B-Human

Team Report and Code Release 2018

Thomas Röfer[1,2], Tim Laue[2],
Arne Hasselbring[2], Jannik Heyen[2], Bernd Poppinga[2],
Philip Reichenberg[2], Enno Röhrig[2], Felix Thielke[2]

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

Revision: November 14, 2018

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, B-Human has won eight RoboCup German Open competitions, the RoboCup European Open 2016 competition, and has become RoboCup world champion six times.

After we had claimed the world champion title in two years in a row, we only came second in the *Champions Cup* in Montréal, Canada. However, we again won the *Mixed Team Competition*, in which we teamed up with *rUNSWift* from the University of New South Wales this year and formed the joint team *B-Swift*. We also won this year's *Technical Challenge* again.

The 2018 team consisted of the following persons (most of them are shown in Fig. 1.1):

Students: Andreas Baude, Jan Buschmann, Tryggve Gahrmann, Gerrit Felsch, Jan Fiedler, Marvin Franke, Martin Gerken, Mario Grobler, Paul Luca Habermann, Arne Hasselbring, Jannik Heyen, Markus Ihrig, Jan-Henrik Kasper, Jonah Klöckner, Daniel Krause, Gregor Kuhn, Jonas Kuball, Florian Maaß, Bernd Poppinga, Lukas Post, Philip Reichenberg, Enno Röhrig, René Schröder, Nicole Schrader, Lukas Schulze, Alexander Stöwing, Felix Thielke, Timo Urban, Lars Wimmel.

Active Alumni: Alexis Tsogias.

Leaders: Tim Laue, Thomas Röfer.

Associated Researchers: Udo Frese, Jesse Richter-Klug.

1.2 About the Document

In this document, we give an overview of the changes that we made in our system since last year and provide descriptions of the approaches used in the additional competitions. The most comprehensive reference to our system remains our team report of 2017 [4].

The remainder of this document is organized as follows: Chapter 2 gives a short introduction on how to build the code including the required software and how to run the NAO with our software. The major changes made to the system since last year – in particular improvements of the infrastructure, some of the perception and motions approaches, and our behavior implementation

Figure 1.1: The majority of the B-Human team members for the RoboCup season 2018

– are described in Chapter 3. Finally, a brief description of our participation in the *Penalty Shot Challenge* and the *Mixed Team Competition* is given in Chapter 4.

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 on Windows, Xcode on macOS, or make on Linux, setting up the NAO, copying the files to the robot, and starting the software. In addition, all calibration procedures are described here.

This code release only supports NAO versions 4 and 5. The code will not run on other versions, in particular not on NAO V6. Trying to set up an unsupported NAO with our software will not work and may have negative effects on the robot.

2.1 Download

The code release can be downloaded from GitHub at `https://github.com/bhuman`. Store the code release to any folder. 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, and CodeLite.

Src contains the source code of the B-Human software including the B-Human User Shell (cf. [4, Chapter 10.2]).

Util contains auxiliary and third party libraries (cf. Sect. 5) as well as our simulator SimRobot (cf. [4, Chapter 10.1]).

2.2 Components and Configurations

The B-Human software is usable on Windows, Linux, and macOS. 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 (cf. [4, Chapter 10.2]).

Controller is a static library that contains NAO-specific extensions of the simulator and the interface to the robot code framework. 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 2017 code release [4, Chapter 3.1].

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

Nao is the B-Human executable for the NAO. It depends on *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*, *SimulatedNao*, and some third-party libraries. 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 macOS). *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.

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

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

The different configurations for *Nao* and *SimulatedNao* can be looked up in Tab. 2.1.

	without assertions (NDEBUG)	debug symbols (compiler flags)	debug libs[1] (_DEBUG, compiler flags)	optimizations (compiler flags)	debugging support[2]
Release					
Nao	✓	×	×	✓	×
SimulatedNao	✓	×	×	✓	✓
Develop					
Nao	×	×	×	✓	✓
SimulatedNao	×	✓	×	×	✓
Debug					
Nao	×	✓	✓	×	✓
SimulatedNao	×	✓	✓	×	✓

[1] - on Windows - `https://docs.microsoft.com/en-us/cpp/c-runtime-library/debug`
[2] - See [4, Chapter 3.6]

Table 2.1: Effects of the different build configurations.

2.3 Building the Code

2.3.1 Project Generation

The scripts *generate* (or *generate.cmd* on Windows) in the *Make/<OS/IDE>* directories generate the platform or IDE specific files that are needed to compile the components. The script collects all the source files, headers, and other resources if needed and packs them into a solution matching the system (i. e. Visual Studio projects and a solution file for Windows, a CodeLite project for Linux, and an Xcode project for macOS). 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. 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 10 64 bit or later
- Visual Studio 2017[1] or later
- A Unix base system. There are two alternatives:
  1. Windows Subsystem for Linux (WSL). Execute the PowerShell script *Make/VS2017/installWSL.ps1*. The script will guide through the installation. It will install the Windows Subsystem for Linux (unless it is already installed), then the Windows installer will ask for a reboot of the computer. Then script has to be executed again to download and install "Ubuntu-WSL" with all required packages.
  2. Cygwin x86 / x64 (available at `http://www.cygwin.com`) with the additional packages *rsync*, *openssh*, *ccache*, and *clang*. Let the installer add an icon to the start

[1]Visual Studio 2017 Community Edition Version 15.8 with only the "VC++ 2017 v141 Toolset (x86, x64)" and "Windows 10 SDK" installed is sufficient.

menu (the *Cygwin Terminal*). Add the *...\cygwin64\bin* directory to the beginning of the PATH environment variable (before the Windows system directory, since there are some commands that have the same names but work differently). Make sure to start the *Cygwin Terminal* at least once, since it will create a home directory.

- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-linux32.tar.gz*) the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `https://community.ald.softbankrobotics.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/VS2017/generate.cmd* and open the solution *Make/VS2017/B-Human.sln* in Visual Studio. Visual Studio will then list all the components (cf. Sect. 2.2) of the software in the "Solution Explorer". Select the desired configuration (cf. Sect. 2.2, *Develop* would be a good choice for starters) and build the desired project: *SimRobot* compiles every project used by the simulator, *Nao* compiles every project used for working with a real NAO, and *Utils/bush* compiles the B-Human User Shell (cf. [4, Chapter 10.2]). Either *SimRobot* or *Utils/bush* can be selected as "StartUp Project".

2.3.3 Xcode on macOS

2.3.3.1 Required Software

The following components are required:

- macOS 10.13 or later
- Xcode 10 or later
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-linux32.tar.gz*) the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `https://community.ald.softbankrobotics.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/macOS/generate*.[2] Open the Xcode project *Make/macOS/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[3]

[2] Xcode must have been executed at least once before to accept its license and to install its components.

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

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.

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).[4] 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 `-b` is provided by default, which will restart the B-Human 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/Includes/connect.con* that is also written by the *login* script and used by the *RemoteRobot* simulator scene. The options are stored in *Make/macOS/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.

2.3.3.3 Support for Xcode

Calling the script *Make/macOS/generate* also installs various development supports for Xcode:

Data formatters. If the respective file does not already exist, a symbolic link is created to formatters that let Xcode's debugger display summaries of several *Eigen* datatypes.

Source file templates. Xcode's context menu entry *New File...* contains a category *B-Human* that allows to create some B-Human-specific source files.

Code snippets. Many code snippets are available that allow adding standard constructs following B-Human's coding style as well as some of B-Human's macros.

Source code formatter. A system text service for formatting B-Human code is available to be used from Xcode's menu *Xcode→Services*.

2.3.4 Linux

The following has been tested and works on Ubuntu 18.04 64-bit. It should also work on other Linux distributions (as long as they are 64-bit); however, different or additional packages may be required.

2.3.4.1 Required Software

The build has been tested using the software versions provided by the current Ubuntu distribution repositories. Earlier versions of, e. g., clang may work, but are untested.

Requirements (listed by common package names) for Ubuntu 18.04:

- clang
- make
- qtbase5-dev
- libqt5opengl5-dev

[4] Before this can be done, the NAO has to be set up first (cf. Sect. 2.4).

- libqt5svg5-dev
- libglew-dev
- net-tools
- graphviz – Optional, for generating module graphs and the behavior graph.
- xterm – Optional, for opening an ssh session from the B-Human User Shell *bush*.
- alcommon – For the extraction of the required alcommon library and compatible boost headers from the *C++ SDK 2.1.4 Linux 32* (*naoqi-sdk-2.1.4.13-linux32.tar.gz*) the script *Install/installAlcommon* can be used, which is delivered with the B-Human software. The required package has to be downloaded manually and handed over to the script. It is available at `https://community.ald.softbankrobotics.com` (account required). Please note that this package is only required to compile the code for the actual NAO robot.

On Ubuntu 18.04, the following command can be executed to install all requirements except for *alcommon*:

```
sudo apt install clang make qtbase5-dev libqt5opengl5-dev libqt5svg5-dev
    libglew-dev net-tools graphviz xterm
```

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
```

to build the whole solution or

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

to build single components.

To clean up the whole solution, use:

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

As an alternative, there is also support for the integrated development environment CodeLite that works 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. The latest reported compatible version of CodeLite is 10.0.0.

2.4 Setting Up the NAO

2.4.1 Requirements

First of all, the atom system image, e. g. version 2.1.4 (*opennao-atom-system-image-2.1.4.13_2015-08-27.opn*), and the *Flasher*, e. g. version 2.1.0, must be downloaded for the corresponding operating system from the download area of `https://community.ald.softbankrobotics.com` (account required). In order to flash the robot, a USB flash drive with at least 2 GB of free space and a network cable is required.

To use the scripts in the directory *Install*, the following tools are required[5]:

sed, *rsync*.

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

The commands in this chapter are shell commands. They must be executed inside a Unix shell, i. e. on Windows, *bash* has to be started first. All shell commands should be executed from the *Install* directory.

2.4.2 Installing the Operating System

After the robot specific configuration files were created (cf. Sect. 2.4.3 and Sect. 2.4.4), plug in the USB flash drive and start the *NAO flasher tool*[6][7]. Select the *opennao-atom-system-image-2.1.4.13.opn* and the 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 that is switched off 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 the computer to the robot using the network cable and configure the network for DHCP. Once the reboot is finished, the NAO will do its usual wake-up procedure. Now the NAO will say its current IP address by pressing the chest button.

2.4.3 Creating Robot Configuration Files for a NAO

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

Before creating the 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 the 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. It is assumed that the robot is already connected via an ethernet connection and has reported its IP address to be 169.254.54.28 (via pressing the chest button):

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

[6] On Linux and macOS the flasher has to be started with root permissions. Usually this can be done with sudo ./flasher

[7] On Linux there may be an error about a missing *zlib* version. This can be resolved by removing the three files starting with *libz* in the *lib* directory

```
cd Install
./createRobot -t 3 -r 25 Penny
./addRobotIds -ip 169.254.54.28 Penny
```

If the NAO is not available, the serial numbers can also be specified manually:

```
./addRobotIds -ids ALDxxxxxxxxxxx ALDxxxxxxxxxxx Penny
```

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

Note: When upgrading from an older B-Human code release running *createRobot* is not necessary. Nevertheless, the script *addRobotIds* has to be executed for robots that were installed with code releases before 2016.

2.4.4 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.

Later, different configurations can be selected by calling the script *setprofile* on the NAO, which overwrites the *default* configuration.

```
setprofile SPL_A
setprofile Home
```

Another way to switch between different configurations is by using the tools *copyfiles* (cf. Sect. 2.5) or *bush* (cf. [4, Chapter 10.2]).

2.4.5 Installing the Robot

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

```
./installRobot 169.254.54.28
```

Follow the instructions on the screen until the robot reboots.[8]

Now *copyfiles* (cf. Sect. 2.5) or *bush* (cf. [4, Chapter 10.2]) can be used to copy compiled code and configuration files to the NAO.

2.5 Copying the Compiled Code

The script *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 macOS, an IDE can be used with *copyfiles*. In Visual Studio, the script can be executed by "building" the project *copyfiles*, which can be built in all configurations. If

[8] The password *nao* will only be required to enter if the ssh key has not been copied yet, i. e. if neither *addRobotIds -ip* nor *installRobot* ran before for this robot.

the code is not up-to-date in the desired configuration, it will be built. After a successful build, a promp to enter the parameters described below will appear. On macOS, a successful build for the NAO always ends with a dialog asking for *copyfiles*' command line options. The script can also be executed 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*)[9], and second, the IP address of the robot. To adjust the desired settings, it is possible to set the following optional parameters:

Option	Description
-b	Restarts *bhuman* (and *naoqi* if necessary) after copying.
-c <color>	Sets the team color to *blue*, *red*, *yellow*, *black*, *white*, *green*, *orange*, *purple*, *brown*, or *gray* replacing the value in the *settings.cfg*.
-d	Removes all log files from the robot's */home/nao/logs* directory before copying files.
-h | --help	Prints the help.
-l <location>	Sets the location, replacing the value in the *settings.cfg*.
-m <number>	Sets the magic number. Robots with different magic numbers will ignore each other when communicating.
-n	Stops *naoqi*.
-nc	Never compiles, even if binaries are outdated.
-nr	Does not check whether the robot to deploy to is reachable.
-o <port>	Overwrite team port (default is 10000 + team number).
-p <number>	Sets the player number, replacing the value in the *settings.cfg*.
-r <n> <ip>	Copies to IP address <ip> and sets the player number to *n*. This option can be specified more than once to deploy to multiple robots.
-s <scenario>	Sets the scenario, replacing the value in the *settings.cfg*.
-t <number>	Sets team number, replacing the value in the *settings.cfg*.
-v <percent>	Sets NAO's sound volume.
-w <profile>	Sets wireless profile.

Possible calls could be:

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

The destination directory on the robot is */home/nao/Config*. Alternatively, the B-Human User Shell (cf. [4, Chapter 10.2]) can be used to copy the compiled code to several robots at once.

2.6 Working with the NAO

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

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

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

the file *Config/Scenes/Includes/connect.con*. Thus a later use of the SimRobot scene *RemoteRobot.ros2* will automatically connect to the same robot. On macOS, the IP address is also the default address for deployment in Xcode.

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

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

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

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

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

status shows the status of NAOqi and *bhuman*.

stop stops running instances of NAOqi and *bhuman*.

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

reboot reboots the NAO.

2.7 Starting SimRobot

On Windows and macOS, SimRobot can either be started from the development environment or by starting a scene description file in *Config/Scenes*[10]. 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 a 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.

[10] 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 macOS, starting a scene description file bears the risk of executing a different version of SimRobot than the one that was just compiled.

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*. A promp will appear to enter the NAO's IP address.[11] In a remote connection, the simulation scene is usually empty. Therefore, it is not necessary to open a scene view.

2.8 Calibrating the Robots

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

In addition to that, the B-Human software uses four color classes (cf. [4, Chapter 4.1.4]) which have to be calibrated as well (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 up the robot (cf. Sect. 2.8.2). In some cases it may be necessary to repeat these steps, because big changes in the joint calibration may invalidate the camera calibration.

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 (cf. [4, Chapter 8.3.4]). 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

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

angles. Otherwise, the balancing mechanism of the motion engine might move the legs, messing up the joint calibrations. This can be done with

```
get representation:MotionRequest
```

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

When the calibration is finished, it should be saved:

```
save representation:JointCalibration
```

Manually Adjusting Joint Offsets

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

```
mr StandArmRequest CalibrationStand
mr StandLegRequest CalibrationStand
```

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

```
get representation:JointCalibration
```

modifying the calibration returned and then setting it, or by using a Data View (cf. [4, Chapter 10.1.4.5]):

```
vd representation:JointCalibration
```

which is more comfortable.

Using the JointCalibrator

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

```
call Calibrators/Joint
```

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

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

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

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

or with:

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

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

Straightening Up the NAO

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

```
get representation:CameraCalibration
call Calibrators/Joint
```

(a) (b)

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

Copy the values of *bodyRotationCorrection* (representation *CameraCalibration*) into *bodyRotation* (representation *JointCalibration*). Afterwards, set *bodyRotationCorrection* (representation *CameraCalibration*) to zero. Another way to make these actions more or less automatically is possible by using the AutomaticCameraCalibrator with the automation flag (cf. Sect. 2.8.3).

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

For calibrating the cameras (cf. [4, Chapter 4.1.2.1]) using the module AutomaticCameraCalibrator, 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 *Calibrators/Camera.con* (in the console type *call Calibrators/Camera*). This will initialize the calibration process and furthermore print commands or help to the simulator console that will be needed later on.

3. Announce the robot's position on the field (cf. [4, Chapter 4.1.2]) using the `AutomaticCameraCalibrator` module (e. g. for setting the robot's position to the penalty mark of a field, type *set module:AutomaticCameraCalibrator:robotPose rotation = 0; translation* = {*x = -3200; y = 0;*}; in the console).

4. To automatically generate the commands for the following joint calibration to correct the body rotation, a flag can be set via *set module:AutomaticCameraCalibrator:setJointOffsets true*. After finishing the optimization, the rotation can be corrected by entering the generated commands.

(a) (b) (c)

Figure 2.2: The three interesting camera calibration stages. a) is the start of the calibrator. b) is the view after the control start with gathered samples. c) is the stage after optimization.

5. To start the point collection, use the command *dr module:AutomaticCameraCalibrator:start* and wait for the output "Accumulation finished. Waiting to optimize...". The process includes both cameras and will collect samples for the calibration and make the head motions to cover the whole field. The samples for the upper camera are drawn blue and the samples for the lower camera red. A drawing above the images signalizes if the sample amount is sufficient for optimization (green) or not (red).

6. If some specific samples should not be considered, they can now be deleted by left-clicking onto the sample in the image in which it has been found. If there are some samples missing, they can be added manually by *Ctrl* + left-clicking into the corresponding image.

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

2.8.4 Color Calibration

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

```
get representation:CameraSettings
```

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

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

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

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

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

autoExposure: Enable (`true`) / disable (`false`) the automatism for exposure. When playing under static light conditions such as in the standard indoor tournament, this parameter

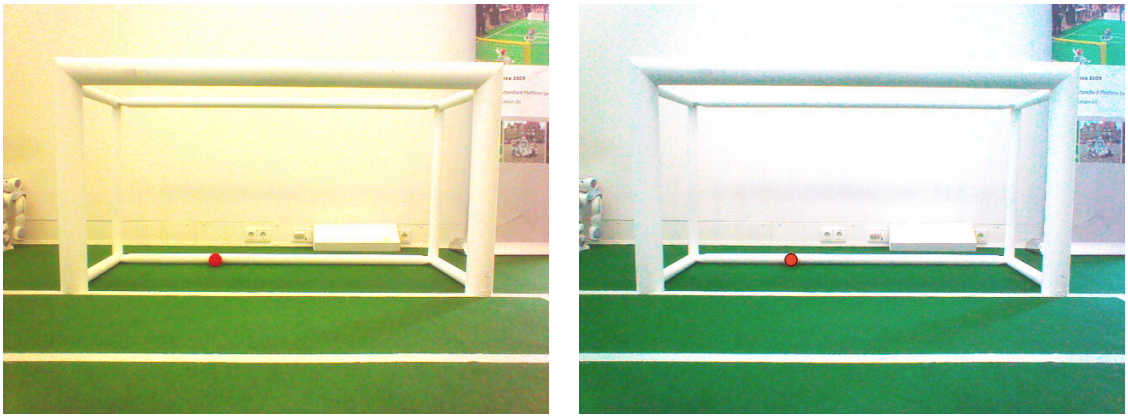


Figure 2.3: The left figure shows an image with improper white balance. The right figure shows the same image with better settings for white balance.

should always be disabled, since the automation will often choose higher values than necessary, which will result in blurry images. However, for dynamic light conditions as were present in the Outdoor Competition at RoboCup 2016, using the automatism of the camera driver may be a necessity. In this case, its behavior can be altered using the parameters autoExposureAlgorithm and brightness.

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

```
dr module:CameraProvider:doWhiteBalanceUpper
dr module:CameraProvider:doWhiteBalanceLower
```

After setting up the parameters of the camera driver, the parameters of the color classes must be updated (cf. [4, Chapter 4.1.4]). To do so, one needs to open the views with the segmented upper and lower camera images and the color calibration view (cf. [4, Chapter 10.1.4.1]). 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 *cameraSettingsV5.cfg* (or *cameraSettingsV4.cfg* if the NAO is a V4 model) and *fieldColorsCalibrationV5.cfg* (or *fieldColorsCalibrationV4.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 large amount of configuration files which can be altered without causing recompilation. All the files that are used by the software[12] are located within the directory *Config*.

Scenarios 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/Scenarios* and placing configuration files in it. Those configuration files are only taken into account if the scenario is activated in the file *Config/settings.cfg*.

Locations can be used to configure the software for use in different locations, e. g. in the lab or at different competitions. For instance, the field dimensions and the color calibration can depend

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

on the location the robots are used in.

Robots. Besides the global configuration files, there are some files which depend on the robot's head, body, or both. To differentiate the locations of these files, the names of the head and the body of each robot are used. They are defined in the file *Config/Robots/robots.cfg* that maps the serial numbers of the heads and the bodies of the robots to their actual names. In the Simulator, both names are always "Nao".

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

1. *Config/Robots/<head name>/Head/<filename>*
   - Used for files that only depend on the robot's **head**
   - e.g.: *Robots/Amy/Head/cameraIntrinsics.cfg*
2. *Config/Robots/<body name>/Body/<filename>*
   - Used for files that only depend on the robot's **body**
   - e.g.: *Robots/Alex/Body/walkingEngine.cfg*
3. *Config/Robots/<head name>/<body name>/<filename>*
   - Used for files that depend on both, the robot's head **and** body.
   - e.g.: *Robots/Amy/Alex/cameraCalibration.cfg*
4. *Config/Locations/<current location>/<filename>*
5. *Config/Scenarios/<current scenario>/<filename>*
6. *Config/Robots/Default/<filename>*
7. *Config/Locations/Default/<filename>*
8. *Config/Scenarios/Default/<filename>*
9. *Config/<filename>*

Whether a configuration file is robot-dependent, location-dependent, scenario-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. [4, Chapter 3.3.5]).

Chapter 3

Changes Since 2017

3.1 Infrastructure

3.1.1 Type Registration

Many datatypes in the B-Human system are streamable, i. e. they can be serialized to and from data streams, e. g. for reading them from a file. This technique was originally developed as part of the GermanTeam framework [8]. Part of the streaming architecture is the ability to determine the specification of datatypes at runtime, for instance, to be able to read them from a structured configuration file. However, in its original implementation, streaming was mainly used as a debugging feature, e. g. to inspect data structures with an external PC at runtime. Therefore, the approach that was selected to acquire the specification of datatypes did not need to be very efficient, because it was rarely used when not debugging. As a result, the specification was determined during streaming, i. e. while the data was serialized its specification was recorded. In the current B-Human system, streaming is used much more often during the normal execution than in the GermanTeam system. In particular, the robots log a lot of data while they play soccer. It turned out that determining the specification over and over again while logging created a significant overhead. For instance last year, logging took around 1 ms per frame in the thread that performs image processing.[1]

Therefore, in the 2018 B-Human framework, the specification of all streamable datatypes is now determined only once at the beginning of the program. This would normally require a larger manual coding overhead, because a separate method is needed for each streamable datatype to record the specification. However, since most streamable datatypes in the B-Human system are generated from macros by the C++ preprocessor, this change did not require any manual adaptation for most datatypes. The new approach simplifies the access to the specification of datatypes, because it is now immediately available, while before, it was only available for datatypes that had been streamed at least once. In addition, streaming the data to be logged is now significantly faster, freeing up processing time for other tasks.

3.1.1.1 Registering

In case a class is not created using the macro `STREAMABLE` (cf. [4, Chapter 3.4.4]), its specification must be registered manually. Therefore, a class-static method must be defined that registers the class's name as well as all of its attributes. By convention, that method is always called

[1] The actual writing to disk is performed in a separate thread in the background. Its speed is only limited by the write speed of the target medium.

`reg`. The class is registered with `REG_CLASS` or – if it is derived from another registered class – with `REG_CLASS_WITH_BASE`. Each attribute is afterwards registered with `REG`. As the following example shows, it is also possible to register virtual attributes, i. e. ones that are just created during streaming (`c` in the example).

```
struct SimpleType : public Streamable
{
  int a = 0;
  int b = 0;

  void serialize(In* in, Out* out) override
  {
    int& c = b;
    STREAM(a);
    STREAM(c);
  }

private:
  static void reg()
  {
    PUBLISH(reg);
    REG_CLASS(SimpleType);
    REG(a);
    REG(int, c);
  }
};
```

3.1.1.2 Publishing

To perform the actual type registration, the `reg` methods of all types must be executed. This is accomplished by naming the method as the parameter of the macro `PUBLISH`. That macro will add the address of the method to a global, template-based list that is executed once at the beginning of the program. Methods passed to `PUBLISH` must have global linkage, i. e. they must either be global or class-static, but not C-style static. As shown in the example, methods can publish themselves, because it is the instantiation of a template that adds the method to the list, not the execution of some local code. However, this only works if the method containing the macro `PUBLISH` is actually linked into the final binary of the program. If the class is a template or linked from a static library, the linker might ignore the method because there is no external reference to it. So, the macro `PUBLISH` must be inside a method that is guaranteed to be linked. In the example above, the method `serialize` would be a good alternative.

3.1.1.3 Storing

The `TypeRegistry` stores all the type information collected. This information is used for two purposes: On the one hand, the names of enumeration constants can be looked up at runtime. This is required for streaming enumeration types as text. On the other hand, a `TypeInfo` object can be filled. `TypeInfo` objects contain the same information as the `TypeRegistry`, but in a platform independent (demangled) representation, and `TypeInfo` is streamable. Thus it can be sent from a robot to a PC or stored in log files.

3.1.2 Inference of Neural Networks

With the new ball detector using a neural network for classification it became a necessity to add the possibility of inferring neural networks to the B-Human code base. The functionality for handling neural networks is split into three classes in *Src/Tools/NeuralNetwork/*: `Model`, `CompiledNN` and `SimpleNN`.

Instances of `Model` contain the architecture and the weights of a network as well as the functionality to load this information from a file. For storing neural network models, we use a simple binary file format and created a Python script based on kerasify [6] to export Keras [2] models into that format.

The class `CompiledNN` defines an interface for using neural networks at runtime. After a `Model` was initially passed to its `compile()` method, a `CompiledNN` instance can later apply that model on input data using the `apply()` method. Internally, the inference of the given model is translated into pure x86 machine code.

This approach offers very good locality of code and data which is a good prerequisite for caching and allows for low-level optimization of the generated code for the given purposes. Not only can it utilize all SIMD features of the NAO's CPU, but also exploit several static pieces of information about a given model during compilation to generate faster code. For example, depending on the size of kernels, inputs and outputs in a given layer, loops can be unrolled or different strategies for accumulating results can be chosen. Optimizations on a larger scale are possible as well, e. g. if a convolutional or dense layer does not have an activation function and is followed by a batch normalization along the feature axis, the normalization can be integrated into the layer by adjusting its weights accordingly.

Currently, the code generated by `CompiledNN` is mostly optimized for rather small data sizes for every step in the network—e. g. convolutional layers perform best for at most 24 entries in the feature dimension. While larger network architectures are supported, there is no special handling for them as we supposed that the resulting runtime would still be unfeasible for real time processing on the NAO's CPU.

`SimpleNN` provides another means for inferring a neural network. However, unlike `CompiledNN`, it uses plain C++ code as it was not optimized for speed. Instead, it primarily serves as a reference implementation to test the results of the optimized implementation. Additionally, `SimpleNN` is stateless; `SimpleNN::apply()` simply takes a model (or a single layer) and input data and computes the result.

3.2 Perception

3.2.1 Controlling Camera Exposure

Under more natural lighting conditions, it is necessary to dynamically control the cameras' exposures during the games. Otherwise it might be impossible to detect features on the field if part of it is well lit while other parts are in the shadow. NAO's cameras can determine the exposure automatically. However, normally they will use the whole image as input. At least for the upper camera, it can often happen that larger parts of the image are of no interest to the robot, because they are outside the field. In general, the auto-exposure of the cameras cannot know what parts of the image are very important for a soccer-playing robot and which are less important. However, the cameras offer the possibility to convey such priorities by setting a weighting table that splits the image into five by five rectangular regions (cf. Fig. 3.1). This

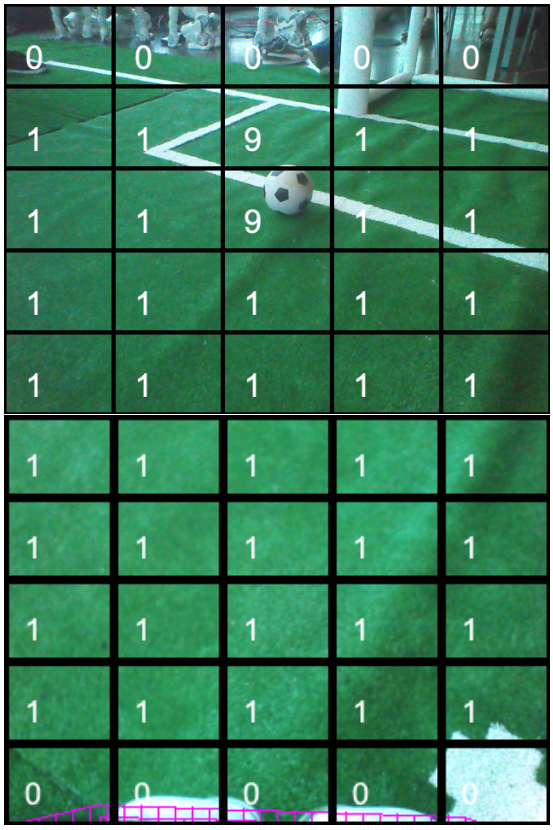


Figure 3.1: Weights for exposure control of the upper and lower camera.

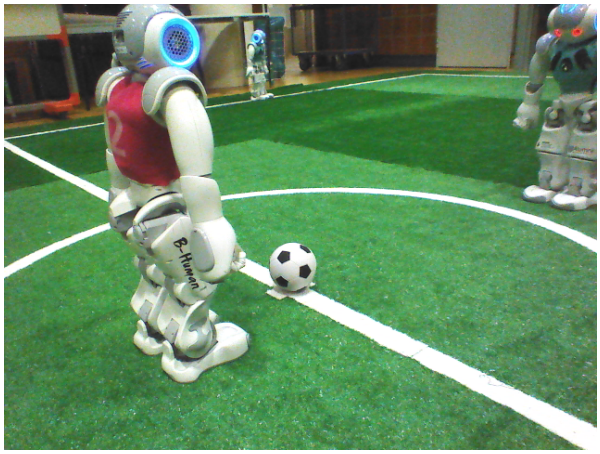
feature is supported by our version of the camera driver.

Therefore, our code makes use of this feature. Regions that depict parts of the field that are further away than 3 m are ignored. Regions that overlap with the robot's body are also ignored, because the body might be well lit while the region in front of it is in a shadow. If the ball is supposed to be in the image, the regions containing it are weighted in a way that they make up 50% of the overall weight (cf. Fig. 3.1).

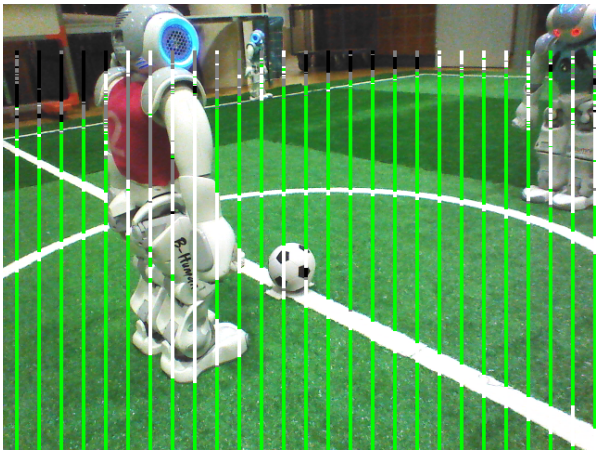
Changing the parameters of the camera takes time (mainly waiting). Therefore, the code talking with the camera driver runs in a separate thread to avoid slowing down our main computations. It turned out that four values could be changed per image. Therefore, the weights are computed in a way that they do not change that often by limiting the values for the field to 0 and 1 and by only using different values for the ball. A maximum of four changes per frame is then sent to the driver.

3.2.2 Detecting the Field Boundary

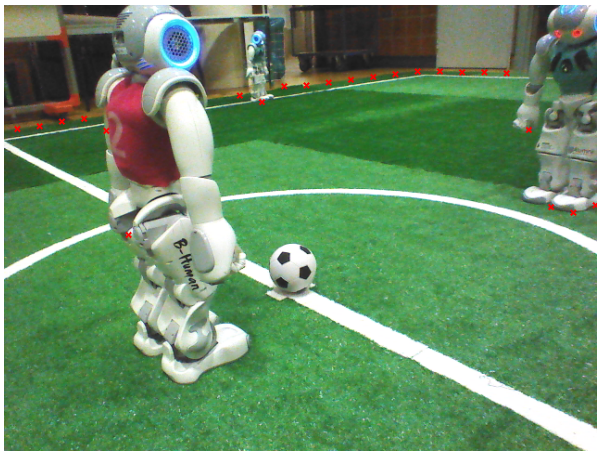
The rulebook of the Standard Platform League specifies in detail how a field and everything allowed to be on it during a game look like. In contrast, very little is specified about the appearance of the world outside the field. The only exception is that another field that is visible must be at least three meters away. Given that everything that is relevant to soccer playing robots is located on their field, it makes a lot of sense to limit image processing to the area of the field they are playing on or at least to reject object detections that do not overlap with that field. To be able to accomplish this, the extent of the field must be determined, i. e. its boundary in the current camera image must be detected.



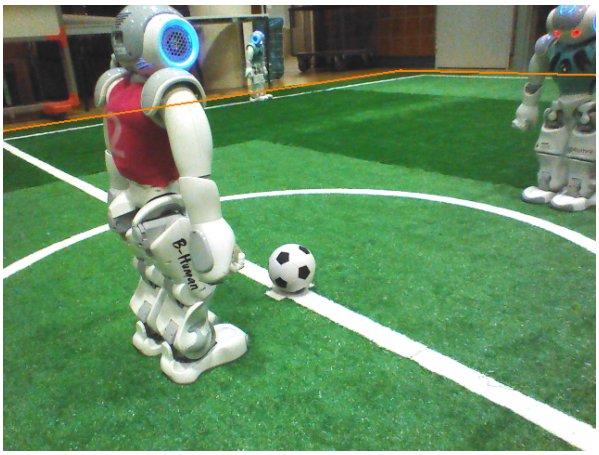
(a) The normal robot view.



(b) Vertical scan lines.



(c) Detected field boundary spots (red crosses).



(d) The detected field boundary with the RANSAC algorithm (orange line).

Figure 3.2: The main steps of the field boundary detection

3.2.2.1 Candidate Spots

The boundary of the field is basically an edge below which the image is mostly green and above which the image is mostly not green. This rule is employed when searching for candidate spots along vertical scan lines in the image (cf. Fig. 3.2b). These scan lines start at the lower border of the image or – if the robot's body is visible in the image – above the assumed contour of the robot's body. They end at the position in the image the boundary would appear when being furthest away possible (i. e. assuming the robot standing in one corner of the field looking towards the opposite corner.). For each scan along a vertical line, a score is maintained that is increased for each field-colored pixel found and decreased for each non-field-colored pixel. The position of the pixel where this score reached its maximum is used as a candidate spot for the field boundary (cf. Fig. 3.2c). However, spots that are very close to the robot are ignored, because it is assumed that it will always have a minimum distance to the actual field boundary.

3.2.2.2 Guessing a Model

Not all candidate spots are located on the actual field boundary, because it can be hidden by other robots, goal posts, and referees. Therefore, it must be determined which spots are really

located on the field boundary and which spots have to be ignored. Domain knowledge is used to ease this decision process. The borders of the actual field consist of straight lines that are perpendicular to each other. A robot can either see one, two, or three of these lines at the same time. Since the case of seeing three lines is very rare and one of those three lines will often appear as very short in the image, our implementation ignores this case and only models the other two.

The approach chosen uses the RANSAC method. By random, three points are drawn from the set of candidate spots in a way that they are ordered from left to right in the image. A straight line is constructed from the first two points. A second line is determined by projecting all three points to the field plane and dropping a perpendicular from the third point to the line spanned by the first two points. If the intersection point of the two lines fulfills a number of criteria (e. g. the corner must be convex and must be left of the third point), it is considered as a corner and the second line is also used in the following step.

In that step, the squared sums are calculated of a) the distances of all points before the corner to the first line, b) the distances of all points after the first corner to the first line, and c) the distances of all points after the corner to the second line.[2] Distances above a certain threshold are clipped to that threshold to avoid that, e. g., spots resulting from objects that hide the boundary significantly influence the outcome of the computation. In addition, distances of points above a line are weighted more than points below the line, again, because of objects that might hide the boundary. Two models are then considered: either the field boundary only consists of the first line (sum (a) plus sum (b)) or it consists of the first and the second line (sum (a) plus sum (c)). The model with the smaller sum is selected, i. e. the model that is supported more by the point set, because the points deviate less from it.

This process is repeated several times and the best model is kept (cf. Fig. 3.2d). It is noteworthy that summing up the distances can stop early whenever the current sum gets bigger than the sum of the best model found so far, because the current model will definitely be worse. Thus, quite a number of models can be checked in a short amount of time. The process ends when a model with a deviation sum below a certain threshold is found or after a maximum number of iterations is reached. On average, the process takes about 0.25ms on the NAO.

3.2.2.3 Projection between Camera Images

In a certain direction, the actual field boundary can usually only appear in one of the two cameras (except for a small overlap between the images), i. e. either in the upper camera or in the lower one. Therefore, the field boundary determined from the previous image is projected to the current image before a new one is computed, considering odometry and head motion. Under some conditions, candidate spots are simply computed from the projected field boundary, namely if the projection is below the current image or if it is above and the search reached the upper border of the current image.

3.2.3 Detecting Obstacles

On a field in the Standard Platform League, there are three kinds of obstacles that a robot wants to avoid when walking or kicking: other robots, goal posts, and referees. Other robots can either be part of the own team or of the opponent team. They can be upright or fallen to the ground. Since 2014, our team has basically used the same approach to detect these objects. That algorithm was well-suited for the task back then. However, the setup of the league has changed

[2] If there is no corner, only a single sum is computed.

since then, which resulted in a number of problems. For instance, the algorithm considers everything that is white and not a field line to be an obstacle. Although the black and white ball that was introduced in 2016 is not detected as an obstacle when surrounded by green, it is often considered to be an obstacle when appearing next to field lines or field line crossings. Also, black regions were considered as obstacles (referees), but with the more dynamic lighting used today, there are often dark shadows on the field, which should not be detected as obstacles. Finally, robots were only allowed to wear the official magenta or cyan jerseys in 2014, while teams can now design jerseys on their own from a wide range of colors.

While some of these issues have already been addressed in the past, the obstacle detection module was completely rewritten this year. It still follows the same general approach, but uses more refined solutions for some details. In general, it is distinguished between obstacle regions detected in the image and obstacle positions in robot-centric field coordinates. While obstacle regions in the image can always be computed, obstacles can only be projected to the field plane if their lower end is visible in the image (e. g. a robot's feet). On the other hand, if a robot is visible in the lower camera image, its jersey is usually not, which prevents an immediate distinction between teammate and opponent. Therefore, some information can be determined from a single image while other information requires combining information from images taken by both cameras.

3.2.3.1 Candidate Spots

Similar to our other object detection methods, candidate spots for the lower ends of obstacles are determined by scanning along vertical grid lines. However, a different scan grid is used (cf. Fig. 3.3). On the one hand, the grid has to be denser in horizontal direction to have a bigger chance to find the feet of robots that are even further away and to reasonably distinguish them from field lines. On the other hand, the grid can be less dense in vertical direction, because the objects searched for are upright and do not get as small in the image as, e. g., horizontal field lines would. Scans begin slightly below the field boundary (cf. Sect. 3.2.2) and continue until the lower border of the image or the contour of the robot's own body, whatever is reached first. Two scores are maintained, and the lowest pixel in the image with both scores above a certain threshold is used as candidate spot. A short range score is immediately reset when a field-colored pixel is found (green or (shadow) black), while a long range score is decreased more slowly. They are both increased with each non-field-colored pixel found. The effect is that the scan recovers relatively quick from individual field-colored pixels, but it takes longer to recover from sequences of non-obstacle-colored pixels. The candidate spots found are below parts of robots (feet, torso, arms), the ball, when it appears below or above a field line, and steep diagonal field lines (cf. Fig. 3.3).

3.2.3.2 Clustering Spots to Detect Obstacles

The basic idea of detecting the obstacles from the candidate spots is to cluster neighboring spots that appear on a similar height in the image. If spots result from field lines, the field lines would be diagonal in the image, i. e. neighboring spots would not be on the same height. In addition, the sum of the long range scores of all clustered spots must be above a certain threshold, which means that an obstacle must either be high (e. g. an upright robot) or wide (a fallen robot). The clustering is performed from the lowest to the highest spot in the image. Starting from each spot, spots to the left and to the right on a similar height are grouped together, accepting a certain amount of gaps in between. If the set of spots is wide enough and dense enough, it is accepted as an obstacle. Afterwards, all spots in the grouped range plus an addition range to

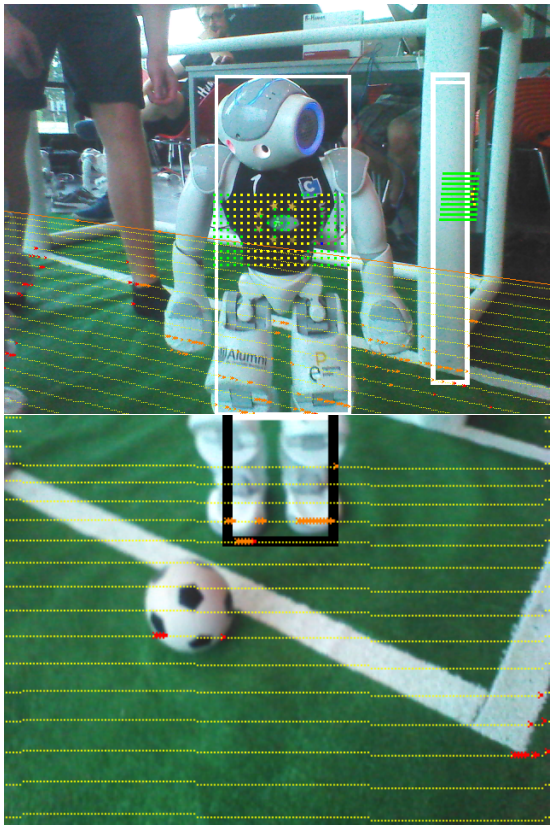


Figure 3.3: Obstacle detection in upper and lower camera images. The thin yellow dots mark pixels scanned from top to bottom. The orange and red crosses mark candidate spots. The red ones were used as starting points for obstacle detection. The orange ones were removed after an obstacle was successfully detected. The thick yellow, blue, and green dots mark the region that was scanned for the jersey color. Yellow dots mark the own (black) team, blue ones the opponent (red) team, and green ones were ignored. The obstacle found in the lower image is marked as black, because a black jersey was detected in the upper image.

both sides are removed (cf. Fig. 3.3) and the algorithm continues with finding the next obstacle. The additional range of spots is removed to prevent the arms of an already detected robot from being detected as separate obstacles.

3.2.3.3 Determining the Jersey Color

The obstacle detection does not distinguish between different types of obstacles, i. e. robots, goal post, and referees. In fact, it does not detect referees wearing black trousers at all, because black is considered to be a possible color of the field. All obstacles found are treated as robots. Therefore, it is tried to detect their jersey color. To avoid a calibration of the colors, a differential approach is used. For each obstacle, the image is sampled at a certain height in the region of a perspectively distorted rectangle.[3] For each non-green pixel it is determined, whether it more likely belongs to the own team color or to the opponent team color. These decisions are counted and if there is a clear majority for one of the two possibilities, that jersey color is assigned to the obstacle.

[3]If the lower end of the obstacle is not in the image, the position of the sample region is guessed from the width of the obstacle.

The color classification first distinguishes between saturated and non-saturated colors. If both colors are saturated, the colors' hue values are used. If they are both non-saturated, their classification to either black or white is used. For gray, a region below the jersey is scanned to get an estimate for the brightness of white. Then, gray is expected to be in a certain range relative to that brightness. Please note that this approach will classify goal posts as robots with jerseys if one of the two teams plays in white. Green jerseys will not be detected at all.

3.2.3.4 Propagating Information from Upper to Lower Image

If an obstacle is detected in the upper image, but its lower end is not inside the image, it is still provided as an obstacle region in image coordinates, but it is not provided as a robot-centric obstacle in field coordinates. Instead, it is reused when the next image from the lower camera is processed. The scan scores at the lower end of the upper image are used as starting points in the lower image to basically continue the scans that were started in the upper image. In addition, the jersey colors of incomplete obstacles in the upper image will be assigned to obstacles in the lower image if both detections were in the same direction (cf. Fig. 3.3).

3.2.4 Detecting the Ball

The Neural Network Ball Perceptor (NNBallPerceptor) takes the spots which are provided in the representation *BallSpots* (cf. [4, Chapter 4.2.1]) and classifies them into balls and other objects. To do this, for each spot the size is calculated which a ball would have at this particular position in the camera image. Because the spots are not always in the middle of the ball a squared patch with an edge length of 3.5 times the radius is used. The resulting area is then scaled to 32x32 pixels by leaving out pixels for downscaling and taking pixels several times for upscaling. The resulting patch is committed to the neural network, if its projected position is on the field.

The core of the perceptor is the neural network (Table 3.1) which classifies the patches whether they show a ball or not. There are three stages of classification:

- If the result of the neural network is at least 0.9 the loop over all spots stops immediately and the spot is returned.
- Otherwise if the result of the neural network is at least 0.5 the patch is assumed to contain a ball. The best patch of this category is returned if no better patch of a better category is found.
- Otherwise if the result of the neural network is at least 0.3 the patch is assumed to contain a *guessed ball*. This means it could be a ball but not necessarily. The best patch of this category is returned if no better patch of a better category is found.

Sometimes the ball spot is not in the middle of the ball which leads to worse results in the classification. However, most images get a really low rating, therefore a threshold of 0.1 is used to consider spots for resampling. In the resampling procedure the spot is moved for the half radius to the top, bottom, left and right and evaluated again. The maximum value of the resampling is used to categorize the ball spot.

After the classification a Hough transformation is used to find the circle like ball in the patch to refine its position.

Typ	Strides	Output Size
Input		32x32x1
Batch Normalization		32x32x1
Convolutional	(1,1)	30x30x4
Max Pooling		15x15x4
Batch Normalization		15x15x4
Convolutional	(1,1)	13x13x8
Max Pooling		6x6x8
Batch Normalization		6x6x8
Convolutional	(2,2)	2x2x8
Max Pooling		1x1x8
Flatten		8
Batch Normalization		8
Dense		1

Table 3.1: Net architecture of the ball classifier.

3.3 Motion Control

3.3.1 Enhancement of the Kick Range

To counter the increasing wear of our old robots and the subsequent decrease of the kick range, we added a small extension to our KickEngine. Directly before a robot hits the ball, the leg joints are set to their maximal values (Figure 3.4). This makes the joint controllers run with maximal power, which increases the kick range about two meters.

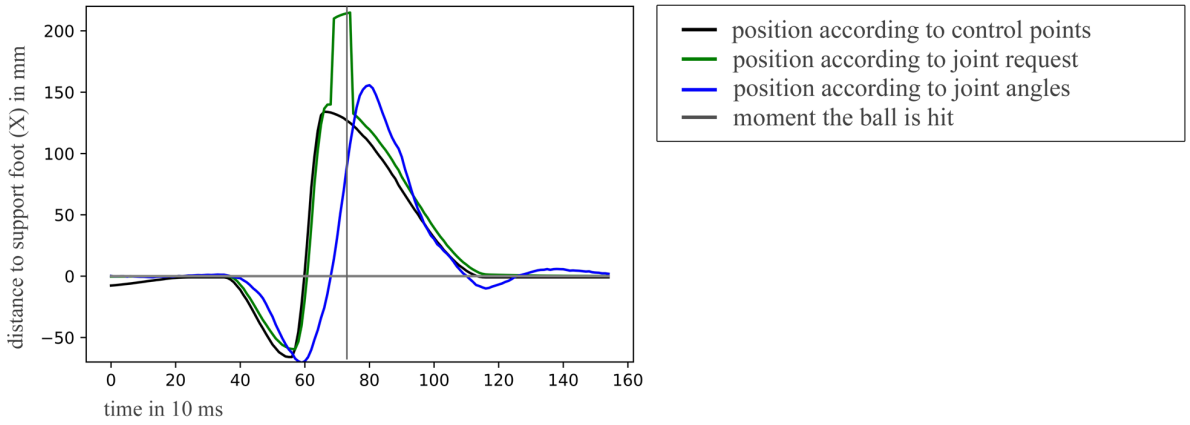


Figure 3.4: Comparison of the requested and executed leg trajectory during kicking.

3.3.2 Getup Motions

During a game, robots might fall to the ground for many different reasons. They might bump into other robots. Sometimes, the ground is not flat enough for stable walking or it gives way when the robot steps on it. Some robots might even have so much play in their joints that walking simply fails. In any of these cases, the robot must get up after it fell down. When on the ground, the robot can either lie on its front or on its back, which requires two different kinds of motions to get up. In addition, the robots might be in different states of wear and tear

and their motors are able to exert different amounts of torque, depending on their temperatures estimated by their controllers. All this makes getting up a difficult task.

Last year, the success rate for a NAO to get up after falling was only about 70%. In addition, up to three attempts with different kinds of motions were executed in a row to find one that actually worked. They were ordered by their speed of execution. Since we wanted to have as few motions as possible to work with and wanted to balance our motions better, we extended our infrastructure to integrate similar motions into one, made changes to our balancing process, and added a way to react to different situations based on sensor feedback. This was partially successful and effectively increased the success rate of getting up to about 80% at RoboCup 2018.

We used the same motion control system as last year (cf. [4, Chapter 8]), but extended it with a few extra features. To see how the get up motions work in general, see [4, Chapter 8.6].

3.3.2.1 The Different Motions

We reduced the number of get up motions to a total of five. We have two motions for getting up when fallen to the front, two for getting up when fallen to the back and a fifth that is executed as a follow-up motion by two of the other motions.

The fastest get up motion when fallen to the front is `frontFast` (cf. Fig. 3.5). The NAO moves its arms to the side and pulls them together in front of the body to get on its feet. Then, it pushes the torso up to a squatting position followed by straightening the legs to a standing position. It takes between 3 and 4 seconds.

Figure 3.5: Fast get up motion from a fall to the front

A slower variant is `frontFreeJoints` (cf. Fig. 3.6). The NAO moves its arms to the back, squeezes its legs to the front and swings over so it can get on its feet. At this point, the motion `sitDownFreeJoints` will take over. In total, it takes about 6.5 seconds until the NAO stands again.

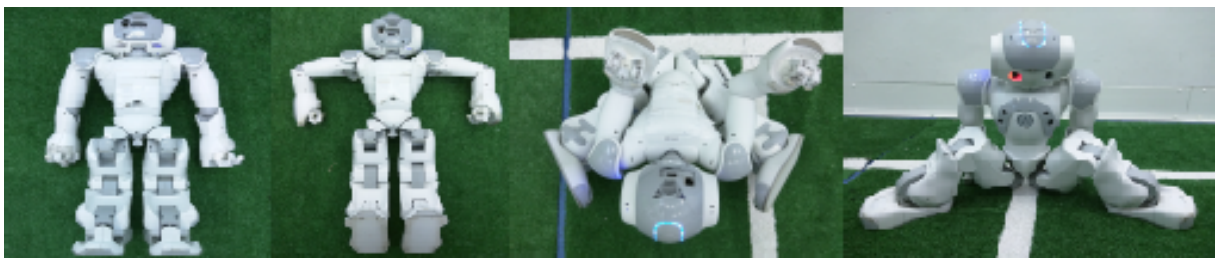


Figure 3.6: Getting up from the front to a wide crouching position

The third motion is `backFreeJoints` (cf. Fig. 3.7), which is our slower motion to get the NAO up from the back. The NAO moves its arms under its back, so it can swing over on its feet. Again, the motion `sitDownFreeJoints` will be executed from here. In total, it takes about 5.3 seconds until the NAO stands.

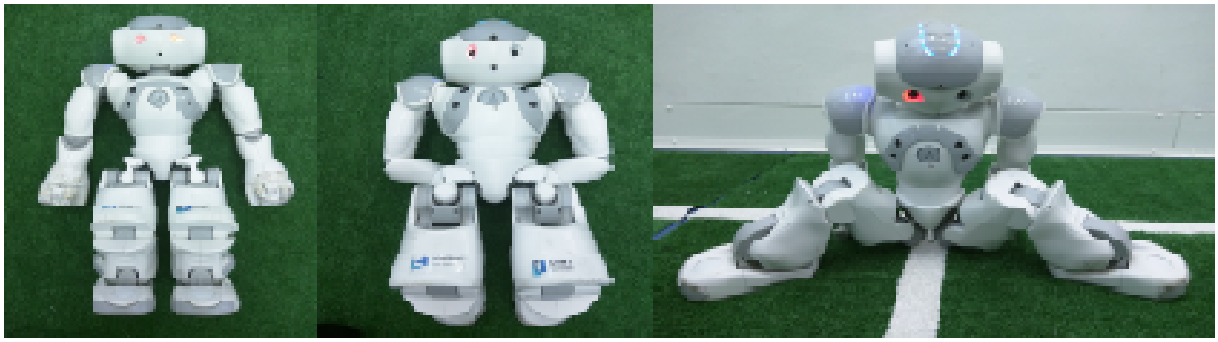


Figure 3.7: Getting up from the back to a wide crouching position

The motion `sitDownFreeJoints` (cf. Fig. 3.8) is always executed after either `frontFreeJoints` or `backFreeJoints`. The NAO will transfer its weight on its left foot, then pull over its right foot and pull together both knees. From the resulting squatting position, the robot can lift itself up to a standing position.

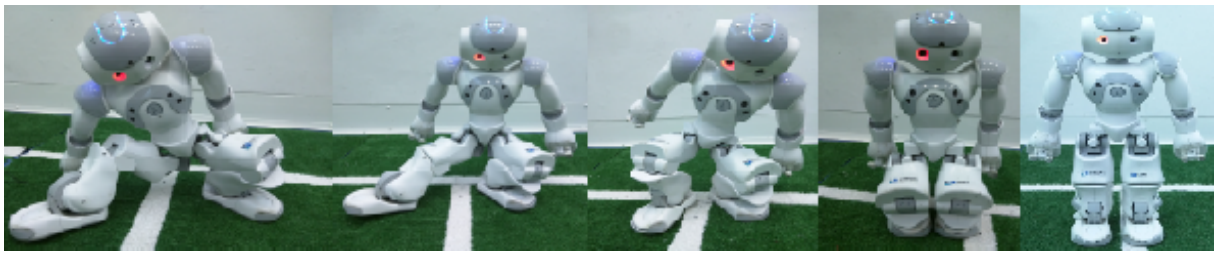


Figure 3.8: Getting up from a wide crouching position

The final motion is `backFast` (cf. Fig. 3.9). It starts the same as the motion `backFreeJoints`. However, instead of executing the motion `sitDownFreeJoints` it uses a different approach to get from the wide crouching position to the stand. The NAO moves on its left foot, waits, then moves on the right foot, waits, transitions into a sitting position and then stands up. This motion takes about 4.7 seconds.

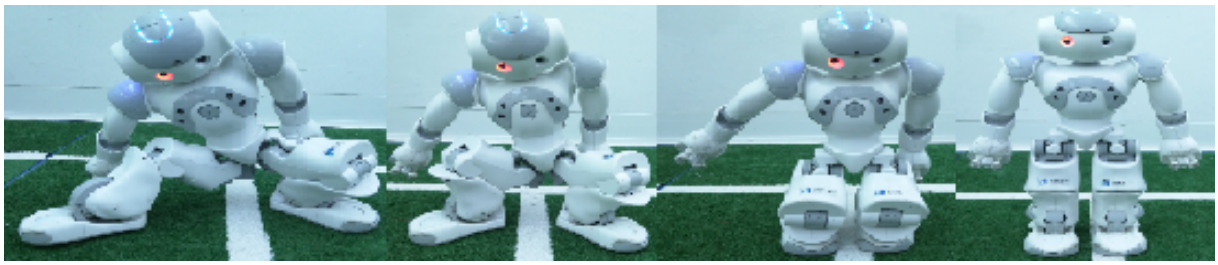


Figure 3.9: Alternative for getting up from the back

3.3.2.2 Configuring Motions

Not all getup motions work on all robots. Some of the faster ones might not always work. Therefore, a sequence of getup motions can be configured that the robot will try one by one until it successfully got up or until no motion is left. We prefer that the robot stops trying to getup instead of damaging itself by trying over and over. The sequence of getup motions to try can be configured in the file *damageConfigurationBody.cfg* that is usually located in the

robot-specific directory *Config/Robots/<robot name>/Body*. The two fields `getUpFront` and `getUpBack` specify the sequence of getup motions, using the names given in the section above.

3.3.2.3 Balancing

Since 2017, we have used a gyro-based PID controller (cf. [4, Chapter 8.6]) and a ZMP balancer (cf. [4, Chapter 8.7]) to balance our robots. To improve the success rate of getting up, the gyro-based controller can be activated or deactivated for each individual phase of each getup motion. Two different levels of impact can even be chosen for each balancing direction, i. e. forwards and sideways.

We still use the same ZMP balancer as last year (cf. [4, Chapter 8.7]) that uses the arms and/or legs to balance the robot. It can be activated from a certain phase in each getup motion and it runs until the motion is finished. Different parameter sets can be selected in different phases.

3.4 Behavior Control

3.4.1 Behavior Infrastructure

Since 2013, B-Human has used CABSL [7] to describe the robot behavior as a hierarchy of finite state machines. In addition, so-called libraries provide functionality that is called from the state machine but does not fit into the dataflow oriented framework of representations. However, the behavior hierarchy is still completely contained in a single state machine. This has some disadvantages: These components cannot be exchanged easily or load their own numeric parameters from configuration files. It is also hard to create coordinated team plans for specific situations, such as set plays. This is especially important due to the new free kick rules.

In the RoboCup Small Size League, an architecture called "Skills, Tactics, and Plays" has been established for many years [1] and recently been employed in the Mid Size League [3]. To implement a similar system based on these ideas and to solve some of the problems mentioned above, another abstraction called *behavior option* has been introduced. Behavior options are declared similarly to modules (cf. [4, Chapter 3.3.2]), such that they can specify their own requirements for representations and parameter sets. However, all behavior options are executed in the context of a single module which inherits the requirements of all of them. A behavior option can contain a CABSL state machine, a simple switch-case-block or a completely different representation of a behavior.

The most important addition to the old system is the PlaySelector. It is invoked one level above the normal role selection (cf. [4, Chapter 6.2.1]), so team-wide special situations can override the normal play. The playbook configures the set of plays among which can be chosen during the game. The selection method is based on binary applicability conditions and a numeric value specified by each play to compare them. Team coordination is achieved by letting each robot send its currently active play in its WiFi message. On each robot, the PlaySelector checks if there is another robot leading a play and, in that case, checks whether that play is possible to execute according to the own world model.

3.4.2 Free Kicks

A particular instance of situations in which the standard behavior must be overridden are the free kicks that have been introduced in the SPL this year. As their behaviors are completely

independent of each other, there are different plays for free kicks executed by our team (*own free kicks*) and free kicks executed by the opponent team (*opponent free kicks*).

3.4.2.1 Own Free Kick

Our own free kick behavior firstly determines the task that a robot has in the free kick. This can be either playing the ball, positioning to receive a pass or falling back to normal behavior. A robot should play the ball if it has the *Striker* or *Keeper* role, depending on whether the ball is near the own penalty area. The robot that walks to the pass reception position is the one with the *Supporter* role, in case that it is currently present on the field. All other robots execute their normal behavior (i. e. the defenders or the keeper if it does not play the ball) by invoking the RoleSelector behavior option.

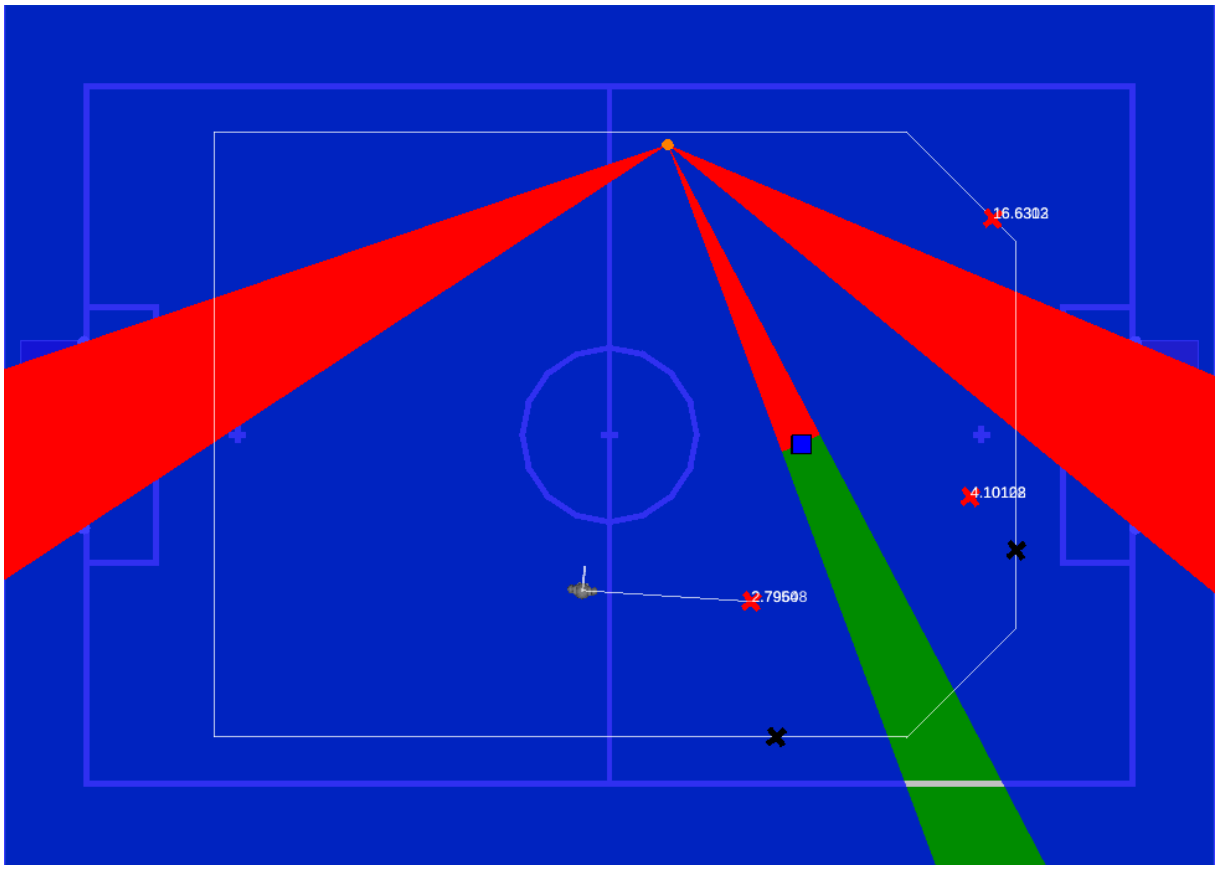


Figure 3.10: Visualization of the pass reception pose calculation. The ball is shown as orange circle. The red sectors are occupied by robots or a goal and thus not further considered in the calculation. The black crosses mark the points at which the potential pass rays intersect with the boundary polygon. The red crosses mark potential pass reception poses. The white line connects the robot to its selected target.

The free kick striker has the choice between two options: Attempting a direct kick towards the goal or passing to a teammate. If there is a large enough free part of the goal according to the KickPose (cf. [4, Chapter 6.5]), a fast forward kick in that direction is performed immediately. Simultaneously, the module PassPoseProvider provides the representation PassSelection which contains the number of a pass candidate. This calculation uses the amount and distance of opponent robots to the line between the ball and the pass candidate and the distance of the potential target to the opponent goal. If the striker thinks that the pass candidate will reach its

target before the free kick ends, it waits for this teammate and then kicks the ball in its direction. If after some time (depending on obstacles near of the ball that are likely to be removed due to the *Illegal Defender* rule) there is still no pass candidate and there is no free part of the goal, a fast forward kick in the direction that is indicated by the KickPose is performed. All the free kick pass receiver does is to walk to a position that is contained in the PassReceptionPose representation. This position should suggest the free kick striker to choose it as a pass receiver. It is calculated in polar coordinates from the point of view of the ball. Sectors that are occupied by robots or a goal are discarded. For each remaining sector a ray is intersected with a boundary polygon and a position between it and the ball is evaluated according to a scoring function. This function depends, among other factors, on the opening angle of the containing sector, the angular distance to the goal, whether the robot would need to pass the line between the ball and the goal to get there, and the distance to the current position of the robot. The position with the best score (combined with the orientation towards the ball) is chosen as the final pass reception pose. The calculation of the pass reception pose is visualized in Fig. 3.10.

3.4.2.2 Opponent Free Kick

Another new play implements a behavior to defend opponent free kicks. The main concept is to keep the positioning as near to the regular positioning as possible, but also to prevent a possible shot at the goal. The solution was a wall in a one meter distance (75cm + tolerance) to the ball, blocking the direct path to the goal. Depending of how many players are close to the free kick area, the wall can consist of one or two robots. Players that are not part of the wall use their usual positioning and behavior. The goalkeeper has its own behavior option in the play, as it does not have to keep a distance while standing in the penalty area. If the free kick is far away, the keeper follows its regular behavior and positions itself accordingly. Near the penalty area, the goal keeper tries to prevent a goal by standing as close as possible in front of the ball.

3.4.3 Kick Pose Provider

Since we were able to expand the range of our kicks, we rewrote our kick selection module to handle wide range kicks considering the position of all teammates and opponents.

We start with the computation of all openings between other robots and the goalposts (Figure 3.11). They are combined with all possible kicks to get all reachable targets. For each target, a score is computed, which represents an estimation of the time until we could score a goal. The fastest way to score is to kick directly into the opponent's goal. If this is possible, we always try it, which corresponds to our previous strategy. Otherwise we would have kicked as close as possible to the goal. We replaced this with a more sophisticated approach, which consists of the following components. In Figure 3.12 is an example of a score with the described components.

Position of Teammates. The time the next teammate needs to reach and kick the ball in the direction of the goal.

Position of Opponents. We add a penalty depending on the distance of the next opponent in the view area of of which the ball could stop considering the kick inaccuracy. As we do not know the orientation of opponents, we assume that they are aligned towards our goal, which is mostly the case. Otherwise they would need a lot of time to turn around the ball, which makes them unimportant.

Distance to the Goal. We consider the distance to the goal as the time a robot needs to walk that far. This perfectly balances dribble kicks and wide range kicks if no other teammates

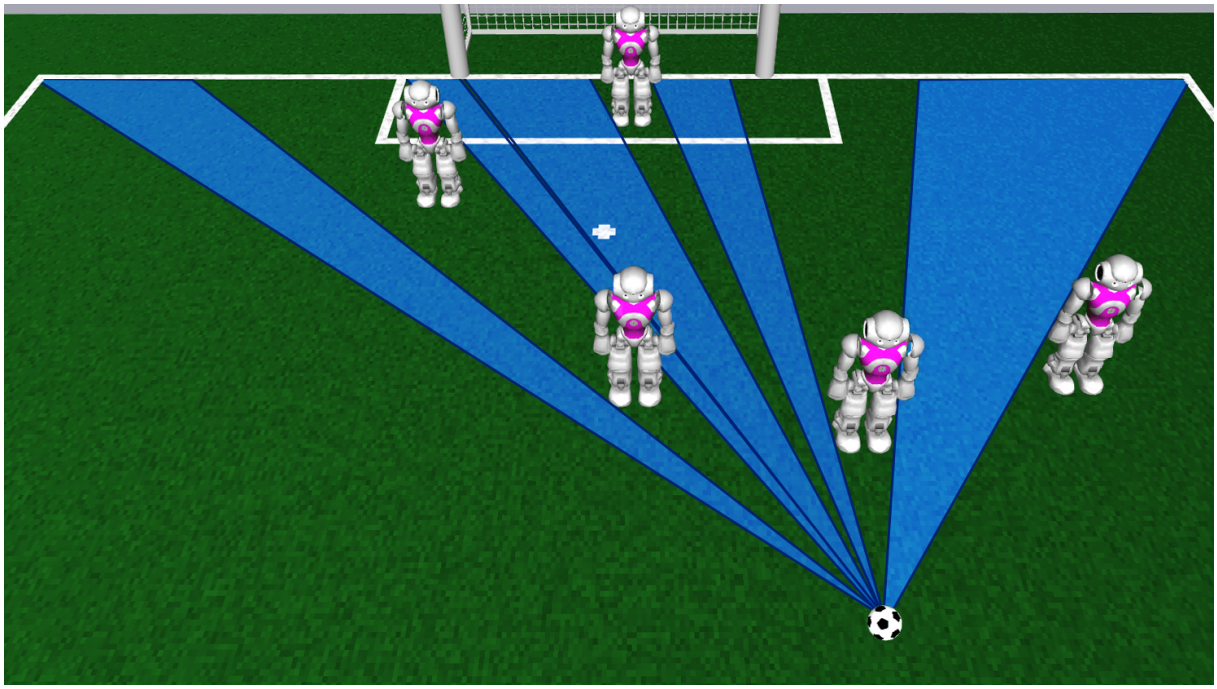


Figure 3.11: Visualization of all openings between other robots.

are around. If the target is inside the goal, we reset the previously calculated score, because the position of teammates and opponents is not relevant. If the ball will reach the goal even with the maximal kick deviation, there is an extra bonus to prefer these kicks.

Time to Kick. The relative position to the ball depends on the kick type and the direction. That is why the time to reach this position, which we call *KickPose*, differs. Some kick types need a preparation time, which is also considered. In case of a free kick, the time to score is ignored.

Penalties. There are penalties for some cases: The target is outside the field or inside the own penalty area, the opening angle is smaller than the deviation of the kick direction, or the selected kick type or direction differs from the last choice.

Special Situation: Own Penalty Area. In the case the ball is inside the own penalty area, it is most important to kick very fast, instead of considering teammate positions or the distance to the opponent's goal.

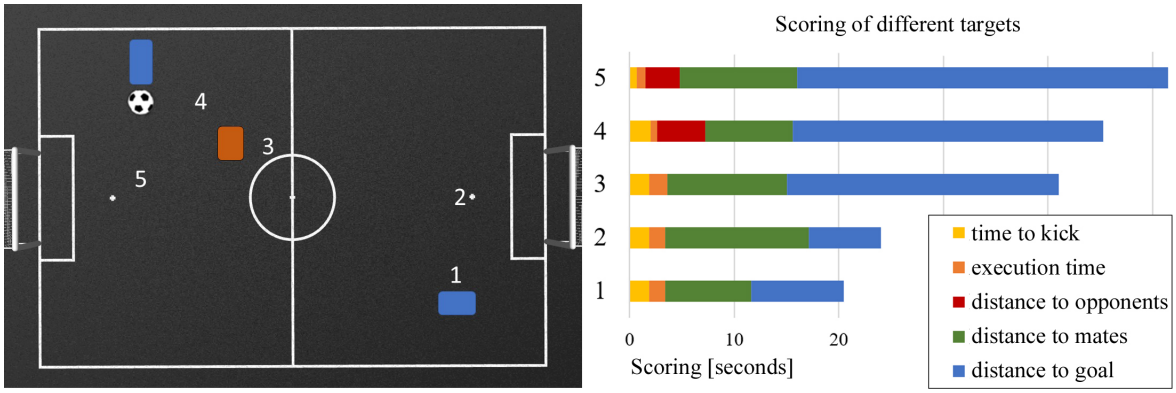


Figure 3.12: Example of the scoring from selected target positions. The right side shows the composition of the scoring for different target positions, which are marked on the left side. The blue rectangle next above the ball represents our robot. In front of it is an opponent (orange rectangle) and next to the goal is a teammate.

Chapter 4

Technical Challenge and Mixed-Team Competition

In addition to the main soccer competition, B-Human also participated in the *General Penalty Kick Challenge* as well as in the *Mixed Team Competition*. For these competitions, all standard modules that have been described in the previous chapters have been used and only a few minor adaptions have been necessary.

4.1 General Penalty Kick Challenge

As in 2017, the technical challenge at RoboCup 2018 was a penalty shootout tournament. The general penalty kick challenge differed from a usual penalty shootout in the way that penalty taker is placed randomly at five possible angles one meter behind the ball. This required some minor adaptions in our normal penalty shootout behavior and localization: In order to avoid touching the ball when starting from one of the outer positions, the penalty taker first walks an arc to a position that is on the line through the center of the goal and the penalty mark. From then on, the situation is the same for all starting positions and the normal penalty shootout behavior is used (cf. [4, Chapter 6.3]). Furthermore, the different positions are handled in the localization by resetting the samples of the particle filter (cf. [4, Chapter 5.1]) to the five possible poses.

The penalty taker has been improved using the kick engine enhancements described in Sect. 3.3.1. As a result, our penalty taker was able to successfully score at all but one attempts during the *General Penalty Kick Challenge*, scoring 9 goals in 4 games, where in the missed attempt the opponent's keeper was in a blocking pose before our taker even kicked the ball. Figure 4.1 shows a successful attempt.

For the penalty keeper, we increased the robustness in our detection of the kick instance and direction against balls that do not lie exactly on the penalty mark. Our keeper is currently fast and reliable enough to block all shots that are kicked with medium speed. One example from the quarter final challenge shootout is shown in Fig. 4.2. Due to its reactivity, our keeper let only three shots pass during the whole challenge. A weak point is the gap beneath the robot's body that forms shortly before the robot reaches the ground when jumping, even when the arm is already blocking parts of the goal that are further to the side of the goal.

An overview of the games is given at the league's web site at `http://spl.robocup.org/standard-platform-league-results-2018/`.

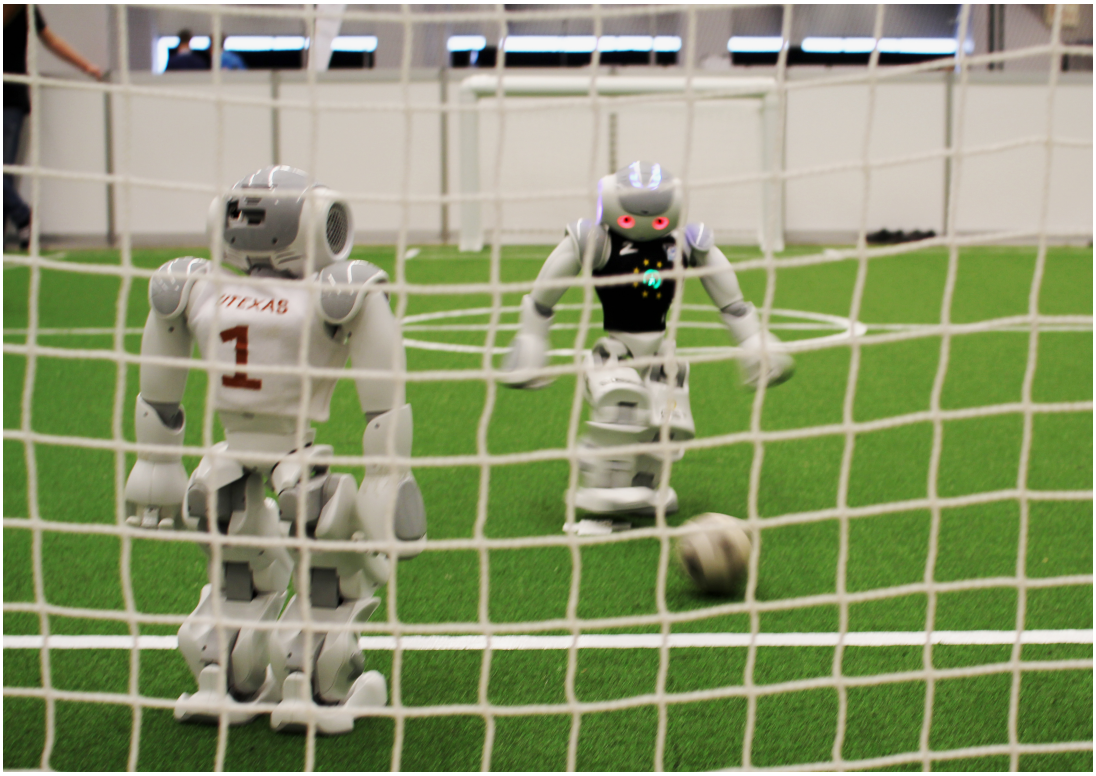


Figure 4.1: B-Human penalty taker in the quarter final of the *General Penalty Kick Challenge* against *UT Austin Villa*

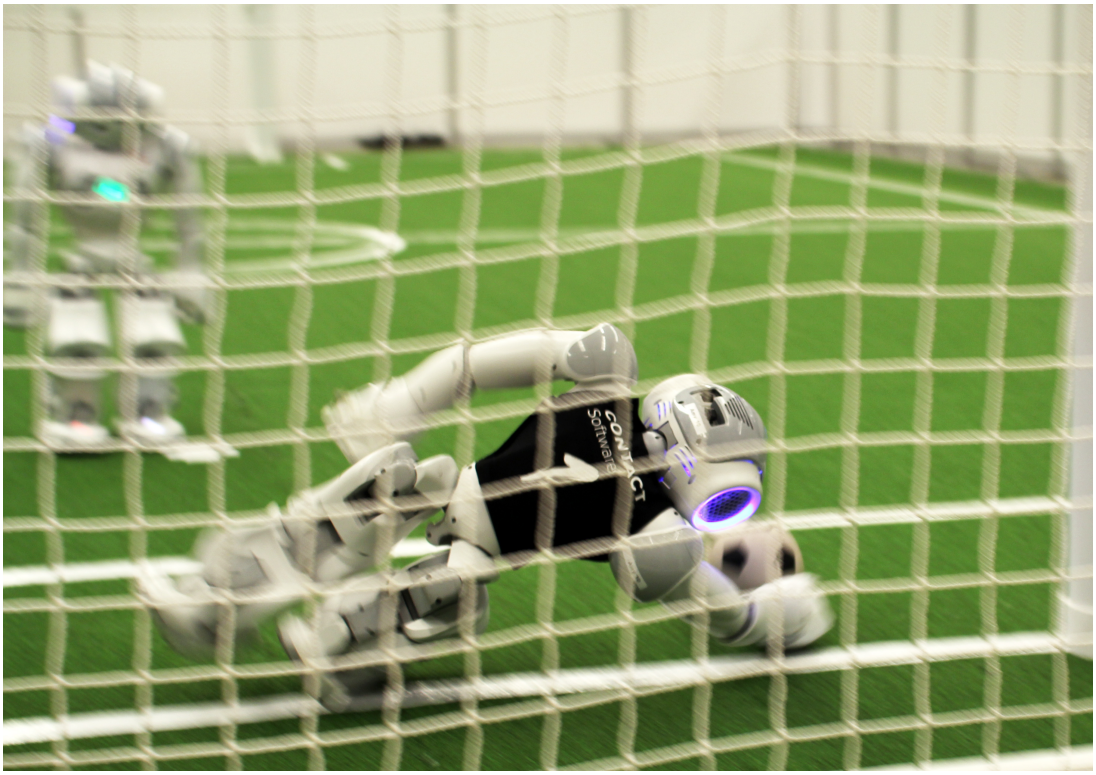


Figure 4.2: B-Human penalty keeper blocking the ball in the quarter final of the *General Penalty Kick Challenge* against *UT Austin Villa*

4.2 *B-Swift* in the Mixed Team Competition

In 2018, the *Mixed Team Competition* was held for the second time. After the successful participation in 2017 together with the HULKs, we were looking for a different challenge which would not allow for extensive tests before the competition. In contrast to choosing the geographically closest team as our partner, we therefore decided for *rUNSWift* from the University of New South Wales, Sydney, Australia. Together, we are *B-Swift*. Most team members are shown in Fig. 4.3.

As opposed to last year's strategy of the *B-HULKs* [5], we did not develop a shared communication protocol beyond what is mandated by the SPL standard message. We had a basic agreement that rUNSWift robots would be responsible for defense (including the goalkeeper) and B-Human's constitute the offense. Thus, the main challenge consisted in selecting the ball playing robot among the mixed team members. Since the SPL standard message does not contain any field to communicate information like this, but both teams have their own mechanisms to assign roles, we agreed on a two-stage solution: Each robot would calculate an approximated time to reach the ball for itself and all team members only using information that is available from the SPL standard message. If the robot with the lowest time is from the same team as the calculating robot, it would then run the usual role assignment algorithm using data from the custom message part. Otherwise it would be assumed that a robot of the other team plays the ball. This protocol does not inhibit frequent role switches or multiple robots going to the ball (e. g. when two robots have almost the same approximated time to reach the ball), but provides a reasonable solution in the absence of other data.

Furthermore, in addition to any technical issues, we would like to highlight the fact that *B-Swift* has been the only team that used custom robot jerseys in this competition. The orange color combines black and yellow which are the colors usually worn by B-Human and rUNSWift, respectively. The jerseys are shown in Fig. 4.4.

During the competition at RoboCup 2018, *B-Swift* played four games and won all of them. As both teams have robust implementations of all required basic abilities, such as stable walking, ball recognition, and self-localization, all robots on the field were able to play together reliably according to the previously defined strategy. To sum up, one could say that the robots from *B-Human* and *rUNSWift* equally contributed to our success in this competition.

An overview of the results is given at the league's web site at `http://spl.robocup.org/standard-platform-league-results-2018/`.

Figure 4.3: The human team members of *B-Swift* who participated in RoboCup 2018 in Montréal

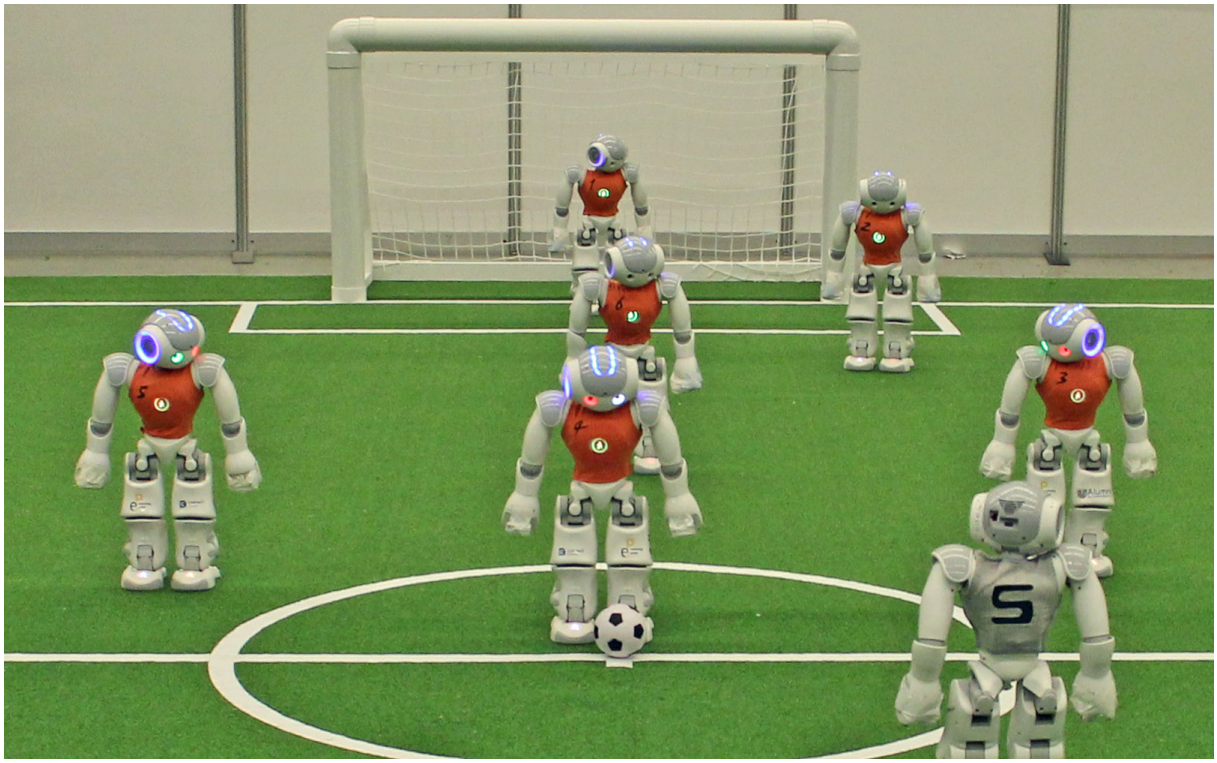


Figure 4.4: The robot team members of *B-Swift* before kickoff

Chapter 5

Acknowledgements

We gratefully acknowledge the support given by SoftBank Robotics. We also would like to thank our team sponsor CONTACT Software and our other sponsors TME, IGUS, Portescap, Alumni of the University of Bremen, engineering people, neuland, and MLP for funding parts of our project. Since B-Human 2018 did not start its software from scratch, we also want to thank the previous team members as well as the members of the GermanTeam 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:

Artistic Style: Source code formatting in the *AStyle for B-Human* text service on macOS.
(`http://astyle.sourceforge.net`)

AsmJit: A JIT assembler for C++.
(`https://github.com/asmjit/asmjit`)

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`)

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

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

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

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

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

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

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

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

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

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

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

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

qtpropertybrowser: Extends the Qt framework with a property browser.
(`https://github.com/qtproject/qt-solutions/tree/master/qtpropertybrowser`)

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

Walk2014Generator: The module `Walk2014Generator` is based on the class of the same name released by the team UNSW Australia as part of their code release. The team kindly gave us the permission to release our derived module under our license.
(`https://github.com/UNSWComputing/rUNSWift-2016-release`)

Bibliography

[1] Brett Browning, James Bruce, Michael Bowling, and Manuela Veloso. STP: Skills, tactics and plays for multi-robot control in adversarial environments. *IEEE Journal of Control and Systems Engineering*, 219:33–52, 2005.

[2] François Chollet et al. Keras. `https://keras.io`, 2015.

[3] Lotte de Koning, Juan Pablo Mendoza, Manuela Veloso, and Renë van de Molengraft. Skills, tactics and plays for distributed multi-robot control in adversarial environments. In Oliver Obst, Flavio Tonidandel, Hidehisa Akiyama, and Claude Sammut, editors, *RoboCup 2017: Robot World Cup XXI*, Lecture Notes in Artificial Intelligence. Springer, 2018. To appear.

[4] Thomas Röfer, Tim Laue, Yannick Bülter, Daniel Krause, Jonas Kuball, Andre Mühlenbrock, Bernd Poppinga, Markus Prinzler, Lukas Post, Enno Roehrig, René Schröder, and Felix Thielke. B-Human team report and code release 2017, 2017. Only available online: `http://www.b-human.de/downloads/publications/2017/coderelease2017.pdf`.

[5] Thomas Röfer, Tim Laue, Arne Hasselbring, Jesse Richter-Klug, and Enno Röhrig. B-human 2017 - team tactics and robot skills in the standard platform league. In *RoboCup 2017: Robot World Cup XXI*, Lecture Notes in Artificial Intelligence. Springer, to appear.

[6] Robert W. Rose. kerasify. `https://github.com/moof2k/kerasify`, 2016.

[7] Thomas Röfer. CABSL – C-based agent behavior specification language. In *RoboCup 2017: Robot World Cup XXI*, Lecture Notes in Artificial Intelligence. Springer, 2018. To appear.

[8] Thomas Röfer, Tim Laue, Michael Weber, Hans-Dieter Burkhard, Matthias Jüngel, Daniel Göhring, Jan Hoffmann, Benjamin Altmeyer, Thomas Krause, Michael Spranger, Oskar von Stryk, Ronnie Brunn, Marc Dassler, Michael Kunz, Tobias Oberlies, Max Risler, Uwe Schwiegelshohn, Matthias Hebbel, Walter Nisticó, Stefan Czarnetzki, Thorsten Kerkhof, Matthias Meyer, Carsten Rohde, Bastian Schmitz, Michael Wachter, Tobias Wegner, and Christine Zarges. GermanTeam RoboCup 2005, 2005.