Team Report and Code Release 2019

Thomas Röfer[1,2], Tim Laue[2],
Andreas Baude[2], Jan Blumenkamp[2], Gerrit Felsch[2], Jan Fiedler[2], Arne Hasselbring[2],
Tim Haß[2], Jan Oppermann[2], Philip Reichenberg[2], Nicole Schrader[2], Dennis Weiß[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: October 1, 2019

Contents

Chapter 1

Introduction

1.1 About Us

B-Human is a joint RoboCup team of the Universität 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 nine RoboCup German Open competitions, the RoboCup European Open 2016 competition, and has become RoboCup world champion seven times.

After becoming the runner-up world champion in 2018, we regained the world champion title at RoboCup 2019 in Sydney, Australia. We won all matches in the Champions Cup competition, achieving a goal difference of $46 : 1$. Furthermore, we won the *Mixed Team Tournament* for the third time in a row, in which we teamed up with *Berlin United – Nao Team Humboldt* from the Humboldt University of Berlin this year and formed the joint team *B&B*. In addition, we became the runner up in the overall ranking of this year's *Technical Challenges*.

The 2019 team consisted of the following persons (most of them are shown above):

Students: Andreas Baude, Jan Blumenkamp, Jan Buschmann, Gerrit Felsch, Jan Fiedler, Tim Haß, Arne Hasselbring, Lam Duy Le, Jan Oppermann, Philip Reichenberg, Simon Schirrmacher, Nicole Schrader, Dennis Weiß, Lars Wimmel.

Active Alumni: Jonas Kuball, Lukas Post, Jesse Richter-Klug, Felix Thielke.

Leaders: Tim Laue, Thomas Röfer.

Associated Researcher: Udo Frese.

1.2 About the Document

This document provides a survey of this year's code release, continuing the tradition of annual releases that was started several years ago. This document is based on the code releases of the last two years [23, 24] and aims to provide a complete description of the system that we used at RoboCup 2019. The major changes made to the system since last year are shortly enumerated in Section 1.3.

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. Chapter 3 describes our framework's architecture. Our image processing approaches are presented in Chapter 4, followed by the state estimation approaches in Chapter 5. Chapter 6 explains the use of our behavior description system based on *skills* and *cards* and gives an overview of the behavior used at RoboCup 2019. Surveys of the sensor reading and motion control parts of the system are given in Chapter 7 and Chapter 8 respectively. A description of our participation in the *Technical Challenges* and the *Mixed Team Tournament* is given in Chapter 9. Finally, in Chapter 10, our simulation and remote control environment *SimRobot* and the *B-Human User Shell* are presented.

1.3 Major Changes Since 2018

The major changes made since RoboCup 2018 are described in the following sections:

2 NAO V6
Our software works exclusively on the NAO V6.

3.2 Thread Layout
In order to exploit the four processor cores of the NAO V6, our framework has been made more flexible and the number of threads in our system was increased from two to four.

3.7 Logging
There is only a single online logger for all threads. Therefore, only a single log file is created and merging is not necessary.

4.1.2.1 Automatic Camera Calibration
A new camera calibration procedure based on the penalty area has been introduced.

4.2.2 Ball Detection
We use a new neural network architecture for detecting the ball, including prediction of the center of the ball within an image patch.

4.4 Detecting Other Robots
A deep neural network for detecting robots in the upper camera image has been developed. Furthermore, a new analytic method for the lower camera is used.

5.2.1 Local Ball Model
The distinction between stationary and moving balls and the integration of ball contacts with the own body have been improved.

5.2.3 Team Ball Model
The combined team ball model now features clustering in order to reject outliers.

5.5 Whistle Recognition
The whistle recognition has been changed to be more robust against loud noise. The vote of each robot in the team decision has been turned into a continuous number.

6.1 Skills and Cards
A new system for behavior specification has been introduced.

6.2 Behavior of 2019
Using the new behavior framework, much of its content has been rewritten with a focus on handling uncertainty and team play.

7.7 Foot Bumper Recognition
Collision detections with the foot bumpers are filtered based on the foot movement to prevent false positives.

7.8 Foot Pressure
After some time, the maximum values for the measured pressures are reinitialized. Also a prediction for the support foot switch is calculated.

7.9 Gyro Offset Detection
Offsets in the gyro values and a disconnect to the robot's hardware are detected and handled.

8.3 Walking
Smaller improvements in the odometry calculation and overall stability. Also, the balancing parameters are automatically adjusted.

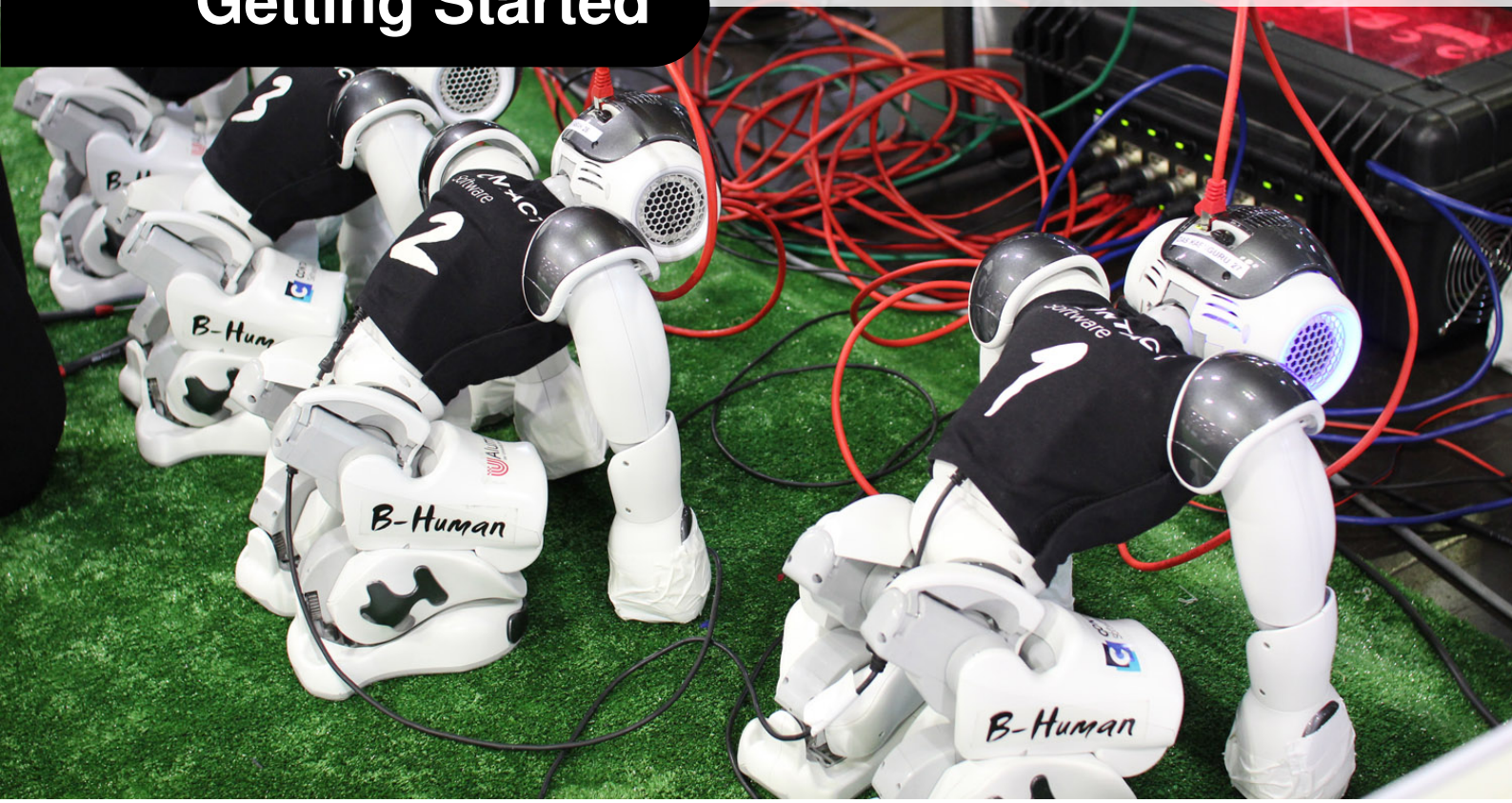
8.6 Getting Up
A new get up module with a very high reliability without further adjustments needed to work on different NAOs.

8.9 Energy Saving Stand
Joints are automatically adjusted while standing to prevent overheating.

8.10 Kicks
Stability and accuracy improvements for our long distance kicks with less calibration needed.

9 *Technical Challenges* and *Mixed Team Tournament*
In addition to the main soccer competition, we also participated in the *Directional Whistle Challenge*, *Open Research Challenge*, and – together with Berlin United – in the *Mixed Team Tournament*.

Chapter 2
Getting Started



This chapter describes how to use or code on Windows, macOS, and Linux. The code can be built for our simulator and for the actual robot (NAO V6 only). We first show how to compile the code before we describe how to setup a NAO robot to be used with our software. If you want to setup a NAO and run our code, you will need access to the **RoboCup-only** version of the operation system (currently version 2.8.5). The RoboCup Standard Platform League Technical Committee can give you access if you already joined the league or plan to do so. Without that special version of the operating system, our installation routine for the actual NAO will not work.

This manual assumes that you cloned the B-Human repository or downloaded a working copy. All paths mentioned in this description will be relative to the main directory of the working copy. The version numbers of software that we specify are the ones we currently use. That does not mean that newer or older versions will not work, but it is not guaranteed that they will.

2.1 Targets and Configurations

The B-Human project consists of the main build targets *SimRobot*, *Nao*, *bush*, and *Tests* (cf. Table 2.1). These targets can all be built in three different configurations[1] (cf. Table 2.2). The general idea is that *Debug* contains all debugging support, but is slow, *Release* sacrifices debugging support for speed, and *Develop* is a reasonable compromise between the two. The actual selection of features depends on the platform.

[1] For the *bush*, the configurations *Develop* and *Release* are identical. That's why the Xcode project does not offer a scheme to build the target in the configuration *Develop*.

Target	Description
SimRobot	The code is built to run in simulation. The whole build consists of our simulator *SimRobot* together with its libraries *SimRobotCore2* and *SimRobotEditor* and the robot control program *SimulatedNao*. The connection between the robot control program and the simulator is implemented in the library *Controller*, which also relies on building the two libraries *libqxt*[2] and *qtpropertybrowser*.
Nao	The code is built to run on the NAO.
bush	The *B-Human User Shell* is a tool to deploy and manage multiple robots at the same time.
Tests	Some unit tests.[3] Can be ignored.

Table 2.1: Build targets of the B-Human system.

	without assertions (NDEBUG)	debug symbols (compiler flags)	debug libs[4] (_DEBUG, compiler flags)	optimizations (compiler flags)	debugging support[5]
Release					
Nao	✓	×	×	✓	×
SimulatedNao	✓	×	×	✓	✓
SimRobot	✓	×	×	✓	✓
Develop					
Nao	×	×	×	✓	✓
SimulatedNao	×	✓	×	×	✓
SimRobot	✓	×	×	✓	✓
Debug					
Nao	×	✓	✓	×	✓
SimulatedNao	×	✓	✓	×	✓
SimRobot	×	✓	✓	×	✓

Table 2.2: Effects of the different build configurations. *SimRobot* represents everything in the simulation except for *SimulatedNao*.

2.2 Building the Code

2.2.1 Microsoft Windows

2.2.1.1 Prerequisites

- Microsoft Windows 10 64 bit Version 1903
- Microsoft Visual Studio Community 2019 Version 16.2.5. Installing the workload *Desktop development with C++* as well as the packages *MSVC v142 - VS 2019 C++ x86/x64 build tools*, *Windows 10 SDK 10.0.18362.0*, and *C++-ATL for v142 build tools (x86 & x64)* is sufficient.

[2] On macOS, it is compiled as a part of *Controller*.
[3] On macOS, this target is compiled and run through Xcode's *Test Navigator*.
[4] On Windows, see this link
[5] See Chapter 3

- Windows Subsystem for Linux (WSL). Execute the PowerShell script *Make/VS2019/installWSL.ps1*. You have to confirm that executing the script is allowed. The script will install the Windows Subsystem for Linux (unless it is already installed) and ask for a reboot of the computer. Afterwards, the script has to be executed again to download and install Ubuntu 18.04 into *Util/WSL* with all packages required.

2.2.1.2 Compiling

Run *Make/VS2019/generate.cmd* and open the solution *Make/VS2019/B-Human.sln* in Visual Studio. Select the desired configuration (*Develop* is a good start) and build one of the targets mentioned in Section 2.1[6]. Either *SimRobot* or *Utils/bush* can be selected as *StartUp Project* to run it.

2.2.2 macOS

2.2.2.1 Prerequisites

- macOS 10.14
- Xcode 11

2.2.2.2 Compiling

Xcode must have been executed at least once before to accept its license and to install its components. Run *Make/macOS/generate* and open the Xcode project *Make/macOS/B-Human.xcodeproj*. The schemes in the toolbar allow building the targets mentioned in Section 2.1 in different configurations.

When building for the NAO, a successful build will open a dialog to deploy the code to a robot (which will only work if the robot was already set up to work with our software). If the script *Make/macOS/login* was used before to login to a NAO, the IP address used will be provided as default. 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 script *Make/macOS/login* and used by the simulator scene *Config/Scenes/RemoteRobot.ros2*. The options are stored in *Make/macOS/copyfiles-options.txt*.

2.2.2.3 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.

[6] Instead of *Nao*, you can also build *Utils/copyfiles*, which allows to directly deploy the code to a NAO robot, if it is already set up for deploying our software.

Source code formatter. A system text service for formatting the currently selected text is available in the menu *Xcode→Services→AStyle for B-Human*.

2.2.3 Linux

2.2.3.1 Prerequisites

- Ubuntu 18.04
- Run the following command to install additional packages:

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

2.2.3.2 Compiling

To compile one of the targets mentioned above, enter the directory *Make/Linux* and run to following command for one combination of target and configuration mentioned in Section 2.1 (*Develop* is the default configuration):

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

As an alternative, there is also support for the integrated development environment CodeLite (*sudo apt install codelite*) that works similar to Visual Studio for Windows. To use it, run *Make/LinuxCodeLite/generate* and open *Make/LinuxCodeLite/B-Human.workspace* afterwards.

2.3 Setting Up the NAO

In the Standard Platform League, each team is assigned a unique *team number*. This number is used to identify the team in the *GameController* application, but it is also used to avoid conflicts in IP addresses and UDP ports. In the B-Human system, the team number must be entered into the file *Config/settings.cfg* as well as the team port (team number plus 10000). When deploying with the *bush*, the team number can also be specified there, but scripts usually look it up in the file *Config/settings.cfg*.

2.3.1 Subnet Configuration

All NAOs will be set up to use specific subnet addresses for wired and wireless communication. The two files *Install/Network/wired.service* and *Install/Network/wireless.service* are the templates that will be instantiated when creating individual configuration files for each robot. By default, they are set up to use the network *192.168.x.y* for wired and *10.0.x.y* for wireless communication (usually the setup used at RoboCup events). You have to edit these files if you want to use different subnets **before** you create configurations for individual robots.

2.3.2 Wireless Network Configuration

Wireless configurations are stored in *Install/Network/Profiles*. The one named *default* will be used after the initial setup. Other profiles can be selected later, when deploying the software

to the robot. The profiles will not only be copied to the NAO during its initial setup, but also be updated while deploying our software, i. e. the contents of the directory can be changed later.

2.3.3 Installing the RoboCup-only Operating System

Follow these instructions ⍁ to flash the NAO with the **RoboCup-only** operating system image (use the option *Factory reset*). When the installation is complete, connect your computer to the NAO with an Ethernet cable and press the chest button to determine NAO's ip address. It usually takes a while until it has selected one that follows the pattern *169.254.x.y*.

2.3.4 Creating a Robot Configuration

Only once for each NAO and while being connected to that NAO, run

```
Install/createRobot [-t <team>] -r <robot> -i <ip> <name>
```

<*team*> is your team's number (the third number of the robot's target ip address). If omitted, the team number from the file *Config/settings.cfg* is used. <*robot*> is the number of the robot (the fourth number of its target ip address). <*ip*> is its current ip address. <*name*> is the name you will use to refer to the NAO in the future. The script will create files in *Install/Robots/*<*name*> and *Config/Robots/*<*name*> and update some existing configuration files.

2.3.5 Installing the NAO

To actually install a NAO, run

```
Install/installRobot <ip>
```

<*ip*> is the current ip address of the robot. Follow the instructions on the screen. When the script is done, it will reboot the robot. The NAO is now fully set up to be used with our software. Please note that the robot will have different ip addresses after the reboot, e. g. the Ethernet adapter will use *192.168.*<*team*>*.*<*robot*> if you did not change the subnet configuration.

2.4 Deploying the Software

Deploying the software to a NAO means copying the directory *Config* (without the subdirectories *Images*, *Keys*, *Logs*, and *Scenes*) together with the executable *bhuman* to */home/nao/Config* on the robot. In addition, the network profiles in *Install/Network/Profiles* are copied to */home/nao/Profiles*. The latter does not automatically update the current network profile.

The software is deployed by executing the script *Make/Common/copyfiles* that is also linked from some other subdirectories under *Make*:

```
usage: copyfiles [Release|Develop|Debug] [<ipaddress>|(-r <playernumber>
    <ipaddress>)*] {options}
  options:
    -b                      restart bhuman
    -c <color>              set team color to blue, red, yellow, black, white,
                            orange, purple, brown, or gray
    -d                      delete all log files before copying
    -h | --help | /h | /?   print this text
    -l <location>           set location
    -m <number>             set magic number for teamcomm (0-255). Set -1 for
```

```
                          random.
  -nc                     never compile
  -nr                     do not check whether target is reachable
  -o <port>               overwrite team port
  -p <number>             set player number
  -r <n> <ip>             copy to <ip>, and set playernumber to <n> (one -r
                          per robot)
  -s <scenario>           set scenario
  -t <number>             set team number
  -v <percent>            set NAO's volume
  -w <profile>            set wireless profile
examples:
  ./copyfiles Develop 192.168.5.14 -p 1
  ./copyfiles Release -r 1 10.0.5.14 -r 3 10.0.0.2
```

Without the option *-nc*, *copyfiles* will build the code before deploying it. The script is also called by Visual Studio when building the project *Utils/copyfiles* and by Xcode when building the scheme *Nao*. In both cases, the configuration (*Debug*/*Develop*/*Release*) is automatically provided to the script. In Xcode, there is an additional option *-a*, which prevents the deploy dialog from being shown in the future. Instead, the current parameters 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.

The standard method for deploying our software to multiple NAOs is the *B-Human User Shell* (*bush*, see Section 10.2). It is a graphical frontend for *copyfiles* with a lot of additional functionality.

2.5 Working with the NAO

After pressing the chest button to turn on the NAO, it takes about 25 seconds until our software runs. It is important not to move the NAO while it is booting up, because NAOqi calibrates the gyroscopes during this time. Our software will give an acoustic alarm if the gyroscope was not calibrated correctly. In that case, the robot has to be rebooted.

To connect to the NAO, the subdirectories of *Make* contain a *login* script for each supported platform. The only parameter of that script is the IP address of the robot to login. It automatically uses the appropriate SSH key to login. In addition, the IP address specified is written to the file *Config/Scenes/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 *bhuman* via SSH. Those scripts are copied to the NAO upon installing the B-Human software.

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 as a background service. The script *Make/Common/copyfiles* always stops *bhuman* before deploying. If *copyfiles* is started with option *-b*, it will restart *bhuman* after all files were copied.

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

reboot reboots the NAO.

2.6 Working with SimRobot

On Windows and macOS, SimRobot can either be started from the development environment or by starting a scene description file in *Config/Scenes*[7]. In the first case, a scene description file has to be opened manually, whereas it will already be loaded in the latter case. On Linux, just run *Build/SimRobot/Linux/<configuration>/SimRobot*, either from the shell or from 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 is automatically executed if it exists, setting up the robot(s) as desired. The name of the script file is the same as the name of the scene description file but with the extension *.con*. Together with the ability of SimRobot to store the window layout, the software can be configured to always start with a setup suitable for a certain task.

Although any object in the scene graph can be opened, only displaying certain entries in the object tree makes sense, namely the main scene *RoboCup*, the objects in the group *RoboCup/robots*, and all other views, which can be found under the tree roots that are marked with the B-Human-Logo.

To connect to a real NAO, open the scene *Config/Scenes/RemoteRobot.ros2*. A prompt will appear to enter the NAO's IP address.[8] In a remote connection, the simulation scene is usually empty. Therefore, it is not necessary to open a scene view, but you can open any of the other views.

2.7 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. Section 2.7.2) and the cameras (cf. Section 2.7.3). Checking those calibrations from time to time is important, especially after a repair.

In addition to that, the B-Human software uses four color classes (cf. Section 4.1.5) which have to be calibrated, too (cf. Section 2.7.4). Changing locations or lighting conditions might require them to be adjusted.

2.7.1 Overall Physical Calibration

The physical calibration process can be split into two 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

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

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

in correct position. Our software assumes that the centers of the feet are 10 cm apart from each other. The feet should not be rotated when the robot is in the calibration stand, and its standing height should be 23.5 cm (cf. Section 2.7.2). The second step is the camera calibration (cf. Section 2.7.3). This step also measures the tilt of the body with respect to the feet.

2.7.2 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 inverse kinematics to do the same (cf. Section 8.3.5). When switching between those two methods, it is necessary to save the `JointCalibration`, redeploy the NAO and restart *bhuman*. Otherwise, the changes done previously will not be used.

Before changing joint offsets, the robot has to be set in a standing position with fixed joint angles. Otherwise, the balancing mechanism of the motion engine might move the legs, messing up the joint calibrations. Using the `JointCalibrator` this happens automatically, for manually adjusting the offsets this can be done with:

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

Either way, when the calibration is finished it should be saved:

```
save representation:JointCalibration
```

Manually Adjusting Joint Offsets

There are two ways to adjust the joint offsets: either by requesting the *JointCalibration* representation with a *get* command:

```
get representation:JointCalibration
```

modifying the calibration returned and then setting it, or by using a data view (cf. Section 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
```

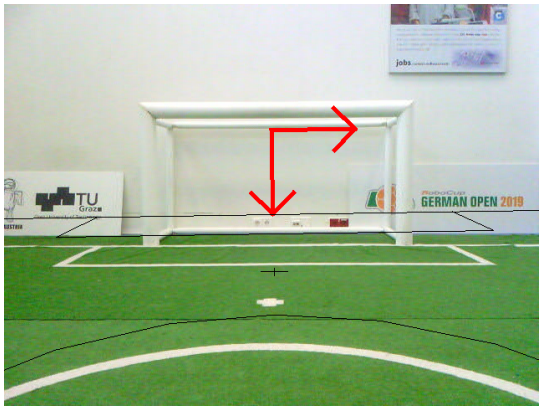
Afterwards, the translation and rotation of the feet can be modified. Again either with

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

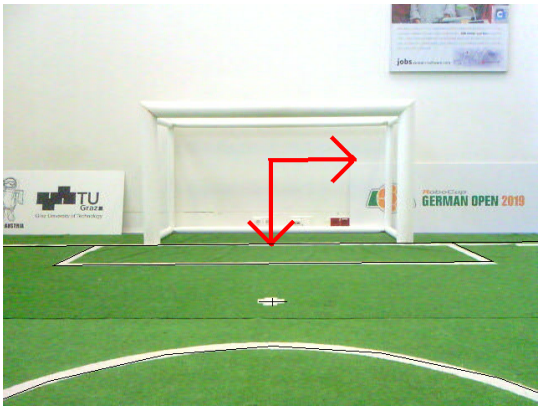
or with:

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

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



(a) Projected field lines before calibration.



(b) Projected field lines after calibration.

Figure 2.1: Effect of a camera calibration to the projection between world and image coordinates.

2.7.3 Camera Calibration

For calibrating the cameras (cf. Section 4.1.2.1), one of the modules `AutomaticCameraCalibratorPA` or `AutomaticCameraCalibrator` can be used. The first one requires only a penalty area including the penalty mark for the calibration, the second one a complete half of the field. Figure 2.1 illustrates the importance of calibrating the camera with any one of the two.

Using the AutomaticCameraCalibratorPA

For calibration with the `AutomaticCameraCalibratorPA`, the NAO must be initially positioned next to one of the two corners of the penalty area and look diagonally at it. Figure 2.2a shows this initial position. After the NAO is positioned correctly, the calibration can be performed by following the procedure below.

1. Initialize the calibration process with:

   ```
   call Calibrators/CameraPA
   ```

   Furthermore this will print commands to the simulator console that will be needed later on.

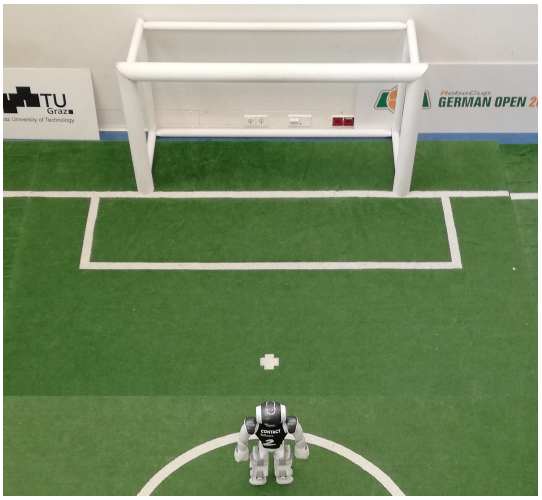
2. To start the collection of samples use the command:

   ```
   dr module:AutomaticCameraCalibratorPA:start
   ```

   Sampling is always performed in the upper camera first and when the necessary samples are collected the head is tilted upwards to also search for samples in the lower camera. When samples were found with both cameras, the NAO turns its head to the next position. Pick up the NAO and place it such that all required lines are inside its field of view again. These are the baseline, the front penalty area line and the closer short connecting line.

3. As soon as the NAO has turned its head forward again, the second calibration step begins. Now the robot has to see the penalty mark as well as the baseline and the front penalty area line. Place the NAO so that he can see these features, e. g. as shown in

(a) Calibration position for the first step.



(b) Calibration position for the second step.

Figure 2.2: The different calibration positions using the `AutomaticCameraCalibratorPA`

Fig. 2.2b. Again, pick up the NAO and realign it as soon as the head has turned to a new position.

4. A blinking head LED during sampling indicates that not all necessary features are visible and the NAO has to be realigned. If the left eye is red (normally it is white), all necessary features are seen but discarded. In this case some waiting is needed before they are accepted.

5. After all necessary samples have been collected, the optimization process begins. Wait until it converges and save the calibration using:

   ```
   save representation:CameraCalibration
   ```

Using the AutomaticCameraCalibrator

For calibration with the `AutomaticCameraCalibrator`, the NAO must be positioned in the middle of the own half of the center circle, facing the opponent's goal. The inner tips of the heels must correspond to the rear edge of the line. Figure 2.3a shows this initial position. After the NAO is positioned correctly, the calibration can be performed by following the procedure below.

1. Initialize the calibration process with:

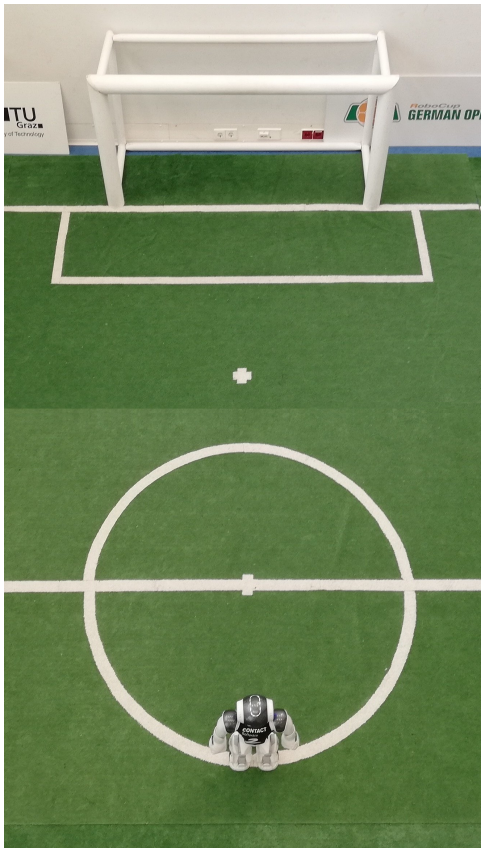
   ```
   call Calibrators/Camera
   ```

   Furthermore this will print commands and help to the simulator console that will be needed later on.

2. To start collecting sample points use:

   ```
   dr module:AutomaticCameraCalibrator:start
   ```

   Wait for the console 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) (cf. Fig. 2.3b).



(a) The initial calibration position.

(b) The view after the control start with gathered samples.

(c) The stage after optimization.

Figure 2.3: The interesting camera calibration stages using the `AutomaticCameraCalibrator`.

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

4. Run the automatic calibration process, wait until the optimization has converged, and finally save the calibration using following commands:

```
dr module:AutomaticCameraCalibrator:optimize
save representation:CameraCalibration
```

2.7.4 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. Alternatively it is possible to use a data view (cf. Section 10.1.4.5):

```
vd representation:CameraSettings
```

The most important parameters are:

autoExposure: Enable (`true`) / disable (`false`) the automatism for exposure. When playing under static lighting conditions, this parameter should always be disabled, since the automation will often choose higher values than necessary, which will result in blurry images. However, for dynamic lighting conditions, using the automatism of the camera driver may be a necessity.

autoExposureBrightness: The brightness the auto exposure tries to achieve. The available interval is [-255, 255].

exposure: The exposure time used when auto exposure is disabled. The available interval is [0, 1048575]. Be aware that high exposures lead to blurred images.

gain: The gain in the range [0, 1023]. Brightens the image and has only an effect when the automatism for exposure is disabled. Be aware that high gain values lead to noisy images.

saturation: The saturation in the range [0, 255]. Adjusts the color intensity and should be set such that the green in the image looks natural.

contrast: The contrast in the range [0, 255].

autoWhiteBalance: Enable (`true`) / disable (`false`) the automatism for white balance.

Furthermore 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 executing one of the commands, new parameters for the white balance are printed to the console. These have to be copied to the parameters (red|green|blue)Gain. Ensure that the automatic white balance is deactivated, otherwise it will have no effect. Alternatively to the one-time auto white balance, these parameters can also be used to adjust the white balance manually.

(a) Image with improper white balance. (b) Image with better settings for white balance.

Figure 2.4: The effect of the white balance to the obtained camera image.

After setting up the parameters of the camera driver, the parameters of the color classes must be updated (cf. Section 4.1.5). To do so, one needs to open the views with the segmented upper and lower camera images and the color calibration view (cf. Section 10.1.4.1 and Section 10.1.4.1). After finishing the color class calibration and saving the current parameters, copyfiles/bush (cf. Section 2.4) can be used to deploy the current settings. Ensure the updated files *cameraSettings.cfg* and *fieldColors.cfg* are stored in the correct location.

2.8 Directory Structure

2.8.1 Main Directories

Directory	Contents
Build	All files created during builds are located in this directory. The directory structure follows the pattern *Build/<os>/<target>/<configuration>*. Deleting the directory will remove all build files. The directory initially does not exist.
Config	All configuration files are located in this directory. Some of the subdirectories are described in Section 2.8.2. *Config/Scenes* contains the simulation scenes (*.ros2*) together with the console scripts that are executed when the simulation is started (*.con*)
Install	Scripts and other files for setting up the NAO robot.
Make	Scripts and project files for software development. *Make/<os/ide>* contains project files for a specific IDE or build system. *Make/Common* contains the shared part of the build system. It also contains scripts for deploying the code and downloading log files that are used by other tools. *Make/Hooks/installHooks[.cmd]* can be used to install git hooks that will automatically call the script *Make/<os/ide>/generate* after most git operations.
Src	All source files of the B-Human system.
Util	3rd-party code that is not part of the B-Human system.

2.8.2 Configuration File Search Path

All configuration files are searched along a path that contains directories that can depend on the name of a robot (separately for head and body, so mixing those is possible), the location where the robot is used (e. g. outdoor field vs. indoor field), and the scenario in which it is used (e. g. normal game vs. technical challenge):

- *Config/Robots/<head name>/Head*
- *Config/Robots/<body name>/Body*
- *Config/Robots/<head name>/<body name>*
- *Config/Locations/<location>*
- *Config/Scenarios/<scenario>*
- *Config/Robots/Default*
- *Config/Locations/Default*
- *Config/Scenarios/Default*
- *Config*

In simulation, the name of both head and body is *Nao*. The current location and scenario are defined in the file *Config/settings.cfg*. They can also be changed while deploying the code to an actual NAO (cf. Section 2.4).

Chapter 3

Architecture

The B-Human architecture [22] is based on the framework of the GermanTeam 2007 [21], adapted to the NAO. This chapter summarizes the major features of the architecture: binding, threads, modules and representations, communication, and debugging support.

3.1 Binding

On the NAO, the B-Human software consists of a single binary file and a number of configuration files, all located in */home/nao/Config*. When the program is started, it first contacts *LoLA* to retrieve the serial number of the body that is then mapped to a body name using the file *Config/Robots/robots.cfg*. The Unix hostname is used as the robot's head name. Then, the connection to *LoLA* is closed again and the program forks itself to start the main robot control program. After that has terminated, another connection to *LoLA* is established to sit down the robot if necessary, switch off all joints, and display the exit status of the main program in the eyes (blue: terminated normally, red: crashed).

In the main robot control program, the module `NaoProvider` exchanges data with *LoLA* following the protocol defined by SoftBank Robotics for RoboCup teams. For efficiency, only the first packet received is actually parsed, determining the addresses of all relevant data in that packet and using these addresses for later packets rather than parsing them again. Similarly, a template for packets sent to *LoLA* is only created once and the actual data is patched in in each frame. The `NaoProvider` uses blocking communication, i. e. the whole thread *Motion* (cf. Section 3.2) waits until a packet from *LoLA* is received before it continues.

3.2 Threads

Most robot control programs use concurrent processes and threads. Of those, we only use threads in order to have a single shared memory context. The number of parallel threads is best dictated by external requirements coming from the robot itself or its operating system. The NAO provides images from each camera at a frequency of 30 Hz and accepts new joint angle commands at 83 Hz. For handling the camera images, there would actually have been two options: either to have two threads each of which processes the images of one of the two cameras and a third one that collects the results of the image processing and executes world modeling and behavior control, or to have a single thread that alternately processes the images of both cameras and also performs all further steps. We are currently using the first approach in order to better exploit the multi-core hardware of the robot. This makes it possible to process both images in parallel and leads to significantly more available computation time per image. These threads run at 30 Hz and handle one camera image each. They both trigger a third thread, which processes the results as described above. This one runs with 60 Hz in parallel to the image processing. In addition, there is a thread that runs at the motion frame rate of the NAO, i. e. at 83 Hz. Another thread performs the TCP communication with a host PC for the purpose of debugging.

This results in the five threads *Upper*, *Lower*, *Cognition*, *Motion* and *Debug* actually used in the B-Human system (cf. Fig. 3.1). The perception threads *Upper* and *Lower* receive camera images from *Video for Linux*. In addition, they receive data from the thread *Cognition* about the world model as well as sensor data from the thread *Motion*. They processes the images and send the detection results to the thread *Cognition*. This thread actually uses this information together with sensor data from the thread *Motion* for world modeling and behavior control and sends high-level motion commands to the thread *Motion*. This one actually executes these commands by generating the target angles for the 25 joints of the NAO. It sends these target angles through the `NaoProvider` to the NAO's interface *LoLA*, and it receives sensor readings such as the actual joint angles, body acceleration and gyro measurements, etc. In addition, *Motion* reports about the motion of the robot, e. g. by providing the results of dead reckoning. The thread *Debug* communicates with the host PC. It distributes the data received from it to the other threads and it collects the data provided by them and forwards it back to the host machine. It is inactive during actual games.

3.3 Modules and Representations

A robot control program usually consists of several modules, each performing a certain task, e. g. image processing, self-localization, or walking. Modules require a certain input and produce a certain output (so-called *representations*). Therefore, they have to be executed in a specific order to make the whole system work. The module framework introduced in [21] simplifies the definition of the interfaces of modules and automatically determines the sequence in which the modules must be executed. It consists of the *blackboard*, the *module definition*, and a visualization component (cf. Section 10.1.4.5).

3.3.1 Blackboard

The blackboard [10] is the central storage for information, i. e. for the representations. Each thread is associated with its own instance of the blackboard. Representations are transmitted through inter-thread communication if a module in one thread requires a representation that is

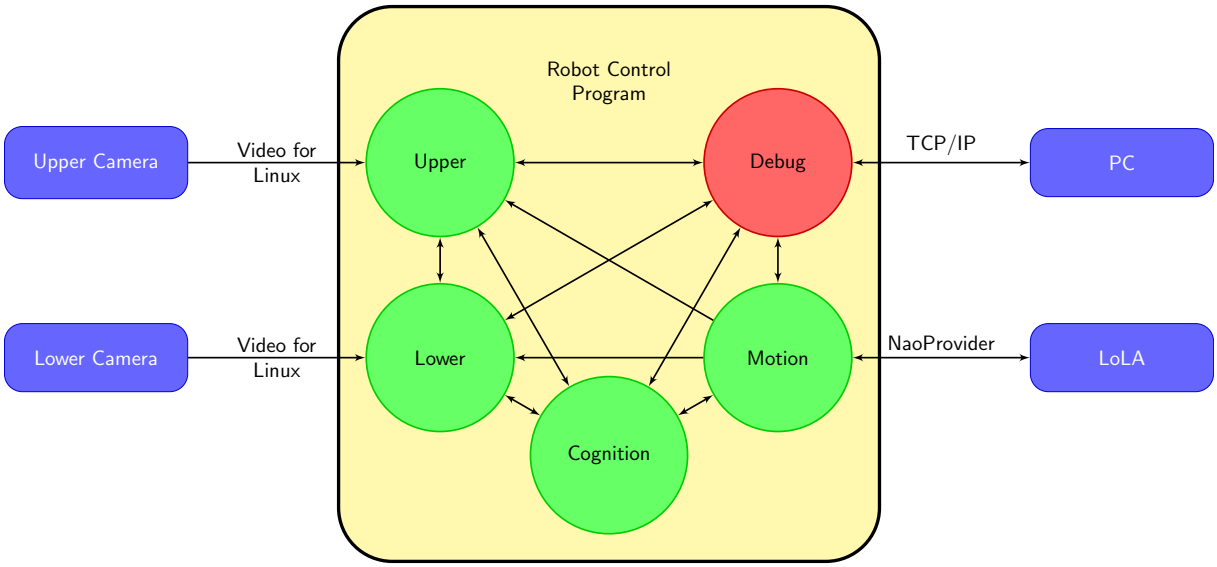


Figure 3.1: The threads used on the NAO

provided by a module in another thread. The blackboard itself is a map that associates names of representations to reference-counted instances of these representations. It only contains entries for the representations that at least one of the modules in associated thread actually requires or provides.

3.3.2 Module Definition

The definition of a module consists of three parts: the module interface, its actual implementation, and a statement that allows instantiating the module. Here is an example:

```
MODULE(SimpleBallLocator,
{,
  REQUIRES(BallPercept),
  REQUIRES(FrameInfo),
  PROVIDES(BallModel),
  DEFINES_PARAMETERS(
  {,
    (Vector2f)(5.f, 0.f) offset,
    (float)(1.1f) scale,
  }),
});

class SimpleBallLocator : public SimpleBallLocatorBase
{
  void update(BallModel& ballModel)
  {
    if(theBallPercept.wasSeen)
    {
      ballModel.position = theBallPercept.position * scale + offset;
      ballModel.wasLastSeen = theFrameInfo.time;
    }
  }
}

MAKE_MODULE(SimpleBallLocator, modeling);
```

The module interface defines the name of the module (e. g. `SimpleBallLocator`), the repre-

sentations that are required to perform its task, the representations provided by the module, and the parameters of the module, the values of which can either be defined in place or loaded from a file. The interface basically creates a base class for the actual module following the naming scheme <ModuleName>Base. The actual implementation of the module is a class that is derived from that base class. The following statements are available in a module definition:

REQUIRES declares that this module has read-only access to this representation in the blackboard (and only to this) via the<representationName>. As is described in Section 3.3.3, modules can expect that all their required representations have been updated before any of their provider methods is called. If a required representation is provided by multiple threads, a so-called alias must be used to specify which representation it uses. An alias is a representation that has a thread name prefix before the actual name, e. g. <thread><representation>. The representation must be derived from the actual representation without adding members to it.

USES declares that this module accesses a representation, which does not necessarily have to be updated before the module is run. This should only be used to resolve conflicts in the configuration. Note that USES is not considered when exchanging data between threads.

PROVIDES declares that this module will update this representation. It must define an update method for each representation that is provided. For each representation the execution time can be determined (cf. Section 3.6.7) and it can be sent to a host PC or even altered by it.

PROVIDES_WITHOUT_MODIFY does exactly the same as PROVIDES, except that the representation can not be inspected or altered by the host PC.

DEFINES_PARAMETERS and LOADS_PARAMETERS allow the modifiable parameterization of modules, as described in Section 3.3.5. It is recommended to use this for all parameters.

Finally, the MAKE_MODULE statement allows the module to be instantiated. Its second parameter defines a category that is used for a more structured visualization of the module configuration (cf. Section 10.1.4.5). The optional third parameter can override the static function that delivers the list of required and provided representations to the system (cf. Section 3.3.3). This can be used to add requirements to a module that can not be part of the module definition, as is needed by the behavior framework (cf. Section 6.1). While the module interface is usually part of the header file, the MAKE_MODULE statement has to be part of the implementation file.

MODULE is a macro that gets all the information about the module as parameters, i. e. they are all separated by commas. The macro ignores its second and its last parameter, because by convention, these are used for opening and closing curly brackets. These let some source code formatting tools indent the definitions as a block. Currently, MODULE is limited to up to 90 definitions between the curly brackets. When the macro is expanded, it creates a lot of hidden functionality. Each entry that references a representation makes sure that it is created in the blackboard when the module is constructed and freed when the module is destroyed. The information that a module has certain requirements and provides certain representations is not only used to generate a base class for that module, but is also available for sorting the providers, and can be requested by a host PC. On a host PC, the information can be used to change the configuration and for visualization (cf. Section 10.1.4.5). If a MessageID id<representation> exists, the representation can also be logged (cf. Section 3.7).

If a representation provided defines a parameterless method draw, that method will be called after the representation was updated. The method is intended to visualize the representation

using the techniques described in Section 3.6.3. If the representation defines a parameterless method `verify`, that method will be called in *Debug* and *Develop* builds after the representation was updated as well. A `verify` method should contain `ASSERT`s that check whether the contents of the representation are plausible. Both methods are only called if they are defined in the representation itself and not if they are inherited from its base class.

3.3.3 Configuring Providers and Threads

Since modules can provide more than a single representation, the configuration has to be performed on the level of providers. For each representation, it can be selected which module provides it or that it is not provided at all. Normally, the configuration is read from the file *Config/Scenarios/<scenario>/threads.cfg* during the start-up of the robot control program, but it can also be changed interactively when the robot has a debug connection to a host PC using the command *mr* (cf. Section 10.1.6.3). This file defines which threads exist at all and which providers run in them. This allows for completely different configurations in different scenarios.

The configuration does not specify the sequence in which the providers are executed. This sequence is automatically determined at runtime based on the rule that all representations required by a provider must already have been provided by other providers before, i. e. those providers have to be executed earlier. This is calculated in the thread *Debug*. Only valid configurations are then sent to all other threads.

In some situations, it is required that a certain representation is provided by a module before any other representation is provided by the same module, e. g., when the main task of the module is performed in the `update` method of that representation, and the other `update` methods rely on results computed in the first one. Such a case can be implemented by both requiring and providing a representation in the same module.

3.3.4 Default Representations

During the development of the robot control software, it is sometimes desirable to simply deactivate a certain provider or module. As mentioned above, it can always be decided not to provide a certain representation, i. e. all providers generating the representation are switched off. However, not providing a representation typically makes the set of providers inconsistent, because other providers rely on that representation, so they would have to be deactivated as well. This has a cascading effect. In many situations, it would be better to be able to deactivate a provider without any effect on the dependencies between the modules. In this case it is possible to enter representations to a separate list in the thread configuration file. It will never change any of the representations – so they basically remain in their initial state – but it will make sure that they exist, and thereby, all dependencies can be resolved. However, in terms of functionality, a configuration using that list is not complete.

3.3.5 Parameterizing Modules

Modules usually need some parameters to function properly. Those parameters can also be defined in the module's interface description. Parameters behave like protected class members and can be accessed in the same way. Additionally, they can be manipulated from the console using the commands *get parameters:<ModuleName>* or *vd parameters:<ModuleName>* (cf. Section 10.1.6.3).

There are two different parameter initialization methods. In the hard-coded approach, the initialization values are specified as part of the module definition. They are defined using the DEFINES_PARAMETERS macro. This macro is intended for parameters that may change during development but will never change again afterwards. In contrast, loadable parameters are initialized to values that are loaded from a configuration file upon module creation, i. e. the initialization values are not specified in the source file. These parameters are defined using the LOADS_PARAMETERS macro. By default, parameters are loaded from a file with the same base name as the module, but starting with a lowercase letter[1] and the extension *.cfg*. For instance if a module is named SimpleBallLocator, its configuration file is *simpleBallLocator.cfg*. This file can be placed anywhere in the usual configuration file search path (cf. Section 2.8.2). It is also possible to assign a custom name to a module's configuration file by passing the name as a parameter to the constructor of the module's base class.

Only either DEFINES_PARAMETERS or LOADS_PARAMETERS can be used in a module definition. They can both only be used once. Their syntax follows the definition of generated streamable classes (cf. Section 3.4.3). Parameters may have any data type as long as it is streamable (cf. Section 3.4.5).

3.4 Serialization

In most applications, it is necessary that data can be serialized, i. e. transformed into a sequence of bytes. While this is straightforward for data structures that already consist of a single block of memory, it is a more complex task for dynamic structures, e. g. lists, trees, or graphs. Our implementation for streaming data follows the ideas introduced by the C++ iostreams library, i. e., the operators $<<$ and $>>$ are used to implement the process of serialization. In contrast to the iostreams library, our implementation guarantees that data is streamed in a way that it can be read back without any special handling, even when streaming into and from text files, i. e. the user of a stream does not need to know about the representation that is used for the serialized data (cf. Section 3.4.1).

On top of the basic streaming class hierarchy, it is also possible to derive classes from class Streamable and implement the mandatory method serialize(In*, Out*). In addition, the basic concept of streaming data was extended by a mechanism to register information on the structure of serializable datatypes. This information is used to translate between the data in binary form and a human-readable format that reflects the hierarchy of a data structure, its variables, and their actual values.

As a third layer of serialization, two macros allow defining classes that automatically implement the method serialize(In*, Out*) and datatype registration.

3.4.1 Streams

The foundation of B-Human's implementation of serialization is a hierarchy of streams. As a convention, all classes that write data into a stream have a name starting with "Out", while classes that read data from a stream start with "In". In fact, all writing classes are derived from the class Out, and all reading classes are derivations of the class In. All classes support reading or writing basic datatypes with the exceptions of long, unsigned long, and size_t. They also provide the ability to read or write raw binary data.

[1] Actually, if a module name begins with more than one uppercase letter, all initial uppercase letters but the last one are transformed to lowercase, e. g. the module LEDHandler would read the file *ledHandler.cfg* if it would read its parameters from a file.

All streaming classes derived from `In` and `Out` are composed of two components: One for reading/writing the data from/to a physical medium and one for formatting the data from/to a specific format. Classes writing to physical media derive from `PhysicalOutStream`, classes for reading derive from `PhysicalInStream`. Classes for formatted writing of data derive from `StreamWriter`, classes for reading derive from `StreamReader`. The composition is done by the `OutStream` and `InStream` class templates.

A special case are the `OutMap` and the `InMap` streams. They only work together with classes that are derived from the class `Streamable`, because they use the structural information that is gathered in the `serialize` method. They are both directly derived from `Out` and `In`, respectively.

Currently, the following classes are implemented:

PhysicalOutStream: Abstract class

- **OutFile:** Writing into files
- **OutMemory:** Writing into memory
- **OutMemoryForText:** Writing into memory, ensuring that it is null-terminated
- **OutMessageQueue:** Writing into a MessageQueue

StreamWriter: Abstract class

- **OutBinary:** Formats data binary
- **OutText:** Formats data as text
- **OutTextRaw:** Formats data as raw text (same output as “cout”)

Out: Abstract class

- **OutStream<PhysicalOutStream, StreamWriter>:** Abstract template class
  - **OutBinaryFile:** Writing into binary files
  - **OutTextFile:** Writing into text files
  - **OutTextRawFile:** Writing into raw text files
  - **OutBinaryMemory:** Writing binary into memory
  - **OutTextMemory:** Writing into memory as text
  - **OutTextRawMemory:** Writing into memory as raw text
  - **OutBinaryMessage:** Writing binary into a MessageQueue
  - **OutTextMessage:** Writing into a MessageQueue as text
  - **OutTextRawMessage:** Writing into a MessageQueue as raw text
- **OutMap:** Writing into a stream in configuration map format (cf. Section 3.4.6). This only works together with serialization (cf. Section 3.4.5), i. e. a streamable object has to be written. This class cannot be used directly.
  - **OutMapFile:** Writing into a file in configuration map format
  - **OutMapMemory:** Writing into a memory area in configuration map format

PhysicalInStream: Abstract class

- **InFile:** Reading from files
- **InMemory:** Reading from memory
- **InMessageQueue:** Reading from a MessageQueue

StreamReader: Abstract class

InBinary: Binary reading

InText: Reading data as text

InConfig: Reading configuration file data from streams

In: Abstract class

InStream<PhysicalInStream, StreamReader>: Abstract class template

InBinaryFile: Reading from binary files

InTextFile: Reading from text files

InConfigFile: Reading from configuration files

InBinaryMemory: Reading binary data from memory

InTextMemory: Reading text data from memory

InConfigMemory: Reading config-file-style text data from memory

InBinaryMessage: Reading binary data from a MessageQueue

InTextMessage: Reading text data from a MessageQueue

InMap: Reading from a stream in configuration map format (cf. Section 3.4.6). This only works together with serialization (cf. Section 3.4.5), i. e. a streamable object has to be read. This class cannot be used directly.

InMapFile: Reading from a file in configuration map format

InMapMemory: Reading from a memory area in configuration map format

3.4.2 Streaming Data

To write data into a stream, *Tools/Streams/OutStreams.h* must be included, a stream must be constructed, and the data must be written into the stream. For example, to write data into a text file, the following code would be appropriate:

```
#include "Tools/Streams/OutStreams.h"
// ...
OutTextFile stream("MyFile.txt");
stream << 1 << 3.14 << "Hello Dolly" << endl << 42;
```

The file will be written into the configuration directory, e. g. *Config/MyFile.txt* on the PC. It will look like this:

```
1 3.14 "Hello Dolly"
42
```

As spaces are used to separate entries in text files, the string “Hello Dolly” is enclosed in double quotes. The data can be read back using the following code:

```
#include "Tools/Streams/InStreams.h"
// ...
InTextFile stream("MyFile.txt");
int a, d;
double b;
std::string c;
stream >> a >> b >> c >> d;
```

It is not necessary to read the symbol `endl` here, although it would also work, i. e. it would be ignored.

For writing to text streams without the separation of entries and the addition of double quotes, `OutTextRawFile` can be used instead of `OutTextFile`. It formats the data such as known from the ANSI C++ `cout` stream. The example above is formatted as following:

```
13.14Hello Dolly
42
```

To make streaming independent of the kind of the stream used, it could be encapsulated in functions. In this case, only the abstract base classes `In` and `Out` should be used to pass streams as parameters, because this makes the functions usable independently from the type of the streams:

```
#include "Tools/Streams/InOut.h"

void write(Out& stream)
{
  stream << 1 << 3.14 << "Hello Dolly" << endl << 42;
}

void read(In& stream)
{
  int a, d;
  double b;
  std::string c;
  stream >> a >> b >> c >> d;
}
// ...
OutTextFile stream("MyFile.txt");
write(stream);
// ...
InTextFile stream("MyFile.txt");
read(stream);
```

3.4.3 Generating Streamable Classes

Values of basic types are not the only kind of data that can be streamed. Instances of classes can be streamed as well. However, in contrast to basic data types, the streaming framework cannot know how to stream an object that is composed of several member variables. Therefore, additional information must be provided that allows serializing and de-serializing the instances of a class and even transmitting and querying its specification. The underlying details are given in Section 3.4.5. Here, we only describe how two macros (defined in *Tools/Streams/AutoStreamable.h*) generate this extra information for a streamable `struct`[2] and optionally also initialize its member variables. The first is:

```
STREAMABLE(<class>,
{ <header>,
  <comma-separated-declarations>,
});
```

The second is very similar:

```
STREAMABLE_WITH_BASE(<class>, <base>, ...
```

The parameters have the following meaning:

class: The name of the struct to be declared.

[2]Meaning, the default access for member variables is `public`.

base: Its base class. It must be streamable itself.

header: Everything that can be part of a class body except for the member variables that should be streamable. Please note that this part must not contain commas that are not surrounded by parentheses, because C++ would consider it to be more than a single macro parameter otherwise. A workaround is to use the macro `COMMA` instead of an actual comma. However, the use of that macro should be avoided if possible, e. g. by defining constructors with comma-separated initializer lists outside of the `STREAMABLE`'s body.

comma-separated-declarations: Declarations of the streamable member variables[3] in two possible forms:

```
(<type>) <var>
(<type>)(<init>) <var>
```

type: The type of the member variable that is declared.

var: The name of the member variable that is declared.

init: The initial value of the member variable, or if an object is declared, the parameter(s) passed to its constructor.

Please note that all these parts, including each declaration of a streamable member variable, are separated by commas, since they are parameters of a macro. Here is an example:

```
STREAMABLE(Example,
{
  ENUM(ABC,
  {,
    a,
    b,
    c,
  });

  Example()
  {
    std::memset(array, 0, sizeof(array));
  },

  (int) anInt,
  (float)(3.14f) pi,
  (int[4]) array,
  (Vector2f)(1.f, 2.f) aVector,
  (ABC) aLetter,
  (MotionRequest::Motion)(MotionRequest::stand) motionId,
});
```

In this example, all member variables except for `anInt` and `aLetter` would be initialized when an instance of the class is created.

Instances of `Example` can now be streamed like any basic data type. Since it is sometimes necessary to do some post-processing after an object was read from a stream, e. g. to recompute member variables that were not streamed, a method `onRead()` can be defined within the `STREAMABLE`, which is called after all streamed member variables were read.

[3] Currently, the macros support up to 119 entries.

3.4.4 Type Registry

The class `TypeRegistry` stores the specification of all streamable data types. It is automatically filled at the start of the program and is not changed afterwards. It has mainly three purposes:

1. Sending the type specifications to a remote PC. Thereby, the PC can exchange and visualize data, even if it is not running the same version of the B-Human software.

2. Providing the specification of all types for logging. This allows to replay log files even if some of the type specifications have changed since the file was recorded.

3. Mapping between the values and names of enumeration constants when streaming data as text.

The specification of a type is recorded through a static or global function. For instance, the `STREAMABLE` used as an example in Section 3.4.3 generated a function like this:

```
static void reg()
{
  REG_CLASS(Example);
  REG(int, anInt);
  REG(float, pi);
  REG(int[4], array);
  REG(Vector2f, aVector);
  REG(ABC, aLetter);
  REG(MotionRequest::Motion, motionId);
}
```

If the class that is registered has a streamable base class, it must be registered with another macro, because otherwise the type registry cannot know about this relation:

```
REG_CLASS_WITH_BASE(Example, SomeBaseClass);
```

To execute the function at the start of the program, it must be published:

```
PUBLISH(reg);
```

`PUBLISH` must be placed in a position of the code that is guaranteed to be linked. It publishes the registration function through its pure existence[4], not by being run through the normal flow of execution. In most cases, `PUBLISH` can just be placed in the registration function itself. However, for templates, it must be written into a function that is actually referenced from somewhere, otherwise it will never be executed, because it is not part of the executable. The same is true for code that is linked in from a static library.

The `TypeRegistry` is optimized for performance. It mostly stores addresses of names rather than the whole strings. Type names are kept in the mangled form the compiler generated. This means the data is only valid for the current execution of the program. To store the information into a file or to transmit it to another computer, an instance of the class `TypeInfo` must be filled first, which can then be streamed. `TypeInfo` can represent specification data that is independent from the current executable, e. g., it can contain the type specifications read from an older log file. All type names are de-mangled in a platform-independent way.

[4] Technically, a template is instantiated that adds itself to a linked list of all registration functions at the start of the program.

3.4.5 Streamable Classes

A class is made streamable by deriving it from the class `Streamable`, implementing the abstract method `serialize(In* in, Out* out)`, and adding a method that registers its type (cf. Section 3.4.4). For data types derived from `Streamable`, streaming operators are provided, meaning they may be used as any other data type with standard streaming operators implemented. The following two macros can be used to specify the data to stream in the method `serialize`. Both expect that the parameters of `serialize` are actually called `in` and `out`:

STREAM_BASE(<class>) streams the base class. If present, it must be the first `STREAM` statement in the `serialize` method.

STREAM(<member>) streams a member variable.

For instance, the `STREAMABLE` used as an example in Section 3.4.3 generated a function like this:

```
virtual void serialize(In* in, Out* out)
{
  STREAM(anInt);
  STREAM(pi);
  STREAM(array);
  STREAM(aVector);
  STREAM(aLetter);
  STREAM(motionId);
}
```

In addition to these two macros there is also a version to be used in streaming operators, i. e. in `operator<<` and `operator>>`. These macros do not expect that the stream has a pre-defined name. Instead, the stream is passed as their first parameter:

```
template<typename T> Out& operator<<(Out& out, const Range<T>& range)
{
  STREAM_EXT(out, range.min);
  STREAM_EXT(out, range.max);
  return out;
}
```

3.4.6 Configuration Maps

Configuration maps introduce the ability to handle serialized data from files in an arbitrary order. The sequence of entries in the file does not have to match the order of the member variables in the C++ data structure that is filled with them. In contrast to most streams presented in Section 3.4.1, configuration maps do not contain a serialization of a data structure, but rather a hierarchical representation.

Since configuration maps can be read from and be written to files, there is a special syntax for such files. A file consists of an arbitrary number of pairs of keys and values, separated by an equality sign, and completed by a semicolon. Values can be lists (encased by square brackets), structured values (encased by curly brackets), or plain values. If a plain value does not contain any whitespaces, periods, semicolons, commas, or equality signs, it can be written without quotation marks, otherwise it has to be encased in double quotes. Configuration map files have to follow this grammar:

```
map     ::= record
record  ::= field ';' { field ';' }
```

```
field   ::= literal '=' ( literal | '{' record '}' | array )
array   ::= '[' element { ',' element } [ ',' ] ']'
element ::= literal | '{' record '}'
literal ::= '"' { anychar1 } '"' | { anychar2 }
```

`anychar1` must escape doublequotes and the backslash with a backslash. `anychar2` cannot contain whitespace and other characters used by the grammar. However, when such a configuration map is read, each `literal` must be a valid literal for the datatype of the variable it is read into. As in C++, comments can be denoted either by `//` for a single line or by `/* ... */` for multiple lines. Here is an example:

```
// A record
defensiveGoaliePose = {
  rotation = 0;
  translation = {x = -4300; y = 0;};
};
/* An array of
 * three records
 */
kickoffTargets = [
  {x = 2100; y = 0;},
  {x = 1500; y = -1350;},
  {x = 1500; y = 1350;}
];
// Some individual values
outOfCenterCircleTolerance = 150.0;
ballCounterThreshold = 10;
```

Configuration maps can only be read or written through the streams derived from `OutMap` and `InMap`. Accordingly, they require an object of a streamable class to either parse the data in map format or to produce it. Here is an example of code that reads the file shown above:

```
STREAMABLE(KickOffInfo,
{,
  (Pose2f) defensiveGoaliePose,
  (std::vector<Vector2f>) kickoffTargets,
  (float) outOfCenterCircleTolerance,
  (int) ballCounterThreshold,
});

InMapFile stream("kickOffInfo.cfg");
KickOffInfo info;
if(stream.exists())
  stream >> info;
```

3.4.7 Enumerations

To support streaming, enumeration types should be defined using the macro `ENUM` defined in *Src/Tools/Streams/Enum.h* rather than using the C++ `enum` keyword directly. The macro's first parameter is the name of the enumeration type. The second and the last parameter are reserved for curly brackets and are ignored. All other parameters are the elements of the enumeration type defined. It is not allowed to assign specific integer values to the elements of the enumeration type, with one exception: It is allowed to initialize an element with the symbolic value of the element that has immediately been defined before (see example below). The macro automatically registers the enumeration type and its elements (except for elements used to initialize other elements) in the `TypeRegistry` (cf. Section 3.4.4).

The macro also automatically defines a constant `numOf<Typename>s`, which reflects the number of elements in the enumeration type. Since the name of that constant has an added "s" at the end, enumeration type names should be singular. If the enumeration type name already ends with an "s", it might be a good idea to define a constant outside the enumeration type that can be used instead, e. g. `static constexpr unsigned char numOfClasses = numOfClasss` for an enumeration type with the name `Class`.

The following example defines an enumeration type `Letter` with the "real" enumeration elements `a`, `b`, `c`, and `d`, a user-defined helper constant `numOfLettersBeforeC`, and an automatically defined helper constant `numOfLetters`. The numerical values of these elements are `a = 0, b = 1, c = 2, d = 3, numOfLettersBeforeC = 2, numOfLetters = 4`.

```
ENUM(Letter,
{,
  a,
  b,
  numOfLettersBeforeC,
  c = numOfLettersBeforeC,
  d,
});
```

3.4.7.1 Iterating over Enumerations

It is often necessary to enumerate all constants defined in an enumeration type. This can easily be done using the `FOREACH_ENUM` macro:

```
FOREACH_ENUM(Letter, letter, numOfLettersBeforeC)
{
  // do something with "letter", which is of type "Letter"
}
```

The third parameter is optional. If specified, it defines a different upper limit than the end of the enumeration.

3.4.7.2 Enumerations as Array Indices

In the B-Human code, enumerations are often used as indices for arrays, because this gives entries a name, but still allows to iterate over these entries. However, in configuration files and in the UI, it is hard to find specific entries in arrays, in particular if they have a larger number of elements. For instance, the arrays of all joint angles have 26 elements, making it hard to identify the angle of a specific joint. Therefore, a special macro (defined in *Src/Tools/Streams/EnumIndexedArray.h*) that is backed by a template class allows to define an array indexed by an enumeration type that solves this problem by streaming such an array as if it were a structure with a member variable for each of its elements named after the respective enumeration constant. Technically, such an array is derived from `std::array` and simply defines the method `serialize` (cf. Section 3.4.5). For example, an array of all joint angles could be defined by using the enumeration `Joints::Joint` as index:

```
ENUM_INDEXED_ARRAY(Angle, Joints::Joint) jointAngles;
```

`jointAngles` still behaves like an array, i. e. it is derived from `std::array<Angle, Joints::numOfJoints>`, but when, e. g., written to a file using `OutMapFile` (cf. Section 3.4.6), it would appear differently:

```
headYaw = 0deg;
```

```
headPitch = 0deg;
lShoulderPitch = 0deg;
...
```

3.4.8 Functions

Not all data in a representation can reasonably be computed before it is used. This would lead to a certain overhead, because the data has to be computed in advance without knowing whether it will actually be required in the current situation. In addition, there is a lot of data that cannot be computed in advance, because its computation depends on external values. For instance, a path planner must know the target to which it has to plan a path before it can be executed. However, in many situations a path planner is not needed at all, because the motion is generated reactively.

Functions in representations allow modules that require the representation to execute code of the module that provided that representation at any time and they allow to pass parameters to that implementation. These modules can be switched to other implementations as all other B-Human modules can, giving a greater flexibility when improving the functionality of the system.

These functions are based on `std::function` from the C++ runtime library and the assigned implementations are usually lambda expressions. However, to make them more compliant with the general B-Human architecture, they differ from the normal standard implementation in their behavior if no value was assigned to them (their *default* behavior). Instead of throwing an exception as `std::function` would do when called, they simply do nothing. If they have a return value, they return the default value of the type returned or `Zero()` in case of Eigen types.

However, there are some specialties to functions in representations:

- Functions cannot be streamed in any way, although they must be part of a class that is derived from the class `Streamable`. This also means that a module cannot call a function of a representation the provider of which is executed in another thread. However, representations containing functions can be streamed, but the functions cannot be part of the data that is streamed, i. e. they must be ignored during streaming.
- Representations containing functions are treated in a special way. They are always reset when their provider is switched. The reason is that the function object would otherwise still contain a reference to the module that originally provided the implementation, but the module does not exist anymore. In order to make this work with classes derived from classes containing functions, but not declaring functions themselves, the derived class has to contain the macro `BASE_HAS_FUNCTION` in its body.

For instance, the path planner provides its functionality through the representation `PathPlanner`:

```
STREAMABLE(PathPlanner,
{
  FUNCTION(MotionRequest(const Pose2f& target, const Pose2f& speed,
           bool excludePenaltyArea)) plan,
});
```

It assigns the function in its method `update`:

```
pathPlanner.plan = [this](const Pose2f& target, const Pose2f& speed,
                          bool excludePenaltyArea) -> MotionRequest
```

```
{ // ...
};
```

In the behavior control, the path planner is executed by this call:

```
theMotionRequest = thePathPlanner.plan(target, Pose2f(speed, speed, speed),
    avoidOwnPenaltyArea);
```

3.5 Communication

Three kinds of communication are implemented in the B-Human framework: *inter-thread communication*, *debug communication*, and *team communication*.

3.5.1 Inter-thread Communication

The representations sent back and forth between the threads (so-called shared representations) are automatically calculated by the `ModuleGraphCreator` based on the representations required by modules loaded in the respective thread but not provided by modules in the same thread. The directions in which they are sent are also automatically determined by the `ModuleGraphCreator`.

All inter-thread communication is triple-buffered. Thus, threads never block each other, because they never access the same memory blocks at the same time. In addition, a receiving thread always gets the most current version of a packet sent by another thread.

3.5.2 Message Queues

The *debug communication*, parts of the *team communication*, and *logging* are all based on the same technology: *message queues*. The class `MessageQueue` allows storing and transmitting a sequence of messages. Each message has a type (defined in *Src/Tools/MessageQueue/MessageIDs.h*) and a content. Each queue has a maximum size, which is defined in advance. On the robot, the amount of memory required is pre-allocated to avoid allocations during runtime. On the PC, the memory is allocated on demand, because several sets of robot threads can be instantiated at the same time, and the maximum size of the queues is rarely needed.

Since almost all data types are streamable (cf. Section 3.4), it is easy to store them in message queues. The class `MessageQueue` provides different write streams for different formats: messages that are stored through `out.bin` are formatted binary. The stream `out.text` formats data as text and `out.textRaw` as raw text. After all data of a message was streamed into a queue, the message must be finished with `out.finishMessage(MessageID)`, giving it a *message id*, i. e. a type.

```
MessageQueue m;
m.setSize(1000); // can be omitted on PC
m.out.text << "Hello world!";
m.out.finishMessage(idText);
```

To declare a new message type, an id for the message must be added to the enumeration type `MessageID` in *Src/Tools/MessageQueue/MessageIDs.h*. The enumeration type has two sections: the first for representations that should be recorded in log files, and the second for infrastructure messages.

Messages are read from a queue through a message handler that is passed to the queue's method `handleAllMessages(MessageHandler&)`. Such a handler must implement the method `handleMessage(InMessage&)` that is called for each message in the queue. It must be implemented in a way as the following example shows:

```
class MyClass : public MessageHandler
{
protected:
  bool handleMessage(InMessage& message)
  {
    switch(message.getMessageID())
    {
      default:
        return false;

      case idText:
      {
        std::string text;
        message.text >> text;
        return true;
      }
    }
  }
};
```

The handler has to return whether it handled the message or not. Messages are read from a `MessageQueue` via streams. Thereto, `message.bin` provides a binary stream, `message.text` a text stream, and `message.config` a text stream that skips comments.

3.5.3 Debug Communication

For debugging purposes, there is a communication infrastructure between the threads and the PC. This is accomplished by *message queues*. Each thread has two of them: `theDebugSender` and `theDebugReceiver`. The macro `OUTPUT(<id>, <format>, <sequence>)` defined in *Src/Tools/Debugging/Debugging.h* simplifies writing data to the outgoing debug message queue. *id* is a valid message id, *format* is `text`, `bin`, or `textRaw`, and *sequence* is a streamable expression, i. e. an expression that contains streamable objects, which – if more than one – are separated by the streaming operator `<<`.

```
OUTPUT(idText, text, "Could not load file " << filename << " from " << path);
OUTPUT(idCameraImage, bin, CameraImage());
```

For receiving debugging information from the PC, each thread also has a message handler, i. e. it implements the method `handleMessage` to distribute the data received.

The thread *Debug* manages the communication of the robot control program with the tools on the PC. For each of the other threads, it has a sender and a receiver for their debug message queues (cf. Fig. 3.1). Messages that arrive via Wi-Fi or Ethernet from the PC are stored in `debugIn`. The method `Debug::handleMessage(InMessage&)` distributes all messages in `debugIn` to the other threads. The messages received from *Upper*, *Lower*, *Cognition* and *Motion* are stored in `debugOut`. When a Wi-Fi or Ethernet connection is established, they are sent to the PC via TCP/IP.

The debug communication and the thread *Debug* are not available on the actual NAO when the software deployed was compiled in the configuration *Release*.

3.5.4 Team Communication

The purpose of the team communication is to send messages to the other robots in the team. These messages are always broadcasted via UDP, so all teammates can receive them. Sending and receiving team messages is done in the thread *Cognition* using the representations `TeamData` for handling received messages and `BHumanMessageOutputGenerator` for generating messages to send. These representations are both generated and filled with their functionality by the module `TeamMessageHandler`.

The format of team messages is given by the `SPLStandardMessage`, which consists of a standardized and a non-standard part. The custom data part is subdivided into three sections: The first one is the `MixedTeamHeader` that enables sharing some data with our mixed team partner (cf. Section 9.3). It is followed by the `BHumanStandardMessage`, which contains many representations in a compressed version, e. g. timestamps are quantized and made relative to the timestamp when the message is sent to use fewer bits and some floating point numbers are quantized, too. Finally, the `BHumanArbitraryMessage` is a message queue that is used especially for variable-length representations that may not fit into the message at all.

In order to be transmitted by the `TeamMessageHandler`, a representation has to extend the struct `BHumanMessageParticle` and provide implementations for reading and writing its data to or from a message. If a representation is only supposed to be included in the message queue and does not fill in any standardized fields of the `SPLStandardMessage`, it can also just extend the struct `PureBHumanArbitraryMessageParticle`. The representation has to be added in three locations: the `handleMessage` method of the `Teammate` struct and the `generateMessage` and `parseMessageIntoBMate` methods of the `TeamMessageHandler`.

According to the rules, team communication packets are only broadcasted once every second. The `BHumanMessageOutputGenerator` contains a flag set by the `TeamMessageHandler` that states whether a team communication packet will be sent out in the current frame or not. The `TeamMessageHandler` also implements the network time protocol (*NTP*) and translates time stamps contained in the messages it receives into the local time of the robot.

3.6 Debugging Support

Debugging mechanisms are an integral part of the B-Human framework. They are all based on the debug message queues already described in Section 3.5.3. These mechanisms are available in all project configurations except for *Release* on the actual NAO.

3.6.1 Debug Requests

Debug requests are used to enable and disable parts of the source code. They can be seen as runtime switches for debugging.

The debug requests can be used to trigger certain debug messages to be sent as well as to switch on certain parts of algorithms. They can be sent using the SimRobot software when connected to a NAO (cf. command *dr* in Section 10.1.6.3). The following macros ease the use of the mechanism as well as hide its implementation details:

DEBUG_RESPONSE(<id>) executes the following statement or block if the debug request with the name `id` is enabled.

DEBUG_RESPONSE_ONCE(<id>) executes the following statement or block *once* when the

debug request with the name `id` is enabled.

DEBUG_RESPONSE_NOT(<id>) executes the following statement or block if the debug request with the name `id` is *not* enabled.

These macros can be used anywhere in the source code, allowing for easy debugging. For example:

```
DEBUG_RESPONSE("test") test();
```

This statement calls the method `test()` if the debug request with the identifier "test" is enabled. Debug requests are commonly used to send messages on request as the following example shows:

```
DEBUG_RESPONSE("sayHello") OUTPUT(idText, text, "Hello");
```

This statement sends the text "Hello" if the debug request with the name `"sayHello"` is activated. Please note that only those debug requests are usable that are in the current path of execution. This means that only debug requests in those modules can be activated that are currently executed. To determine which debug requests are currently available, a method called *polling* is employed. It asks each debug response to report the name of the debug request that would activate it. This information is collected and sent to the PC (cf. command *poll* in Section 10.1.6.3).

3.6.2 Debug Images

Debug images are used for low level visualization of image processing debug data. They can either be displayed as background image of an image view (cf. Section 10.1.4.1) or in a color space view (cf. Section 10.1.4.1). Each debug image consists of pixels in one of the formats *RGB*, *BGRA*, *YUYV*, *YUV*, *Grayscale*, *Colored*, *Hue*, *Binary*, and *Edge2*, which are defined in *Src/Tools/ImageProcessing/PixelTypes.h*. In the *RGB*, *RGBA*, *BGRA*, *YUYV*, and *YUV* formats, pixels are made up of four unsigned byte values, each representing one of the color channels of the format. The *YUYV* format is special as one word of the debug image describes two image pixels by specifying two luminance values, but only one value per U and V channel. Thus, it resembles the *YUV422* format of the images supplied by the NAO's cameras. Debug images in the *Grayscale* format only contain a single channel of unsigned byte values describing the luminance of the associated pixel. The *Colored*, *Hue*, and *Binary* formats consist of only one channel, too. Values in the *Colored* channel are entries of the `FieldColors::Color` enumeration, which contains identifiers for the color classes used for image processing (cf. Section 4.1.5). The *Edge2* format has one channel each for the intensity in x and y direction.

Debug images are supposed to be declared as instances of the template class `Image`, instantiated with one of the pixel formats named above, or the `CameraImage` class, which is only used for images provided by the NAO's cameras. The following macros are used to transfer debug images to a connected PC.

SEND_DEBUG_IMAGE(<id>, <image>, [<method>]) sends the debug image to the PC. The identifier given to this macro is the name by which the image can be requested. The optional parameter is a pixel type that allows to apply a special drawing method for the image.

COMPLEX_IMAGE(<id>) only executes the following statement if the creation of a certain debug image is requested. For debug images that require complex instructions to paint, it

can significantly improve the performance to encapsulate the drawing instructions in this macro (and maybe additionally in a separate method).

These macros can be used anywhere in the source code, allowing for easy creation of debug images. For example:

```
class Test
{
private:
  Image<GrayscaledPixel> testImage;

public:

  void doSomething()
  {
    // [...]
    COMPLEX_IMAGE("test") draw();
    // [...]
  }

  void draw()
  {
    testImage.setResolution(640, 480);
    memset(testImage[0], 0x7F, testImage.width * testImage.height);
    SEND_DEBUG_IMAGE("test", testImage);
  }
};
```

The example calls the `draw()` method if the `"test"` image was requested, which then initializes a grayscale debug image, paints it gray, and sends it to the PC.

3.6.3 Debug Drawings

Debug drawings provide a virtual 2-D drawing canvas and a number of drawing primitives, as well as mechanisms for requesting, sending, and drawing these primitives to the screen of the PC. In contrast to debug images, which are raster-based, debug drawings are vector-based, i. e., they store drawing instructions instead of a rasterized image. Each drawing has an identifier and an associated type that enables the application on the PC to render the drawing to the right kind of drawing canvas. In the B-Human system, two standard drawing canvases are provided, called “drawingOnImage” and “drawingOnField”. This refers to the two standard applications of debug drawings, namely drawing in the system of coordinates of an image and drawing in the system of coordinates of the field. Hence, all debug drawings of type “drawingOnImage” can be displayed in an image view (cf. Section 10.1.4.1) and all drawings of type “drawingOnField” can be rendered into a field view (cf. Section 10.1.4.1).

The creation of debug drawings is encapsulated in a number of macros in *Src/Tools/Debugging/DebugDrawings.h*. Most of the drawing macros have parameters such as pen style, fill style, or color. Available pen styles (`solidPen`, `dashedPen`, `dottedPen`, and `noPen`) and fill styles (`solidBrush` and `noBrush`) are part of the namespace `Drawings`. Colors can be specified as `ColorRGBA`. The class also contains a number of predefined colors such as `ColorRGBA::red`. A few examples for drawing macros are:

DECLARE_DEBUG_DRAWING(<**id**>**,** <**type**>**)** declares a debug drawing with the specified *id* and *type*.

COMPLEX_DRAWING(<**id**>**)** only executes the following statement or block if the creation of a certain debug drawing is requested. This can significantly improve the performance

when a debug drawing is not requested, because for each drawing instruction it has to be tested whether it is currently required or not. By encapsulating them in this macro (and maybe in addition in a separate method), only a single test is required. However, the macro `DECLARE_DEBUG_DRAWING` must be placed outside of `COMPLEX_DRAWING`.

DEBUG_DRAWING(<id>, <type>) is a combination of `DECLARE_DEBUG_DRAWING` and `COMPLEX_DRAWING`. It declares a debug drawing with the specified *id* and *type*. It also executes the following statement or block if the debug drawing is requested.

CIRCLE(<id>, <x>, <y>, <radius>, <penWidth>, <penStyle>, <penColor>, <fillStyle>, <fillColor>) draws a circle with the specified radius, pen width, pen style, pen color, fill style, and fill color at the coordinates (x, y) on the virtual drawing canvas.

LINE(<id>, <x1>, <y1>, <x2>, <y2>, <penWidth>, <penStyle>, <penColor>) draws a line with the pen color, width, and style from the point $(x1, y1)$ to the point $(x2, y2)$ on the virtual drawing canvas.

DOT(<id>, <x>, <y>, <penColor>, <fillColor>) draws a dot with the pen color and fill color at the coordinates (x, y) on the virtual drawing canvas. There also exist two macros `MID_DOT` and `LARGE_DOT` with the same parameters that draw dots of larger size.

DRAWTEXT(<id>, <x>, <y>, <fontSize>, <color>, <text>) writes a text with a font size in a color to a virtual drawing canvas. The left end of the baseline of the text will be at coordinates (x, y).

TIP(<id>, <x>, <y>, <radius>, <text>) adds a tool tip to the drawing that will pop up when the mouse cursor is closer to the coordinates (x, y) than the given radius.

ORIGIN(<id>, <x>, <y>, <angle>) changes the system of coordinates. The new origin will be at (x, y) and the system of coordinates will be rotated by $angle$ (given in radians). All further drawing instructions, even in other debug drawings that are rendered afterwards in the same view, will be relative to the new system of coordinates, until the next origin is set. The origin itself is always absolute, i. e. a new origin is not relative to the previous one.

THREAD(<id>, <thread>) defines as part of which thread a drawing should be managed. The default is the thread that contains the drawing. However, if a drawing in a thread belongs to data that was received from another thread, it should be linked to that thread by using this macro. This is particularly true for image drawings that always have to be linked to one of the two image processing threads, i. e. *Upper* and *Lower*, but it can also be necessary for field drawings.

These macros can be used wherever statements are allowed in the source code. For example:

```
DECLARE_DEBUG_DRAWING("test", "drawingOnField");
CIRCLE("test", 0, 0, 1000, 10, Drawings::solidPen, ColorRGBA::blue,
       Drawings::solidBrush, ColorRGBA(0, 0, 255, 128));
```

This example initializes a drawing called `test` of type `drawingOnField` that draws a blue circle with a solid border and a semi-transparent inner area.

3.6.4 3-D Debug Drawings

In addition to the aforementioned two-dimensional debug drawings, there is a second set of macros in *Src/Tools/Debugging/DebugDrawings3D.h*, which provides the ability to create three-dimensional debug drawings.

3-D debug drawings can be declared with the macro `DECLARE_DEBUG_DRAWING3D(<id>, <type>)`. The `id` can then be used to add three dimensional shapes to this drawing. `type` defines the coordinate system in which the drawing is displayed. It can be set to "field", "robot", "camera", or any named part of the robot model in the scene description. Note that drawings directly attached to hinges will be drawn relative to the base of the hinge, not relative to the moving part. Drawings of the type "field" are drawn relative to the center of the field, whereas drawings of the type "robot" are drawn relative to the origin of the robot, i. e. the middle between the two hip joints. It is often used as reference frame for 3-D debug drawings in the current code. The type "camera" is an alias for the camera of the perception thread from which the drawing is generated.

The parameters of macros adding shapes to a 3-D debug drawing start with the `id` of the drawing this shape will be added to, followed, e. g., by the coordinates defining a set of reference points (such as corners of a rectangle), and finally the drawing color. Some shapes also have other parameters such as the thickness of a line. Here are a few examples for shapes that can be used in 3-D debug drawings:

LINE3D(<id>, <fromX>, <fromY>, <fromZ>, <toX>, <toY>, <toZ>, <size>, <color>) draws a line between the given points.

QUAD3D(<id>, <corner1>, <corner2>, <corner3>, <corner4>, <color>) draws a quadrangle with its four corner points given as 3-D vectors and specified color.

SPHERE3D(<id>, <x>, <y>, <z>, <radius>, <color>) draws a sphere with specified radius and color at the coordinates (x, y, z).

COORDINATES3D(<id>, <length>, <width>) draws the axes of the coordinate system with specified length and width into positive direction.

COMPLEX_DRAWING3D(<id>) only executes the following statement or block if the creation of the debug drawing is requested (similar to `COMPLEX_DRAWING(<id>)` for 2-D drawings).

DEBUG_DRAWING3D(<id>, <type>) is a combination of `DECLARE_DEBUG_DRAWING3D` and `COMPLEX_DRAWING3D`. It declares a debug drawing with the specified *id* and *type*. It also executes the following statement or block if the debug drawing is requested.

The header file furthermore defines some macros to scale, rotate, and translate an entire 3-D debug drawing:

SCALE3D(<id>, <x>, <y>, <z>) scales all drawing elements by given factors for x, y, and z axis.

ROTATE3D(<id>, <x>, <y>, <z>) rotates the drawing counterclockwise around the three axes by given radians.

TRANSLATE3D(<id>, <x>, <y>, <z>) translates the drawing according to the given coordinates.

An example for 3-D debug drawings (analogous to the example for regular 2-D debug drawings):

```
DECLARE_DEBUG_DRAWING3D("test3D", "field");
SPHERE3D("test3D", 0, 0, 250, 75, ColorRGBA::blue);
```

This example initializes a 3-D debug drawing called `test3D`, which draws a blue sphere. Because the drawing is of type `field` and the origin of the field coordinate system is located in the center of the field, the sphere's center will appear 250 mm above the center point.

In order to add a drawing to the scene, a debug request (cf. Section 10.1.6.3) for a 3-D debug drawing with the ID of the desired drawing prefixed by "debugDrawing3d:" must be sent. For example, to see the drawing generated by the code above, the command would be *dr debugDrawing3d:test3D*. The rendering of debug drawings can be configured for individual scene views by right-clicking on the view and selecting the desired shading and occlusion mode in the "Drawings Rendering" submenu.

3.6.5 Plots

The macro `PLOT(<id>, <number>)` allows plotting data over time. The plot view (cf. Section 10.1.4.5) will keep a history of predefined size of the values sent by the macro `PLOT` and plot them in different colors. Hence, the previous development of certain values can be observed as a time series. Each plot has an identifier that is used to distinguish the different plots from each other. A plot view can be created with the console commands *vp* and *vpd* (cf. Section 10.1.6.3).

For example, the following code statement plots the measurements of the gyro for the pitch axis in degrees. It should be placed in a part of the code that is executed regularly, e. g. inside the update method of a module.

```
PLOT("gyroY", theInertialSensorData.gyro.y().toDegrees());
```

The macro `DECLARE_PLOT(<id>)` allows using the `PLOT(<id>, <number>)` macro within a part of code that is not regularly executed as long as the `DECLARE_PLOT(<id>)` macro is executed regularly.

3.6.6 Modify

The macro `MODIFY(<id>, <object>)` allows inspecting and modifying data on the actual robot during runtime. Every streamable data type (cf. Section 3.4.5) can be manipulated and read, because its inner structure was gathered in the `TypeRegistry` (cf. Section 3.4.4). This allows generic manipulation of runtime data using the console commands *get* and *set* (cf. Section 10.1.6.3). The first parameter of `MODIFY` specifies the identifier that is used to refer to the object from the PC, the second parameter is the object to be manipulated itself. When an object is modified using the console command *set*, it will be overridden each time the `MODIFY` macro is executed.

```
int i = 3;
MODIFY("i", i);
MotionRequest m;
MODIFY("representation:MotionRequest", m);
```

The macro `PROVIDES` of the module framework (cf. Section 3.3) includes the `MODIFY` macro for the representation provided. For instance, if a representation `Foo` is provided by `PROVIDES(Foo)`, it is modifiable under the name `representation:Foo`. If a representation provided should not be modifiable, e. g., because its serialization does not register all member variables, it must be provided using `PROVIDES_WITHOUT_MODIFY`.

3.6.7 Stopwatches

Stopwatches allow the measurement of the execution time of parts of the code. The macro `STOPWATCH(<id>)` (declared in *Src/Tools/Debugging/Stopwatch.h*) measures the runtime of the statement or block that follows. id is a string used to identify the time measurement. To activate the time measurement of all stopwatches, the debug request `dr timing` has to be sent. The measured times can be seen in the timing view (cf. Section 10.1.4.5). By default, a stopwatch is already defined for each representation that is currently provided, and for the overall execution of all modules in each thread.

An example to measure the runtime of a method called `myCode`:

```
STOPWATCH("myCode") myCode();
```

Using the `STOPWATCH` macro also adds a plot with the stopwatch identifier prefixed by "stopwatch:".

3.7 Logging

The B-Human framework offers sophisticated logging functionality that can be used to log the values of selected representations while the robot is playing. There are two different ways of logging data: online logging and remote logging.

3.7.1 Online Logging

The online logging feature can be used to log data directly on the robot during regular games. It is implemented as part of the robot control program and is designed to log representations in real-time. Log files are written into the directory */home/nao/logs* or – if available – into */media/usb/logs*, i. e. a USB flash drive on which the directory called *logs* will be created automatically. To use a USB flash drive for logging, it must already have been plugged in when the B-Human software is started.

Online logging starts as soon as the robot enters the *ready* state and stops upon entering the *finished* state. The log files are compressed on the fly using Google's Snappy compression [7]. The name of the log file consists of the names of the head and body of the robot as well as its player number, the scenario, and the location. If connected to the GameController, the name of the opponent team and the half are also added to the log file name. Otherwise, "Testing" is used instead. If a log file with the given name already exists, a number is added that is incremented for each duplicate.

To retain the real-time properties of the threads, the heavy lifting, i. e. compression and writing of the file, is done in a separate thread without real-time scheduling. This thread uses the remaining processor time that is not used by one of the real-time parts of the system. Communication with this thread is realized using a very large ring buffer (usually around 2.4 GB). Each element of the ring buffer represents one frame of data. If a buffer is full, the current frame cannot be logged.

Due to the limited buffer size, the logging of very large representations such as the `CameraImage` is not possible. It would cause a buffer overflow within less than two minutes rendering the resulting log file unusable. However, without images a log file is nearly useless, therefore loggable *thumbnail images* (cf. Section 3.7.7) or *JPEG images* are generated and used instead.

In addition to the logged representations, online log files also contain timing data, which can be seen using the TimingView (cf. Section 10.1.4.5).

3.7.2 Configuring the Online Logger

The online logger can be configured by changing the following values in the file *Config/Scenarios/<scenario>/logger.cfg*:

enabled: The logger is enabled if this is true. Note that it is not possible to enable the logger inside the simulator.

path: Log files will be stored in this path on the NAO.

numOfBuffers: The number of buffers allocated.

sizeOfBuffer: The size of each buffer in bytes. Note that numOfBuffers$\times$sizeOfBuffer is always allocated if the logger is enabled.

writePriority: Priority of the logging thread. Priorities greater than zero use the real-time scheduler, zero uses the normal scheduler, and negative values (-1 and -2) use idle priorities.

minFreeDriveSpace: The minimum amount of disk space that should always be left on the target device in MB. If the free space falls below this value the logger will cease operation.

representationsPerThread: List of all representations that should be logged per thread.

3.7.3 Remote Logging

Online logging provides the maximum possible amount of debugging information that can be collected without breaking the real-time capability. However in some situations one is interested in high precision data (i. e. full resolution images) and does not care about real-time. The remote logging feature provides this kind of log files. It utilizes the debugging connection (cf. Section 3.5.3) to a remote computer and logs all requested representations on that computer. This way the heavy lifting is outsourced to a computer with much more processing power and a bigger hard drive. However sending large representations over the network severely impacts the NAO's performance and can result in a loss of the real-time capability.

To reduce the network load, it is usually a good idea to limit the number of representations to the ones that are really needed for the task at hand. The follow listing shows the commands that need to be entered into SimRobot to record a minimal vision log. Alternatively, *call Includes/VisionLog* can be executed.

```
dr off
dr representation:BodyContour
dr representation:CameraImage
dr representation:CameraInfo
dr representation:CameraMatrix
dr representation:ImageCoordinateSystem
dr representation:OdometryData
log start
log stop
log save <filename>
```

3.7.4 Log File Format

In general, log files consist of serialized message queues (cf. Section 3.5.2). Each log file consists of up to three chunks. Each chunk is prefixed by a single byte defining its type. The enum `LogFileFormat` in *Src/Tools/Logging/LoggingTools.h* defines these chunk identifiers in the namespace `LoggingTools` that have to appear in the given sequence in the log file if they appear at all:

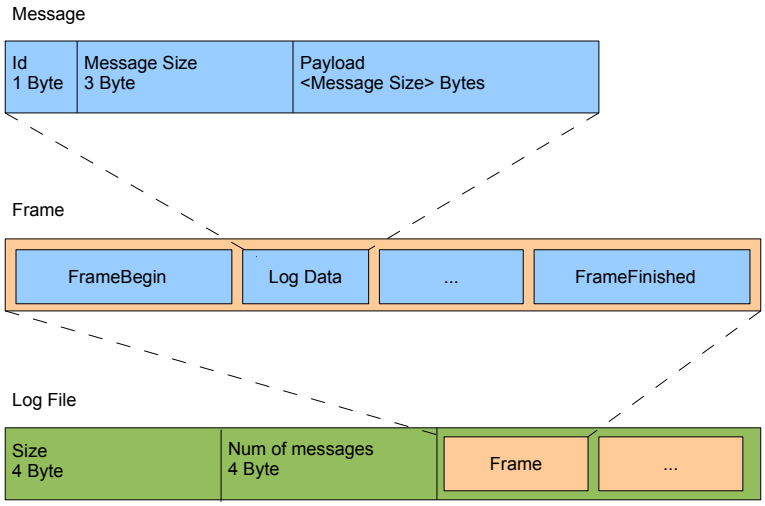


Figure 3.2: The format of a chunk of the type `logFileUncompressed`. Updated from [31].

logFileMessageIDs: This chunk contains a string representation of the `MessageIDs` (cf. Section 3.5.2) stored in this log file. It is used to convert the `MessageIDs` from the log file to the ones defined in the version of *SimRobot* (cf. Section 10.1) that is replaying the log file. Thereby, log files still remain usable after the enumeration `MessageID` was changed.

logFileTypeInfo: This chunk contains the specification of all datatypes used in the log file. It is used to convert the logged data to the specifications that are defined in the version of *SimRobot* that is replaying the log file. If the specification changed, messages will appear in SimRobot's console about the representations that are converted. Please note that a conversion is only possible for representations the specification of which is fully registered. This is not the case for representations that use `read` and `write` methods to serialize their data, e. g. `CameraImage`, `JPEGImage`, and `Thumbnail`. Therefore, such representations cannot be converted and will likely crash SimRobot when trying to replay log files containing them after they were changed.

logFileUncompressed: Uncompressed log data is stored in a single `MessageQueue`. Its serialized form starts with an eight byte header containing two values. These are the size used by the log followed by the number of messages in the queue. Those values can also be set to -1 indicating that the size and number of messages is unknown. In this case the amount will be counted when reading the log file. The header is followed by several frames. A frame consists of multiple log messages enclosed by an `idFrameBegin` and `idFrameFinished` message. Every log message consists of its id, its size and its payload (cf. Fig. 3.2). This kind of chunk is created by remote logging.

logFileCompressed: This chunk contains a sequence of compressed `MessageQueues`. Each queue contains a single frame worth of log data and is compressed using Google's Snappy [7] compression (cf. Fig. 3.3). Each compressed `MessageQueue` is prefixed by its compressed size. This kind of chunk is created by online logging.

The first two chunks are optional. Each log file must contain one of the latter two chunks, which is also the last chunk in the file.

Figure 3.3: A compressed log file consists of several regular log files and their compressed sizes.

3.7.5 Replaying Log Files

Log files are replayed using the simulator. A special module, the `LogDataProvider`, automatically provides all representations that are present in the log file. All other representations are provided by their usual providers, which are simulated. This way log file data can be used to test and evolve existing modules without access to an actual robot. If the name of a log file follows the naming convention used by the online logger, the head name, body name, player number, scenario, and location will be set to the ones given in the log file name before the log file is replayed. Thereby, modules will read the matching versions of their configuration files.

The SimRobot scene *ReplayRobot.ros2* can be used to load the log file. In addition to loading the log file this scene also provides several keyboard shortcuts for navigation inside the log file. However, the most convenient way to control log file playback is the *log player view* (cf. Section 10.1.4.5). If you want to replay several log files at the same time, simply create a file *Config/Scenes/Includes/replay.con* and add several *sl* statements (cf. Section 10.1.6.1) to it. If you want to replay a folder or an entire folder structure use the *sml* command (cf. Section 10.1.6.1). The data of each log file will be fed into a separate instance of the B-Human code.

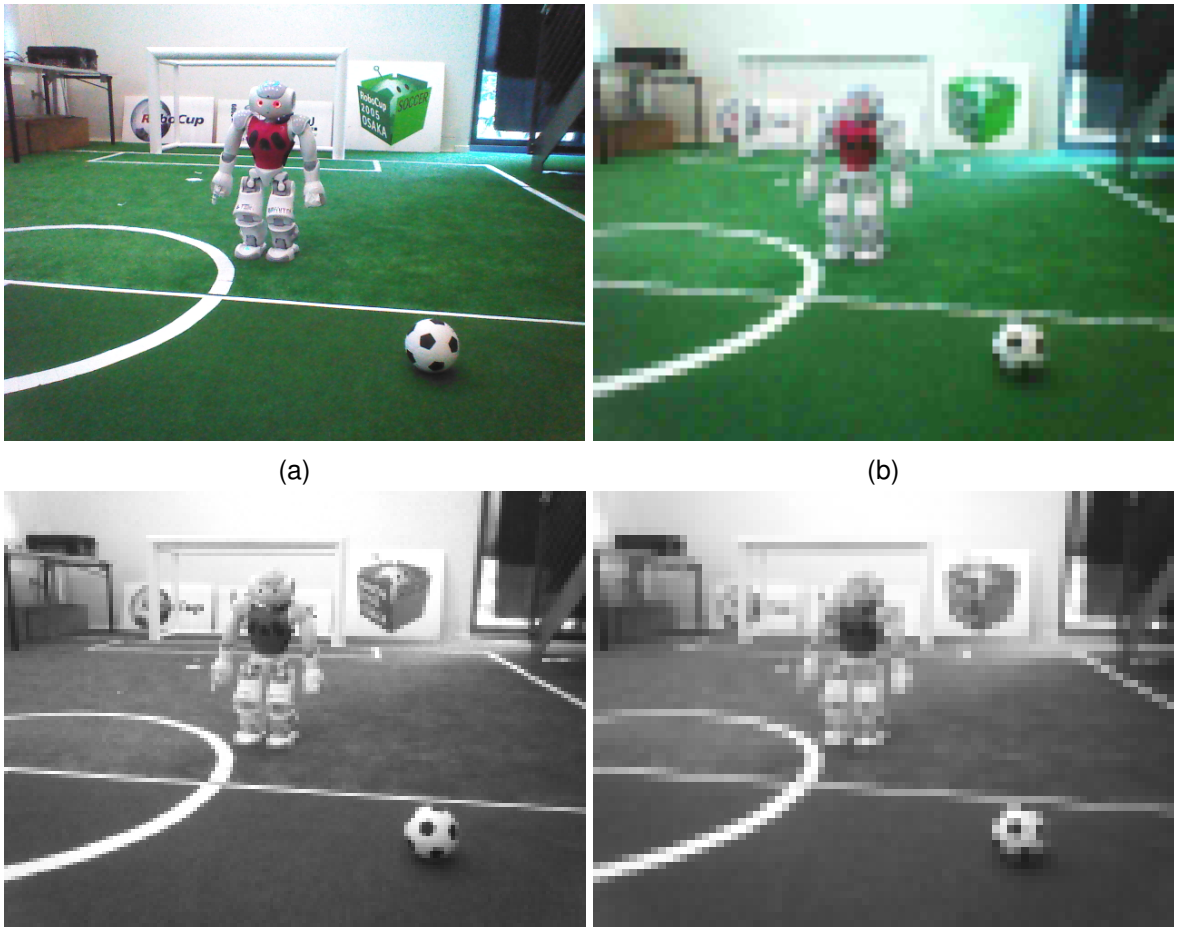
3.7.6 Annotations

To further enhance the usage of log files, there is the possibility for our modules to annotate individual frames of a log file with important information. This is, for example, information about a change of game state, the execution of a kick, or other information that may help us to debug our code. Thereby, when replaying the log file, we may consult a list of those annotations to see whether specific events actually did happen during the game. In addition, if an annotation was recorded, we are able to directly jump to the corresponding frame of the log file to review the emergence and effects of the event without having to search through the whole log file.

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

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

It is advised to be careful to not send an annotation in each frame because this will clutter the log file. When using annotations inside the behavior, the skill `Annotation` should be used to make sure annotations are not sent multiple times.



(a) (b)

(c) (d)

Figure 3.4: (a) Image from the upper camera of the NAO and (b) the corresponding colored thumbnail with a downscale factor of 2 as well as the corresponding grayscale thumbnails with downscale factors (c) 2 and (d) 3.

The annotations recorded can be displayed while replaying the log file in the annotation view (cf. Section 10.1.4.5).

3.7.7 Thumbnail Images

Logging raw YUV422 images as the cameras capture them seems impossible with the current hardware. One second would consume about 23 MB. This amount of data could not be stored fast enough on the internal drive or a USB flash drive and would fill it up within a few minutes. Thus, images have to be compressed. This compression must happen within a very small time frame, since the robot has to do a lot of other tasks in order to play football. Despite the compression, the resulting images must still contain enough information to be useful.

For compressing the images in a way that fulfills the criteria mentioned above, two separate methods have been implemented. One of these generates grayscaled images by taking the luminance channel (Y) of the camera image and averaging blocks of adjacent pixels. The other method compresses the YUV camera image by first eliminating every second row of pixels and then averaging every channel of adjacent blocks of pixels. Afterwards the size of each pixel is reduced to two bytes by eliminating one of the two Y channels in each pixel and using only six

bits for the remaining luminance channel and five bits for each of the two chroma channels. In both of the two possible methods, averaging adjacent pixels is done using SSE instructions to speed up the computation.

The total compression rate as well as the usefulness of the resulting images depends on the `downscales` parameter of the `ThumbnailProvider`, which determines the size of the blocks of pixels being averaged as $2^{\texttt{downscales}} \times 2^{\texttt{downscales}}$ pixels. For a `downscales` parameter of 2, the resulting grayscale thumbnail images are 32 times smaller and the resulting colored images are 128 times smaller than the original camera image. Although a lot of detail is lost in the thumbnail images, it is still possible to see and assess the situations the robot was in (cf. Fig. 3.4).

Currently, the thumbnail images are also used by the module `PlayerDeeptector` as input for player detection.

Chapter 4

Perception

The perception modules run in the context of the threads *Lower* resp. *Upper*, depending on the used camera. They detect features in the image that was just taken by the camera and can be separated into four categories. The modules of the perception infrastructure provide representations that deal with the perspective of the image taken, provide the image in different formats, and provide representations that limit the area interesting for further image processing steps. Based on these representations, modules detect features useful for self-localization, the ball, and other robots. All information provided by the perception modules are relative to the robot's position.

4.1 Perception Infrastructure

4.1.1 Obtaining Camera Images

The NAO robot is equipped with two video cameras that are mounted in the head of the robot. The first camera is installed in the middle forehead and the second one approximately 4 cm below. The lower camera is tilted by 39.7° with respect to the upper camera and both have a vertical opening angle of 43.7°. Because of that, the overlapping parts of the images are too small for stereo vision. It is also impossible to get images from both cameras at the exact same time, as they are not synchronized on a hardware level. This is why we analyze them separately and do not stitch them together. In order to be able to analyze the pictures of both cameras in real-time without losing any images, the threads *Lower* and *Upper* run each at 30 Hz.

During normal play, the lower camera sees only a very small portion of the field directly in front

of the robot's feet. Therefore, objects in the lower image are close to the robot and rather big. We take advantage of this fact and run the lower camera with half the resolution of the upper camera, thereby saving a lot of computation time. Hence the upper camera provides 640×480 pixels while the lower camera only provides 320×240 pixels.

A major problem that can occur with V6's cameras are corrupted images. From experience, this mainly happens at the resolution we use for the lower camera and appears as fragmented camera images in which rows are not updated and occur repeatedly across multiple images. To counter this problem, a random image row is periodically selected for which the hash value is calculated in the following images. We monitor whether hash values occur repeatedly and reinitialize the cameras when this is the case.

Both cameras deliver their images in the *YUV422* format. As the perception of features in the images relies either on color classes (e. g. for region building) or luminance values of the image pixels (e. g. for computing edges in the image), the *YUV422* images are converted to the "extracted and color-classified" `ECImage`. The `ECImage` consists of multiple images: the grayscaled image obtained from the Y channel of the camera image, the saturated resp. hued image containing saturation and hue values for each image pixel, and a so-called "colored" image mapping each image pixel to a color class.

The modules processing an image need to know from which camera it comes. For this purpose the representation `CameraInfo` is used, which contains this information as well as the resolution of the current image and the intrinsic camera parameters.

4.1.2 Definition of Coordinate Systems

The global coordinate system (cf. Fig. 4.1) is described by its origin lying at the center of the field, the x-axis pointing toward the opponent's goal, the y-axis pointing to the left, and the z-axis pointing upward. Rotations are specified counter-clockwise with the x-axis pointing toward $0°$ and the y-axis pointing toward $90°$.

In the robot-relative system of coordinates (cf. Fig. 4.2), the axes are defined as follows: the x-axis points forward, the y-axis points to the left, and the z-axis points upward.

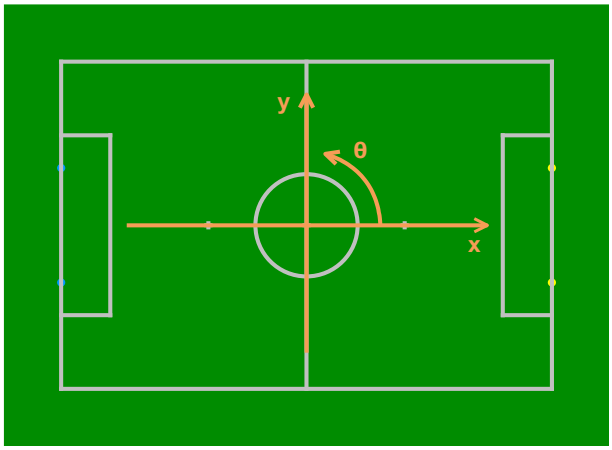


Figure 4.1: Visualization of the global coordinate system (opponent goal is marked in yellow)

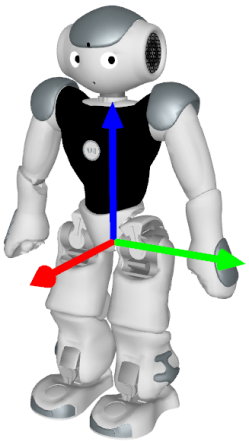


Figure 4.2: Visualization of the robot-relative coordinate system. The x-axis is red, the y-axis is green and the z-axis is blue.

4.1.2.1 Camera Matrix and Camera Calibration

The `CameraMatrix` is a representation containing the transformation matrix of the camera used in the respective thread (cf. Section 4.1.1) that is provided by the `CameraMatrixProvider`. It is used for projecting objects onto the field as well as for the creation of the `ImageCoordinateSystem` (cf. Section 4.1.2.2). It is computed based on the `TorsoMatrix` that represents the orientation and position of a specific point within the robot's torso relative to the ground (cf. Section 7.4). Using the `RobotDimensions` and the current joint angles, the transformation of the camera matrix relative to the torso matrix is computed as the `RobotCameraMatrix`. The latter is used to compute the `BodyContour` (cf. Section 4.1.3). In addition to the fixed parameters from the `RobotDimensions`, some robot-specific parameters from the `CameraCalibration` are integrated, which are necessary, because the camera cannot be mounted perfectly plain and the torso is not always perfectly vertical. A small variation in the camera's orientation can lead to significant errors when projecting farther objects onto the field.

The process of manually calibrating the robot-specific correction parameters for the cameras is a very time-consuming task since the parameter space is quite large (8 parameters for calibrating both cameras) and it is not always obvious which parameters have to be adapted if a camera is miscalibrated. In particular during competitions, the robots' cameras require recalibration very often, e. g. after a robot returned from repair.

In order to overcome this problem, an automatic camera calibration performed either by the `AutomaticCameraCalibrator` or `AutomaticCameraCalibratorPA` is used. The detailed calibration procedures are described in Section 2.7.3. Both collect samples that are used in the Gauss-Newton algorithm[1] to optimize the camera parameters. The modules therefore follow the same approach, however they use different sample types.

AutomaticCameraCalibratorPA

The `AutomaticCameraCalibratorPA` performs the optimization of the camera parameters based on samples composed of the penalty mark and lines of the penalty area. The error function used in the Gauss-Newton algorithm projects the components of the samples to field coordinates and compares the distances resp. angles between those with the known optimal ones. The different sample types used are the angle and distance between the ground line and the front penalty area line, the angle between either of them and the short connecting line, and the distance from the penalty mark to the ground line resp. the front penalty area line.

The calibration is divided into two steps, whereby these differ only in the position of the robot on the field and the sample types to be collected. Specifically, the samples using the penalty mark are only collected in the second step. In each step the head moves to predefined angles, in order to collect the required samples. The information to construct the samples are gathered from the `LinesPercept` provided by the `LinePerceptor` (cf. Section 4.3.1) and the `PenaltyMarkPercept` provided by the `PenaltyMarkPerceptor` (cf. Section 4.3.5).

The lines from the `LinesPercept` are not used directly because they do not describe the actual field lines accurately enough, i. e., they are not fitted. Instead, a hough transformation is performed to obtain a more accurate description of them. The start and end points of a line from the `LinesPercept` as well as its orientation are used to limit the considered angle range and the image area in which the hough transformation should be performed. The global maximum in the resulting hough space is used to calculate new start and end points. The remaining local

[1] Actually, the Jacobian used in each iteration is approximated numerically

maxima are used to determine whether the global maximum corresponds to the upper or lower edge of the line to take this into account by means of an offset when calculating the error.

A possible risk during sampling is that wrong lines are used to construct the samples, e. g. a detected line in the goal frame. This would lead to a corruption of the parameter optimization and result in an unusable calibration. In order to address this problem, an already almost perfect calibration is initially assumed and the lines are projected to field coordinates. In case the distances between the projected lines differ too much from the known actual distances, they are not used to construct samples but discarded. However, if lines are discarded continuously without being able to find suitable ones, the tolerated deviation is gradually increased, since it is known that the correct lines should be seen.

AutomaticCameraCalibrator

The information the `AutomaticCameraCalibrator` requires to optimize the camera parameters are line spots, gathered from the `LinesPercept` provided by the `LinePerceptor` (cf. Section 4.3.1). They are collected for both cameras while the head moves to predefined angles.

During this procedure it is assumed that the robot is placed on a predefined spot on the field. Since the user is typically unable to place the robot exactly on that spot and a small variance of the robot pose from its desired pose results in a large systematic error, additional correction parameters for the `RobotPose` are introduced and optimized simultaneously.

It may happen that parts of the field lines are covered (e. g. by robots or other team members, especially during competitions), or that lines, in particular those further away, are not detected due to a suboptimal color calibration. This can lead to an inappropriate number of collected line spots and in consequence to inaccurate values for the estimated tilts of the cameras. To address this problem it is possible to mark arbitrary points on the field lines by *CTRL* + left-clicking into the image at the point the additional sample should be. Furthermore inaccurate samples can be removed by left-clicking into the image.

The error function used in the Gauss-Newton algorithm takes the distance of a point to the next line in image coordinates into account. Image coordinates are preferred over field coordinates because the parameters and the error are in angular space making these a more accurate error approximation. Since this algorithm is designed specific for non-linear least squares problems like this, the time to converge takes an average of 5–10 iterations.

4.1.2.2 Image Coordinate System

Based on the camera transformation matrix, another coordinate system is provided which applies to the camera image. The `ImageCoordinateSystem` is provided by the module `CoordinateSystemProvider`. The origin of the y-coordinate lies on the horizon within the image (even if it is not visible in the image). The x-axis points right along the horizon whereby the y-axis points downwards orthogonal to the horizon (cf. Fig. 4.3). For more information see also [28].

Using the stored camera transformation matrix of the previous cycle in which the camera took an image enables the `CoordinateSystemProvider` to determine the rotation speed of the camera and thereby interpolate its orientation when recording each image row. As a result, the representation `ImageCoordinateSystem` provides a mechanism to compensate for different recording times of images and joint angles as well as for image distortion caused by the rolling shutter. For a detailed description of this method, applied to the Sony AIBO, see [19].

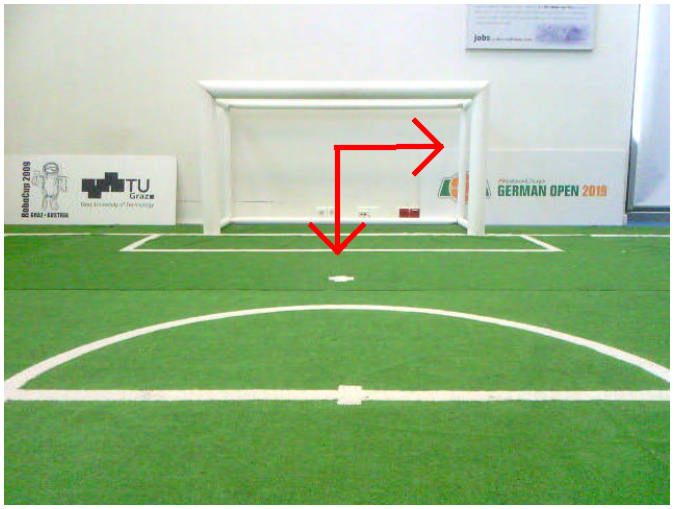


Figure 4.3: Origin of the `ImageCoordinateSystem`

4.1.3 Body Contour

If the robot sees parts of its body, it might confuse white areas with field lines or other robots. However, by using forward kinematics, the robot can actually know where its body is visible in the camera image and exclude these areas from image processing. This is achieved by modeling the boundaries of body parts that are potentially visible in 3-D (cf. Fig. 4.4 left) and projecting them back to the camera image (cf. Fig. 4.4 right). The part of the projection that intersects with the camera image or above is provided in the representation `BodyContour`. It is used by image processing modules as lower clipping boundary. The projection relies on the representation `ImageCoordinateSystem`, i. e., the linear interpolation of the joint angles to match the time when the image was taken.

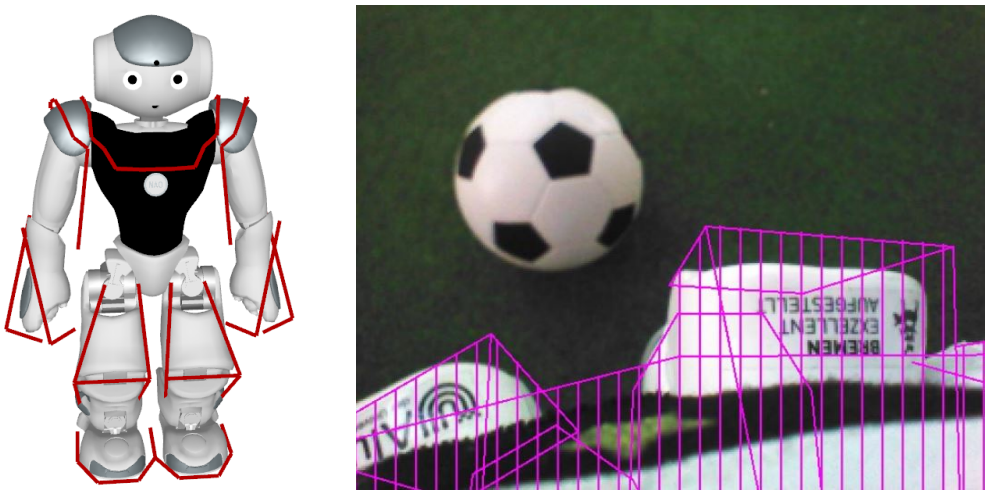


Figure 4.4: Body contour in 3-D (left) and projected to the camera image (right).

4.1.4 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

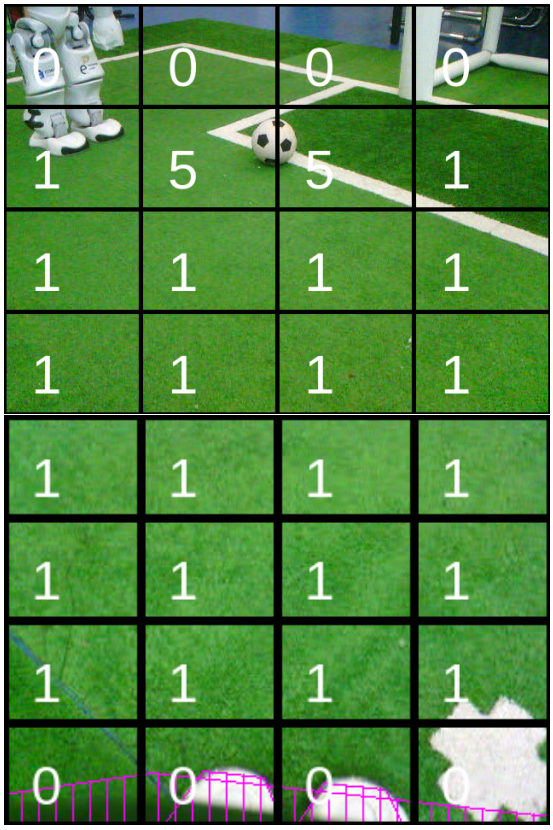


Figure 4.5: Weights for exposure control of the upper and lower camera.

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 four by four rectangular regions (cf. Fig. 4.5).

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. 4.5). Changing the parameters of the camera takes time (mainly waiting). Therefore, the code communicating with the camera driver runs in a separate thread to avoid slowing down our main computations.

4.1.5 Color Classification

Identifying the color classes of pixels in the image is done by the `ECImageProvider` when computing the `ECImage`. In order to be able to clearly distinguish different colors and easily define color classes while still being able to compute the `ECImage` for every camera image in real time, the YHS2 color space is used.

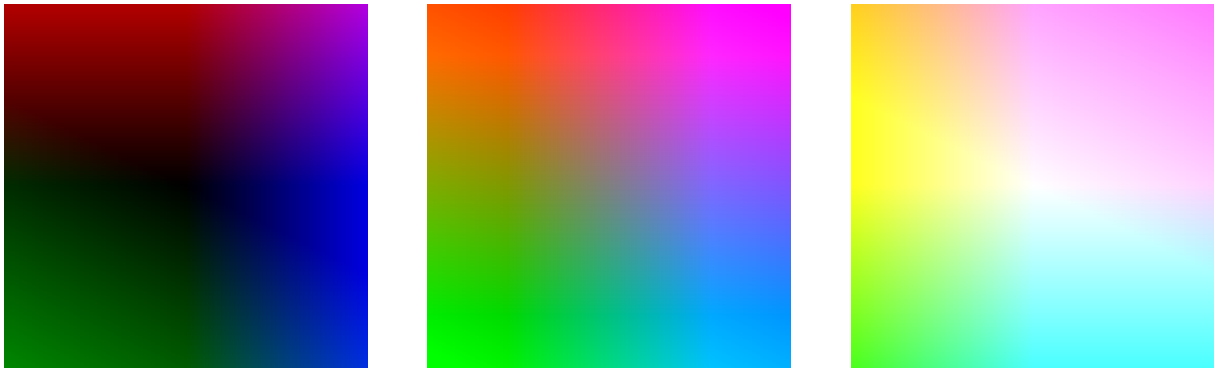


Figure 4.6: The UV plane of the YUV color space for $Y = 0$, $Y = 128$ and $Y = 255$.

YHS2 Color Space

The YHS2 color space is defined by applying the idea behind the HSV color space, i. e. defining the chroma components as a vector in the RGB color wheel to the YUV color space. In YHS2, the hue component H describes the angle of the vector of the U and V components of the color in the YUV color space, while the saturation component S describes the length of that vector divided by the luminance of the corresponding pixel. The luminance component Y is just the same as it is in YUV. By dividing the saturation by the luminance, the resulting saturation value describes the actual saturation of the color more accurately, making it more useful for separating black and white from actual colors. This is because in YUV, the chroma components are somewhat dependent on the luminance (cf. Fig. 4.6).

Classification Method

Classifying a pixel's color is done by first applying a threshold to the saturation channel. If it is below the given threshold, the pixel is considered to describe a non-color, i. e. black or white. In this case, whether the color is black or white is determined by applying another threshold to the luminance channel. However, if the saturation of the given pixel is above the saturation threshold, the pixel is of a certain color if its hue value lies within the hue range defined for that color.

In order to classify the whole camera image in real time, both the color conversion to YHS2 and the color classification are done using SSE instructions.

Figure 4.7 shows representations of an image from the upper camera in the YHS2 color space and a classification based on it for the colors white, black, green and "none" (displayed gray).

4.1.6 Segmentation and Region-Building

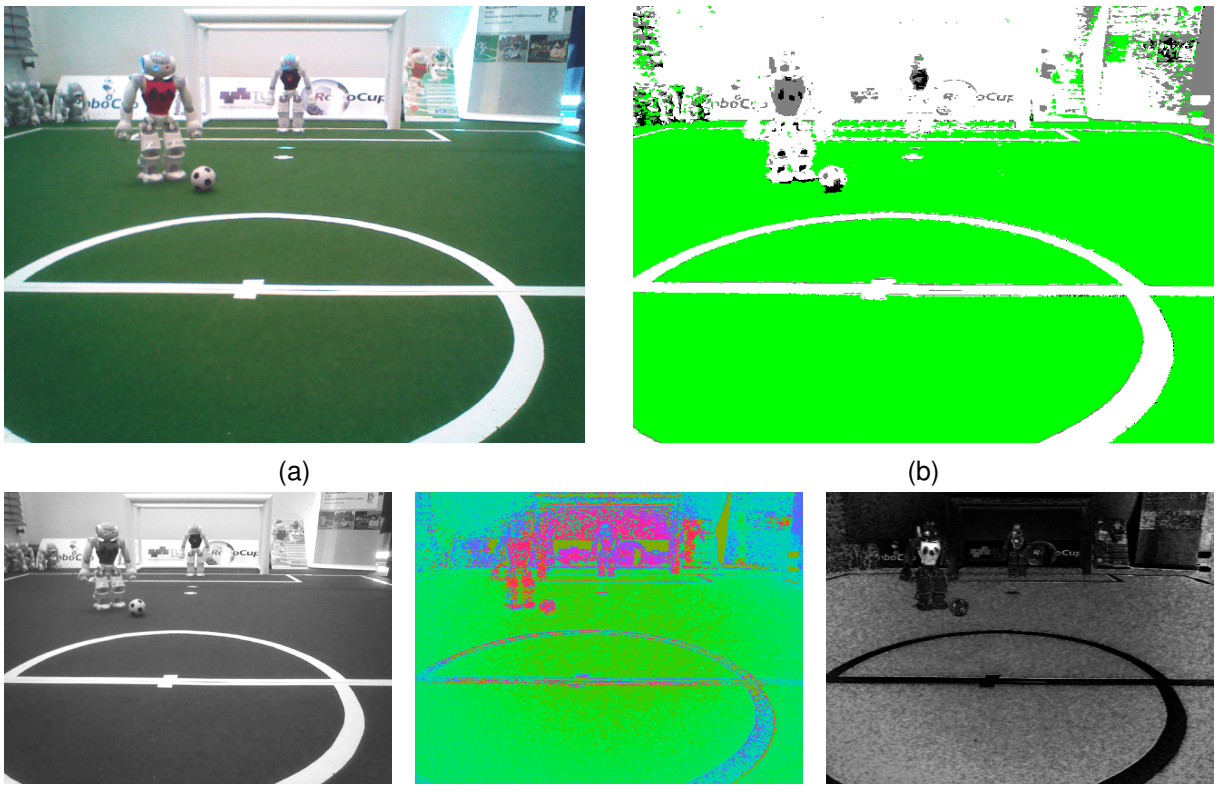
The `ColorScanlineRegionizer` scans along vertical and horizontal scan lines whose distances from each other are small enough to detect the field boundary, field lines, and the ball. These scan lines are segmented into regions, i. e. line segments of a similar color based on the `colored` part of the `ECImage`. In vertical direction, the sampling frequency is not constant. Instead, it is given by the `ScanGrid`, the steps of which depend on the vertical orientation of the camera and basically correspond to a distance of a bit less than the expected width of a horizontal field line at that position in the image. Whenever the color classes of two successive scan points on a vertical line differ, the region in between is scanned to find the actual posi-

tion of the edge between the two neighboring regions. The best position is determined to be one pixel above the position where enough pixels were found to be classified as color classes different from the one of the previous region. Figure 4.8 visualizes the subsampling.

In order to save computational time, the `ScanGrid` starts at the horizon as given by the `ImageCoordinateSystem`. Additionally, the vertical scan lines are divided into "low resolution" and "high resolution" scan lines. At first, only the low resolution scan lines are computed and used for determining the field boundary. Afterwards, the high resolution scan lines are computed for further processing, starting at the field boundary.

4.1.7 Detecting the Field Boundary

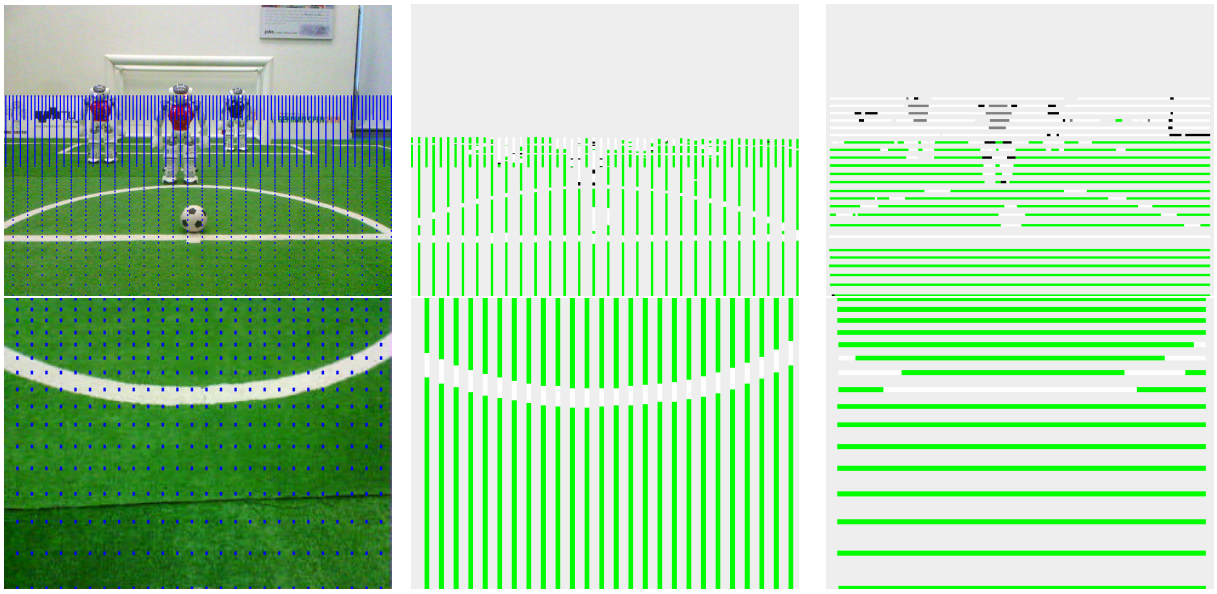
The rulebook of the Standard Platform League specifies in detail how a field and everything allowed to be on it during a game looks like. In contrast, very little is specified about the appearance of the world outside the field. The only exception is that another field that is visible must be at least 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 that, the extent of the field must be determined, i. e. its boundary in the current camera image must be detected.



(a) (b)

(c) (d) (e)

Figure 4.7: (a) An image from the upper camera of a NAO with (b) a corresponding color classification based on its (c) luminance, (d) hue and (e) saturation channels in the YHS2 color space.



(a) The `ScanGrid`: the blue dots are checked for building regions of colors.

(b) The high resolution vertical scan line regions.

(c) The horizontal scan line regions.

Figure 4.8: Visualization of the subsampling performed to compute the scan line regions.

4.1.7.1 Candidate Spots

The boundary of the field is basically an edge below where the image is mostly green and above where the image is mostly not green. This rule is employed when searching for candidate spots along vertical scan lines in the image (cf. Fig. 4.9b). 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. 4.9c). 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.

4.1.7.2 Guessing a Model

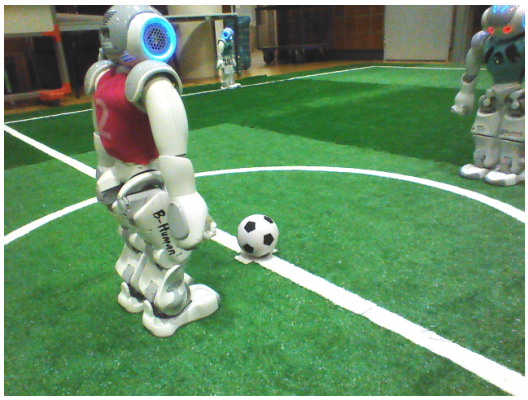
Not all candidate spots are located on the actual field boundary, because it can be hidden by other robots, goal posts, and referees. Therefore, it must be determined which spots are really located on the field boundary and which spots have to be ignored. Domain knowledge is used to ease this decision 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

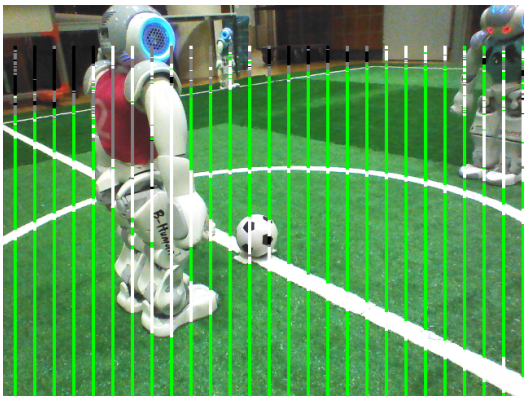
points to the field plane and dropping a perpendicular from the third point to the line spanned by the first two points. If the intersection point of the two lines fulfills a number of criteria (e. g. the corner must be convex and must be left of the third point), it is considered as a corner and the 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. 4.9d). 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.

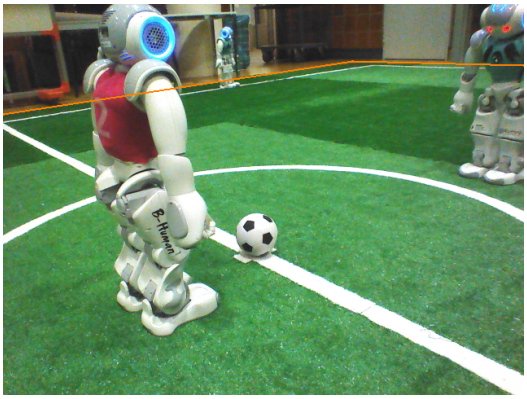


(a) The normal robot view.



(b) Vertical scan lines.

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



(d) The detected field boundary (orange line).

Figure 4.9: The main steps of the field boundary detection.

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

4.1.7.3 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.

4.2 Detecting the Ball

For the detection of the ball we use a multi-step approach. First, vertical scan lines are searched for ball candidates. A neural network based classification is then used to determine the real ball. The center and radius of the ball are finally estimated by another network in case the ball was identified successfully.

4.2.1 Searching for Ball Candidates

Our vision system scans the image vertically using scan lines of different density based on the size that objects, in particular the ball, would have at a certain position inside the image (cf. Section 4.1.6). To determine ball candidates, the `BallSpotProvider` searches these scan lines for sufficiently large gaps in the green that also have a sufficiently large horizontal extension and contain enough white. Only the first and second resolutions of the scanlines are used. From a specific distance, the ball is that small inside the image that the calculation can not provide good results. For this reason, ball spots farther away than this distance will be generated on white blobs that paled of green. In addition, all initial spots are ignored if they are near a previous one or are clearly inside a detected robot.

4.2.2 Spot Classification and Center Prediction

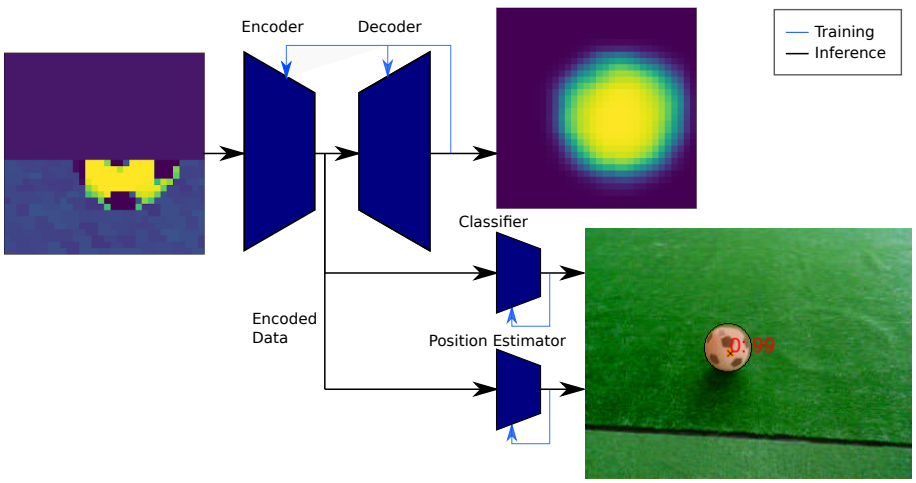


Figure 4.10: System of neural networks for ball classification and position estimation. The encoder is a Convolutional Neural Network (CNN), the classifier and the position estimator are Deep Neural Networks (DNNs).

The `BallPerceptor` takes the previously described spots which are provided in the representation `BallSpots` and classifies them into balls and other objects. For each spot the size a ball would have is calculated at the particular position in the camera image. Since the spots are not always in the middle of the ball a square patch with an edge length of 3.5 times the radius is used. The resulting area is then scaled to 32×32 pixels by leaving out pixels for downscaling and taking pixels several times for upscaling. The resulting patch is fed to the neural network if its projected position is on the field.

The core of the system consists of three neural networks whose architectures are described in Table 4.1. They form a system as shown in Fig. 4.10 that classifies the patches whether they show a ball or not and also predicts the size and position of the ball in the image. Those three neural networks perform each a different part necessary for this. The encoder, a CNN, extracts features from the patches that are both suitable for classification and position estimation. Based on these features the classifier determines if a ball is present or not. This prediction is a value between 0 and 1 and is used to classify the patches in three groups:

- If the result of the classifier is at least 0.9 the spot is accepted as ball and all other spots are discarded.
- Otherwise if the result of the classifier is at least 0.8 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 classifier is at least 0.7 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.

Upon a successful ball classification the third network estimates the position of the ball in the patch from the previously extracted features.

Layer Type	Output Size
Input	32x32x1
Convolutional	32x32x8
Batch Normalization	32x32x8
ReLU Activation	32x32x8
Max Pooling	16x16x8
Convolutional	16x16x16
Batch Normalization	16x16x16
ReLU Activation	16x16x16
Max Pooling	8x8x16
Convolutional	8x8x16
Batch Normalization	8x8x16
ReLU Activation	8x8x16
Max Pooling	4x4x16
Convolutional	4x4x32
Batch Normalization	4x4x32
ReLU Activation	4x4x32
Max Pooling	2x2x32

(a) Encoder

Layer Type	Output Size
Input	2x2x32
Flatten	128
Dense + Batch Norm + ReLU	32
Dense + Batch Norm + ReLU	64
Dense + Batch Norm + ReLU	16
Dense + Batch Norm + Sigmoid	1

(b) Ball Classifier

Layer Type	Output Size
Input	2x2x32
Flatten	128
Dense + Batch Norm + ReLU	32
Dense + Batch Norm + ReLU	64
Dense + Batch Norm + ReLU	3

(c) Position Estimator

Table 4.1: Architectures of the three neural networks for ball detection

4.3 Localization Features

To provide input for the self-localization (cf. Section 5.1) there exist multiple approaches to extract information from the visual sensors. Every (visual) knowledge relevant for position estimation that our robot is aware of is based on the field line detection. Although the penalty mark and the center circle are basically lines as well, they are – at least partly – detected separately to ensure more robustness for each single detection. Together, the representations of all three basic detections make up the first layer in the process chain of the visual perception. Each layer combines previously known information to provide more specific features. Information of every layer can be used by the self-localization. The main layers after the basic feature detection are the intersections perception and the following field feature perception.

4.3.1 Detecting Lines

The perception of field lines by the `LinePerceptor` relies mostly on the scanline regions. In order to find horizontal lines in the image, adjacent white vertical regions that are not within a perceived obstacle (cf. Section 4.4) are combined to line segments. Correspondingly, vertical line segments are constructed from white horizontal regions. These line segments and the center points of their regions, called *line spots*, are then projected onto the field. Using linear regression of the line spots, the line segments are then merged together and extended to larger line percepts. During this step, line segments are only merged together if at least a given ratio of the resulting line consists of white pixels in the image. Figure 4.11 shows the process of finding lines in the camera image.

4.3.2 Detecting the Center Circle

Besides providing the `LinesPercept` containing perceived field lines, the `LinePerceptor` also detects the center circle in an image if it is present. In order to do so, when combining line spots to line segments, their field coordinates are also fitted to circle candidates. After the `LinesPercept` was computed, spots on the circle candidates are then projected back into the image and adjusted so they lie in the middle of white regions in the image. These adjusted spots are then again projected onto the field and it is once again tried to fit a circle through them, excluding outliers.

If searching for the center circle using this approach did not yield any results, another method of finding the center circle is applied. We take all previously detected lines whose spots describe an arc and accumulate the center points of said arcs to cluster them together. If one of these clusters contains a sufficient number of center points, the average of them is considered to be the center of the center circle.

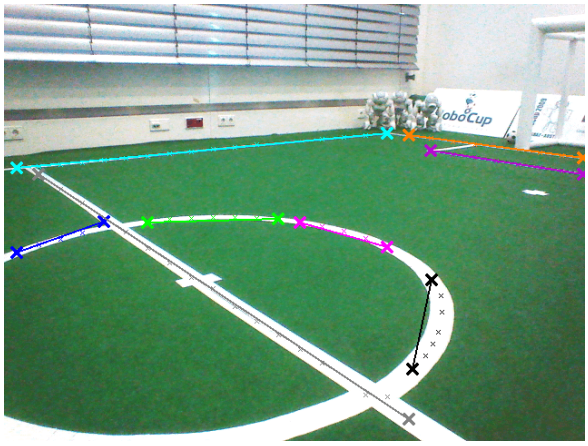
If a potential center circle was found by any of these two methods, it is accepted as a valid center circle only if – after projecting spots on the circle back into the image – at least a certain ratio of the corresponding pixels is white (cf. Fig. 4.11c).

4.3.3 Line Coincidence Detection

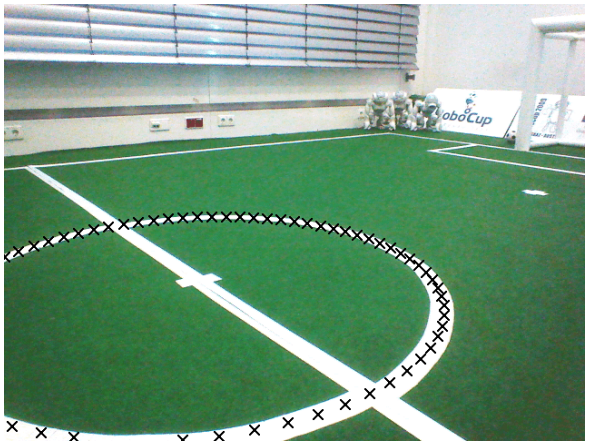
To find points where two lines coincide, we are calculating the intersections of the previously perceived lines (cf. Section 4.3.1). This approach is probably much faster and more accurate[3]

[3] Assuming a sufficiently accurate line detection.

(a) Line spots constructed from the scanline regions. Spots from horizontal regions are marked red, spots from vertical regions blue.



(b) The detected line segments in the image.



(c) Points on the circle candidate that were projected back into the image for checking if the corresponding pixels are white.

(d) Filtered line and circle percepts in the `FieldLines` representation.

Figure 4.11: The main steps of the line and circle detection.

than any separate (visual) perception such as e. g. corner search, but it means that detecting a line intersection without a line (percept) is impossible. In particular this disadvantage applies to line corners of a penalty area that extends into the image.

The `IntersectionsProvider` fills the `IntersectionPercept` based on the `LinesPercept`. Each known line will be compared to all other known lines, excluding those that are known to be part of the center circle. Because of the way the field is set up (cf. the current rules [2]), an intersection is only possible if the inclination of two compared lines in field coordinates is roughly $\frac{\pi}{2}$. In that case, the point of intersection of the two lines is calculated. In order to be valid, this point must lie in between both detected line segments. Next, we also want to specify the type of each intersection. The types of intersections are congruent with their names: `L`, `T`, and `X`. This means that intersections lying clearly in-between both lines are of type `X` while those that are clearly inside one line but roughly at one end of the other are considered to be of type `T`. If the point of intersection lies roughly at the ends of both lines, it is assumed to belong to an `L`-type intersection.

In case that parts of both lines are seen but the point of intersection does not lie on either line –

e. g. because an obstacle is standing in front of the intersection point – the lines will be virtually stretched to some extent so that the point of intersection lies roughly at their ends. The possible distance by which a line can be extended depends on the length of its recognized part.

The `IntersectionsPercept` stores the type of each intersection as well as its position, its base lines, and its rotation.

4.3.4 Preprocessed Lines and Intersections

Some features of the `LinesPercept` and `IntersectionsPercept` – such as the spots from which the lines were fitted and image coordinates – are not needed for further processing. On the other hand, now that the positions of all lines and intersections are known, additional information can be gained by making assumptions based on the known layout of the field lines. Therefore, as a next step, the `FieldLinesProvider` provides adapted representations of the `LinesPercept` and `IntersectionsPercept` for further use.

The representation `FieldLines` contains the processed line percepts (cf. Fig. 4.11d). For this, every line that is assumed to be seen on the center circle or inside the goal frame or net (cf. Section 4.3.6) was sorted out. Lines that extend into the goal frame are clipped to its boundaries. In addition to that, every line that is as long as a ball would appear at the corresponding position in the image is removed in order to avoid further calculations on false-positive lines that are fitted inside the ball. Apart from removing invalid percepts, lines in `FieldLines` also have additional properties. A line is marked as `mid` if it goes through the center circle, which means that it is probably the center line. The `long` attribute is given to any line that is longer than a specific on-field distance. A long line means the detected line probably belongs to the outer lines, center line or penalty front lines. For this reason, it is advisable to check on a distance that is longer than the penalty area sidelines or a part line inside the center circle, but as short as possible to receive the additional information as often as possible.

The representation `FieldLineIntersections` contains the processed line intersection percepts. It consists only of intersections that do not apply to any previously removed lines. If one of the lines belonging to an intersection was cut, the intersection type is corrected accordingly. Additionally, an intersection can now have a type of `mid` or `big`. If both lines of the intersection are of type `mid`, the intersection is also considered to be `mid`. Analogous to that, an intersection consisting of two `long` lines has the attribute `big`. After all this processing, `L`-type intersections are filtered out, if the angle between their two lines is not about ninety degrees.

4.3.5 Penalty Mark Perception

The penalty mark is one of the most prominent features on the field since it only exists twice. In addition, it is located in front of the penalty area and can thereby be easily seen by the goal keeper. By eliminating false positive detections, the penalty mark can be used as a reliable feature for self-localization. This is achieved by limiting the detection distance to 2 m and by using a contrast-based shape check. As a result, e. g., the B-Human goal keeper detected the penalty mark 6819 times during the 2019 SPL final and none of these detections was a false positive.

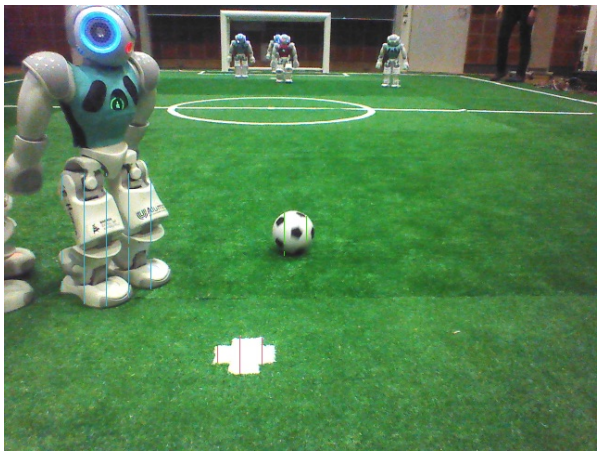
4.3.5.1 Approach

Similar to the ball detection (cf. Section 4.2) it is first searched for a number of candidates which are checked afterwards for being the actual penalty mark. However, while it is very important to

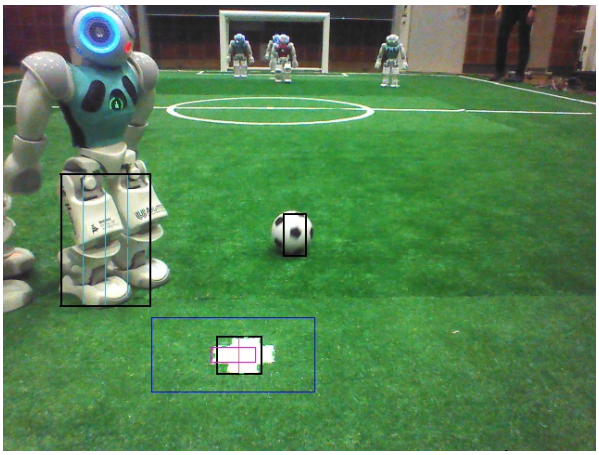
detect the ball as often as possible, it is sufficient for the self-localization to detect the penalty mark less frequently. Therefore, only a maximum of three candidates is checked per image to save computation time. Also similar to the ball detection, the search for candidates and the actual check are separated into two different modules. The check needs a contrast normalized Sobel (CNS) image (cf. Fig. 4.13) for the candidate region and since the computation of these regions costs time all the regions requested by different modules are grouped together to avoid computing overlapping regions twice. Therefore, the search for candidates has to be performed before the CNS image is computed and the shape check has to be executed afterwards.

4.3.5.2 Finding Candidate Regions

The `PenaltyMarkRegionsProvider` first collects all regions of the scanned low resolution grid that were not classified as green, i. e. white, black, and unclassified regions. In addition, the regions are limited to a distance of 2 m from the robot, assuming they represent features on the field plane. Vertically neighboring regions are merged (cf. Fig. 4.12a) and the amount of pixels that were actually classified as white in each merged region is collected. Regions at the lower end and the upper end of the scanned area are marked as invalid, i. e. there should be a field colored region below and above each valid region. The regions are horizontally grouped using the *Union Find* approach. To achieve a better connectedness of, e. g., diagonal lines, each region is virtually extended by the expected height of three field line widths when checking for the neighborhood between regions. For each group, the bounding box (cf. Fig. 4.12b) and the ratio between white and non-green pixels is determined. For all groups with a size similar to the expected size of a penalty mark that are sufficiently white and that do not contain invalid regions, a search area for the center of the penalty mark and a search area for the outline of the penalty mark are computed. As the search will take place on 16×16 cells, the dimensions are extended to multiples of 16 pixels.



(a) The vertical scanline regions are filtered for those that were not classified as green. They are clipped at a distance of 2 m from the observer (colored lines inside the robot, the ball, and the penalty mark). They are joined together as grouped regions (each group has a different color).



(b) Bounding boxes are computed for each group (black). For groups with the expected size and minimum white ratio, a search region for the center (red) and a region for the CNS image (blue) is computed. The contour points of the penalty mark are then fitted inside these regions (bright green).

Figure 4.12: Steps of the penalty mark detection

Figure 4.13: The normal robot view (left) and the contrast-normalized Sobel image (right). The colors indicate the directions of the gradients.

4.3.5.3 Computing CNS Regions

The `CNSRegionsProvider` integrates the outline regions into the set of regions for which the CNS image has to be computed. The `CNSImageProvider` will then perform the computation.

Figure 4.13 shows a representation of a computed CNS image for a whole picture.

4.3.5.4 Checking Penalty Mark Candidates

The `PenaltyMarkPerceptor` goes through the list of possible center regions for the penalty mark and determines the best match of the contour of the penalty area in each of these regions. It searches in one-pixel steps and checks for four different orientations, i. e. in 22.5° steps. However, as the contour detector also iteratively refines each match, the orientation can also be matched arbitrarily precise. If the match is sufficiently good, it will be checked more thoroughly whether the candidate is surrounded by green. If the candidate is surrounded by at least 90% green, it is returned as being a valid penalty mark.

4.3.6 Field Features

The self-localization uses field lines, their intersections, the center circle, and the penalty marks as measurements. All of these field elements are distributed over the whole field and can be detected very reliably, providing a constant input of measurements in most situations.

The *Field Features* are created by combining multiple basic field elements in a way that a robot pose (in global field coordinates) can be derived directly. The handling of the field symmetry, which leads to actually two poses, is described in Section 5.1.2. Overall, this approach provides a high number of reliable pose estimates, as the field lines on which it is based can be seen from many perspectives and have a reliable robust context (straight white lines surrounded by green carpet).

Every `FieldFeature` is a `Pose2f`, which is the pose of this in relative field coordinates. In addition, it has attributes that associate several percepts with explicitly defined global field elements such as the center line. Normally, all feature detections are using the preprocessed field elements (cf. Section 4.3.4). We are currently computing the following field features:

Penalty Area The `PenaltyAreaPerceptor`, which provides `PenaltyArea`, tries to find a penalty area with the help of the penalty mark and a line or two intersections. If a penalty mark is known, it searches a line that matches the corresponding penalty area front line. The penalty area can also be found by combining two `small` (not `big`) `T`-intersections or one `small` `T`-intersection and one `small` `L`-intersection.

Mid Circle The `MidCirclePerceptor` is combining a detected center circle with a detected `mid` line to provide the `MidCircle`.

Mid Corner A `MidCorner`, which is provided by the `MidCornerPerceptor`, is simply derived from a `big` `T`-intersection.

Outer Corner The `OuterCorner` refers to the areas where ground lines meet side lines, which means that the outer corner is on the right or left side. To find such a corner, the `OuterCornerPerceptor` combines a `big` `L`-intersection with an element of the penalty area.

Goal Frame Besides the obvious reason to have another field feature to get more information to put into the self-locator, the `GoalFrame` is also used to sort out false-positive lines that were detected inside the goal net. Thus, the `GoalFramePerceptor` has to calculate directly based on the line percepts. In general, the module searches for points around the goal and validates the rotation with a special use of the `FieldBoundary`.

4.4 Detecting Other Robots

The reliable detection of other robots is a complex and challenging problem for multiple reasons. Robots are not fully symmetrical and thus look differently depending on the angle of view, they can have different postures (e. g. standing, lying, or getting up) or can cover each other. Furthermore, changing lighting conditions might make it difficult to use the robot's colored jersey as a reliable cue.

Additional factors have to be taken into account for the detection in the upper camera. Since it covers larger parts of the field, robots appear in many different scales (very far away as well as very close) and other objects that could be confused with a robot, such as the goal or the referee, are often included in the image, making a robust detection even more difficult. In contrast, the lower camera only captures a very small portion of the field, which is directly in front of the robot's feet. Therefore, objects in the lower image are close to the robot and rather big, making other robots comparatively easy to recognize as more or less large, continuous blobs with some additional characteristics.

Because of these, we use a more complex, machine learning based method for the robot detection in the upper camera, while a more simple, conventional method is sufficient for the detection in the lower camera.

4.4.1 Detecting Robots in the Upper Image

For detecting robots in the upper camera, a neural network is utilized. Same as for the ball detection, we use only the Y channel of the YUV images, since it suffices to achieve good results and contributes to more robustness. However, in contrast to the method for ball detection, the network does not analyze individual image patches, but processes an entire image at once. For this purpose, we use the `Thumbnail` image in the resolution 80×60 as input. Before passing

Figure 4.14: Overview of the robot detection pipeline used for the upper camera. A gray- and downscaled (80 × 60 pixels) version of the camera image is passed to the neural network, which determines the bounding boxes for all robots present in the image.

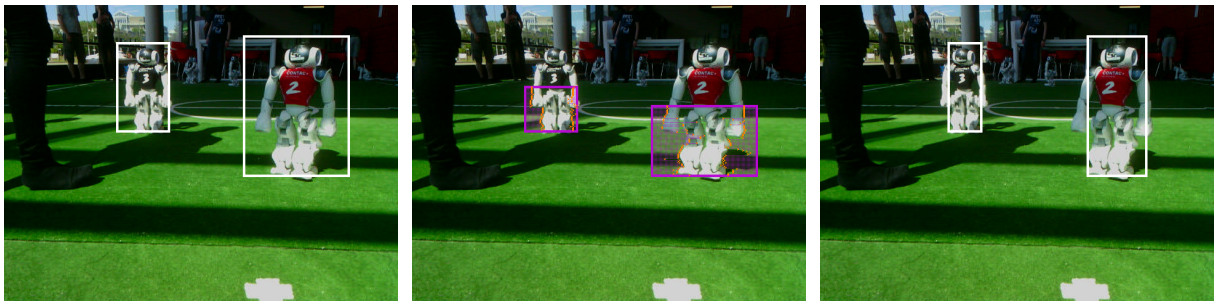


Figure 4.15: The left image shows the original bounding boxes without post-processing. In the middle image the regions within the post-processing is performed are indicated by violet rectangles, the considered spots are marked as violet dots. The orange crosses are the spots at which significant contrast changes have been detected, from which the new boundaries, which can be seen in the right picture, are determined.

an image to the network, a 2 % min-max normalization is applied on it as an additional pre-processing step, to be more robust against changing and challenging lighting conditions. After these pre-processing steps the neural network determines the bounding boxes for all robots present in the image (cf. Fig. 4.14). The used network architecture is shown in Table 4.2. For a more detailed description of this approach, see [18].

The coordinates of the resulting bounding boxes are converted from the scaled down resolution to the original one. Since the network uses the principle of anchor boxes to determine the bounding boxes, they are sometimes wider than necessary. For this reason, an additional post-processing step is performed. Roughly estimated that the legs of the robot make up the lower half of a bounding box, we define horizontal scan lines in fixed steps from the center of the bounding box to the lower border of it. Along all of these scan lines we determine the outermost spots with significant contrast changes and use the medians of them as new left/right boundaries (cf. Fig. 4.15).

4.4.2 Detecting Robots in the Lower Image

The detection of robots in the lower camera uses a contrast based approach. For this the image is first divided into 16 × 16 regions (cf. Fig. 4.16a) and scanned along a grid, in order to locate spots with a significant change of contrast in the horizontal, vertical or both of these directions. Thereby only image areas below the `FieldBoundary` and above the `BodyContour`

Layertype	Filters	Filter Size	Strides	Padding	Output
BNorm	-	-	-	-	80×60
Conv	16	3×3	1	Same	80×60
SConv	24	3×3	2	Same	40×30
BNorm	-	-	-	-	40×30
SConv	16	3×3	1	Same	40×30
SConv	20	3×3	1	Same	40×30
SConv	20	3×3	2	Same	20×15
BNorm	-	-	-	-	20×15
SConv	20	3×3	1	Same	20×15
SConv	20	3×3	1	Same	20×15
SConv	24	3×3	1	Same	20×15
SConv	24	3×3	2	Same	10×8
BNorm	-	-	-	-	10×8
Conv	24	3×3	1	Same	10×8
Conv	24	3×3	1	Same	10×8
Conv	24	3×3	1	Same	10×8
Conv	24	3×3	1	Same	10×8
Conv	20	1×1	1	Same	10×8

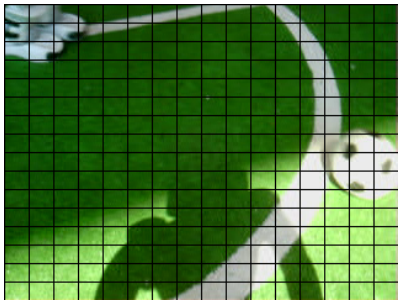
Table 4.2: The network architecture used for the robot detection in the upper camera.

are scanned. To exclude contrast changes within the field, caused e. g. by lighting conditions, only pixels whose saturation is below a rough threshold are considered (cf. Fig. 4.16b). The spots found are noted accordingly in the region in which they are located. In addition, spots with low saturation but high brightness are also registered separately, as large parts of a robot are plain white/greyish.

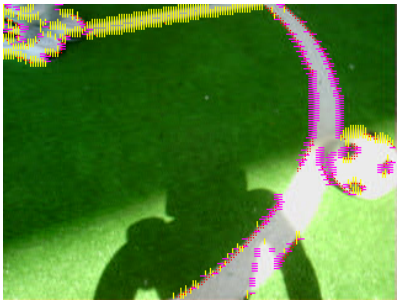
After scanning the image, the defined 16×16 image regions are divided into different classes, based on the number and type of the spots they contain (cf. Fig. 4.16c). Due to the nature of field lines and goalposts, e. g. mostly rectangular with two distinct straight edges, image regions containing these show a similar distribution of contained spots and are thus usually in the same class. On the other hand, image regions containing robots are much more heterogeneous due to the different shapes of the individual body parts. Based on this observation, contiguous areas of regions in the same class are marked negligible and are no longer considered in the following. This way, field lines, except field line intersections, and goalposts can be effectively filtered out (cf. Fig. 4.16d).

As the next step, a density-based clustering is performed for the remaining regions. Clusters containing a ball, a penalty mark or a field line intersection are rather small and often don't reach the top image border (cf. Fig. 4.16e). Clusters that fulfill both of these criteria are therefore discarded. For those that reach the top image border but are comparatively small, there must be a detection in the upper camera image that ends at the lower image border and lies in the same direction as the cluster found in the lower image.

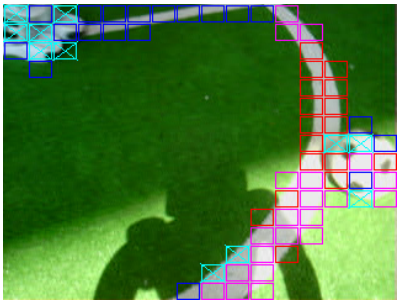
The remaining clusters are assumed to be robots and bounding boxes are constructed for them. For this the coordinates of the regions contained in the clusters are used to determine initial boundaries which getting refined using the same procedure as for the upper camera, except that this time the whole height of the bounding boxes is considered (cf. Fig. 4.16f).



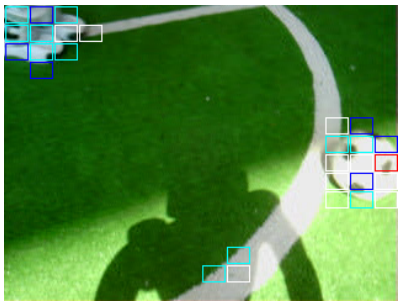
(a) The original image with the 16×16 image regions overlayed. Large parts of the image are covered by the ball and field lines, additionally the field has strong changes in brightness.



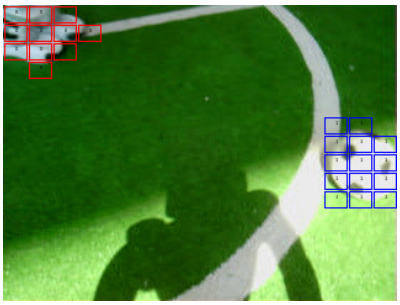
(b) The found changes in contrast. The yellow lines indicate vertical contrast changes, the purple lines horizontal ones and the red dots indicate spots with vertical as well as horizontal contrast changes.



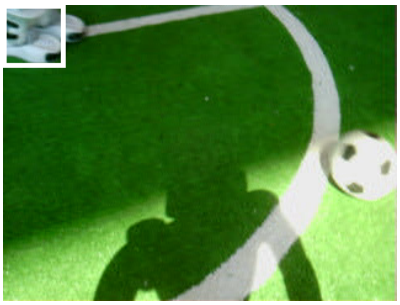
(c) The regions assigned to different classes based on the contained spots, indicated by different colors. The regions marked magenta contain mainly bright spots, the cyan marked ones shouldn't be discarded as they don't contain a clear majority of any spot type.



(d) The remaining regions after discarding contiguous, homogeneous areas. The white marked regions are the ends of such areas which were not discarded due to their proximity to heterogeneous areas.



(e) The clusters formed out of the remaining image regions. Clusters that would be too small are directly discarded.



(f) The robot detections derived from the formed clusters. Due to its size and the position in the middle of the image, the cluster containing the ball does not represent a valid robot obstacle.

Figure 4.16: The main steps of the robot detection pipeline used for the lower camera.

4.4.3 Determining the Jersey Color

After a robot is detected in the image it is attempted to determine its jersey color. To avoid a calibration of the colors, a differential approach is used. For each detected robot, the image is sampled at a certain height in the region of a perspectively distorted rectangle.[4] 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 robot.

The color classification first distinguishes between saturated and unsaturated 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. Note that green jerseys will not be detected at all by this approach.

[4] 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.

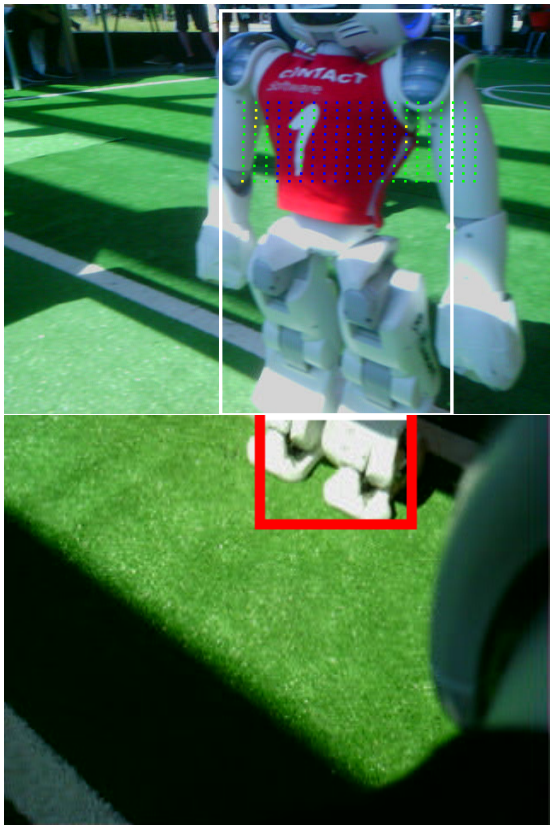


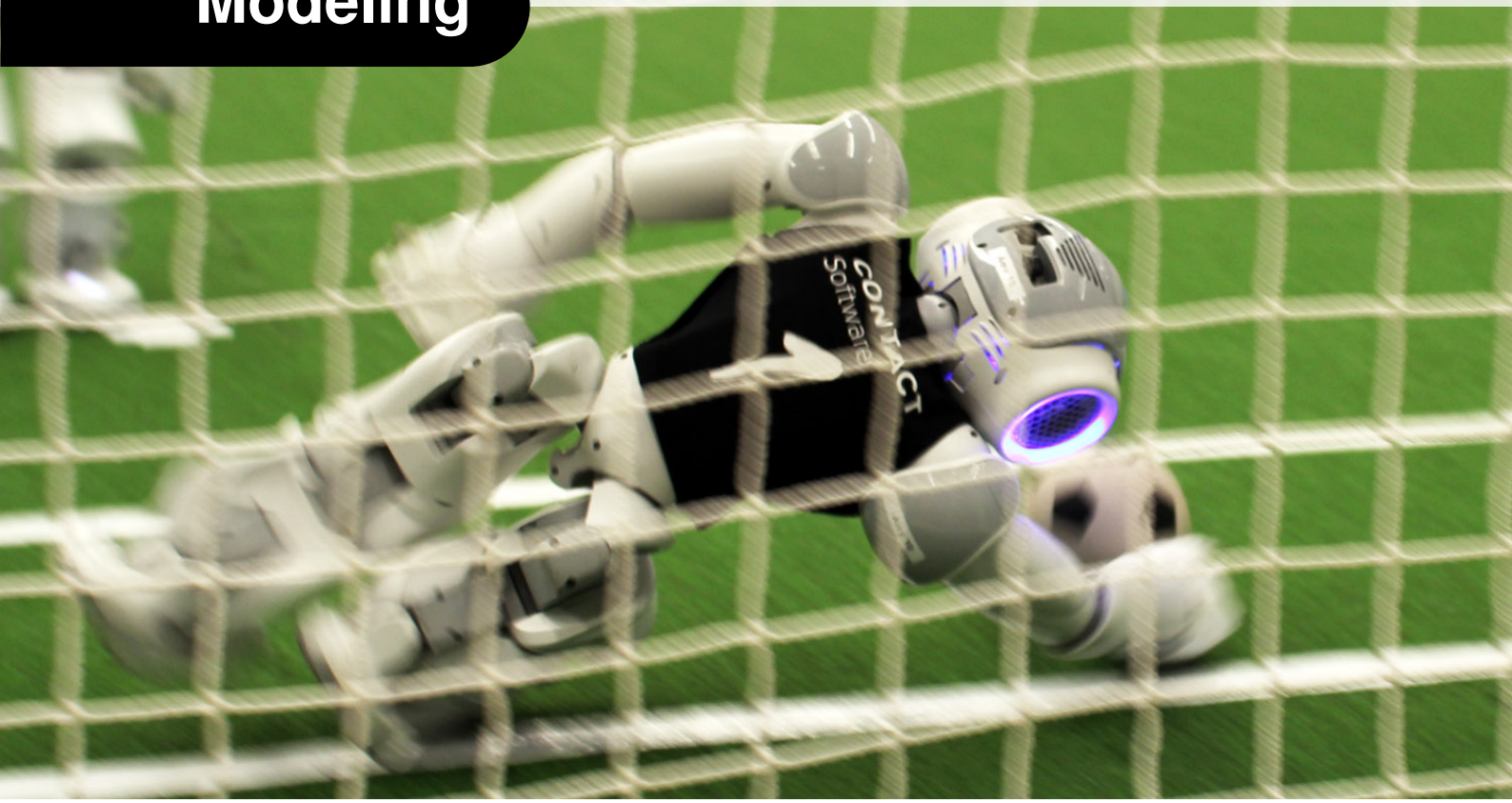
Figure 4.17: Combined robot detection in upper and lower camera images. The thick yellow, blue, and green dots mark the region that was scanned for the jersey color. Yellow dots mark the own (black) team, blue ones the opponent (red) team, and green ones were ignored. The robot found in the lower image is marked as red, because a red jersey was detected in the upper image.

4.4.4 Propagating Information from Upper to Lower Image

If a robot 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 jersey colors of incomplete robots in the upper image will be assigned to robots in the lower image if both detections were in the same direction (cf. Fig. 4.17).

Chapter 5

Modeling



To compute an estimate of the current world state – including the robot's own position and orientation, the ball's position and velocity, and the positions of other robots – given the noisy and incomplete data provided by the perception layer, a set of modeling modules is necessary. Some of these modules also incorporate information sent by teammates, e. g. to compute a team-wide ball state estimate. Furthermore, for some years now, the sound perceived by the robot's microphones has to be analyzed to detect the referee's whistle. An overview of the major modules for these tasks is given in this chapter.

5.1 Self-Localization

A robust and precise self-localization has always been an important requirement for successfully participating in the Standard Platform League. B-Human has always based its self-localization solutions on probabilistic approaches [30] as this paradigm has been proven to provide robust and precise results in a variety of robot state estimation tasks. However, due to the high amount of noise and the problems arising from the field's symmetry, the main state estimation module is complemented by additional modules that generate new robot pose alternatives and check, if a robot's current orientation is point-symmetrically flipped. Overall, B-Human's implementation has not significantly changed in recent years, except for some minor adaptions and bug fixes, which are not described in this report.

5.1.1 Probabilistic State Estimation

The pose state estimation is handled by multiple hypotheses that are each modeled as an Unscented Kalman Filter [11]. The hypotheses management and hypotheses resetting (cf. Section 5.1.2) is realized as a particle filter [4]. Both approaches are straightforward textbook implementations [30], except for some adaptions to handle certain RoboCup-specific game states, such as the positioning after returning from a penalty. In addition, we only use a very low number of particles (currently 12), as multimodalities do not often occur in RoboCup games.

The current self-localization implementation is the `SelfLocator` that provides the `RobotPose`. Furthermore, for debugging purposes, the module also provides the loggable representation `SelfLocalizationHypotheses` which contains the states of all currently internally maintained pose hypotheses.

Overall, this combination of probabilistic methods enables a robust, precise, and efficient self-localization on the RoboCup field. However, eventually, the result always depends on the quality and quantity of the incoming perceptions.

5.1.1.1 Perceptions Used for Self-Localization

Several years ago, B-Human used goals as a dominant feature for self-localization. When the field was smaller and the goal posts were painted yellow, they were easy to perceive from most positions and provided precise and valuable measurements for the pose estimation process. In particular the sensor resetting part (cf. Section 5.1.2), i. e. the creation of alternative pose estimates in case of a delocalization, was almost completely based on the goal posts perceived. In 2015, we still relied on this approach, using a detector for the white goals [27]. However, as it turned out that this detector required too much computation time and did not work reliably in some environments (requiring lots of calibration efforts), we decided to perform self-localization without goals but by using the new complex field features (cf. Section 4.3.6 as well as [26]) instead. However, the new NAO's computing power and our recent progress in the application of deep neural networks might lead to new robust goal perceptions in the near future. These perceptions could be integrated in the current implementation easily.

In addition, the self-localization still uses basic features such as field lines, their crossings, the penalty mark, and the center circle as measurements. All these field elements are distributed over the whole field and can be detected very reliably, providing a constant input of measurements in most situations.

The complex field features are also used as measurements but not as a perception of a relative landmark. Instead, an artificial measurement of a global pose is generated, reducing the translational error in both dimensions as well as the rotational error at once. Furthermore, in contrast to the basic field elements that are not unique, no data association (cf. Section 5.1.1.2) is required. The handling of the field symmetry, which leads to actually two poses, is similar to the one that is done for the sensor resetting as described in Section 5.1.2.

5.1.1.2 Resampling based on Particle Validity

One important part of any Monte Carlo localization approach is the *resampling* (cf. [4]), i. e. to determine which particles become copied how often to the new particle set representing the current state's probability distribution. For this purpose, each particle has a weighting that describes how "good" it is; the higher the weighting, the higher the likelihood that the particle will be copied to the new set. There is no general-purpose approach to compute these weights,

each application requires its own way to determine the quality of a particle instead.

In our implementation, each particle's validity is directly used as its weighting (combined with a base weighting, as described below). The validity is our measure for a particle's compatibility to the recent perceptions and is computed during the process of data association. To use a perception in the UKF's measurement step, it needs to be associated to a field element in global field coordinates first. For instance, given a perceived line and a particle's current pose estimate, the field line that is most plausible regarding length, angle, and distance is determined. In a second step, it is checked whether the difference between model and perception is small enough to consider the perception for the measurement step. For all kinds of perceptions, different thresholds exist, depending on the likelihood of false positives and the assumed precision of the perception modules. After the association process has been carried out for all perceptions, the current validity v_c can be computed by setting the number of successfully associated perceptions in relation to the total number of perceptions in this frame. For this computation, we consider different weights for different kinds of perceptions, i. e. a field feature – which is a very reliable perception – that cannot be associated has a higher impact on the validity than a field line, which might erroneously be detected in other robots or a goal net, that has not been matched.

To avoid strong oscillations of the validity, which might cause instabilities within the sample set and hence a robot pose that reacts too quickly to false measurements, a particle's validity v_p is filtered over n frames (currently, n is configured to 60):

$$v_p = \frac{v_p^{old} \times (n-1) + v_c}{n} \tag{5.1}$$

Furthermore, the addition of a base weighting w_b in the computation of a particle's weighting w_p contributes to a more stable resampling process:

$$w_p = w_b + (1 - w_b) \times v_p \tag{5.2}$$

5.1.2 Sensor Resetting based on Field Features

When using a particle filter on a computationally limited platform, only few particles, which cover the state space very sparsely, can be used. Therefore, to recover from a delocalization, it is a common approach to perform *sensor resetting*, i. e. to insert new particles based on recent measurements [15]. The field features provide exactly this information – by combining multiple basic field elements in a way that a robot pose (in global field coordinates) can be derived directly – and thus are used by us for creating new particles. These particles are provided as `AlternativeRobotPoseHypothesis` by the `AlternativeRobotPoseProvider`.

As false positives can be among the field features, e. g. caused by robot parts overlapping parts of lines and thereby inducing a wrong constellation of elements, an additional filtering step is necessary. All robot poses that can be derived from recently observed field features are clustered and only the largest cluster, which also needs to contain a minimum number of elements, is considered as a candidate for a new sample. This candidate is only inserted into the sample set in case it significantly differs from the current robot pose estimate.

To resolve the field's symmetry when handling the field features, we use the constraints given by the rules (e. g. all robots are in their own half when the game state switches to *Playing* or when they return from a penalty) as well as the assumption that the alternative that is more compatible to the previous robot pose is more likely than the other one. An example is depicted in Fig. 5.1. This assumption can be made, as no teleportation happens in real games. Instead,

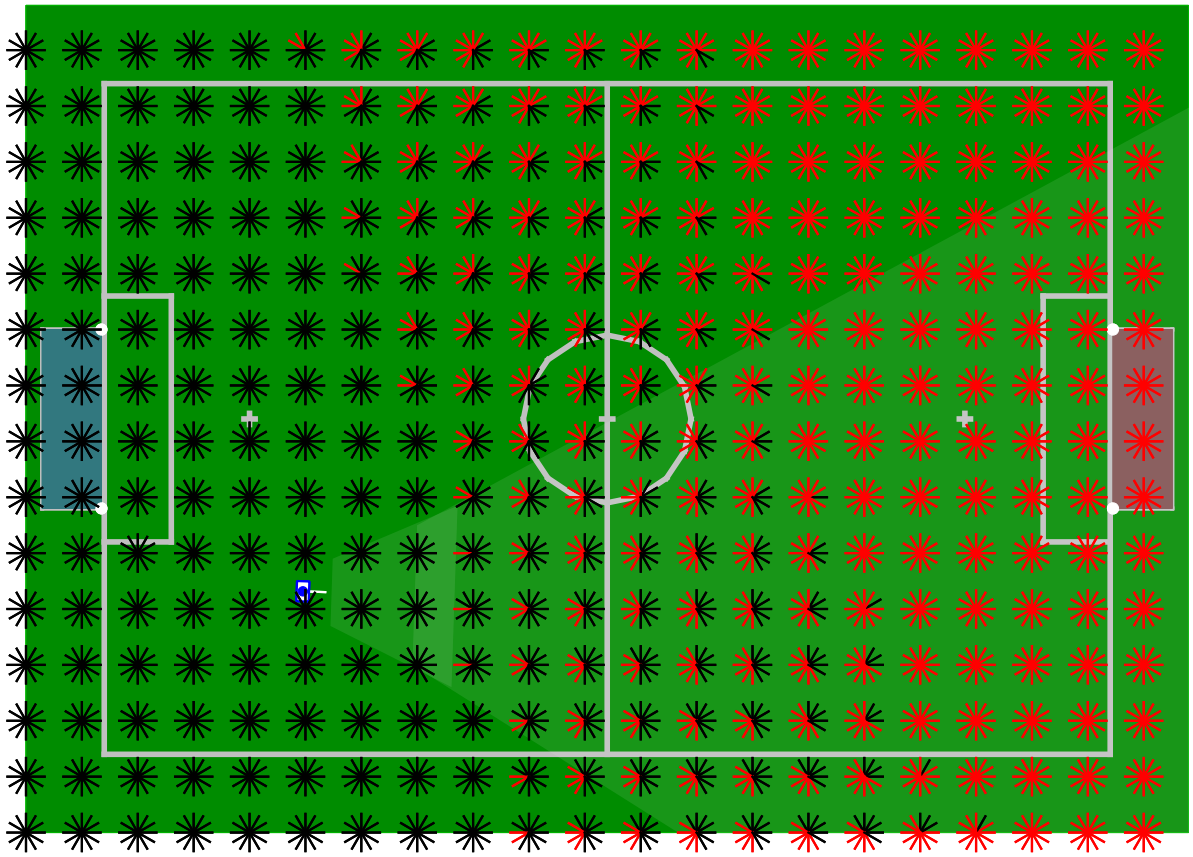


Figure 5.1: Whenever a new alternative pose is inserted into the sample set, it must be decided, if it can be used directly or if it needs to be mirrored. This decision depends on the current robot pose. This figure visualizes the possible decisions for one example robot pose (located on the right side of the own half). The stars denote example alternative poses. A red line means: *if a new pose is computed here, it must be mirrored before its insertion into the sample set*. Consequently, black lines mean that the pose can be used directly. As one can see, the current formula prefers the direction towards the opponent goal over the current position.

most localization errors result from situations in which robots slightly lose track of their original position and accumulate a significant error by repeatedly associating new perceptions to false parts of the field model.

5.1.3 Detecting and Correcting Mirrored Pose Estimates

As aforementioned, the self-localization already includes some mechanisms that keep a robot playing in the right direction. However, it still happens that sometimes a single robot loses track of its playing direction. To detect such situations, a separate module – the `SideConfidenceProvider` – compares a robot's ball observations with those of the teammates. If the current constellation indicates that the robot has probably been flipped, the representation `SideConfidence` is set accordingly and the subsequently executed `SelfLocator` can mirror all particles.

The comparison of the ball observations is implemented by a list of mutual agreements: Whenever a teammate communicates a new ball, it is checked, whether both robots can *agree* about the ball position or if they *disagree*, i. e. agree about the mirrored alternative. If none of both

alternatives appears to be likely, the status is *unknown*. The `SideConfidenceProvider` keeps an internal list of all teammates and the result of the last comparisons.

To agree about a ball position, both observations need to have occurred within a certain amount of time (currently one second) and at a similar place (currently up to one meter distance). As not all robots can see the ball frequently, each agreement / disagreement will be remembered for some time (currently eight seconds). However, if the (own or teammate) ball is located at certain positions, it will not be used to compute any agreement / disagreement: as the field is point-symmetric, the area around the field's center is useless for any computations; as there might be false positives on penalty marks or on lines that are close to robots or close to other lines, all balls close to any of these are discarded. Furthermore, as the robots have to agree within a certain time slice, a fast rolling ball could be perceived at very different positions and has to be ignored, too.

Finally, to make a decision whether a robot has flipped or not, the list of mutual agreements is checked:

- The localization is considered as being correct, if the robot agrees with more teammates about the ball position than it disagrees with.
- The robot is considered a being on the mirrored pose, if it disagrees with multiple teammates but does not agree with anyone.

During the whole RoboCup 2019 competition, we did not observe any robot that required such a correction of its pose estimate. Thus, one can assume that the other self-localization components are currently precise and reliable enough to render the `SideConfidenceProvider` useless.

5.2 Ball Tracking

To keep track of the current ball state, all robots maintain two ball models: a local one that estimates the ball position and velocity based on the own observations and a global one that fuses the own observations with observations communicated by the teammates. Furthermore, to predict upcoming positions of rolling balls, a model for the current carpet's friction can be learned.

5.2.1 Local Ball Model

Given the ball perceptions described in Section 4.2, the `BallLocator` uses Kalman filters to derive the current ball position and velocity, represented as the `BallModel`. Since ball motion on a RoboCup soccer field has its own peculiarities, our implementation extends the usual Kalman filter approach in several ways described below. The major parts of the ball tracking implementation remained unchanged for many years. Before 2016, the number of false positive ball perceptions has been zero in most games. Hence, the ball tracking was implemented as being as reactive as possible, i. e. every perception was considered. However, the introduction of the new black and white ball required the addition of a set of additional checks, which are now part of a new supporting module, the `BallPerceptFilter`. Furthermore, to keep the core ball tracking implementation as clear as possible, the collision detection was also moved to an additional module, the `BallContactCheckerProvider`.

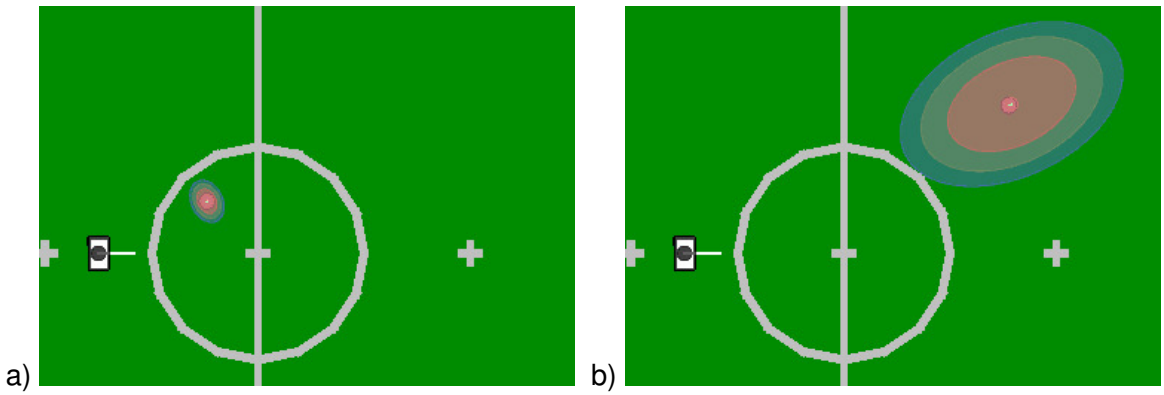


Figure 5.2: Ball model and measurement covariances for (a) short and (b) medium distances. The orange circles show the ball models computed from the probability distribution. The larger ellipses show the assumed covariances of the measurements.

5.2.1.1 State Estimation

First of all, the problem of multimodal probability distributions, which is naturally handled by the particle filter, deserves some attention when using a Kalman filter. Instead of only one, we use twelve multivariate Gaussian probability distributions to represent the belief concerning the ball. Each of these distributions is used independently for the prediction step and the correction step of the filter. Effectively, there are twelve Kalman filters running in every frame. Only one of these distributions is used to generate the actual ball model. That distribution is chosen depending on how well the current measurement, i. e. the position the ball is currently seen at, fits and how small the variance of that distribution is. That way, we get a very accurate estimation of the ball motion while being able to quickly react on displacements of the ball, for example when the ball is moved by the referee after being kicked off the field.

To further improve the accuracy of the estimation, the twelve distributions are equally divided into two sets, one for rolling balls and one for balls that do not move. Both sets are maintained at the same time and get the same measurements for the correction steps. In each frame, the worst distribution of each set gets reset to effectively throw one filter away and replace it with a newly initialized one. The covariance matrix determining the process noise for the prediction step is fixed over the whole process. Contrary to that, the covariance for the correction step is derived from the actual measurement; it depends on the distance between robot and ball (cf. Fig. 5.2).

5.2.1.2 Filtering Ball Perceptions

Although our ball perception is quite robust in general (cf. Section 4.2), several false positives per game cannot be avoided due to the similarity between the ball's shape and surface and some robot parts. The `BallPerceptFilter` implements a number of different checks for typical cases. All perceptions that are not rejected by this module become provided as `FilteredBallPercepts`, which are the actual input for the state estimation process as described above.

First of all, there must be multiple ball perceptions within a certain area and within a maximum time frame before a perception can be considered for the tracking process. This slightly reduces the module's reactivity but is still fast enough to allow the execution of ball blocking moves in a timely manner. Furthermore, a common problem is the detection of balls inside close robots.

A part of these perceptions, i. e. those resulting from our teammates, is excluded by checking against the communicated teammate positions. As our self-localization is very precise and robust, we can use the robot's estimated position to reject ball perceptions that appear to be outside the field, e. g. caused by stuff lying near the field.

By adding many checks to exclude potentially false ball observations in perception (cf. Section 4.2) as well as during state estimation, many true positives become rejected, too. Especially many rolling balls lead to a quite low response of the detector. These perceptions, which are annotated as *guessed* balls, fall into the same category as many false positives and cannot directly be used for a reliable state estimation. However, perceiving rolling balls and predicting their velocity is important to successfully perform blocking motions. To compensate for this problem, two mechanisms have been implemented: If a *guessed* ball is perceived close to a position at which recently a ball observation with a high response was detected, it is considered to be the same ball and thus passed to the state estimation process. Furthermore, the `BallPerceptFilter` performs a check, if a series of perceptions (*guessed* one as well as those of higher quality) is on a line that is compliant with the hypothesis of a rolling ball. If this is the case, these perceptions can be used in the ball state estimation process, too, as sequences of false positives that resemble a rolling ball are highly unlikely.

The ball percept filtering is an important process to keep the ball tracking reliable. Nevertheless, it should be noted that a few false positives still pass all checks.

5.2.1.3 Checking Collisions with the Ball

The robot itself can influence the motion of the ball either by kicking or just standing in the way of a rolling ball. These events need to be considered to reliably keep track of the ball. The `BallContactCheckerProvider` implements this functionality by geometrically clipping a ball position at the robot's feet, incorporating a possible velocity of the colliding foot. As a result, a new ball position and a new ball velocity can be computed, including new covariances for both. To allow these computations to be made for all ball hypothesis that are maintained by the `BallLocator`, the `BallContactCheckerProvider` provides the `BallContactChecker` representation that contains the required collision function.

5.2.2 Friction and Prediction

For a precise Kalman filter prediction step as well as for an estimation of a rolling ball's end position, which is required by some behaviors such as passing a goalkeeper reactions, a model of the friction between ball and ground is essential. For this model, we assume a simple linear model for ball deceleration

$$s = v \times t + \frac{1}{2} \times a \times t^2 \tag{5.3}$$

with s being the distance rolled by the ball, t the time, and a the friction coefficient. The coefficient can be configured for each individual field and has to be set in the file *ballSpecification.cfg*.

The coefficient can be guessed by rolling a ball and comparing its end position with the predicted end position. To make this process more convenient, the code contains the module `FrictionLearner`, which is deactivated during games, in order to automatically determine the floor-dependent friction parameter. When the module is active, a robot with remote connection can be placed on the field and the ball has to be rolled through its field of view with medium speed. Preferably, the start and end position of the ball should both be outside the robot's field

of view and the field should be even. The module buffers all ball perceptions and after the ball is outside the field of view or has come to a stop, a least squares optimization determines the coefficient a that provides the best explanation for the ball's movement. The result is printed to the SimRobot console window afterwards. As this process is subject to noise and outliers happen quite often, it should be carried out multiple times and the results have to be checked and compared carefully by the user.

All computations for ball state prediction that involve friction have been encapsulated in the class *BallPhysics* to be used by different modules.

Some modules, such as the ball perception (cf. Section 4.2), might require the current ball position estimate before the `BallModel` has been updated in current execution frame. Hence, the `WorldModelPredictor` provides a `WorldModelPrediction` by using the previous frame's robot pose and ball state estimate and applying odometry and motion updates.

5.2.3 Team Ball Model

The team ball model is calculated locally by each robot but takes the ball models of all teammates into account. This means that the robot first collects the last valid ball model of each teammate, which is in general the last received one, except for the case that the teammate is not able to play, for instance because it is penalized or has fallen down. In this case, the last valid ball model is used. The only situation in which a teammate's ball model is not used at all is if the ball was seen outside the field, which is considered as a false perception.

After the collection of the ball models, they become clustered. Two balls are added to the same cluster, if the distance between them is below a certain threshold. Here, a ball can be part of multiple clusters. Afterwards, all balls of a cluster become merged in a weighted sum calculation. There are multiple factors that are considered in this calculation:

- The approximated validity of the self-localization of the perceiving robot: the higher the validity, the higher the weight.
- The time since the ball was last seen: the higher the time, the less the weight.
- The approximated deviation of the ball perception based on the distance: the higher the deviation, the less the weight.

Finally, the cluster that contains most balls is chosen. If multiple clusters have the same size, the one with the higher weighting is preferred. As a result, a common ball model, containing an approximated position and velocity, is calculated and provided as the representation `TeamBallModel`.

Among other things, the team ball model is currently used to make individual robots hesitate to start searching for the ball if they currently do not see it, but their teammates agree about its position.

5.3 Obstacle Modeling

Similarly, to the previously described ball tracking, all robots maintain two different models for the obstacles in their environment, one based solely on own observations and one incorporating information sent by teammates. Both implementations remained nearly unchanged for several years now.

5.3.1 Local Obstacle Model

To compute the local obstacle model, namely the `ObstacleModel`, the `ObstacleModelProvider` creates and maintains an Extended Kalman Filter for each obstacle. All obstacles are held in a list. The position of an obstacle is relative to the position of the robot on the field plane. Sources for measurements are visually recognized robots and sensed contacts with arms and feet. Arm and foot contacts are interpreted as an obstacle somewhere near the shoulder or foot. In the prediction step of the Extended Kalman Filter, the odometry offset since the last update is applied to all obstacle positions. The update step tries to find the best match for a given measurement by using Euclidean distance and updates that Kalman sample accordingly. If there is no suitable sample to match, a new one is created. Due to noise in images and motion, not every measurement might create an obstacle if no best match was found. To prevent false positive obstacles in the model, there is a minimum number of measurements within a fixed duration required to generate an obstacle. If an obstacle was not seen for some time, it is removed. This might also happen if an obstacle cannot be perceived although its estimated position is in the current field of view.

Measurements of visually recognizing robots also include a team identification based on the known jersey colors of both playing teams, which is sent by the GameController. If the vision system was not able to clearly determine a team identification, this value is set to *unknown*. As the team identification cannot be robustly detected, this information is quite noisy and thus becomes averaged over multiple measurements that are associated to a hypothesis. However, currently, the behavior does not include this particular information for decision-making.

5.3.2 Global Obstacle Model

As in some situations, a local model as base for team cooperation is not sufficient, an additional global obstacle model – the `TeamPlayersModel` – is computed by the `TeamPlayersLocator`, which makes use of communicated percepts.

5.3.2.1 Positions of Teammates

For the coordination of the own team, for instance for the role selection or for the execution of certain tactics such as set plays, it is important to know the current positions of all teammates. Computing these positions does not require special calculations such as filtering or clustering, because each robot sends its already filtered position to the teammates via team communication. This means that each robot is able to get an accurate assumption of all positions of its teammates just by listening to the team communication. Besides to the coordination of the team, the positions of the teammates are of particular importance for distinguishing whether perceived obstacles belong to own or opponent players.

5.3.2.2 Positions of Opponent Players

In contrast to the positions of the teammates, it is more difficult to estimate the positions of the opponent players. The opponent robots need to be recognized by vision to compute their positions. Compared to only using the local models, which are solely based on local measurements and may differ for different teammates, it is more difficult to get a consistent model among the complete team based on all local models. For a “global” model of opponent robot positions, which is intended here, it is necessary to merge the models of all teammates to get accurate

positions. The merging consists of two steps, namely clustering of the positions of all local models and reducing each cluster to a single position.

For the clustering of the individual positions, an ε-neighborhood is used, which means that all positions that have a smaller distance to each other than a given value are put into a cluster. It is important for the clustering that only positions that are located near a teammate are used.

After the clustering is finished, the positions inside each cluster that represents a single opponent robot have to be merged to a single position. For this reduction, the covariances of the measurements are merged using the *measurement update* step of a Kalman filter. The result of the whole merging procedure is a set of positions that represent the estimated positions of the opponent robots. In addition, the four goal posts are also regarded as opponent robots because they are obstacles that are not teammates.

5.4 Field Coverage

Since the introduction of the new ball in 2016, it is not possible to detect it across the whole field. In addition, there exist several configurations that obstruct ball detection from certain perspectives, such as a ball behind another robot. As a consequence, there are more periods of time during which no robot knows the current ball position. To overcome this problem, we carry out a cooperative ball search (as presented in [13]), which takes into account which parts of the field are actually visible for each robot and accordingly computes search areas for each individual robot. Due to the team-wide ball model `TeamBallModel`, a robot only needs to actively search for the ball if all team members lost knowledge of the ball position. Consequently, if a robot is searching, it can be sure that its team members do the same. Knowing which parts of the field are visible to the team members, searching the ball can happen dynamically and effective.

5.4.1 Local Field Coverage

To keep track of which parts of the field are visible to a robot, the field is divided into a very coarse grid of cells, each cell is a square of size $0.25\ m^2$. To determine which of the cells are currently visible, the current field of view is projected onto the field. Then, all cells whose center lies within the field of view are candidates for being marked as visible.

There may be other robots obstructing the view to certain parts of the field. Thus, depending on the point of view of the viewing robot, another robot may create a “shadow” on the field. No cell whose center lies within such a shadow is marked as visible. An example local field coverage is depicted in Fig. 5.3.

Having determined the set of visible cells, each of the cells gets a timestamp. These timestamps are later used to build the global field coverage model. They are also used to determine the cell, which has not been visited the longest, to generate the head motion for scanning the field while searching for the ball.

A special case is the ball’s movement as represented in the `BallModel`. If the ball touches a cell, its timestamp is set back to zero. This way, if the ball is lost while moving, the robots will search for it along its last known path first.

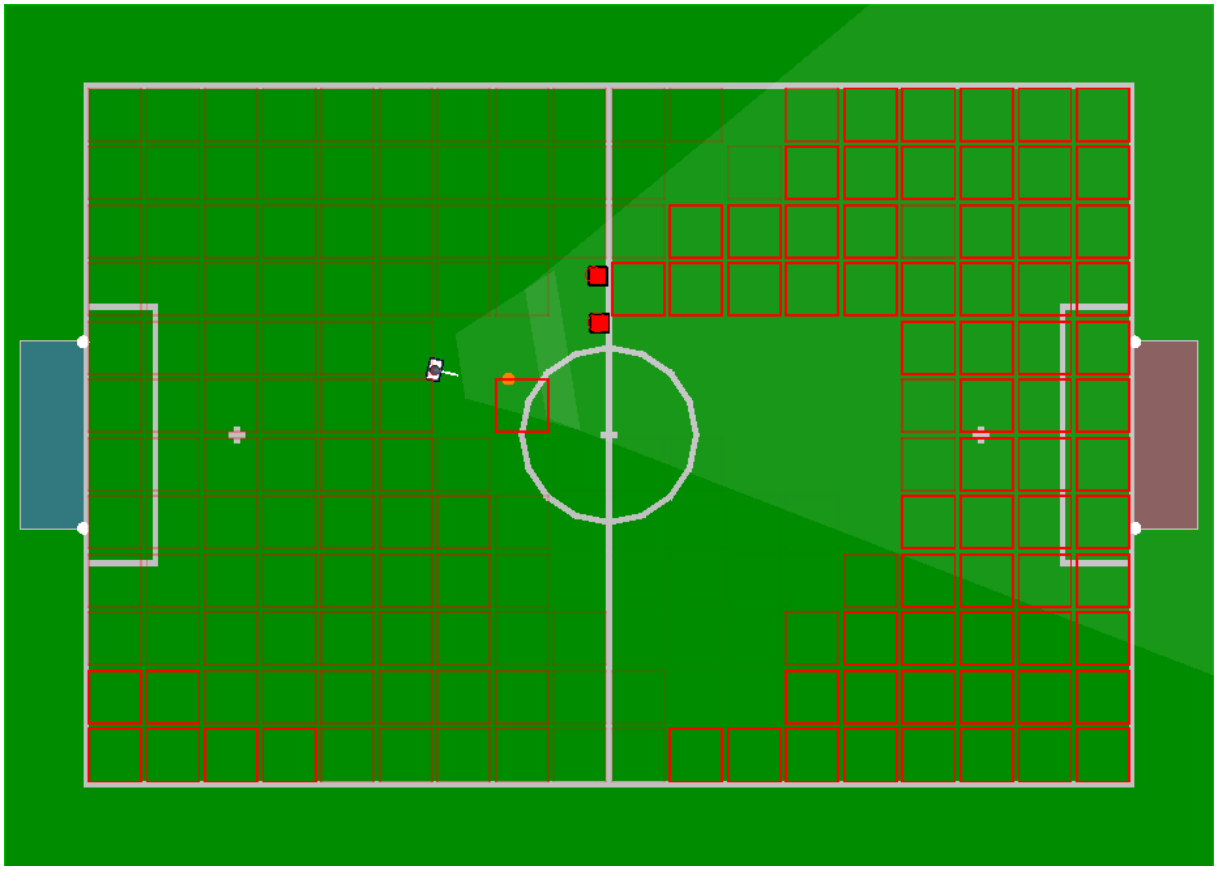


Figure 5.3: The local field coverage grid after walking around on the field. The more intense the red borders of the cells are, the less these cells are covered. Two opponent robots prevent cells on the top right of the field from being marked as visible.

5.4.2 Global Field Coverage

To make use of the field coverage grids of the other robots, each robot has to communicate its grid to its teammates. To do so, each robot maintains a global field coverage grid, which is incrementally updated in every team communication cycle and includes all information from its own local field coverage. This global grid looks roughly the same for all teammates. Hence, calculations based on the grid come to sufficiently similar results for all teammates.

The merging of the grids is done simply by looking at the time stamp of each cell in the existing global grid and a received local grid. If the received cell has a more recent timestamp, it is carried over to the global grid.

Given the current field dimensions, 4 byte time stamps, and a cell's edge length of $0.5m$, there are 4 bytes $*\frac{6m*9m}{(0.5m)^2} = 864$ bytes that have to be sent in every team communication cycle in addition to all other communication our robots do during gameplay. Since the resulting bandwidth requirement would be beyond the bandwidth cap set by the current rules, the time stamps for every grid cannot be sent. Sending only one line of the grid per message to reduce the size, is not possible anymore because it would result in an update rate of twelve seconds for the whole field what is not acceptable. Instead, only one timestamp is sent on the beginning of the message and for every grid is sent an offset of two bits. The two bits represents if the grid was scanned within the last second, the last five seconds, the last twenty seconds or not. This inaccuracy is acceptable because it results in a smaller size of only 2 bits $*\frac{6m*9m}{(0.5m)^2} + 4 * 8$ bits =

464 bits what is equal to 58 bytes.

5.5 Whistle Recognition

Our approach for detecting the referee whistle is based on the correlation between the robot's current audio input and several previously recorded reference whistles. The whistle recognition is implemented in the `WhistleRecognizer` module, providing the `Whistle` representation. In addition to this detection of an individual robot, we also implemented a joint decision of the team whether a whistle was detected or not.

5.5.1 Correlating Whistle Signals

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

First, we need a reference signal s_{ref}. The reference signal is also recorded by the robot, where we take a window length of N samples. We use the sampling frequency f_s to which the actual sampling frequency of the robot's audio hardware is downsampled.[1] The input signal is normalized by its maximum amplitude $s_{max} = \max_i |s_{\text{ref}}[i]|$, because we assume that the whistle is the loudest noise and it should always appear similarly strong in the input.

$$s'_{\text{ref}}[k] = \frac{s_{\text{ref}}[k]}{s_{max}} \tag{5.4}$$

The signal must be extended with zeros by length N, because the result of the correlation is twice the window length. In addition, correlating signals by Fourier-transforming them results in a cyclic correlation, which is avoided by the zero padding. This signal is transformed to the frequency domain by a Fast Fourier Transformation (we use the highly efficient FFTW3 implementation by [5]).

$$\mathcal{F}(s'_{\text{ref}}[n]) = \underline{S}_{\text{ref}}[k]. \tag{5.5}$$

The underline denotes complex values. This signal is stored as a reference and has not to be recalculated in every step. For the actual recorded signal s_{act}, we first check whether the signal is loud enough, i. e. s_{max} must at least reach the predefined volume threshold s_{min}. If it does, we use the same procedure as for the reference signal, i. e. normalization, zero padding, and transformation into the frequency domain to get $\underline{S}_{\text{act}}$. We make use of the fact that a correlation can be represented by a convolution using one signal in reverse order. The advantage of a convolution is that it can be calculated easily in the frequency domain by just multiplying the signals. As we do not want to store the signal in reverse order, we can transform the reversion into the frequency domain, too. Thus, we get

$$s'_{\text{ref}}[n] * s'_{\text{act}}[-n] = \underline{S}_{\text{ref}}[k] \cdot \underline{S}^*_{\text{act}}[k], \tag{5.6}$$

where $\underline{S}^*_{\text{act}}$ denotes the conjugate complex of the signal. As it does not matter which signal is conjugated, we store the reference signal as conjugate. Multiplying both signals gives us the result in the frequency domain and the inverse Fourier Transform is the correlation result s_{corr}

[1] Downsampling is only supported by integer factors. In regular games, audio data was recorded at 8 kHz and was not downsampled at all.

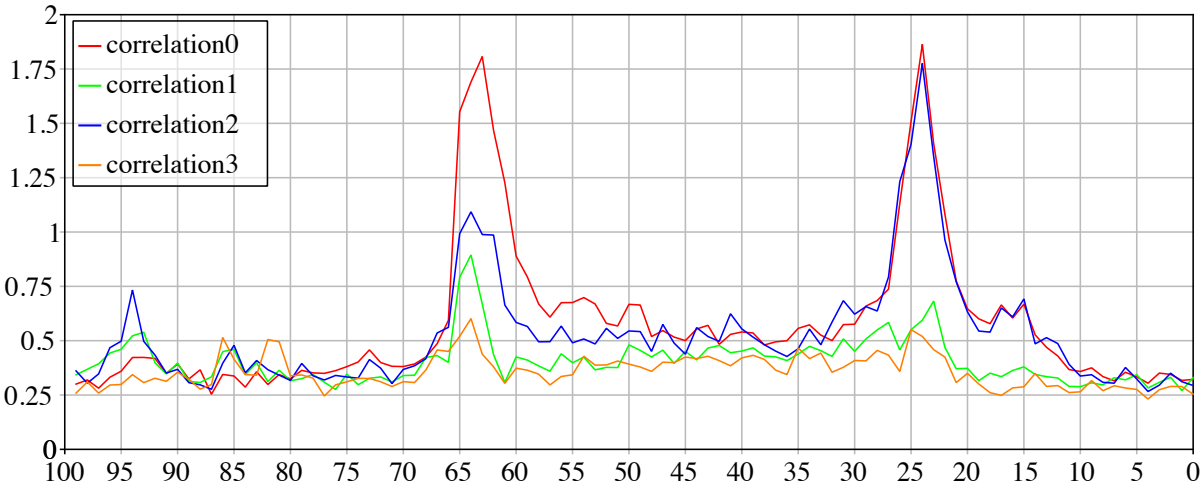


Figure 5.4: Correlation between the current signal and four pre-recorded reference whistle signals over 100 detection frames (128 ms frames that overlap by 50%, i. e. 64 ms steps). During this time, a whistle has been blown twice, but sounded slightly differently. The correlation results are divided by the detection threshold, i. e. values above 1 mean *detected*. The volume threshold was deactivated for this plot.

in the time domain, of which the maximum gives the strongest correlation (we actually use its square root).

The best result can be achieved if the reference signal is equal to the actual signal or the negative actual signal. Using this value, which is the autocorrelation value, and dividing the correlation signal s_{corr} by this factor, we always get a ratio of the best possible result. An example is depicted in Fig. 5.4, where correlation values were additionally divided by a detection threshold. As step by step formulation, we have the following implementation in discrete form:

$$
\begin{aligned}
s_{max} &= \max_i |s_{\text{act}}[i]| \\
s'_{\text{act}}[k] &= \frac{s_{\text{act}}[k]}{s_{max}} \\
s''_{\text{act}}[n] &= \begin{pmatrix} s'_{\text{act}}[k] & \mathbf{0}_N \end{pmatrix} \\
\underline{S}_{\text{act}}[k] &= \mathcal{F}\left(s''_{\text{act}}[n]\right) \\
s_{\text{corr}}[n] &= \mathcal{F}^{-1}\left(\underline{S}_{\text{act}}[k] \cdot \underline{S}^*_{\text{ref}}[k]\right) \\
\varphi_{cc} &= \begin{cases} \frac{\sqrt{\max(|s_{\text{corr}}[n]|)}}{\varphi_{ac\max}} & \text{if } s_{max} \geq s_{min} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

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

The correlation procedure is repeated for each audio channel (we used two) and every stored reference whistle. The computed `Whistle` representation contains the information about the signal that has the best average correlation for all channels. The module's required computing time scales linearly with the number of reference whistles. During RoboCup 2019, we used the reference signals of four different whistles. However, all computations are only executed during the game's *SET* state.

5.5.2 Team Consensus

To decide whether a whistle has been blown, we do not rely on the detection of each individual robot alone. Instead, we take all detections on the different audio channels of each robot into account. Each robot broadcasts on how many channels it is actually listening. This number can be lower than the maximum if microphones are broken or if the robot is currently playing back sound (then, it is not listening). The robots also broadcast the last correlation value that was above their detection threshold and when this happened. From this information, each robot can collect the detections within a certain time window and weight the correlation values by the number of channels each robot was listening on. The weighted average must then be above a second threshold for a whistle to be detected by the team. For this to work, the detection threshold for the individual robots must be lower than the threshold for the joint detection, so that robots also broadcast whistle detections that they are not sure about, but that can support stronger detections of their teammates.

Chapter 6

Behavior Control

The part of the B-Human system that performs the action selection is called *Behavior Control*. Its modules also run in the context of the thread *Cognition*. The behavior is composed of units called *skills* and *cards* (cf. Section 6.1) which run in the modules `BehaviorControl` and `TeamBehaviorControl`. Additionally, there are some auxiliary modules that provide representations and functionality to the main behavior or do post-processing, such as the `PathPlannerProvider` (cf. Section 6.3), the `CameraControlEngine` (cf. Section 6.4), and the `LEDHandler` (cf. Section 6.5).

This chapter begins with an overview of the concepts and software framework that are used to specify the behavior. The main part describes the contents of this system that constitute the soccer behavior used by B-Human at RoboCup 2019.

6.1 Skills and Cards

Decomposition and hierarchy are necessary to specify behaviors for complex tasks such as playing soccer. From 2013 to 2017, B-Human used a single hierarchy of CABSL [20] options to specify the behavior. This approach had some shortcomings: Adding or removing high level behaviors required modification of other options. This also meant that an option would often be in a different place than the conditions under which it would become active, which is not easy to maintain. Furthermore, some behaviors are simply not suitable to be modeled with finite state machines, need large calculations, or keep additional state. Some functionality was therefore outsourced to so-called *libraries*, which also spread closely related code across different places. Another minor point were static parameters that are normally defined explicitly

in order to be able to load them from a configuration file or modify them from the simulator (cf. Section 3.3.5). Since the complete behavior was a single module, parameters for all options had to be defined in a single place (actually, a representation `BehaviorParameters` existed for that purpose). However, it was rather impractical to have all those parameters in one large file, and thus seldom used.

In 2018, we already started to move away from a single hierarchy of CABSL options (cf. [24, p. 34]). The main problem with the *behavior options* in 2018 was that they could not be passed any parameters, making them useless for behaviors such as walking to a point or doing a specific kick. On the other hand, this anonymity was a desired property to achieve exchangeability of behavior components. We realized that it might not be a good idea to try to fit all behavior levels in the same formalization, but instead split the behavior into two layers: one that would decide *what* the robot should do, where options could easily be added or removed, and another one that would realize *how* the robot fulfills this request. This insight has been the foundation of our new behavior framework which we call the *Skills and Cards* system.

6.1.1 Skills

A *skill* is a behavior component that executes a task because a higher-level component requests it to do so. It can be passed parameters by its caller. Skills can call other skills to further decompose the behavior into sub-tasks, forming a hierarchy eventually filling the low-level commands to the motion system. Technically, skills consist of two code constructs: a *skill interface* and a *skill implementation*. The interface exports the signature by which it can be called by other skills. This also allows to have multiple implementations for the same skill without the callers needing to know which of them they should use.

Skill interfaces are declared in the `Skills` namespace in the file *Representations/BehaviorControl/Skills.h* using the `SKILL_INTERFACE` macro. The first argument to the macro is the name of the skill. Optional further arguments specify the parameters that the skill takes, starting with the type in parentheses followed by its name (similar to the definition of members in streamable classes, cf. Section 3.4.3). Default arguments are possible by specifying the default value in parentheses after the type, but as usual, can only be followed by other default arguments. This is an example:

```
namespace Skills
{
  SKILL_INTERFACE(GoToBallAndKick, (const Pose2f&) kickPose,
                                   (KickType) kickType,
                                   (float)(10000.f) length);
}
```

This macro actually declares a struct with the given skill name that functions as container for the parameters. Additionally, a class called `<name>Skill` is generated which is a proxy object that dispatches external calls to the skill implementation.

The definition of a skill interface is similar to that of a module (cf. Section 3.3.2), i. e. a base class is macro-generated which is inherited by the actual implementation. The following statements are available in a skill interface definition:

`IMPLEMENTS` declares that this class contains an implementation for a specific skill interface. It adds the virtual methods `reset`, `isDone`, `isAborted`, `preProcess` and `postProcess` and the pure virtual method `execute`. Each of them takes a constant reference to the skill interface, allowing a method to access the values that have been passed to the skill and overloading them in case multiple skills are implemented in the same class. `preProcess`

and `postProcess` are called every frame before respectively after the behavior is executed. This is useful e. g. to declare debug drawings. `reset` is called immediately before `execute` if the skill has not been called in the frame before. `isDone` and `isAborted` can report the status to the caller.

`CALLS` declares that this skill implementation calls another skill. It is available under the name `the<name>Skill` and can be called with the function call operator. The status can be queried with `isDone` and `isAborted`.

`REQUIRES` declares that this skill implementation accesses a representation that has to be provided before the behavior is run.

`USES` declares that this skill implementation accesses a representation which does not necessarily have to be updated before the behavior is run.

`MODIFIES` declares that this skill implementation modifies a representation that is provided by the module in which the skill is run. This is restricted to the representations that are part of the `SkillRegistry`.

`DEFINES_PARAMETERS` and `LOADS_PARAMETERS` have exactly the same meaning as in module definitions. However, `LOADS_PARAMETERS` prepends *BehaviorControl/* to the generated file name.

Skill implementations have to be published to the system using the `MAKE_SKILL_IMPLEMENTATION` macro which takes the name of the skill implementation class as an argument. The following is an example of a skill implementation:

```
SKILL_IMPLEMENTATION(GoToBallAndKickImpl,
{,
  IMPLEMENTS(GoToBallAndKick),
  CALLS(Stand),
  CALLS(WalkToTarget),
  REQUIRES(MotionInfo),
  REQUIRES(RobotPose),
  MODIFIES(MotionRequest),
});

class GoToBallAndKickImpl : public GoToBallAndKickImplBase
{
  void execute(const GoToBallAndKick& p) override
  {
    if(isReadyToKick(p.kickPose, p.kickType) ||
       (theMotionRequest.motion == MotionRequest::kick &&
        theMotionInfo.motion != MotionRequest::kick))
    {
      theMotionRequest.motion = MotionRequest::kick;
      theMotionRequest.kickRequest.kickMotionType = p.kickType;
    }
    else
      theWalkToTargetSkill(p.kickPose);
  }
};

MAKE_SKILL_IMPLEMENTATION(GoToBallAndKickImpl);
```

There can be multiple implementations for the same skill. To select which one should be used, the file *skills.cfg* exists. Skills that only have one implementation *must* not be mentioned in this file. It looks like this:

```
skillImplementations = [
  {skill = WalkToPoint; implementation = WalkToPointImpl;},
];
```

6.1.2 Cards

A *card* is a behavior component that associates actions with the conditions under which they should be executed. While skills execute requests, cards can decide for themselves whether they should be run. The following example illustrates this: The `GoToBallAndKick` skill takes a kick pose and a kick type as parameters. It does not decide whether and where to kick. In contrast, the `KickAtGoalCard` evaluates whether it is possible to do so and then calls the `GoToBallAndKick` skill with the appropriate parameters. Of course, the `GoToBallAndKick` skill is also called from other cards, such as passing to a teammate.

Technically, this conceptual difference to skills manifests in the following ways:

- Cards cannot take parameters and do not need separate interface declarations. Since a card contains only one behavior, its methods do not take a parameter as no overloading is necessary.
- Cards cannot modify output representations (they should call skills for that).
- Cards are invoked in a different way than skills (not via the `CALLS` macro).
- Cards do not have the `isDone` and `isAborted` methods.
- Cards have the `preconditions` and `postconditions` methods to influence when they are run. The preconditions specify the conditions under which the card can be entered, i. e. whether it should be run when it was not active in the previous frame. The postconditions specify the conditions under which the card should be left, i. e. whether it should not be run when it was active in the previous frame. By default, the postconditions are the negation of the preconditions.

Otherwise, the definition of a card is very similar to that of a skill:

```
CARD(KickAtGoalCard,
{,
  CALLS(GoToBallAndKick),
  REQUIRES(BallModel),
  REQUIRES(RobotPose),
  DEFINES_PARAMETERS(
  {,
    /** Distance of the ball to the goal to enable kicking. */
    (float)(3000.f) maxGoalDistance,
  }),
});

class KickAtGoalCard : public KickAtGoalCardBase
{
  bool preconditions() const override
  {
    return (theRobotPose * ballPosition
            - Vector2f(4500.f, 0.f)).norm() < maxGoalDistance;
  }

  bool postconditions() const override
  {
```

```
    return (theRobotPose * ballPosition
            - Vector2f(4500.f, 0.f)).norm() > maxGoalDistance + 1000.f;
  }

  void execute() override
  {
    const Pose2f kickPose = getKickPoseFromBallAndDirection(ballPosition,
                            (theRobotPose.inversePose * Vector2f(4500.f, 0.f)
                             - ballPosition).angle(), KickType::walkForwards);
    theGoToBallAndKickSkill(kickPose, KickType::walkForwards);
  }

  const Vector2f& ballPosition = theBallModel.estimate.position;
};

MAKE_CARD(KickAtGoalCard);
```

To build hierarchies of cards the system uses so-called *decks* and *dealers*. Decks are collections of cards that a dealer can choose from. The only dealer that is currently implemented is the `PriorityListDealer`. It simply takes the deck as a list sorted from highest to lowest priority and selects the highest priority runnable card, where *runnability* is determined based on the preconditions and postconditions. A deck can have the *sticky* property which leads to slightly different behavior in the dealer: Before going through the deck, it checks whether the postconditions of the card from the previous frame are still not fulfilled, and if so, it is dealt again. This means that the previously selected card can continue to run even if there are higher priority cards that can also run.

6.1.3 CABSL

For skills as well as cards, there are specialized dialects of CABSL [20] to specify their internal behavior. Since modularity is already achieved through decomposition into skills and cards, calling options from other options is not supported in both of them. The macros are defined in *Tools/BehaviorControl/Framework/Skill/CabslSkill.h* and *Tools/BehaviorControl/Framework/Card/CabslCard.h*, respectively. Only one CABSL header can be included per compilation unit, and it should be the last included file.

When defining a CABSL option in a skill, the argument of the `option` macro must match the name of the skill that it implements. Therefore, within a skill implementation class one can decide individually per skill whether it shall be CABSL option or not. Within the option, parameters can be accessed via the variable `p`, as it is conventionally called in skills. Using CABSL for a skill automatically overrides the `execute`, `reset`, `isDone` and `isAborted` methods. Note that the body of an `option` is just a method, so if some calculations need to be done before the execution of the state machine, code can be added after the opening brace of the option block.

In a card, the `option` does not even take an argument because it is already identified by the name of the card and there can only be one option per card. Target states and aborted states do not exist in cards, either. Only the `execute` method is overridden, the `preconditions` and `postconditions` must be specified additionally as usual.

6.2 Behavior Used at RoboCup 2019

The complete cards and skills system is instantiated twice in the B-Human software: once for the individual robot behavior in the module `BehaviorControl`, and once for the team be-

havior in the module TeamBehaviorControl. All macros, types and files thus exist with the prefix TEAM_, Team or team, too. Technically, they differ in the representations that skills have writing access to: The individual behavior can write to the representations MotionRequest, ArmMotionRequest, HeadMotionRequest, BehaviorStatus and TeamTalk, while the team behavior can only set the TeamBehaviorStatus.

The remainder of this section starts with a description of our team behavior and the upper levels of the individual behavior that handle situations in which the robot is not playing. Then, the different roles as used during normal play are presented, followed by the special handling in free kicks, kick-offs and the ball search. This section concludes with our penalty shoot-out behavior.

6.2.1 Team Behavior

The team behavior determines which role in the team a robot should take in order to contribute to the team winning the game. On the topmost level, our 2019 behavior distinguishes between three roles:

1. The striker plays the ball.
2. The goalkeeper stays mostly in the own penalty area to block long distance shots, intercepts the ball if it would otherwise reach the own goal, and clears the ball if it is close to the penalty area.
3. Supporters take positions on the field to help the team. Supporters get an additional *index* that indicates how far back or forward they are in relation to their teammates. Most supporter cards use the index in their pre-/postconditions, e. g. the rearmost supporter in the team should usually be defender, while the foremost robot can take a position upfield.

A role thus comprises the flags playBall, isGoalkeeper and the number supporterIndex (which should be -1 if one of the flags is set). Additionally, the number numOfActiveSupporters is calculated, which enables reversing the index.

Different team cards can implement different role assignment methods. The PlayingTeamCard assigns the striker role according to the time it takes to reach the ball, while the KickoffCard that runs during the *ready* state assigns the striker role to the robot closest to the kick-off position. In both cases, the supporter indices are calculated by sorting all supporters by their global x-coordinate, where the comparison function adds a stability offset to favor the previous relation.

6.2.2 Superordinate Individual Behavior

At the topmost level, the module BehaviorControl is a state machine to handle some convenience and safety mechanisms and penalties. The following states exist:

Inactive: Initially, the robot is in a “dead” state with all joints off. Pressing the chest button once will transition it to the gettingUp state.

Getting Up: The robot interpolates to a stand motion and transitions to the playing or penalized state. This state has nothing to do with robots that are fallen.

Playing: This is the main state of the behavior. The configured root card is called.

Penalized: The robot stands with stretched knees until its penalty is removed.

Sitting Down: Pressing the chest button thrice in the `penalized`, `playing` and `gettingUp` states lets the robot sit down and transitions to the `inactive` state.

Low Battery: This state is entered from the `penalized` and `playing` states if the code is compiled with assertions and the battery level is below 1%. The robot sits down and transitions to `inactive`.

Broken Camera: This state is entered from the `penalized` and `playing` states if one of the cameras has stopped working. The robot sits down and transitions to `inactive`.

The root card is normally the `GameControlCard`. This card does nothing more than dealing from a deck in its configuration file *gameControlCard.cfg*. There are cards for the game states *initial*, *finished* and *set*, which basically request the robot to stand. The `FallenCard` becomes active when the robot is fallen and requests a get-up motion (cf. Section 8.6). The penalty shoot-out cards (cf. Section 6.2.9) are also included directly in this deck (just below the `SetCard`). Finally, if the game state is *ready* or *playing*, the robot is upright and no penalty shoot-out is ongoing, the `GameplayCard` becomes active.

The `GameplayCard` represents the hierarchy level on which the robot can actually *choose* its behavior, i. e. do more than just standing around. It is also a dealer, but has five decks: `ownKickoff` and `opponentKickoff` for the *ready* state, `ownFreeKick` and `opponentFreeKick` for free kicks, and `normalPlay` for all other situations.

6.2.3 Striker

The *striker* is the robot which plays the ball, as long as the goalkeeper does not want to do this. All striker cards check the `playBall` flag of the `TeamBehaviorStatus` to check whether they should run. A general pattern is that decisions are made from the ball's point of view in polar coordinates. Calculating from the ball has the advantage that the decision does not significantly change as the robot approaches the ball. Using polar coordinates is motivated by the fact that a kick decision is easy to represent by a direction and the kick range. Additionally, most cards use the expected end position of the ball if it is moving, in order to avoid walking to the ball on an arc.

Sector Wheels

The utility class `SectorWheel` handles the common task of calculating which sectors (i. e. angular intervals) are free to play the ball to, are blocked by opponents, are part of the goal etc. Each sector has a distance and a type. All sectors form a complete partition of the angle space from $-\pi$ to π. A `SectorWheel` is initialized with the ball position which becomes the center of the wheel. Then, sectors can be added by passing either positions including their radius or angular intervals directly. The `SectorWheel` calculates occlusion between existing sectors and the new one based on their distances, such that the final wheel describes exactly the "visible" objects from the center's view.

Determining Goal Post Tangents

When the striker wants to move the ball towards the goal, it is important to correctly calculate the angular range in which the ball can enter the goal. Instead of simply taking the angles of

the goal post centers relative to the ball, the goal post tangent angles are calculated, i. e. the angles at which the ball's edge touches the inner goal post edges in one point. As the goal post tangent is perpendicular to the line between the goal post's center and the ball's hypothetical position next to the goal post, having radius r_{post} and r_{ball} respectively, the angular offset to the angle between goal post center and ball is

$$\Delta\theta = \arcsin\left(\frac{r_{\text{post}} + r_{\text{ball}}}{|\mathbf{x}_{\text{post}} - \mathbf{x}_{\text{ball}}|}\right).$$

Situations in which, due to world model errors, the ball position is already inside the goal post, are handled separately anyway (as the $\arcsin$ argument would be larger than 1).

In the following, all cards that constitute the striker behavior during normal play are described in the order of their priority:

HandleBallAtOwnGoalPostCard: When the ball is near one of the own goal posts, it is not possible to select arbitrary kick directions because the robot can not stand inside the goal posts. Furthermore, it is rather urgent to clear the ball in those situations, such that quickness is a major concern. Therefore, as soon as the ball is within a 50 cm circle around the goal posts, this card is activated.

First, the target sector is calculated, which is between the global x-axis and the outer tangent of the goal post which is being avoided. Then, a `SectorWheel` with obstacles is created, which serves as a template for the following calculations. The remaining part of the card determines the fastest kick that moves the ball in the target sector and does not need to stand in the goal post. Only the forward, turn and sidewards in-walk kicks are considered, because `KickEngine` kicks would take too long and do not provide an advantage in these situations. On the other hand, the three kicks make it possible to clear the ball from anywhere around a goal post. Thus, for each kick, the `SectorWheel` is completed by erasing the sector that would require invalid kick poses. This sector can be calculated from an approximated convex robot shape, the angle of the ball relative to the goal post and the offset of the robot to the ball that is necessary to execute that kick. For the remaining sectors, the kick pose that can be reached first is calculated. Among all valid kicks, the fastest is selected and finally executed.

DuelCard: A duel is a situation in which an opponent is close and ball possession has to be gained or defended. In fact, any obstacle can trigger a duel regardless of its team affiliation. To enter a duel, an obstacle must be in a certain area around the robot, the ball must be close and the robot not too close to one of the penalty areas. The duel is left if either the ball is far away or there are no obstacles in a range slightly larger than the one needed to activate it.

The duel behavior itself is based on the evaluation of a rating function for a number of potential kicks and kick directions. The rating comprises a static component that depends only on the position on the field and a dynamic component based on the distances to obstacles and teammates (cf. Fig. 6.1). The search space is an angular range for the robot's rotation during the kick, centered around its current rotation. The size of the range varies between $0°$ and $30°$ depending on the current distance to the ball (the closer the ball already is, the less change in direction is desired). For all possible in-walk kicks, kick poses in the range are sampled and evaluated by the expected ball position after the kick. The best combination of kick pose and kick according to the rating is then executed.

KickAtGoalCard: The primary goal of the striker is to score whenever possible. This card tries to achieve this by evaluating which parts of the opponent's goal are not covered by

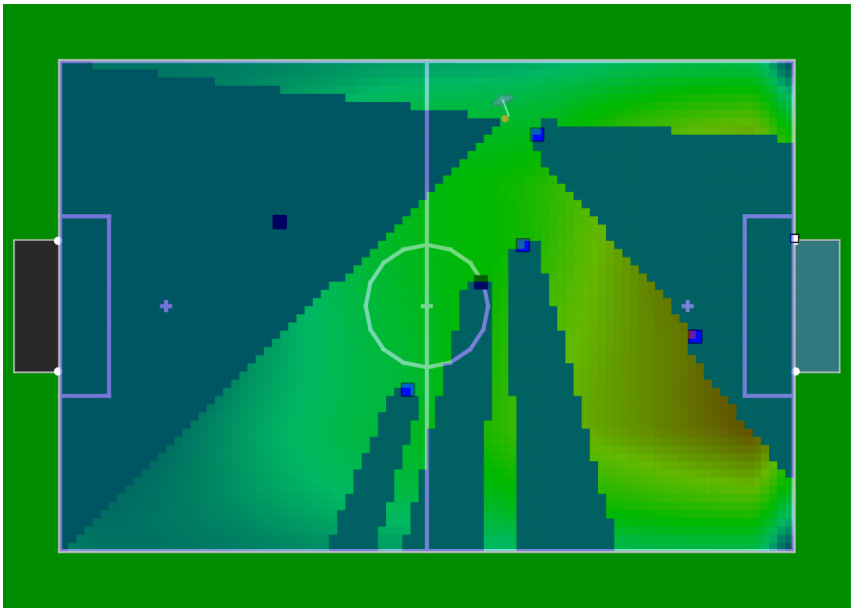


Figure 6.1: The duel rating function evaluated on a grid over the field. Blue means a low rating, red is a high rating.

obstacles and whether there are kicks available that have the required range to make the ball cross the goal line. Due to the kick-off goal rule (cf. [2, p. 27]), the card must not become active as long as the ball has not left the center circle since kick-off.

First of all, the card checks whether the ball is already ahead of the front edge of the goal posts, which has two sub-cases: If the ball is next to the goal or there is no space between the goal post tangents, it is impossible to score directly from there, so the card indicates via its pre-/postconditions that it should not be executed. If the ball is between the goal posts (i. e. the ball is already on the goal line), a goal can be scored in any direction between the goal post tangents (clamped to $\pm 90°$). In the latter case, a kick pose is calculated for each available kick and among them, the best one is selected.

In the default case, the ball is not on the ground line, such that both goal post tangents are in the $\pm 90°$ range. An important idea for the following steps is a technique we call *obstacle culling*, which is motivated by some observations:

1. Kicking the ball over large distances is imprecise. If it is aimed at some gap between an obstacle and a goal post, it is unlikely to hit that gap exactly. On the contrary, if it is aimed at an obstacle, it might as well be deflected in our favor, perhaps entering the goal.
2. When still far away from the goal, it is already an improvement to get the ball into the opponent's penalty area.
3. It is often a very strong move to make the opponent's goalkeeper dive for the ball.

Therefore, obstacles that are close to the opponent's goal do not necessarily need to be considered if the ball is still far away. This behavior is achieved by defining a line that is parallel to the ground line, beyond which obstacles *may* be removed if they prevent a sufficiently large kick sector from being found. The x-coordinate of the cull line is interpolated between the opponent's penalty mark and the ground line depending on the x-coordinate of the ball.

To prepare the calculation of the target kick sector, all known obstacles are transformed to their angular range from the ball's perspective, together with their distance from the ball and their global x-coordinate (they are already discarded at this stage if they do not intersect with the goal range). Then, `SectorWheel`s with the remaining obstacles are calculated iteratively, each time removing the obstacle with the largest global x-coordinate, until either a sufficiently large sector has been found or there are no more obstacles behind the cull line.

The result of this process is a list of possible kick sectors (which might be empty; kicking at the goal is not preferred then). They are sorted by their angular size (with a stability bonus for the sector which contains the previous kick direction) and traversed until either the size becomes too small to be reasonably certain that the kick will be inside the sector's bounds or a suitable kick with sufficient range could be found for a sector.

For each sector, every kick must be evaluated. The range of a kick can further reduce the sector's size, which makes a difference especially when the ball is close to a sideline. After incorporating this, a safety margin is subtracted from the sector boundaries, representing uncertainty in the kick direction. From the resulting angular range (which might have been contracted to a point now) a kick pose is calculated. All possible kicks for a sector are compared by the time it takes to execute them and the fastest one is finally chosen. Only if no kick has the required range for a sector, the next (smaller) sector is examined.

PassUpfieldCard: If no direct shot at the goal is possible for the current striker, there may be supporting teammates closer to the goal which could receive passes. Our robots do not try to kick directly at the receiving robot, but instead target a region between the robot and the goal, such that, upon reaching the passed ball, the receiver can consequently play towards the opponent's goal without needing to walk around the ball.

The pass receiver has to fulfill a number of criteria:

1. Its global x-coordinate must be ahead of the ball (meaning that it is "upfield").
2. In an intermediate step, it is determined whether the pass should arrive left or right of the receiver (left and right are well-defined after the previous check). The calculation uses the current rotation of the receiver candidate and whether its position is left or right of the line between the ball and the goal. Based on this decision, a pass direction (aiming a bit to the previously determined side of the receiver) and the hypothetical ball position after the pass are calculated.
3. Its position must be closer to the ball after the pass than before.
4. There must not be any obstacles in the sector around the planned ball trajectory.

Each of the criteria includes some stability offsets depending on the previous decision in order to keep a pass plan even under uncertainty. If a teammate is found that the ball can be passed to, a matching kick pose and kick type are selected (currently only forward kicks from the `KickEngine` are considered) and executed using the `GoToBallAndKick` skill. Additionally, the player number of the targeted robot and the expected ball position after the pass are communicated to the team.

DribbleToGoalCard: If it is neither possible to score a goal nor there is a teammate to pass to, the robot has no choice but to get the ball and itself to a better position by dribbling. This card has no further conditions than being executed by the ball-playing robot and can always run. Its main task is therefore to determine the direction in which the ball should be dribbled.

To that end, an angular interval is calculated from which the final direction is chosen to require minimal rotation from the robot's current pose. The interval calculation is based on the idea that while being far from the opponent's goal, it is not that important to dribble towards the goal, but merely in the opponent's direction. The closer the ball comes to the goal, the more the range narrows down to the goal post tangent angles. This introduces a natural way of playing along the outer parts of the field, depending on the initial rotation of the dribbling robot. In order to be useful in the opponent's corners, the goal post radius is artificially increased depending on the global y-coordinate of the ball. This rotates the goal post tangent away from the ground line to reduce the danger of moving the ball out of the field (which nevertheless occurs when the robot hits the ball with the wrong foot before the actual dribble step, but this is rather a skill issue) and get a better score angle. The resulting interval has a minimum size which is enforced by extending it to the side of the far goal post (i. e. if the ball is in the right half of the field, the left angular boundary is rotated further counterclockwise to yield the minimum size). There is special handling when the ball is close to the goal to avoid dribbling it out of the goal when it is already on the goal line and to keep stable decisions when it is close to a goal post. However, it should be noted that imprecise localization near the goal posts is always bad, as only few centimeters make the difference between playing the ball in opposite directions.

Figure 6.2 shows some sample ball trajectories that could result from this card being played. The `DribbleToGoalCard` does not do any obstacle avoidance. As soon as an obstacle is close, the `DuelCard` becomes active anyway.

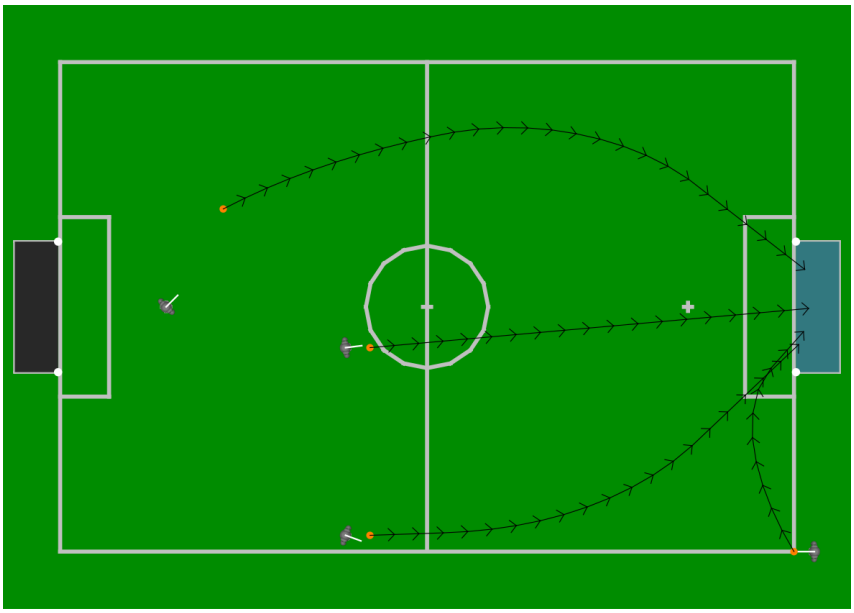


Figure 6.2: Trajectories along which the ball can move by the `DribbleToGoalCard`.

6.2.4 Goalkeeper

The goalkeeper behavior consists of five cards, which are described here ordered by priority:

KeeperCatchBallCard: The most important task for a goalkeeper is to prevent the ball from entering the own goal. Since only a very small fraction of the goal can be covered by standing and it is not allowed to stand in a wide stance for more than 5 seconds as per the rule book [2, p. 29], the robot has to deliberately execute motions when a ball is

moving towards the own goal in danger of entering it. This is the case when the ball is rolling towards the goal and such that it will cross the robot's y-axis in the near future. To avoid spurious actions due to a wrong ball state estimate, the condition to intercept the ball must hold continuously for 100 ms. During this time, the arms are already raised to accelerate the following potential dive. The robot has the choice between three actions: Jumping left or right, such that the robot lies on the ground with an outstretched arm, and sitting down with the legs spread. The decision is made according to the predicted y-coordinate where the ball will pass the goalkeeper. Additionally, if a jump would hit a goalpost, it is inhibited.

KeeperSearchForBallCard: The goalkeeper has its own ball search behavior that becomes active if the whole team has not seen the ball in a while. The goalkeeper wants to reach a position on its own goal line to get the best possible overview of its own half. To get there, however, care must be taken to avoid scoring own goals when the ball lies behind the goalkeeper. Therefore, the goalkeeper does not walk backwards, but instead, makes some steps away from the goal first, then turns around to its target on the goal line (possibly finding the ball and returning to normal play), and finally walks there. This state can only be left after the ball has been seen again.

KeeperClearBallCard: In some cases, the goalkeeper itself must play the ball instead of the striker. For this to happen, in addition to expecting to reach the ball before the current striker, the goalkeeper and the ball must be close to the penalty area. If necessary, this card applies the same special handling as the `HandleBallAtOwnGoalPostCard` does. Otherwise, the kick that is selected from the `KickPoseProvider` [24, p. 36] will be executed (which is the only remaining use of this module, thus it is not described in detail here).

ReturnToGoalCard: If the goalkeeper is currently far from its penalty box (e. g. after a penalty), it wants to get back there as quickly as possible.

GuardGoalCard: The default behavior of the goalkeeper is to cover the goal, i. e. to stand at a position where the part of the goal that is covered and the potential for intercepting the ball is at its maximum. This is calculated by the `GoaliePoseProvider` in the representation `GoaliePose`. It is always on the bisector between the lines from the ball to the goal posts. The x-coordinate is restricted to be at least as much behind the front penalty area line to be able to detect it in order to improve the localization. The positioning is illustrated in Fig. 6.3.

6.2.5 Supporter

Supporters help the team by positioning in strategically advantageous locations. Their number in the team can range from none (if only a goalkeeper and a striker are present) to four (if the goalkeeper handles the ball and all players are present). The following cards are available:

BlockAfterBallLostCard: It is a common situation that a robot loses a duel against an opponent in the way that the opponent kicks the ball past it. If the robot loses the striker role due to this and turns into a supporter, i. e. there is another robot in its team that takes over the ball handling, it tries to prevent the opponent from reaching the ball, giving its teammate more time to play the ball. This happens as long as there is an opponent immediately in front of it and no other opponent is known that is closer to the ball. Blocking is done by moving immediately in front of the opponent, facing it, with the back towards

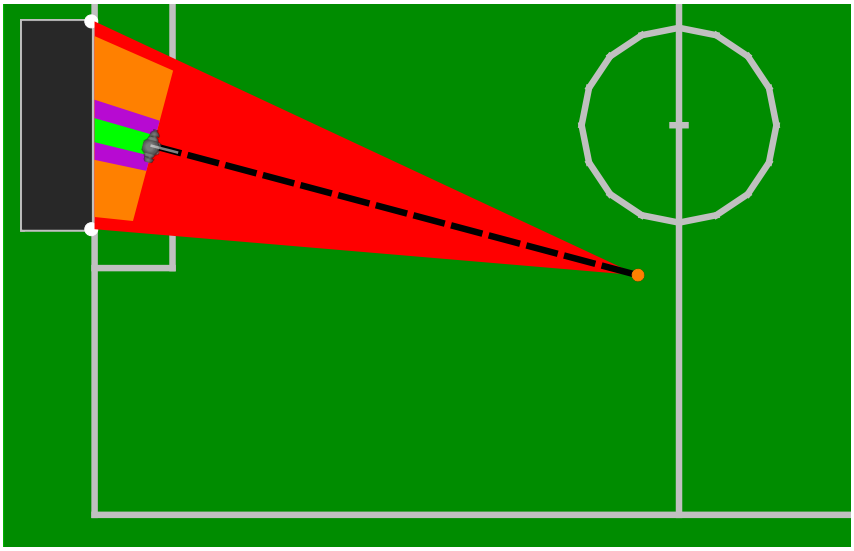


Figure 6.3: The goalkeeper position is on the bisector between the lines from the ball to the goal posts. The areas covered by standing / sitting down with the legs spread / jumping are colored in green / violet / orange.

the ball. The robot attempts to mirror the movements of the opponent and still keep the same distance to the ball.

DefendGoalCard: Players that execute this card are also called *defenders*. They assist the goalkeeper in covering the own goal to block distant shots at the goal. At least the rearmost support wants to execute this card, while another player (the second to rearmost) may join if it is already clearly in the own half or there is at least one more supporter in the offense. The defender has different modes about which it decides individually based on the position of the other defender.

The first decision it has to make whether it is left or right (in relation to the goalkeeper-ball line). A single defender decides for the same side as before as long as the ball is not too far on the other side. In the case of two defenders, both of them have to be on different sides, so they compute their (signed) distance to the goalkeeper-ball line and check which of them is more left than the other. If this difference in distances is significant, the decision is accepted, otherwise, the y-coordinate of the robots on the field is checked. If they are still not significantly different, the previous decision is kept. Note that the side used for the position calculation can differ from this if the ball is near one of the own corners of the field where standing next to the goalkeeper-ball line would be outside the field.

Next, each robot decides whether it is forward or back. Being forward means having a larger distance to the own penalty area than standing just in front of it. Only one defender at a time can be forward, thus, this decision is always in view of two players. For both players, the current distance from the center of the own goal (the *radius*) is calculated. If exactly one of them is larger than a threshold, that player is set to be forward and the other as back. If, on the other hand, both are below this threshold and the ball is not clearly in the opponent half, both stay back. In the remaining cases, the robots' orientations are checked if one of them is facing the own goal and the other the opponent goal. The one facing the opponent goal is then selected as forward. Otherwise, the player on the side where the ball is becomes forward.

While the left / right decision determines which "gap" between the goalkeeper and goal

post the defender should fill, the forward / back decision influences the distance from the own goal at which the robot selects its position. A back defender stands on a circle that has its origin in the center of the own goal and just completely includes the penalty area. The circle is interrupted by a straight line segment in front of the own penalty area. In contrast, the radius of a front defender goes almost to the center circle. There is a special case when there is only one defender and no goalkeeper: The defender will then stand on the line that would otherwise be covered by the goalkeeper.

Since the goalkeeper has to see the ball in order to react and eventually intercept it, it is important that its field of view is not unnecessarily occluded by the defenders. Therefore, they calculate whether their sides are very close to the goalkeeper-ball line and if so, take the respective arm behind their back.

WaitUpfieldCard: The purpose of this card is twofold: Firstly, it provides a passing opportunity to the `PassUpfieldCard` running on the striker. Secondly, it can quickly reach the ball after a failed goal shot. To select this card, a robot must be the foremost supporter in the team and the ball must be sufficiently far back on the field. The robot decides for a side according to whether it is currently left or right of the ball-goal-line, in order not to cross it. Then, a position is selected on a diagonal line starting at a point on the center line close to the sideline and ending at a point near the penalty area corner. The x-coordinate of the position is chosen to keep the distance to the ball within a reasonable range for passes, which also determines the y-coordinate via the aforementioned line. The orientation of the target pose makes a compromise between keeping the ball in the possible field of view and being oriented towards the opponent's goal.

WalkNextToStrikerCard: The default behavior of a supporter if none of the above is applicable is to stay close to the striker to be available when the striker fails. Similar to the `DefendGoalCard`, this card can be played by up to two robots simultaneously and there is a left and a right mode. If there are two players of this card, the side is determined by the relative position of the robots to each other. Otherwise, a single robot wants to stay on the side where it currently is, as long as it is not too far from the center of the field. The closer the ball gets to the opponent's goal, the more the robot decides for the inner side. The position is selected in a fixed distance of the ball, while the angle to it varies to avoid getting too close to the sideline.

WalkNextToKeeperCard: This card is different to the others in that it is not executed by a supporter in the team behavior sense, but instead is active on the striker robot when the goalkeeper plays the ball. There are three modes: blocking, walking left and walking right. Blocking is selected when the ball is inside or close to the own penalty area. The robot then walks to a position outside in an attempt to prevent opponents from reaching the ball before the goalkeeper can clear it. Otherwise, the robot just walks with an offset left or right to the goalkeeper, depending on the relative position when this card became active.

6.2.6 Free Kicks

Free kicks are a kind of set play that occur during the *playing* state of the game in response to either the ball leaving the field of play or a forbidden action. The team that is awarded a free kick has the exclusive right to the next ball contact for up to 30 s while the other team must clear the area around the ball. The behavior for free kicks is split into two categories: one for the behavior during own free kicks (also called *offensive*) and one for the behavior during a free kick for the opponent team (called *defensive*). Both situations have their own deck in the `GameplayCard`.

6.2.6.1 Offensive

Since the rule book [2] defines four causes for a free kick, the offensive free kick behavior is also divided into four parts. With the exception of the goal free kick each part has a card for a kicking robot and a card for a supporting robot. Before any of the special behaviors is considered, the striker evaluates the chances for a direct goal kick and executes it if possible by placing the `KickAtGoalCard` (cf. Section 6.2.3) first in this deck. If this is not the case, the striker will switch to one of the following behaviors (prompting a supporter to execute the corresponding supporting card):

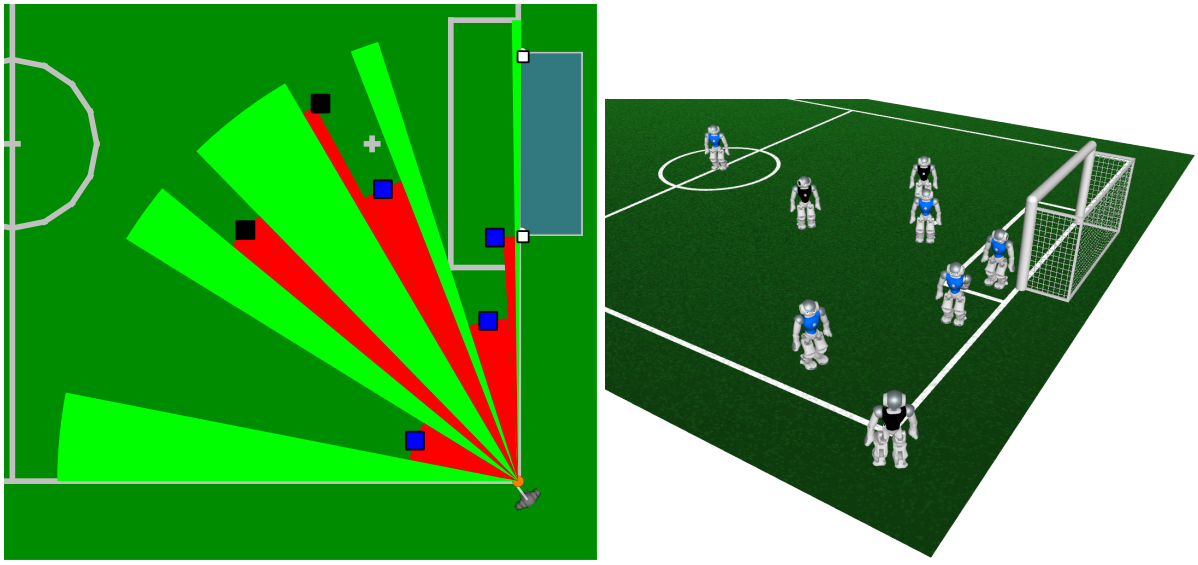
CornerKickToOwnRobotCard and WaitForCornerBallCard: If the opposing team plays the ball over its ground line, an offensive corner kick occurs, activating the `CornerKickToOwnRobotCard` on the striker. In fact, this behavior is also used for all other free kicks if they occur close to one of the opponent's corners. The striker walks to the corner of the field and positions itself behind the ball in the direction of the field. Meanwhile, two supporters running the `WaitForCornerBallCard` move to waiting positions in front of the goal and near the center circle. Standing behind the ball, the striker determines all possible directions to which it can kick the ball without hitting one of the opponent robots using a `SectorWheel` centered around the position of the ball.

In the first step, the sideline and the goal post closer to the ball are set as boundaries of the `SectorWheel` and the area outside the field is declared as unavailable. The next step is to add all known obstacles on the field. Figure 6.4a visualizes the resulting sectors. A minimum sector width is defined such that only sectors wider than it are considered as kick targets. In each frame, the angles from the two sectors closest to the goal are calculated and then communicated to the supporters. Using this information, they position themselves behind the kick angle from the viewpoint of the opponent's goal and maintain a safety distance. They also keep the same distance to the ball as there is between the ball and the goalpost.

The kick is executed as soon as the selected angles no longer change significantly. This is determined by comparing the sector in which the current best kick angle is located with the sector in which the first kick angle candidate was located in the previous frame. If the intersection of these sectors is as wide as the minimum sector width, the kick angle is considered stable. Should this not happen until `15 s` before the free kick expires, the kick will still be executed. As long as no kick angles can be determined at all, the robot waits until time runs out and then resumes to normal play.

KickInCard and WaitForKickInBallCard: If the ball is played over the sideline by the opposing team, there is a kick-in at the point where the ball left the field. While up to two robots take positions in the center and closer to the opposite side line, the striker decides whether to shoot the ball directly at the goal or to play a robot close to the goal, so that it has the possibility to shoot at the opponent's goal or dribble there.

GoalFreeKickCard and WaitForGoalFreeKickCard: If an opponent touches the ball before it goes out over the own ground line, an offensive goal kick occurs. The striker then executes the `GoalFreeKickCard` which walks to the ball and decides whether the goal kick is executed immediately or after a few seconds. This decision depends on the up to two teammates executing the `WaitForGoalFreeKickCard`, which move to fixed positions. If this target position can be reached before the time for the goal kick has elapsed, then the striker waits. If the target position cannot be reached in time anyway, the striker plays the ball forward immediately. Here the striker decides in which direction the ball comes



(a) The sectors used to determine the kick direction. Green sectors are considered as targets (provided they are wide enough), red sectors are occupied by an obstacle.

(b) A 3-D rendering of the left scene

Figure 6.4: Simulated scene of an offensive corner kick

furthest forward based on its teammates and the opponent's players on the field. The position of the supporting teammate also plays a role. The teammate should be able to play the ball after the goal kick in the direction of the opponent's goal.

PushingFreeKickCard: An offensive pushing free kick is initiated when an opponent pushes an own robot while being near the ball, in addition to the offending robot being removed. The robot taking the free kick decides whether to shoot the ball directly at the goal or near it by calculating a `SectorWheel`. The ball is shot directly at the goal if there is a possibility to score a goal. If no direct goal can be scored, a position close to the goal is aimed for so that a teammate or the kicking robot itself has the possibility to kick the ball into the opponent's goal from there.

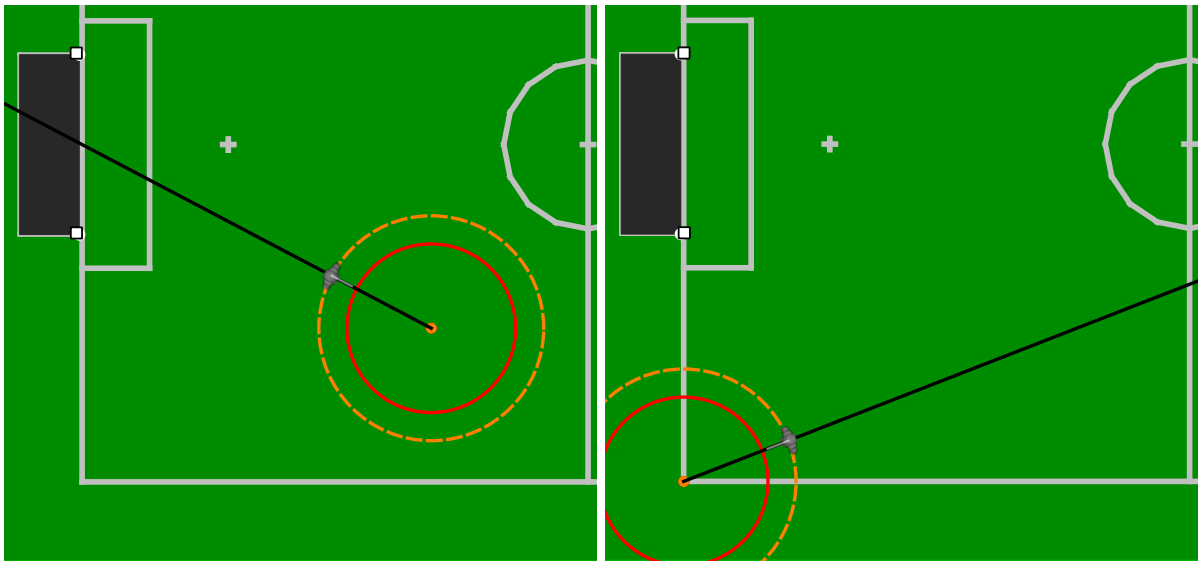
6.2.6.2 Defensive

The behavior during an opponent's free kick is always aimed at preventing a goal and does so in different ways depending on the possibilities available. All ball-playing cards are removed from this deck, however, there are three special cards in addition to the usual supporting behavior during normal play:

BuildWallCard: One robot tries to form a wall between the own goal and the ball. This is subject to the restriction that according to the rule book [2, p. 16], a minimum distance of 75 cm to the ball must not be violated. The position is chosen just outside the clear area around the ball in an angle such that the own goal is covered as wide as possible (cf. Fig. 6.5a). Keeping a robot close to the ball has the additional purpose to regain ball possession quickly when the free kick is not executed properly.

A special case is when the ball is in one of the own corners, such as in an opponent's corner kick. In this case, the robot does not stand between the ball and its own goal but

walks to a position close to the sideline. The goal is then still covered by the defenders and the goalkeeper. This is depicted in Fig. 6.5b.



(a) A free kick in field (b) A corner kick

Figure 6.5: Examples of robots positioning as a wall. The red circle marks the area that must be cleared by the defending team. The dashed orange circle is at the distance and the black line shows the direction from the ball at which the wall is formed.

BlockRobotFreeKickCard: If a free kick takes place directly in front of the own goal such that a wall between the ball and the own goal is no longer possible, an attempt is made to delay or prevent the opponent from taking the free kick. This is done by walking in front of the opponent which is nearest to the ball. This forces the opponent to walk around the blocking robot (which continuously adjusts its position) or even makes it lose track of the ball. If the opponent is already closer to the ball, the free kick can not be defended by any field player anymore.

MarkUpRobotCard: If there are five robots or more in the defending team, one of them considers marking an opponent. Candidates are opponents that are closer to the own goal than the ball and probably not taking the free kick. The goal is to make the robot unattractive as pass receiver for the opponent or to gain ball possession if a pass is done nevertheless.

6.2.7 Kick-Off

The kick-off is yet another set play. In contrast to free kicks, the kick-off involves a separate setup phase, which is technically realized through the game states *ready* and *set*. Since in the *set* state it is forbidden for the robots to locomote, it is handled in the `SetCard` as mentioned in Section 6.2.2.

The role assignment in the *ready* state assigns the goalkeeper to the robot with the player number `1` and the striker to the robot that is closest to the central kick-off position. The central kick-off position is directly behind the center spot for an own kick-off and behind the center circle for an opponent kick-off. The remaining players get their supporter indices as usual. An exception is when the *ready* state follows the *initial* state: All roles including the striker flag and the supporter indices are then assigned by the player number, as the positions where we put

the robots at the sidelines are also fixed. The positioning itself is handled by the `ownKickoff` and `opponentKickoff` decks in the `GameplayCard`. There is one card each for the goalkeeper and the striker which walk to fixed positions, and additional cards for supporters exist.

We currently only have special kick-off behavior for the own kick-off. The only relevant behavior for an opponent kick-off is obeying the *illegal defender* rule, i. e. not entering the center circle too early. This is achieved by a high priority card that lets the robot stand as soon as it gets too close to the center circle as long as the ball is not in play.

The default kick-off we used at RoboCup 2019 is the passing kick-off. It consists of three robots standing behind the center line, one of them behind the ball and one per side of the field. After the whistle, the two outer robots move into the opponent half, while the striker evaluates possible kick directions based on present obstacles. The decision is communicated to the other robots, one of which will then adjust its position to be able to consequently play the ball towards the goal after the pass. Once the receiving robot is close enough or too much time has passed, the striker walks to the ball and passes it there.

This kick-off consists of two cards: One for the striker (`PassingOffenseStrikerCard`) and one for the potential pass receivers (`PassingOffenseTargetCard`). The pass receiver card is also present in the `ownKickoff` deck to walk to the position behind the center line. The striker does not need any special positioning in the *ready* state, thus the `PassingOffenseStrikerCard` only needs to contain the behavior after the whistle. First, it stands and looks around for two seconds to get an overview of all obstacles in the opponent half. Then, it continuously calculates a target direction which must be in front of one of its teammates and not blocked by an obstacle using a `SectorWheel`. The sector that is closest to the goal direction while still having a minimum size is chosen and communicated to the teammates.

As a fallback, i. e. when there are too few teammates available to do the passing kick-off, the striker can also do the kick-off alone with the `KickoffStrikerCard`. It also stands and looks around for two seconds to get a complete model of obstacles in the opponent half. Then, it immediately walks to the ball and kicks it using an in-walk kick. The direction is calculated from a `SectorWheel` in the $\pm 60^\circ$ range. This card, too, makes use of obstacle culling as described in the `KickAtGoalCard` (cf. Section 6.2.3).

6.2.8 Ball Search

The normal playing behavior is only sensible and active as long as there is at least one robot that has seen the ball recently. If it has not been seen by any robot for multiple seconds, the team has to find the ball again. The specific search behavior depends on the current situation of the robot and its position. All ball search methods here apply to field players only, as the goalkeeper has its own ball search behavior (cf. Section 6.2.4).

NearFieldSearchForBallCard: When the striker is dueling with an opponent, it can happen that the ball is moved without its immediate notice. If the ball has disappeared for a short amount of time, i. e. it is not seen although it should be in the field of view, and is not likely occluded by the opponent, an attempt is made to quickly regain sight of the ball. Depending on a guess whether the ball is on the left or right side, the robot looks there and walks backwards in an arc to get a good view at the supposed ball location. If no guess of the side could be made, the robot walks straight backwards and turns the head alternately left and right.

SearchForBallAtBallPositionCard: If the ball is far away and not seen anymore (maybe just due to another robot walking through the field of view), it probably remained at the last

known position because it was not moved by any robot. That is why the striker is walking to the position where the ball was last seen while looking actively for the ball.

SearchForBallAtDropInPositionCard: If the position of the ball is not known during a free kick, the ball will be searched where it should have been put according to the rules. In case of a corner kick and a goal free kick, there are only two possible positions for the ball depending on the side where the ball left the field. In case of a kick-in, the ball will be replaced on the field line where the ball left the field. These positions are calculated in the `BallDropInLocator`. Therefore, the ball is only searched at these positions during a corner kick, goal free kick or kick-in. For the pushing free kick the ball will only be replaced if it moved significantly after the foul has been committed. The ball is usually seen during the situation of a foul so that the ball is again searched at the last valid position where the ball was seen.

SearchForBallFieldCoverageCard: If none of the other ball search behaviors was successful or is applicable, the position of the ball is not known at all. Then the ball can be anywhere on the field and every part of the field must be scanned for the ball by the team in a coordinated manner. The `GlobalFieldCoverage` contains a grid discretization of the field in which every cell has the timestamp when it was last seen and a rating for that cell. In order to calculate its patrol target, each robot sorts the cells of the grid descending by this rating. For each cell in this list, the distance to each robot is calculated. If the distance of the cell to the current robot is less than the distance to all of its teammates, a valid patrol target is found.

6.2.9 Penalty Shoot-Out

The penalty shoot-out handling is distributed over one skill, one team card, and two cards that contain the striker and goalkeeper behavior, respectively. The `PenaltyShootoutTeamCard` does nothing more than setting the `isGoalkeeper` flag and `playBall` flag according to the `kickingTeam` field in the GameController packet and sets the remaining fields of the `TeamBehaviorStatus` to their default values. This ensures that nothing goes wrong in the rest of the system which might read them.

Being in a penalty shoot-out also influences some of the perception and modeling components. Especially the ball perception and model for the penalty goalkeeper are tweaked for reactivity.

PenaltyStrikerCard: After waiting for a short amount of time, the striker walks directly to the ball, looking at it, with reduced speed. After about half the distance, it stops for the first time to look for localization features and become more certain about its pose. Then, it randomly decides for one of the goal corners. However, if obstacles are percepted that are far off the center of the goal, the corresponding side is excluded. The robot approaches the kick pose using the `PenaltyStrikerGoToBallAndKick` skill which once again stops shortly before reaching the ball. This is to be sure that the ball is not accidentally hit. Finally, when being precisely aligned behind the ball, the robot executes the standard forward kick from the `KickEngine`.

PenaltyKeeperCard: When the penalty shot attempt starts, the penalty keeper kneels down immediately with its arms spread as that posture allows for a faster jump towards the goal posts. As soon as the ball moves towards the goalkeeper, depending on where the ball is assumed to pass the robot, it either jumps left or right or spreads its legs. Because the robot is manually placed and does not locomote, it does not need to actively localize and therefore only keeps the ball in its field of view.

6.3 Path Planner

When robots want to get to some location on the field, they usually call the `WalkToPoint` skill. If the target is close, it uses a reactive method to avoid obstacles and align to the target orientation. In some situations, however, robots have to walk longer distances to reach their next target location, e. g., when walking to their kick-off positions or when walking to a distant ball. In these cases, a purely reactive control can be disadvantageous, because it usually would not consider obstacles that are further away, which might result in getting stuck.

6.3.1 Approach

The current path planning implementation is a visibility-graph-based 2-D A* planner (cf. Fig. 6.6). It represents obstacles as circles on which they can be surrounded and the path between them as tangential straight lines. As a result, a path is always an alternating sequence of straight lines and circle segments. There are four connecting tangents between each pair of non-overlapping obstacle circles, only two between circles that overlap, and none if one circle contains the other. With up to nine other robots on the field, four goal posts, and the ball, the number of edges in the visibility graph can be quite high. Thus, the creation of the entire graph could be a very time-consuming task. Therefore, the planner creates the graph while planning, i. e. it only creates the outgoing edges from nodes that were already reached by the A* planning algorithm. Thereby, the A* heuristic (which is the Euclidean distance) not only speeds up the search, but it also reduces the number of nodes that are expanded. When a node is expanded, the tangents to all other *visible* nodes that have not been visited before are computed. *Visible* means that no closer obstacle circles intersect with the tangent, which would prevent traveling directly from one circle to another. To compute the visibility efficiently, a sweep line approach is used. As a result, the planning process never took longer than 1 ms per *Cognition* cycle in typical games.

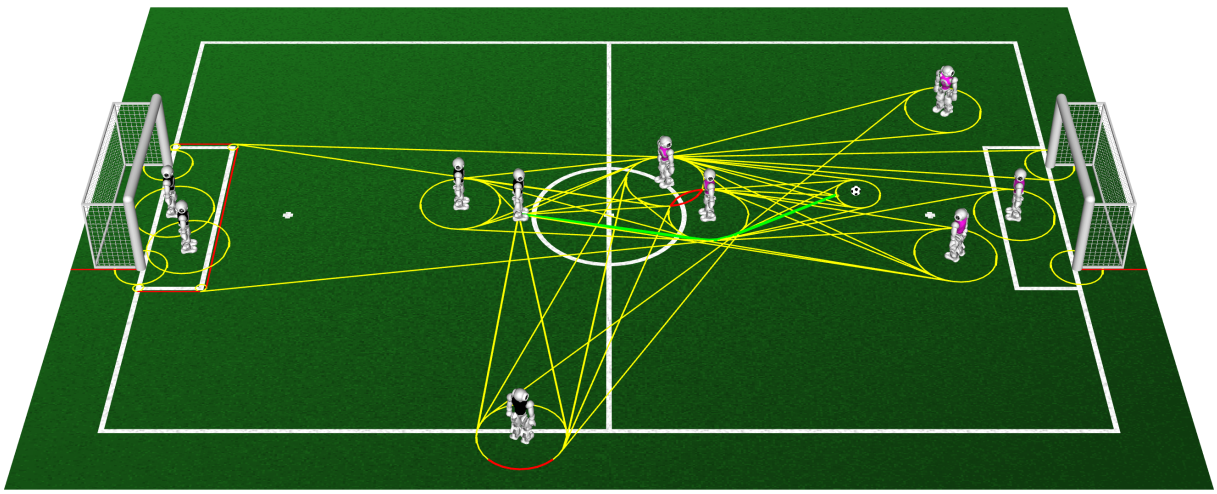


Figure 6.6: Visualization of the planning process. The planning robot is the one left of the center circle. The target position is next to the ball. Obstacle circles are depicted in yellow. Yellow lines indicate expanded edges. Red lines are barriers that must not be crossed. Red circle segments must not be part of the final path. The shortest path is depicted in bright green.

6.3.2 Avoiding Oscillations

Re-planning in each Cognition cycle bears the risk of oscillations, i. e. repeatedly changing the decision, for instance, to avoid the closest obstacle on either the left or the right side. The planner introduces some stability into the planning process by adding an extra distance to all outgoing edges of the start node based on how far the robot had to turn to walk in the direction of that edge and whether the first obstacle is passed on the same side again (no extra penalty) or not (extra penalty). Note that this does not violate the requirement of the A* algorithm that the heuristic is not allowed to overestimate the remaining distance, because the heuristic is never used for the outgoing edges of the start node.

6.3.3 Overlapping Obstacle Circles

The planning process is a little bit more complex than it appears at first sight: As obstacles can overlap, ingoing and outgoing edges of the same circle are not necessarily connected, because the robot cannot walk on their connecting circle segment if this is also inside another obstacle region. Therefore, the planner manages a set of walkable (non-overlapping) segments for each circle, which reduces the number of outgoing edges that are expanded when a circle is reached from a certain ingoing edge. However, this also breaks the association between the obstacle circles and the nodes of the search graph, because since some outgoing edges are unreachable from a certain ingoing one, the same circle can be reached again later through another ingoing edge that now opens up the connection to other outgoing edges. To solve this problem, circles are cloned for each yet unreached segment, which makes the circle segments the actual nodes in the search graph. However, as the graph is created during the search process, this cloning also only happens on demand.

6.3.4 Forbidden Areas

There are two other extensions in the planning process. Another source for unreachable segments on obstacle circles is a virtual border around the field. In theory, the shortest path to a location could be to surround another robot outside of the carpet. The virtual border makes sure that no paths are planned that are closer to the edge of the carpet than it is safe. On demand, the planner can also activate lines surrounding the own penalty area to avoid entering it. The lines prevent passing obstacles on the inner side of the penalty area. In addition, edges of the visibility graph are not allowed to intersect with these lines. To give the planner still a chance to find a shortest path around the penalty area, four obstacle circles are placed on its corners in this mode. A similar approach is also used to prevent the robot from walking through the goal nets.

6.3.5 Avoiding Impossible Plans

In practice, it is possible that the robot should reach a position that the planner assumes to be unreachable. On the one hand, the start position or the target position could be inside obstacle circles. In these cases, the obstacle circles are “pushed away“ from these locations in the direction they have to be moved the least to not overlap with the start/target position anymore before the planning is started[1]. On the other hand, due to localization errors, the start and target location could be on different sides of lines that should not be passed. In these cases, the closest line is “pushed away”. For instance, if the robot is inside its penalty area although

[1] Note that when executing the plan, these situations are handled differently to avoid bumping into other robots.

it should not be, this would move the closest border of the penalty area far enough inward so that the robot's start position appears to be outside for the planning process so that a plan can be found.

6.4 Camera Control Engine

The `CameraControlEngine` takes the `HeadMotionRequest` provided by the `BehaviorControl` module and modifies it to provide the `HeadAngleRequest`, which is used to set the actual head angles. The main function of this module is to make sure that the requested angles are valid. In addition, it provides the possibility to either use the so called *PanTiltMode*, where the user can set the desired head angles manually, or the target mode. In target mode, the `CameraControlEngine` takes targets on the field (provided by the user) and calculates the required head angles by using inverse kinematics (cf. Section 8.3.5). It must also be ensured that the cameras cover as much of the field as possible. Therefore, the `CameraControlEngine` calculates which camera suits best the provided target and sets the angles accordingly. Also, because of this, most of the time the provided target will not be in the middle of either camera image.

6.5 LED Handler

The LEDs of the robot are used to show information about the internal state of the robot, which is useful when it comes to debugging the code.

Right Eye

Color	Role
Red	Playing the ball
Blue	Goalkeeper
Magenta	Striker (currently not playing the ball)
White	Supporter with index 0
Yellow	Supporter with index 1
Green	Supporter with index 2
Cyan	Supporter with index 3

Left Eye

Color	Information
Yellow	No ground contact
White	Ball was seen
Blue	Field feature was seen
Red	Ball and field feature were seen

Torso (Chest Button)

Color	Game / Robot State
Off	Initial / Finished
Blue	Ready
Green	Playing
Yellow	Set
Red	Penalized

Feet

- The left foot shows the team color. For instance, if the team is currently the red team, the color of the LED is red. If the team color is black, this LED will be switched off.
- The right foot shows whether the team has kick-off or not. If the team has kick-off, the color of the LED is white, otherwise it is switched off. In case of a penalty shoot-out, the color of the LED is green, when the robot is the penalty taker, and yellow if it is the goalkeeper.

Ears

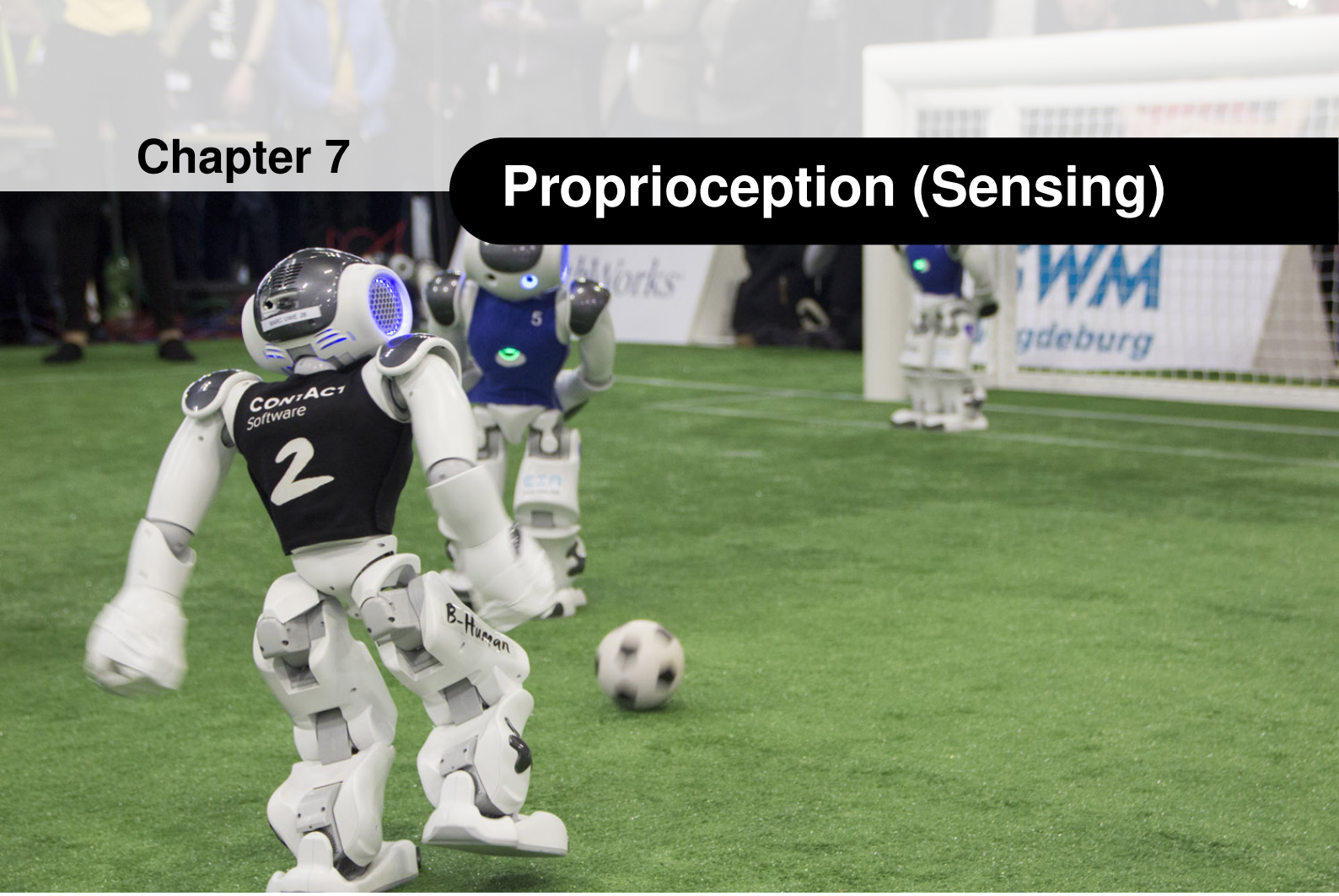
- The right ear shows the battery level of the robot. For each 10% of battery loss, an LED is switched off.
- The left ear shows the number of players connected through the wireless. For each connected player, two LEDs are switched on (upper left, lower left, lower right, upper right). Without game controller access, two LEDs are flashing.

Head

If the robot is charging, the LEDs will be turned on and off one after another. Due to the circular arrangement of the LEDs, this looks like a rotating beam of light.

Chapter 7

Proprioception (Sensing)



The NAO has an inertial measurement unit (IMU) with three acceleration sensors (for the x-, y-, and z-direction) and three gyroscopes (for the rotation around the x-, y- and z-axes). By means of these measurements, the IMU board calculates rough approximations of the robot's torso angles relative to the ground, which are provided together with the other sensor data. Furthermore, the NAO has a sensor for the battery level, eight force sensing resistors on the feet, two ultrasound sensors with different measurement modes, and sensors for the load, temperature, and angle of each joint. We currently do not use the data from the ultrasound sensors.

In the thread *Motion*, the `NaoProvider` receives all sensor readings from LoLA, adds a configured calibration bias for each sensor, and provides them as `FsrSensorData`, `InertialSensorData`, `JointSensorData`, `KeyStates`, and `SystemSensorData`. Also the `JointAngles` encapsulates the `JointSensorData` for the whole system.

The `RobotModel` (cf. Section 7.2) provides the positions of the robot's limbs and the `GroundContactDetector` provides the `GroundContactState` (cf. Section 7.1), which is the information whether the robot's feet are touching the ground. Based on the `JointAngles`, the `InertialData`, the `RobotModel`, the `GroundContactState`, and the currently executed motion, it is possible to calculate the `TorsoMatrix` (cf. Section 7.4), which describes a transformation from the ground to an origin point within the torso of the NAO. The representation `InertialData` is also considered to detect whether a robot has fallen down by the module `FallDownStateDetector`, which provides the `FallDownState` (cf. Section 7.5).

7.1 Ground Contact Detection

To improve the handling of the robots for team members and assistant referees, it is advantageous if the robots can detected when they are picked up and when they are put back on the ground, because they can then stop moving if in the air. This information is provided by the `GroundContactDetector` in the representation `GroundContactState`. The `GroundContactDetector` uses the FSR sensors under the feet and the gyroscopes of the NAO to provide this information. It uses a simple state machine with only two states:

1. If the robot has had ground contact so far, it assumes that it has been picked up if the overall pressure on both feet has been below a threshold for at least 100 ms.
2. If the robot has had no ground contact so far, it assumes that it has regained ground contact if the overall pressure on both feet has been above a threshold, the individual pressure on each foot has been above another threshold, the measurements (rotational speeds of the torso) of the forward and sideways gyroscopes have been below a third threshold, and all of this for consecutive 300 ms. Thereby, it is ensured that the robot is standing on both feet and not shaking when regaining ground contact.

7.2 Robot Model Generation

The `RobotModel` is a simplified representation of the robot. It provides the positions and rotations of the robot's limbs relative to its torso as well as the position of the robot's center of mass (*CoM*). All limbs are represented by homogeneous transformation matrices (*Pose3D*) whereby each limb maps to a joint. By considering the measured joint angles of the representation `JointAngles`, the calculation of each limb is ensured by the consecutive computations of the kinematic chains. Similar to the inverse kinematic (cf. Section 8.3.5), the implementation is customized for the NAO, i. e., the kinematic chains are not described by a general purpose convention such as Denavit-Hartenberg parameters to save computation time.

The *CoM* is computed by Eq. (7.1) with n = number of limbs, $\vec{r}_i$ = position of the center of mass of the i-th limb relative to the torso, and m_i = the mass of the i-th limb.

$$\vec{r}_{com} = \frac{\sum_{i=1}^{n} \vec{r}_i m_i}{\sum_{i=1}^{n} m_i} \tag{7.1}$$

For each limb, $\vec{r}_i$ is calculated by considering the representation `RobotDimensions` and the position of its *CoM* (relative to the limb origin). The limb *CoM* positions and masses are provided in the representation `MassCalibration`. They can be configured in the file *massCalibration.cfg*. The values used were taken from the NAO documentation by SoftBank Robotics.

7.3 Inertia Sensor Data Filtering

The `InertialDataProvider` module determines the orientation of the robot's torso relative to the ground. Therefore, the calibrated IMU sensor readings (`InertialSensorData`) and the measured stance of the robot (`RobotModel`) are processed using an Unscented Kalman filter (UKF) [11].

A three-dimensional rotation matrix, which represents the orientation of the robot's torso, is used as the estimated state in the Kalman filtering process. In each cycle, the rotation of the torso is predicted by adding an additional rotation to the estimated state. The additional rotation is computed using the readings from the gyroscope sensor. Because of noisy measurements by the gyroscopes the prediction has to be corrected. This is achieved by accelerometer measurements of the gravity vector. To avoid confusion in existing modules the orientation is calculated with and without the rotation around the z-axis. The resulting orientations are provided in the representation `InertialData`.

In case the `GyroOffsetProvider` (cf. Section 7.9) detected a malfunction, where the robot lost the connection to its body, the UKF is not updated anymore. When the connection returns, the UKF is reinitialized.

7.4 Torso Matrix

The `TorsoMatrix` describes the three-dimensional transformation from the projection of the middle of both feet on the ground up to the center of hip within the robot torso. Additionally, the `TorsoMatrix` contains the alteration of the position of the center of hip including the odometry. The `CameraMatrix` within the thread *Cognition* is computed based on the `TorsoMatrix`.

In order to calculate the `TorsoMatrix`, the vector of each foot from ground to the torso (f_l and f_r) is calculated by rotating the vector from the torso to each foot (t_l and t_r). This can be calculated by the kinematic chains, according to the estimated rotation (cf. Section 7.3). The estimated rotation is represented as rotation matrix R.

$$f_l = -R \cdot t_l \tag{7.2}$$
$$f_r = -R \cdot t_r \tag{7.3}$$

The next step is to calculate the span s between both feet (from left to right) by using f_l and f_r:

$$s = f_r - f_l \tag{7.4}$$

Now, it is possible to calculate the translation part of the torso matrix p_{im} by using the longer leg. The rotation part is already known since it is equal to R.

$$p_{im} = \begin{cases} s/2 + f_l & \text{if } (f_l)_z > (f_r)_z \\ -s/2 + f_r & \text{otherwise} \end{cases} \tag{7.5}$$

7.5 Detecting a Fall

It is important to detect when a robot falls, when it reaches the ground, and in which direction it has fallen on the ground. Fall detection has two phases. The first one is to determine that the robot is currently falling down and in which direction. The second one is to detect that the robot has reached the ground. It is important to prepare for the ground impact while falling. In case of our team, depending on the fall direction, different fall motions are executed to prevent damage on the head and the body overall (cf. Section 8.4). After the robot has hit the ground, a getup motion is started.

Our `FallDownStateDetector` uses an UKF to determine an accurately position of the *CoM*. The *CoM* modeled as an inverted pendulum is used as the dynamic model. A prediction of the *CoM*, projected on the support foot plane, is then used to determine, if it is still inside the support polygon. The support polygon consists of both feet, to prevent false detections of falls, when the *CoM* leaves the support area of one foot but will eventually enter the one of the other foot (cf. Fig. 7.1).

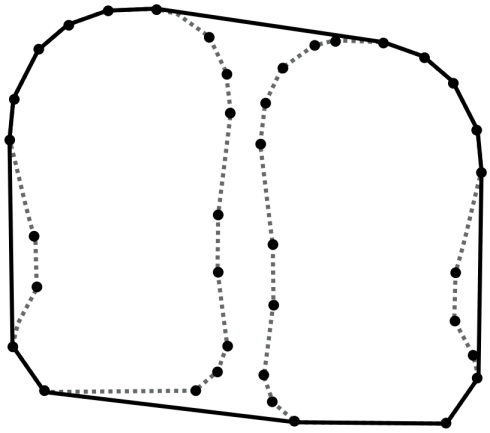


Figure 7.1: The support polygon of the feet

Also to minimize more false positives, if the predicted *CoM* is outside the support area and a fall would be detected, the angle between the directional vector of the velocity of the *CoM* and of the fall direction must be below a certain threshold (cf. Fig. 7.2). Otherwise no fall is detected.

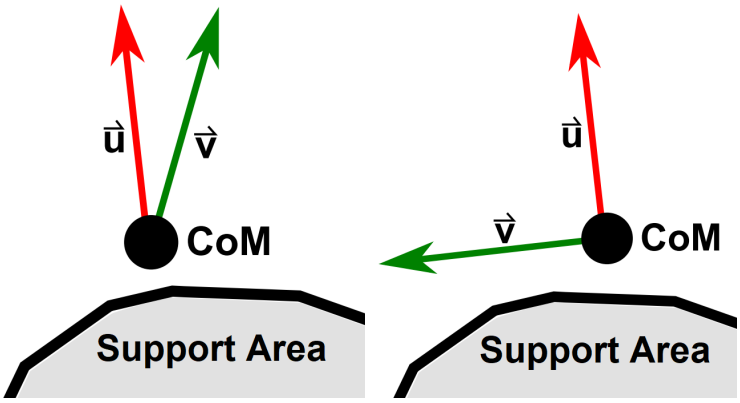
7.6 Arm Contact Recognition

Robots should detect whether they touch obstacles with their arms in order to improve close-range obstacle avoidance. When getting caught in an obstacle with an arm, there is a good chance that the robot gets turned around or results in falling.

In order to improve the precision as well as the reactivity, the `ArmContactModelProvider` is executed within the thread *Motion*. This enables the module to query the required joint angles about 83 times per second. The difference of the intended and the actual position of the shoulder joint per frame is calculated and also buffered over several frames. This allows to calculate an average error. Each time this error exceeds a certain threshold, an arm contact is reported. Using this method, small errors caused by arm motions in combination with low stiffness can be smoothed. Hence, we are able to increase the detection accuracy. Due to the joint slackness, the arm may never reach certain positions. This might lead to prolonged erroneous measurements. Therefore, a malfunction threshold has been introduced. Whenever an arm contact continues for longer than this threshold, all further arm contacts will be ignored until no contact is measured.

The current implementation provides several features that are used to gather information while playing. For instance, we are using the error value to determine in which direction an arm is being pushed. Thereby, the average error is converted into compass directions relative to the robot. In addition, the `ArmContactModelProvider` keeps track of the time and duration of the current arm contact. This information may be used to improve the behavior.

In order to detect arm contacts, the first step of our implementation is to calculate the difference between the measured and commanded shoulder joint angles. Since we noticed that there is



(a) $\langle \vec{v}, \vec{u} \rangle \approx 0.91$ (b) $\langle \vec{v}, \vec{u} \rangle = 0$

Figure 7.2: Comparison of the impact of different fall directions $\vec{v}$ on the scalar with the velocity vector $\vec{u}$ of the *CoM*. (a) would be barely detected as a fall, (b) would not be detected as a fall.

a small delay between receiving new measured joint positions and commanding them, we do not compare the commanded and actual position of one shoulder joint from one frame. Instead we are using the commanded position from n frames[1] earlier as well as the newest measured position.

In our approach, the joint position of one shoulder consists of two components: x for pitch and y for roll. Given the desired position p and the current position c, the divergence d is simply calculated as:

$$d.x = c.x - p.x$$
$$d.y = c.y - p.y$$

In order to overcome errors caused by fast arm movements, we added a bonus factor f that decreases the average error if the arm currently moves fast. The aim is to decrease the precision, i. e. to increase the detection threshold for fast movements in order to prevent false positives. The influence of the factor f can be modified with the parameter `speedBasedErrorReduction` and is calculated as:

$$f = \max\left(0, 1 - \frac{|handSpeed|}{speedBasedErrorReduction}\right)$$

So for each arm, the divergence value d_a actually being used is:

$$d_a = d \cdot f$$

As mentioned above, the push direction is determined from the calculated error of an arm shoulder joint. This error has two signed components x and y denoting the joint's pitch and roll divergences. One component is only taken into account if its absolute value is greater than its contact threshold.

Table 7.1 shows how the signed components are converted into compass directions for the *right* arm. The compound directions NW, NE, SE, SW are constructed by simply combining the above rules if **both** components are greater than their thresholds, e. g. $x < 0 \wedge y < 0$ results into direction SW. The push direction of each arm is used to add an obstacle to the robot's obstacle

[1] Currently, $n = 5$ delivers accurate results.

	x	y
is positive	N	E
is negative	S	W

Table 7.1: Converting signed error values into compass directions for the right shoulder and with $-y$ for the left

model, causing the robot to perform an evasion movement when it hits an obstacle with one of its arms.

Arm contacts are also used to move the arm out of the way to avoid further interference with the obstacle (cf. Section 8.8). Please note that while arm motions from the `ArmMotionEngine` are active, no arm contacts are detected for that arm.

7.7 Foot Bumper Recognition

It is unavoidable that at some point our robots need to duel for the ball in close range or try to walk somewhere when a robot is lying in the way. To detect those other robots, in case they were not recognized in the images, the foot bumpers are used. The bumpers have two sensors on each foot. We sample them over a second and an obstacle is detected, if the contacts per second exceed a threshold. To avoid false positives, we assume a malfunction if a contact is detected longer than a given threshold. Also when the robots are turning while walking the foot bumpers may touch each other causing false detections. To filter these out, the inner sensors are ignored when they are too close together, calculated with the `RobotModel`. Also at least two of the four sensors need to have a detection at least once per second, to assume a valid detection.

7.8 Foot Pressure Filtering

The `FootSupportProvider` filters the `FsrSensorData` and calculates how much weight is supported relative of the feet to each other. This procedure is taken from the walking of rUNSWift [8]. For each of the eight sensors the all time highest measured weight is saved. Then for each sensor the percentage for each measured weight relative to the highest measured weight is calculated. Afterwards a weighted sum of all eight sensors is calculated, which is used to decide if more weight is on one foot than on the other and if the support foot switched. This information is used for the walking (cf. Section 8.3) to start a new step.

Additionally after a period of time the highest measured values are replaced by the highest measured weights of the past few seconds. This is done, because as a result of a ball holding motion or when the robots where placed back on the fields by hand, the sensors might have measured a new highest weight.

Also a simple prediction for when the support foot might switch is calculated. It uses the changing rate of the support foot ratio and approximates a support ratio for a motion frame in advance.

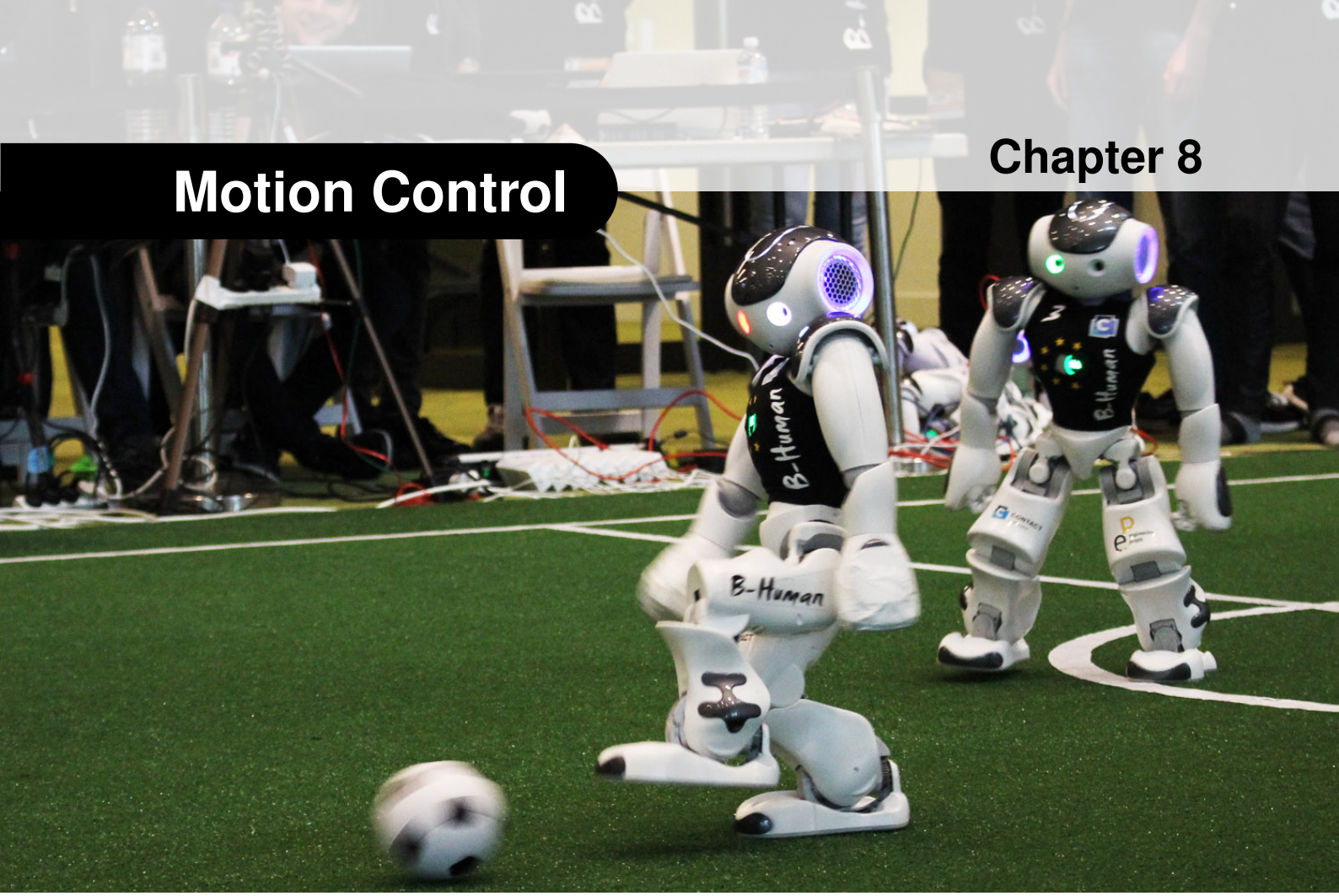
7.9 Gyro Offset Detection

The `GyroOffsetProvider` provides the representation `GyroOffset` and detects a malfunction in the gyroscope. There are two possible malfunctions that can occur. The first one is after a NAO

V6 is started, it can happen that the measurements in the gyro values have offsets. As a result even if the robot is standing in a static pose and is not moving, the offsets in the gyros result in a `TorsoMatrix`, which is rotating around the axis with the offsets. This problem happens when the NAO is moved when booting up. These offsets are detected by calculating the average differences between the gyro values over a period of time. If the average differences are below a certain threshold and the average gyro values above another threshold, then a gyro malfunction is detected and the offsets are saved in the representation `GyroOffset`. These offsets are not used, because it is better to restart the robot to remove them completely.

The other malfunction that can occur is when the robot loses the connection to the body for a short period. In such a case the robot can not be controlled anymore, loses stiffness in all joints and the sensor data is not updated anymore. This can happen randomly for old robots but also when a robot is falling hard on the floor. This disconnection is detected if the time of the last change of the gyro values is above a threshold.

Chapter 8
Motion Control



The B-Human motion control system provides and controls the motions needed to play soccer with a robot. There are six different types of motions that can be requested: *walk*, *stand*, *kick*, *fall*, *getUp* and *specialAction*. These motions are generated by the corresponding motion engines. Additionally, there are specific engines for the head motions (cf. Section 8.7) and arm motions (cf. Section 8.7). The *walk* and *stand* motions are dynamically generated by the `WalkingEngine` (cf. Section 8.3). Moreover, this module provides a special kind of kicks, the in-walk kicks (cf. Section 8.3.3). All the other kicks are generated by the `KickEngine` [17] that models kicks by using Bézier splines and inverse kinematics. When a robot is unintentionally falling the `FallEngine` prevents damage (cf. Section 8.4). A robot which is laying on the floor can get back to stand by using the `GetUpEngine` (cf. Section 8.6). The module `SpecialActions` takes a sequence of predefined joint angles as an input and creates a motion on that basis. It is used wherever an own engine would be too expensive. It is used mainly for ball catching motions.

Each motion engine generates joint angles. Depending on the selected motions (cf. Section 8.1) for the arms and legs, these angles are combined into the `JointRequest` at the end of the thread *Motion* (cf. Section 8.2).

8.1 Motion Selection

According to the representation `MotionRequest` the `MotionSelector` determines which motion is executed and calculates interpolation ratios in order to switch between them. In doing so this module takes into account which motion is demanded and which motion is currently executed,

and whether those motions can be interrupted. The interruptibility of a motion is handled by the corresponding motion engine in order to ensure the motion is not interrupted in an unstable state.

An exception to this behavior is the *fall* motion which activates itself when needed, to achieve a reflex reaction. Thereby it ignores the interruptibility because if the robot is falling it is unstable by default.

The information about the chosen motion is then provided in the representations `LegMotionSelection` and `ArmMotionSelection`. At first the `LegMotionSelection` needs to be chosen. After that, `ArmMotionSelection` can be filled. In the most cases these two representations do not differ from each other. That means the engine which provides the `LegMotionRequest`, also provides the `ArmMotionRequest`.

8.2 Motion Combination

Motion combination is the two-layered task of merging the generated joint angles of all active motions into one. On the upper layer, the `MotionCombinator` takes the three partial `JointRequests` for the head, arms as well as legs and puts them into a full `JointRequest`. On the lower layer each partial `JointRequest` is put together from different motions according to the ratios provided by the `MotionSelector`. This is done in separate subtasks for each motion request (arm, leg and head) which enables an architecture where one part of the motion can be generated dependent on the others. This can be important for balancing tasks.

The `HeadMotionCombinator` takes the HeadYaw and HeadPitch joint angles from the `HeadMotionEngine` by default. These are overwritten by the selected target motion if the regarding joint angles are not set to be ignored. Afterwards, the chosen joint angles are interpolated and stored in the `HeadJointRequest`.

The `ArmMotionCombinator` merges the joint angles for the arms, by taking into account the outputs of all modules producing such angles. Its output is the `ArmJointRequest` which contains those angles and the `ArmMotionInfo` describing the executed motions per arm.

The `LegMotionCombinator` is responsible for merging the joint angles for the legs and outputs them in form of the `LegJointRequest`.

The `MotionCombinator` is executed at the end of the motion cycle. In addition to the things mentioned above, it fills the `MotionInfo`, which contains the actual executed motion, and the `OdometryData`.

8.3 Walking

The `WalkingEngine` provides the joint angles for walking and standing. The module coordinates the actual generation of the walk pattern with the execution of in-walk kicks (cf. Section 8.3.3) and it combines the odometry computed from the request motion with the odometry computed from measured motion.

It is based on the walk developed by Bernhard Hengst [8] for the team UNSW Australia/rUNSWift.

8.3.1 Walk2014Generator

The *Walk2014*, as the walk is called, is based on three major ideas:

1. As in all walks, the body swings from left to right and back during walking to shift away the weight from the swing foot, allowing it to be lifted above the ground. In contrast to many other walks, this is achieved without any roll movements in the leg's joints (unless walking sideways), i. e. the swing foot is just lifted from the ground and set back down again, which is enough to keep the torso in a pendulum-like swinging motion.

2. As a result, there is a clearly measurable event, when the weight of the robot is transferred from the support foot to the swing foot, which then becomes the new support foot. This event is detected by the pressure sensors under the feet of the robot. At the moment this weight shift is detected, the previous step is finished and the new step begins. This means that a transition between a step and its successor is not based on a model, but on a measurement, which makes the walk quite robust to external disturbances.

3. During each step, the body is balanced in the forward direction by the support foot. The approach is very simple but effective: the lowpass-filtered measurements of the pitch gyroscope are scaled by a factor and directly added to the pitch ankle joint of the support foot. As a result, the faster the torso turns forward, the more the support foot presses against this motion.

8.3.2 Generating a Step

A step can be generated in three different requests, all coming from the `MotionRequest`, as *speedMode*, *targetMode* and *stepSizeMode*. The *speedMode* is used to command the robot to walk at a specific forward, sideways and turn speed, the *targetMode* is used to let the robot walk to a specific point and the *stepSizeMode* is used for our kicks while walking. When using the *targetMode*, the requested target is converted into a speed request. If the target can not be reached within one step, the *speedMode* is used with the converted speed request. In the *speedMode* the speed request is first clipped based on an ellipsoid clamping algorithm. This is done to avoid a requested step which is unreachable for the feet, but also to remain a stable walk in all walking directions. Afterwards, the speed values are clipped based on the configured maximal speed values for forward, backward, sideways and turning. Also in all three cases, a planned step duration and odometry offset are calculated and based on the step duration the variables used to calculate the feet positions to execute the step.

Also if it is detected, that the robot is stuck on one foot, e. g. resulting from walking against a goal post, an emergency step is requested, where the robot pushes the current swing leg into the ground. If this does not resolve the situation, the robot returns into a standing position. While in the emergency mode, the request from the `MotionRequest` is ignored.

If the robot starts its first step, a pre-step is executed to start the pendulum like swinging, unless the feet are close together. In such a case the first step is executed as normal.

For every following motion frame, the current positions of both feet are calculated based on inverse kinematic. If an in-walk kick (cf. Section 8.3.3) is requested, the offsets for the feet are also added.

Since our robots sometimes take their arms behind their back to better avoid obstacles, the center of mass gets shifted. Therefore, a dynamic center of mass computation, which considers the impact of the arms' positions, is used to counteract the arm positions. Iteratively, the torso

is tilted in a way that the center of mass's projection to the ground keeps at the same forward position relative to the feet as the center of mass with regular arm positions would have.

Afterwards, the lowpass-filtered measured gyroscope value is added with a factor on the ankle pitch request. The factor is different for positive and negative values because the feet are shorter at the back and stability benefits from slightly more aggressive balancing in that direction. We also found that a lot of differences between different robots can be compensated by varying the balancing parameters. Also only when the robots are standing they are balancing sideways with the ankle rolls.

A new step starts when the weight shifts from one leg to the other. We noticed that because of the three motion frame delay until a request for the joints is executed, the robots tend to push their swing leg, which switched to the support leg, into the ground and therefor pushing themself backwards again. To counteract this problem, we use the foot switch prediction from the `FootSupport` (cf. Section 7.8) to start a new step on average one motion frame earlier.

Also, we noticed that the robots are capable of walking a small amount faster forward if they are also walking sideways. One explanation could be, that because we increase the step duration when walking sideways combined with the more spread out legs, disturbances are partly suppressed. We added two parameters to allow for higher forward walking speed. If the robot is walking sideways at least with the speed defined by the parameter `triggerForwardBoostByThisLeftAmount`, his maximal allowed forward speed is replaced by the value of the parameter `sideForwardMaxSpeed`.

8.3.3 In-Walk Kicks

Besides walking and standing, the `WalkingEngine` has also minor kicking capabilities. The tasks of walking and kicking are often treated separately, both solved by different approaches. In the presence of opponent robots, such a composition might waste precious time as certain transition phases between walking and kicking are necessary to ensure stability. A common sequence is to walk, stand, kick, stand, and walk again. Since direct transitions between walking and kicking modules are likely to let the robot stumble or fall over, the `WalkingEngine` can carry out simple kicks while walking.

At the moment, there are five kick types: `forward`, `forwardLong`, `forwardShort`, `sidewardsOuter` and `turnOut`. Those kicks are described individually by the config files located in *Config/WalkKicks*. Every kick is defined by its duration, step sizes before and during the kick, and keyframes for the 3D position and rotation of the kicking foot. These keyframes are later interpolated by cubic splines to describe the actual trajectory of the foot, by overlaying the original trajectory of the swinging foot and thereby describing the actual kicking motion. In doing so the overlaying trajectory starts and stops at 0 in all dimensions both in position and velocity. Thus, the kick retains the start and end positions as well as the speeds of a normal step. The instability resulting from the higher momentum of the kick is compensated by the walk during the steps following the kick.

8.3.4 Learning Parameters Online

At the past competitions we had the problem, that when the robots heated up or played a lot of games, they started to fall more often because they could not walk anymore. To counteract this, we added an automatic adjustment system for the gyro balancing parameters, which are used to balance with the ankle pitches. It uses the past peaks in the gyro measurements and adjusts the balance parameters by random, but always increasing and then decreasing them.

Afterwards, the sampled gyro peaks are used to determine, which parameters resulted in a more stable walk.

We noticed, that at the moment the joints are too hot and their stiffness is reduced, the balance parameters start to increase, which furthermore results in a stable walk. Also after a few minutes playing the parameters stabilized themself. That way we can adjust the starting parameters for separate robots after a game, without the time investment, because we already know good working parameters.

8.3.5 Inverse Kinematic

The inverse kinematics for the ankles of the NAO is a central component of the module `WalkingEngine`. In general, they are a handy tool for generating motions, but solving the inverse kinematics problem analytically for the NAO is not straightforward, because of two special circumstances:

- The axes of the hip yaw joints are rotated by 45 degrees.
- These joints are also mechanically connected among both legs, i. e., they are driven by a single servo motor.

The target of the feet is given as homogeneous transformation matrices, i. e., matrices containing the rotation and the translation of the foot in the coordinate system of the torso. In order to explain our solution we use the following convention: A transformation matrix that transforms a point p_A given in coordinates of coordinate system A to the same point p_B in coordinate system B is named $A2B$, so that $p_B = A2B \cdot p_A$. Hence the transformation matrix $Foot2Torso$ is given as input, which describes the foot position relative to the torso. The coordinate frames used are depicted in Fig. 8.1.

The position is given relative to the torso, i. e., more specifically relative to the center point between the intersection points of the axes of the hip joints. So first of all the position relative to the hip is needed[1]. It is a simple translation along the y-axis[2]

$$Foot2Hip = Trans_y\left(\frac{l_{dist}}{2}\right) \cdot Foot2Torso \tag{8.1}$$

with l_{dist} = distance between legs. Now the first problem is solved by describing the position in a coordinate system rotated by 45 degrees so that the axes of the hip joints can be seen as orthogonal. This is achieved by a rotation around the x-axis of the hip by 45 degrees or $\frac{\pi}{4}$ radians.

$$Foot2HipOrthogonal = Rot_x(\frac{\pi}{4}) \cdot Foot2Hip \tag{8.2}$$

Because of the nature of the kinematic chain, this transformation is inverted. Then the translational part of the transformation is solely determined by the last three joints, which means they can be computed directly.

$$HipOrthogonal2Foot = Foot2HipOrthogonal^{-1} \tag{8.3}$$

The limbs of the leg and the knee form a triangle, in which an edge equals the length of the translation vector of $HipOrthogonal2Foot$ (l_{trans}). Because all three edges of this triangle are

[1] The computation is described for one leg and can be applied to the other leg as well.

[2] The elementary homogeneous transformation matrices for rotation and translation are noted as $Rot_{<axis>}(angle)$ resp. $Trans_{<axis>}(translation)$.

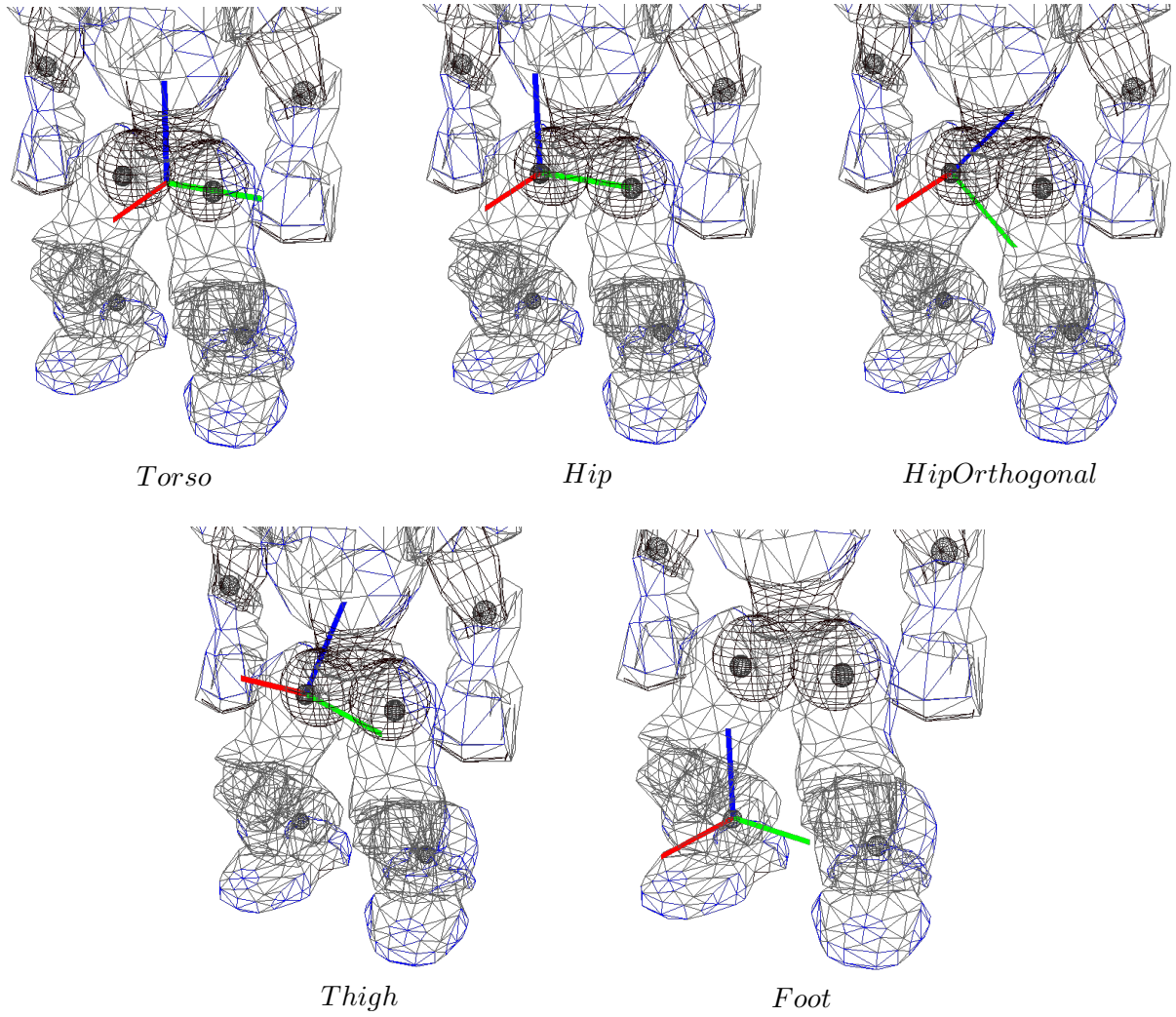


Figure 8.1: Visualization of coordinate frames used in the inverse kinematic. Red = x-axis, green = y-axis, blue = z-axis.

known (the other two edges, the lengths of the limbs, are fix properties of the NAO) the angles of the triangle can be computed using the law of cosines (Eq. (8.4)). Knowing that the angle enclosed by the limbs corresponds to the knee joint, that joint angle is computed by Eq. (8.5).

$$c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos\gamma \tag{8.4}$$

$$\gamma = \arccos \frac{{l_{upperLeg}}^2 + {l_{lowerLeg}}^2 - {l_{trans}}^2}{2 \cdot l_{upperLeg} \cdot l_{lowerLeg}} \tag{8.5}$$

Because γ represents an interior angle and the knee joint is being stretched in the zero-position, the resulting angle is computed by

$$\delta_{knee} = \pi - \gamma \tag{8.6}$$

Additionally, the angle opposite to the upper leg has to be computed, because it corresponds to the foot pitch joint:

$$\delta_{footPitch1} = \arccos \frac{{l_{lowerLeg}}^2 + {l_{trans}}^2 - {l_{upperLeg}}^2}{2 \cdot l_{lowerLeg} \cdot l_{trans}} \tag{8.7}$$

Now the foot pitch and roll joints combined with the triangle form a kind of pan-tilt-unit. Their joints can be computed from the translation vector using atan2.[3]

$$\delta_{footPitch2} = \text{atan2}(x, \sqrt{y^2 + z^2}) \tag{8.8}$$

$$\delta_{footRoll} = \text{atan2}(y, z) \tag{8.9}$$

where x, y, z are the components of the translation of $Foot2HipOrthogonal$. As the foot pitch angle is composed of two parts it is computed as the sum of its parts.

$$\delta_{footPitch} = \delta_{footPitch1} + \delta_{footPitch2} \tag{8.10}$$

After the last three joints of the kinematic chain (viewed from the torso) are determined, the remaining three joints that form the hip can be computed. The joint angles can be extracted from the rotation matrix of the hip that can be computed by multiplications of transformation matrices. For this purpose, another coordinate frame $Thigh$ is introduced that is located at the end of the upper leg, viewed from the foot. The rotation matrix for extracting the joint angles is contained in $HipOrthogonal2Thigh$ that can be computed by

$$HipOrthogonal2Thigh = Thigh2Foot^{-1} \cdot HipOrthogonal2Foot \tag{8.11}$$

where $Thigh2Foot$ can be computed by following the kinematic chain from foot to thigh.

$$Thigh2Foot = Rot_x(\delta_{footRoll}) \cdot Rot_y(\delta_{footPitch}) \cdot Trans_z(l_{lowerLeg}) \cdot Rot_y(\delta_{knee}) \cdot Trans_z(l_{upperLeg}) \tag{8.12}$$

To understand the computation of those joint angles, the rotation matrix produced by the known order of hip joints (yaw (z), roll (x), pitch (y)) is constructed (the matrix is noted abbreviated, e. g. c_x means $\cos \delta_x$).

$$Rot_{Hip} = Rot_z(\delta_z) \cdot Rot_x(\delta_x) \cdot Rot_y(\delta_y) = \begin{pmatrix} c_y c_z - s_x s_y s_z & -c_x s_z & c_z s_y + c_y s_x s_z \\ c_z s_x s_y + c_y s_z & c_x c_z & -c_y c_z s_x + s_y s_z \\ -c_x s_y & s_x & c_x c_y \end{pmatrix} \tag{8.13}$$

The angle δ_x can obviously be computed by $\arcsin r_{21}$.[4] The extraction of δ_y and δ_z is more complicated, they must be computed using two entries of the matrix, which can be easily seen by some transformation:

$$\frac{-r_{01}}{r_{11}} = \frac{\cos \delta_x \cdot \sin \delta_z}{\cos \delta_x \cdot \cos \delta_z} = \frac{\sin \delta_z}{\cos \delta_z} = \tan \delta_z \tag{8.14}$$

Now δ_z and, using the same approach, δ_y can be computed by

[3] atan2(y, x) is defined as in the C standard library, returning the angle between the x-axis and the point (x, y).

[4] The first index, zero based, denotes the row, the second index denotes the column of the rotation matrix.

$$\delta_z = \delta_{hipYaw} = \text{atan2}(-r_{01}, r_{11}) \tag{8.15}$$

$$\delta_y = \delta_{hipPitch} = \text{atan2}(-r_{20}, r_{22}) \tag{8.16}$$

At last, the rotation by 45 degrees (cf. Eq. (8.2)) has to be compensated in joint space.

$$\delta_{hipRoll} = \delta_x - \frac{\pi}{4} \tag{8.17}$$

Now all joints are computed. This computation is done for both legs, assuming that there is an independent hip yaw joint for each leg.

The computation described above can lead to different resulting values for the hip yaw joints. From these two joint values, a single resulting value is determined, in which the interface allows setting the ratio. This is necessary because if the values differ, only one leg can realize the desired target. Normally, the support leg is supposed to reach the target position exactly. By applying this fixed hip joint angle the leg joints are computed again. In order to face the six parameters with the same number of degrees of freedom, a virtual foot yaw joint is introduced, which holds the positioning error provoked by the fixed hip joint angle. The decision to introduce a foot *yaw* joint was mainly taken because an error in this (virtual) joint has a low impact on the stability of the robot, whereas other joints (e. g. foot pitch or roll) have a huge impact on stability. The computation is almost the same as described above, except it is the other way around. The need to invert the calculation is caused by the fixed hip joint angle and the additional virtual foot joint because the imagined pan-tilt-unit is now fixed at the hip and the universal joint is represented by the foot.

This approach can be realized without any numerical solution, which has the advantage of a constant and low computation time and a mathematically exact solution instead of an approximation.

8.4 Falling

When a robot is unintentionally falling to the ground there is a high risk that the robot breaks. Therefore, falls are detected (cf. Section 7.5), so that damage can be prevented by executing certain motions and softening some limbs. Depending on whether the robot is falling to the front or the back, different motions are executed. If the robot falls backwards a squatting position is adopted to decrease the drop height of the head. Additionally, the arms are brought back to a neutral position, if necessary, and the head is tilted to the front. If the robot falls forward the *stand* motion is called and the head is placed in the neck. In this case, the squatting position is not adopted because this would produce an unfavorable angle between the head and the floor when hitting it. Once the `FallEngine` has been activated, it is left if a *getUp* motion is demanded or the robot has been put up manually. To avoid a false activation of the `FallEngine`, certain situations are excluded: For example, the `GetUpEngine` cannot trigger the `FallEngine` and *specialAction* motions are entirely ignored if no game controller packages are received for testing issues.

8.5 Special Actions

Special actions are hardcoded motions that are provided by the module `SpecialActions`. By executing a special action, different target joint values are sent consecutively, allowing the robot to perform actions such as a goalie dive or the high stand posture. Those motions are defined in *.mof* files that are located in the folder *Config/mof*. A *.mof* file starts with the unique name of the special action, followed by the label *start*. The following lines represent sets of joint angles, separated by whitespace. The order of the joints is as follows: head (pan, tilt), left arm (shoulder pitch/roll, elbow yaw/roll, wrist, hand), right arm (shoulder pitch/roll, elbow yaw/roll, wrist, hand), left leg (hip yaw-pitch/roll/pitch, knee pitch, ankle pitch/roll), and right leg (hip yaw-pitch[5]/roll/pitch, knee pitch, ankle pitch/roll). A '*' does not change the angle of the joint (keeping, e. g., the joint angles set by the head motion engine), a '–' deactivates the joint. Each line ends with two more values. The first decides whether the target angles will be set immediately (the value is 0); forcing the robot to move its joints as fast as possible, or whether the angles will be reached by interpolating between the current and target angles (the value is 1). The time this interpolation takes is read from the last value in the line in milliseconds. If the values are not interpolated, the robot will set and hold the values for that amount of time instead.

It is also possible to change the stiffness of the joints during the execution of a special action. This is done by a line starting with the keyword *stiffness*, followed by a value between 0 and 100 for each joint (in the same order as for specifying actual joint angles). In the file *Config/stiffnessSettings.cfg* default values are specified. If only the stiffness of certain joints should be changed, the others can be set to '*'. This will cause those joints to use the default stiffness. At the end of one stiffness line, the time has to be specified that it will take to reach the new stiffness values. This interpolation time runs in parallel to the interpolation time between target joint angles. In addition, the stiffness values defined will not be reached if another stiffness command is executed before the interpolation time has elapsed.

Transitions are conditional statements. If the currently selected special action is equal to the first parameter, the special action given in the second parameter will be executed next, starting at the position of the label specified as the last parameter. Note that the currently selected special action may differ from the currently executed one because the execution costs time. Transitions allow defining constraints such as *to switch from A to B, C has to be executed first*. There is a wildcard condition *allMotions* that is true for all currently selected special actions. Furthermore, there is a special action called *extern* that allows leaving the module `SpecialActions`, e. g., to continue with walking. *extern.mof* is also the entry point to the special action module. Therefore, all special actions must have an entry in that file to be executable. A special action is executed line by line until a transition is found the condition of which is fulfilled. Hence, the last line of each *.mof* file contains an unconditional transition to *extern*.
An example of a special action:

```
motion_id = stand
label start
stiffness 20 20 60 60 60 60 60 60 60 60 60 60 60 60 90 90 90 90 90 90 90 90 90 90 90 90 50
"HP HT AL0 AL1 AL2 AL3 AL4 AL5 AR0 AR1 AR2 AR3 AR4 AR5 LL0 LL1 LL2 LL3 LL4 LL5 LR0 LR1 LR2 LR3 LR4 LR5 Int Dur
  *  *   0 -50  -2 -40   0   0   0 -50  -2 -40   0   0  -6  -1 -43  92 -48   0  -6  -1 -43  92 -48  -1   1 100
transition allMotions extern start
```

To receive proper odometry data for special actions, they have to be manually set in the file *Config/specialActions.cfg*. It can be specified whether the robot moves at all during the execution of the special action, and if yes, how it has moved after completing the special action,

[5] Ignored

or whether it moves continuously in a certain direction while executing the special action. It can also be specified whether the motion is stable, i. e., whether the camera position can be calculated correctly during the execution of the special action. Several modules in the thread *Cognition* will ignore new data while an unstable motion is executed to protect the world model from being impaired by unrealistic measurements.

8.6 Get Up Motion

When a robot falls, it must get up on its own. This should be done in the shortest amount of time as possible, to minimize the time it can not contribute to the game and therefore the time the team is in a numerical disadvantage. It is also important that the get up motion works with the first try. Otherwise, the robot damages itself even more and risks to be unable to play for the rest of the game with a follow up repair afterwards.

To execute get up tries with a high success rate, we developed several approaches, that go hand-in-hand, to prevent incorrect motion states and balance the robot in a way, that it can stand up even in really harsh environments. Yet the defined motions are still a chain of different predefined keyframes, which are linear interpolated.

When the robot falls and the module `GetUpEngine` goes active, there are three possible states it could be in. If it is lying on the back or the front, then it executes a recover motion, so it can start the get up motion from a default position. If it is lying on the side, then depending on how the torso is tilted it executes a motion to tilt over to the front or back. Last but not least it could have executed a SpecialAction before, like a ball holding motion with spread legs. In such a case we do two checks: if the robot is still upright we assume it is still standing on its feet and execute a special motion to bring the robot into a default position, so it can continue getting up from such a position. Otherwise, we assume the robot is lying on its back or front and execute the normal recover motion.

After the recover motion, we check if the robot is lying on its arms when lying on its front. In such a case we execute a motion to free up the arms, otherwise, our front motion for getting up will not work.

When the robot started to execute a motion to get up, several developed algorithms are active to ensure, that disturbances like pushing robots or issues with the field like bumps do not result in failed get up tries. These system parts are a PID-Controller for balancing (cf. Section 8.6.1), conditional keyframes and wait times (cf. Section 8.6.2), a joint compensation (cf. Section 8.6.3) and some additional smaller stuff (cf. Section 8.6.4) to increase the success rate and reduce as much damage to the robot as possible.

Not always every situation can be handled and the robot will inevitably fall again and need to start a new get up try. For such a case every keyframe has defined thresholds for how much the torso is allowed be tilted. If the torso is tilted too much the get up try is aborted, the joints stiffness is set to a low value and the head is moved into a safe position, to reduce as much damage as possible. After a short wait time, the process to decide which motion should be executed is started again. Also, a counter how many tries were executed is increased. If this counter exceeds a maximal try counter, the robot goes into a help me state. In this state, the robot keeps lying on the ground with no stiffness and waits until his torso is upright again, resulting from the help of a human.

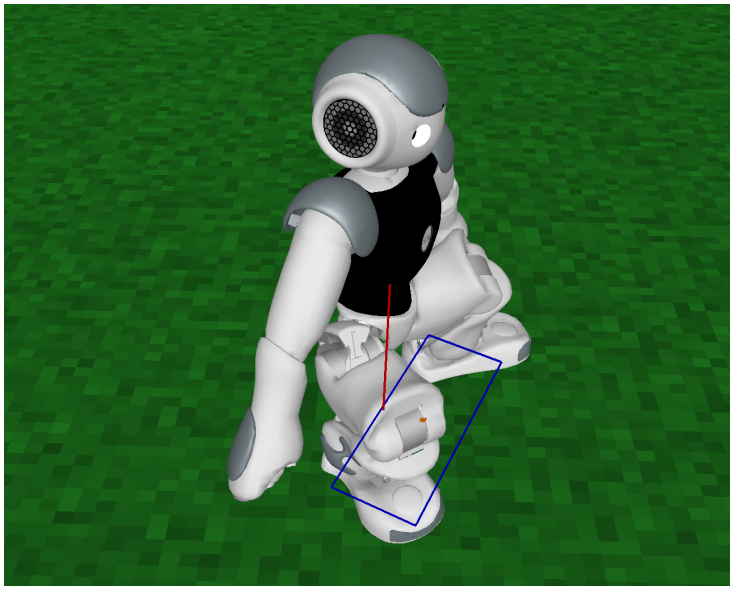


Figure 8.2: The center of mass projected on the ground (red line) in the support polygon (blue) with the middle point of the support polygon (orange) while getting up.

8.6.1 Balancing

For normal balancing, we use a PID-Controller. The input is the difference of the middle point of a simplistic support polygon and the center of mass projected on the ground (cf. Fig. 8.2). Additionally, we defined different states for the keyframes and more than one keyframe can be in the same state, like sitting, lying or standing. Every state has different parameters for the PID-Controller, hand tuned, to ensure that the controller balances out disturbances depending on the executed motion. Also, the balancing is increased the more the robot deviates from the ideal movement to prevent falling to the front or back.

8.6.2 Conditions

Every keyframe can have conditions, which must be true to result in an execution of this keyframe. Otherwise, it is skipped. Also following keyframes can be linked together, so a block of keyframes is only executed if their conditions are fulfilled. Also, every keyframe can have waiting conditions. If a keyframe finished, then the motion is stopped for a given time until the waiting condition is fulfilled. Every condition value is represented as a number and a condition is fulfilled if this number lies between two given thresholds. Additionally, wait conditions have a time parameter. If the wait time exceeds the time threshold, the waiting process is aborted and the motion is continued as normal. Examples for conditions are the torso tilting, the current gyro values or a number, which comes from the `DamageConfiguration` of the robot.

We use the waiting conditions to ensure that specific motion processes like letting the robot tilt over from a lying position to an upright position are not interrupted too early and result in a robot, that keeps lying on the ground, or after the robots reached a sitting position wait a moment to balance out disturbances, to ensure they don't fall over when interpolating into a stand.

The keyframe conditions are used to check for problems when starting a get up try, like incorrectly positioned arms.

8.6.3 Joint Compensation

We analyzed the get up tries from the past years with a focus on the joints. We wanted to know what caused get up tries to fail and why some robots had no problems while others could only get up with hand tuned keyframes. We noticed that for nearly every get up try at some point the hipYawPitch joint requested and measured angles differed by more than 10 degrees, often even more than 20 degrees, which resulted in a torso tilting of 10 to 20 degrees more to the front, just before falling. In cases where the robots did not fall to the front based on this issue, they often fell afterwards to the back, because the hipYawPitch joint suddenly was able to move and literally ejected the robot. A similar problem occurred for our fast front get up motion. At some point, when the robot should have moved the legs together, the requested and measured angles for the hipYawPitch always differed by more than 10 degrees and in cases, where the robots fell back to the front, the hipYawPitch was completely unable to move.

To counteract this problem we expanded the definition of our keyframes. Every keyframe can have several joints that shall be compensated and a number of joints that shall be used for compensation with factors. The compensation algorithm takes the measured angle of the to compensated joint and subtract it with the requested angle from 3 frames before. This value is then multiplied with the factors defined in the keyframes and added on top of the target angles of the joints used for compensation.

This ensures that the robots keep an upright position while some joints are unable to move. But at some point, the stuck joints start to move again and try to reach the requested angles, which could differ by 10 to up to 50 degrees. To ensure that the robots do not eject themselves in these scenarios we check in every motion frame if the stuck joints started to move again. In such a case, we remove the compensation of the previous and current keyframe over a few frames.

8.6.4 Other Smaller Improvements

To reduce the likelihood of failed get up tries, we implemented several ideas. One is a balance out phase after the robot finished the get up motion and reached a standing position. In such a case the robot is still balancing until some given conditions are fulfilled. We use the torso angle, the gyro and the differential output of our PID-Controller as conditions. Also if the balancing takes too long we just take the risk and leave the `GetUpEngine` so the robot can take part in the game again.

Another idea is the use of our conditional keyframes in another way. If a wait time of a previous keyframe was too long or the motion is aborted, this information can be used to execute another keyframe instead of continuing to abort.

Also when a robot is still lying on the ground and aborted the get up try, we use the information in which keyframe the motion aborted and retry from a specific keyframe from that motion. One example is for our back motion, where at some point the robot moves its legs up in the air to tilt over. Sometimes, resulting from disturbances, the robot keeps lying on the ground. In such a case it would be a waste of time to restart the whole getting up process because we assume the robot is already in a correct pose. So we revert to a previous keyframe and just continue the get up try from there, without increasing the counter for the get up tries.

8.7 Head Motions

Besides the motion of the body (arms and legs), the head motion is handled separately. This task is encapsulated within the separate module `HeadMotionEngine` for two reasons. The first reason is that the modules `WalkingEngine` and `KickEngine` manipulate the center of mass. Therefore these modules need to consider the mass distribution of the head *before* its execution in order to compensate for head movements.

The other reason is that the module smoothens the head movement by limiting the speed as well as the maximum angular acceleration of the head. In encapsulating this, the other motion modules do not have to concern themselves with this.

The `HeadMotionEngine` takes the representation `HeadAngleRequest` generated by the `CameraControlEngine` (cf. Section 6.4) and produces the representation `HeadJointRequest`.

8.8 Arm Motions

In addition to the arm motions that are calculated combined with the leg motions (e. g. during a special action or a kick) we also use a motion engine that provides only output for the arm joints: The `ArmKeyFrameEngine` will only be selected during a walking or standing (leg) motion to follow keyframes.

Although we are mainly using this feature to move the arms behind the robot's back to prevent (further) contact with other robots, it generally supports any desired arm keyframe motion. A motion is defined by a set of states, which consist of all target angles of an arm (similar to a special action, cf. Section 8.5). Upon executing an arm motion, the motion engine interpolates intermediate angles between two states to provide a smooth motion. When reaching the final state, the arm remains there until the engine gets a new request. The `ArmKeyFrameEngine` performs all the calculations and interpolations for the desired arm motion.

Arm motions are configured in the file *armKeyFrameEngine.cfg*. It defines the sets of targets as well as some other configuration entries:

actionDelay: Delay (in ms) after which an arm motion can be triggered again by an arm contact.

targetTime: Time (in *Motion* frames) after which an arm should be moved back to its default position (applies only for motions triggered by arm contact).

allMotions: Array of different arm motions. The entry with the id `useDefault` is a native motion, which should neither be modified nor should it get another index within the array.

Despite the available motions, you may add further ones to the file following the same structure under a new *id*. The new *id* has to be appended to the end of the enumeration `ArmKeyFrameId` within the representation `ArmKeyFrameRequest`.

```
// [..] config entries left out
allMotions = [
  {
    // [..] native motions left out
  },{
    id = sampleArmMotion;
    states = [
      {angles=[-2.05, 0.2, 0.174, -0.2, -90deg, 0deg]; stiffness = [90, 60,
          80, 90, 90, 90]; steps = 80; },
```

```
        {angles=[-2.11, -0.33, 0.4, -0.32, -90deg, 1]; stiffness = [90, 60, 80,
            90, 90, 90]; steps = 80; }
      ];
    }
  ];
```

Listing 8.1: Example of an arm motion definition

It is important to know that all motions will only be declared for the left arm. Inside the `ArmKeyFrameEngine`, the motion will then be correctly mirrored for the right arm. The angles and stiffness values of each state correspond in their order of occurrence to: `ShoulderPitch`, `ShoulderRoll`, `ElbowYaw`, `ElbowRoll`, `WristYaw`, and `Hand`. For each stiffness entry, the special value -1 may be used, which indicates the use of the default stiffness value for that joint as specified in the file *stiffnessSettings.cfg*. The attribute `step` specifies how many *Motion* frames should be used to interpolate intermediate angles between two entries in the array `states`.

8.9 Energy Efficient Stand

Inspired by the SPL team Berlin United, we developed a similar approach to cool down our robots while standing [16]. The main idea is the same: when standing if the current of a joint is above a threshold take the requested and measured angle and adjust the request until the current is below a certain threshold.

In contrast to Berlin United, we adjust the requested angles for all joints with a current above a certain threshold, two times a second. Also, the adjustment is made in steps. If the difference between the request and the measurement is too high, the request is adjusted by a large value, otherwise, a value smaller than the minimal gear step length is used. Also if the current is not decreasing on a joint for some time, the request is adjusted again by a larger value. This is done because we noticed that especially the knees, while standing, only reduce their current when the request forces the knees to close a small amount more. Also when a robot is in a high stand we apply the energy saving on the arms too, but with a smaller current threshold.

With this approach, the robots do not heat up when in the high stand. At the GermanOpen two of our robots needed to stand for one hour at the referee meeting and the leg joint temperatures did only rise for 1°C, from 27°C to 28°C. In case a robot is placed by hand into our normal stand position or was walking and stopped by itself, we got mixed results. For our new NAO V6 so far the joints do not heat up but will cool down. For our old robots some joints will slowly heat up, but much slower than without the energy saving.

8.10 Stability Increasement for the Kicks

We did some small quality of life changes to our kicking, which is based on [17] because we wanted to reduce the amount of needed calibration between new and old robots. We noticed that it is not hard to let the robot kick the ball far, but to make sure that the robot will not fall afterwards. Many falls result from the problem that after the ball is kicked, the kicking foot touches the ground too early and pushes the robot away resulting in a fall. To prevent this we added a check that after the ball was kicked the kicking foot x- and z-translations are frozen when the foot would touch the ground.

Additionally, we revisited the idea of the dynamic points. Originally when dynamic points are added the control points are rotated around them. We removed this part because we noticed that the ball is rolling more straight and does less curves without this feature.

Chapter 9
Challenges & Mixed Team

In addition to the main soccer competition, B-Human also participated in the *Open Research Challenge*, *Directional Whistle Challenge* as well as in the *Mixed Team Tournament*. While the Technical Challenges required special modules which are described in the following sections, the Mixed Team Tournament was played with our standard code and only minor changes.

9.1 Open Research Challenge

In recent years, we continuously increased our research towards applications of Deep Learning in the SPL. While a ball detection based on Deep Learning appears to be state of the art, the detection of other robots is not very common among other teams. In this year's Open Research Challenge, we aimed to demonstrate recent results in ball and robot detection as well as state estimation by letting a robot (referred to as the player) score a goal blindfolded through guidance of two teammates (referred to as the observers).

In our setup, the observers are each placed at one of the intersections of the center line and the sidelines, looking towards the center circle. The ball is placed somewhere in the opponent half, for instance at the penalty mark. Finally, the blindfolded player is placed with random position and rotation inside the center circle. After being unpenalized, the observers will move freely around the field to take useful observing poses and to estimate the pose of the player and the position of the ball. This information is transferred wirelessly to the player. The player merges the percepts it receives from the observers into an estimate of its pose and the ball position. Based on that, it aims to walk blindly towards the ball and attempts to score a goal. A possible constellation that could occur during the demonstration is depicted in Fig. 9.1 and a video of

Figure 9.1: A possible constellation during the demonstration

the performance is available online[1].

9.1.1 Perception

During the last year, we attempted to realize a semantic segmentation of the field. The core idea was to replace the `ECImage`, which is currently segmented based on thresholds that have to be defined manually and globally, with a neural network based semantic segmentation. The main difficulty in finding global thresholds are firstly that the lighting conditions change globally in the image (i. e. clouds moving in front of the sun) and secondly that hard shadows/patches of light and therefore extreme contrasts make finding a global threshold difficult to impossible. A pixel-wise classification of the frame could solve both problems elegantly.

We implemented a small neural network similar to U-Net [29] with only two interconnections and separable convolutions as described in Table 9.1. We conclude that a semantic segmentation neural network capable of running on the NAO robot and yielding acceptable results can have a maximum size of about $120\,\text{px} \times 88\,\text{px}$. The inference of such a network can be performed in about $14\,\text{ms}$, which would be quite a lot for a plain field segmentation. Nonetheless, it is to be expected that the network is capable of differentiating between complex features, which is why we decided to make the same network predict multiple classes, among others whether a pixel belongs to a robot. This works reasonably well and precisely. An example of a segmentation can be seen in Fig. 9.2.

We trained our model with data generated in UERoboCup [9] and augmented that data with image-to-image translation methods.

A major disadvantage of semantic segmentation is the mandatory post processing. Since our image processing pipeline depends on a color segmentation, we can exploit the segmentation generated from the neural network by feeding it to the obstacle perceptor described in [24]. This is possible because the old obstacle perceptor mainly uses the pixels classified as white. Since the pixels classified as robot from the neural network segmentation is a subset of such

[1] `https://www.youtube.com/watch?v=uolQ2xY328U`

Layer	Scale	N	F
Input	1	1	3
3×3-SConv2D, BN, LeakyReLU	1	1	8
3×3-SConv2D, BN, LeakyReLU	1	1	8
3×3-SConv2D, BN, LeakyReLU	1/2	1	8
3×3-SConv2D, BN, LeakyReLU	1/2	2	16
3×3-SConv2D, BN, LeakyReLU	1/4	1	16
3×3-SConv2D, BN, LeakyReLU	1/4	6	24
UpSampling2D	1/2	1	24
Concat	1/2	1	40
3×3-SConv2D, BN, LeakyReLU	1/2	3	16
UpSampling2D	1	1	16
Concat	1	1	24
3×3-SConv2D, BN, LeakyReLU	1	3	8
3×3-SConv2D, BN, LeakyReLU	1	1	5

Table 9.1: Segmentation network architecture. N denotes the number of duplications of the layer, F denotes the amount of channels after that layer.

pixels, the obstacle perceptor can easily be used to convert the pixel classification from the segmentation to robot percepts.

9.1.2 State Estimation

While the observers run the default modeling modules as described in Chapter 5 (apart from the assumed initial position in the `SelfLocator`), the player needs a completely different state estimation for its pose on the field as well as for the location of the ball, accounting for the completely different observation models in this challenge. The self localization for the player uses the same principle as the normal `SelfLocator`, i. e. it is a particle filter where each hypothesis is an Unscented Kalman Filter. The motion update, resampling and best sample selection steps are exactly the same, but it does not feature any sensor resetting except for the initialization. As long as the player is penalized, it deletes its particle set. Without a sample set, it reports a pose validity of 0 to indicate the absence of information to the behavior. The initialization of the sample set happens as soon as one of the observers sees a robot inside the center circle

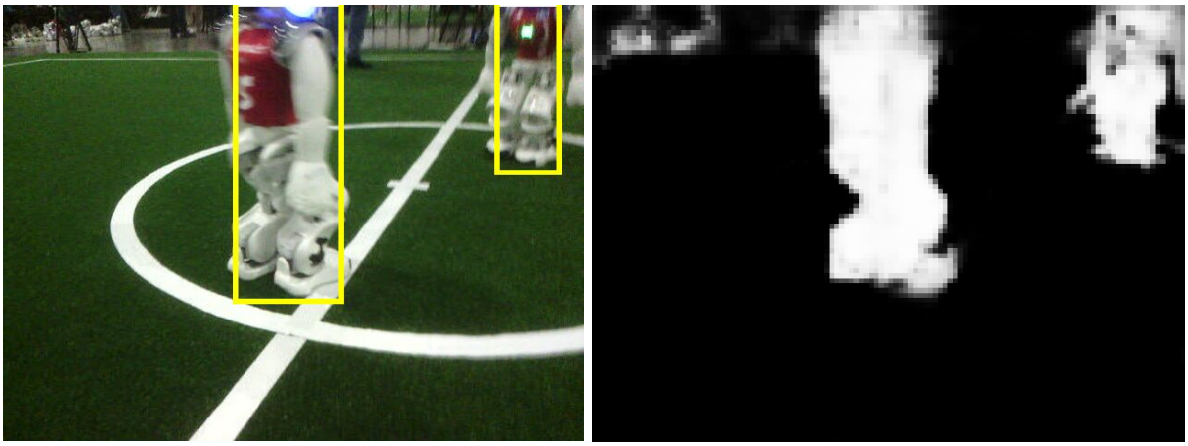


Figure 9.2: Original image with robot percepts (left) and robot class segmentation (right).

(i. e. where the player is assumed to be initially) while already not too far from it. A set of 12 particles is generated at the observed position with rotations covering the entire angular space in 30° steps and a fixed initial variance.

A sensor update happens every time there is a new team message from an observer. For each observer independently, an observation is selected from its `ObstacleModel`. The obstacle closest to the previous pose estimate that is not too close to the pose of the other observer is used. Then, for each combination of sample and observation, the distance between the position of the sample and the observed position is calculated. If this distance is small enough to assume that the observed obstacle is indeed the player, the position measurement is applied as an update to the sample's UKF. Additionally, whether having received the update or not, the validity of each sample is updated inversely to its association distance, i. e. a small deviation supports a high validity of the sample while a large deviation indicates that the sample might be wrong. As the player moves, this decreases the validity of samples with wrong initial rotation estimates as the paths diverge, letting the sample set quickly converge around the correct position and rotation.

The ball model of the player does no filtering on its own. It copies the most recent available ball model of an observer that can be related to an obstacle measurement of the player. Under the assumption that the difference in robot pose rotations of the observer to the player is correct, the ball state is transformed from the observer's coordinate system into the player's using the relative position of the player obstacle to the observer. This should make the ball model more precise relative to the player than just transforming it using the estimated player pose. A motion update, i. e. application of the inverse odometry offset and the motion of the ball, is performed each step to bridge time intervals without new updates.

9.1.3 Behavior

The behavior is kept as simple as possible. The observers do nothing else than tracking the player from both sides and keeping a minimum distance to it. Since the orientation from the player has to be estimated from its walking direction and initially no orientation is known, the player will move forwards and backwards until it is certain about its orientation. It will then walk forward while the observers track it until the observers see the ball. The observers will start looking actively for the ball as soon as they are close to the goal. When one of the observers sees the ball, the player will walk toward that position while the observers keep a safe distance to the player. Lastly, the player attempts to score a goal. The procedure can be repeated as often as desired.

9.2 Directional Whistle Challenge

The second technical challenge in 2019 was the *Directional Whistle Challenge*. This challenge intended to investigate the possibilities of localizing the point where the referee whistle is blown. In this section, we would like to share our approach to localizing the whistle and point out some of the problems we came across.

9.2.1 Constraints

With the given hardware platform, the NAO V6, the whistle localization proved to be a challenging task for all participating teams. This was mainly due to the inherently complex nature

of sound propagation as well as the NAO V6 not being designed for localizing high-frequency sound sources like whistles.

The NAO V6 is equipped with four omnidirectional microphones which are located inside the head with little holes to let the outside sound reach the microphones mostly undamped. They are arranged in a trapezoid-like shape with the two microphones in the front being higher and closer together and the two microphones in the back being lower and further apart (cf. Fig. 9.3).

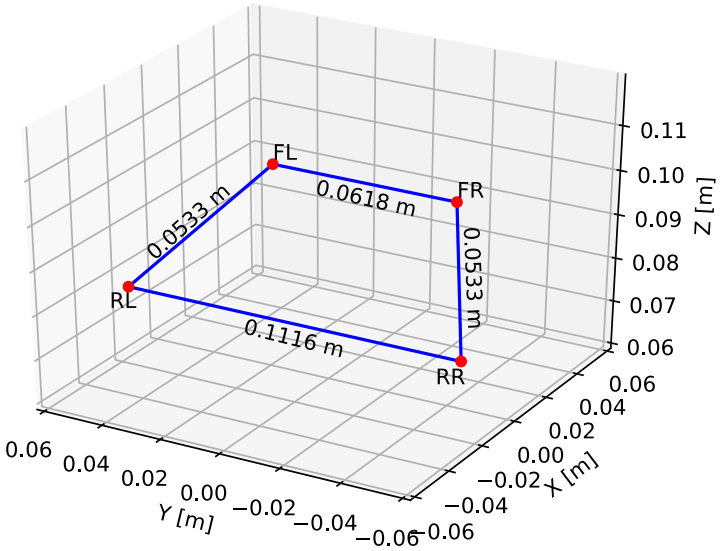


Figure 9.3: The arrangement of the 4 microphones (Front-Left, Front-Right, Rear-Left, Rear-Right) with the respective distances between them.

In order to simplify the problem, we assume that the robot's head is tilted to the maximum front position, effectively moving the microphone plane parallel to the XY-plane. Furthermore, we are not considering the z-coordinate of the sound source (i. e. the height of the person blowing the whistle). This decision was made because of the coplanar microphone arrangement on the NAO V6. It is proven that to reconstruct a 3-D point by comparing the time differences the sound arrives at the microphones, at least four non-coplanar microphones are needed [1]. Unfortunately, this *will* have negative effects on the direction of arrival (DOA) estimation when the whistle is blown close to the robot because of the neglected increased vertical angle the sound arrives from.

Each microphone records sound at $16\,\text{bit}$ with a sample frequency of $48\,\text{kHz}$. The latter plays a vital role in localizing the whistle as at lower rates, the displacement of samples across channels in time is quantized increasingly coarse.

Depending on the DOA and the frequency of the sound source, acoustic shadowing can occur to the microphones further away from the source. This can impact the amplitudes of the signal making any possible cross-correlations prone to error if they are not normalized beforehand.

9.2.2 Algorithmic Overview

In order to find the position of the sound source, each robot is individually estimating a DOA. To do that we have applied a delay-and-sum beamformer to estimate the respective DOA of three microphone pairs (RL - FL, FL - FR, FR - RR). Each of those microphone pairs calculates a beam signal for each angle between $-90°$ and $+90°$ with a step size of $1°$. Because of the

isotropic microphones, those angles are simply mirrored for the other side leaving us with 360 beam signals. These 360 signals are then transformed to be relative to the head direction. This step is necessary because of the rotated microphone pairs RL - FL and FR - RR. If it weren't for the coplanar characteristic of the microphone array there should also be a fix for the translation difference of the microphone pairs at this point. This is important for close range sound sources since small variations in the position of the sound source lead to high variations in the estimated DOA.

Due to reasons explained below, the beam signals of each microphone pair contain at least 2 types of ambiguity: *spatial aliasing* and *mirroring*. This effectively leaves every microphone pair with at least 2 (up to 6) possible DOAs. To reduce the effect of these ambiguities the signals are averaged creating an energy-weighted DOA histogram containing all 360 directions. This way, directions where many microphone pairs contain possible DOAs get a high energy value. The direction with the maximum energy value is then presented as the final DOA for the analyzing robot.

In theory, all 5 robots will estimate a local DOA this way and send the result to all other robots. The first robot to receive 4 DOAs (not including its own) is going to calculate an intersection of those DOAs taking into account the poses of the other robots. This is done by applying a least squares optimizer and finding the lowest error on the intersection of all 5 DOAs.

9.2.3 DOA Estimation in Detail

For the sake of explanation, we assume two things to be constant: the base frequency of the whistle (3200 Hz) and the speed of sound (343 $\frac{\text{m}}{\text{s}}$). In reality, they depend on the type of whistle used, and the air temperature and humidity, respectively. Furthermore, we are assuming that the sound source is not close (< 30 cm) as we would otherwise have to include the curvature of the sound waves into our calculations.

To estimate a local DOA we take advantage of the fact that the sound arrives faster at one microphone than another (cf. Fig. 9.4). The resulting delay in the two audio streams coming from our two microphones contains the information we need in order to calculate the DOA.

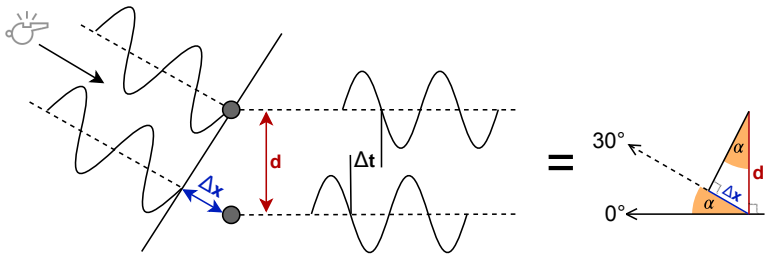


Figure 9.4: The two microphones receive the whistle at different times resulting in a time delay.

Usually, we could run a normalized cross-correlation, get the delay and calculate the DOA α via:

$$\Delta t = \frac{\Delta x}{343\ \frac{\text{m}}{\text{s}}}$$

$$\alpha = \sin^{-1}\left(\frac{\Delta t \cdot 343\ \frac{\text{m}}{\text{s}}}{d}\right)$$

Unfortunately, this is not possible for most of our microphone pairs in all possible permutations because of *spatial aliasing*. Similar to the Nyquist frequency in temporal aliasing, spatial

aliasing occurs if the microphones are further apart than half the wavelength of the highest frequency to measure [3]. The highest frequency we are interested in is $\approx 3200\,\text{Hz}$, leaving us with a maximum distance between microphones of

$$\frac{343\,\frac{\text{m}}{\text{s}}}{2 \cdot 3200\,\text{Hz}} = 0.0536\,\text{m}.$$

Not conforming to this distance results in the appearance of so called *grating lobes* in our directivity diagram. These are a special form of side lobes which have the same amplitude as our main lobe, thus introducing additional ambiguities in our DOA response (cf. Fig. 9.5 right). Table 9.2 shows a comparison of our four eligible microphone pairs.

Microphone pair	Distance [m]	max. Frequency [Hz]	Grating lobe angle [±deg] @ 3200 Hz
RL - FL	0.0533	3217	/
FL - FR	0.0618	2775	60.14
FR - RR	0.0533	3217	/
RR - RL	0.1116	1536	28.70

Table 9.2: A comparison of the four outer microphone pairs

This makes it immediately apparent that microphone pair RR - RL would introduce way too many ambiguities. As a result, we are only focusing on the remaining three outer pairs.

The second source of ambiguities comes from the fact that our 3 pairs resemble very small linear arrays. This means that they can not distinguish between sources coming from the front or the back (with respect to their direction). This creates a figure-8 style directivity pattern because of the same amplitude of the back and main lobe (cf. Fig. 9.5 left).

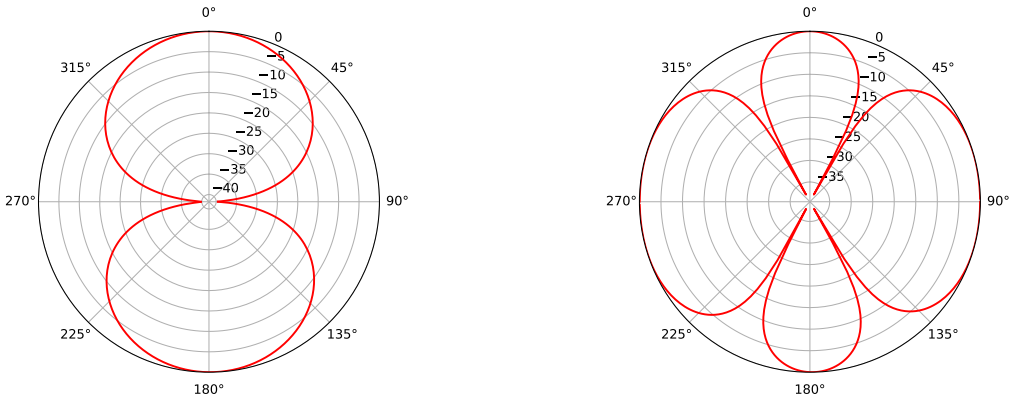


Figure 9.5: Directivity diagrams (5 dB/div) for the microphone pairs FL - RL (left) and RL - RR (right)

We have therefore employed a beamforming algorithm to generate a beam signal for 360 directions. This is done by steering the main lobe to the angle of interest and summing the two audio streams. By doing that, the sound waves coming from our steered direction get increased while other sound waves coming from other directions get canceled out. Since we can not move the microphone array itself to steer the beam, the beamformer simulates a physical steering by delaying one audio stream before summation. This delay is calculated beforehand and depends on the angle of interest. Remembering our 48 kHz sample rate, each wavelength of our 3200 Hz whistle is approximately 15 frames in our data buffer. Because our beamformer needs to calculate 180 different angles, the respective 180 delays will not be integral frame delays most of the

time. We therefore applied linear interpolation to the delayed audio stream to simulate those rational frames.

The resulting 180 angles get mirrored and averaged with the beam signals of the other microphone pairs creating an energy-weighted DOA histogram of which the maximum is the result DOA (cf. Fig. 9.6).

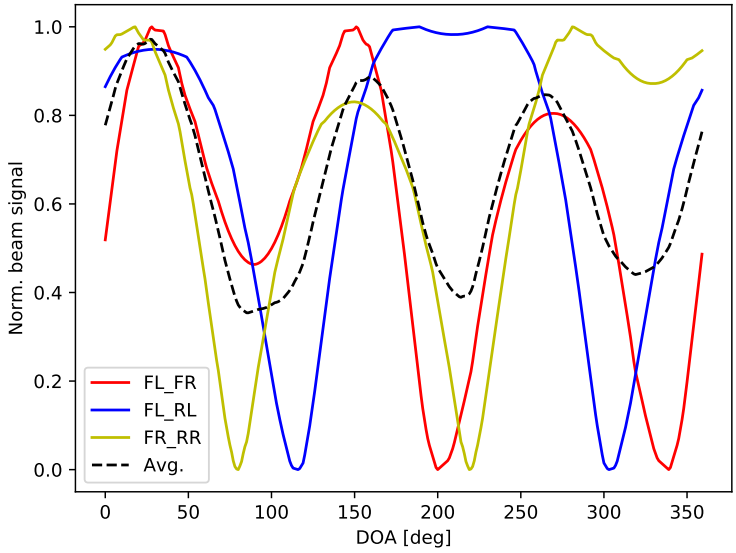


Figure 9.6: The DOA histogram of all three microphone pairs and their average. The actual DOA was 30°.

9.2.4 Accuracy

Because of the high number of possible ambiguities, the DOA histogram can become quite cluttered. This can result in false detections if the grating lobes just happen to overlap each other at specific directions. In Fig. 9.6 this is shown by the two other visible spikes in the average signal. If additional effects like reverberation come into play this can lead to a drop in accuracy. Any false detections of overlapping grating lobes of any robot greatly affect the quality of the least squares optimizer solution. With five robots this unfortunately happens more often than not. Therefore, we only used a single robot in the actual challenge, not sending any distance information and guessing one of the two possible directions $\mod 180°$. Unfortunately, we always selected the wrong one. A possible solution to this could be to try to compare the timestamps of the beginning of the whistle on each robot to deduce a more general direction. This information could then be used to exclude some of the false positives.

9.3 *B&B* in the Mixed Team Tournament

In 2019, the *Mixed Team Tournament* was held for the third time. After two successful participations in 2017 (together with the *HULKs* as the team *B-HULKs*) and 2018 (together with *rUNSWift* as team *B-Swift*), we again took the opportunity to cooperate with a different team as this always leads to a worthwhile exchange of ideas. Thus, for this year's competition, we

Figure 9.7: The human team members of *B&B* who participated in RoboCup 2019 in Sydney

teamed up with *Berlin United – Nao Team Humboldt* from the Humboldt University of Berlin, who have participated in the Standard Platform League since the very beginning. The name of our mixed team is *B&B*[2], most team members are shown in Fig. 9.7.

For the previous competitions, we evaluated two very different approaches for forming a mixed team. Thanks to their geographical proximity, the *B-HULKs* were able to meet quite often and to agree on a sophisticated strategy as well as on a shared communication protocol beyond what is mandated by the SPL standard message (details are described in [25]). This was hardly possible for *B-Swift* in 2018. Thus, the team only used the SPL standard message and made a few basic agreements about role selection. Nevertheless, both approaches were successful. The *B&B* team generally took the latter direction, i. e. employing mostly pre-coordination rather than online communication. A significant difference to *B-Swift* is that *B&B*, apart from the fact that the goalkeeper was always a B-Human robot, did not assign certain players to fixed parts of the field, i. e. both B-Human and Berlin United robots could be in the offense as well as defense. Therefore, a small header in the team message was defined at a meeting in Berlin. It contains whether the robot is penalized (to ease testing without a GameController), whether it wants to play the ball, and a number corresponding to the role of the robot. The striker was determined based on a common formula for the time to reach the ball, such that it can be calculated by all robots for themselves and all teammates.

The Mixed Team Tournament at RoboCup 2019 consisted of a preliminary round in which all five participating teams played games against each other team. The two best teams proceeded to the final. We clearly won three of our preliminary round games and played one draw against *Team Team*, which was formed by *Nao-Team HTWK* from Leipzig and the Bembelbots from

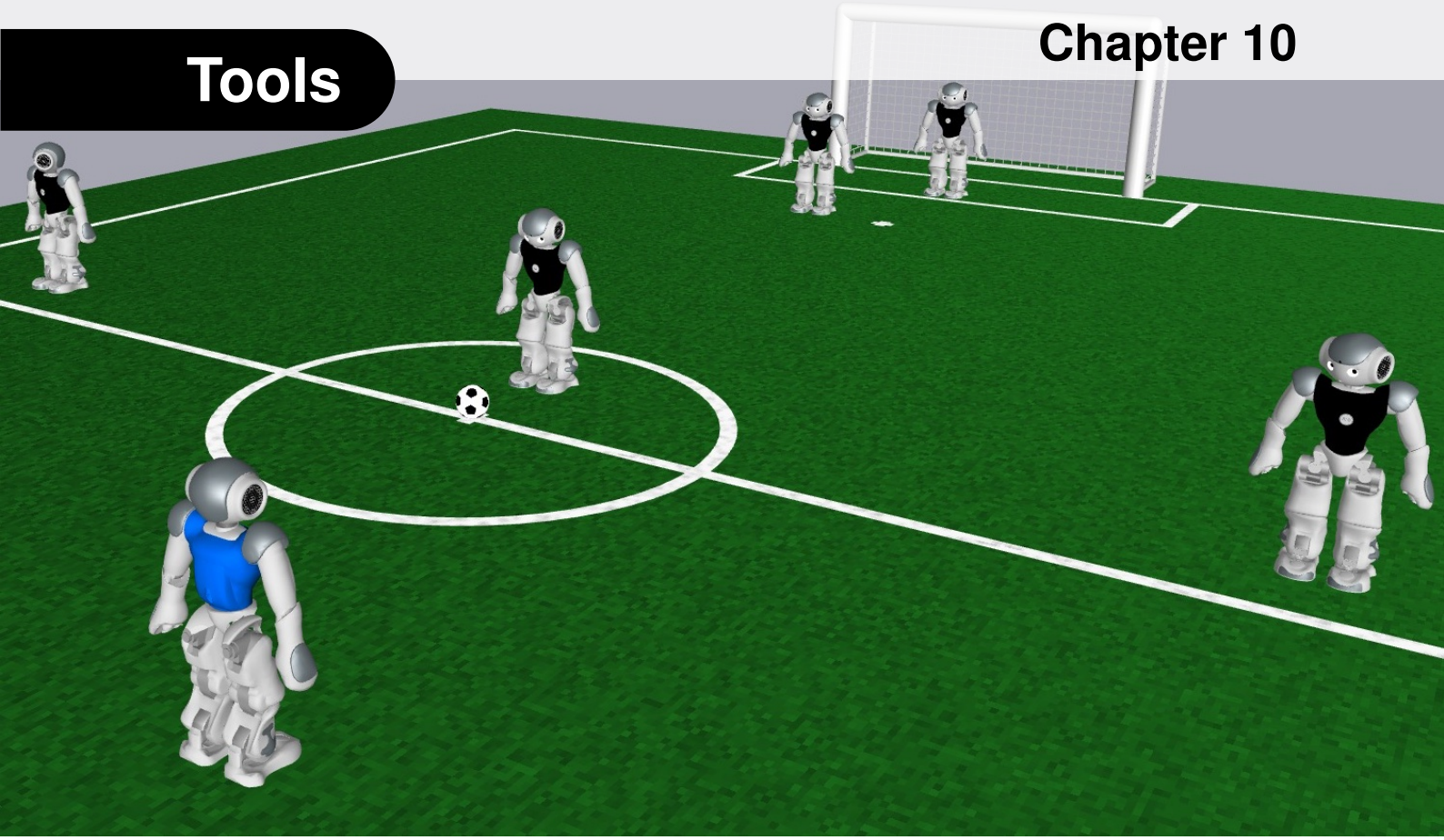
[2] It remains up to the reader to decide which *B* refers to which team.

Frankfurt. As they also won their other games, we met them again in the final. In that game, the equal strength of both mixed teams became obvious: the final game as well as the following penalty shootout ended in a draw, too. Thus, it needed a sudden death shoot-out, in which we finally won. An overview of all results is given at the league's web site at `https://spl.robocup.org/results-2019/`.

As both parts of *B&B* 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 our intended strategy. We are very happy about our success and thank *Berlin United – Nao Team Humboldt* for playing with us!

Chapter 10

Tools



The following chapter describes B-Human's simulator, SimRobot, as well as the B-Human User Shell (*bush*), which is used during games to deploy the code and the settings to several NAO robots at once.

10.1 SimRobot

The B-Human software package uses the physical robotics simulator SimRobot [14, 12] as front end for software development. The simulator is not only used for working with simulated robots, but it also functions as graphical user interface for replaying log files and connecting to physical robots via Ethernet or Wi-Fi.

10.1.1 Architecture

Three dynamic libraries are created when SimRobot is built. These are *SimulatedNao*, *SimRobotCore2* and *SimRobotEditor* (cf. Fig. 10.1)[1].

SimRobotCore2 is the simulation core. It is the most important part of the SimRobot application, because it models the robots and the environment, simulates sensor readings, and executes commands given by the controller or the user. The core is platform-independent and it is connected to a user interface and a controller via a well-defined interface.

The library *SimulatedNao* is in fact the controller that consists of the two projects *SimulatedNao*

[1] The actual names of the libraries have platform-dependent prefixes and suffixes, i. e. *.dll*, *.dylib*, and *lib .so*.

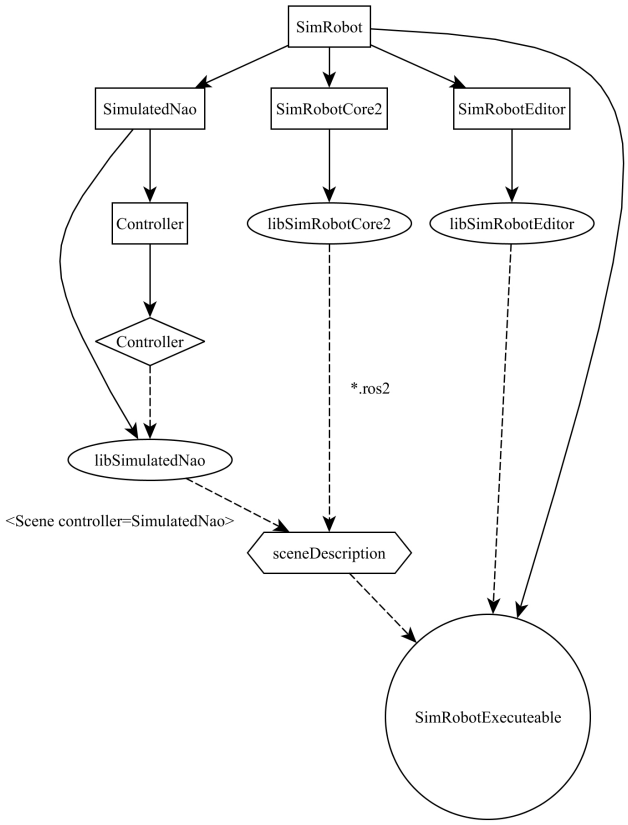


Figure 10.1: This figure shows the most important libraries of *SimRobot* (excluding third-party libraries). The rectangles represent the projects which create the appropriate library. Dynamic libraries are represented by ellipses and the static one by a diamond. Note: The static library will be linked together with the *SimulatedNao* code. The result is the library *SimulatedNao*.

and *Controller*. *SimulatedNao* creates the code for the simulated robot. This code is linked together with the code created by the *Controller* project to the *SimulatedNao* library. In the scene files, this library is referenced by the `controller` attribute within the `Scene` element.

10.1.2 B-Human Toolbar

The B-Human toolbar is part of the general SimRobot toolbar which can be found at the top of the application window (cf. Fig. 10.2). It allows setting the representations `MotionRequest` and `HeadAngleRequest` to frequently used values.


Lets the robot stand up.


Lets the robot sit down.


Allows moving the robot's head by hand.

10.1.3 Scene View

The scene view (cf. the central view in Fig. 10.3) appears if the *scene* is opened from the scene graph, e. g. by double-clicking on the entry *RoboCup*. The view can be translated by dragging, rotated by dragging with the *Shift* key pressed, zoomed, and it supports several mouse operations on objects:

- Left-clicking an object allows dragging it to another position. Robots and the ball can be moved in that way.
- Left-clicking while pressing the *Shift* key allows rotating objects around their centers. The axes of the rotating objects can be selected first by pressing *x*, *y* or *z* key, before pressing and holding the *Shift* key.
- Double-clicking an *active* robot selects it. Robots are active if they are defined in the compound *robots* in the scene description file (cf. Section 10.1.5). Robot console commands are sent to the selected robot only (see also the command *robot*).

10.1.4 Information Views

In the simulator, *information views* are used to display debugging output such as debug drawings. Such output is generated by the robot control program, and it is sent to the simulator via *message queues* (Section 3.5.2). The views are interactively created using the console window, or they are defined in a script file. Since SimRobot is able to simulate more than a single robot, the views are instantiated separately for each robot. There are 14 kinds of information views, which are structured here into the five categories *cognition*, *behavior control*, *sensing*, *motion control*, and *general debugging support*. All information views can be selected from the scene graph (cf. the left widget in Fig. 10.3).

10.1.4.1 Cognition

Image Views

An image view (cf. Fig. 10.4a) displays debug information in the coordinate system of the camera image. It is created using the command *vi* (cf. Section 10.1.6.3). The background of an image view can either be empty, a regular camera image (optionally JPEG-compressed to reduce the bandwidth needed), or any debug image sent using the `SEND_DEBUG_IMAGE` macro. Additionally, the image can be color-classified in the simulator. Additionally, all image views created by the standard startup scripts have the feature that clicking inside them while holding the *Shift* key moves the robot's head to make the clicked point the center of the image. The console command *vid* adds debug drawings to an image view. For instance, a view of the segmented lower image with detected ball, lines and obstacles can be created with the following commands:

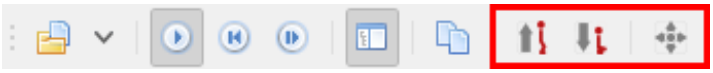


Figure 10.2: This figure shows the three buttons from the *BHToolBar*.

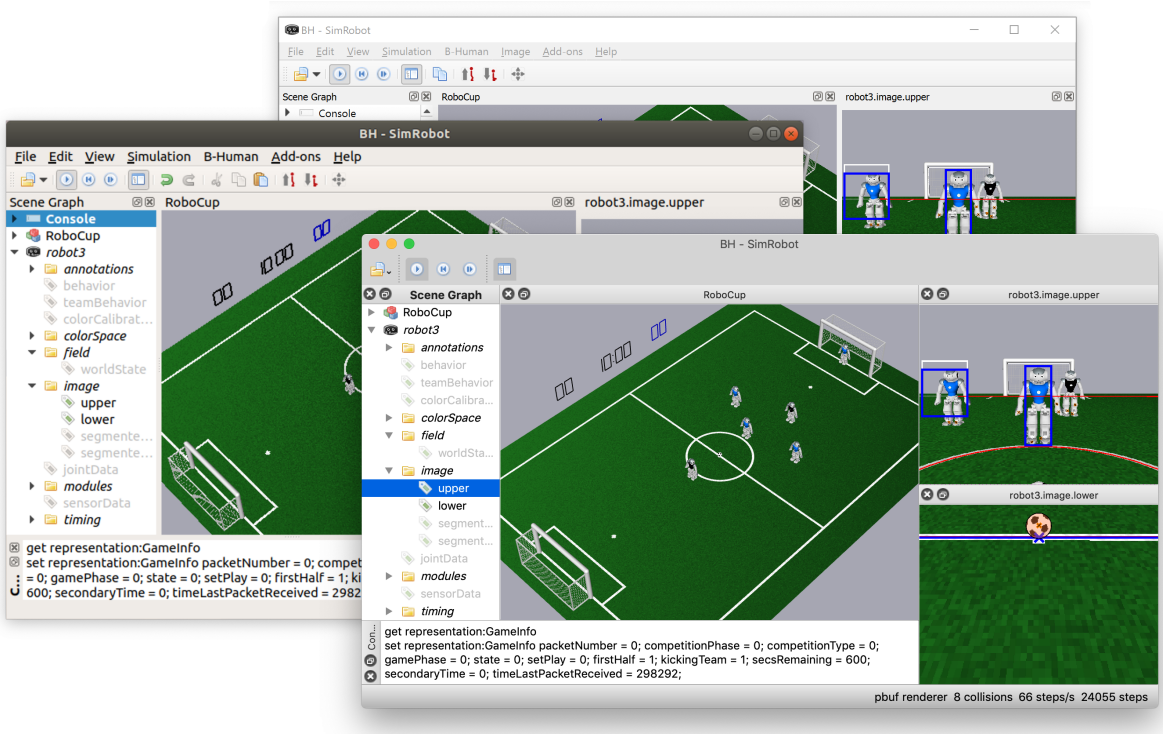


Figure 10.3: SimRobot running on Windows, Linux, and macOS. The left pane shows the scene graph, the center pane shows a scene view, and on the right there are two views showing the images of both cameras. The console window is shown at the bottom.

```
vi image segmented
vid segmentedLower representation:BallPercept:image
vid segmentedLower representation:LinesPercept:image
vid segmentedLower representation:ObstaclesImagePercept:image
```

To remove a debug drawing from a view, the *off* option can be appended to a *vid* command.

Color Calibration View

The color calibration view provides direct access to the parameter settings for the four color thresholds. All parameters can be changed by moving the sliders. Color classes are defined in the YHS2 color space as described in Section 4.1.5.

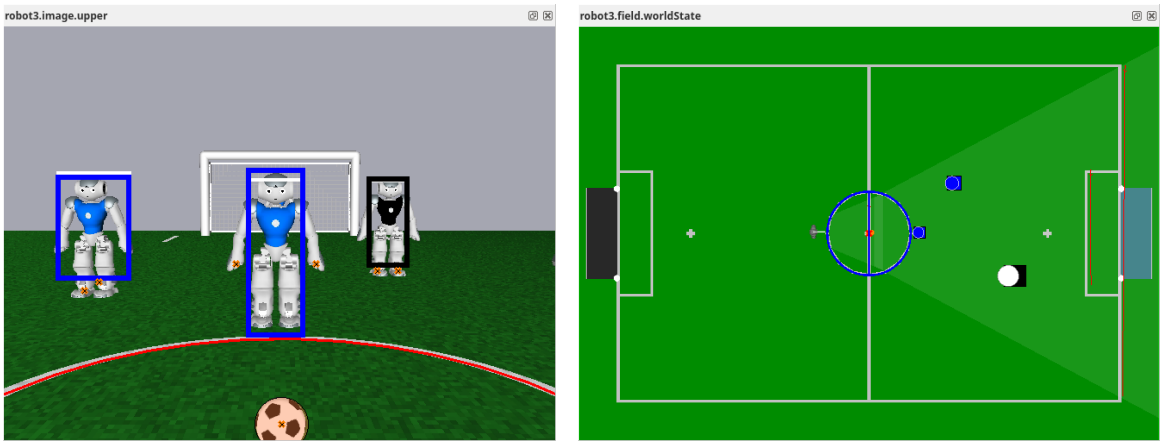
The following buttons are added to the toolbar when the *colorCalibration* view is focused.


undoes the latest slider change


saves the local color configuration

redoes a reverted slider change

Figure 10.5 shows an example calibration for the color delimiter, the field color, and the black-white delimiter. The color delimiter defines the minimum saturation of a pixel necessary to be classified as colored. *Field* defines the hue range for pixels being classified as field color if they are colored. The black-white delimiter defines the minimum brightness for non-colored pixels to be classified as white instead of black.



(a) (b)

Figure 10.4: Image view and field view with several debug drawings

Figure 10.5: An example calibration in the color calibration view

Color Space Views

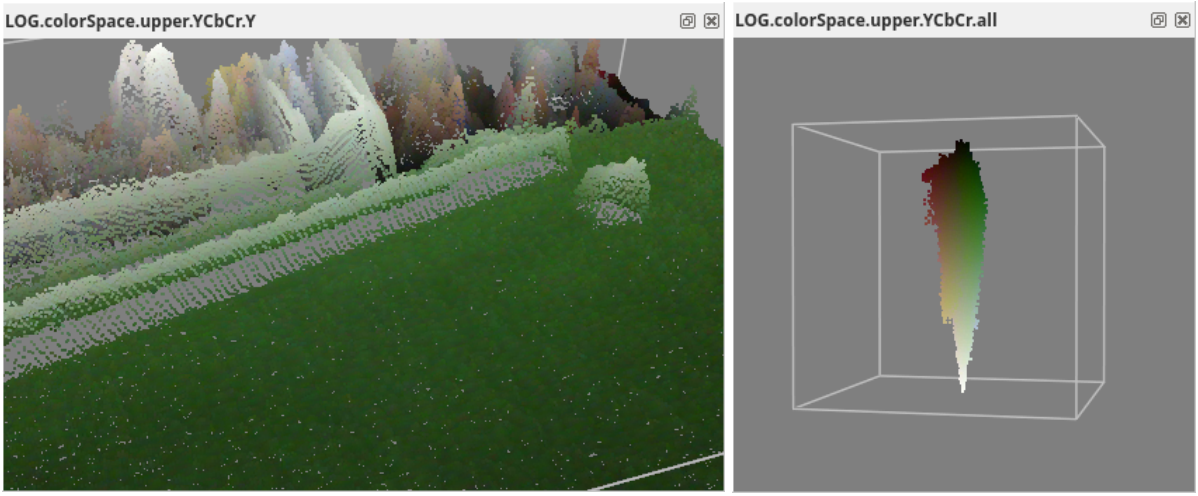
Color space views visualize image information in 3-D (cf. Fig. 10.6). They can be rotated by clicking into them with the left mouse button and dragging the mouse afterwards. There are two kinds of color space views:

Image Color Channel: This view displays an image while using a certain color channel as height information (cf. Fig. 10.6a).

Image Color Space: This view displays the distribution of all pixels of an image in a certain color space (*HSI*, *RGB*, or *YCbCr*). It can be displayed by selecting the entry *all* for a certain color space in the scene graph (cf. Fig. 10.6b).

The two kinds of views can be instantiated for the camera image (which is already done in most startup scripts) or any debug image. For instance, to add a set of views for the lower camera image under the name *raw*, the following command has to be executed:

```
v3 image raw
```



(a) Image color channel view (b) Image color space view

Figure 10.6: 3-D color space views

Field Views

A field view (cf. Fig. 10.4b) displays information in the system of coordinates of the soccer field. The commands to create and manipulate it are defined similar to the ones for image views. For instance, the view *worldState* is defined as:

```
# field views
vf worldState
vfd worldState fieldLines
vfd worldState goalFrame
vfd worldState fieldPolygons
vfd worldState representation:RobotPose
# ...

# views relative to robot
vfd worldState cognition:RobotPose
vfd worldState representation:BallModel:endPosition
# ...

# back to global coordinates
vfd worldState cognition:Reset
```

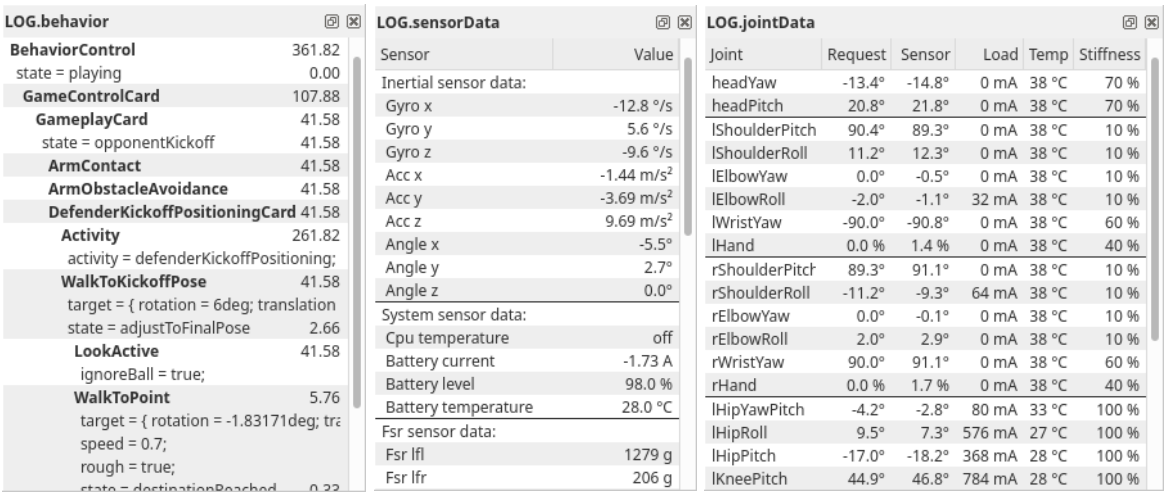
Please note that some drawings are relative to the robot rather than relative to the field. Therefore, special drawings exist that change the system of coordinates for all drawings added afterwards, until the system of coordinates is changed again.

The field can be zoomed in or out by using the *mouse wheel*, touch gestures, or the *page up/down* buttons. It can also be dragged around with the left mouse button or by touch gestures. Double clicking the view resets it to its initial position and scale.

10.1.4.2 Behavior Control

Behavior View

The *behavior view* (cf. Fig. 10.7a) shows all currently active cards and skills and the states they are in. Since the behavior consists of two passes (cf. Section 6.2), there are two instances of



(a) (b) (c)

Figure 10.7: Behavior view, sensor data view and joint data view

this view: once for the team behavior and once for the individual behavior. To be able to display both of them, the following debug requests must be sent (which are, however, already part of all startup scripts):

```
dr representation:ActivationGraph
dr representation:TeamActivationGraph
```

10.1.4.3 Sensing

Sensor Data View

The sensor data view displays all the sensor data taken by the robot that is not joint-related, e. g. accelerations, gyro measurements and foot pressure readings (cf. Fig. 10.7b). To display this information, the following debug requests must be sent (which are, however, already part of all startup scripts):

```
dr representation:InertialSensorData
dr representation:SystemSensorData
dr representation:FsrSensorData
dr representation:KeyStates
```

Joint Data View

Similar to sensor data view, the joint data view displays all the joint data taken by the robot, e. g. requested and measured joint angles, temperatures, and loads (cf. Fig. 10.7c). To display this information, the following debug requests must be sent (which are, however, already part of all startup scripts):

```
dr representation:JointRequest
dr representation:JointSensorData
```

10.1.4.4 Motion Control

Kick View

The idea of the kick view shown in Fig. 10.8 is to visualize and edit basic configurations of motions for the `KickEngine` described in [17]. To do so, the central element of this view is a 3-D robot model. Regardless of whether the controller is connected to a simulated or a real robot, this model represents the current robot posture.

Since the `KickEngine` describes motions as a set of Bézier curves, the movement of the limbs can easily be visualized. Thereby the sets of curves of each limb are represented by combined cubic Bézier curves. Those curves are attached to the 3-D robot model with the robot center as origin. They are painted into the three-dimensional space. Each curve is defined by Eq. (10.1):

$$c(t) = \sum_{i=0}^{n} B_{i,n}(t) P_i \tag{10.1}$$

To represent as many aspects as possible, the kick view has several sub views:

3-D View: In this view each combined curve of each limb is directly attached to the robot model and therefore painted into the 3-dimensional space. Since it is useful to observe the motion curves from different angles, the view angle can be rotated by clicking with the *left mouse button* into the free space and dragging it in one direction while holding the *Shift* key. In order to inspect more or less details of a motion, the view can also be zoomed in or out by using the *mouse wheel* or the *page up/down* buttons. The *left/right* arrow keys can translate the robot sidewards.

A motion configuration is not only visualized by this view, it is also editable. The user can click on one of the visualized control points (cf. Fig. 10.8 at *E*) and drag it to the desired position. In order to visualize the current dragging plane, a light green area (cf. Fig. 10.8 at *B*) is displayed during the dragging process. This area displays the mapping between the screen and the model coordinates and can be adjusted by using the *right mouse button* and choosing the desired axis configuration.

Another feature of this view is the ability to display unreachable parts of motion curves. Since a motion curve defines the movement of a single limb, an unreachable part is a set of points that cannot be traversed by the limb due to mechanic limitations. The unreachable parts will be clipped automatically and marked with orange color (cf. Fig. 10.8 at *C*).

1-D/2-D View: In some cases a movement only happens in the 2-dimensional or 1-dimensional space (e. g. raising a leg is a movement along the z-axis only). For that reason, more detailed sub views are required. Those views can be displayed as an overlay to the 3-D view by using the context menu, which opens by clicking with the right mouse button and choosing *Display 1D Views* or *Display 2D Views*. This only works within the left area where the control buttons are (cf. Fig. 10.8 at *D*). By clicking with the right mouse button within the 3-D view, the context menu for choosing the drag plane appears. The second way to display a sub view is by clicking at the *KickEdit* entry in the title menu. This displays the same menu which appears as context menu.

Because clarity is important, only a single curve of a single phase of a single limb can be displayed at the same time. If a curve should be displayed in the detailed views, it has to be activated. This can be done by clicking on one of the attached control points.

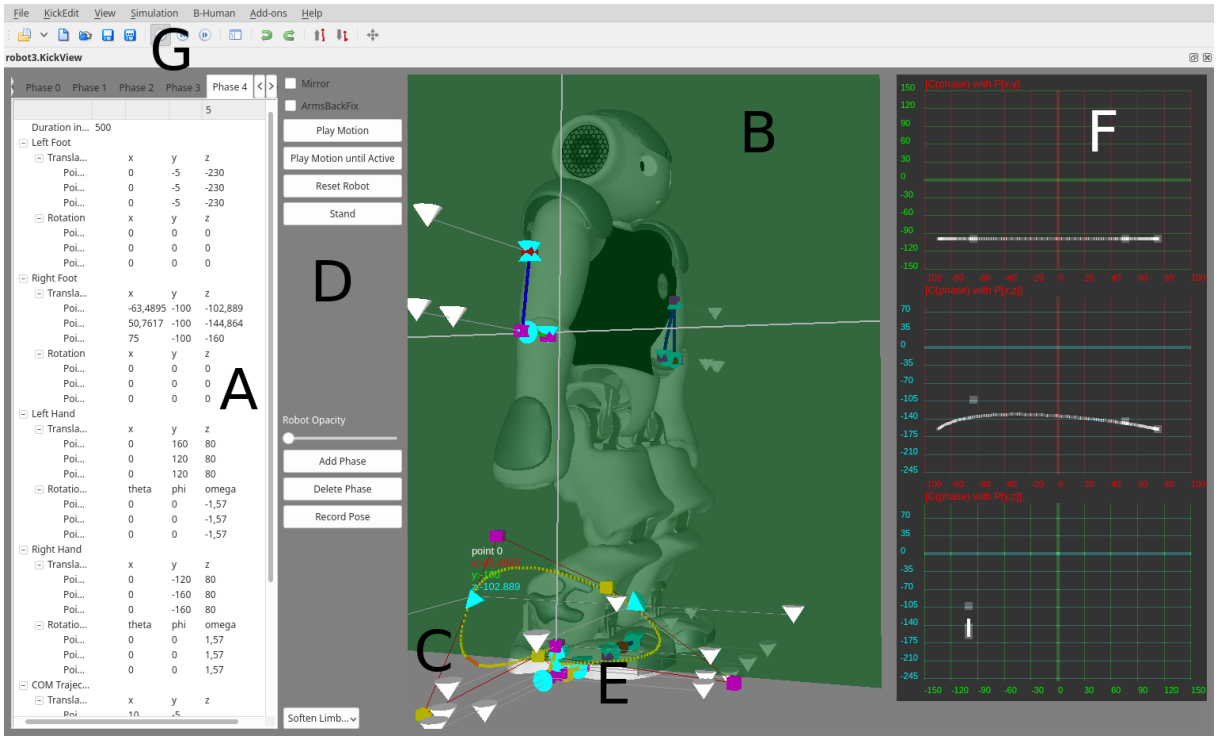


Figure 10.8: The kick view. *A* marks the editor view, *B* denotes the drag and drop plane, *C* points at a clipped curve, *D* tags the buttons that control the 3D-View, e. g. play the motion or reset the robot to a standing position, *E* labels one of the control points, *F* points at the subviews and *G* points at the tool bar, where a motion can be saved or loaded.

The 2-D view (cf. Fig. 10.9b) is divided into three sub views. Each of these sub views represents only two dimensions of the activated curve. The curve displayed in the sub views is defined by Eq. (10.1) with $P_i = \begin{pmatrix} c_{x_i} \\ c_{y_i} \end{pmatrix}$, $P_i = \begin{pmatrix} c_{x_i} \\ c_{z_i} \end{pmatrix}$ or $P_i = \begin{pmatrix} c_{y_i} \\ c_{z_i} \end{pmatrix}$.

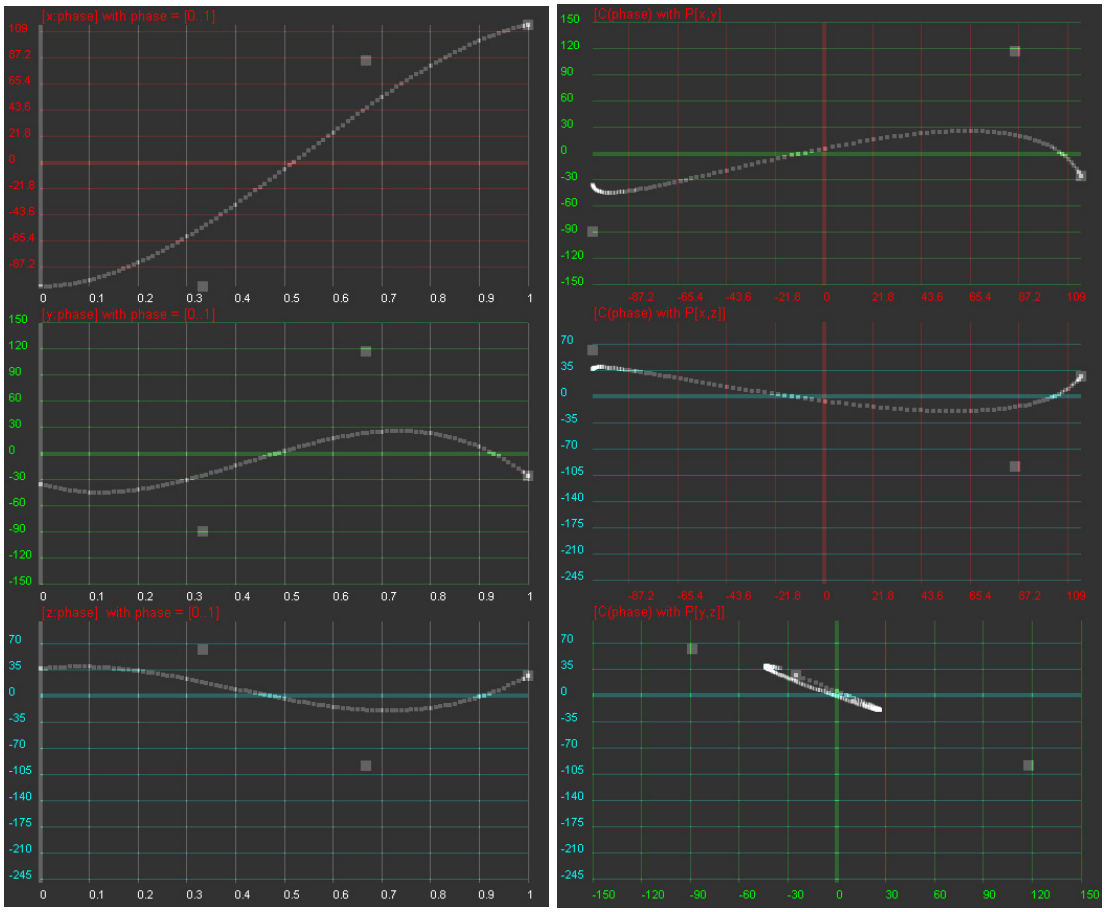
The 1-D sub views (cf. Fig. 10.9a) are basically structured as the 2-D sub views. The difference is that each single sub view displays the relation between one dimension of the activated curve and the time t. That means that in Eq. (10.1) P_i is defined as: $P_i = c_{x_i}$, $P_i = c_{y_i}$, or $P_i = c_{z_i}$.

As in the 3-D view, the user can edit a displayed curve directly in any sub view by drag and drop.

Editor Sub View: The purpose of this view is to constitute the connection between the real structure of the configuration files and the graphical interface. For that reason, this view is responsible for all file operations (for example open, close, and save). It represents loaded data in a tabbed view, where each phase is represented in a tab and the common parameters in another one.

By means of this view the user is able to change certain values directly without using drag and drop. Values directly changed will trigger repainting the 3-D view, and therefore, changes will be visualized immediately. This view also allows phases to be reordered by drag and drop, to add new phases, or to delete phases.

To save or load a motion the kick view has to be the active view. Appropriate buttons will appear in the tool bar (cf. Fig. 10.8 at *G*).



(a) (b)

Figure 10.9: 1-D sub views (left) and 2-D sub views (right)

10.1.4.5 General Debugging Support

Module Views

Since all the information about the current module configuration can be requested from the robot control program, it is possible to generate a visual representation automatically. The graphs, such as the one that is shown in Fig. 10.10, are generated by the program *dot* from the *Graphviz* package [6]. Modules are displayed as yellow rectangles and representations are shown as blue ellipses. Representations that are received from another thread are displayed in orange and have a dashed border. If they are missing completely due to a wrong module configuration or because they are alias representations, both label and border are displayed in red. The modules are separated into the categories that were specified as the second parameter of the macro `MAKE_MODULE` when they were defined. There are module views for the categories *infrastructure*, *perception*, *communication*, *modeling*, *behaviorControl*, *sensing*, and *motionControl*.

The module graph can be zoomed in or out by using the *mouse wheel*, touch gestures, or the *page up/down* buttons. It can also be dragged around with the left mouse button or by touch gestures.

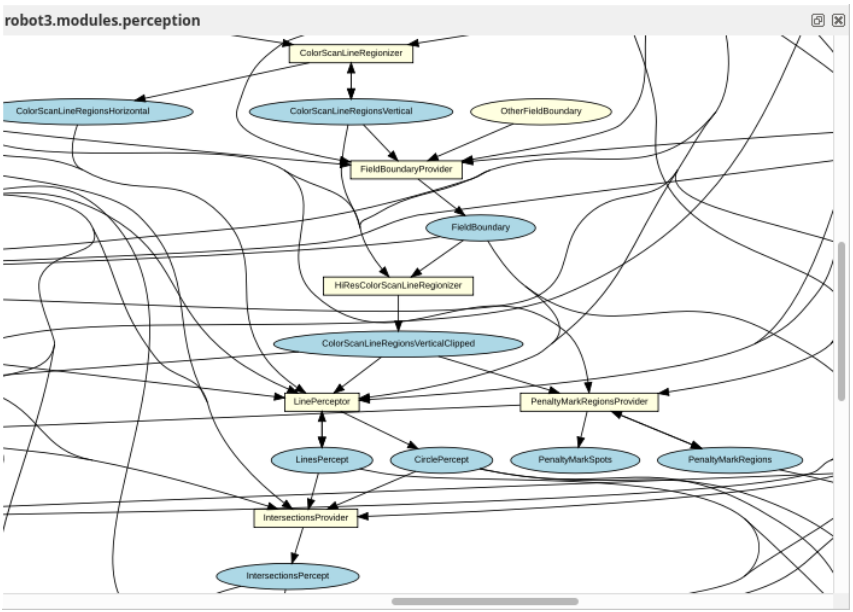
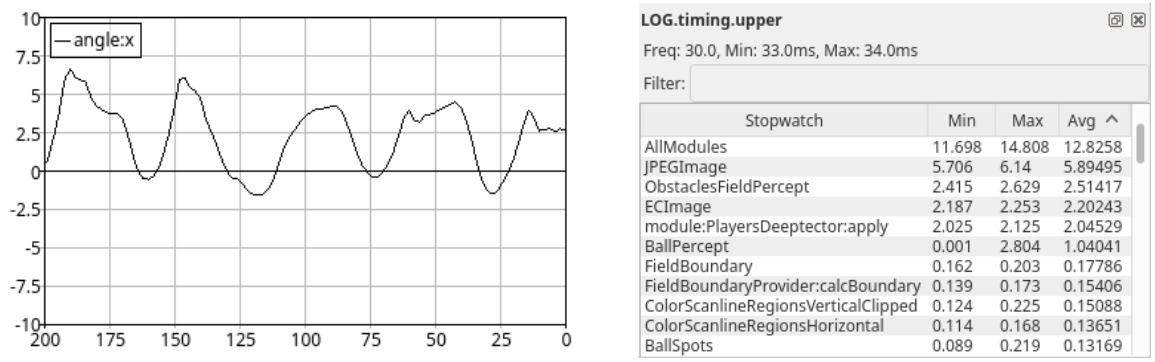


Figure 10.10: The module view shows a part of the modules in the category *perception*.



(a) (b)

Figure 10.11: Plot view and timing view

Plot Views

Plot views allow plotting data sent from the robot control program through the macro `PLOT` (cf. Fig. 10.11a). They keep a history of the values sent, up to a defined size. Several plots can be displayed in the same plot view in different colors. A plot view is defined by giving it a name using the console command *vp* and by adding plots to the view using the command *vpd* (cf. Section 10.1.6.3).

For instance, the view in Fig. 10.11a was defined as:

```
vp orientationX 200 -10 10
vpd orientationX representation:InertialData:angle:x
```

Timing View

The timing view displays statistics about all currently active stopwatches in a thread (cf. Fig. 10.11b). It shows the minimum, maximum, and average runtime of each stopwatch in

milliseconds as well as the average frequency of the thread. All statistics sum up the last 100 invocations of the stopwatch. Timing data is transferred to the PC using debug requests. By default the timing data is not sent to the PC. Execute the console command *dr timing* to activate sending timing data.

Data View

SimRobot offers two console commands (*get* & *set*) to view or edit anything that the robot exposes using the `MODIFY` macro. While those commands are enough to occasionally change some variables, they can become quite annoying during heavy debugging sessions.

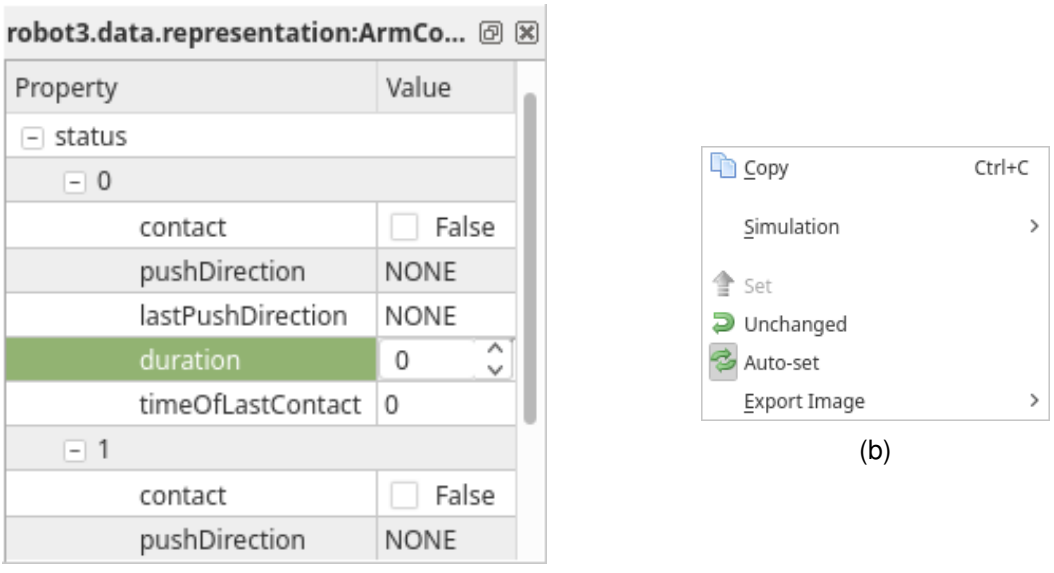
For this reason, the data view exists. It displays modifiable content using a property browser (cf. Fig. 10.12a). Property browsers are well suited for displaying hierarchical data and should be well known from various editors such as Microsoft Visual Studio or Eclipse.

A new data view is constructed using the command *vd* in SimRobot. For example, *vd representation:ArmContactModel* will create a new view displaying the `ArmContactModel`. Data views can be found in the data category of the scene graph.

The data view automatically updates itself ten times per second. Higher update rates are possible, but result in a much higher CPU usage.

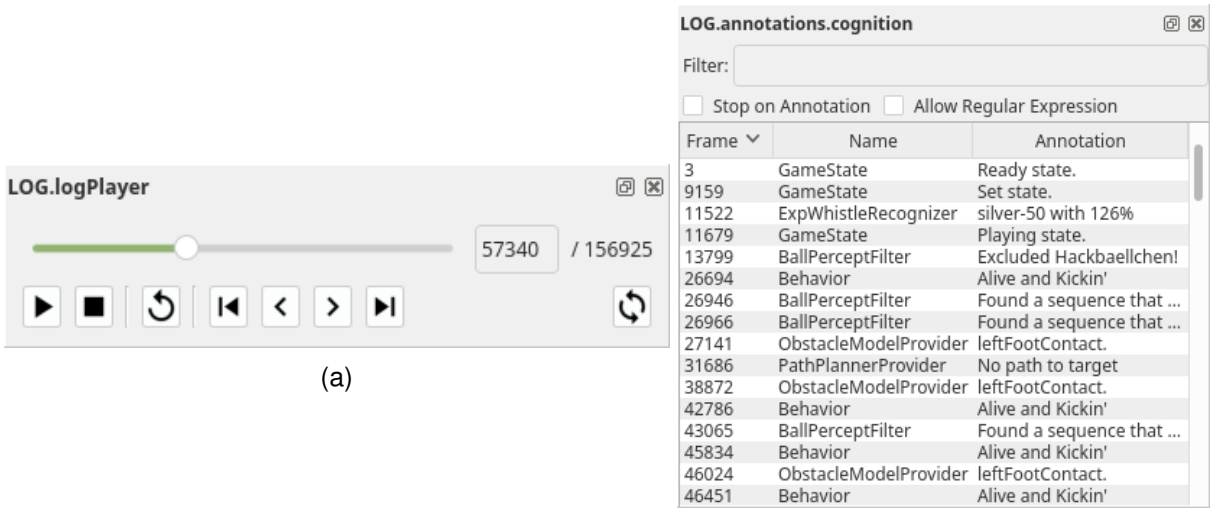
To modify data, just click on the desired field and start editing. The view will stop updating itself as soon as you start editing a field. The editing process is finished either by pressing enter or by deselecting the field. By default, modifications will be sent to the robot immediately. This feature is called the auto-set mode. It can be turned off using the context menu (cf. Fig. 10.12b). If the auto-set mode is disabled, data can be transmitted to the robot using the *set* item from the context menu.

Once the modifications are finished, the view will resume updating itself. However you may not notice this since modification continuously overwrites the data on the robot side, so the updated data are the same as entered in the data view. To stop overwriting the data, use the *unchanged*



(a) (b)

Figure 10.12: The data view can be used to remotely modify data on the robot.



(a) (b)

Figure 10.13: The log player view and the annotation view

item from the context menu. As a result, the data will be unfrozen on the robot side and you should see the data changing again.

Log Player View

The log player (cf. Fig. 10.13a) allows to control the replay of log files using the following buttons:

▶	start playback	⏮	go to the previous frame with an image
■	stops playback	‹	go to the previous frame
↻	repeat the current frame	›	go to the next frame
⟳	run in a loop	⏭	go to the next frame with an image

In addition, the current frame can be directly entered as a number or selected using a slider.

Annotation View

The annotation view displays all annotations (cf. Section 3.7.6) contained in a log file (cf. Fig. 10.13b). Double clicking an annotation will cause the log file to jump to the given frame number.

It is also possible to view annotations in real-time when using the simulated NAO or a direct debug connection to a real NAO. This has to be activated by using the following debug request:

```
dr annotation
```

10.1.5 Scene Description Files

The language of scene description files is an extended version of RoSiML [14]. Scene files that use SimRobotCore2 have to end with *.ros2*, such as *BH.ros2*. In the following, the most important elements, which are necessary to add robots, dummies, and balls, are shortly described (based upon *BH.ros2*). For a more detailed documentation, see Appendix A.

<**Include href="..."**>**:** First of all the descriptions of the NAO itself, a specific setup of NAOs, the ball and the field are included. The names of include files end with *.rsi2*.

<**Scene ...**>**:** This element begins the instantiation section of the scene description. While everything outside the scene element, i. e. the files included above, only declares templates for objects, everything that follows is actually instantiated in the scene. The attributes of this element define some important properties of the scene, e. g. the controller library and the duration of each simulation step.

<**Compound name="teamColors"**>**:** This compound contains two appearances of which only the names matter. It defines the jersey colors of the two teams which the simulated GameController assumes. This is mainly important to make the jersey detection work in the simulation and does not have an effect on the actual appearance of the robots (see below).

<**Compound ref="robots"**/ >**:** This compound contains all *active* robots, i. e. robots for which will be controlled by the B-Human modules. However, each of them will require a lot of computation time. In the tag *Body*, the attribute *ref* specifies which NAO model should be used and *name* sets the robot name in the scene graph of the simulation. Legal robot names are "robot1" ... "robot12", where the first six robots are assumed to play in the first team (with player numbers 1... 6) while the other six play in the second team (again with player numbers 1... 6). The standard color of the NAO's jersey is set to black. To set it to red, use <Set name="NaoColor" value="red"> within the *Body* element.

<**Compound ref="extras"**/ >**:** This compound contains *passive* robots, i. e. robots that just stand around, but are not controlled by a program. Passive robots can be activated by moving their definition to the compound *robots* and changing the referenced model from "NaoDummy" to "Nao".

<**Compound name="balls"**>**:** This compound contains an instance of the ball.

<**Compound ref="field"**/ >**:** This element instantiates the field from the included file as part of the scene.

Predefined Scenes

A lot of scene description files can be found in *Config/Scenes*. There are two types of scene description files: the ones required to simulate one or more robots, and the ones that connect to a real robot.

Simulating multiple robots is expensive. To overcome this and enable decent frame rates, some scenes come in special variants. In scenes ending with *PerceptOracle*, no camera images are rendered and percepts are provided by the simulator. However, models, for example the `BallModel`, still have to be calculated. This is different in the scenes with the suffix *Fast*, in which even the models are replaced with ground truth data from the simulator.

BH[**PerceptOracle**|**Fast**]**:** A single robot with five dummies.

OneTeam[**PerceptOracle**|**Fast**]**:** A team of five robots against dummies.

Game[**PerceptOracle**|**Fast**]**:** A game five against five.

KickViewScene[**Remote**]**:** For working on kicks (cf. Section 10.1.4.4).

ReplayRobot: Used to replay log files (cf. Section 3.7.5).

RemoteRobot: Connects to a remote robot and displays images and data.

10.1.6 Console Commands

Console commands can either be directly typed into the console window or they can be executed from a script file. There are three different kinds of commands. The first kind will typically be used in a script file that is executed when the simulation is started. The second kind are *global commands* that change the state of the whole simulation. The third type is *robot commands* that affect currently *selected robots* only (see command *robot* to find out how to select robots).

10.1.6.1 Initialization Commands

cl <**location**>
Changes the location that is used in the configuration file search path of simulated robots. This command is special, because it only has an effect if run directly from the script that is executed when a scene is loaded. This command and the following one are always executed before any other command in the script, because the location must be set before any robot code is executed, as otherwise configuration files would have already been loaded from the default path.

cs <**scenario**> [**team1**|**team2**]
Changes the scenario that is used in the configuration file search path of simulated robots. As for the command *cl*, this command only has an effect if run directly from the script that is executed when a scene is loaded. This command and the previous one are always executed before any other command in the script, because the scenario must be set before any robot code is executed, as otherwise configuration files would have already been loaded from the default path. The optional second parameter can be used to set the scenario for only one of the two simulated teams.

sc <**name**> <**a.b.c.d**>
Starts a remote connection to a real robot. The first parameter defines the *name* that will be used for the robot. The second parameter specifies the IP address of the robot. The command will add a new robot to the list of available robots using *name*, and it adds a set of views to the scene graph. When the simulation is reset or the simulator is exited, the connection will be terminated.

sl <**name**> [<**file**>]
Replays a log file. The command will instantiate a complete set of threads and views. The threads will be fed with the contents of the log file. The first parameter of the command defines the *name* of the virtual robot. The name can be used in the *robot* command (see below), and all views of this particular virtual robot will be identified by this name in the

tree view. The second parameter specifies the path to the log file. If the path is not an absolute path, *Config/Logs* is the base directory. *.log* is the default extension of log files. It will be automatically added if no extension is given. If no file is given at all, the previously started log file is used.

When replaying a log file, the replay can be controlled by the *Log Player* (cf. Section 10.1.4.5) or the command *log* (see below). It is even possible to load a different log file during the replay.

sml <**directory**>
Replays all log files in a directory and its subdirectories, each in its own robot instance.

10.1.6.2 Global Commands

ar off | **on**
Enables or disables the automatic referee.

call <**file**> [<**file**>]
Executes a script file. A script file contains commands as specified here, one command per line. The default location for scripts is the *Config/Scenes* directory, their default extension is *.con*. If the optional script file is present, that one is executed instead.

ci off | **on** | <**fps**>
Switches the calculation of images on or off. If an image frame rate is explicitly specified instead of "on", that one is used instead of the default value of 60 Hz. The simulation of the robot's camera image costs a lot of time, especially if several robots are simulated. In some development situations, it is better to switch off all low level processing of the robots and to work with ground truth world states, i. e., world states that are directly delivered by the simulator. In such cases there is no need to waste processing power by calculating camera images. Therefore, it can be switched off. However, by default this option is switched on. Note that this command only has an effect on simulated robots.

cls
Clears the console window.

dt off | **on** | <**fps**>
Switches simulation dragging to real-time on or off. If a frame rate is explicitly specified instead of "on", that one is used instead of the default value of 83.3 Hz. Normally, the simulation tries not to run faster than real-time. Thereby, it also reduces the general computational load of the computer. However, in some situations it is desirable to execute the simulation as fast as possible. By default, this option is activated.

echo <**text**>
Prints text into the console window. The command is useful in script files to print commands that can later be activated manually by pressing the *Enter* key in the printed line.

gc initial | **ready** | **set** | **playing** | **finished** | **competitionTypeNormal** | **competitionTypeMixedTeam** | **competitionPhasePlayoff** | **competitionPhaseRoundRobin** | **manualPlacementFirstTeam** | **manualPlacementSecondTeam** | **goalByFirstTeam** | **goalBySecondTeam** | **goalFreeKickForFirstTeam** | **goalFreeKickForSecondTeam** | **pushingFreeKickForFirstTeam** | **pushingFreeKickForSecondTeam** | **cornerKickForFirstTeam** | **cornerKickForSecondTeam** | **kickInForFirstTeam** | **kickInForSecondTeam** | **kickOffFirstTeam** | **kickOffSecondTeam** | **gamePenaltyShootout** | **gameNormal**
Sets the current state of the GameController.

(**help** | **?**) [<**pattern**>]
Displays a help text. If a pattern is specified, only those lines of the help are printed that contain the pattern.

mvo <**name**> <**x**> <**y**> <**z**> [<**rotx**> <**roty**> <**rotz**>]
Moves the object with the given name to the given position and optionally rotates it.

robot ? | **all** | <**name**> {<**name**>}
Selects a robot or a group of robots. The console commands described in the next section are only sent to *selected robots*. By default, only the robot that was created or connected last is selected. Using the *robot* command, this selection can be changed. Type *robot ?* to display a list of the names of available robots. A single simulated robot can also be selected by double-clicking it in the scene view. To select all robots, type *robot all*.

st off | **on**
Switches the simulation of time on or off. Without the simulation of time, all calls to `SystemCall::getCurrentSystemTime()` will return the real time of the PC. However if the simulator runs slower than real-time, the simulated robots will receive less sensor readings than the real ones. If the simulation of time is switched on, each step of the simulator will advance the time by 12 ms. Thus, the simulator simulates real-time, but it is running slower. By default this option is switched off.

<**text**>
Begins a comment line. Useful in script files.

10.1.6.3 Robot Commands

bc [<**red%**> [<**green%**> [<**blue%**>]]]
Defines the background color for 3-D views. The color is specified in percentages of red, green, and blue intensities. All parameters are optional. Missing parameters will be interpreted as 0%.

dr ? [<**pattern**>] | **off** | <**key**> (**off** | **on**)
Sends a debug request. B-Human uses debug requests to switch *debug responses* (cf. Section 3.6.1) on or off at runtime. Type *dr ?* to get a list of all available debug requests. The resulting list can be shortened by specifying a search pattern after the question mark. Debug responses can be activated or deactivated. They are deactivated by default. Specifying just *off* as the only parameter returns to this default state. Several other commands also implicitly send debug requests, e. g. to activate the transmission of debug drawings.

get ? [<**pattern**>] | <**key**> [**?**]
Shows debug data or shows its specification. This command allows displaying any information that is provided in the robot code via the `MODIFY` macro. If one of the strings that are used as first parameter of the `MODIFY` macro is used as parameter of this command (the *modify key*), the related data will be requested from the robot code and displayed. The output of the command is a valid *set* command (see below) that can be changed to modify data on the robot. A question mark directly after the command (with an optional filter pattern) will list all the modify keys that are available. A question mark after a modify key will display the structure of the associated data type rather than the data itself.

jc hide | **show** | **motion** (**1** | **2**) <**command**> | (**press** | **release**) <**button**> <**command**>
Sets a joystick command. If the first parameter is *press* or *release*, the number following

is interpreted as the number of a joystick button. Legal numbers are between 1 and 40. Any text after this first parameter is part of the second parameter. The <*command*> parameter can contain any legal script command that will be executed in every frame while the corresponding button is pressed. The prefixes *press* or *release* restrict the execution to the corresponding event. The commands associated with the 26 first buttons can also be executed by pressing *Ctrl+Shift+A*... *Ctrl+Shift+Z* on the keyboard. If the first parameter is *motion*, an analog joystick command is defined. There are two slots for such commands, number 1 and 2, e. g., to independently control the robot's walking direction and its head. The remaining text defines a command that is executed whenever the readings of the analog joystick change. Within this command, $1...$8 can be used as placeholders for up to eight joystick axes. The scaling of the values of these axes is defined by the command *js* (see below). If the first parameter is *show*, any command executed will also be printed in the console window. *hide* will switch this feature off again, and *hide* is also the default.

jm <**axis**> <**button**> <**button**>
Maps two buttons on an axis. Pressing the first button emulates pushing the axis to its positive maximum speed. Pressing the second button results in the negative maximum speed. The command is useful when more axes are required to control a robot than the joystick used actually has.

js <**axis**> <**speed**> <**threshold**> [<**center**>]
Sets the axis maximum speed and ignore threshold for command *jc motion*. *axis* is the number of the joystick axis to configure (1...8). *speed* defines the maximum value for that axis, i. e., the resulting range of values will be $[-speed \dots speed]$. The $threshold$ defines a joystick measuring range around zero in which the joystick will still be recognized as centered, i. e., the output value will be 0. The $threshold$ can be set between 0 and 1. An optional parameter allows for shifting the center itself, e. g. to compensate for bad calibration of a joystick.

kick
Adds the `KickEngine` view to the scene graph.

log ...
This command supports both recording and replaying log files. The latter is only possible if the current set of robot instances was created using the initialization command *sl* (cf. Section 10.1.6.1). This command has several sub-commands, i. e. the following commands are parameters of the *log* command, such as *log start*.

? [<**pattern**>]
Prints statistics on the messages contained in the current log file. The optional pattern limits output to messages corresponding to the pattern.

mr [**list**]
Sets the provider of all representations from the log file to `LogDataProvider`. If *list* is specified the module request commands will be printed to the console instead.

start | **stop**
If replaying a log file, starts and stops the replay. Otherwise, the commands will start and stop the recording.

pause | **forward** [**image**] | **backward** [**image**] | **repeat** | **goto** <**number**> | **time** <**minutes**> <**seconds**> | **fastForward** | **fastBackward**

Controls replay of a log file. *pause* stops the replay without rewinding to the beginning, *forward* and *backward* advance a single step in the respective direction. With the optional parameter *image*, it is possible to skip all frames in between that do not contain images. *repeat* just resends the messages from the current frame. *goto* allows jumping to a certain frame in the log file. If the log file contains the representation `GameInfo`, the command *time* allows to jump to the first frame with a certain remaining game time. *fastForward* and *fastBackward* advance by 100 steps forward or backward.

cycle | **once**
Decides whether the log file is replayed only once or continuously repeated.

clear | (**keep** | **remove**) <**message**> {<**message**>} | **keep** (**ballPercept** [**seen** | **guessed**] | **ballSpots** | **circlePercept** | **lower** | **option** <**option**> [<**state**>] | **penaltyMarkPercept** | **upper**)
clear removes all messages from the log file, while *keep* and *remove* only delete a selected subset based on the set of message IDs specified. *keep* also has a second form, in which a condition can be specified that is used to keep the frames (not just the messages) that meet the condition. The conditions are hardcoded in the implementation of the `RobotConsole`. They check, e. g., whether the image was recorded from a specific camera, a certain object was detected or the behavior was in a specific state in a frame.

(**load** | **save** [**split** <**parts**>]) <**file**>
These commands load or save the log file stored in memory from or to a file, respectively. If the filename is not a absolute path, it is relative to *Config/Logs*. Otherwise, the full path is used. *.log* is the default extension of log files. It will be automatically added if no extension is given. When saving a log file, it can be split into a specified number of parts with an equal number of messages.

trim (**until** <**end frame**> | **from** <**start frame**> | **between** <**start frame**> <**end frame**>)
Keeps only the given section of the log and saves the log file afterwards.

saveAudio [<**file**>]
Saves the audio data from the current log file to a *.wav* file. If no filename is given, the file is created at the log file's location with the *.wav* extension. If the specified filename is not an absolute path, it is relative to *Config/Sounds*.

saveImages [**raw**] [**onlyPlaying**] [<**takeEachNth**>] [<**dir**>]
Saves images from the current log file as PNG files to a directory. If *raw* is specified, images are left in their YUV format, otherwise they are converted to RGB. *onlyPlaying* only selects images that were taken while the robot was in the *playing* state, unpenalized and upright. Images can be skipped using the parameter *takeEachNth*. If no target directory is given, the images are put in a new directory next to the log file. If the specified directory is not an absolute path, it is relative to *Config/Images*.

(**saveInertialSensorData** | **saveJointAngleData**) [<**file**>]
Saves sensor data from the current log file to a *.csv* file. If no filename is given, the file is created at the log file's location with the *.csv* extension.

saveLabeledBallSpots [<**file**>]
Saves image patches around ball spots from the current log file, including a *.csv* file with label information. For this, the representation `LabelImage` must be present in the log file, which has to be patched in by an external annotation tool. The image patches are stored in a new directory at the log file's location.

saveTiming [<**file**>]
Saves timing data from all stopwatches for each frame from the current log file to a *.csv* file. If no filename is given, the file is created at the log file's location with the *.csv* extension.

full | **jpeg**
Decides whether uncompressed images received from the robot will also be written to the log file as full images, or JPEG-compressed on the recording PC before. When the robot is connected by cable, sending uncompressed images is usually a lot faster than compressing them on the robot. By executing *log jpeg*, they can still be saved in JPEG format, saving memory space during recording as well as disk space later. Note that running image processing routines on JPEG images does not always give realistic results, because JPEG is not a lossless compression method, and it is optimized for human viewers, not for machine vision.

mof Recompiles all special actions and if successful, sends the result to the robot.

msg off | **on** | **log** <**file**> | **disable** | **enable**
Switches the output of text messages on or off, or redirects them to a text file. All threads can send text messages via their message queues to the console window. As this can disturb entering text into the console window, printing can be switched off. However, by default text messages are printed. In addition, text messages can be stored in a log file, even if their output is switched off. The file name has to be specified after *msg log*. If the file already exists, it will be replaced. If no path is given, *Config/Logs* is used as default. Otherwise, the full path is used. *.txt* is the default extension of text log files. It will be automatically added if no extension is given. *enable* and *disable* can switch between handling messages from the robot at all, which is enabled by default.

mr ? [<**pattern**>] | **modules** [<**pattern**>] | **save** | <**representation**> (**?** [<**pattern**>] | <**module**> [<**thread**>] | **default** | **off**)
Sends a module request. This command allows selecting the module that provides a certain representation in a certain thread. If the representation is currently provided in exactly one thread, the *thread* argument can be omitted. If a representation should not be provided anymore, it can be switched *off*. Deactivating the provision of a representation is usually only possible if no other module requires that representation. Otherwise, an error message is printed and the robot is still using its previous module configuration. Sometimes, it is desirable to be able to deactivate the provision of a representation without the requirement to deactivate the provision of all other representations that depend on it. In that case, the provider of the representation can be set to *default*. Thus no module updates the representation and it simply keeps its previous state.

A question mark after the command lists all representations. A question mark after a representation lists all modules that provide this representation. The parameter *modules* lists all modules with their requirements and provisions. All three listings can be filtered by an optional pattern. *save* saves the current module configuration to the file *threads.cfg* which it was originally loaded from. Note that this usually has not the desired effect, because the module configuration has already been changed by the start script to be compatible with the simulator. Therefore, it will not work anymore on a real robot. The only configuration in which the command makes sense is when communicating with a remote robot.

mv <**x**> <**y**> <**z**> [<**rotx**> <**roty**> <**rotz**>]
Moves the selected simulated robot to the given position and rotates it if a rotation is

specified. x, y, and z have to be specified in mm, the rotations have to be specified in degrees. Note that the origin of the NAO is about 330 mm above the ground, so z should be 330.

mvb <**x**> <**y**> <**z**>
Moves the ball to the given position. x, y, and z have to be specified in mm. Note that the radius of the ball is 50 mm above the ground, so z should be at least 50.

poll
Polls for all available debug requests and debug drawings. Debug requests and debug drawings are dynamically defined in the robot control program. Before console commands that use them can be executed, the simulator must first determine which identifiers exist in the code that currently runs. Although the acquisition of this information is usually done automatically, e. g. after the module configuration was changed, there are some situations in which a manual execution of the command *poll* is required. For instance if debug responses or debug drawings are defined inside another debug response, executing *poll* is necessary to recognize the new identifiers after the outer debug response has been activated.

pr none | **illegalBallContact** | **playerPushing** | **illegalMotionInSet** | **inactivePlayer** | **illegalDefender** | **leavingTheField** | **kickOffGoal** | **requestForPickup** | **localGameStuck** | **illegalPositioning** | **substitute** | **manual**
Penalizes a simulated robot with the given penalty, or unpenalizes it when used with *none*. If the automatic referee is turned on, the necessary placement is done automatically: When penalized, the simulated robot will be moved to the sideline, looking away from the field. When unpenalized, it will be turned, facing the field again, and moved to the sideline that is further away from the ball.

save ? [<**pattern**>] | <**key**> [<**path**>]
Save debug data to a configuration file. The keys supported can be queried using the question mark. An additional pattern filters the output. If no path is specified, the name of the configuration file is looked up from a table, and its first occurrence in the search path is overwritten. Otherwise, the path is used. If it is relative, it is appended to the directory *Config*.

set ? [<**pattern**>] | <**key**> (**?** | **unchanged** | <**data**>)
Changes debug data or shows its specification. This command allows changing any information that is provided in the robot code via the `MODIFY` macro. If one of the strings that are used as first parameter of the `MODIFY` macro is used as parameter of this command (the *modify key*), the related data in the robot code will be replaced by the data structure specified as second parameter. It is best to first create a valid *set* command using the *get* command (see above). Afterwards that command can be changed before it is executed. If the second parameter is the key word *unchanged*, the related `MODIFY` statement in the code does not overwrite the data anymore, i. e. it is deactivated again. A question mark directly after the command (with an optional filter pattern) will list all the modify keys that are available. A question mark after a modify key will display the structure of the associated data type rather than the data itself.

si reset [<**number**>] | (**upper** | **lower**) [**number**] [**grayscale**] [**region** <**left**> <**top**> <**right**> <**bottom**>] [<**file**>]
Saves the raw image of a robot. The image will be saved in the format derived from the file extension in color, unless the *grayscale* option is specified. If no path is specified,

Config/raw_image.bmp will be used as default. If *number* is specified, a number is appended to the filename that is increased each time the command is executed. With the *region* option, the image can be cropped before being saved. The option *reset* resets the counter to a specified value or 0 if none is specified.

v3 ? [<**pattern**>] | <**image**> [**jpeg**] < [**thread**>] [<**name**>]
Adds a set of 3-D color space views for a certain image (cf. Section 10.1.4.1). The image can either be the camera image (simply specify *image*) or a debug image. The camera image can also be JPEG-compressed by using the *jpeg* option. By default, the debug image is taken from the *Lower* thread. However, the *thread* parameter can switch to the image from another thread. The last parameter is the name that will be given to the set of views. If the name is not given, it will be the same as the name of the image plus the name of the thread from which the image is taken. A question mark followed by an optional filter pattern will list all available images.

vd <**debug data**> (**on** | **off**)
Adds a data view (cf. Section 10.1.4.5) for debug data that is provided in the robot code via the `MODIFY` macro. Data views can be found in the data category of the scene graph. They provide the same functionality as the *get* and *set* commands (see above). However they are much more comfortable to use.

vf <**name**>
Adds a field view (cf. Section 10.1.4.1). A field view is the means for displaying debug drawings in field coordinates. The parameter defines the *name* of the view.

vfd ? [<**pattern**>] | **off** | (**all** | <**name**>) (**?** [<**pattern**>] | <**drawing**> (**on** | **off**))
(De)activates a debug drawing in a field view. The first parameter is the name of a field view that has been created using the command *vf* (see above) or *all* to add the drawing to all field views. The second parameter is the name of a drawing that is defined in the robot control program. Such a drawing is activated if the third parameter is *on* or is missing. It is deactivated if the third parameter is *off*. The drawings will be drawn in the sequence they are added, from back to front. Adding a drawing a second time will move it to the front unless the drawing is an origin drawing (those can be “drawn” multiple times). A question mark directly after the command will list all available field views. A question after a valid field view will list all available field drawings. Both question marks have an optional filter pattern that reduces the number of answers. *vfd off* will turn off all debug drawings except for the background in all field views.

vi ? [<**pattern**>] | <**image**> [**jpeg**] [**segmented**] [<**thread**>] [<**name**>] [**gain** <**value**>] [**ddScale** <**value**>]
Adds an image view (cf. Section 10.1.4.1). An image view is the means for displaying debug drawings in image coordinates. The image can either be the camera image (simply specify *image*), a debug image, or no image at all (*none*). The camera image can also be JPEG-compressed by using the *jpeg* option. If *segmented* is specified, the image will be segmented using the current color calibration. By default, the debug image is taken from the *Lower* thread. However, the *thread* parameter can switch to the image from another thread. The next parameter is the name of the view that is needed when adding drawings to it. If the name is not given, it will be the same as the name of the image plus the word *Segmented* if it should be segmented and the name of the thread from which the image is taken. With the argument of the *gain* parameter the image gain can be adjusted, if no gain is specified the default value will be 1.0. The argument of *ddScale* can set a custom

scaling of debug drawings in the view, which is 1.0 by default. A question mark followed by an optional filter pattern will list all available images.

vic ? [<**pattern**>] | (**all** | <**name**>) [**alt** | **noalt**] [**ctrl** | **noctrl**] [**shift** | **noshift**] <**command**>
Sets a command to be executed when clicking on an image view. The first parameter is the name of an image view that has been created using the command *vi* (see above) or *all* to set the command for all image views. Then, for the modifier keys *Alt*, *Control* and *Shift*, it can be selected whether they either must or must not be pressed while clicking for the command to be executed. Modifiers that are not mentioned at all are ignored. Everything after the modifier specification is the command that will be executed on clicking the image view. In the command, the placeholders \$1...\$3 can be used to insert the x-coordinate, y-coordinate and the lower case thread identifier to which the image view belongs, respectively. A question mark directly after the command will list all available image views.

vid ? [<**pattern**>] | **off** | (**all** | <**name**>) (**?** [<**pattern**>] | <**drawing**> (**on** | **off**))
(De)activates a debug drawing in an image view. The first parameter is the name of an image view that has been created using the command *vi* (see above) or *all* to add the drawing to all image views. The second parameter is the name of a drawing that is defined in the robot control program. Such a drawing is activated if the third parameter is *on* or is missing. It is deactivated if the third parameter is *off*. The drawings will be drawn in the sequence they are added, from back to front. Adding a drawing a second time will move it to the front. A question mark directly after the command will list all available image views. A question mark after a valid image view will list all available image drawings. Both question marks have an optional filter pattern that reduces the number of answers. *vid off* will turn off all debug drawings in all image views.

vp <**name**> <**numOfValues**> <**minValue**> <**maxValue**> [<**yUnit**> [<**xUnit**> [<**xScale**>]]]
Adds a plot view (cf. Section 10.1.4.5). A plot view is the means for plotting data that was defined by the macro `PLOT` in the robot control program. The first parameter defines the *name* of the view. The second parameter is the number of entries in the plot, i. e. the size of the x axis. The plot view stores the last *numOfValues* data points sent for each plot and displays them. *minValue* and *maxValue* define the range of the y axis. The optional parameters can improve the appearance of the plots by adding labels to both axes and by scaling the time-axis. The axis label drawing can be toggled by using the context menu of the plot view.

vpd ? [<**pattern**>] | <**name**> (**?** [<**pattern**>] | <**drawing**> (**?** [<**pattern**>] | <**color**> [<**description**>] | **off**))
Plots data in a certain color in a plot view. The first parameter is the name of a plot view that has been created using the command *vp* (see above). The second parameter is the name of plot data that is defined in the robot control program. The third parameter defines the color for the plot. *white*, *black*, *red*, *green*, *blue*, *yellow*, *cyan*, *magenta*, *orange*, *violet*, *gray*, *brown*, and six-digit hexadecimal RGB color codes are supported. An optional fourth parameter can set a custom description of the plot data. The plot is deactivated when the third parameter is *off*. The plots will be drawn in the sequence they were added, from back to front. Adding a plot a second time will move it to the front. A question mark directly after the command will list all available plot views. A question after a valid plot view will list all available plot data. A question mark after a valid plot view and a valid plot data identifier will list the available colors. All question marks have an optional filter pattern that reduces the number of answers.

10.1.6.4 Input Dialogs

Input dialogs may be used as placeholders in scripting-files to request input from the user. The dialogs will be opened before their respective script-lines are executed and the user selected value handled as usual script, e. g. a command or parameter.

${<**label**>}[2]
This expression opens a plain text input dialog.

${<**label**>,<**relativePath**>}[2]
This expression opens a file selection dialog at the relative path. The path may also specify legal file names via the asterisk-character (*). This is used, e. g., in the file *Config/Scenes/Include/replayDialog.con*: *$*{*Log File:,../../Logs/*.log*}.

${<**label**>,,<**relativePath**>}[2]
This expression opens a directory selection dialog at the relative path.

${<**label**>,<**value-1**>,<**value-2**>{,<**value-x**>}}[2]
This expression opens a drop-down list with the respective name and value options. This is used, e. g., to open an IP address dialog in the file *Config/Scenes/Include/connectDialog.con*: *$*{*IP address:,10.0.1.1,192.168.1.1*}.

10.2 B-Human User Shell

The B-Human User Shell (*bush*) accelerates and simplifies the deployment of code and the configuration of the robots. It is especially useful when controlling several robots at the same time, e. g., during the preparation for a soccer match.

10.2.1 Configuration

Since the *bush* can be used to communicate with the robots without much help from the user, it needs some information about the robots. Therefore, each robot has a configuration file *Config/Robots/<RobotName>/network.cfg* which defines the name of the robot and how it can be reached by the *bush*.[3] Additionally you have to define one (or more) teams, which are arranged in tabs. The data of the teams is used to define the other properties, which are required to deploy code in the correct configuration to the robots. The default configuration file of the teams is *Config/teams.cfg* which can be edited within the *bush* or with a text editor. Each team can have the configuration variables shown in Table 10.1.

10.2.2 Commands

The *bush* supports two types of commands. There are local commands (cf. Table 10.2) and commands that interact with selected robot(s) (cf. Table 10.3). Robots can be selected by checking their checkbox or with the keys *F1* to *F12*.

[2]The outer curly braces are literals.
[3]The configuration file is created by the script *createRobot* described in Section 2.3.4.

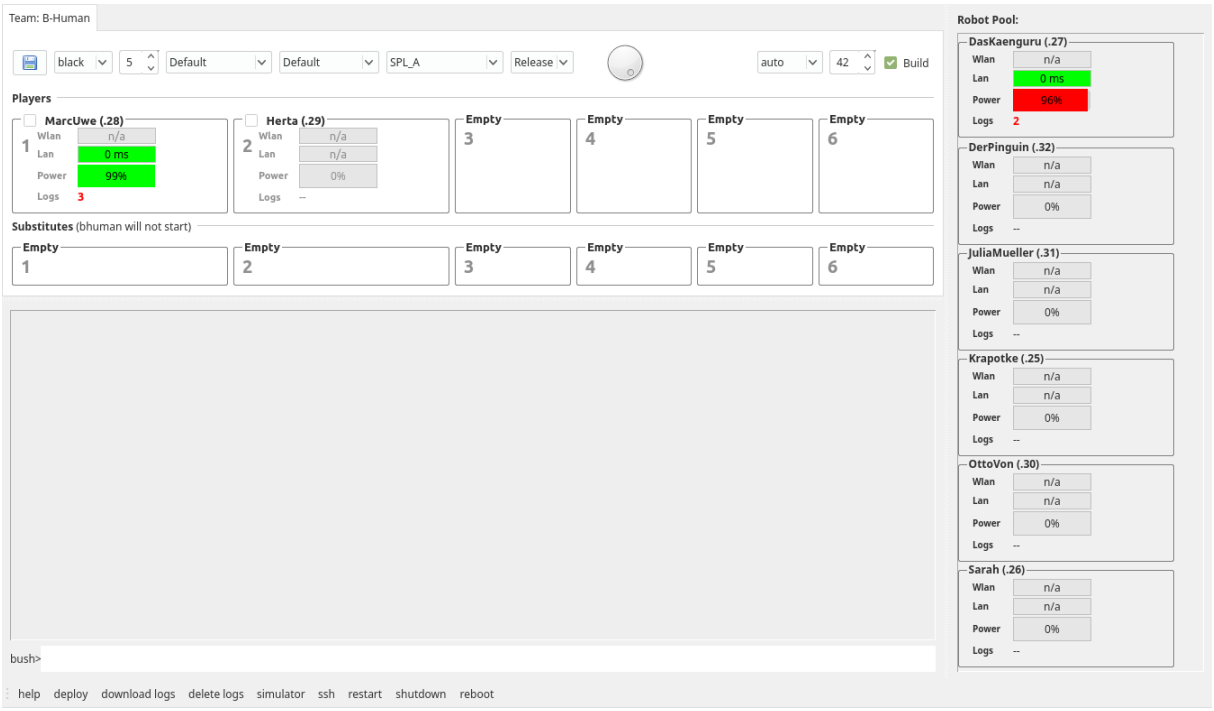


Figure 10.14: An example screenshot of the *bush*

10.2.3 Deploying Code to the Robots

For the simultaneous deployment of several robots the command *deploy* should be used. It accepts a single optional parameter that designates the build configuration of the code to be deployed to the selected robots. If the parameter is omitted the default build configuration of the currently selected team is used. It can be changed with the drop-down menu at the top of the *bush* user interface.

If the *Build* checkbox is set, the *deploy* command checks whether the binaries are up-to-date before it copies code to the robots. If needed, they are recompiled by the *compile* command, which can also be called independently from the *deploy* command. Depending on the platform, the *compile* command uses *make*, *xcodebuild*, or *MSBuild* to compile the binaries required.

The deployment itself uses the *copyfiles* script (cf. Section 2.4). This will stop the *bhuman* software before copying all necessary files. On all selected robots in the first row, *bhuman* is started again after the deployment. Furthermore, the *bhuman* software can be restarted manually using the *restart* command. This command can also reboot robots when called with the *robot* parameter. To inspect the configuration files copied to the robots, you can use the command *show*, which knows most of the files located on the robots and can help with finding the desired files with tab completion.

10.2.4 Substituting Robots

The robots known to the *bush* are arranged in two rows. The entries in the upper row represent the playing robots and the entries in the lower row the robots which stand by as substitutes. To select which robots are playing and which are not, you can move them by drag&drop to the appropriate position. Since this view only supports twelve robots at a time, there is another view called *RobotPool* at the right side of the *bush* window which contains all other robots. The

Entry	Description
name	The name of the team.
number	The team number.
port	The port which is used for team communication messages.
color	The team color.
scenario	The scenario used when accessing configuration files (cf. Section 2.8.2).
location	The location used when accessing configuration files (cf. Section 2.8.2).
compile	Should the code be compiled before it is deployed?
buildConfig	The name of the build configuration which should be deployed to the robot (cf. Section 2.1).
wlanConfig	The name of the wireless profile (cf. Section 2.3.2). Can be `NONE` to turn off the wireless.
volume	The audio playback volume to which the robots should be set to.
deployDevice	The device which should be used to connect to the robots. Either an Ethernet or a Wi-Fi connection can be established. The default entry is `auto`. This chooses the best device, depending on ping times.
magicNumber	The magic number for the team communication. Robots with different numbers will ignore each others messages. The special value -1 will set the last component of the IP address as magic number.
players	The list of robots the team consists of. The list must have twelve entries, where each entry must either be a name of a robot (with an existing file *Config/Robots/<RobotName>/network.cfg*), or an underscore for empty slots. The first six robots are the main players and the last six their substitutes.

Table 10.1: Configuration variables in the file *Config/teams.cfg*

robots displayed there can be exchanged with robots from the main view. If a robot from the lower row is deployed, *bhuman* will be shut down. This is important because the robots should be operational to be replaced as fast as possible, but are not allowed to send and receive network packets. It is important to note that after rebooting the robot, *bhuman* will run again.

10.2.5 Monitoring Robots

The *bush* displays some information about the robots' states as can be seen in Fig. 10.14: wireless connection ping time, wired connection ping time, battery status, and number of logfiles stored on the robot. The color of the power status bar indicates whether the battery is charging: A green bar indicates that the battery is connected to a power supply while a red bar means that the battery is not charging.

Command	Parameter(s) Description
compile	[<config> [<project>]] Compile a project with a specified build configuration. The default is to build the project Nao in the configuration Develop.
exit	 Exit the *bush*.
help	 Print a help text with all commands.
sim	 Open the *RemoteRobot.ros2* scene in the simulator.

Table 10.2: General *bush* commands.

Command	Parameter(s) Description
deploy	[<config>] Deploys code and all settings to the robot(s) using *copyfiles*.
downloadLogs	 Downloads all logs from the robot(s) and stores them at *Config/Logs*.
deleteLogs	 Deletes all logs from the robot(s).
restart	[bhuman | robot] Restarts *bhuman* or the robot. If no parameter is given, *bhuman* will be restarted.
scp	@<path on NAO> <local path> | <local path> <path on NAO> Copies a file to or from the robot(s). The first argument is the source and the second the destination path.
show	<config file> Prints the configuration file stored on the robot(s).
shutdown	 Executes a shutdown on the robot(s).
ssh	[<command>] Executes a command via ssh or opens an ssh session.

Table 10.3: *Bush* commands that need at least one selected robot.

Chapter 11

Acknowledgments

© Photographer@Large

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, Alumni of the University of Bremen, engineering people, neuland, and sysGen for funding parts of our project. Since B-Human 2019 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`)

Flite: For speech synthesis on Linux (including the NAO).
(http://www.festvox.org/flite/)

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/)

HDF5: For reading Keras model files.
(https://www.hdfgroup.org/solutions/hdf5/)

ld: The GNU linker is used for cross linking on Windows and macOS.
(http://sourceware.org/binutils/docs-2.30/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 color 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)

MD5: To efficiently check for repeating image rows (in response to a problem with the NAO V6's cameras).
(https://bobobobo.wordpress.com/2010/10/17/md5-c-implementation/)

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)

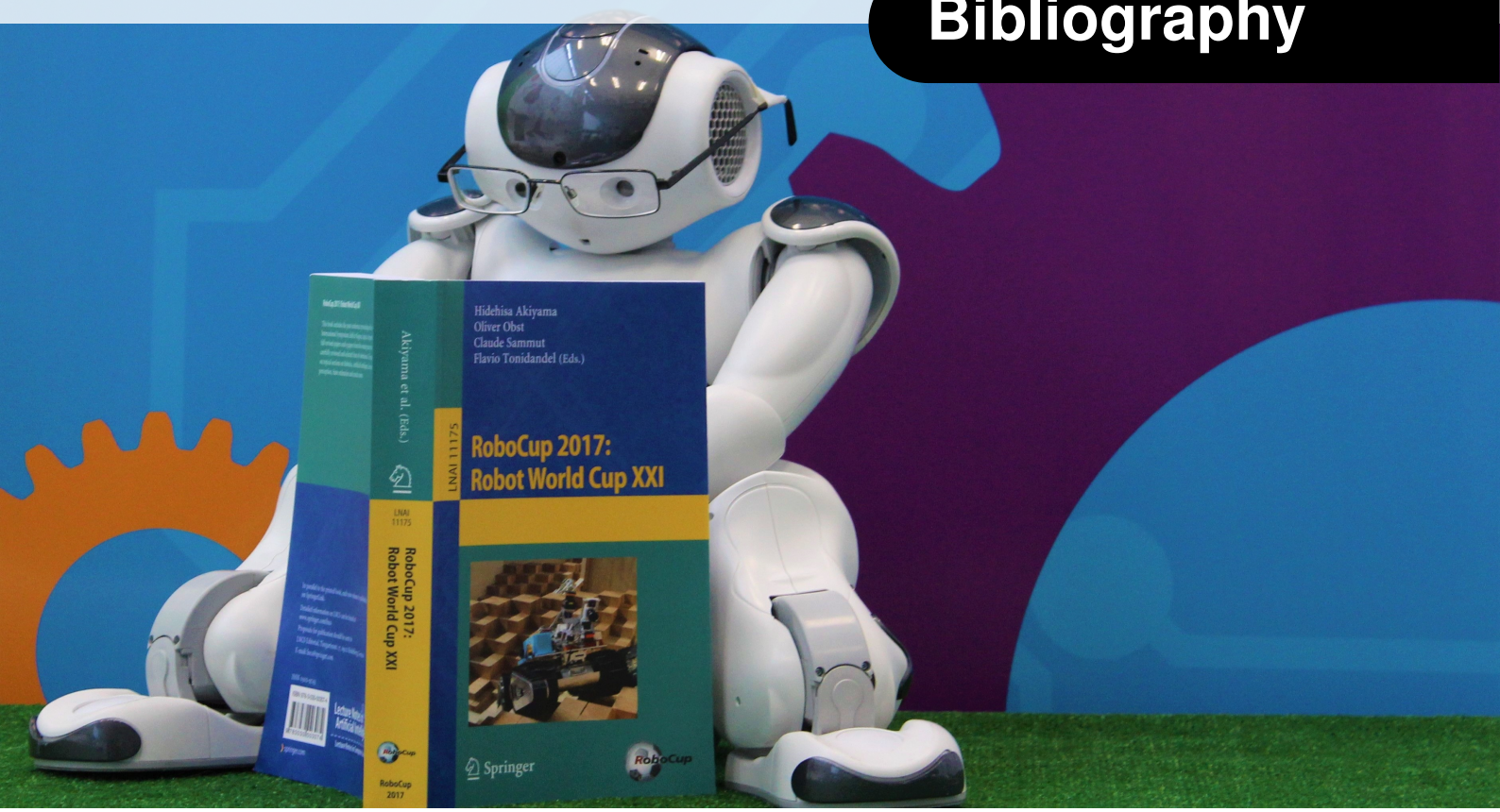
Qt: The GUI framework of the simulator.
(http://www.qt.io)

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] Xavier Alameda-Pineda and Radu Horaud. A geometric approach to sound source localization from time-delay estimates. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, 22, 11 2013.

[2] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) Rule Book, 2019. Only available online: `https://spl.robocup.org/wp-content/uploads/downloads/Rules2019.pdf`.

[3] J. Dmochowski, J. Benesty, and S. Affes. On spatial aliasing in microphone arrays. *IEEE Transactions on Signal Processing*, 57(4):1383–1395, April 2009.

[4] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte-Carlo Localization: Efficient Position Estimation for Mobile Robots. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 343 – 349, Orlando, FL, USA, 1999.

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

[6] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.

[7] Google. Snappy – a fast compressor/decompressor. Online: `https://google.github.io/snappy/`, September 2019.

[8] Bernhard Hengst. rUNSWift Walk2014 report. Technical report, School of Computer Science & Engineering University of New South Wales, Sydney 2052, Australia, 2014. http://cgi.cse.unsw.edu.au/~robocup/2014ChampionTeamPaperReports/20140930-Bernhard.Hengst-Walk2014Report.pdf.

[9] Timm Hess, Martin Mundt, Tobias Weis, and Visvanathan Ramesh. Large-scale stochastic scene generation and semantic annotation for deep convolutional neural network training in the RoboCup SPL. In Hidehisa Akiyama, Oliver Obst, Claude Sammut, and Flavio Tonidandel, editors, *RoboCup 2017: Robot World Cup XXI*, volume 11175 of *Lecture Notes in Artificial Intelligence*, pages 33–44. Springer, 2018.

[10] V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, editors. *Blackboard Architectures and Applications*. Academic Press, Boston, 1989.

[11] Simon J. Julier, Jeffrey K. Uhlmann, and Hugh F. Durrant-Whyte. A New Approach for Filtering Nonlinear Systems. In *Proceedings of the American Control Conference*, volume 3, pages 1628–1632, 1995.

[12] Tim Laue and Thomas Röfer. SimRobot – Development and applications. In Heni Ben Amor, Joschka Boedecker, and Oliver Obst, editors, *The Universe of RoboCup Simulators – Implementations, Challenges and Strategies for Collaboration. Workshop Proceedings of the International Conference on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR 2008)*, Venice, Italy, 2008.

[13] Tim Laue, Thomas Röfer, Katharina Gillmann, Felix Wenk, Colin Graf, and Tobias Kastner. B-Human 2011 – eliminating game delays. In Thomas Röfer, N. Michael Mayer, Jesus Savage, and Uluç Saranlı, editors, *RoboCup 2011: Robot Soccer World Cup XV*, volume 7416 of *Lecture Notes in Artificial Intelligence*, pages 25–36. Springer, 2012.

[14] Tim Laue, Kai Spiess, and Thomas Röfer. SimRobot – A general physical robot simulator and its application in RoboCup. In Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors, *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Artificial Intelligence*, pages 173–183. Springer, 2006.

[15] Scott Lenser and Manuela Veloso. Sensor resetting localization for poorly modelled mobile robots. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation (ICRA 2000)*, volume 2, pages 1225–1232, San Francisco, CA, USA, 2000.

[16] Heinrich Mellmann, Benjamin Schlotter, Steffen Kaden, Philipp Strobel, Thomas Krause, Etienne Couque-Castelnovo, Claas-Norman Ritter, Tobias Hübner, and Schahin Tofangchi. Berlin United – Nao Team Humboldt team report 2018, 2019. Only available online: https://www.naoteamhumboldt.de/wp-content/papercite-data/pdf/naoth-report18.pdf.

[17] Judith Müller, Tim Laue, and Thomas Röfer. Kicking a Ball – Modeling Complex Dynamic Motions for Humanoid Robots. In Javier Ruiz del Solar, Eric Chown, and Paul G. Ploeger, editors, *RoboCup 2010: Robot Soccer World Cup XIV*, volume 6556 of *Lecture Notes in Artificial Intelligence*, pages 109–120. Springer, 2011.

[18] Bernd Poppinga and Tim Laue. JET-Net: Real-time object detection for mobile robots. In *RoboCup 2019: Robot World Cup XXIII*. Springer, to appear.

[19] Thomas Röfer. Region-Based Segmentation with Ambiguous Color Classes and 2-D Motion Compensation. In Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI*, volume 5001 of *Lecture Notes in Artificial Intelligence*, pages 369–376. Springer, 2008.

[20] Thomas Röfer. CABSL – C-based agent behavior specification language. In Hidehisa Akayama, Oliver Obst, Claude Sammut, and Flavio Tonidandel, editors, *RoboCup 2017: Robot World Cup XII*, volume 11175 of *Lecture Notes in Artificial Intelligence*, pages 135 – 142. Springer, 2018.

[21] Thomas Röfer, Jörg Brose, Daniel Göhring, Matthias Jüngel, Tim Laue, and Max Risler. GermanTeam 2007. In Ubbo Visser, Fernando Ribeiro, Takeshi Ohashi, and Frank Dellaert, editors, *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*, Atlanta, GA, USA, 2007. RoboCup Federation.

[22] Thomas Röfer and Tim Laue. On B-Human's code releases in the Standard Platform League – software architecture and impact. In Sven Behnke, Manuela Veloso, Arnoud Visser, and Rong Xiong, editors, *RoboCup 2013: Robot World Cup XVII*, volume 8371 of *Lecture Notes in Artificial Intelligence*, pages 648–656. Springer, 2014.

[23] 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`.

[24] Thomas Röfer, Tim Laue, Arne Hasselbring, Jannik Heyen, Bernd Poppinga, Philip Reichenberg, Enno Roehrig, and Felix Thielke. B-Human team report and code release 2018, 2018. Only available online: `http://www.b-human.de/downloads/publications/2018/CodeRelease2018.pdf`.

[25] 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 Hidehisa Akiyama, Oliver Obst, Claude Sammut, and Flavio Tonidandel, editors, *RoboCup 2017: Robot World Cup XXI*, volume 11175 of *Lecture Notes in Artificial Intelligence*, pages 461 – 472. Springer, 2018.

[26] Thomas Röfer, Tim Laue, and Jesse Richter-Klug. B-Human 2016 – robust approaches for perception and state estimation under more natural conditions. In Sven Behnke, Raymond Sheh, Sanem Sarıel, and Daniel D. Lee, editors, *RoboCup 2016: Robot World Cup XX*, volume 9776 of *Lecture Notes in Artificial Intelligence*, pages 503 – 514. Springer, 2017.

[27] Thomas Röfer, Tim Laue, Jesse Richter-Klug, Jonas Stiensmeier, Maik Schünemann, Andreas Stolpmann, Alexander Stöwing, and Felix Thielke. B-Human team description for RoboCup 2015. In *RoboCup 2015: Robot World Cup XIX Preproceedings*, Hefei, China, 2015. RoboCup Federation.

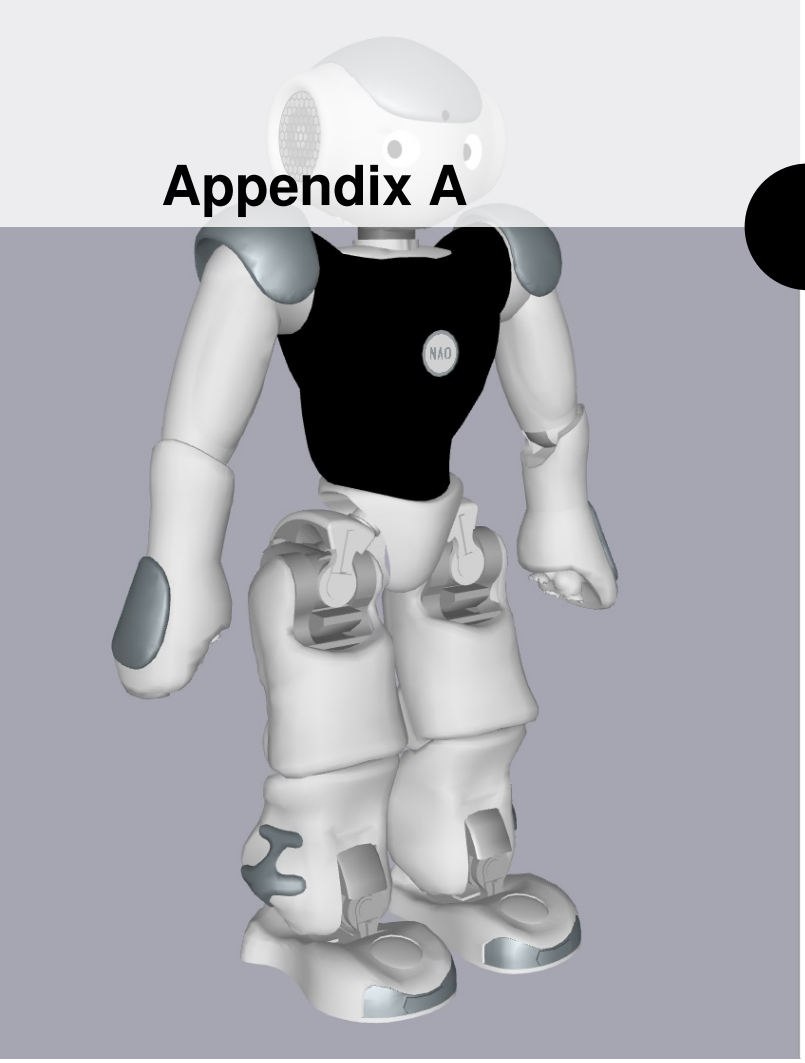
[28] Thomas Röfer, Tim Laue, Michael Weber, Hans-Dieter Burkhard, Matthias Jüngel, Daniel Göhring, Jan Hoffmann, Benjamin Altmeyer, Thomas Krause, Michael Spranger, Oskar von Stryk, Ronnie Brunn, Marc Dassler, Michael Kunz, Tobias Oberlies, Max Risler, Uwe Schwiegelshohn, Matthias Hebbel, Walter Nisticó, Stefan Czarnetzki, Thorsten Kerkhof, Matthias Meyer, Carsten Rohde, Bastian Schmitz, Michael Wachter, Tobias Wegner, and Christine Zarges. GermanTeam RoboCup 2005, 2005. Only available online: `http://www.informatik.uni-bremen.de/kogrob/papers/GT2005.pdf`.

[29] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *Lecture Notes in Computer Science*, pages 234–241. Springer, 2015. (available on arXiv:1505.04597 [cs.CV]).

[30] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, Cambridge, 2005.

[31] Max Trocha. Werkzeug zur taktischen Auswertung von Spielsituationen. Bachelor's thesis, University of Bremen, 2010.

Appendix A



Scene Description

```
<Simulation>
  <Body name="Nao"> <!-- Torso -->
    <Translation z="400mm"/>

    <Set name="NaoColor" value="black"/>

    <BoxGeometry depth="100mm" width="120mm" height="190mm">
      <Translation z="20mm"/>
    </BoxGeometry>
    <Geometry name="origin">
      <Translation z="-0.085"/>
    </Geometry>

    <Geometry name="$NaoColor"/>

    <Appearance ref="naoTorsoV6"/>

    <Gyroscope>
      <Translation x="-0.008" y="0.006" z="0.029"/>
    </Gyroscope>
    <Accelerometer>
      <Translation x="-0.008" y="0.00606" z="0.027"/>
    </Accelerometer>
    <!-- When setting the max distance of ApproxDistanceSensor
         will result in false measurements from the ground.
    -->
    <ApproxDistanceSensor name="SonarCenterRight" min="0mm" ma
      <Translation x="0.0507" y="-0.00375" z="0.06"/>
      <Rotation z="-2.5degree"/>
    </ApproxDistanceSensor>
    <ApproxDistanceSensor name="SonarRight" min="0mm" max="140
      <Translation x="0.0507" y="-0.03785" z="0.06"/>
      <Rotation z="-22.5degree"/>
```

A.1 EBNF

In the next section the structure of a scene description file is explained by means of an EBNF representation of the language. In the following, you can find an explanation of the symbols used.

Symbols surrounded by ?(...)?

Parentheses with question marks mean that the order of all elements between them is irrelevant. That means every permutation of elements within those brackets is allowed. For example: $something$ =?($firstEle$ $secondEle$ $thirdEle$)? can also be written as $something$ =?($secondEle$ $firstEle$ $thirdEle$)? or as $something$ = ?($thirdEle$ $firstEle$ $sencondEle$)? and so on.

Symbols surrounded by !

$!x, y, \ldots!$ means, that each rule is required. In fact the exclamation marks should only underline that all elements between them are absolutely required. In normal EBNF-Notation a rule like $Hinge = \ldots!bodyClass,\ axisClass!\ldots$ can be written as $Hinge = \ldots bodyClass\ axisClass\ldots$.

+[...]+

$+[x, y]+$ means, that x and y are optional. You could also write $somewhat = +[x, y, z]+$ as $somewhat = [x]\ [y]\ [z]$.

{...}

Elements within curly braces are repeatable optional elements. These brackets have the

normal EBNF meaning.

“...”
Terminal symbols are marked with quotation marks.

A.2 Grammar

```
appearanceClass          = Appearance | BoxAppearance | CapsuleAppearance
                         | ComplexAppearance | CylinderAppearance
                         | SphereAppearance;
axisClass                = Axis;
bodyClass                = Body;
compoundClass            = Compound;
deflectionClass          = Deflection;
extSensorClass           = ApproxDistanceSensor | Camera | DepthImageSensor
                         | ObjectSegmentedImageSensor | SingleDistanceSensor;
frictionClass            = Friction | RollingFriction;
geometryClass            = BoxGeometry | CapsuleGeometry | CylinderGeometry
                         | Geometry | SphereGeometry;
infrastructureClass      = Include | Simulation;
intSensorClass           = Accelerometer | CollisionSensor | Gyroscope;
jointClass               = Hinge | Slider;
lightClass               = Light;
massClass                = BoxMass | InertiaMatrixMass | Mass | SphereMass;
materialClass            = Material;
motorClass               = PT2Motor | ServoMotor | VelocityMotor;
normalsClass             = Normals;
primitiveGroupClass      = Quads | Triangles;
rotationClass            = Rotation;
sceneClass               = Scene;
setClass                 = Set;
solverClass              = QuickSolver;
surfaceClass             = Surface;
texCoordsClass           = TexCoords;
translationClass         = Translation;
userInputClass           = UserInput;
verticesClass            = Vertices;


Appearance               = "<Appearance>" ?( +[translationClass,
                           rotationClass]+ {setClass | appearanceClass} )?
                           "</Appearance>" | "<Appearance/>;
BoxAppearance            = "<BoxAppearance>" ?( !surfaceClass!
                           +[translationClass, rotationClass]+
                           {setClass | appearanceClass} )?
                           "</BoxAppearance>";
CapsuleAppearance        = "<CapsuleAppearance>" ?( !surfaceClass!
                           +[translationClass, rotationClass]+
                           {setClass | appearanceClass} )?
                           "</CapsuleAppearance>";
ComplexAppearance        = "<ComplexAppearance>" ?( !surfaceClass,
                           verticesClass, primitiveGroupClass!
                           +[translationClass, rotationClass, normalsClass,
                           texCoordsClass]+ {setClass | appearanceClass
                           | primitiveGroupClass} )? "</ComplexAppearance>";
```

```
CylinderAppearance         = "<CylinderAppearance>" ?( !surfaceClass!
                             +[translationClass, rotationClass]+
                             {setClass | appearanceClass} )?
                             "</CylinderAppearance>";
SphereAppearance           = "<SphereAppearance>" ?( !surfaceClass!
                             +[translationClass, rotationClass]+
                             {setClass | appearanceClass} )?
                             "</SphereAppearance>";

Axis                       = "<Axis>" ?( +[motorClass, deflectionClass]+
                             {setClass} )? "</Axis>" | "<Axis/>";

Body                       = "<Body>" ?( !massClass! +[translationClass,
                             rotationClass]+ {setClass | jointClass
                             | appearanceClass | geometryClass | massClass
                             | intSensorClass | extSensorClass
                             | userInputClass} )? "</Body>";

Compound                   = "<Compound>" ?( +[translationClass, rotationClass]+
                             {setClass | jointClass | compoundClass | bodyClass
                             | appearanceClass | geometryClass | extSensorClass
                             | userInputClass} )? "</Compound>" | "<Compound/>";

Deflection                 = "<Deflection></Deflection>" | "<Deflection/>";

ApproxDistanceSensor       = "<ApproxDistanceSensor>" ?( +[translationClass,
                             rotationClass]+ )? "</ApproxDistanceSensor>"
                             | "<ApproxDistanceSensor/>";
Camera                     = "<Camera>" ?( +[translationClass,
                             rotationClass]+ )? "</Camera>" | "<Camera/>";
DepthImageSensor           = "<DepthImageSensor>" ?( +[translationClass,
                             rotationClass]+ )? "</DepthImageSensor>"
                             | "<DepthImageSensor/>";
ObjectSegmentedImageSensor = "<ObjectSegmentedImageSensor>"
                             ?( +[translationClass, rotationClass]+ )?
                             "</ObjectSegmentedImageSensor>"
                             | "<ObjectSegmentedImageSensor/>";
SingleDistanceSensor       = "<SingleDistanceSensor>" ?( +[translationClass,
                             rotationClass]+ )? "</SingleDistanceSensor>"
                             | "<SingleDistanceSensor/>";

Friction                   = "<Friction></Friction>" | "<Friction/>";
RollingFriction            = "<RollingFriction></RollingFriction>"
                             | "<RollingFriction/>";

BoxGeometry                = "<BoxGeometry>" ?( +[translationClass,
                             rotationClass, materialClass]+ {setClass
                             | geometryClass} )?  "</BoxGeometry>"
                             | "<BoxGeometry/>";
CapsuleGeometry            = "<CapsuleGeometry>" ?( +[translationClass,
                             rotationClass, materialClass]+ {setClass
                             | geometryClass} )? "</CapsuleGeometry>"
                             | "<CapsuleGeometry/>";
CylinderGeometry           = "<CylinderGeometry>" ?( +[translationClass,
                             rotationClass, materialClass]+ {setClass
                             | geometryClass} )? "</CylinderGeometry>"
```

```
                          | "<CylinderGeometry/>";
Geometry                = "<Geometry>" ?( +[translationClass, rotationClass,
                          materialClass]+ {setClass | geometryClass} )?
                          "</Geometry>" | "<Geometry/>";
SphereGeometry          = "<SphereGeometry>" ?( +[translationClass,
                          rotationClass, materialClass]+ {setClass
                          | geometryClass} )? "</SphereGeometry>"
                          | "<SphereGeometry/>";

Include                 = "<Include></Include>" | "<Include/>";
Simulation              = "<Simulation>" !sceneClass! "</Simulation>";

Accelerometer           = "<Accelerometer>" ?( +[translationClass,
                          rotationClass]+ )? "</Accelerometer>"
                          | "<Accelerometer/>";
CollisionSensor         = "<CollisionSensor>" ?( +[translationClass,
                          rotationClass]+ {geometryClass} )?
                          "</CollisionSensor>" | "<CollisionSensor/>";
Gyroscope               = "<Gyroscope>" ?( +[translationClass,
                          rotationClass]+ )? "</Gyroscope>"
                          | "<Gyroscope/>";

Hinge                   = "<Hinge>" ?( !bodyClass, axisClass!
                          +[translationClass, rotationClass]+ {setClass} )?
                          "</Hinge>";
Slider                  = "<Slider>" ?( !bodyClass, axisClass!
                          +[translationClass, rotationClass]+ {setClass} )?
                          "</Slider>";

Light                   = "<Light></Light>" | "<Light/>";

BoxMass                 = "<BoxMass>" ?( +[translationClass, rotationClass]+
                          {setClass | massClass} )? "</BoxMass>"
                          | "<BoxMass/>;
InertiaMatrixMass       = "<InertiaMatrixMass>" ?( +[translationClass,
                          rotationClass]+ {setClass | massClass} )?
                          "</InertiaMatrixMass>" | "<InertiaMatrixMass/>";
Mass                    = "<Mass>" ?( +[translationClass, rotationClass]+
                          {setClass | massClass} )? "</Mass>" | "<Mass/>";
SphereMass              = "<SphereMass>" ?( +[translationClass,
                          rotationClass]+ {setClass | massClass} )?
                          "</SphereMass>" | "<SphereMass/>";

Material                = "<Material>" ?( {setClass | frictionClass} )?
                          "</Material>" | "<Material/>";

PT2Motor                = "<PT2Motor></PT2Motor>" | "<PT2Motor/>";
ServoMotor              = "<ServoMotor></ServoMotor>" | "<ServoMotor/>";
VelocityMotor           = "<VelocityMotor></VelocityMotor>"
                          | "<VelocityMotor/>";

Normals                 = "<Normals>" Normals Definition "</Normals>";

Quads                   = "<Quads>" Quads Definition "</Quads>";
Triangles               = "<Triangles>" Triangles Definition "</Triangles>";
```

```
Rotation                = "<Rotation></Rotation>" | "<Rotation>";

Scene                   = "<Scene>" ?( +[solverClass]+ {setClass | bodyClass
                          | compoundClass | lightClass | userInputClass} )?
                          "</Scene>";

Set                     = "<Set></Set>" | "<Set/>";

QuickSolver             = "<QuickSolver></QuickSolver>" | "<QuickSolver/>";

Surface                 = "<Surface></Surface>" | "<Surface/>";

TexCoords               = "<TexCoords>" TexCoords Definition "</TexCoords>";

Translation             = "<Translation></Translation>" | "<Translation/>";

UserInput               = "<UserInput></UserInput>" | "<UserInput/>";

Vertices                = "<Vertices>" Vertices Definition "</Vertices>";
```

A.3 Structure of a Scene Description File

A.3.1 The Beginning of a Scene File

Every scene file has to start with a *<Simulation>* tag. Within a *Simulation* block a *Scene* element is required, but there is one exception: files included via *<Include href=...>* must start with *<Simulation>*, but there is no *Scene* element required. A *Scene* element specifies which controller is loaded for this scene via the *controller* attribute (in our case all scenes set the *controller* attribute to *SimulatedNao*, so that the library *SimulatedNao* is loaded by SimRobot). It is recommended to include other specifications per *Include* before the scene description starts (compare with *BH.ros2*), but it is not necessary.

A.3.2 The ref Attribute

An element with a *name* attribute can be referenced by the *ref*-attribute using its name, i. e. elements that are needed repeatedly in a scene need to be defined only once. For example there is only one description of a NAO in its definition file (*NaoV6H25.rsi2*), but NAOs with different jersey colors are needed on a field. For each NAO on the field, there is a reference to the original model. The positioning of the NAOs is done by *Translation* and *Rotation* elements. The color is set by a *Set* element, which is described below.

```
<Body name="Nao">
  <Set name="NaoColor" value="blue"/>
⋮
</Body>
⋮
<Body ref="Nao" name="BlueNao">
  <Translation x="-2" y="0.4" z="320mm"/>
</Body>
<Body ref="Nao" name="RedNao">
  <Translation x="-1.5" y="-0.9" z="320mm"/>
```

```
    <Rotation z="180degree"/>
    <Set name="NaoColor" value="red"/>
  </Body>
  ⋮
```

A.3.3 Placeholders and Set Element

A placeholder has to start with a $ followed by an arbitrary string. A placeholder is replaced by the definition specified within the corresponding *Set* element. The attribute *name* of a *Set* elements specifies the placeholder, which is replaced by the value specified by the attribute *value* of the *Set* element.

In the following code example, the color of NAO's jersey is set by a *Set* element. Within the definition of the body *Nao* named *RedNao*, the *Set* element sets the placeholder color to the value *red*. The placeholder named *NaoColor* of *Nao*, which is defined in the general definition of a NAO, is replaced by *red* in all elements of the model, also in the ones that are just referenced, such as the appearances of individual body parts. So the *Surface* elements reference a *Surface* named *nao-red*.

```
  ⋮
  <ComplexAppearance name="naoTorsoV6_jersey">
    <Surface ref="nao-$NaoColor"/>
  ⋮
  </ComplexAppearance>
  ⋮
  <Surface name="nao-red" diffuseColor="rgb(100%, 0%, 0%)" ambientColor="rgb
      (20%, 12%, 12%)"/>
  ⋮
```

A.4 Attributes

A.4.1 appearanceClass

- **Appearance** Specifies an appearance (the visual shape of an object).
  - name The name of the appearance.
    - **Use:** optional
    - **Range:** String
- **BoxAppearance** Specifies a box-shaped appearance.
  - name The name of the appearance.
    - **Use:** optional
    - **Range:** String
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CapsuleAppearance** Specifies a capsule-shaped appearance.
  - name The name of the appearance.
    - **Use:** optional
    - **Range:** String
  - height The height of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - radius The radius of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **ComplexAppearance** Specifies an appearance with an arbitrary mesh.
  - name The name of the appearance.
    - **Use:** optional
    - **Range:** String

- **CylinderAppearance** Specifies a cylinder-shaped appearance.
  - name The name of the appearance.
    - **Use:** optional
    - **Range:** String
  - height The height of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - radius The radius of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **SphereAppearance** Specifies a sphere-shaped appearance.
  - name The name of the appearance.
    - **Use:** optional

        - **Range:** String
    - radius The radius of the sphere.
        - **Units:** mm, cm, dm, m, km
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.2 axisClass

- **Axis** Specifies the axis of a joint.
    - x The x direction of the axis.
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - y The y direction of the axis.
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - z The z direction of the axis.
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - cfm The cfm (constraint force mixing) value for this axis.
        - **Default:** -1
        - **Use:** optional
        - **Range:** $[0, 1]$

A.4.3 bodyClass

- **Body** Specifies an object that has a mass and can move.
    - name The name of the body.
        - **Use:** optional
        - **Range:** String

A.4.4 compoundClass

- **Compound** Specifies an object that cannot move.
    - name The name of the compound.
        - **Use:** optional
        - **Range:** String

A.4.5 deflectionClass

- **Deflection** Specifies the maximum and minimum deflection of a joint.
  - min The minimal deflection.
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximal deflection.
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - init The initial deflection.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - stopCFM The cfm (constant force mixing) value of the limits.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - stopERP The erp (error reducing parameter) value of the limits.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$

A.4.6 extSensorClass

- **ApproxDistanceSensor** Instantiates a sensor that measures the distance in an area in front of it.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
  - min The minimum distance this sensor can measure.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - angleX The maximum angle in x-direction the ray of the sensor can spread.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** $(0, MAXFLOAT]$
  - angleY The maximum angle in y-direction the ray of the sensor can spread.

        - **Units:** degree, radian
        - **Use:** required
        - **Range:** $(0, MAXFLOAT]$

- **Camera** Instantiates a color image camera.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the camera image.
    - **Use:** required
    - **Range:** $(0, MAXINTEGER]$
  - imageHeight The height of the camera image.
    - **Use:** required
    - **Range:** $(0, MAXINTEGER]$
  - angleX The horizontal opening angle.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** $(0, MAXFLOAT]$
  - angleY The vertical opening angle.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** $(0, MAXFLOAT]$

- **DepthImageSensor** Instantiates a depth image camera.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the image.
    - **Use:** required
    - **Range:** $(0, MAXINTEGER]$
  - imageHeight The height of the image.
    - **Default:** 1
    - **Use:** optional
    - **Range:** $(0, MAXINTEGER]$
  - angleX The horizontal opening angle.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** $(0, MAXFLOAT]$
  - angleY The vertical opening angle.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** $(0, MAXFLOAT]$

  - min The minimum distance this sensor can measure.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - projection The kind of projection.
    - **Default:** perspective
    - **Use:** optional
    - **Range:** perspective, spheric

- **ObjectSegmentedImageSensor** Instantiates a camera which renders an objected segmented image.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
  - imageWidth The width of the camera image.
    - **Use:** required
    - **Range:** $(0, MAXINTEGER]$
  - imageHeight The height of the camera image.
    - **Use:** required
    - **Range:** $(0, MAXINTEGER]$
  - angleX The horizontal opening angle.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** $(0, MAXFLOAT]$
  - angleY The vertical opening angle.
    - **Units:** degree, radian
    - **Use:** required
    - **Range:** $(0, MAXFLOAT]$

- **SingleDistanceSensor** Instantiates a sensor that measures a distance on a single ray.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
  - min The minimum distance this sensor can measure.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum distance this sensor can measure.
    - **Default:** 999999
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.7 frictionClass

- **Friction** Specifies the friction between this material and another material.
  - material The other material the friction belongs to.
    - **Use:** required
    - **Range:** String
  - value The value of the friction.
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
- **RollingFriction** Specifies the rolling friction of a material.
  - material The other material the rolling friction belongs to.
    - **Use:** required
    - **Range:** String
  - value The value of the rolling friction.
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$

A.4.8 geometryClass

- **BoxGeometry** Specifies a box-shaped geometry.
  - color A color definition, see A.4.23
    - **Use:** optional
  - name The name of the geometry.
    - **Use:** optional
    - **Range:** String
  - width The width of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - depth The depth of the box.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
- **CapsuleGeometry** Specifies a capsule-shaped geometry.
  - color A color definition, see A.4.23
    - **Use:** optional

  - name The name of the geometry.
    - **Use:** optional
    - **Range:** String
  - radius The radius of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the capsule.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **CylinderGeometry** Specifies a cylinder-shaped geometry.
  - color A color definition, see A.4.23
    - **Use:** optional
  - name The name of the geometry.
    - **Use:** optional
    - **Range:** String
  - radius The radius of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - height The height of the cylinder.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **Geometry** Specifies a geometry (the physical shape of an object).
  - name The name of the geometry.
    - **Use:** optional
    - **Range:** String

- **SphereGeometry** Specifies a sphere-shaped geometry.
  - color A color definition, see A.4.23
    - **Use:** optional
  - name The name of the geometry.
    - **Use:** optional
    - **Range:** String
  - radius The radius of the sphere.
    - **Units:** mm, cm, dm, m, km
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.9 infrastructureClass

- **Include** Includes another scene description file.
  - href The path to the included file.
    - **Use:** optional
    - **Range:** String
- **Simulation** Has to be the outermost element of every file.

A.4.10 intSensorClass

- **Accelerometer** Instantiates an accelerometer on a body.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
- **CollisionSensor** Instantiates a collision sensor on a body which uses geometries to detect collisions with other objects.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String
- **Gyroscope** Instantiates a gyroscope on a body.
  - name The name of the sensor.
    - **Use:** optional
    - **Range:** String

A.4.11 jointClass

- **Hinge** Defines a hinge joint. Requires an axis element to specify the axis and a body element which defines the body this hinge is connected to.
  - name The name of the joint.
    - **Use:** optional
    - **Range:** String
- **Slider** Defines a slider joint. Requires an axis element to specify the axis and a body element which defines the body this slider is connected to.
  - name The name of the joint.
    - **Use:** optional
    - **Range:** String

A.4.12 lightClass

- **Light** Specifies a light source.
  - diffuseColor Diffuse color definition, see A.4.23
    - **Use:** optional
  - ambientColor Ambient color definition, see A.4.23
    - **Use:** optional
  - specularColor Specular color definition, see A.4.23
    - **Use:** optional
  - x The x coordinate of the light source.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - y The y coordinate of the light source.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - z The z coordinate of the light source.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - constantAttenuation The constant attenuation of the light.
    - **Default:** 1
    - **Use:** optional
    - **Range:** $[0, MAXFLOAT]$
  - linearAttenuation The linear attenuation of the light.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[0, MAXFLOAT]$
  - quadraticAttenuation The quadratic attenuation of the light.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[0, MAXFLOAT]$
  - spotCutoff The opening angle of the light spot cone.
    - **Units:** radian, degree
    - **Default:** 180degree
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - spotDirectionX The x direction of the light spot.

        - **Units:** mm, cm, dm, m, km
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - spotDirectionY The y direction of the light spot.
        - **Units:** mm, cm, dm, m, km
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - spotDirectionZ The z direction of the light spot.
        - **Units:** mm, cm, dm, m, km
        - **Default:** -1
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - spotExponent
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[0, 128]$

A.4.13 massClass

- **BoxMass** Specifies a box-shaped mass.
    - name The name of the mass.
        - **Use:** optional
        - **Range:** String
    - value The mass of the box.
        - **Units:** g, kg
        - **Use:** required
        - **Range:** $[0, MAXFLOAT]$
    - width The width of the box.
        - **Units:** mm, cm, dm, m, km
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - height The height of the box.
        - **Units:** mm, cm, dm, m, km
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - depth The depth of the box.
        - **Units:** mm, cm, dm, m, km
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **InertiaMatrixMass** Specifies a mass with a given inertia matrix (which is symmetric and positive-definite).
  - name The name of the mass.
    - **Use:** optional
    - **Range:** String
  - value The total mass.
    - **Units:** g, kg
    - **Use:** required
    - **Range:** $[0, MAXFLOAT]$
  - x The x coordinate of the center of mass.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - y The x coordinate of the center of mass.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - z The x coordinate of the center of mass.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - ixx Moment of inertia around the x-axis when the object is rotated around the x-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - ixy Moment of inertia around the y-axis when the object is rotated around the x-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - ixz Moment of inertia around the z-axis when the object is rotated around the x-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - iyy Moment of inertia around the y-axis when the object is rotated around the y-axis.
    - **Units:** $g * mm^2$, $kg * m^2$
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

  - iyz Moment of inertia around the z-axis when the object is rotated around the y-axis
    * **Units:** $g * mm^2$, $kg * m^2$
    * **Default:** 0
    * **Use:** optional
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - izz Moment of inertia around the z-axis when the object is rotated around the z-axis.
    * **Units:** $g * mm^2$, $kg * m^2$
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **Mass** Specifies a mass.
  - name The name of the mass.
    * **Use:** optional
    * **Range:** String

- **SphereMass** Specifies a sphere-shaped geometry.
  - name The name of the mass.
    * **Use:** optional
    * **Range:** String
  - value The mass of the sphere.
    * **Units:** g, kg
    * **Use:** required
    * **Range:** $[0, MAXFLOAT]$
  - radius The radius of the sphere.
    * **Units:** mm, cm, dm, m, km
    * **Use:** required
    * **Range:** $[0, MAXFLOAT]$

A.4.14 materialClass

- **Material** Specifies the physical properties of a material.
  - name The name of the material.
    * **Use:** optional
    * **Range:** String

A.4.15 motorClass

- **PT2Motor** Instantiates a motor using the PT2 model.
  - T The time constant of the system.
    * **Use:** required
    * **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - D The damping constant of the system.

        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - K The steady state gain of the system.
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - V The maximum velocity of this motor.
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - F The maximum force of this motor.
        - **Units:** N
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **ServoMotor** Instantiates a position-controlled servo motor.
    - maxVelocity The maximum velocity of this motor.
        - **Units:** radian/s, degree/s
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - maxForce The maximum force of this motor.
        - **Units:** N
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - p The p value of the motor's pid interface.
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - i The i value of the motor's pid interface.
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - d The d value of the motor's pid interface.
        - **Default:** 0
        - **Use:** optional
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$

- **VelocityMotor** Instantiates a velocity-controlled motor.
    - maxVelocity The maximum velocity of this motor.
        - **Units:** radian/s, degree/s
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$
    - maxForce The maximum force of this motor.
        - **Units:** N
        - **Use:** required
        - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.16 rotationClass

- **Rotation** Specifies the rotation of an object relative to its parent.
  - x Rotation around the x-axis.
    - **Units:** radian, degree
    - **Default:** 0degree
    - **Use:** optional
  - y Rotation around the y-axis.
    - **Units:** radian, degree
    - **Default:** 0degree
    - **Use:** optional
  - z Rotation around the z-axis.
    - **Units:** radian, degree
    - **Default:** 0degree
    - **Use:** optional

A.4.17 sceneClass

- **Scene** Describes a scene and specifies the controller of the simulation.
  - name The identifier of the scene object (must always be *RoboCup*).
    - **Use:** optional
    - **Range:** String
  - controller The name of the controller library (without prefix *lib*; in our case it is *SimulatedNao*).
    - **Use:** optional
    - **Range:** String
  - color The background color of the scene, see A.4.23.
    - **Use:** optional
  - stepLength The duration of each simulation step.
    - **Units:** s
    - **Default:** 0.01s
    - **Use:** optional
    - **Range:** $(0, MAXFLOAT]$
  - gravity Sets the gravity in this scene.
    - **Units:** $\frac{mm}{s^2}$, $\frac{m}{s^2}$
    - **Default:** $-9.80665\frac{m}{s^2}$
    - **Use:** optional
  - CFM Sets the ODE cfm (constraint force mixing) value.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - ERP Sets the ODE erp (error reducing parameter) value.

    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - contactSoftERP Sets another erp value for colliding surfaces.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - contactSoftCFM Sets another cfm value for colliding surfaces.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $[0, 1]$
  - bodyCollisions Whether collisions between different bodies should be detected.
    - **Default:** true
    - **Use:** optional
    - **Range:** true, false

A.4.18 setClass

- **Set** Sets a placeholder referenced by the attribute *name* to the value specified by the attribute *value*.
  - name The name of a placeholder.
    - **Use:** required
    - **Range:** String
  - value The value the placeholder is set to.
    - **Use:** required
    - **Range:** String

A.4.19 solverClass

- **QuickSolver** Requires that the simulation uses ODE's QuickStep method.
  - iterations The number of iterations that the QuickStep method performs per step.
    - **Default:** -1
    - **Use:** optional
    - **Range:** $(0, MAXINTEGER]$
  - skip Controls how often the normal solver is used instead of QuickStep.
    - **Default:** 1
    - **Use:** optional
    - **Range:** $(0, MAXINTEGER]$

A.4.20 surfaceClass

- **Surface** Specifies the visual properties of a material.
  - diffuseColor The diffuse color, see A.4.23.
    - **Use:** required
  - ambientColor The ambient color, see A.4.23.
    - **Use:** optional
  - specularColor The specular color, see A.4.23.
    - **Use:** optional
  - emissionColor The color of the emitted light, see A.4.23.
    - **Use:** optional
  - shininess The shininess value.
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[0, 128]$
  - diffuseTexture The path to a texture.
    - **Use:** optional
    - **Range:** String

A.4.21 translationClass

- **Translation** Specifies a translation of an object relative to its parent.
  - x Translation along the x-axis.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - y Translation along the y-axis.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - z Translation along the z-axis.
    - **Units:** mm, cm, dm, m, km
    - **Default:** 0m
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.22 userInputClass

- **UserInput** Combines an actuator and a sensor. The values set for the actuator are directly returned by the sensor. Using the actuator view, this allows to feed user input to the controller.
  - name The name of the user input.
    - **Use:** optional
    - **Range:** String
  - type The kind of data for which user input is provided.
    - **Default:** length
    - **Use:** optional
    - **Range:** angle, angularVelocity, length, velocity, acceleration
  - min The minimum value that can be set.
    - **Units:** Units matching the type
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - max The maximum value that can be set.
    - **Units:** Units matching the type
    - **Use:** required
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$
  - default The value returned by the sensor if no value is set in the actuator view.
    - **Units:** Units matching the type
    - **Default:** 0
    - **Use:** optional
    - **Range:** $[-MAXFLOAT, MAXFLOAT]$

A.4.23 Color Specification

There are two ways of specifying a color for a color-attribute.

- **HTML-Style** To specify a color in html-style the first character of the color value has to be a # followed by hexadecimal values for red, blue, green (and an optional fourth value for the alpha-channel). These values can be one-digit or two-digits, but not mixed.
  - #rgb e. g. #f00
  - #rgba e. g. #0f0a
  - #rrggbb e. g. #f80011
  - #rrggbbaa e. g. #1038bc80
- **CSS-Style** A css color starts with rgb (or rgba) followed by the values for red, green, and blue put into parentheses and separated by commas. The values for r, g, and b have to be between 0 and 255 or between 0% and 100%, the a-value has to be between 0 and 1.
  - rgb(r, g, b) e. g. rgb(255, 128, 0)
  - rgba(r, g, b, a) e. g. rgba(0%, 50%, 75%, 0.75)