B-Human

Team Report and Code Release 2014

Thomas Röfer[1,2], Tim Laue[2], Judith Müller[2], Dennis Schüthe[2],
Michel Bartsch[2], Arne Böckmann[2], Dana Jenett[2],
Sebastian Koralewski[2], Florian Maaß[2], Elena Maier[2],
Caren Siemer[2], Alexis Tsogias[2], Jan-Bernd Vosteen[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: September 8, 2014

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 six RoboCup German Open competitions and in six RoboCups. We always won the German Open and became world champion four times.

This year, we received the new *Drop-in Player Competition Most Valuable Player Award* both at the RoboCup German Open[1] and at the RoboCup itself. There, we also again won the *Technical Challenges* by placing first in two out of three (the *Any Place Challenge* and the *Open Challenge*) and reaching the second best score in the third one (the *Sound Recognition Challenge*). In the regular soccer competition, we came third by only losing to the new world champion *rUNSWift* from Australia.

The current team consists of the following persons:

Team Leaders / Staff: Judith Müller, Tim Laue, Thomas Röfer.

Students: Michel Bartsch, Jonas Beenenga, Arne Böckmann, Dana Jenett, Vanessa Klose, Sebastian Koralewski, Florian Maaß, Elena Maier, Paul Meißner, Caren Siemer, Andreas Stolpmann, Alexis Tsogias, Jan-Bernd Vosteen, Robin Wieschendorf

Associated Researchers: Udo Frese, 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 will focus a little bit more on the two competitions that we won. The ultimate reference to our system still is last year's team report [8], and this year's report only gives some pointers about what was changed some of which is not described in detail.

Chapter 2 gives a short introduction on how to build our 2014 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 2013 that are not described elsewhere in this document. Chapter 4 describes the approaches we used in the Drop-in Player Competition. Chapter 5 focuses on our

[1] Where it was called *Best Player Award*.

Figure 1.1: The team members of B-Human 2014

contributions to this year's technical challenges. 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.

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 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 [8, Chapter 8.2]

Util contains auxiliary and third party libraries (cf. Sect. 6) as well as our tools, e. g. SimRobot [8, 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 [8, 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 [8, 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.

SpecialActions are the motion patterns (*.mof* files) that are compiled into an internal format using the *URC* [8, Chapter 6.2.2].

URC stands for Universal Resource Compiler and is a small tool for compiling special actions [8, Chapter 6.2.2].

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.120).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, which 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 2013 Update 2 or later
- Cygwin $\geq$ 1.7.9 x86 or later (x86_64 will not work!) (available at `http://www.cygwin.com`) with the additional packages *rsync* and *openssh*. Let the installer add an icon to the start menu (the *Cygwin Terminal*). Add the *...\cygwin\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 1.14.5 for Linux 32 bits* (*naoqi-sdk-1.14.5-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-robotics.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/VS2013/generate.cmd* and open the solution *Make/VS2013/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 platform (Win32[1] or x64), 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. [8, Chapter 8.2]). You may also 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 10.9 or later
- Xcode 5.1 or later
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *NAOqi C++ SDK 1.14.5 for Linux 32 bits* (*naoqi-sdk-1.14.5-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-robotics.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

[1] The platform Win32 is only included since "Edit and Continue" does not work with x64 projects.
[2] Note that the cross compiler builds 32 bit code, although the scheme says "My Mac 64-bit".
[3] Before you can do that, you have to setup the NAO first (cf. Sect. 2.4).

that will restart the software on the NAO after it was deployed. Both the IP address selected 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 key Shift 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 $\geq$ 3.2
- 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 1.14.5 for Linux 32 bits* (*naoqi-sdk-1.14.5-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-robotics.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 current version of the *opennao-atom-system-image-1.14.5.opn* and the *nao-flasher-1.12.5-<your OS>.tar.gz* software from the download area of `https://community.aldebaran.com` (account required). In order to flash the robot, you furthermore 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, *scp*, *sshpass*.

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 a Robot Configuration

Before you start to set up the NAO, you need to create configuration files for each robot you want to set up. To create a robot configuration, run *createRobot*. The 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 is used as host name in the NAO operating system and is saved in the chestboard of the NAO as *BodyNickname*.

Before creating your first robot configuration, check whether the network configuration template files *wireless* and *wired* in *Install/Network* and *default* in *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 with IP xxx.xxx.3.25:

```
./createRobot -t 3 -r 25 Penny
```

Help for *createRobot* is available using the option *-h*.

Running *createRobot* creates all needed files to install the robot. This script also creates a robot directory in *Config/Robots*.

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

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 will be 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
```

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-1.14.5.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 IP address with 169.254.220.1/24[6]. Once the reboot is finished, the NAO will do its usual Aldebaran wake up procedure.

Unfortunately, copying files to the NAO using *scp* freezes the robot's operating system from time to time. Therefore, it is necessary to use the USB flash drive to handle the installation of the B-Human basic system. For more information, have a look at [8, Appendix B].

Unplug the USB flash drive from the NAO. Format this USB flash drive or another one using the FAT32 filesystem. Afterwards mount the USB flash drive and use the *installRobot.usb* script. The script takes the three parameters *name*, *address*, and *path*. *Name* is the name of the robot you have just configured in Section 2.4.2. *Address* is the IP address of the robot you obtain when pressing the chest button of the NAO. *Path* is the root path of your mounted USB flash drive. On Windows, use */cygdrive/<driveletter>* to specify the drive.

For example run:

```
./installRobot.usb Penny 169.254.220.18 /media/usb
```

When running *installRobot.usb*, it will check whether your computer is connected to the robot properly. If it is not, make sure that the network cable is plugged in correctly and restart the NAO by holding the chest button for a few seconds.

Follow the instructions inside the shell. After the installation has finished, the NAO will be restarted automatically. You should now be able to ping the robot using the IP address you configured in Section 2.4.2.

After the first installation of the B-Human basic system using *installRobot.usb*, the robot's operating system should not freeze anymore while copying files via *scp*. Therefore, following installations[7] can be handled using the *installRobot* script, which works quite similar to the *installRobot.usb* script, but without the necessity of an USB flash drive. **Note:** Installing the

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

[6] This is not necessary on OS X if using DHCP.

[7] Re-installation might be useful when updating scripts etc.

B-Human basic system the first time after flashing the robot should always be executed using the *installRobot.usb* script.

For example, to update an already installed NAO, run:

```
./installRobot Penny 169.254.220.18
```

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*)[8], 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*.
-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
-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. [8, Chapter 8.2]) can be used to copy the compiled code to several robots at once.

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

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 an executable (*bhuman*) also loaded at startup.

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 stop *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. If NAOqi is running, this can also be done by pressing the chest button longer than three seconds while also pressing a foot bumper.

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*[9]. 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

[9]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.

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.

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.[10] 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-space to wold-space (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 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.

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

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 [8, 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 those two methods, it is necessary 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. Followed by a

```
get representation:JointRequest
```

and a set of the returned *JointRequest*.

When the calibration is finished it should be saved

```
save representation:JointCalibration
```

and

```
set representation:JointRequest unchanged
```

should be used so the head can be moved when calibrating the camera.

Manually Adjusting Joint Offsets

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 [8, Chapter 8.1.4.5]

```
vd representation:JointCalibration
```

which is more comfortable. The units of the offsets are in degrees.

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*:

```
mr JointCalibration 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 radian.

Straightening Up the NAO

Note: after the camera calibration it is not necessary to set the *JointRequest* for this step.

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

```
get representation:CameraCalibration
mr JointCalibration JointCalibrator
get module:JointCalibrator:offsets
```

Copy the value of *bodyTiltCorrection* (representation *CameraCalibration*) into *bodyRotation.y* (representation *JointCalibration*) and *bodyRollCorrection* (representation *CameraCalibration*) into *bodyRotation.x* (representation *JointCalibration*). Afterwards, set *bodyTiltCorrection* and *bodyRollCorrection* (representation *CameraCalibration*) to zero.

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 [8, Chapter 4.1.2.1] of the robots. Both are explained within the following paragraphs.

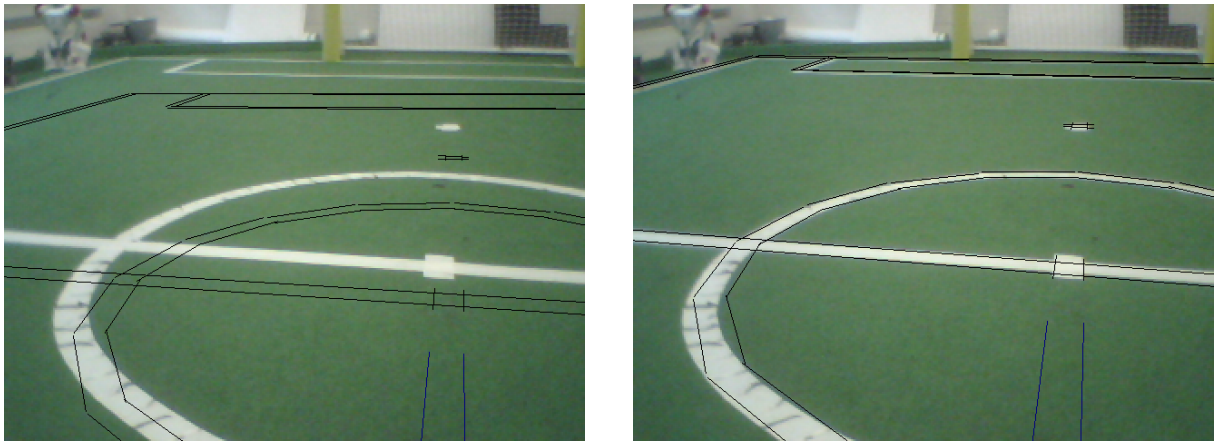
Manual Calibration with the CameraCalibrator

For manual 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 CameraCalibrator* 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 [8, 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).



(a) (b)

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

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* down.[11] The head will then look at that position.

5. Run the automatic calibration process by pressing *Shift+Ctrl+O*[12] 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. 2.1 for a desirable calibration).

Automatic Calibration with the CameraCalibratorV6

For an automatic camera calibration using the module CameraCalibratorV6 (cf. Sect. 3.1), follow the steps below:

1.-3. Same process as for the CameraCalibrator. But instead of typing *CameraCalibrator* use *CameraCalibratorV6*.

4. To start the point collection use the command *dr module:CameraCalibratorV6:start* and wait for the output "Accumulation finished. Waiting to optimize...". The process includes both, the upper and lower camera.

5. Run the automatic calibration process using *dr module:CameraCalibratorV6:optimize* and wait until the optimization has converged.

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

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 14ms. Be aware that high exposures lead to blurred images.

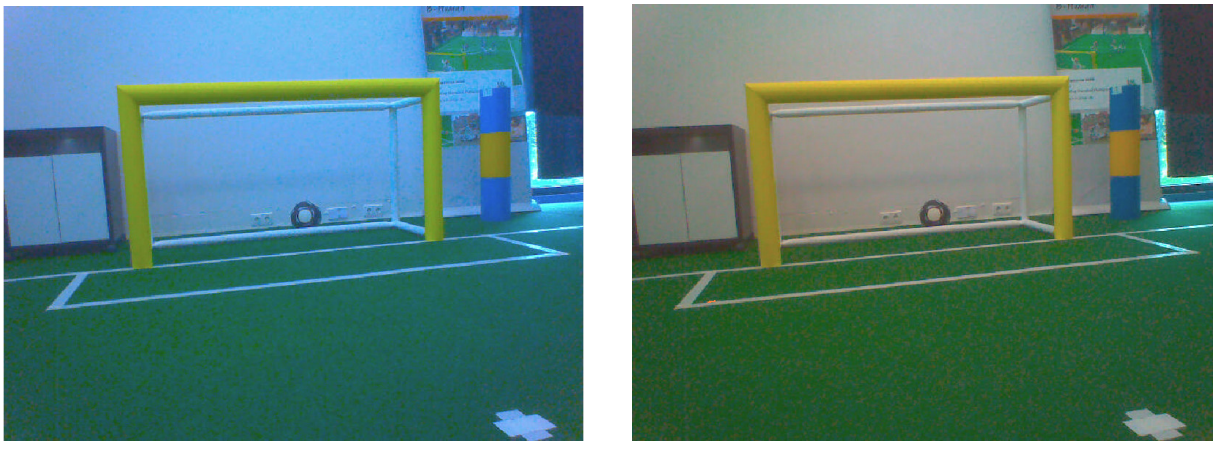
gain: The gain used. The available interval is [0, 255]. Usually the gain is set to 50 - 70. Be aware that high gain values lead to noisy images.

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

The new camera driver (cf. Sect. 3.6) can do a one-time auto white balance. This feature can be triggered with the commands:

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



(a) (b)

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

After setting up the parameters of the camera driver, the parameters of the color classes must be updated (cf. [8, 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 [8, 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 altered without causing recompilation. All the files that are used by the software[13] are located below the directory *Config*.

Besides the global configuration files there are specific files for each robot. These files are located in *Config/Robots/<robotName>* where *<robotName>* is the name of a specific robot. They are only taken into account if the name of the directory matches the name of the robot where the code is executed on. In the Simulator, the robot name is always "Nao". On a NAO, it is the name stored in the chestboard.

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*.

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/<robot name>/<filename>*
2. *Config/Robots/Default/<filename>*
3. *Config/Locations/<current location>/<filename>*
4. *Config/Locations/Default/<filename>*
5. *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. [8, Chapter 3.3.5], but note that the syntax has changed).

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

Chapter 3

Changes Since 2013

In this chapter we describe changes made to our system that are not directly related to the Drop-in Player Competition or the Technical Challenges.

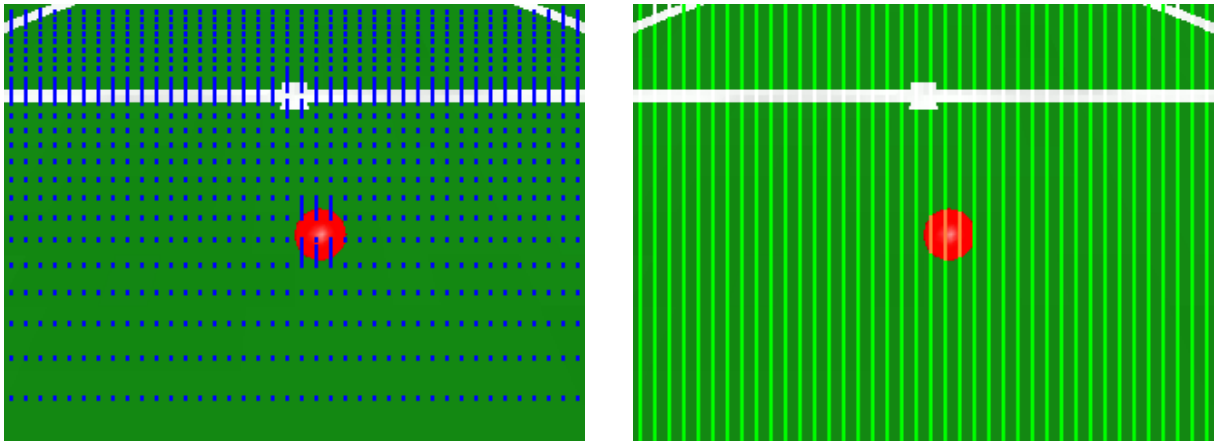
3.1 Camera Calibration

Calibrating the camera with the existing CameraCalibrator module is a very time-consuming task, since one has to mark points on the actual field lines so that the module can optimize the camera orientations based on those points and the internal model of the field (see Sect. 2.8.3). The module CameraCalibratorV6 is based on the CameraCalibrator module, but it collects points on the field lines fully autonomously. The points on the lines required are provided by the LineSpotProvider (see Sect. 3.2.4). They are collected for both cameras, after the head moved to predefined angles. Although this new calibrator significantly reduces the time needed for a calibration, it has a drawback in terms of precision. The line detection has its problems in detecting lines that are further away, in particular when the color calibration is not very good. This can lead to inaccurate values for the estimated tilts of the cameras.

3.2 Vision

When our previous vision system was introduced in 2009, a single module called the Regionizer segmented the image along vertical scan lines in a fixed resolution. Most other perception modules relied on the results of this scanning, only the goals were recognized using a different approach. Over the years, the vision system was extended several times to detect new features in the images. In the past two years, more and more modules scanned the image by themselves instead of using the result of the Regionizer, because they needed a different sampling scheme or a different way to handle image noise. While the Regionizer contributed less and less to the overall image processing, it still required most computation time of all modules. Therefore, it needed to be replaced, preferably with an approach that mimics the scanning already used by the other modules, so that they can again use the result of this new module rather than scanning the image by themselves.

As of the time of writing, the migration to the new ScanlineRegionizer is not completed. While the detection of the field boundary, field lines, and seeding points for the ball recognition are based on the new representation *ScanlineRegions* that it provides, the RobotPerceptor and the GoalPerceptor still use direct image access. The GoalPerceptor has always used direct image



(a) (b)

Figure 3.1: a) The blue spots mark the pixels that have been analyzed by the ScanlineRegionizer. The vertical sample rate decreases with increasing distance to the robot. When colors change, each pixel is analyzed to detect the precise position of the edge between the regions. b) The resulting regions.

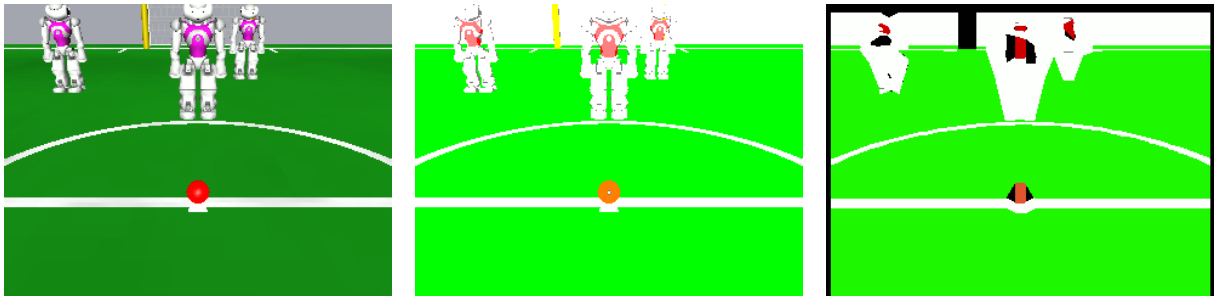
access and will likely remain this way, because it scans horizonatally rather than vertically. The RobotPerceptor might be migrated in the future, however there is no pressing need to do so right now because the removal of the Regionizer and the rewrite of the line detection have freed up several milliseconds of runtime.

The following subsections describe the new vision modules in more detail.

3.2.1 Regionizing the Image

The ScanlineRegionizer scans along vertical scanlines the distance of which is close enough to detect the field boundary, field lines, and in most cases also the ball. These scanlines are segmented into regions, i. e. vertical line segments of a similar color. The segmentation is based on a color table, i. e. the mapping of YC_BC_R color values to a small set of symbolic color classes. Although the scanlines are vertical, the image is traversed horizontally proceeding with all scanlines in parallel from the bottom of the image up to the horizon, i. e. the line in the image that shows the part of the world that is on the same height as the camera itself, or the top of the image, whatever comes first. Processing the image horizontally rather than vertically better complies with the way the processor accesses the memory. In vertical direction the sampling frequency is not constant. The steps depend on the vertical orientation of the camera and basically correspond to a distance of a bit less than the expected width of a horizontal field line at that position in the image. Whenever the color classes of two successive scan points on a vertical line differ, the region in between is scanned to find the actual position of the edge between the two neighboring regions. The best position is the one, above which enough pixels are found that are classified as color classes different from the one of the previous region. A similar approach was used last year to detect the field boundary. Figure 3.1 visualizes the subsampling.

The subsampling obviously leads to a certain amount of data loss. However, it is necessary to cope with the limited processing capability of the robot. Figure 3.2 shows the difference between an original image and one that has been reconstructed based on the regions provided by the ScanlineRegionizer.



(a) (b) (c)

Figure 3.2: Comparison between the original image (a), the color classified image (b), and a reconstruction of the image based on the *ScanlineRegions* (c). The image is reconstructed by filling the parallelograms between adjacent regions of the same color. The black areas are areas in which the neighboring regions do not have the same color and the top black area is omitted because it is above the horizon. The data loss is obvious but the relevant objects are still recognizable.

3.2.2 Detecting the Field Boundary

Since the *ScanlineRegions* are now created in a similar ways as the *FieldBoundary* was last year, no extra scanning of the image is required anymore, which frees up about one millisecond of computation time. Since the detection of the ball (cf. Sect. 3.2.3) and the field lines(cf. Sect. 3.2.4) should only take place inside the area of the field, the *ScanlineRegions* are clipped by the *FieldBoundary* resulting in a representation called *ScanlineRegionsClipped* that is used for further processing. Since clipping away the area outside the field also means to remove the regions that are potentially cluttered the most, ignoring them in further processing also benefits the computation time.

3.2.3 Detecting the Ball

The *ScanlineRegionsClipped* are searched for orange regions indicating a possible ball. Due to the fixed horizontal sampling rate of the ScanlineRegionizer, orange regions that are far away might not be recognized, because a distant ball is smaller than the distance between two scanlines. To be able to detect such balls, the BallSpotProvider scans the image between the scanlines for orange pixels as well. All orange regions found are provided as *BallSpots* to the BallPerceptor for further analysis.

This year, the actual detection of the ball makes use of the full resolution of the image, which increases the number of pixels that can be used to, e. g., judge the shape of a ball candidate. Normally, we only use every second row of an image, because we group together two horizontally neighboring 4:2:2 YC_BC_R pixel as a single 4:4:4 YC_BC_R pixel and we want to keep the pixels quadratic. However, the ball detection assumes 2×2 4:4:4 YC_BC_R pixels instead, which is a real doubling of the vertical resolution, but only a virtual one in horizontal direction, because the two neighboring pixels still share their C_B and C_R values.

3.2.4 Detecting Field Lines

The line detection analyzes the *ScanlineRegionsClipped* to detect field lines, line crossings, and the center circle. It can be split up into four distinct tasks.

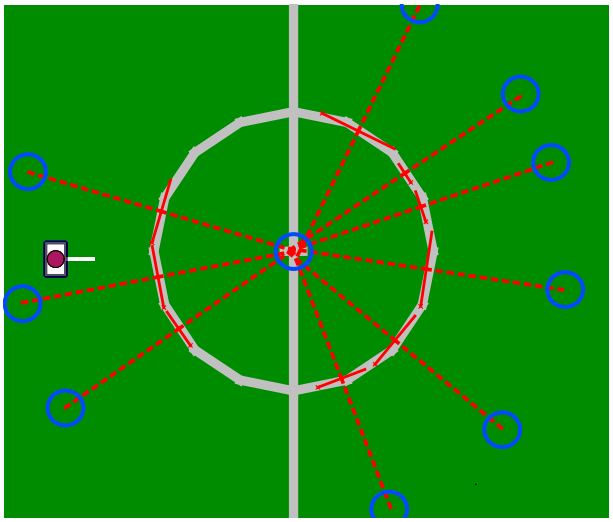


Figure 3.3: Clustering of the field lines to detect the center circle. The red lines depict detected field lines. The dashed lines show the normals of these field lines. The blue circles mark clusters.

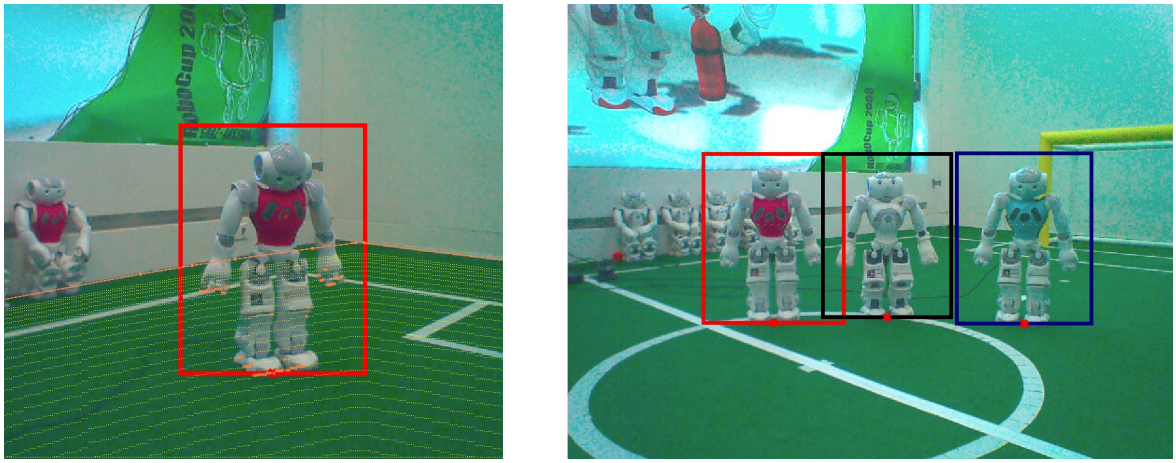
First, all regions that are loosely *line shaped* are selected. A region is line shaped, if it is white and borders on green regions, has roughly the height that a field line would have at the current image location, and does not overlap with the area around a detected robot (cf. Sect. 3.2.5).

Second, the line shaped regions are combined using a simple greedy algorithm that iteratively enlarges a given line and is applied repeatedly until all lines with a minimum of four points have been found:

1. Select two regions from neighboring scanlines, the centers of which can be connected with a nearly white line, i. e. most of the pixels on the line between the centers of the two regions have to be white.
2. Fit a straight line through the two center points.
3. Search among the selected regions on neighboring scanlines for regions that can be connected to the outermost regions of the line with a nearly white line.
4. Update the line parameters using linear regression after two additional points have been found.
5. Search again to find all remaining fitting regions.
6. Update the line parameters again after all fitting points have been found to get the final line.

Third, the variance of the region height is calculated for each line. A large variance indicates that some of the regions on the line might actually be inside an undetected robot or another big white object. If the variance is bigger than a pre-defined threshold, the line is assumed to be a false-positive and is discarded.

Finally, the crossings and the center circle are detected. Whenever two lines projected to the field intersect under a nearly perpedicular angle, a crossing has been found. The type of the crossing is derived from the proximity of the end points of the two lines from the intersection point. If one end point of both lines is close to the intersection, it is classified as an L crossing.



(a) (b)

Figure 3.4: It is scanned for a robot below the boundary of the field (a). Jerseys are found on robots (b): red, unknown, blue.

If only one point is close, it is assumed to be a T crossing. If no end point is close, it is marked as an X crossing.

The center circle is detected using a two-step algorithm. The first step clusters the lines into groups. Each group represents a possible circle. In the second step, linear regression is used to fit a center circle into each cluster. If the fitting error for one of the clusters is small enough, it is accepted as being the actual center circle.

To cluster the lines, the normal is calculated for each line and it is normalized to the radius of the center circle. The lines are clustered based on the distance of the end points of the normals to each other using hierarchical agglomerative clustering (cf. Fig. 3.3).

3.2.5 Robot Detection

Our new robot detection module recognizes standing and fallen robots up to seven meters away in less than two milliseconds of computation time. This is an improvement compared to the module we used last year, which had a lower range and a longer computation time. In addition, the new module scans for jerseys on the robots to get their team color.

Initially, the approach searches for robot parts within the field boundary, which is marked as orange line in the picture (cf. Fig. 3.4a). From there on it scans down in vertical lines with growing space between the pixels looking at. The space and its growing rate are calculated from the distance of the corresponding points on the field. Every pixel scanned this way is marked yellow in the picture. If there are a couple of non-green pixels scanned in a vertical line, then the lowest of these pixels gets marked with a small orange cross. These are possible spots at the feet and hands of a robot, and they can be merged to a bounding box. After calculating the surrounding box, the method searches for a jersey within it and sets the color of the box to red or blue if successful (cf. Fig. 3.4b).

3.3 Obstacle Model

The joint obstacle model creates an Extended Kalman Filter for each obstacle. All obstacles are held in a list. The position of an obstacle is relative to the position of the robot. Visually

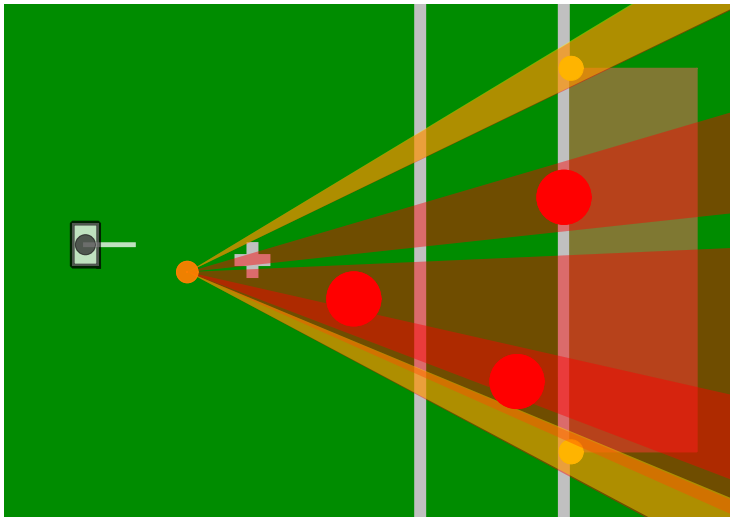


Figure 3.5: The robot on the left models the ball (orange), three other robots (red), and the goal posts (yellow). It determines the sectors between the ball and the goal blocked by obstacles and the ones that are free, i. e. the possible score directions.

recognized obstacles (e. g. goal posts, robots) are transformed from the image coordinate system to field coordinates. Contacts with arms and feet are interpreted as an obstacle somewhere near the shoulder or foot in field coordinates. There is already an obstacle model based on ultrasound measurements that provides estimated positions on the field [8]. In the prediction step of the Extended Kalman Filter, the odometry offset since the last update is applied to all obstacle positions. The estimated velocity of an obstacle is also added to the position. The update step tries to find the best match for a given measurement and updates that Kalman sample. If there is no suitable sample to match, a new sample is created. In this step, we ensure that goal posts have a velocity of 0 mm/s in x and y directions. Due to noise in images and motion, not every measurement might create an obstacle if no best match was found. To prevent false positive obstacles in the model, there is a minimum number of measurements in a fixed duration required to generate an obstacle. If an obstacle was not seen for a second, its "seen" counter is decreased. If a threshold is reached, the obstacle is deleted.

3.4 Score Directions

Our robots are constantly aware where they have to go to be in a position to potentially score a goal. If further away from the opponent goal, the center of the goal is always targeted, because this gives the biggest chance to actually hit the goal given that kicks cannot be 100% precise. When being closer to the opponent goal, obstacles such as goal posts, the goal keeper, and also field players are taken into account, resulting in a number of directional intervals in which a goal could be scored without hitting an obstacle.

Until last year, the best of these intervals was called the *largest free part of the opponent goal*. However, the computation of that information was not compatible with the new vision system (cf. Sect. 3.2) anymore and it also had other shortcomings. The free parts were basically modeled as the segments of the goal line the robot could see, which means that they were evaluated from the perspective of the robot instead from the one of the ball. This is significantly less stable over time, because the perspective constantly changes while the robot approaches the ball. In addition, the largest free part was always the metrically longest, not the one with the largest

angular interval. This makes a difference when the ball is close to the goal, but not in front of the largest free part, because seen from the ball, the largest free part might actually look quite narrow in such a situation. Finally, the precision was rather low, because a one-dimensional grid was used to model the area between the two goal posts.

The new module ScoreDirectionsProvider computes the possible score directions geometrically based on the obstacle model described in Sect. 3.3 (cf. Fig. 3.5). Obstacles, including goal posts both based on perception and self-localization, are modeled as circles. Seen from the ball, a panorama is created that shows which angular intervals to the goal are blocked and which are not using a sweep line method. All obstacles are described as angular sectors surrounding the ball. The opening angle of each sector is determined from the diameter of the obstacle plus the diameter of the ball, modeling the closest direction the ball could still pass the obstacle on each side. The two edges of each sector, i. e. the direction in which an obstacle begins and the direction in which it ends, are added to an array that is then sorted by the angles. After that, the array is traversed while updating an obstacle counter. Whenever a starting edge is encountered, the counter is increased, and when an ending edge is found, the counter is lowered. Whenever the counter is zero, a free sector was found. The resulting intervals are represented as the limiting points of each sector, i. e. the points where the sector touches the closest obstacle on either side. The result is sorted descendingly by the size of the angular intervals, i. e. the first interval is always the largest one. ScoreDirectionsProvider offers an option to give the interval that was previously selected a certain bonus to avoid switching between different intervals in case that they have similar sizes. However, this is currently not used.

3.5 Simulated Ground Truth and Perceptions

The current B-Human framework allows to play full 5 vs 5 matches (with each robot running the complete B-Human software) inside the SimRobot simulation. Unfortunately, most computers do not have enough processing power to carry out such a simulation with decent speed. One particular bottleneck is the generation of simulated camera images that requires graphics hardware that is capable of efficient offline rendering and fast data transfer from graphics memory to the computer's main memory. Another CPU-intensive task is the robot vision itself, as it has to process a huge amount of data and has to be executed 10 times in case of a full game simulation. To overcome these problems, two new modules for generating artificial models and perceptions have been developed.

The OracledWorldModelProvider replaces all image processing and most state estimation modules by accessing the simulation's internal object states and providing perfect and undistorted robot poses, ball models, and obstacle models. A similar functionality (limited to ball and robot pose) already existed in previous code releases as part of the framework.

For more realistic testing, the OracledPerceptsProvider uses the simulation's internal object states for computing artificial perceptions, such as goal posts, lines, line crossings, balls, obstacles, robots, and the field border. This replaces the image processing modules but keeps all state estimation modules, such as self-localization and obstacle tracking, running. For computing the perceptions, each robot's current field of view is taken into account. This allows a much more realistic testing of many behaviors that involve tasks like finding the ball or keeping certain landmarks within the field of view. In addition, a configurable amount of noise as well as the likelihood of false negatives is implemented. Future work might include the consideration of false positives and occlusions.

3.6 B-Human Framework

After the 2013 code release [8], we switched to 64 bit with the desktop side of our software, but also kept the 32 bit support for Microsoft Windows, because Microsoft's Visual Studio lacks some features when dealing with 64 bit applications. In addition, we upgraded to Visual Studio 2013 on Windows, which supports the use of more C++11 features. The *STREAMABLE* macros introduced in the 2013 code release now use C++'s in-place initialization that allows streamable representations to have regular default constructors again. In addition, a similar syntax is now used to define the dependencies of our modules, i. e. their requirements and what they provide. This gives us more flexibility for the module base classes that are generated. Thereby, we were able to drop the central blackboard class without modifying existing modules and replace it with a hashtable-based blackboard that is not a bottleneck in terms of code dependencies anymore.

In addition, this allowed us to simplify our online logger. The latter was also improved in terms of performance. We also introduced downscaled grayscale images for online logging, which again reduces the logging bandwidth required by a factor of two. As an alternative it is now also possible to log full images rarely, e. g., once a second, to store in-game images that, e. g., could be used to improve the image processing system. In general, we logged all games on all robots this year and used the data to improve our code for subsequent games.

The encapsulation of the behavior specification language CABSL that was introduced in 2013 was improved, which makes it a more general means to provide a state-machine to a class. For instance, the logger now also uses CABSL to manage its different processing states.

A new camera driver was implemented, which adds a few more controls and is capable of doing a one-time auto white balance. It can be found in our *BKernel* GitHub repository[1] and is also provided with this code release. Be aware that this driver is not compatible with the NAOqi camera interface, but it is required for the B-Human framework to work out of the box. A detailed description on the control settings can be found in the *README.md* files of the repository.

3.7 GameController

The GameController mainly developed by B-Human team members is used as the official referee application in the SPL since 2013 and in the Humanoid League since 2014. Due to the rule changes for the RoboCup 2014 it was required to implement these rule changes in the GameController. To adapt it to the latest rule changes, we added the official coach interface and support for a substitute player. In addition, the SPL drop-in competition was integrated as a mode with a different set of penalties and neither coach nor substitute. The GameController now also supports *all-time team numbers*, i. e. teams keep their number over the years and do not need to reconfigure it for each competition.

[1] `https://github.com/bhuman/BKernel`

Chapter 4

Drop-in Player Competition

In the Drop-in Player Competition [2], robots from different teams have to play together. They have a common goal, namely to win their matches, but also compete with each other, because being both active and a good teammate counts for the overall score. The main challenge of the Drop-in Player Competetion is that all robots of a team run different software. Therefore, it is not quite clear for each robot what to expect from its teammates. Although the league defines a standard message that is exchanged between the team members, it is not guaranteed that all robots fill in all the fields correctly or whether they use the same semantics for a concept such as the "intention". There always is the risk that a player gets a negative scoring for behaving strangely, just because it was fed wrong information by its teammates.

Our approach for playing in this competition is twofold. On the one hand, we mainly reuse the behavior that our robots also employ in the regular soccer competition and try to reconstruct information our robots usually exchange from the fields available in the standard message. On the other hand, we estimate the reliabilty of our teammates in the sense that we check whether the information they send is consistent with our expectations.

4.1 Communication

In the regular soccer competition, the coordination between our players heavily relies on information that is exchanged as part of the non-standardized section of the `SPLStandardMessage`. Therefore, such information is originally not available in drop-in player games. As a result, we have to reconstruct it from the fields that are available. A part of the reconstruction is done in our encapsulation of the `SPLStandardMessage`, when the message is received. For example the attribute `ballAge` and the position of the ball given in the standard message are converted into the format that B-Human uses. More complex handling is implemented in the module TeamDataProvider. For instance, we estimate the time teammates would need to reach the ball. For this we use several pieces of information. Each player can send a point on the field where it is currently walking to in the standard message. We estimate the time the player would need to walk to that point and add the estimated time the player would need from that point to the ball. Furthermore, teammates with low reliability (see Sect. 4.2) get time added depending on how much we trust them. The time a player needs to get to the ball is important, because it is the main decision-guidance our players use to determine which role they are playing next. In a regular game, our player would send its current role, but in a drop-in game, this is constructed locally from the intention and the estimated time to reach the ball. As in previous years, the code for the role selection as well as the behavior is not part of the code release.

4.2 Reliability of Teammates

As some information given by other players might be imprecise, wrong, or even missing, the reliabilities of all teammates are estimated during a game. These reliabilities are used by the behavior to decide whether we try to carry out some actions all by ourselves or support other robots and let them, for instance, play the ball.

The reliability estimation is done by continuously analyzing the consistency, completeness, and plausibility of the standard messages received. This analysis consists of a number of different tests. Currently, only the content of the most important elements is considered: the ball position and the robot pose.

In a first step, it is checked, whether the robot pose coordinates have been set to valid numbers at all. This is done by buffering the values and checking, whether the value ranges exceed a certain threshold. This approach allows to exclude teammates that do not set this field (i. e. leaving the default value) as well as teammates that – for whatever reason – use meters instead of millimeters as length unit. In addition, to exclude teammates that use degrees instead of radians, the value for the robot's rotation has to be within the interval $[-\pi, \pi]$. If all these numbers appear to be valid, the communicated robot position as well as the ball position must be on the field.

Finally, it is checked, whether communicated ball perceptions are compatible to our own perceptions, i. e. the difference (in absolute field coordinates) between both positions is below are given threshold (which is currently set to 1.2 meters). Only if this check is successful, the other robot has reached the highest reliability status and is treated similar to a teammate in a normal game by the behavior. This state lasts for at least several seconds. However, if both robots disagree about the current ball position, the teammate's reliability becomes decreased again.

For future competitions, we intend to implement even more tests and to cover more elements of the standard message to be able to make more sophisticated distinctions.

However, as described in Sect. 4.4, estimating the teammates' reliability is sometimes only of limited use, as it might occur that teammates do not communicate with us at all.

4.3 Behavior

An aim of our implementation was to support passing between teammates. Therefore, we try to find a good teammate to pass to when our path to the goal is blocked. Which teammate is best for a pass is decided based on its reliability and its position. When we are in a supporting role and a teammate is playing the ball, we try to find a good position to receive a pass. Part of that behavior is to walk while being oriented towards the ball and our playing teammate.

In drop-in games, a player does not always know the position of its teammates. Therefore, sometimes multiple players plan to kick the ball at the same time. This can result in pushing between teammates. In order to not be part of this crowd, we try to visually identify teammates with a close proximity to the ball. The *ExpObstacleModel* (Sect. 3.3) supplies the information needed (for further details see Sect. 3.2.5). Teammates, which are closer to the ball and seem to walk towards it, get a time bonus. At the same time, our player is planning its way more carefully around other players, but still has the intention to play the ball. If the time bonus is up before the teammate kicked the ball, our player uses its usual behavior to quickly reach the ball.

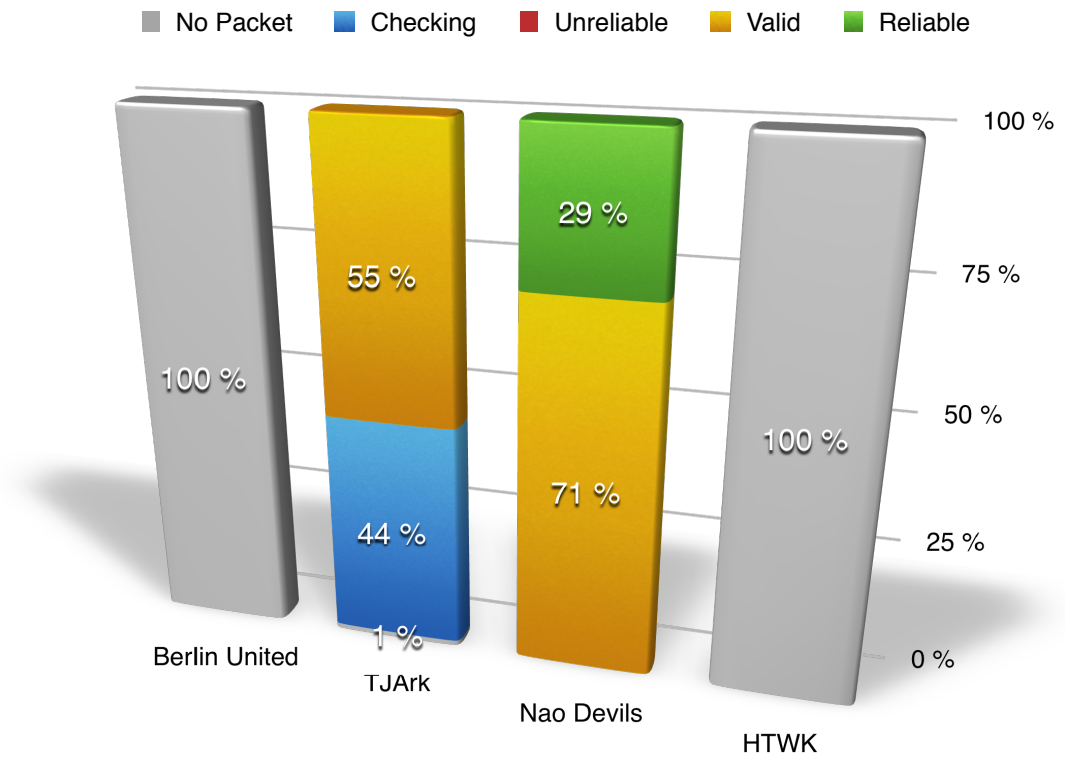


Figure 4.1: The estimate of the reliability of our teammates in the state "playing" of the second half of the game "rUNSWift vs Drop-in All Stars".

4.4 Results

We won both the RoboCup drop-in player competition and the RoboCup German Open drop-in player competition. Our player was well integrated into the team, received passes, and was convincing in the role on the ball. The scores of both the goal difference as well as the vote of the jury were best. In our games, we had some reliable teammates, which made it possible to show team-play. Nevertheless, the communication between teammates still needs to be improved to reach the level of a regular game.

As winner of the Drop-in Player Competition we also participated in the *All Stars Drop-in Player Game* that was played against rUNSWift, the reigning world champion (rUNSWift:Drop-in All Stars 4:2), where we scored both goals for the All Stars team. Figure 4.1 shows the reliabilty our player estimated for its teammates during the second half of that game. Unfortunately, we did not receive any packets from two of our teammates, i. e. from Nao-Team HTWK and Berlin United. Their packets either never arrived or were rejected because they did not comply to the specification of the `SPLStandardMessage`. The other two robots were considered as valid teammates most of the time, i. e. both sent packets that contained data within the expected specifications. Unfortunately, the player from TJArk remained in a checking state for a long time. This state should indicate that a communication exists, but not enough packets arrive for making a reasonable estimate. In fact, the long duration resulted from a bug in our system in combination with no packets from TJArk during the "ready" state. Based on our estimations, the best teammate was the player of the Nao Devils, because there were many situations, in which our player shared the same belief about the state of the world as their player.

Chapter 5

Technical Challenges

In this chapter, we describe our contributions to the three Technical Challenges of the 2014 SPL competition, i. e. the Open Challenge, the Any Place Challenge, and the Sound Recognition Challenge. We won the two former challenges and achieved the second highest score in the latter. These results lead to the win of the overall challenge competition [1].

5.1 Open Challenge: Sorry, No Humans Allowed – Robots Refereeing Robots

In all RoboCup soccer leagues that use robots, refereeing is still performed by humans. However, some situations that referees have to decide upon are, at least partially, already detected by the players themselves. For instance B-Human's robots estimate where the ball has to reenter the field after it was kicked out and they also know when the ball is free after a kick-off. Therefore, it makes sense to think about which situations could be detected by robots and to start implementing robot referees.

Our Open Challenge contribution was the implementation of a robot head referee and its two assistants (see Fig. 5.1). The presentation involved these robots as well as field players of different teams. There was no communication between the referee robots and the robots playing. The players provoked some situations that the referees should identify, i. e. *Illegal Defender*, a *Fallen Robot*, and a ball kicked out or into a goal.

5.1.1 New Roles: Referee and Referee Assistant

In order to realize refereeing robots, we used human soccer matches as a model. In general, a game has a head referee and two assistants. Therefore, we created two new roles within the code, the referee and the referee assistant. The referee is the head of the referee team and follows the ball to identify relevant situations. He pronounces his decisions with a lifted arm and a short verbal message.

Unlike the head referee, a referee assistant stays at the same point on the sideline on one field side during the whole match and looks around. He can recognize the same situations as the head referee but doesn't pronounce them. Instead, he communicates them to the head referee.

In order to make the decision of the referees robust against outliers, a situation has to be seen several times before the referees declare them as seen. A ring buffer is used to store the last 20 outcomes for checking, if a situation occurred. If a referee detects a situation (e. g. the ball was

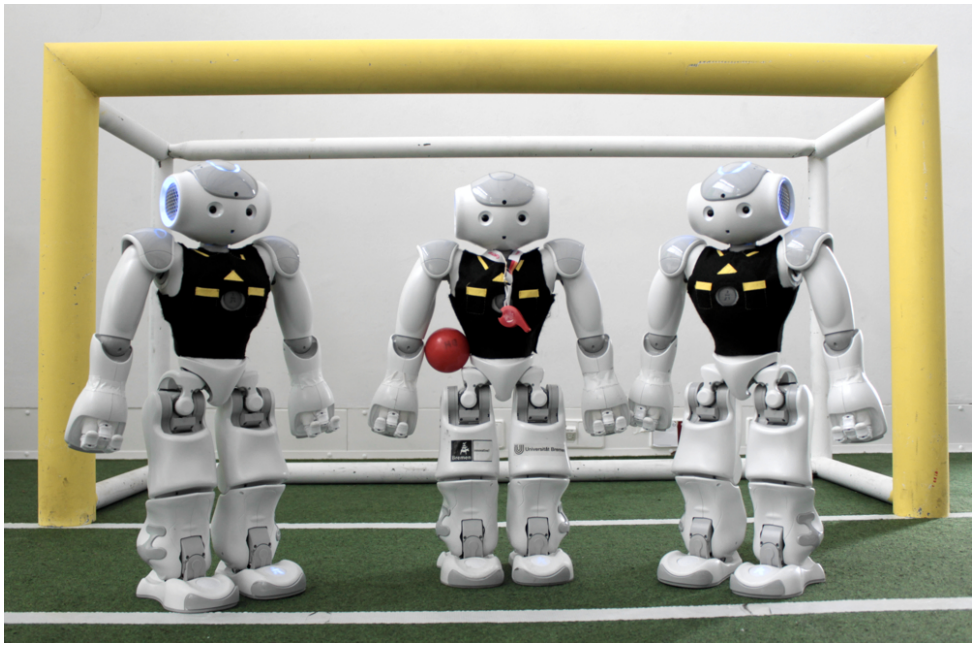


Figure 5.1: The head referee robots and the two assistants

kicked out) he adds this to the associated buffer and checks if a significant part of the last 20 frames included the same detection. Only if this is the case, he pronounces this decision or, if he is an assistant, passes it to the head referee. In the current implementation, the head referee does not validate the information that he receives from his assistants but only pronounces them.

5.1.2 Key Frame Motion

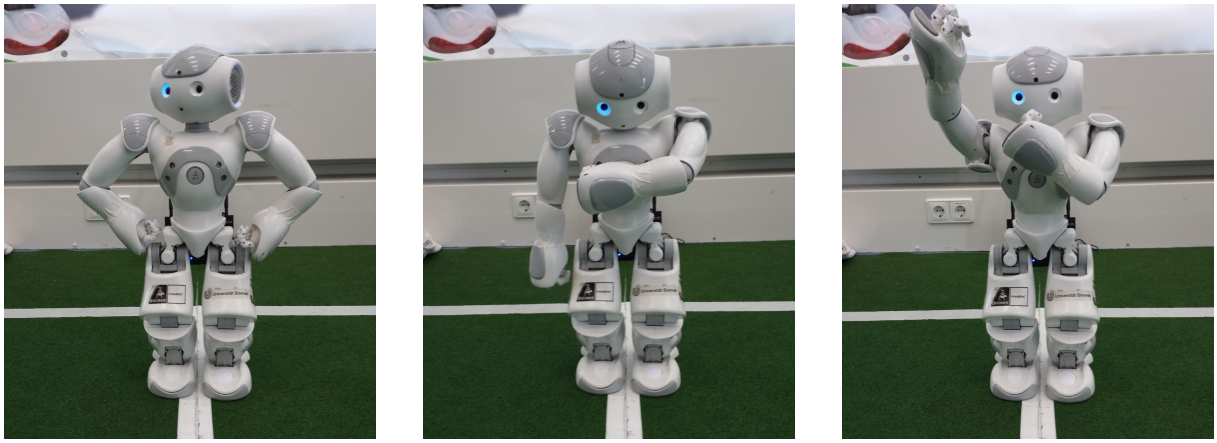
To start the game, the head referee performs a new kick-off key frame motion: At first, he puts his arms at his waist (see Fig. 5.2a). After this, he looks at a clock on his wrist (see Fig. 5.2b). Finally, he raises his right arm while whistling (see Fig. 5.2c).

5.1.3 Positional Play of the Referee

The way a robot referee walks over the field is based on human referees in real soccer matches. The positions and walking paths of the head referee and his two assistants are displayed in Fig. 5.3a. The two assistants are located on opposing field sides (red circles in the picture). The referee walks on a smooth diagonal curve, often called “lazy S”, depending on the current ball position. Therefore, all relevant events are always taking place between him and one of the assistants, so all situations can be seen from two perspectives. Thus, it is less likely that none of them has seen a relevant situation.

The head referee follows the ball’s x-position on the curve which is shown in Fig. 5.3b. The intersection between the yellow dotted line and the diagonal red curve (see white point in Fig. 5.3b) indicates the position of the head referee if the ball lies in the shown position. While the ball is within the center circle, the walking curve is interrupted (see Fig. 5.3b) in order to avoid interferences with the field players.

The recognized ball position of the robot is not precise and is occasionally distorted. While the ball is within the center circle, small deviation result in a large position changes. To prevent such an oscillation between the two field sides, the head referee’s position for a ball within the



(a) (b) (c)

Figure 5.2: The referee whistles to start the game.

circle is always on the same point. It is shown in Fig. 5.4a as a white point on the blue center circle.

5.1.4 Refereed Rules

Currently, a small subset of the SPL rules has been implemented for the robot referees.

5.1.4.1 Illegal Defender at Kickoff

The *Illegal Defender* that our referees detect is the one in the kickoff situation. If one robot of the defending team gets into the center circle too early, the referee recognizes it.

To realize this, the head referee keeps the time after he initiated the kickoff. Then he checks within the following ten seconds if a robot of the defending team gets into the center circle. For recognizing the robots, the new obstacle model is used (see 3.3). After the ten seconds, this check is inactive.

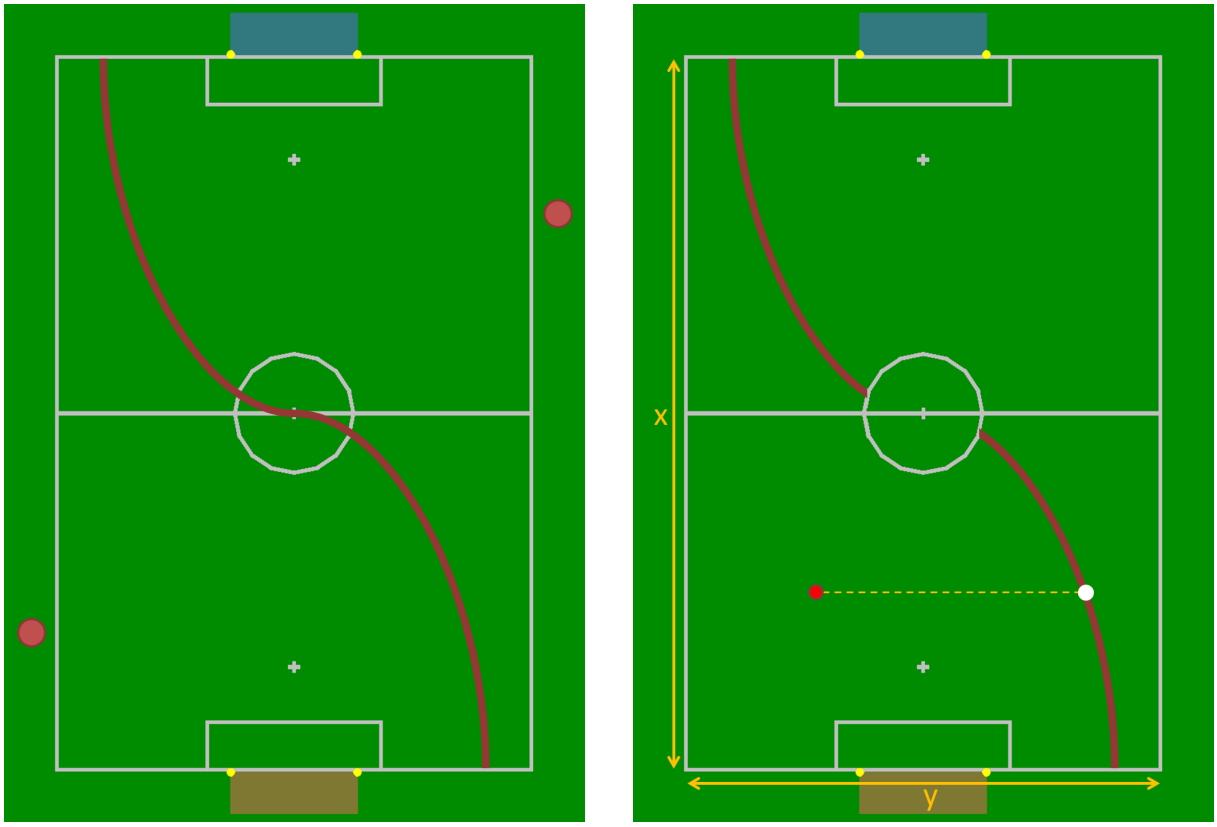
5.1.4.2 Fallen Robot

The referees are able to detect fallen robots but cannot distinguish between a red or blue robot. Just like for the identification of an *Illegal Defender*, the new obstacle module is used for the detection of fallen robots.

5.1.4.3 Ball out

To play soccer, the robot has to detect the ball and know its position on the global field. These abilities can be reused to identify a ball leaving the field or entering a goal.

If the ball is outside the field but not inside a goal (see purple areas in Fig. 5.4a), the referees can assume that the ball is out. But they do not recognize which team shot the ball out.



(a) (b)

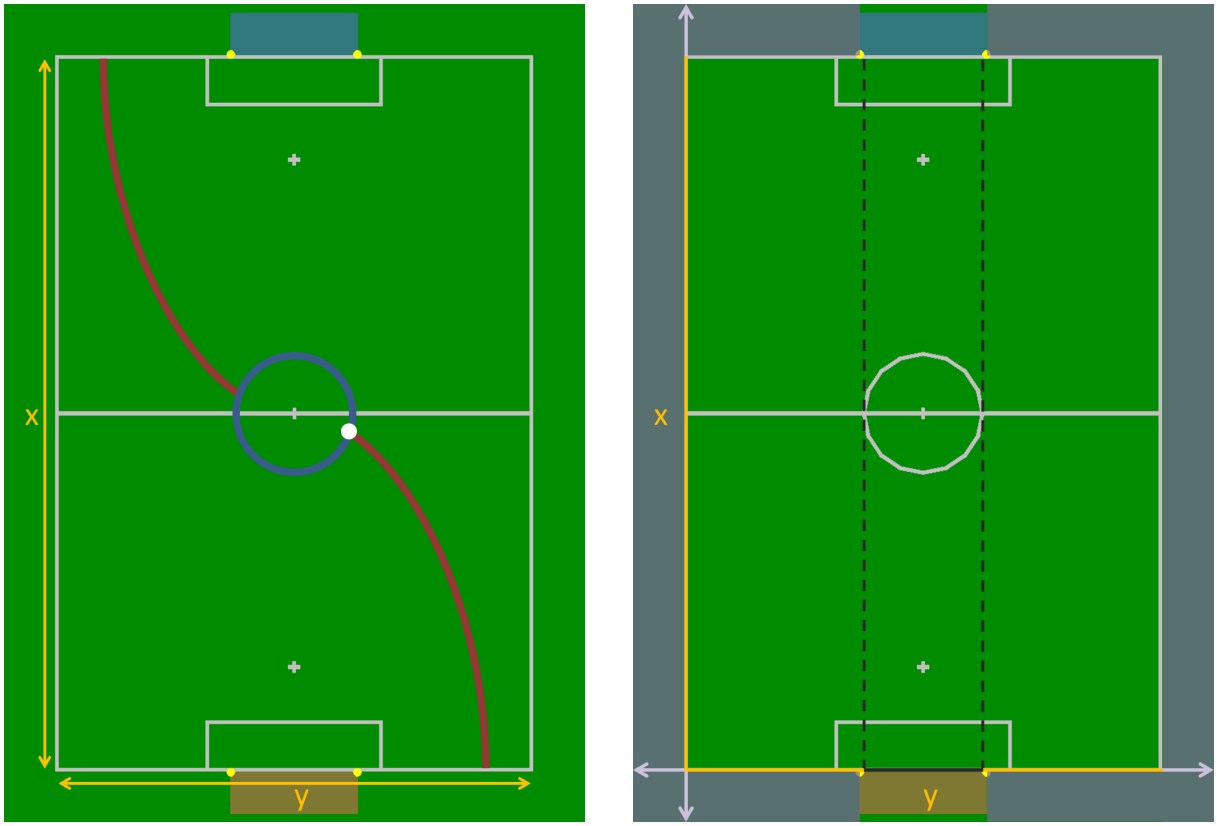
Figure 5.3: The referee "lazy S" diagonal run and the assistant's positions (a) and the head referee's position for an example ball position (b)

5.1.4.4 Goal Scored

A ball outside the field but inside a goal's y-interval (see black lines in Fig. 5.4a) is detected as a scored goal. For this situation, the referees can distinguish if the goal was scored by the red or blue team. This is because the referees have to locate themselves on the field and are therefore aware of the team color of the currently seen goal. Using this information, they can decide which team receives the goal.

5.1.5 Result at the RoboCup

For our presentation the head referee started the game with the kickoff key frame motion. After this, we showed the detection of an *Illegal Defender*. As B-Human's field players are programmed to not take illegal positions, this situation was initiated manually. The head referee detected the *Illegal Defender* and the performing robot was penalized. Subsequently, we demonstrated the walking path of the head referee. Afterwards, one field player was unpenalized and started playing. He shot a ball outside the field and after a short recognition time, the assistant next to the ball detected the ball out. The assistant transmitted this to the head referee who announced it. Next was a successfully detected *goal by blue*. Within the presentation, the referees were unable to detect two ball outs. The first has not been seen by the head referee because he was too close to the field player which blocked the referee's field of view. A similar issue caused the second fault. The ball passed the assistant too close and was therefore not seen.



(a) (b)

Figure 5.4: The head referee position for a ball within the center circle (a) and coordinates to detect the ball outside the field or inside the goal (b)

Despite some problems in our presentation, we won this challenge. We have shown that it is possible to implement referee robots and let them decide some typical game situations.

A video of the presentation is available online[1].

5.1.6 Outlook

There is still a range of improvements necessary to let robots refereeing robots completely. The most important enhancement would be the identification of the field player numbers. This would make it possible to reference announcements to the specific robot that caused it. Furthermore, it would be essential to recognize more situations than the currently implemented ones. To ensure a fluent game, the referees should decide faster if a situation occurred. The tracking of a ball has to be improved to avoid occluded situations. Another important requirement would be an interface to the game controller. This would enable the head referee to announce its decisions directly.

In the current implementation, the referee assistants stayed at the initial start points for the presentations. For a real SPL game, the assistants should follow the ball on the sideline of their field side. Furthermore, the head referee has to avoid ball contact to avoid interventions into the game.

Easily realizable changes would be the tracking of game time and the announcement of halftime

[1] `https://www.youtube.com/watch?v=x6HPlTdGlJA`

and of the end of game. Additionally, at the moment when a penalizing robot can return to the game the referee could annouce it, too.

5.2 Any Place Challenge

This challenge was set up to find out, whether it is possible for the NAO to play soccer in a previously unknown environment, given only a standard SPL goal and ball. No alteration of the robot's software was allowed once the participating teams arrived at the competition's location. The detailed rules can be found in [3].

5.2.1 B-Human's Normal Vision System

As is described in Sect. 3.2, the vision system that we use in the normal soccer competition relies on color lookup tables to classify pixels into a small number of color classes. As most objects have a unique color (goal, ball, field, jerseys), the classification is very efficient and comes close to object detection. However, at the same time, it makes the algorithm susceptible to changes in lighting, since only a single pixel is considered at a time. A global view at lighting conditions is taken only when the lookup table is created through calibration (by a human). Consequently, it fits only the conditions that were present at calibration time.

For detecting the ball and the goal, the two modules BallPerceptor (cf. Sect. 3.2.3) and GoalPerceptor (cf. [8, Chapter 4.1.8]) are used. Both depend directly and indirectly on the color lookup table. For the normal SPL games, this approach is suitable, since the environmental conditions are almost static and the boundary of the green field (cf. Sect. 3.2.2) helps to exclude false positives, e. g. resulting from shoes or human legs, in the environment.

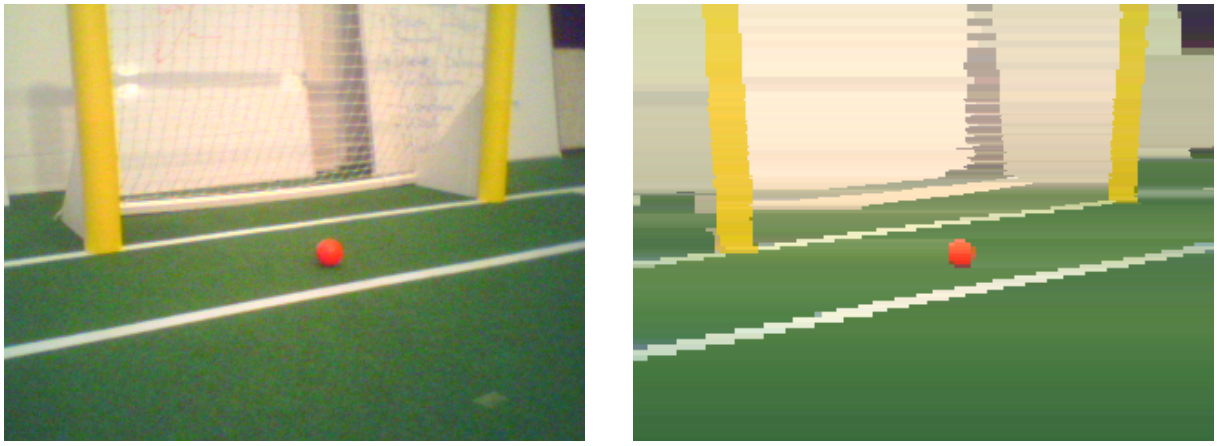
However, for the Any Place Challenge, no field boundary was guaranteed to exist and the lighting conditions have not been known in advance, preventing the preparation of a reliable color lookup table. Therefore, a different vision system had to be used for this challenge.

5.2.2 Vision System Used in the Challenge

We decided to adapt the vision system developed by Härtl et al. [6], as it groups isochromatic regions without any predefined color table and only afterwards uses color information together with shape information to detect objects. Härtl developed this vision system in his diploma thesis [5] as part of the B-Human system. He then went to the University of Miami and joined the RoboCanes team, who have been using the vision system for the past two years.

His approach performs a run length encoding of the camera image creating horizontal runs of similar color. It only uses a subset of the pixels for building the runs considering the perspective of the image (cf. Fig. 5.5), i. e. since close areas are depicted larger than areas farther away, a low resolution is needed for the former and a higher resolution for the latter. For recognizing the ball, the runs are then ranked based on their similarity to both the expected width of the ball at their location in the image and a prototypical ball color. Only a small set of best candidates is then used as starting points for growing regions of similar color that are afterwards checked for being an actual ball. For the goal recognition, runs are added to a one-dimensional Hough space, each entry representing a horizontal position of a potential goal post. Again, only a limited number of best candidates are considered afterwards for growing regions that are checked for being an actual goal post.

We had to adapt the original implementation for the challenge, because it was intended to



(a) (b)

Figure 5.5: a) The original image captured by the NAO. b) A reconstruction of the image after the subsampling and segmentation process [6, Section 2.2].

be used on an SPL field only, i. e., it not only detected field lines, which was not needed for the challenge, but it also expected a green region below the goal posts. Since the system was developed in 2012, it also detected blue goal posts, which had to be deactivated as well, in particular, since sanity checks would discard yellow goal posts if also blue ones (e. g. blue jeans) were present in the same image.

Although the idea of the vision system is to only employ the similarity to prototypical colors as one of the detection criteria, it also uses hard color thresholds to lower the overall number of regions that are considered as candidates for a certain type of object and thereby the overall runtime. For this challenge, we raised the thresholds for yellow and orange to achieve a greater tolerance for the color evaluation. This was necessary, because we could not know what kind of lighting condition we would encounter at the actual challenge location. However, with this change, the NAO sometimes saw balls in goal posts. To solve this problem, we changed the ranking of ball segments, so that the more similar a region is to the yellow color prototype, the lower it gets rated as a potential ball. In addition, we also added an additional threshold that allows discarding balls that are too yellow.

The vision system has originally been developed for the weaker hardware of the NAO V3 robot. Using the newer model with its Atom processor, it was possible to increase the recognition rate by increasing some parameters, e. g., the resolution of the Hough space and number of Hough maxima considered for goal post detection.

5.2.3 Behavior

In addition to the vision system, an adapted behavior was needed for this challenge. First of all, we lowered the maximum walking speed of the robot, because we could not the type of floor the robot would have to walk on. As our normal ball searching behavior relies on more than one searching robot on the whole field and also lacks on speed with a single robot, it needed to be replaced for this challenge. In order to do this, we picked five positions – four near the corners and one at the center of the assumed challenge field – to which the robot walks in random order. The whole ball searching behavior starts with a turn around the robot's own axis which is followed by the walk to one of the five points and then followed by another turn and so on. In case the ball is found during the search, it is immediately interrupted and the normal goal

(a) (b)

Figure 5.6: a) A white goal post in front of a white wall. b) The same scene after the run length encoding step.

kicking behavior of our normal soccer behavior is started.

Another modification was done to the behavior that is started right after the robot is unpenalized, which is usually followed by the normal soccer behavior without any special handling. Since we did not know where the robot will be placed and how it is rotated or if the goal could be seen from that position, the robot first performs a turn around its own axis right after it is unpenalized. This turn is interrupted immediately if a goal was seen. This follows the idea to first find that one cue on the field in order to localize the own position properly. We decided that the robot should first be sure about the own position and then go and find the ball, if it was not already found during the initial turn.

5.2.4 Result at the RoboCup

At the RoboCup, we tested the our implementation in several locations and it always worked. However, since we could not steal an official goal, these tests were limited to the ball recognition, walking, and the behavior. In the actual challenge, B-Human was the only team that scored goals at all. Our robot shot at the goal four times and achieved three hits and one miss.

A video of the presentation is available online[2].

5.2.5 Outlook

As the participation in the challenge showed as well as the fact that it has been used by the RoboCanes for the past two years, the vision can be used for playing soccer without the need of a prior color calibration. However, we will probably not use it in future competitions, because it is currently under discussion to replace the yellow goals by white ones. The current approach requires that the goal posts are surrounded by regions of different color. Since white is quite prominent in RoboCup settings, it is possible that there are white walls behind the goals. As Fig. 5.6 shows, the approach is not well suited for such a situation.

[2] `https://www.youtube.com/watch?v=gHHMANO17gU`

5.3 Sound Recognition Challenge

In this challenge, robots have to recognize different signals based on a *Audio Frequency Shift Keying (AFSK)* as well as the sound of a whistle. As described in the following subsections, we use different approaches for recognizing AFSK signals and the whistle. Both approaches are implemented within the same module (which can be found in our code release as SoundSignalRecognizer) and become executed at the full frame rate of 60 Hz, analyzing sound data from two of the NAO's audio channels. The average execution time of this module is less than 0.3 ms, which appears to be fast enough for a possible use of the sound recognition in actual games.

5.3.1 AFSK Signal Recognition

For recognizing the AFSK, we decided to demodulate the signal incoherent, i. e. without recovering the carrier [10]. We decided to check the frequency spectrum for the incoming signals, i. e. the AFSK signal when it is transmitted and most of the time noise. Therefore, we have to take different aspects into account, which are mainly the sampling frequency with respect to the highest signal frequency, the time domain resolution of the window to be transferred by *discrete Fourier transform (DFT)* (we use the highly efficient implementation by [4]) into the frequency domain, and the window length of the DFT for the frequency resolution. Here we have a conflict between window length in time and DFT window length, as we get a higher resolution of the DFT spectrum with higher window length N, but this leads to a greater time window where the signal length is less than the window length, which is not useful.

The sampling frequency f_s has to be set to more than the doubled highest frequency in the spectrum (Shannon's theorem) $f_s > 2 \times f_{high}$. For our case, we have $f_s > 2 \times 320Hz \rightarrow f_s = 1kHz$. As the NAO microphones' lowest sampling frequency is 8 kHz, we downsample the signal to 1 kHz after we low pass filtered the 8 kHz signal to eliminate aliasing. As low pass filter we use a *Finite Impulse Response (FIR)* filter, which has a constant group delay [9] [7].

For the DFT length, we use approximately a third of the signal length, the rest which is then missing to get a good resolution in the frequency domain is padded with zeros. This so-called *zero padding* can also be used to get a good FFT (Fast Fourier Transform) length, which has to be a power of two. The window used is a sliding window, such that the last samples are also included in the next window frame. The advantage is given by a smoother frequency transition, e. g. from 200 Hz to 320 Hz [7].

Finally, we check the magnitude at the given signal frequencies compared to the overall average magnitude of a set of last spectra as well as to a threshold that denotes the minimum magnitude to consider. In addition, we determine the number of local maxima and the distance of time between them. To distinguish between noise and the signal itself, a configurable number of local maxima (over the magnitude threshold) with configurable maximum and minimum distances needs to be perceived (for an example, see Fig. 5.7). During the challenge at RoboCup 2014, very powerful speakers have been used that allow us to relax most of these parameters and to focus on signals that cause a strong signal magnitude.

5.3.2 Whistle Recognition

For the whistle detection, we use a cross-correlation. To calculate the correlation between a given whistle signal and the actual recorded signal in a fast way, we transform the actual signal into the frequency domain, multiply it with the conjugate complex stored reference whistle signal and transform it back into the time domain, where we detect the compliance in percent from

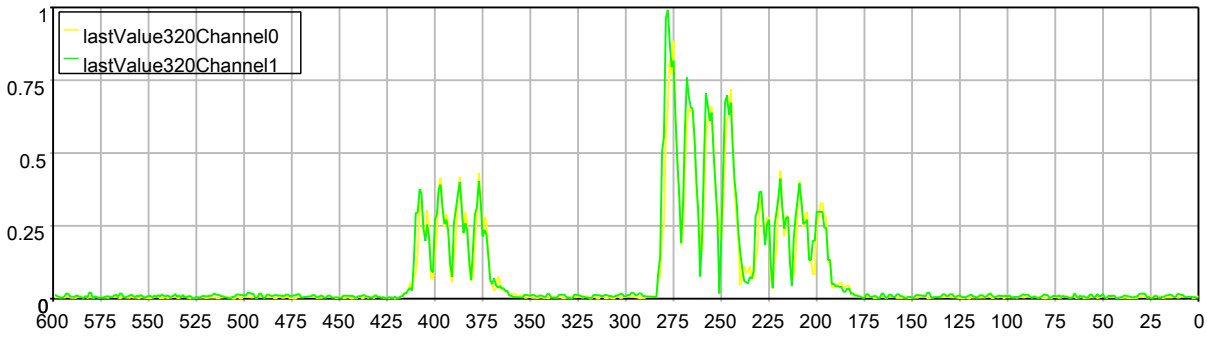


Figure 5.7: Magnitude of 320 Hz signal over 600 frames (equal 10 seconds). During this time, the *stop* signal has been played twice and the *start* signal has been played once. The differently colored lines depict the different results from the two used audio channels.

the reference signal.

Firstly, we need a reference signal s_{ref}. The reference signal is also recorded by the robot, where we take a window length of N-samples. As the whistle's frequency is much higher than the frequencies given in the first task, we can use the sampling frequency f_s of the audio card without downsampling. The signal must be extended with zeros by length N, because the result of the correlation is twice the window length. This signal is transformed to the frequency domain by an FFT.

$$s_{\text{ref}} \circ\!\!-\!\!\bullet \underline{S}_{\text{ref}} = a + jb. \tag{5.1}$$

The underline denotes complex values. This signal is stored as a reference and has not to be recalculated in every step.

For the actual recorded signal s_{act}, we use the same procedure by adding zeros to get the doubled length and transform it into the frequency domain to get $\underline{S}_{\text{act}}$.

We make use of the fact that a correlation can be represented by a convolution using one signal in reorder. The advantage of a convolution is that it can be calculated easily in the frequency domain by just multiplying the signals. As we do not want to store the signal in reverse order, we can transform the reversion into the frequency domain too. So we get

$$s_{\text{ref}}(t) * s_{\text{act}}(-t) \circ\!\!-\!\!\bullet \underline{S}_{\text{ref}}(f) \cdot \underline{S}^*_{\text{act}}(f), \tag{5.2}$$

where $\underline{S}^*_{\text{act}}$ denotes the conjugate complex of the signal.

As it does not matter which signal is conjugated, we store the reference signal as conjugate. Multiplying both signals gives us the result in the frequency domain and the inverse Fourier Transform is the correlation result s_{corr} in time domain.

The best result can be achieved if the reference signal is equal to the actual signal or the negative actual signal. Using this value, which is the autocorrelation value, and dividing the correlation signal s_{corr} by this factor, we always get a percent value from the best expecting result. An example is depicted in Fig. 5.8

If the percentage is above a defined threshold, the whistle is detected. As step by step formulation we have in discrete form the implementation

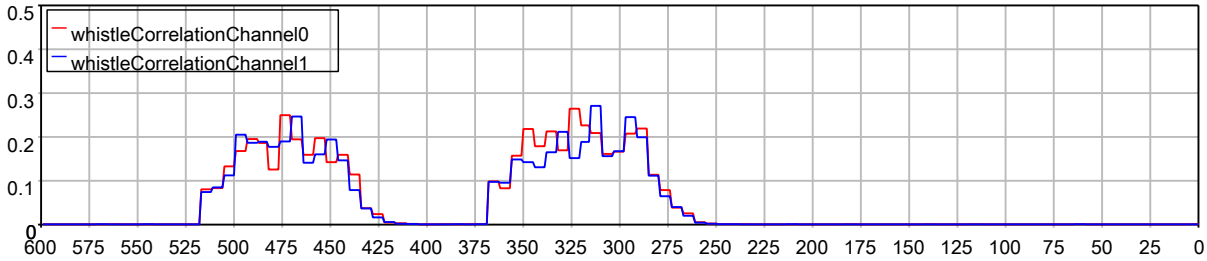


Figure 5.8: Correlation between the current signal and the pre-recorded reference whistle signal over 600 frames (equal 10 seconds). During this time, a whistle has been blown twice. The threshold for whistle detection during this experiment (as well as during the challenge at the RoboCup) has been set to 0.15. The differently colored lines depict the different results from the two used audio channels.

$$\underline{S}_{\text{act}}[k] \bullet\!\!-\!\!\circ s'_{\text{act}}[n] = \begin{pmatrix} s_{\text{act}}[n] & \mathbf{0}_N \end{pmatrix}$$
$$s_{\text{corr}}[n] \circ\!\!-\!\!\bullet \underline{S}_{\text{corr}}[k] = \underline{S}_{\text{act}}[k] \cdot \underline{S}^*_{\text{ref}}[k]$$
$$\varphi_{cc} = \frac{\max(|s_{\text{corr}}[n]|)}{\varphi_{ac\max}} \in (0,1)$$
$$w = \begin{cases} true & \text{for } \varphi_{cc} \geq \alpha \\ false & \text{otherwise} \end{cases}.$$

φ denotes the correlation value for the cross correlation and the auto correlation of the reference signal respectively. The parameters n and k are discrete time and frequency respectively.

5.3.3 Result

In this challenge, we achieved 68 out of 72 possible points. The whistle recognition worked perfect, scoring all 36 points. However, during the AFSK signal recognition phase, two robots each missed to detect one signal, leading to 32 out of 36 possible points.

A video of the presentation is available online[3].

[3] `https://www.youtube.com/watch?v=AGnQqmjnlOY`

Chapter 6

Acknowledgements

We gratefully acknowledge the support given by Aldebaran Robotics. We thank our current sponsors IGUS, TME, Wirtschaftsförderung Bremen (WFB), Sparkasse Bremen, and abat as well as the Deutsche Forschungsgemeinschaft (DFG) for funding parts of our project. Since B-Human 2014 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 used in the Sound Recognition Challenge.
(`http://www.fftw.org`)

getModKey: For checking whether the shift key is pressed in the Deploy target on OS X.
(`http://allancraig.net/index.php?option=com_docman&Itemid=100`, not available anymore)

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.
(`http://dev.libqxt.org/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://qt-project.org`)

qtpropertybrowser: Extends the Qt framework with a property browser.
(`https://qt.gitorious.org/qt-solutions/qt-solutions/source/80592b0e7145fb876ea0e84a6e3dadfd5f7481b6:qtpropertybrowser`)

sshpass: Non-interactive ssh password provider used in the *installRobot* scripts.
(`http://sourceforge.net/projects/sshpass`)

snappy: Used for the compression of log files.
(`http://code.google.com/p/snappy`)

Bibliography

[1] RoboCup Technical Committee. Results2014 – Standard Platform League, 2014. Only available online: `http://www.informatik.uni-bremen.de/spl/bin/view/Website/Results2014#Technical_Challenges`.

[2] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) rule book, 2014. Only available online: `http://www.informatik.uni-bremen.de/spl/pub/Website/Downloads/Rules2014.pdf`.

[3] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) technical challenges, 2014. Only available online: `http://www.informatik.uni-bremen.de/spl/pub/Website/Downloads/Challenges2014.pdf`.

[4] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[5] Alexander Härtl. Robuste, echtzeitfähige Bildverarbeitung für einen humanoiden Fußballroboter. Diploma thesis, University of Bremen, 2012.

[6] Alexander Härtl, Ubbo Visser, and Thomas Röfer. Robust and efficient object recognition for a humanoid soccer robot. In Sven Behnke, Manuela Veloso, Arnoud Visser, and Rong Xiong, editors, *RoboCup 2013: Robot Soccer World Cup XVII*, volume 8371 of *Lecture Notes in Artificial Intelligence*, pages 396–407. Springer, 2014.

[7] Karl Dirk Kammeyer and Kristian Kroschel. *Digitale Signalverarbeitung: Filterung und Spektralanalyse mit MATLAB-Übungen.* Vieweg + Teubner, Wiesbaden, 2009.

[8] 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`.

[9] Daniel Ch. von Grüningen. *Digitale Signalverarbeitung: mit einer Einführung in die kontinuierlichen Signale und Systeme.* Fachbuchverlag Leipzig im Carl Hanser Verlag, München, 2008.

[10] Martin Werner. *Nachrichtentechnik: Eine Einführung für alle Studiengänge.* Vieweg + Teubner, Wiesbaden, 2010.