be prompted to enter the parameters described below. On the Mac, a successful build for the NAO always ends with a dialog asking for *copyfiles*' command line options.

You can also execute the script at the command prompt, which is the only option for Linux users. The script is located in the folder *Make/<OS/IDE>*.

copyfiles requires two mandatory parameters. First, the configuration the code was compiled with (*Debug*, *Develop* or *Release*)[6], and second, the IP address of the robot. To adjust the desired settings, it is possible to set the following optional parameters:

Option	Description
-l <location>	Sets the location, replacing the value in the *settings.cfg*.
-t <color>	Sets the team color to *blue* or *red*, replacing the value in the *settings.cfg*.
-p <number>	Sets the player number, replacing the value in the *settings.cfg*.
-n <number>	Sets team number, replacing the value in the *settings.cfg*.
-o <port>	Overwrite team port (default is 10000 + team number).
-r	Restarts *bhuman* after copying.
-rr	Forces restart of *bhuman* and *naoqi*.
-m n <ip>	Copies to IP address <ip> and sets the player number to *n*. This option can be specified more than ones to deploy to multiple robots.
-wc	Compiles also under Windows if the binaries are outdated.
-nc	Never compiles, even if binaries are outdated. Default on Windows/OS X.
-nr	Do not check whether robot is reachable. Otherwise, ping it once.
-d	Removes all log files from the robot's /home/nao/logs directory before copying files.
-v <percent>	Set NAO's sound volume.
-w <profile>	Set wireless profile.
-h | --help	Prints the help.

Possible calls could be:

```
./copyfiles Develop 134.102.204.229 -n 5 -t blue -p 3 -r
./copyfiles Release -m 1 10.0.0.1 -m 3 10.0.0.2
```

The destination directory on the robot is */home/nao/Config*. Alternatively, the B-Human User Shell (cf. [12, Chapter 8.2]) can be used to copy the compiled code to several robots at once.

2.6 Working with the NAO

After pressing the chest button, it takes about 40 seconds until NAOqi is started. Currently, the B-Human software consists of two shared libraries (*libbhuman.so* and *libgamectrl.so*) that are loaded by NAOqi at startup, and one executable (*bhuman*), which is also loaded at startup.

[6]This parameter is automatically passed to the script when using IDE-based deployment.

To connect to the NAO, the subdirectories of *Make* contain a *login* script for each supported platform. The only parameter of that script is the IP address of the robot to login. It automatically uses the appropriate SSH key to login. In addition, the IP address specified is written to the file *Config/Scenes/connect.con*. Thus, a later use of the SimRobot scene *RemoteRobot.ros2* will automatically connect to the same robot. On OS X, the IP address is also the default address for deployment in Xcode.

Additionally, the script *Make/Linux/ssh-config* can be used to output a valid ssh *config* file containing all robots currently present in the robots folder. Using this configuration file, one can connect to a robot using its name instead of the IP address.

There are several scripts to start and stop NAOqi and *bhuman* via SSH. Those scripts are copied to the NAO upon installing the B-Human software.

naoqi executes NAOqi in the foreground. Press *Ctrl+C* to terminate the process. Please note that the process will automatically be terminated if the SSH connection is closed.

nao start|stop|restart starts, stops or restarts NAOqi. In case *libbhuman* or *libgamectrl* were updated, *copyfiles* restarts NAOqi automatically.

bhuman executes the *bhuman* executable in the foreground. Press *Ctrl+C* to terminate the process. Please note that the process will automatically be terminated if the SSH connection is closed.

bhumand start|stop|restart starts, stops or restarts the *bhuman* executable. *Copyfiles* always stops *bhuman* before deploying. If *copyfiles* is started with option *-r*, it will restart *bhuman* after all files were copied.

status shows the status of NAOqi and *bhuman*.

stop stops running instances of NAOqi and *bhuman*.

halt shuts down the NAO. If NAOqi is running, this can also be done by pressing the chest button longer than three seconds.

reboot reboots the NAO.

2.7 Starting SimRobot

On Windows and OS X, SimRobot can either be started from the development environment or by starting a scene description file in *Config/Scenes*[7]. In the first case, a scene description file has to be opened manually, whereas it will already be loaded in the latter case. On Linux, just run *Build/SimRobot/Linux/<configuration>/SimRobot*, either from the shell or from your favorite file browser, and load a scene description file afterwards. When a simulation is opened for the first time, only the scene graph is displayed. The simulation is already running, which can be noted from the increasing number of simulation steps shown in the status bar. A scene view showing the soccer field can be opened by double-clicking *RoboCup*. The view can be adjusted by using the context menu of the window or the toolbar. Double-clicking *Console* will open a window that shows the output of the robot code and that allows entering commands. All windows can be docked in the main window.

[7]On Windows, the first time starting such a file the *SimRobot.exe* must be manually chosen to open these files. Note that both on Windows and OS X, starting a scene description file bears the risk of executing a different version of SimRobot than the one that was just compiled.

After starting a simulation, a script file may automatically be executed, setting up the robot(s) as desired. The name of the script file is the same as the name of the scene description file but with the extension *.con*. Together with the ability of SimRobot to store the window layout, the software can be configured to always start with a setup suitable for a certain task.

Although any object in the scene graph can be opened, only displaying certain entries in the object tree makes sense, namely the main scene *RoboCup*, the objects in the group *RoboCup/robots*, and all other views.

To connect to a real NAO, open the RemoteRobot scene *Config/Scenes/RemoteRobot.ros2*. You will be prompted to enter the NAO's IP address.[8] In a remote connection, the simulation scene is usually empty. Therefore, it is not necessary to open a scene view.

2.8 Calibrating the Robots

Correctly calibrated robots are very important since the software requires all parts of the NAO to be at the expected locations. Otherwise, the NAO will not be able to walk stable and projections from image coordinates to world coordinates (and vice versa) will be wrong. In general, a lot of calculations will be unreliable. Two physical components of the NAO can be calibrated via SimRobot; the joints (cf. Sect. 2.8.2) and the cameras (cf. Sect. 2.8.3). Checking those calibrations from time to time is important, especially for the joints. New robots come with calibrated joints and are theoretically ready to play out of the box. However, over time and usage, the joints wear out. This is especially noticeable with the hip joint.

In addition to that, the B-Human vision software uses seven color classes which have to be calibrated, too (cf. Sect. 2.8.4). Changing locations or light conditions might require them to be adjusted.

2.8.1 Overall Physical Calibration

The physical calibration process can be split into three steps with the overall goal of an upright and straight standing robot, and a correctly calibrated camera. The first step is to get both feet in a planar position. This does not mean that the robot has to stand straight. It is done by lifting the robot up so that the bottom of the feet can be seen. The joint offsets of feet and legs are then changed until both feet are planar and the legs are parallel to one another. The distance between the two legs can be measured at the gray parts of the legs. They should be 10 cm apart from center to center.

The second step is the camera calibration (cf. Sect. 2.8.3). This step also measures the tilt of the body with respect to the feet. This measurement can then be used in the third step to improve the joint calibration and straighten the robot up (cf. Sect. 2.8.2). In some cases it may be necessary to repeat these steps.

2.8.2 Joint Calibration

The software supports two methods for calibrating the joints; either by manually adjusting offsets for each joint, or by using the JointCalibrator module which uses an inverse kinematic to do the same [12, Chapter 6.2.1.2]. The third step of the overall calibration process (cf. Sect. 2.8.1) can only be done via the JointCalibrator. When switching between these two methods, it is necessary

[8]The script might instead automatically connect to the IP address that was last used for login or deployment.

to save the *JointCalibration*, redeploy the NAO and restart bhuman. Otherwise, the changes done previously will not be used.

Before changing joint offsets, the robot has to be set in a standing position with fixed joint angles. Otherwise, the balancing mechanism of the motion engine might move the legs, messing up the joint calibrations. This can be done with

```
get representation:MotionRequest
```

and then set *motion* = *stand* in the returned statement.

When the calibration is finished it should be saved:

```
save representation:JointCalibration
```

Manually Adjusting Joint Offsets

First of all, the robot has to be switched to a stationary stand, otherwise the balancing mechanism of the motion engine might move the legs, messing up the joint calibration:

```
mr StandOutput CalibrationStand
```

There are two ways to adjust the joint offsets. Either by requesting the *JointCalibration* representation with a *get* call:

```
get representation:JointCalibration
```

modifying the calibration returned and then setting it. Or by using a Data View [12, Chapter 8.1.4.5]

```
vd representation:JointCalibration
```

which is more comfortable.

The *JointCalibration* also contains other information for each joint that should not be changed!

Using the JointCalibrator

First set the JointCalibrator to provide the *JointCalibration* and switch to the CalibrationStand:

```
call JointCalibrator
```

When a completely new calibration is desired, the *JointCalibration* can be reset:

```
dr module:JointCalibrator:reset
```

Afterwards, the translation and rotation of the feet can be modified. Again either with

```
get module:JointCalibrator:offsets
```

or with:

```
vd module:JointCalibrator:offsets
```

The units of the translations are in millimeters and the rotations are in degrees.

Straightening Up the NAO

The camera calibration (cf. Sect. 2.8.3) also calculates a rotation for the body rotation. These values can be passed to the JointCalibrator that will then set the NAO in an upright position. Call:

```
get representation:CameraCalibration
call JointCalibrator
```

Copy the values of *bodyRotationCorrection* (representation *CameraCalibration*) into *bodyRotation* (representation *JointCalibration*). Afterwards, set *bodyRotationCorrection* (representation *CameraCalibration*) to zero. Another way to make these actions more or less automatically is possible by using the AutomaticCameraCalibrator with the automation flag (cf. Sect. 3.2.1).

The last step is to adjust the translation of both feet at the same time (and most times in the same direction) so they are perpendicular positioned below the torso. A plummet or line laser is very useful for that task.

When all is done save the representations by executing

```
save representation:JointCalibration
save representation:CameraCalibration
```

Then redeploy the NAO and restart bhuman.

2.8.3 Camera Calibration

There are two methods to calibrate the camera [12, Chapter 4.1.2.1] of the robots. Both are explained within the following paragraphs.

Manual Calibration with the **CameraCalibrator**

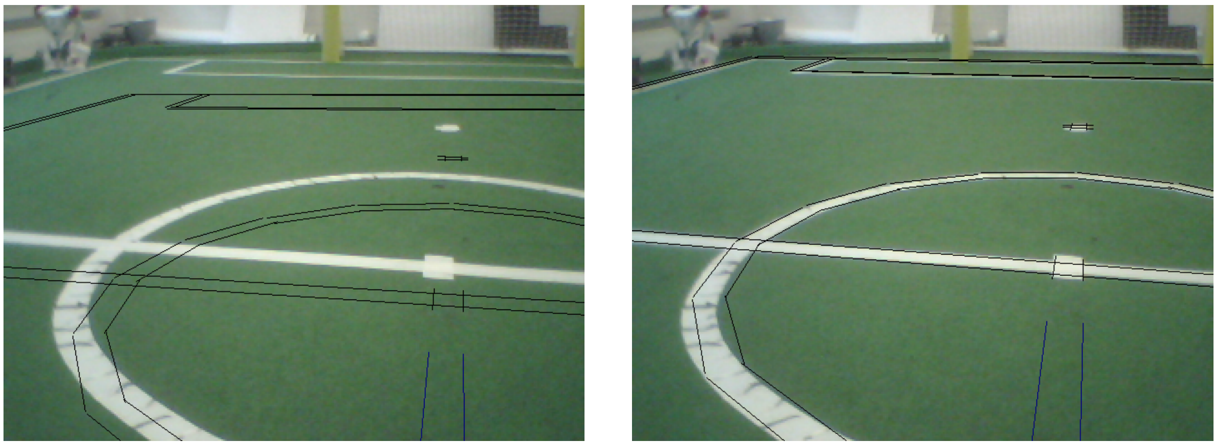

(a) (b)

Figure 2.1: Projected lines before (a) and after (b) the calibration procedure

For manually calibrating the cameras using the module CameraCalibrator, follow the steps below:

1. Connect the simulator to a robot on the field and place it on a defined spot (e. g. the penalty mark).

2. Run the SimRobot configuration file *CameraCalibrator.con*, i. e. type *call CameraCalibrators/Default* in SimRobot's console. This will initialize the calibration process and furthermore print some help and a command to the simulator console that will be needed later on.

3. Announce the robot's position on the field [12, Chapter 4.1.2] using the `CameraCalibrator` module (e. g. for setting the robot's position to the penalty mark of a field, type *set module:CameraCalibrator:robotPose rotation = 0; translation = {x = -3200; y = 0;};* in the console).

4. Start collecting points. Move the head to collect points for different rotations of the head by clicking on field lines. The number of collected points is shown in the top area of the image view. The head can be moved by clicking into an image view while holding the shift key down.[9] The head will then look at that position.

5. Run the automatic calibration process by pressing *Shift+Ctrl+O*[10] and wait until the optimization has converged. You can always go back to collecting more points by pressing *Shift+Ctrl+C*.

The calibration module allows to arbitrarily switch between upper and lower camera during the point collection phase. Both cameras should be considered for a good result. For the purpose of manual refinement of the robot-specific parameters mentioned, there is a debug drawing that projects the field lines into the camera image. To activate this drawing, type *vid raw module:-CameraMatrixProvider:calibrationHelper* in the simulator console. This drawing is helpful for calibrating, because the real lines and the projected lines only match if the camera matrix and hence the camera calibration is correct (assuming that the real position corresponds to the self-localization of the robot). Modify the parameters of *CameraCalibration* so that the projected lines match the field lines in the image (see Fig. 2.1b for a desirable calibration).

Automatic Calibration with the AutomaticCameraCalibrator

For an automatic camera calibration using the module AutomaticCameraCalibrator (cf. Sect. 3.2.1), follow the steps below:

1.-3. Same process as for the CameraCalibrator. But instead of typing *CameraCalibrators/Default* use *CameraCalibrators/Automatic*.

4. To automatically generate the commands for the following joint calibration to correct the body rotation, you can set a flag via *set module:AutomaticCameraCalibrator:setJointOffsets true*. After you finished the optimization you can just enter the generated commands and thereby correct the rotation.

5. To start the point collection use the command *dr module:AutomaticCameraCalibrator:start* and wait for the output "Accumulation finished. Waiting to optimize...". The process includes both, the upper and lower camera.

6. If you are unhappy with the collection of some specific samples you are now able to delete samples by left-clicking onto the sample in the image in which it has been found. If there are some samples missing you can manually add them by *Ctrl* + left-clicking into the correspondent image.

[9]Note that this will actually change the module that controls the head motion.
[10]As always, press *Cmd* instead of *Ctrl* on a Mac.

7. Run the automatic calibration process using *dr module:AutomaticCameraCalibrator:-optimize* and wait until the optimization has converged.

2.8.4 Color Calibration

Calibrating the color classes is split into two steps. First of all, the parameters of the camera driver must be updated to the environment's needs. The commands:

```
get representation:UpperCameraSettings
get representation:LowerCameraSettings
```

will return the current settings. Furthermore, the necessary *set* command will be generated. The most important parameters are:

whiteBalance: The white balance used. The available interval is [2700, 6500].

exposure: The exposure used. The available interval is [0, 1000]. Usually, an exposure of 140 is used, which equals 14 ms. Be aware that high exposures lead to blurred images.

gain: The gain used. The available interval is [0, 255]. Usually, the gain is set to 50 - 70. Be aware that high gain values lead to noisy images.

autoWhiteBalance: Enable(1) / disable(0) the automatism for white balance. This parameter should always be disabled since a change in the white balance can change the color and mess up the color calibration. On the other hand, a real change in the color temperature of the environment will have the same result.

autoExposure: Enable (1) / disable (0) the automatism for exposure. This parameter should always be disabled, since the automation will choose higher values than necessary, which will result in blurry images.

The camera driver can do a one-time auto white balance. This feature can be triggered with the commands:

```
dr module:CameraProvider:DoWhiteBalanceUpper
dr module:CameraProvider:DoWhiteBalanceLower
```

After setting up the parameters of the camera driver, the parameters of the color classes must be updated (cf. [12, Chapter 4.1.4]). To do so, one needs to open the views with the upper and lower camera images and the color calibration view. See [12, Chapter 8.1.4.1] for a detailed description. Unlike stated there, one has to select the color displayed/calibrated for each view separately now. It might be useful to first update the green calibration since some perceptors, e. g. the BallPerceptor, use the *FieldBoundary*, which relies on proper green classification. After finishing the color class calibration and saving the current parameters, copyfiles/bush (cf. Sect. 2.5) can be used to deploy the current settings. Ensure the updated files *upperCameraSettings.cfg*, *lowerCameraSettings.cfg* and *colorCalibration.cfg* are stored in the correct location.

2.9 Configuration Files

Since the recompilation of the code takes a lot of time in some cases and each robot needs a different configuration, the software uses a huge amount of configuration files which can be

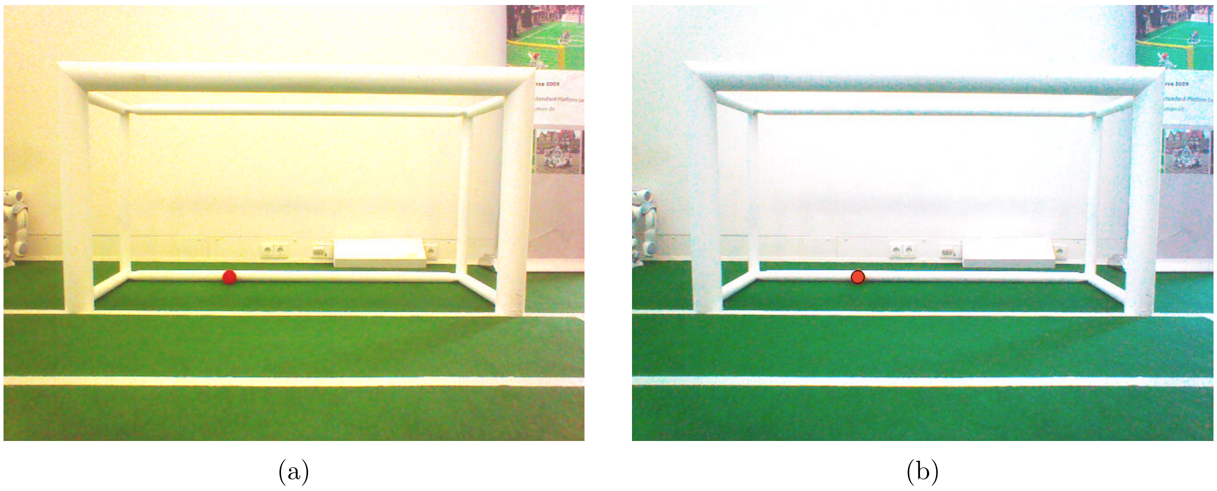


Figure 2.2: a) An image with improper white balance. b) The same image with better settings for white balance.

altered without causing recompilation. All the files that are used by the software[11] are located below the directory *Config*.

Locations can be used to configure the software for different independent tasks. They can be set up by simply creating a new folder with the desired name within *Config/Locations* and placing configuration files in it. Those configuration files are only taken into account if the location is activated in the file *Config/settings.cfg*.

Besides the global configuration files, there are some files which depend on the robot's head, body, or both. To differentiate the locations of these files, the names of the head and the body of each robot is used. The <*head name*> depends on the IP address of the robot which in turn belongs to the robot's head. The <*body name*> is stored in the chestboard of the nao and thus depends on the body. In the Simulator, both names are always "Nao".

To handle all these different configuration files, there are fall-back rules that are applied if a requested configuration file is not found. The search sequence for a configuration file is:

1. *Config/Robots/<head name>/Head/<filename>*
   - Used for files that only depend on the robot's **head**
   - e.g.: *Robots/Amy/Head/cameraIntrinsics.cfg*
2. *Config/Robots/<body name>/Body/<filename>*
   - Used for files that only depend on the robot's **body**
   - e.g.: *Robots/Alex/Body/walkingEngine.cfg*
3. *Config/Robots/<head name>/<body name>/<filename>*
   - Used for files that depend on both, the robot's head **and** body.
   - e.g.: *"Robots/Amy/Alex/cameraCalibration.cfg"*
4. *Config/Robots/<head name>/<filename>*

[11]There are also some configuration files for the operating system of the robots that are located in the directory *Install*.

- If the head and body constellations for your robots are always the same, all files which belong to a robot may also be saved here.

5. *Config/Robots/Default/<filename>*

6. *Config/Locations/<current location>/<filename>*

7. *Config/Locations/Default/<filename>*

8. *Config/<filename>*

So, whether a configuration file is robot-dependent or location-dependent or should always be available to the software is just a matter of moving it between the directories specified above. This allows for a maximum of flexibility. Directories that are searched earlier might contain specialized versions of configuration files. Directories that are searched later can provide fallback versions of these configuration files that are used if no specialization exists.

Using configuration files within our software requires very little effort because loading them is completely transparent for a developer when using parametrized modules (cf. [12, Chapter 3.3.5], but note that the syntax has changed).

Chapter 3

Changes Since 2014

In this chapter, we describe changes made to our system that are not directly related to the subsequently described Technical Challenges.

3.1 Infrastructure

3.1.1 Modules and Representations

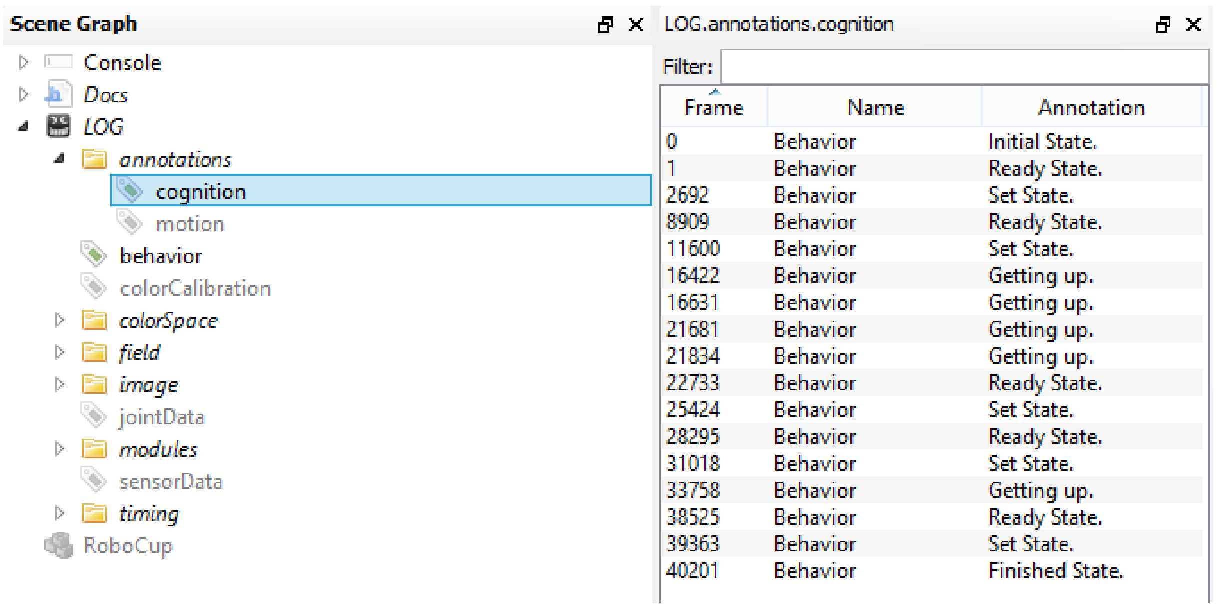
The macro `MODULE` allows specifying the interface of a module, i. e. which representations it requires and which it provides. There have been a lot of variants of the macro `PROVIDES` that defines the latter in the past. Now this has been reduced to only two variants, one of which is rarely used. Previously, `_WITH_OUTPUT` had to be added to indicate that the representation provided could also be logged. This is now determined automatically by searching whether one of the constants defined in the enumeration `MessageID` matches the name of the representation. So far, it also had to be indicated that a representation could interactively be modified by adding `_WITH_MODIFY`. This is now the default and the addition `_WITHOUT_MODIFY` has to be added to turn off this feature. Finally, the addition `_WITH_DRAW` was removed. Instead, it is determined during compile time, whether the representation defines a method `draw()`. If it does, that method will be called automatically.

The macros `STREAMABLE` and `STREAMABLE_WITH_BASE` that are usually used to define representations now generate `struct`s instead of `class`es, because representations are meant to only provide public data fields, and that is what `struct`s do by default. There is also a new class `Angle` that represents angles in radians. However, when reading them from a text stream, the extensions `deg` and `rad` are accepted. In case of `deg`, the angle is converted from degrees to radians. When writing angles to a text stream, they are always converted to degrees.

3.1.2 Log Files

Log files now contain additional information that allows replaying them even after the specification of the data types recorded has changed (in most cases). Based on the specification of each data type stored in a log file and its current specification, the data is automatically translated during the replay if it has changed. Missing fields keep their default values. Removed fields are ignored.

To further enhance the usage of log files, we added the possibility for our modules to annotate individual frames of a log file with important information. This is, for example, information



Frame	Name	Annotation
0	Behavior	Initial State.
1	Behavior	Ready State.
2692	Behavior	Set State.
8909	Behavior	Ready State.
11600	Behavior	Set State.
16422	Behavior	Getting up.
16631	Behavior	Getting up.
21681	Behavior	Getting up.
21834	Behavior	Getting up.
22733	Behavior	Ready State.
25424	Behavior	Set State.
28295	Behavior	Ready State.
31018	Behavior	Set State.
33758	Behavior	Getting up.
38525	Behavior	Ready State.
39363	Behavior	Set State.
40201	Behavior	Finished State.

Figure 3.1: Display of annotations in SimRobot.

about a change of game state, the execution of a kick, or other information that may help us to debug our code. Thereby, when replaying the log file, we may consult a list of those annotations to see whether specific events actually did happen during the game. In addition, if an annotation was recorded, we are able to directly jump to the corresponding frame of the log file to review the emergence and effects of the event without having to search through the whole log file.

This feature is accessed via the `ANNOTATION`-Macro. An example is given below:

```
#include "Tools/Debugging/Annotation.h"
...
ANNOTATION("GroundContactDetector", "Lost GroundContact");
```

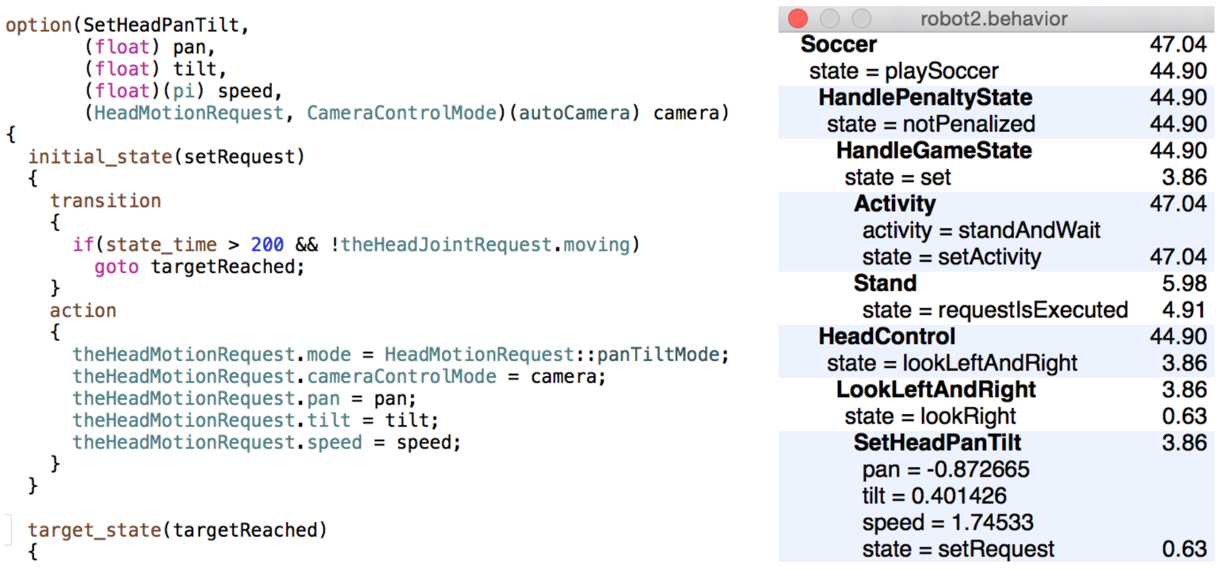
It is advised to be careful to not send an annotation in each frame because this will clutter the log file. When using annotations inside of a behavior option the output option `Annotation` should be used to make sure annotations are not sent multiple times. To view the annotations when replaying a log file, an annotation frame may be opened via the scene graph (cf. Fig. 3.1). Double clicking on an annotation will cause the log file to jump to the given frame number.

As of now, it is **not** possible to log annotations that originate inside a motion module. It is, however, possible to view annotations of motion and cognition modules when using the simulated NAO or a direct debug connection to a real NAO. This has to be activated by using the following debug request:

```
dr annotation
```

3.1.3 C-based Agent Behavior Specification Language (CABSL)

The *C-based Agent Behavior Specification Language (CABSL)* [12] now uses the same syntax for specifying parameters as the streamable data types in our system. Thereby, the actual parameters of each option can be recorded in log files and they can also be shown in the behavior dialog (cf. Fig. 3.2). Please note that the source code shown in Fig. 3.2 is still understood by a C++ compiler and thereby by the editors of C++ IDEs.



```
option(SetHeadPanTilt,
       (float) pan,
       (float) tilt,
       (float)(pi) speed,
       (HeadMotionRequest, CameraControlMode)(autoCamera) camera)
{
  initial_state(setRequest)
  {
    transition
    {
      if(state_time > 200 && !theHeadJointRequest.moving)
        goto targetReached;
    }
    action
    {
      theHeadMotionRequest.mode = HeadMotionRequest::panTiltMode;
      theHeadMotionRequest.cameraControlMode = camera;
      theHeadMotionRequest.pan = pan;
      theHeadMotionRequest.tilt = tilt;
      theHeadMotionRequest.speed = speed;
    }
  }

  target_state(targetReached)
  {
```

```
robot2.behavior
Soccer                          47.04
 state = playSoccer             44.90
  HandlePenaltyState            44.90
   state = notPenalized         44.90
    HandleGameState             44.90
     state = set                 3.86
      Activity                  47.04
       activity = standAndWait
       state = setActivity      47.04
      Stand                      5.98
       state = requestIsExecuted 4.91
  HeadControl                   44.90
   state = lookLeftAndRight      3.86
    LookLeftAndRight             3.86
     state = lookRight           0.63
      SetHeadPanTilt             3.86
       pan = -0.872665
       tilt = 0.401426
       speed = 1.74533
       state = setRequest        0.63
```

Figure 3.2: The behavior view shows the actual parameters, e. g. of the option *SetHeadPanTilt* (left: source, right: view). The display of values equaling default parameters (specified in a second pair of parentheses) is suppressed. On the right of the view, the number of seconds an option or state is active is shown.

3.1.4 Eigen

We decided to replace our own math library with *Eigen* [8] which is a "C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms". We kept some special methods that Eigen does not offer and integrated them through Eigen extensions. Since Eigen adds a considerable source code footprint to compilation, we use precompiled headers in our build system for all projects that include Eigen[1]. It is important to note that the data fields of Eigen objects are initialized with `NaN`s by default when built with the configurations *Debug* and *Develop* and not initialized at all when built with *Release*.

3.1.5 Wrist and Hand Joints, Tactile Sensors, and Head LED

As it is possible (by the new robots) to use the wrist and hand joints, we added them to our whole system. So they can be used like every other joint. For our robots without an H25 body, using the same system, the use of the nonexistent joints has no effect or rather no sensor data (normally 0) will be returned. If someone nevertheless needs to know if a feature is existent within the robot, *RobotInfo* is holding the adequate information. Since we decided against loading the corresponding modules of the NAOqi API, the robot type will be read in as part of the NaoProvider parameters.

In addition to these modifications, we also extended the *KeyStates* by the tactile sensors and the *LEDRequest* by the head LED.

3.1.6 Additional Changes

- The *SensorData* representation was split up into six smaller ones: *SensorData/FsrSensorData*, *SensorData/InertialSensorData*, *SensorData/JointSensorData*, *SensorData/KeyStates*,

[1]On OS X and when targeting Windows.

SensorData/SystemSensorData, *SensorData/UsSensorData*. This way, modules can request only the data they actually need.

- *JointData* was renamed to *JointAngles*.
- *HardnessData* was renamed to *StiffnessData*.
- Removed the InertiaSensorCalibrator since NAOqi does not provide the raw IMU data anymore.
- Removed *FilteredSensorData* and *FilteredJointData*. Use the *SensorData* representations mentioned above and *JointAngles* instead.
- The *InertialData* representation is a combination of *InertiaSensorData* and *OrientationData* and is provided by the InertialDataFilter.
- The URC [12] compiler was removed. Instead, *SpecialActions* are compiled by *bhuman* on startup.

3.2 Tools

3.2.1 Automatic Camera Calibrator

The former CameraCalibratorV6 (cf. [13], chapter 2.8.3) has been renamed to AutomaticCameraCalibrator which is way more meaningful. The other changes to improve the calibration and make the whole process more comfortable are:

1. A distance check between samples. For the purpose of a better weighting of the samples, we now have the constraint that a new sample must have at least a distance of 30 cm to every other sample that is already taken.
2. A manual deletion of samples is now possible by left-clicking into the image and on the point the sample has been taken.
3. A manual insertion of samples is now possible by *CTRL* + left-clicking into the image at the point you want the sample to be.
4. Command generation for correcting the body rotation. In case you don't want the *BodyRotationCorrection* stored in the *CameraCalibration*, you can manually call the JointCalibrator and transfer the values or you can use the command
   set module:AutomaticCameraCalibrator:setJointOffsets true
   before running the optimization. After the optimization, a bunch of commands will be generated and you can enter them in order of appearance to transfer the values into the *JointCalibration*.

3.3 Perception

3.3.1 Goal Perception

Due to the change of the color of the goal from yellow to white, as depicted in the new rules, a new goal perception needed to be built. Our old algorithm searched for yellow areas that

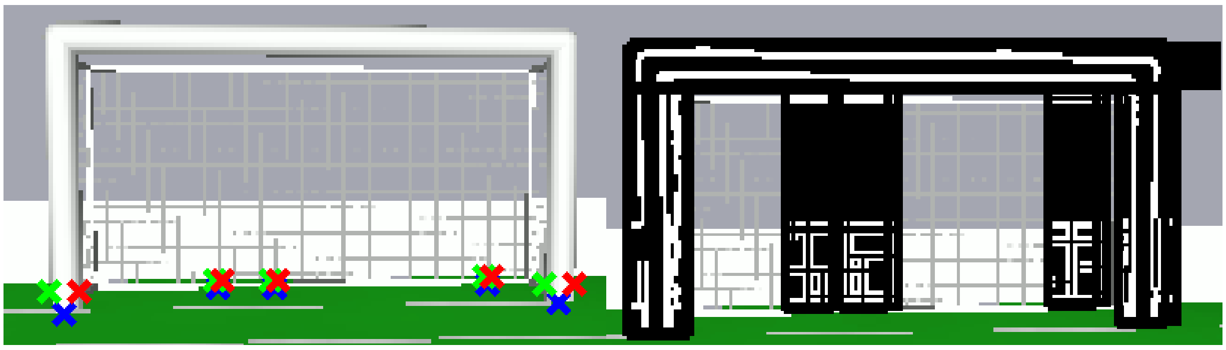


Figure 3.3: The left image shows the found regions, which could represent a goalpost. The blue cross is the base point, the red and green post represents the edge of the region.
On the right, the (simulated) image of a goal is shown, where the areas around the goalposts are convolved with the Sobel operator.

intersect the horizon of the robot [12]. At the German Open 2015, a similar method to detect goals was used, but proved unsuccessful due to the amount of white regions in the environment and on the field.

The new goal perception searches for lines in the edge map of the upper image.

Initially, the algorithm searches for white regions below the field border, which are then evaluated by their height and width. If such a region is found, it should either be a robot or a goalpost, since every other object on the field shouldn't have the color white or is too small to be recognized as a possible goalpost (cf. Fig. 3.3). Afterwards, for all potential regions a model of the goal is calculated by using the lowest horizontally middle point of the region as the base point. The size in pixels of a goalpost can be estimated given the real dimensions of a goal.

To search for edges on the whole image takes a lot of time. To speed up the process, only areas with interesting features should be scanned. By using the goal model, an area where a goalpost should be can be determined. This area is then convolved with the Sobel operator to search for edges around the goalpost (cf. Fig. 3.3).

Under the assumption that a goalpost is roughly vertical in an image, a Hough transform can be executed, with an angle between $\pm x$ degrees, where x is a parameter (we used the value 3 during RoboCup 2015). The Hough transform then extracts all lines that contain at least a certain number of edge points. If a pixel in the edge map is an edge can either be determined by using a constant value or by using the Otsu threshold algorithm.

By only using a threshold to detect the lines, a lot of possible lines might be found. In the next step the Lines which most likely represent a goalpost are chosen, for this all lines are sorted by the proximity of their hough value to the expected hough value. Then the line with the best evaluation and a second line which has a minimal distance and doesn't cross the first one will be chosen (cf. Fig. 3.4).

Afterwards, the numerical evaluations will take place:

- The difference between the angle of both lines
- The distance of the average to the expected value
- The number of pixels contained in the lines of the post

If the resulting value is above a certain threshold, two exclusion criteria might be used (depending on the value of the parameters):

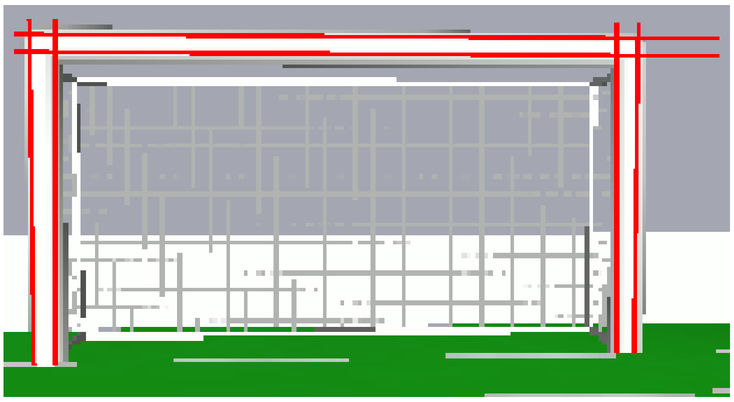


Figure 3.4: Goal with found Hough lines after the filter process

1. Are there more than 40% non-white or pixels with an edge value above the threshold between the two lines?
2. Is the variance between the lines higher than a certain threshold?

As the last step, it needs to be checked, if the found goalposts are part of a goal. If only one goalpost has been found, it is immediately accepted as part of a goal. If more than one goalpost is found, they need to be further checked.

For each combination of goalposts, starting with the best ones, a numerical evaluation takes place. If the resulting value is above a threshold, the search will be stopped and the posts will be accepted as a complete goal. If no combination is above the threshold, only the best post will be accepted as part of the goal.

The numerical evaluation consists of the average value, the distance between and the parallelism of the goalposts. Afterwards, the value is multiplied with the evaluation of a crossbar plus one. One is always added since it is not given that a goalpost is visible between two posts, but if a crossbar is visible, it confirms the hypothesis that both posts are part of a goal.

The result will then be provided in the *GoalPercept*.

Parameters

The module has parameters that affect its behavior:

useAlternativeLineSearch: Instead of the described methods to filter Hough lines, another approach is used which searches for two lines next to a gap of a certain size.

useCheckBetweenLines: If set to true, exclusion criteria 1 is used

useNMS: Set to true to use non-maximum suppression on the edge map

useNoise: If set to true, exclusion criteria 2 is used

useOtsu: Use Otsu to automatically determine the Sobel threshold

Debugging

The module provides five main debug drawings to illustrate its activity:

PossiblePosts: Shows the initially found region which represents possible goalposts

Edges: Shows the created edge maps filtered with a threshold (use EdgesUnfiltered for the original version)

HoughLinesFiltered: Shows the filtered hough lines in the image (use HoughLines for the unfiltered version)

CheckBetweenLines: Shows which pixels are accepted by the exclusion criteria (green) and which not (red)

Validation: Draws all validation values for a goalpost

3.3.2 Penalty Mark Perception

In response to the change of the goal color, which makes the goal perception more difficult, a penalty mark perceptor was built to aid the localization of the robots. The penalty marks can help the localization more than the lines and intersections on the field, because they only exist twice on the field in central positions exactly known.

The recognition of the lines and the penalty marks relies on the recognition of spots that could form a line. While a line consists of multiple close spots, a penalty mark consists of an isolated spot without surrounding white outside of the dimensions of the penalty mark.

To maximize code sharing, the process of detecting the possible line spots has been extracted from the module LineSpotProvider into the PotentialLineSpotsProvider. The LineSpotProvider and the PenaltyMarkPerceptor both require the representation *PotentialLineSpots* provided by the PotentialLineSpotsProvider.

To detect a penalty mark, the line spots found are scanned and evaluated according to the following criteria:

Could it form a penalty mark? From the center of the potential line spot, the endpoints of its surrounding white region are searched in horizontal and vertical direction. The x coordinate of the penalty mark is set to the mean of the x coordinates of the left and right endpoints, the y coordinate is set to the mean of the y coordinates of the upper and lower endpoints. It is then checked, whether the width and height of the region are too big for a penalty mark and whether they are too narrow. If these two checks fail, a rectangle around the region is scanned. For each pixel, it is checked, whether there are too many green pixels in a small square around it. In addition, it is checked if the variance of the Y channels of the pixels scanned is too big. If one of these checks succeeds, the spot is discarded.

Is it far enough away from the field border? A real penalty mark is always far away from the field border. If the distance of the spot to the field boundary is too small, it is discarded.

Is it close enough to the observer? If the distance of the spot from the robot is above a certain threshold, it is discarded because it is most likely a false positive.

Is it not inside a ball? It might happen that a potential line spot is recognized on the surface of a ball. To account for this, it is checked, if the spot is inside a perceived ball. If this is the case, the spot is discarded.

The first spot that fulfills these criteria is set as a recognized penalty mark and no further spots are evaluated, because there cannot be two penalty marks in the same image that both aren't too far away. The properties of the penalty mark recognized are provided in the representation *PenaltyMarkPercept*.

Parameters

The module has parameters that affect its behavior:

expectedSize: The width and height in mm of the penalty mark.

toleratedSizeDeviation: The tolerated percentage deviation of the size of the spot found from the expected size.

minDistToFieldBoundary: The minimal distance in mm that a penalty mark found can have from the field boundary.

noiseAreaSize: The size of the square around a pixel that is searched for non-green pixels.

maxDetectionDistance: The maximum distance in mm from of a penalty mark recognized to the robot.

whiteSpacing: The additional spacing outside of the white region to search for a green surrounding.

numNotGreenMax: The maximum count of non-green pixels.

maxVariance: The maximum variance of the Y channel of the pixels allowed.

Debugging

The module provides the following debug drawings:

penaltyPointScanLines: Shows the rectangle of white regions, which represents potential line spots as well as additional information about the spot such as its width and height.

noiseRect: Shows the noise rectangle.

3.3.3 Players Perception

As it is now allowed to play in custom jerseys, we had to change our previously used PlayersPerceptor (cf. [13], chapter 3.2.5) to identify our own black jerseys and not only the cyan and magenta ones.

The perceptor receives the team color information from the game controller and chooses the corresponding color values from the parameter **colors**. If one of the teams uses black jerseys, we scan up to four times with the first scan looking at the shins of the Nao, to check whether it is a referee or not. Otherwise, we scan up to three times. The other scans are performed below the shoulders, at the height of the chest, and at the stomach, when no jersey was detected in each previous scan.

During each scan, the color values of scanned pixels are analyzed by using the following conditions:

1. Is the **intensity high** and **the saturation low** enough to be white?
   Then a counter for white will be increased.
2. Is the **hue and intensity value like green**?
   Then the pixel will be ignored.
3. **Is one of the jerseys black?**
   - **Yes**
     - Is the **intensity too high for black** but **low enough for a dark** color and the **saturation not too high for gray**?
       Then it could be a black jersey.
   - **No**
     - Is the **intensity too high for black**?
       It will be checked which teamcolor is within the ranges or next to the color of the pixel, if both are within the ranges.

Depending on which color dominates, the obstacle will be marked as an opponent, teammate or obstacle.

Parameter

The module has parameters that affect its behavior:

jerseyYFirst & jerseyYSecond & jerseyYThird: First, second, and third height within a player to scan for jersey colors.

refereeY: If the perceptor finds black pixels at this height, then it is a referee.

maxWhiteS & minWhiteI: Colors with an intensity above maxWhiteI and a saturation below maxWhiteS look like white/gray.

minGreenH & maxGreenH & minGreenI: Colors with a hue between min- and maxGreenH and with an intensity above minGreenI look like the field.

maxBlackI: Colors with a intensity below the value look like black.

maxDarkI: Colors with a intensity below the value are darker.

maxGrayS: Colors with a saturation below the value look like gray.

maxRangeH & maxRangeS & maxRangeI: specifies how big the differences between the hue, saturation, or intensity of the found pixel and the searched color can be to count the pixel.

colors: Prototypical HSI color values for all team colors.

3.4 Motion Selection and Combination

In contrast to the B-Human code release of 2013 [12], the arms are considered more independently. There is now an option to control the arms and legs by using different motion engines.

The MotionSelector determines for all (whole-body and arm) motion engines how intense their output defines the final joint angle request, which is sent to the NAO. Therefore, a whole-body engine (which provides joint angles for legs and arms) will be selected and the interpolation ratios are calculated in order to switch between the whole-body engines. Afterwards, the same will happen per arm for the arm motion engines. Additionally, the selected whole-body engine can force the selection of a placeholder, which means that this whole-body engine also defines the arm joint angles. The GetUpEngine is an example for this since it needs full control over the whole body to accomplish its purpose.

The MotionSelector's decision is based on the behavior requirements, which are defined in the *MotionRequest* and the *ArmMotionRequest*, as well as the interruptibility of the current motion, which is handled by the associated motion engine on purpose to guarantee a motion that is not exited or changed in an unstable situation. The decisions and ratios are stored in the *MotionSelection* and the *ArmMotionSelection*.

The MotionCombinator takes these information and merges the joint angles, provided by the different motion engines, accordingly. The interpolation is linear. If no interpolation between different motions is needed, the output of the engine will be only copied. In addition to filling the final target joint angles, the MotionCombinator also fills the representations *MotionInfo*, *ArmMotionInfo*, and *OdometryData*, which are holding information about the current motion(s), e. g. the current position in the walk cycle, the stability of the motion, and the odometry position.

The MotionCombinator, as the last instance before the joint angle will be sent to the NAO, also performs emergency actions in case the robot is falling. This includes centering the head and the arms as well as reducing the joint stiffness.

3.5 Behavior

The intention of this year's major behavior changes were to play more defensivly and to let the robots move less. To accomplish this, we played with the following lineup:

1. One Keeper: The goalkeeper robot.
2. Two Defenders: Robots that are positioned defensively to help the Keeper by defending their own goal.
3. One Striker: The ball-playing robot.
4. One Supporter: A robot that is positioned offensively to help the Striker.

In a lineup with less than five robots, at least one robot should be the Defender. The ideas behind the changes are to prevent dribbling teams from gaining a goal and to save battery power and avoid joint heat. Saving battery power is important for us since we had issues to play multiple long games on a tournament day with the batteries avalable to us. In the same scenario, it is important to avoid joint heat and the consequential joint stiffness loss to perform motions on the highest possible level if they are really needed.

These major changes did not affect the Striker and the Keeper. Thus, the remainder of this section describes the Defender in more detail. The Supporter has been created by combining some of the old roles.

3.5.1 Defender

A Defender has five modes, which differ only slightly by occurrence. These are: back-left, back-middle, back-right, forward-left, and forward-right – the areas where they position themselves. Whereas the role Defender is taken upon consultation with the robot's team members, the decision of the taken mode is made by the Defender alone. It bases its decision on the received pose of the second Defender (or on the absence of such data). If there are two Defenders, they must be left or right and never on the same side. In addition, one of them must hold a back-mode but the other may be forward. Otherwise, if there is just one defending robot, it must be a back one.

In each frame, the Defender recomputes

1. if it is left or right (or in special case middle),
2. if it is forward,
3. which position it should take, and
4. if it should move.

3.5.1.1 The Left / Right Decision

If the Defender is left, it means that it will calculate a position on the left side of the virtual line between the goalkeeper and the ball. If the goalkeeper is not on the field (or not in the penalty area), the line origin is based in the middle of the own goal. A middle defending position means that the Defender stands on this line. This position is just a special case if neither a Keeper is inside the penalty area nor a team member performs a second Defender. In all other cases, you do not want to interfere with the direct goalkeeper sight of the ball.

- Being a single Defender (with a present Keeper), a robot will always decide for the same side like last frame as long as the ball is not too far on the other side.
- In the case of two Defenders, each robot will compare its (signed) distance to the goalkeeper-ball line and the distance of the last known position of the other Defender to that line. If the robot can clearly decide if it is more left of the line, it takes the decision. If this is not the case, it tries to decide by looking up which of the both poses is clearly more left on the field (left on the field means if you are standing in your own goal and looking in direction of the opponent ones). If the robot still does not know which side it belongs to, it uses the decision of the last frame.

3.5.1.2 The Back / Forward Decision

A back position is defined as a position directly outside the penalty area, a forward position could be farther away. In the case of a single defending robot, the position will be back as mentioned above. In the other case, the robot decides on the basis of its actual position, if it is distinctly forward or back. It does the same for the position of the second Defender. The robot will hold its decision, if it differs from the position about the second Defender or if it is back and the ball is not far away. If both cases are not applicable and the robots are clearly looking in different directions, the robot that is looking in the direction of the opponent goal will move forward. If none of the cases were applicable and the Defender robot is on the same side as the ball, it is moving forward. If the ball side is not clear, both Defenders stay back.

3.5.1.3 The Position Selection

All selected positions for the Defender are lying on a semicircle, which is centered around the center of the own goalposts. The radius is decided by the previously described back / forward mode. If it is back, the semicircle goes barely around the penalty area, but the maximum semicircle for a forward Defender makes a wide bow around the own goal. As long as the ball is not inside the own penalty area, the forward radius is always smaller than the distance of the semicircle origin to the ball. The Defender will take a position on the selected semicircle with a specific angle distance to the goalkeeper-ball line, which let the robots, which are more in the back, stay closer to the line than the others. To avoid that the distance of the back-standing robots to the penalty area gets to great, the semicircles may be cut along the lines of the penalty area. There is also a special handling when the ball is entering the penalty area or is lying close to the own ground line.

3.5.1.4 The Decision to Move

Because of the aim to avoid unnecessary movements, it would be counterproductive to move to every new calculated position. In account to this, the robot will always decide after the position calculation if it should move or could wait. The decision is based on the distance to the new position, the distance to the ball, the distance to the angle and the distance to the goalkeeper-ball line. The threshold for this may vary if the robot was moving before or not.

It is important that the Defender is doing its job, but as the last instance, between opponent scoring or a successful defense, there is still the goalkeeper. To give him time for a in time reaction, it must see the ball early enough. To maximize this time, the Defenders are also performing arm movements to make themselves as tiny as possible, if the goalkeeper-ball line comes close to the robot.

Chapter 4

Technical Challenges

In this chapter, we describe our contributions to the three Technical Challenges of the 2015 SPL competition, i. e. the *Corner Kicks Challenge*, the *Many Carpets Challenge*, and the *Realistic Ball Challenge*. We placed second in each challenge which lead to the win of the overall competition [5]. The detailed rules of each challenge can be found in [6].

4.1 Corner Kicks Challenge

To handle this challenge, we used our normal vision and walking modules and replaced the behavior as well as our kick engine. In contrast to our normal kick engine the one used in this challenge is capable of kicking the ball at arbitrary speeds, thus it is ideal for playing precise passes which where necessary to successfully compete in this challenge. The reason for this is that we did not know the positions of the opponent robots in advance and had to pass the ball to a position where it could be kicked in to the goal quickly by the second robot. The kick engine used in this challenge was developed as part of a master thesis, which hasn't been published [3]. A quick overview is given in 4.1.1.

Furthermore, this challenge was used to test the implementation of a new self-localization module (which is not yet finished and not part of the code release). Due to the field's symmetry, a robot's self-localization cannot rely on field elements only to determine the team's current playing direction. Therefore, additional resetting mechanisms that derive the current field half from transitions of game states (for instance, a robot is definitely in its own half, if the game state changes from SET to PLAY) are common. However, for most technical challenges, the game states are used in a different way. This means that for each technical challenge, a copy of the normal self-localization, which contains a set of workarounds, is necessary. The new implementation is based on a state machine that models the transitions between game states as well as certain robot states. Thereby, the adaption to a different rule set can be done easily by adding a few simple states. The underlying state estimation process is still based on a set of Unscented Kalman filters.

4.1.1 A Kick Engine based on Dynamic Movement Primitives

The kick engine consists of three logical modules: A mathematical model of the motor behavior which is used to predict the current joint positions, a ZMP-based balancer that keeps the robot stable while kicking, and an execution module that executes the kick trajectory.

4.1.1.1 Prediction of Motor Behavior

The NAO's motors react to commands with a delay of approximately 30 ms. This delay greatly complicates any closed loop control approach. To avoid the delay, the behavior of each motor is modeled using a second degree linear time invariant system:

$$T^2\ddot{y}(t) + 2DTmin(\dot{y}(t), V_{max}) + y(t) = Ku(t) \tag{4.1}$$

where $y(t)$ is the angle of the motor, $u(t)$ is the target angle, K and D are dampening constants, T is the time constant and V_{max} is the maximum velocity of the motor. Using this model, the current position of the NAO's motors can be estimated and used for closed loop control.

4.1.1.2 Balancing

To keep the robot dynamically balanced while kicking, a ZMP-based balancer similar to the ones presented in [7] and [16] is used. This approach assumes that the support foot does not move or tilt while executing the kick, i. e. the sole of the support foot is fully connect to the ground plane. This is not always the case, especially when executing strong kicks, the robot might tilt around the edges of the support foot. To improve balancing in that case, an augmented center of mass is used as foundation of the ZMP calculation as shown in [2].

4.1.1.3 Modeling of the Kick Trajectory

The kick trajectory is modeled in Cartesian space using Dynamic Movement Primitives (DMPs). A general introduction to DMPs can be found in [9] and [14]. The original DMP formulation does not allow for an end velocity other than zero. Therefore, the NAO's kick trajectory is based on extended DMP formulations by Katharina Mülling [11] and Jens Kober [10], which have been specifically designed for hitting and batting movements and allow for an end velocity other than zero:

$$\tau\dot{z} = \alpha_z(\beta_z(g_p - y) + \tau\dot{g_p} - z) + \tau^2\ddot{g_p} + sf(s)B \tag{4.2}$$

$$\tau\dot{y} = z \tag{4.3}$$

$$\tau\dot{s} = -\alpha_s s \tag{4.4}$$

Where $\alpha_z = 6.25$ and $\beta_z = 25$ are dampening constants set up for critical dampening, τ is the point in time at which the final position should be reached, g_p, $\dot{g_p}$, and $\ddot{g_p}$ are the target position, velocity, and acceleration, s is the phase and f is the forcing term.

The original formulation by Mülling contains a scaling factor η which is used to ensure that the trajectory scales well when changing the target position. This factor is replaced by the novel scaling factor $B = \frac{\dot{g_p} - \dot{g_0}}{\dot{g_p}^{demo} - \dot{g_0}^{demo}}$ where g_0 is the start position of the trajectory, g_p^{demo} is the goal position of the demonstrated trajectory, and g_0^{demo} is the start position of the demonstrated movement. It ensures that the trajectory scales well when changing the target velocity $\dot{g}$.

4.2 Many Carpets Challenge

Although being intended as a walking challenge, the actual tasks to solve were the behavior and the localization. For this challenge, we were able to use our normal walking engine with slightly modified parameters. We wrote a behavior which observes the surroundings to get an initial localization, then walks to the perceived ball and then tries to localize again by searching for the goal. If the robot is certain that the localisation is correct enough or a certain time threshold has been exceeded, a kick is performed.

The performance of our robot at the challenge was filmed and can be viewed online at `https://www.youtube.com/watch?v=bq3Y7OfQWVI`.

4.3 Realistic Ball Challenge

The aim of the *Realistic Ball Challenge* was to make robots play with arbitrary balls different from the standard orange one, similar to the *Any Ball Challenge* in 2009. In this challenge, five arbitrary balls that were provided by the participating teams as well as two robots were placed within an area of 4×4 meters around the center of the field and the competing robot had to move the balls into or at least towards any of the goals.

In order to compete in this challenge, we replaced the modules for perception of the ball and adjusted the behavior to conform to the rules of the challenge. Concretely, the *BallPercept* is not provided by the *BallPerceptor*, but instead by the new *RealisticBallPerceptSelector* which fills it with one of the balls contained in the representation *RealisticBallPercepts*. This representation is provided by the new perception module *RealisticBallPerceptor* and contains all balls seen in the current camera image. By using this method, it is possible to use the normal *BallLocator* to provide the *BallModel* that is used by the behavior, always targeting one ball at a time.

The performance of our robot at the challenge was filmed and can be viewed online at `https://www.youtube.com/watch?v=V0tVxfKCspw`.

4.3.1 RealisticBallPerceptor

As stated above, the main task of finding balls in the camera images is done by the *RealisticBallPerceptor*. This is done in the following steps:

1. Create an edge map of the image.
2. Use the FRHT algorithm described in [4] to find circles and circular arcs.
3. Perform validation checks to keep only candidates that probably correspond to actual balls.

In order to perform the first step, three different methods were implemented and can be selected via the parameter *edgeDetectMode*. All of these first apply the Sobel operator to the image. The first method then obtains an edge map by simply marking all pixels with a gradient higher than a set threshold as edge pixels. The second method additionally classifies the direction of the edge through each pixel and based on this performs a simple non-maximum suppression. The most complex of the methods is the third which is an implementation of the Edge Drawing algorithm described in [15]. While the first method is the fastest, the edges in the resulting edge map are usually multiple pixels thick, leading to many pixels being checked in step two, so the overall computation time is highest with this method. The third method gives perfect one pixel thick edges which are ideal for circle detection, however it needs longer to compute the edges. As a tradeoff between these two, the second method was implemented and ultimately used in the challenge. Its output can be seen in figure 4.1. In order to save computation power and

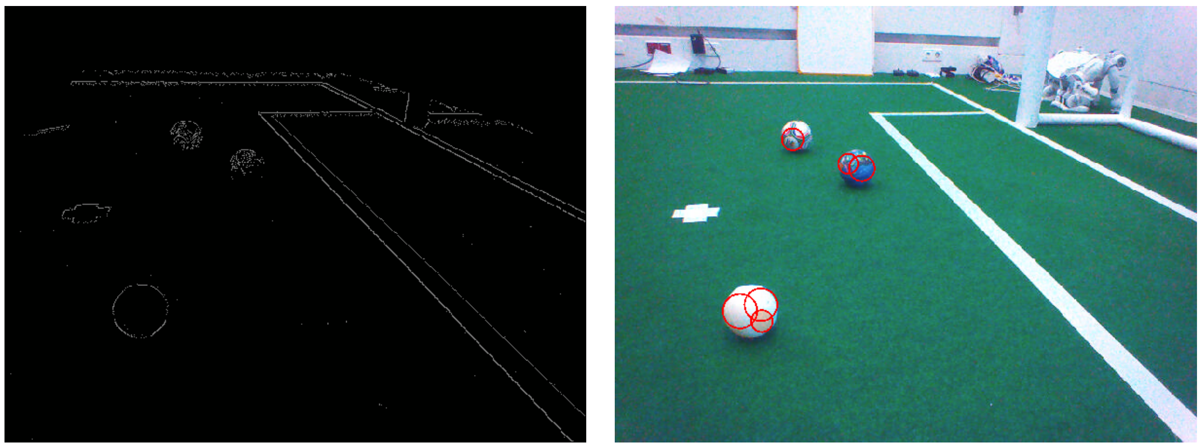


Figure 4.1: On the left side, the edge map created by the *RealisticBallPerceptor* can be seen. Here, the second method of creating the map was used, i. e. applying the Sobel operator and performing a simple non-maximum suppression. The image on the right shows the ball percepts found by the *RealisticBallPerceptor*.

prevent false positives, the edge detection is only done on parts of the image that lie within the *FieldBoundary* and not within the *BodyContour* computed for the current frame.

For step two, the FRHT algorithm was extended by merging the found circle candidates if their centers are within a specified distance from each other and they have roughly the same radius in order to not have multiple circles detected from one circle, which the algorithm often does when the edge maps do not contain perfect circles.

In step three, the following is done for each circle:

- Assert that the center of the circle lies within the image.
- Scan horizontally, vertically and diagonally from the center of the circle to its arc and check if the percentage of pixels that are green according to the color table is either below a set threshold or over another so that the circle can either be assumed to be a non-green ball or a completely green ball. This check exists to prevent from detecting balls in the center circle.
- Compute the position and the radius of the ball on the field.
- Check that the computed radius of the ball is reasonable and the ball lies inside the field.
- If the center pixel of the ball is white according to the color table, assert that the ball does not lie on one of the field lines or the penalty mark. This check is necessary as the FRHT algorithm, which only needs three fitting points to form a circle candidate, sometimes detects circles in field lines.

4.3.2 RealisticBallPerceptSelector

As previously written, the *RealisticBallPerceptSelector* is responsible for selecting one of the percepts that the *RealisticBallPerceptor* found for giving it to the *BallLocator* via the normal *BallPercept* representation.

In order to do so, it first compares the positions of the percepts to those of the *PlayersPerceptor* and the positions of known obstacles to avoid running into an obstacle when a false positive ball detection occured in a robot.

If the last percept was selected within a given timespan (during the competition, this was two seconds), it searches the percepts for one that lies near the last position of the ball model and selects this percept. Otherwise, it selects the percept that is nearest to the robot.

In Fig. 4.1, the selected percept is the circle nearest to the robot, it is filled with a light red color.

Chapter 5

Team Communication Monitor

With the start of the Drop-in Player Competition in 2014, a standard communication protocol, i. e. the *SPLStandardMessage*, was introduced to allow robots of different teams to exchange information. While implementing the standard protocol correctly is a necessity for the Drop-in Player Competition to work, it is also an opportunity for teams to lower the amount of coding they have to do to develop debugging tools, because when all teams use the same communication protocol, the tools they wrote could also be shared more easily. Therefore, B-Human developed the *TeamCommunicationMonitor* (TCM) as a standard tool to monitor the network traffic during SPL games. This project has been partially funded by the RoboCup Federation.

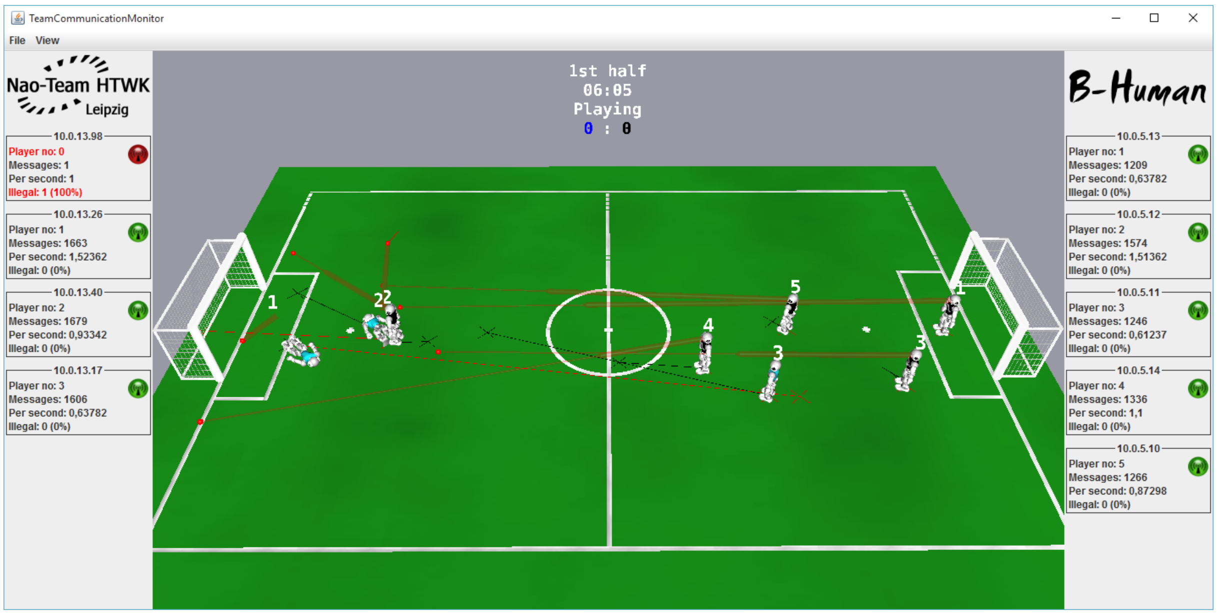
The TCM visualizes all data currently sent by the robots and highlights illegal messages. Furthermore, it uses knowledge about the contents of SPLStandardMessages in order to visualize them in a 3-D view of the playing field. This makes it also suitable for teams to debug their network code. Visualized properties of robots are their position and orientation, their fallen and penalty states (the latter is received from the GameController), where they last saw the ball, and their player numbers. A screenshot of the TCM can be seen in Fig. 5.1.

In order to display all messages of robots sending packets, the TCM binds sockets to all UDP ports that may be used by a team as their team number, i. e. ports 10000 to 10099, and listens for any incoming packets. When a packet is received, it is parsed as an SPLStandardMessage, marking all fields as invalid that do not contain legal values according to the definition of the SPLStandardMessage. As the TCM only listens and does not send anything, it may run on multiple computers in the same network at once without any interferences.

The TCM identifies robots using their IP address and assumes them to be sending on the port that matches their team number. The team number sent by the robots as part of the SPLStandardMessage is marked as invalid if it does not match the port on which the messages are received. For each robot, the TCM holds an internal state containing the last received message as well as the timestamps of the most recently received messages in order to calculate the number of messages per second. A robot's state is discarded if no message was received from the robot for at least two seconds.

Displaying the 3-D view is done with OpenGL using the JOGL library [1]. The visualization subsystem is also extensible to enable teams to write plug-ins containing drawings that visualize data from the non-standardized part of their messages. We have written a plugin to display perceptions of goals, obstacles, and whistles as well as basic status information of the robots which our team communicates via the non-standardized message part.

Besides just visualizing received messages, the TCM also stores all received messages both from robots and the GameController in log files. These can be replayed later, allowing teams and



```
10.0.13.98
Player no: 0
Messages: 1
Per second: 1
Illegal: 1 (100%)

10.0.13.26
Player no: 1
Messages: 1663
Per second: 1,52362
Illegal: 0 (0%)

10.0.13.40
Player no: 2
Messages: 1679
Per second: 0,93342
Illegal: 0 (0%)

10.0.13.17
Player no: 3
Messages: 1606
Per second: 0,63782
Illegal: 0 (0%)

10.0.5.13
Player no: 1
Messages: 1209
Per second: 0,63782
Illegal: 0 (0%)

10.0.5.12
Player no: 2
Messages: 1574
Per second: 1,51362
Illegal: 0 (0%)

10.0.5.11
Player no: 3
Messages: 1246
Per second: 0,61237
Illegal: 0 (0%)

10.0.5.14
Player no: 4
Messages: 1336
Per second: 1,1
Illegal: 0 (0%)

10.0.5.10
Player no: 5
Messages: 1266
Per second: 0,87298
Illegal: 0 (0%)
```

Figure 5.1: Display of the TeamCommunicationMonitor during a game between the teams Nao-Team HTWK and B-Human

organizers to check the communication of robots after the game is over.

The TCM is developed as part of the repository of the GameController (`https://github.com/bhuman/GameController`), which is the standard referee tool of the SPL and Humanoid leagues developed mostly by B-Human team members since 2012, and it shares parts of its code.

After having been successfully used at the RoboCup German Open 2015 to ensure valid messages from all teams during the Drop-in games, the TCM was publicly released in early June and was installed on the referee PC on each SPL field at RoboCup 2015.

Chapter 6

Acknowledgements

We gratefully acknowledge the support given by Aldebaran Robotics. We thank our current sponsors IGUS, TME, and Wirtschaftsförderung Bremen (WFB) as well as our supporters from the unifreunde Bremen for funding parts of our project. Since B-Human 2015 did not start its software from scratch, we also want to thank the previous team members as well as the members of the GermanTeam and of B-Smart for developing parts of the software we use.

In addition, we want to thank the authors of the following software that is used in our code:

AT&T Graphviz: For generating the graphs shown in the options view and the module view of the simulator.
(`http://www.graphviz.org`)

ccache: A fast C/C++ compiler cache.
(`http://ccache.samba.org`)

clang: A compiler front end for the C, C++, Objective-C, and Objective-C++ programming languages.
(`http://clang.llvm.org`)

Eigen: A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
(`http://eigen.tuxfamily.org`)

FFTW: For performing the Fourier transform when recognizing the sounds of whistles.
(`http://www.fftw.org`)

getModKey: For checking whether the shift key is pressed in the Deploy target on OS X.
(by Allan Craig, no download link available anymore)

gtest: A very powerful test framework.
(`https://code.google.com/p/googletest/`)

ld: The GNU linker is used for cross linking on Windows and OS X.
(`https://sourceware.org/binutils/docs-2.21/ld`)

libjpeg: Used to compress and decompress images from the robot's camera.
(`http://www.ijg.org`)

libjpeg-turbo: For the NAO we use an optimized version of the libjpeg library.
(`http://libjpeg-turbo.virtualgl.org`)

libqxt: For showing the sliders in the camera calibration view of the simulator.
(`https://bitbucket.org/libqxt/libqxt/wiki/Home`)

libxml2: For reading simulator's scene description files.
(`http://xmlsoft.org`)

mare: Build automation tool and project file generator.
(`http://github.com/craflin/mare`)

ODE: For providing physics in the simulator.
(`http://www.ode.org`)

OpenGL Extension Wrangler Library: For determining, which OpenGL extensions are supported by the platform.
(`http://glew.sourceforge.net`)

Qt: The GUI framework of the simulator.
(`http://www.qt.io`)

qtpropertybrowser: Extends the Qt framework with a property browser.
(`https://github.com/commontk/QtPropertyBrowser`)

snappy: Used for the compression of log files.
(`http://google.github.io/snappy`)

Bibliography

[1] Jogl - java binding for the opengl api. `http://jogamp.org/jogl/www/`.

[2] J. J. Alcaraz-Jiménez, D Herrero-Pérez, and H Martínez-Barberá. Robust feedback control of zmp-based gait for the humanoid robot nao. *The International Journal of Robotics Research*, 32(9-10):1074–1088, 2013.

[3] Arne Böckmann. Entwicklung einer dynamischen schussbewegung mit dem humanoiden roboter nao. Master thesis, University of Bremen, 2015.

[4] Shih-Hsuan Chiu, Jiun-Jian Liaw, and Kuo-Hung Lin. A fast randomized hough transform for circle/circular arc recognition. *IJPRAI*, 24(3):457–474, 2010.

[5] RoboCup Technical Committee. Results2015 – Standard Platform League, 2015. Only available online: `http://www.informatik.uni-bremen.de/spl/bin/view/Website/Results2015#Technical_Challenges`.

[6] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) technical challenges, 2015. Only available online: `http://www.informatik.uni-bremen.de/spl/pub/Website/Downloads/Challenges2015.pdf`.

[7] Stefan Czarnetzki, Sören Kerner, and Oliver Urbann. Applying dynamic walking control for biped robots. In Jacky Baltes, Michail G. Lagoudakis, Tadashi Naruse, and SaeedShiry Ghidary, editors, *RoboCup 2009: Robot Soccer World Cup XIII*, volume 5949 of *Lecture Notes in Computer Science*, pages 69–80. Springer, 2010.

[8] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[9] Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors. *Neural Computation*, 25(2):328–373, 2012.

[10] Jens Kober, Katharina Mülling, Oliver Krömer, Christoph H. Lampert, Bernhard Schölkopf, and Jan Peters. Movement templates for learning of hitting and batting. In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA 2010)*, pages 853–858, 2010.

[11] Katharina Mülling, Jens Kober, Oliver Krömer, and Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.

[12] Thomas Röfer, Tim Laue, Judith Müller, Michel Bartsch, Malte Jonas Batram, Arne Böckmann, Martin Böschen, Martin Kroker, Florian Maaß, Thomas Münder, Marcel Steinbeck, Andreas Stolpmann, Simon Taddiken, Alexis Tsogias, and Felix Wenk. B-Human

team report and code release 2013, 2013. Only available online: `http://www.b-human.de/downloads/publications/2013/CodeRelease2013.pdf`.

[13] Thomas Röfer, Tim Laue, Judith Müller, Dennis Schüthe, Michel Bartsch, Arne Böckmann, Dana Jenett, Sebastian Koralewski, Florian Maaß, Elena Maier, Caren Siemer, Alexis Tsogias, and Jan-Bernd Vosteen. B-Human team report and code release 2014, 2014. Only available online: `https://www.b-human.de/downloads/publications/2014/CodeRelease2014.pdf`.

[14] Stefan Schaal. Dynamic movement primitives – a framework for motor control in humans and humanoid robotics. In Hiroshi Kimura, Kazuo Tsuchiya, Akio Ishiguro, and Hartmut Witte, editors, *Adaptive Motion of Animals and Machines*, pages 261–280. Springer Tokyo, 2006.

[15] Cihan Topal, Cuneyt Akinlar, Ling Tian, Yimin Zhou, and Yu Sun. Edge drawing: A combined real-time edge and segment detector. *Journal of Visual Communication and Image Representation*, 23(6):862–872, 2012.

[16] Oliver Urbann and Stefan Tasse. Observer based biped walking control, a sensor fusion approach. *Autonomous Robots*, 35(1):37–49, 2013.