HULKs Team Research Report 2021

Andrea Essig, Pascal Gleske, Konrad Nölle, Maximilian Schmidt, Hendrik Sieck, Bennett Wetters

HULKs e. V.
Hamburg University of Technology,
Am Schwarzenberg-Campus 3,
21073 Hamburg, Germany

Contents

Chapter 1

Introduction

HULKs is a RoboCup SPL team from the Hamburg University of Technology. The team was formed in April 2013 and consists of students and alumni.

HULKs team members originate from various fields of study. Therefore our research interests are spread across several disciplines, reaching from the design of our own framework to the development of dynamic motion control. Over the past seasons we improved our performance continuously and subsequently won 3rd place at the GermanOpen 2019 and reached the RoboCup 2019 quarterfinals.

The past season was heavily influenced by COVID-19 related circumstances and therefore taking part in remote events was the main focus of our efforts.

This report serves as partial fulfillment of the pre-qualification requirements for RoboCup 2022. For this purpose, it is accompanied by a recent snapshot of the code.

The remainder of this document is organized as follows. Chapter 2 outlines how to run the code release on a NAO robot and in simulation. Chapter 3 explains the underlying framework of our code base. Image processing algorithms are presented in chapter 4. Chapter 5 explains state estimation and the behavior. Chapter 6 outlines different motion modules. Supplementary debug and visualization tools are presented in chapter 7.

Chapter 2

Running the Code Release

This section contains all information required to run our code release on a NAO robot, inside SimRobot, in Webots and in replay mode on a Linux machine. The HULKs Code Release of 2021 only supports NAO v6, older versions are no longer supported.

2.1 Prerequisites

To build the robot software, a recent Linux operating system is required. While it is possible (however not officially supported) to run our code within SimRobot on Microsoft Windows, building the toolchain, setting up robots and building for the NAO robot is only possible on Linux. This section lists all packages required for specific tasks.

The following packages are required to build the code for local targets (see section 2.4):

C++17 compiler (GCC $\geq$ 7, Clang $\geq$ 4), git, CMake, ccache, curl, unzip, tar, bzip2

To compile and run the code release with SimRobot the following additional dependencies have to be installed:

libxml2, qt5-base, qt5-svg, ode, glew

Additional dependencies (`opencv`, `opusfile`, `portaudio`) are handled by `vcpkg` [10]. To install the required dependencies execute `scripts/setup` which downloads, compiles and installs the required packages.

To build the `SDK` required to cross compile for the NAO robot refer to section 2.3.

2.2 Downloading the Repository

Our code release is hosted in a public repository on GitHub. To clone and checkout the correct version, the following commands can be executed:

Listing 2.2.1

```
git clone https://github.com/HULKs/HULKsCodeRelease
cd HULKsCodeRelease
git checkout coderelease2021
```

2.2.1 Configure the repository

It may be noted that our team number as well as serial numbers have been replaced with placeholders in all scripts and configuration files coming with the code release. In order to deploy the code on a NAO, the following files need to be modified by replacing these placeholders:

Listing 2.2.2

```
etc/configuration/location/default/id_map.json (explanation below)
etc/configuration/location/default/head/default/Brain.json
src/Data/PlayerConfiguration.hpp
src/Vision/ReplayDataProvider/ReplayDataProvider.cpp
tools/pepsi/src/naossh.rs
tools/pepsi/src/util.rs
```

The `id_map.json` needs to contain the serial numbers of all robot parts. This way, a robot is able to load the correct configuration files whenever he plays with a replaced body. The file should look like the example in listing 2.2.3. All occurrences of `####` need to be replaced by the last 4 digits of the serial number of the robot part.

Listing 2.2.3

```
{
"idmap.nao": [
  {
    "bodyid": "P#######A##S##A#####",
    "headid": "P#######A##S##A#####",
    "name": "NAMEnao1"
  },
  {
    "bodyid": "P#######A##S##A#####",
    "headid": "P#######A##S##A#####",
    "name": "NAMEnao2"
  }]
}
```

2.3 Yocto

For NAO operating system image generation and construction of the cross-compilation SDK, the Yocto Project [23] is used. The Yocto Project is an open source project that helps creating custom linux-based systems regardless of the target architecture. This also includes building the linux kernel, all necessary tools, the final OPN image and the SDK for cross compilation targeting the NAO.

2.3.1 Getting Started

The Yocto Project leverages BitBake as task execution engine and offers an abstraction layer to modify and extend existing build configurations. Combined with OpenEmbedded [12], the entire worktree is structured in several layers 2.3.3

Create a directory for the upcoming setup and build phases e.g. worktree/. Make sure there is at least 100 GB empty disk space available.

For project setup the siemens/kas [17] framework is used. To setup kas use the containerized (podman/docker) version via the kas-container script [18] and store it inside the worktree directory. Alternatively setup kas via a python-pip installation, follow the installation steps in the user guide [19].

The meta-hulks layer ships a kas-project.yml project description file. This file defines the project structure kas has to setup for the Yocto build phase. The code release contains the initial layer (meta-hulks) for project generation. Clone the code release into some directory used for the Yocto build e.g. worktree/nao.

Listing 2.3.1

```
git clone https://github.com/HULKs/HULKsCodeRelease worktree/nao
cd worktree/nao
git checkout coderelease2021
cd yocto
```

Afterwards, kas will be used for repository management. Thus, use kas to checkout all the necessary layers.

Listing 2.3.2

```
./kas-container checkout meta-hulks/kas-project.yml
```

The kas tool should download and setup all sources in the respective workspace.

Listing 2.3.3

```
worktree/
|-- build
|-- kas-container
```

```
|-- meta-clang
|-- meta-congatec-x86
|-- meta-hulks
|-- meta-intel
|-- meta-nao
|-- meta-openembedded
`-- poky
```

The NAO v6 uses `LoLA` and `HAL` for communication with the chestboard. All these binaries and libraries necessary to operate the NAO properly are shipped with the `.opn` RoboCupper image and are **not** included in this repository. Contact the RoboCup SPL TC to get this image. To extract the necessary binaries the `extract_binaries.sh` script is used. This script fetches all binaries from inside the RoboCupper image and collects them in an archive for the upcoming build phase. To generate the archive containing the aldebaran binaries run:

Listing 2.3.4

```
cd meta-nao/recipes-aldebaran/aldebaran/
mkdir -p aldebaran-binaries
./extract_binaries.sh -o aldebaran-binaries/aldebaran_binaries.tar.gz nao
    -2.8.5.11_ROBOCUP_ONLY_with_root.opn
```

2.3.2 Building the Image

Execute `kas` from inside the `worktree` directory referencing the `kas-project.yml` to enter the build environment.

Listing 2.3.5

```
./kas-container shell meta-hulks/kas-project.yml
```

The workspace is now able to build a NAO image from inside the `kas` container.

Listing 2.3.6

```
bitbake nao-image
```

This generates and executes all necessary tasks and targets to construct a proper `.opn` file. The initial build phase might take several hours depending on the performance of your build machine and your internet connection. `BitBake` uses a very elaborated caching strategy to speed up following builds of targets. Thus small changes afterwards might only take a few minutes.

As soon as the build has successfully finished, the image can be deployed. After `BitBake` ran all tasks up to `nao-image` a new `.opn` file is generated in `worktree/build/tmp/deploy/images/nao-v6/nao-image-HULKs-OS-[...].ext3.gz.opn`. The image can now be flashed to a USB flash drive.

Listing 2.3.7

```
dd if=image_path.opn of=/dev/sdb bs=4M status=progress oflag=sync
```

2.3.3 Targeting the NAO

The `Yocto Project` contains tasks to build a proper `SDK` to use for development. `BitBake` can be used from inside the `kas` container runtime to construct the `SDK`.

Listing 2.3.8

```
bitbake -c populate_sdk nao-image
```

This build phase may take several hours. After a successful build, the `SDK` is located at `worktree/build/tmp/deploy/sdk/HULKs-OS-toolchain-[...].sh`. To install the `SDK` run the script and follow the instructions. Afterwards, you are able to source the build environment and use the respective cross compilers.

2.4 Building the Robot Software

Currently, the code supports the following targets:

- `NAO` for compiling the code to run on a NAO with the `Yocto SDK`,
- `Webots` for building an executable which can be used within the `Webots` [3] simulator,
- `SimRobot` for targeting the `SimRobot` simulator,
- `Replay` loads a prepared dataset into the code and allows for deterministic testing.

There are three different build types:

- `Debug` compiles witout optimization, the resulting execution is very slow, for debugging purposes only.
- `Develop` compiles with optimization, retains assertions, used during development.
- `Release` is used in actual games and removes assertions. Use this for profiling (see section 7.3).

2.4.1 Webots

Our code release can be compiled against the `Webots` simulator. Our repository also includes world files in which the code can be executed. The first step for getting the code running, is to install the `Webots` simulator from official sources, e.g. from your Linux distribution's package manager. Once installed, our code can be setup for the `Webots` installation (from repository root):

Listing 2.4.1

```
scripts/setup -t Webots
```

Then the code base can be built for `Webots` by executing the compile script.

Listing 2.4.2

```
scripts/compile -t Webots -b <BuildType>
```

For running the code, first start `Webots` and open the world file `webots/worlds/penalized.wbt`. Now, you can click `Play` to start the simulation. Press `Ctrl+Shift+C` to press the chest button. Keyboard commands sometimes do not work instantly in `Webots`. You need to focus the 3D simulation view and then press the keys. The keyboard is only polled at 100 Hz simulation time so it may happen that you need to press the keys longer than usual.

2.4.2 SimRobot

Our repository comes with its own version of SimRobot [2] which one needs to compile first. This can be done by using the setup script as follows (from repository root):

Listing 2.4.3

```
scripts/setup -t SimRobot
```

Then the code base can be built for SimRobot by executing the compile script.

Listing 2.4.4

```
scripts/compile -t SimRobot -b <BuildType>
```

To start SimRobot, simply run the executable inside the build folder. All scenes are stored in `tools/SimRobot/Scenes`.

Listing 2.4.5

```
cd tools/SimRobot/build
./SimRobot
```

2.4.3 Replay

To compile the code for replay one can execute the following commands:

Listing 2.4.6

```
scripts/setup -t Replay
scripts/compile -t Replay -b <BuildType>
```

Details on how to record and use replay files can be found in section 3.6.

2.4.4 NAO

The scripts need to know the location of the `SDK` targeting the NAO platform. Usually the `scripts/sdk` script is used for this purpose and automatically downloads the `SDK` installer from HULKs infrastructure in our lab. To skip the downloading process, place the `SDK` installer at `./sdk/downloads/` and execute the script. Finally, the code can be configured and built:

Listing 2.4.7

```
scripts/setup -t NAO
scripts/compile -t NAO -b <BuildType>
```

2.5 Setting up a Robot

The NAO is set up by flashing the NAO image which is constructed using Yocto. Once the flashing procedure terminated successfully, the NAO should restart itself and be ready to run the code. If the robot was able to find its head id in the `id_map.json`, the robot's hostname changes to the name specified in the `id_map.json`, while the IP address will change to `10.1.TEAMNUMBER.NAONUMBER` during this process. Otherwise the robot uses DHCP to query for an IP address.

2.6 Uploading the Robot Software

The last step is to upload the code to the NAO and run it. This can be done by running the `pepsi`-tool:

Listing 2.6.1

```
scripts/pepsi upload <NAONUMBER>
```

The tool will upload the compiled code and configuration files to `nao<NAONUMBER>` and restart the hulks-service. You can pass multiple numbers or IP addresses separated by spaces and pepsi will upload the code to all given robots. Pepsi makes some assumptions about names, numbers and IP-addresses. If your naming scheme differs from ours, adjust the code accordingly.

2.7 Pepsi

Pepsi is responsible for all NAO related configuration, setup and deployment. The tool is based on the Rust programming language and uses several crates to provide an easy command-line interface. You can use the `--help` flag to get an overview of Pepsi's capabilities. It provides nine sub-commands:

connect Connect to a NAO via ssh

dump-completions Dump shell completions and exit

help Prints a help page or the help of the given sub-command(s)

hulk Control the HULK service

logs Logging on the NAO

playernumber Change player numbers of the NAOs in local configuration

shutdown Shutdown or reboot the NAO

upload Upload hulk to NAOs

wlan Control wireless network on the NAO

2.8 MATE

For the purpose of debugging a tool named MATE (Monitor And Test Environment, see section 7.2 for technical details) exists. It can be found in: `tools/mate`. Before MATE can be started, ensure all Python requirements mentioned in `requirements.txt` are installed.

Listing 2.8.1

```
pip install -r requirements.txt
```

To start MATE run:

Listing 2.8.2

```
cd tools/mate
python run.py
```

After starting MATE connect to a NAO or SimRobot session. By clicking the *New* button a new `Panel` can be opened. To see live images from the robot add an image `Panel` with the desired image key. There is also a feature called layouts to save and load arrangements of panels. These can be saved and loaded in the top control bar.

Chapter 3

Framework

The overall structure of the codebase can be seen in fig. 3.1. To use the framework for different robots, offline processing, or simulation, the `RobotInterface` serves as an abstraction layer. It exposes methods for acquiring sensor data, camera images, and for outputting joint and LED commands for controlling the robot. The framework also provides `Debug` (see section 3.4) and `Configuration` (see section 3.5) capabilities.

3.1 Module Architecture

The majority of algorithms for robot control are organized as independent units called modules. Every module has access to the `Debug`, `Configuration` and `RobotInterface` instances via functions from its base class.

The relationship between modules can be modeled as a bipartite data flow graph made of modules and the data types they exchange. `DataType`s are stored in a `database` for each module manager[1] as can be seen in fig. 3.2. The relation between modules and data types can either be a module producing a data type or depending on it. This is realized by two template classes `Dependency` and `Production` of which a module has member variables for the data types it wants to access. There is also the `Reference` class that takes data from the last cycle. This can be used to break cyclic dependencies.

Each module must have a `cycle` method which executes code running every frame. The order of the cycles is determined at runtime by topological sorting to guarantee that the dependencies of all modules have been produced before their execution. Before a data type is produced, it will be reset to a defined state.

In general, the semantics of the module architecture are similar to the one presented in [16], but the implementation is completely macro-free and instead based on templates. This has the advantage that no design specific language is introduced and every person familiar with C++ can easily comprehend code, at the cost of more verbose declarations of dependencies and productions.

[1] Brain and motion implement the `ModuleManagerInterface`.

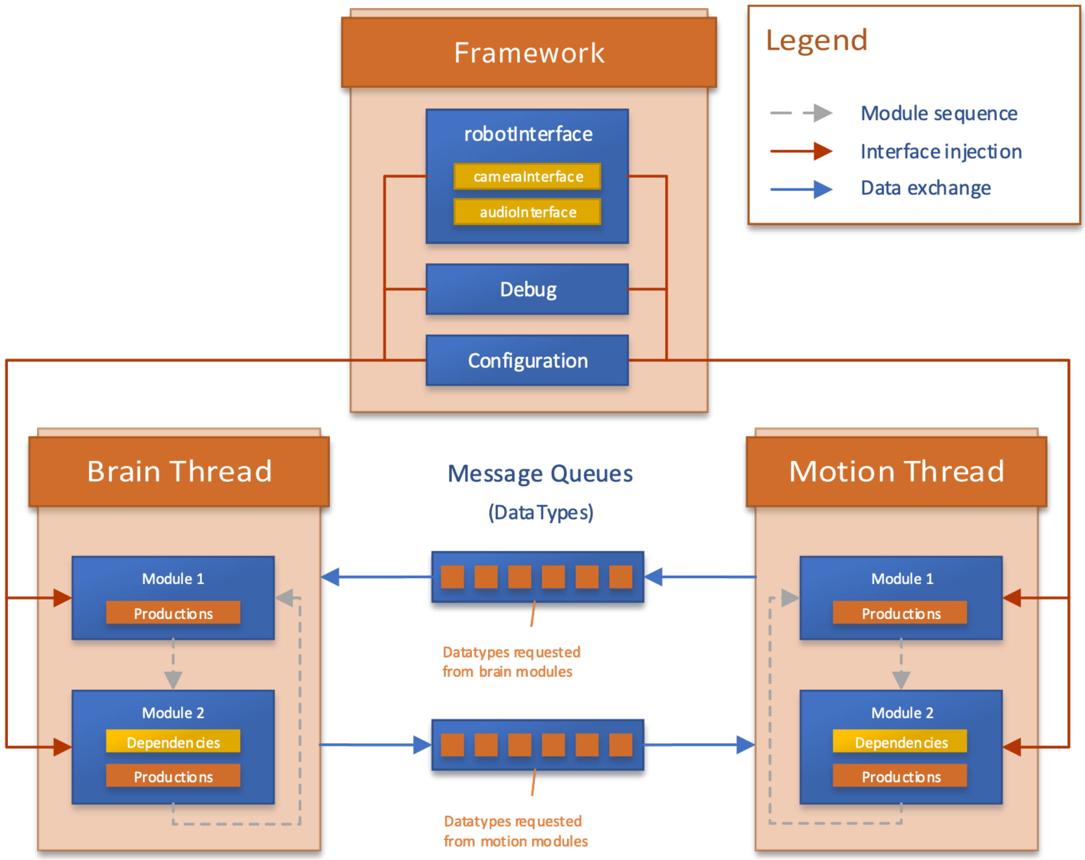


Figure 3.1: Overview over the general module architecture of our framework.

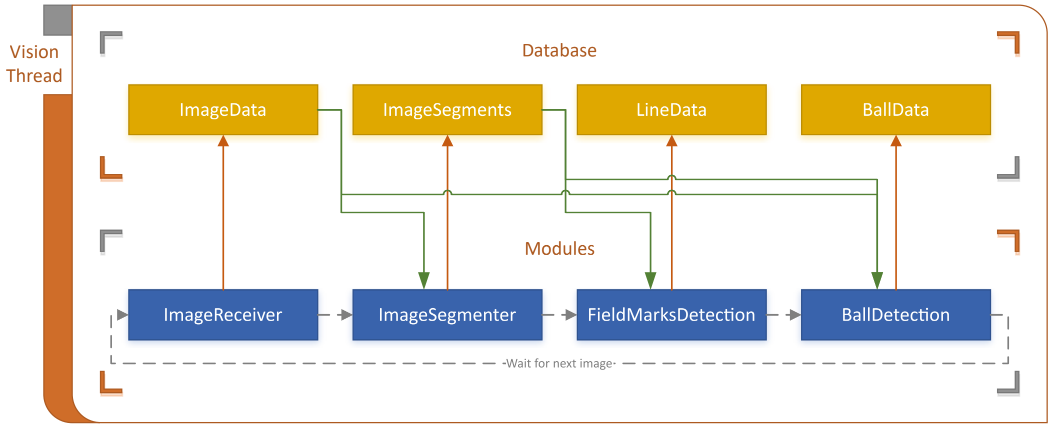


Figure 3.2: An example of a module execution order and the associated `DataTypes` stored in the `Database`. Productions are marked with orange arrows and dependencies with green ones.

Modules can be enabled or disabled before the startup of the `hulk` executable. To do this, every module has a static member variable that denotes its name. With this information in place, a JSON file can be used to change whether a module should run or not. The file structure is explained in section 3.1.1.

3.1.1 Module Setups

It is possible to create multiple module setups. All module setups reside in `etc/configuration/location/default/`. Each filename has to start with `moduleSetup_`. The active setup can be configured inside the `etc/configuration/default/tuhh_autoload.json` entry: `moduleSetup`. Every module has to be listed inside the `moduleSetup_default.json` if it is to be enabled by default or not. To override, edit the specialized module setup file.

The `fullVisionFake` for example disables all vision modules and receives the position, ball and team data directly from the fake data interface. This is especially useful if one wants to simulate the behavior of a whole team in real-time.

3.2 Threads

Autonomous interaction with the environment requires evaluation of a variety of sensor inputs as well as updating the actuator control outputs at an adequate rate. Since the update rates of camera hardware and chestboard differ, our framework features two different threads, each of which is synchronized with the associated hardware update rate. Additionally, messaging infrastructure is provided to safely share data between threads.

The motion thread processes all data provided by LoLA[2] or HULA[3]. These are accelerometer, gyroscope and button interface sensor inputs as well as joint angle measurements and sonar data. Any data provided by HULA is updated every 12ms. Therefore, this is the update rate at which the motion thread is scheduled.

The brain thread processes camera images. Since each camera provides image data at an update rate of 30Hz, the brain thread is running at twice that frequency, using images of both cameras. In addition to the image processing algorithms, the brain thread also runs the modeling and behavior modules, processing incoming information and reevaluating the world model.

3.3 Messaging

Data types produced (and needed) in different threads have to be transferred between them. This is done by having queues connecting the `databases` of their module managers (see fig. 3.1). After each complete cycle of a module manager, it appends the requested data types to the queue of the desired recipient. These data types are then imported into the database of the other module manager when its next cycle starts.

The exchanged data types are determined by each module manager at program startup. All needed data types that are not produced in the same module manager are requested from all connected senders. Only the module manager that originally produces a data type will respond.

3.4 Debugging

Our framework features a variety of debugging and configuration features. It is possible to export variables (including images) with an associated name so that they can be sent to a PC or written to a log file. We can also use *playAudio* to play predefined sounds. On the PC side, MATE (section 7.2) connects to the NAO via a TCP socket (or to a SimRobot/Webots instance via a UNIX socket). If an image or value is subscribed to, it will be transferred to the client as fast as possible. This depends on the network link and the amount of data that needs to be sent.

3.4.1 Export a variable

Listing 3.4.1 is an example showing how to use the variable export:

[2]Low Level API

[3]HULKs Level API, a proxy around LoLA

Listing 3.4.1

```
void cycle()
{
  int variable = 10;
  debug().update(mount_ + ".variable", variable);
}
```

3.4.2 Export an image

Listing 3.4.2 is an example showing how to export an image:

Listing 3.4.2

```
void cycle()
{
  Image image({640, 480});
  debug().sendImage(mount_ + ".image", image);
}
```

3.4.3 Play Audio File

Listing 3.4.3 is an example showing how to play a sound:

Listing 3.4.3

```
void cycle()
{
  debug().playAudio(mount_ + ".playAudio", AudioSound::WEEEEE);
}
```

3.5 Configuration

We have a configuration system that loads parameters from JSON files named identically to the corresponding module. It parses configuration files in directories specific to a NAO head, NAO body or a location such as RoboCup 2019 SPL_A. Parameters that are set in more specific files will override parameter values from more generic files. It is also possible to change parameters at runtime via the aforementioned MATE tool. Receiving new values can cause a callback so that calculations based on these parameters can be reevaluated.

If there are specific JSON files to load, their location name can be specified inside `etc/configuration/location/default/sdk.json` in the parameter `location`. This is used to load configuration files from a subfolder of `etc/configuration/location/`

with that name. Each location folder can add `head` and `body` folders. Each of these folders can consist of a `default` subfolder and subfolders with a robot name as specified in section 2.2.1. All values specified inside the location-based configuration files will overwrite the ones loaded from the `default` location. Also the `default` location can contain default parameters for each robot, e. g., the walking parameters.

Generally speaking, configuration parameters can be arbitrary JSON objects. Sometimes it is handy to have a parameter choose between two values depending on an external condition, e. g., the game state. For this usecase a `ConditionalParameter` is available. A conditional parameter can be given a callback that is evaluated each time the parameter's value is used. The value is selected based on the return value of said callback.

3.6 Replay Recorder

To avoid processing data directly on the robot and logging results to the robot's storage, the `ReplayRecorder` stores all sensor readings and records individual frames. The collected frames can be used later by the HULKs framework in `Replay mode`. This is done by a virtual robot interface, which reads the stored frames as image and sensor data and allows revalidation of algorithms on previously recorded data.

After compiling for target `replay` (see Section 2.4.3), it can be started with:

Listing 3.6.1

```
./build/replay/current/src/tuhhsdk/tuhhReplay webots/controllers/
    tuhhReplay/off.json
```

This loads the minimal `off.json` without any images. If you have recorded images, you should get a JSON and can load that instead. Afterwards you can connect to replay via MATE at localhost.

3.7 NAO v6

In 2019 SoftBank Robotics released the v6 robot version. It features a new software version and new programming interfaces. This version discontinued the DCM and replaced it with LoLA.

3.7.1 LoLA

The joints can now be directly controlled via a `MsgPack` [4] protocol over a UNIX socket at `/tmp/robocup`. When the connection to LoLA is established, it sends hardware information about the specific robot and the current joint sensor readings. Afterwards, LoLA will wait for an input frame which requests new joint positions. When receiving a frame, it sends the new sensor data readings in the next cycle. LoLA sends new data every 12 ms.

3.7.2 HULA

The HULKs Level API (HULA) is a process developed by HULKs which connects to LoLA and proxies all messages to other processes. The code can be found at `tools/hula`. HULA is able to accept connections from multiple clients rather than a single connection like in LoLA. HULA terminates the MsgPack protocol from LoLA and provides a more compressed binary message protocol via a UNIX socket at `/tmp/hula`. Messages in the binary protocol interpret buffer memory as C-style structs which are defined in `tools/hula/src/lola.rs`. The message proxy allows to intercept and inject data into the communication between connected robot control softwares and LoLA, i.e. extracting battery status and injecting LED commands. Since HULA outlives its connected clients, it allows to display information via LEDs and send network messages even if e.g. our robot control software is not running. At startup, HULA connects to LoLA and begins to extract battery information. This battery information is used in two ways: HULA uses it to display a battery state indication via the head LEDs and the battery information is transmitted with other data via the network, i.e. OS versions, NAO head ID, or systemd unit states. These beacon messages are sent over UDP multicast at `224.0.0.42:4242` in a JSON format. Beside the head LEDs, HULA indicates if no clients are connected by playing a red pulsing animation on the eye LEDs. If at least one client is connected, the client has full control over the eye LEDs and all other actuators.

3.7.3 Cameras on the NAO

The cameras are connected to the CPU via a USB - MIPI converter. While the top camera is connected via USB3, the bottom camera is connected via USB2 due to the absence of a second USB3 port.

3.8 SimRobot

SimRobot [2] is a simulator developed at the University of Bremen and the German Research Center for Artificial Intelligence. The `SimRobotInterface` integrates the HULKs code into the simulator. It is possible to simulate multiple robots at once to evaluate team behavior.

Sources of noise to the visual side of the simulation bring it closer to reality. The idea behind this is to obtain better estimates for the robustness of vision algorithms tested inside SimRobot.

3.8.1 Motion Blur

Rendering realistic camera images in simrobot is hard. However, adding motion blur helps in adding a relatively realistic sense of motion to every frame. This effect is attained by mixing the previous frame into the current one, using the arithmetic mean on the pixelvalues. In doing so, lines, balls and other objects are harder to detect when the robot or the object is in motion, as their outlines are blurred and the apparent size can vary, similar to the effects observed on a real robot. When not testing anything related to vision, this effect can be turned off.

In order to reduce the considerable performance impact, instead of iterating over every pixel and color channel, bit-wise operators on larger datatypes (i. e. `int64` instead of `byte`) are used. This reduces the cycles necessary to iterate over the whole image.

3.8.2 New Field Texture

To make the field less monotone, a shadow-texture with a higher resolution image resembles a real grass-texture more closely. The shadow used to render the lines still uses the old shadow definition.

Chapter 4

Vision

Vision is the software component that contains image processing algorithms. It is split into several modules where each of them has a special task. The overall procedure is as follows: Raw camera images are acquired from the hardware interface and the camera matrix matching this image is constructed. Image preprocessing consists of determining the field color (see section 4.6) and segmentation (see section 4.5) of the image to reduce the amount of data to be processed by subsequent object detection algorithms. Subsequently the field border, i. e., the area where the carpet ends, is identified (see section 4.7). Afterwards, penalty spots, field lines, the center circle, the ball and robots are detected. For further reference see section 4.9, section 4.8, section 4.10, section 4.11, and section 4.12, respectively.

4.1 Image Data

The image data is provided through the `RobotInterface` which holds both top and bottom camera. The `Camera` class is configured such that 640×320 pixel images are received in the `YUYV` format. The `RobotInterface` provides a method which returns the current camera object for use in the next vision cycle. Hereby, the camera object with the oldest, not yet processed data gets returned.

The first module to run in the vision pipeline is called `ImageReceiver`. It calls the `RobotInterface` to receive a new image and makes the data available to all other vision modules by producing the `ImageData` DataType.

As the image is in `YUYV` format, all modules and its algorithms use native `YCbCr422` pixel data without treating any Y-values as padding. However, conversion to `YCbCr444` is needed to generate debug images. This is currently done on the robot and not in the corresponding debug tools.

4.2 Camera Configuration

Several values concerning camera configuration are changed for the specific RoboCup SPL use case. Using UVC Extension Units, it is possible to directly set register values on the cameras of the NAO.

`Digital Effects` are a camera internal feature to manipulate the image after capture to make the image more pleasing to the human eye. However, this distorts the image and increases noise. As this is undesired, `Digital Effects` are disabled.

Regarding white balance, the camera provides a configuration parameter called `AWBBias`. This parameter intends to correct the white balance to give the image a more natural look leading to a green tint. For field color detection using chromaticity a neutral white point is required. Therefore, the `AWBBias` is completely switched off.

4.2.1 Auto Exposure

To achieve a consistent brightness in all images the camera's auto exposure is used. The behavior of the auto exposure algorithm is tuned using various parameters. In default configuration the NAO's camera uses the entire image for auto exposure control. In RoboCup SPL context this is not always useful as the upper part of the upper camera can show very bright windows. It is possible to set 16 different weights for a 4 by 4 grid to control auto exposure. For the upper camera we only consider the lower part of the image to ensure a well exposed field.

4.3 Camera Calibration

The two cameras of the NAO have to be calibrated for optimal performance. Camera calibration consist of intrinsic (determining focal lengths and centers) and extrinsic stages (adjusting camera pose matrix using the kinematic chain).

While intrinsic calibration is needed less frequently, the extrinsic stage has to be performed quite often as the cameras tend to physically shift during game play, especially after a fall. The following will explain the calibration procedure for the case of a single camera. In our implementation, the procedure is repeated for both cameras.

Usage of `Numpy` and `OpenCV 3` reduces implementation time by using built-in non-linear solvers such as Levenberg-Marquardt and intrinsic calibration functions based on [24] and others.

For extrinsic calibration an alternative manual method is used. It is based on manipulating sliders for roll, pitch and yaw to match a projected penalty box to the real penalty box while standing on a predefined position in the center of the field.

4.3.1 Kinematic Chain

The kinematic transformations from the ground point (generally positioned between the feet of the robot) to the cameras are crucial for determining distances and positions of detected features. Understanding of this kinematic chain is required in order to perform extrinsic calibration.

The following matrix names use a notation that denotes the initial and destination coordinate systems determined by the respective transformation matrix. For example, *camera2Ground* describes the transformation from the camera to ground; *camera2Head* is the transformation from camera to head after applying the extrinsic calibration in the form of a rotation matrix $R_{ext}(\alpha, \beta, \gamma)$. The transformation described by *camera2HeadUncalib* matrix differs between the two cameras. The composition of kinematic matrices from a ground point to the camera is as follows.

$$\begin{aligned} \mathit{camera2Head}(\alpha,\beta,\gamma) &= \mathit{camera2HeadUncalib} \times R_{ext}(\alpha,\beta,\gamma) && (4.1)\\ \mathit{camera2Ground}(\alpha,\beta,\gamma,t) &= \mathit{torso2Ground}(t) \\ &\quad \times \mathit{head2Torso}(t) \\ &\quad \times \mathit{camera2Head}(\alpha,\beta,\gamma) && (4.2)\\ \mathit{ground2Camera}(\alpha,\beta,\gamma,t) &= \mathit{camera2Ground}(\alpha,\beta,\gamma,t)^{-1} && (4.3) \end{aligned}$$

The matrices containing parameter t indicate their value may change over time due to the NAO moving. This distinction is important at the step of capturing images and kinematic matrices for a given frame.

4.3.2 Extrinsic Calibration

The automatic extrinsic calibration procedure is as follows:

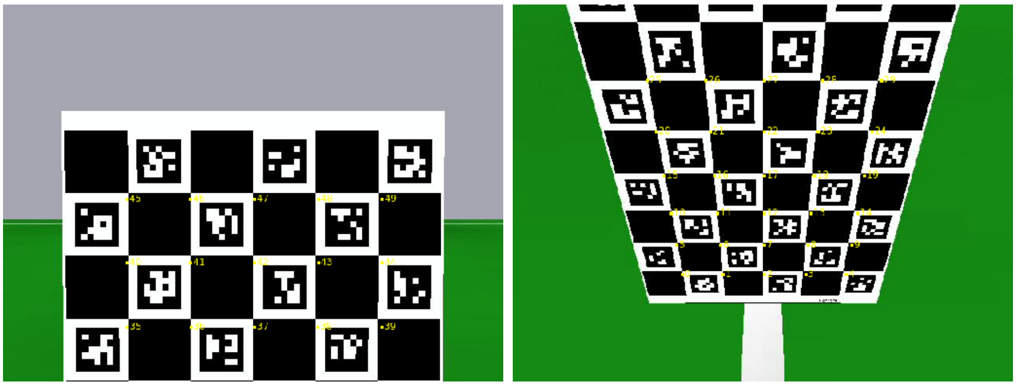
1. Images are captured from the debug tool which are then used for marker detection to obtain an array of 2D points called *DetectedPoints*.

2. Using marker ID values and a key-value set, the physical locations of the markers are determined and projected into the image plane where these form an array of 2D points called *ProjectedPoints*.

3. *ProjectedPoints* and *DetectedPoints* are sorted to form corresponding pairs.

4. Solving for the optimal extrinsic parameters involves minimizing the residual (eq. (4.4)) which can be represented as a non-linear least squares problem. An implementation of the Levenberg-Marquardt algorithm is used to obtain a numerical solution by adjusting α, β and γ. The existing calibration values are supplied as the initial guess to converge faster and to reduce risk of stopping at a local minimum.

$$\mathit{Residual} = \mathit{ProjectedPoints}(\alpha,\beta,\gamma) - \mathit{DetectedPoints} \tag{4.4}$$



(a) The calibration pattern as seen from the top camera.

(b) The calibration pattern as seen from the bottom camera.

Figure 4.1: The calibration pattern as seen via MATE in SimRobot. The yellow marks and text points to the projected corners (and their IDs) of the pattern with the current calibration values.

5. A callback function notifies the UI of the debug tool and the user is visually shown the projections of the markers with different colors for pre- (green) and post- (yellow) calibration (see fig. 4.1). The user is able to identify any potential problems and retry calibration if necessary.

6. Given that the result is satisfactory, the values are updated in the configuration.

The manual method for extrinsic calibration is as follows:

1. A standing robot is placed on the middle of the center circle facing a goal.

2. The penalty box is projected into the robot's view, based on current calibration and kinematic chain.

3. Roll, pitch and yaw parameters are manipulated until the projected penalty box matches the real penalty box.

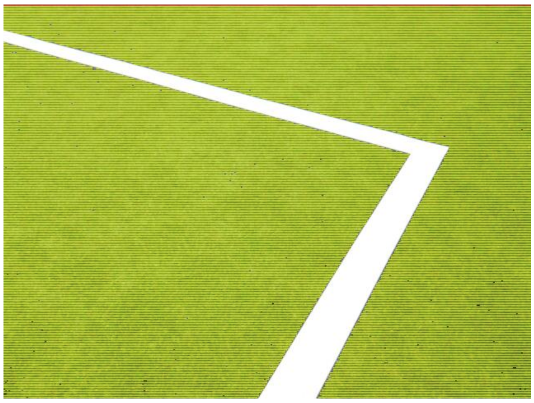
4.3.3 Intrinsic Calibration

The final two steps of extrinsic calibration are also present for intrinsic calibration. Similarly, the differences are displayed and the user can decide to accept or reject the calibration.

4.4 Robot Projection

Body parts, such as shoulders or knees, can appear in the image. To avoid percepts in these regions of the image, knowledge of the forward kinematics is used to project the

(a) Image of the top camera



(b) Image of the bottom camera

Figure 4.2: Field color detection using the green chromaticity approach. The detected field color is colored yellow.

arms, legs and torso into the image. This information is used by subsequent modules to ignore these areas.

The implementation makes use of a set of body contour points. These points are then projected into the image to calculate the approximated contour of the associated body part. All regions within the convex hull of the projected body contour points are marked as invalid by the image segmenter. Those invalid regions are ignored by subsequent vision modules, noticeably reducing the amount of candidates in the ball and line detection algorithms.

4.5 Image Segmentation

The image is segmented along vertical and horizontal scanlines. Vertical scanlines have a fixed distance to each other in pixel coordinates, whereas horizontal scanlines have a fixed distance in the robot coordinate system. The distance is approximated by a static projection matrix of a standing robot. A one-dimensional edge detection algorithm determines along a scanline where a region starts or ends. Subsequently, representative colors of all regions are determined by taking the median of certain pixels from the region. If that color is classified as the field color, the corresponding region is labeled as field region.

4.6 Field Color Detection

The vision pipeline provides two approaches to determine a pixel to be field colored. One approach uses green chromaticity as threshold. The other is based on the k-means clustering method but is not actively used.

4.6.1 Chromaticity

The analysis of multiple manually segmented images from different events and situations shows that the green chromaticity is the most suitable color channel [1] to separate field from non-field. The green chromaticity is defined as green divided by the sum of red, green and blue as shown in eq. (4.5).

$$g = \frac{G}{R + G + B} \tag{4.5}$$

Thresholds for green chromaticity in combination with thresholds for the red and blue chromaticity are used to classify pixels and segments. Results of this thresholding is depicted in fig. 4.2. This approach shows better results in challenging lighting conditions compared to the k-means clustering approach utilizing clustering pixels in the Cb-Cr plane.

4.6.2 *k*-means Clustering

To determine the field color a derived version of k-means clustering of the pixel colors is implemented in Module `OneMeansFieldColorDetection`. As there is only one cluster for the field color, the maximum size of this cluster is parameterized. The initial cluster value can either be given as a parameter or calculated dynamically. The dynamic calculation uses the peaks of the histograms over the Cb and Cr channels of pixels sampled below the projected horizon.

The update step is repeated up to three times. In each step the image is sampled. A sampled pixel is part of the cluster if the distance in the Cb-Cr plane is smaller than a threshold and the Y value is smaller than a configured multiple of the cluster's mean Y value. The mean of the cluster is shifted towards the mean of the pixels that meet these conditions. In order to avoid huge jumps of the cluster, e. g., when the robot is facing a wall or another robot that is very close, the shift of the cluster's mean is limited and remains unchanged in these cases.

4.7 Field Border Detection

The field border detection uses the upper points of the first regions on each vertical scanline that are labeled as field region. Through these points, a line is fitted with the RANSAC method. This chooses the line that is supported by most points. If enough points are left after the first iteration, i. e., points that do not belong to the first line, a second line is calculated with RANSAC. It is only added to the field border if it is approximately orthogonal in robot coordinates to the first one.

The module also creates a second version of the image regions that excludes all regions that are above the field border or labeled as field. The remaining regions are the ones that are most likely to contain relevant objects.

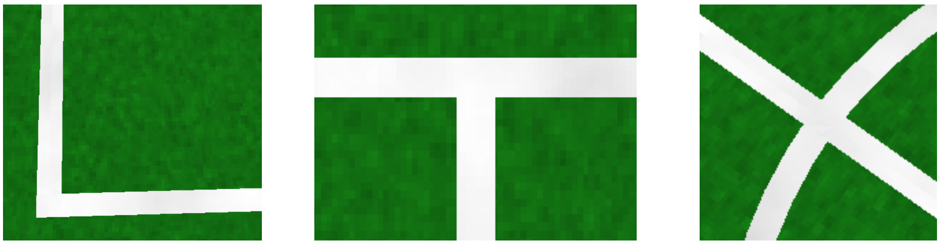
4.8 Line Detection

The line detection considers image regions that start with a rising edge and end with a falling edge. The gradients of both edges must point towards each other and should be parallel. To reduce lines detected in bright areas created by sunlight, the length of a region in robot coordinates has to be similar to the width of an actual line. For each valid region its middle point is added to a set of line points.

RANSAC is used to find up to five lines in the point cloud. If a line has a long gap, it is separated and considered as a new line. Each line has to contain a minimum number of points to be valid.

4.8.1 Line Intersections

To improve measurement input for localization, higher order field features are created from detected lines. A single line can correspond to many different positions in the field. By combining lines which intersect much less frequent features are created and used for more accurate self-localization. For these purposes the `LandmarkFilter` differentiates between three types of intersections as depicted in fig. 4.3.



(a) L-Intersection (b) T-Intersection (c) X-Intersection

Figure 4.3: Different types of line intersections.

Detected lines are merged into intersections by performing the following steps:

1. Find all pairs of lines (A, B) which are orthogonal to each other.
2. For each of these pairs find their point of intersection p_i. This point does not necessarily have to be positioned on one of the segments.
3. Calculate vectors between the point of intersection p_i and the endpoints p_{A1}, p_{A2}, p_{B1}, p_{B2} of the two lines. Name these vectors $v_{i,A1}$, $v_{i,A2}$, $v_{i,B1}$, $v_{i,B2}$.
4. Calculate the dot products $x_A = v_{i,A1} \cdot v_{i,A2}$ and $x_B = v_{i,B1} \cdot v_{i,B2}$.
5. Using the dot products x_A and x_B, the following cases can be distinguished:
   - $x_A > 0$ and $x_B > 0 \Rightarrow$ the pair is an L-intersection
   - either x_A or $x_B > 0 \Rightarrow$ the pair is a T-intersection
   - $x_A < 0$ and $x_B < 0 \Rightarrow$ the pair is an X-intersection

4.9 Penalty Spot Detection

In the first step the penalty spot detection uses the horizontal scanline segments only. There are fewer horizontal scanlines than vertical scanlines due to the fixed distance in the robot coordinate system. Therefore, they can be searched faster to get a rough idea of a potential penalty spot. Field colored segments and segments above the field border are excluded from the search. A segment must not be too small or too far away from the robot. The detection distance is limited to 3 m in order to reduce the false positive rate.

Afterwards, the size of a theoretical penalty spot at the segment's position is calculated. If the size does not match the expected size of the penalty spot at that position, the segment is discarded and the next horizontal segment is evaluated. Otherwise, all vertical segments intersecting the horizontal segment will be considered. They must not be longer than the horizontal segment in pixel coordinates. Penalty spot hypotheses on the ball position are excluded from further observations.

A point of intersection is calculated which is defined as the point in which the two segments overlap if they intersect each other in the segment's mid point. This point represents the penalty spot's center point. Subsequently, twelve equidistant points on an elliptical path are calculated around the center point. The major and minor axis of the ellipse can be calculated by back projections of the real penalty spot dimensions. Every point on the ellipse must be inside the image and have a darker luminance value than the penalty spot center. In addition, each point must have a higher chrominance compared to the center or at least be classified as field color.

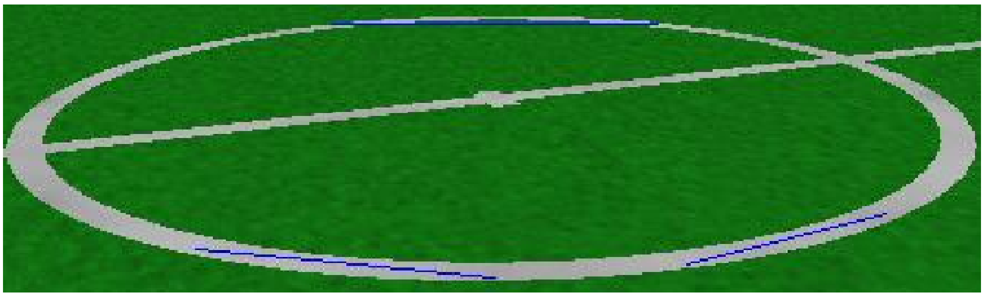
The hypothesis with the smallest euclidean distance between the theoretical center and the segments' intersection represents the penalty spot. Further details about the penalty spot detection and its performance can be found in [9].

4.9.1 Penalty Area Detection

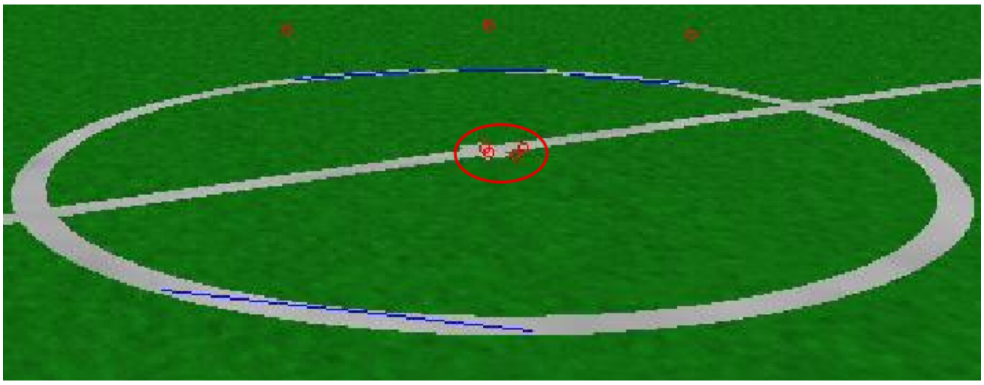
By combining the detected penalty spot with the line from the penalty box, a feature called penalty area is defined. This area has the advantage that it can be used to improve on the orientation estimation for our robot localization. The penalty area is detected by searching for a line which corresponds to the known distance between the penalty spot and the penalty box.

4.10 Center Circle Detection

Since lines on the center circle are detected, as shown in fig. 4.4a, they are used to estimate the position of the center circle. Two points distanced by the center circle radius are placed orthogonally left and right of the line. After repetition of this step for all lines the largest cluster of points is found. Clusters which contain less than a minimum of points or spread too much around the center of the cluster are discarded. The cluster containing the most points is considered as the center circle. One cluster detected this way is shown in fig. 4.4b. In order to be able to also make use of the orientation of the center circle, a line that strikes through the previously found cluster is searched for.



(a) Line segments detected in the center circle.



(b) Added points and resulting cluster (circled in red).

Figure 4.4: Visualization of center circle detection.

4.11 Ball Detection

The ball detection consists of a candidate generator based on the filtered segments and a ball detection which uses multiple neural networks. The candidate generator produces a list of interesting quadratic patches that are then piped through the neural networks. Specially crafted datasets and tooling is available for training the neural networks.

4.11.1 Candidate Generation

The candidate generator creates a perspective grid consisting of quadratic boxes onto the image to fit the patches to the size of a potential ball. Center points of filtered segments are again filtered by removing all points that have already been used in line detection. The remaining filtered segment points are fitted to the boxes of the perspective grid. Boxes that contain at least one point are accepted as candidate for the ball detection.

4.11.2 Neural Network Ball Detection

Candidates from the candidate generator already have a reasonable size which ensures that balls have roughly the same size on each patch. The patches are sampled into a 32×32 pixel grayscale image and used as an input for the first neural network. This neural network is a pre-classification network which is trained using a F_2-score metric. The result is a very sensitive neural network with higher recall, i.e. discarding candidates which are very likely not balls. If the pre-classifier accepts a candidate, the candidate is fed through the second classification network. This network is trained using a F_1-score and has therefore a higher emphasis on the precision than the pre-classifier. If the second neural network also accepts the candidate, it is considered a ball and the third neural network determines the position and radius on the patch. Since patches may overlap, the last step consists of merging ball circles that likely correspond to the same ball.

To achieve maximum candidate throughput, the neural network architecture is optimized using a genetic algorithm. The genetic algorithm favors very fast networks for the pre-classifier and very accurate networks for the classifier and positioner. More information about the genetic hyperparameter search can be found in [11].

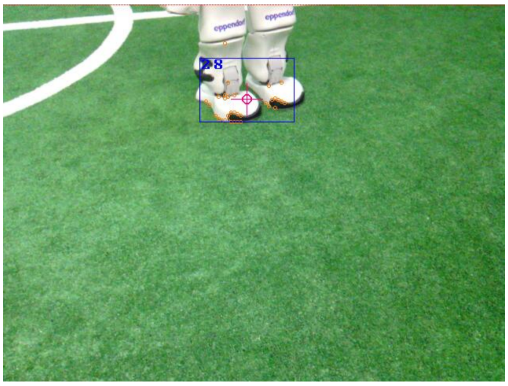
Balls on images of our recorded games have been labeled using specialized tools which can be found in `tools/machine-learning/ball_detection`. The `grid_cropper` is an automatic labeling tool which uses a similar candidate generation and neural network pipeline as mentioned above. Annotated balls can contain false positives and the `remover` is a web-based tool to remove these. The `adder` allows to add balls that have not been detected during automatic labeling. Up to this point, positions may not be precise enough. The `corrector` is a tool that can be used to correct the ball's positions and radii.

4.12 Robot Detection

The visual robot detection extends the detection of obstacles found by the Sonar Filter (see section 6.8). This allows the NAO to detect robots in its field of view and determine their location.

As robots are the only objects on the field containing a significant amount of horizontal edges, the robot detection is based on counting vertical segments. Consecutive vertical non-field-color segments below the field border are counted. A series of consecutive segments constitute a chain. The end of the last segment in such a chain is considered as a seed. Using camera projection, the size of a robot's feet is calculated at the seed's position. A sliding window of this size is used to find the robot candidate containing the most horizontal edges. The candidate is accepted if sufficient edges are present in the window. The center of the detected robot projected onto the ground is calculated and provided for further filtering methods (i.e. section 5.7). This information is depicted in fig. 4.5. When the robot's feet are not entirely visible in the current image, the position cannot be determined exactly and will therefore be omitted (see fig. 4.5b).

To be able to detect multiple robots in the same image, seeds and edges belonging to already detected robots are discarded. Afterwards, the explained method is applied to the remaining seeds until all seeds are evaluated. The order in which the seeds are evaluated is defined by their proximity to the NAO, starting with the closest. This avoids conflicts for overlapping robots in the image.



(a) Successfully detected robot

(b) Detected but cut off robot

Figure 4.5: Visualization of robot detection containing the sliding window (blue/red), the projected center (pink), consecutive horizontal edges (orange). The number inside the sliding window represents the number of edges in the window. The field border is shown as a dotted red line.

Chapter 5

Brain

The ”Brain” part of our code base is divided into two domains: Gaining knowledge and coordinating team behavior. The former is concerned with localization (section 5.2), team ball filtering (section 5.4) and whistle detection (section 5.11) among other things. The latter includes, but is not limited to, role assignment (section 5.1.1), behavior of individual roles (section 5.1.3 to section 5.1.8) and ball search behavior (section 5.5).

5.1 Team Behavior

To coordinate the team behavior, roles are assigned dynamically during the game based on the world model. The world model includes the ball position and the positions of other robots. Based on a robot's role and the roles of its teammates, an action is performed. In addition to the obvious roles (keeper, striker and defender) the roles include a support striker, a bishop, and a replacement keeper. During ball search, a loser and searchers may be assigned. Roughly, the roles have the following description:

1. Keeper: Robot guarding the goal.
2. Striker: The robot playing the ball towards the opponent's goal.
3. Left/Right Defender: Defensively positioned robot within own half.
4. Support Striker: Staying close to the striker in case the striker loses the ball or falls over.
5. Bishop: Trying to be a favorable pass target by occupying an offensive position in the opponent's half.
6. Replacement Keeper: Guarding the goal while the keeper is penalized or far away.
7. Loser: Former striker after losing the ball and entering ball search walking backwards and trying to find the ball again.
8. Searcher: Searching for the ball when there is no team ball.

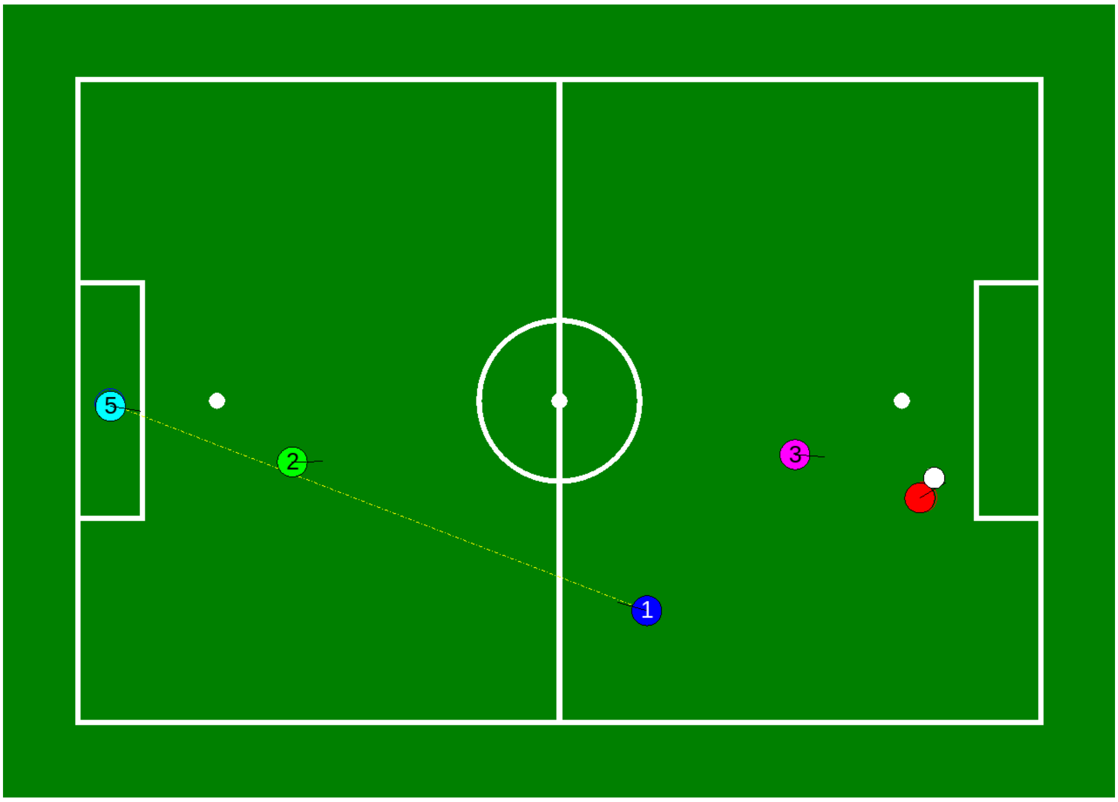


Figure 5.1: Keeper and replacement keeper exist simultaneously. The keeper is far away from the own goal, hence the replacement keeper guards it until the real keeper returns.

For each of these roles, a module exists that provides an appropriate action or position. This is necessary because the module that combines the behaviors to a single output does not have good means to preserve state.

5.1.1 Role Provider

The general procedure of the role assignment is that each robot provides roles for the whole team. Each robot believes the assignment of the teammate with the lowest number that is not penalized. This approach is similar to that of B-Human (cf. [16, chapter 6.2.1]).

The roles are assigned as follows. At first, if the ball position is known, the robot with the smallest estimated time to reach the ball is assigned to be the striker. Thus, there will always be a robot following the ball, even if no other robot is on the field and it will always take the minimum time to interact with the ball. Note that the player with player number one can become striker. If, however, no robot sees the ball with enough confidence, no striker will be assigned. Instead, the loser role will be assigned to the player that was previously striker and close enough to the last known ball position.

If such a player does not exist or if the time in which the ball has not been seen has surpassed a certain duration, the loser role will not be assigned. Second, the player with player number one becomes keeper unless it already is striker or loser, in which case no keeper is assigned. If this is the case or if a keeper exists but it is far away from the own goal a replacement keeper is selected based on the distance of the remaining robots to the own goal. Note that it is possible for a keeper and a replacement keeper to exist simultaneously (cf. fig. 5.1). After striker/loser, keeper, and/or replacement keeper are considered, there are zero to four robots left. When the ball position is known, the remaining robots are assigned, in order, to the following roles: defender, support striker/bishop, defender, support striker/bishop and whichever has not been assigned, yet. During ball search, if there is no loser, the first remaining robot will become searcher. After that, a defender is assigned and any remaining robot will become searcher. If the ball has not been seen for a longer period of time, there is no defender and all remaining robots will be assigned the searcher role. For the defender a left/right decision is made depending on the position of the ball and the robots involved. The `PlayingRoleProvider` can be configured to enable/disable the replacement keeper and the bishop.

5.1.2 Set Position Provider

The `SetPositionProvider` computes the position where a robot should be in the SET state of a game, i. e., where it should go during the READY state. The set of positions that the robots may take is preconfigured but the assignment is calculated dynamically to minimize the overall distance that robots have to walk. These starting positions should match the roles the robots assign themselves when the game state changes to PLAYING.

5.1.3 Keeper

The keeper's responsibility is to avoid receiving goals. To accomplish this the keeper always blocks the line of sight between the ball and the center of the own goal. If the ball is moving towards the own goal with a certain velocity a kneel down motion, termed squatting, can be performed. If the striker is playing the ball close to the own goal the keeper moves away to clear the way for the striker.

5.1.4 Striker

The main task of the striker is to score goals. All other roles assist the striker or defend the own goal. The striker is the only robot that is supposed to play the ball. The position of the ball on the field has implications on the way the striker plays the ball.

If the ball is near the opponent's goal, the striker either dribbles or kicks the ball towards the goal center. However, if the ball is very close to the goal and the current dribble direction would score a goal, the striker will dribble from its current position.

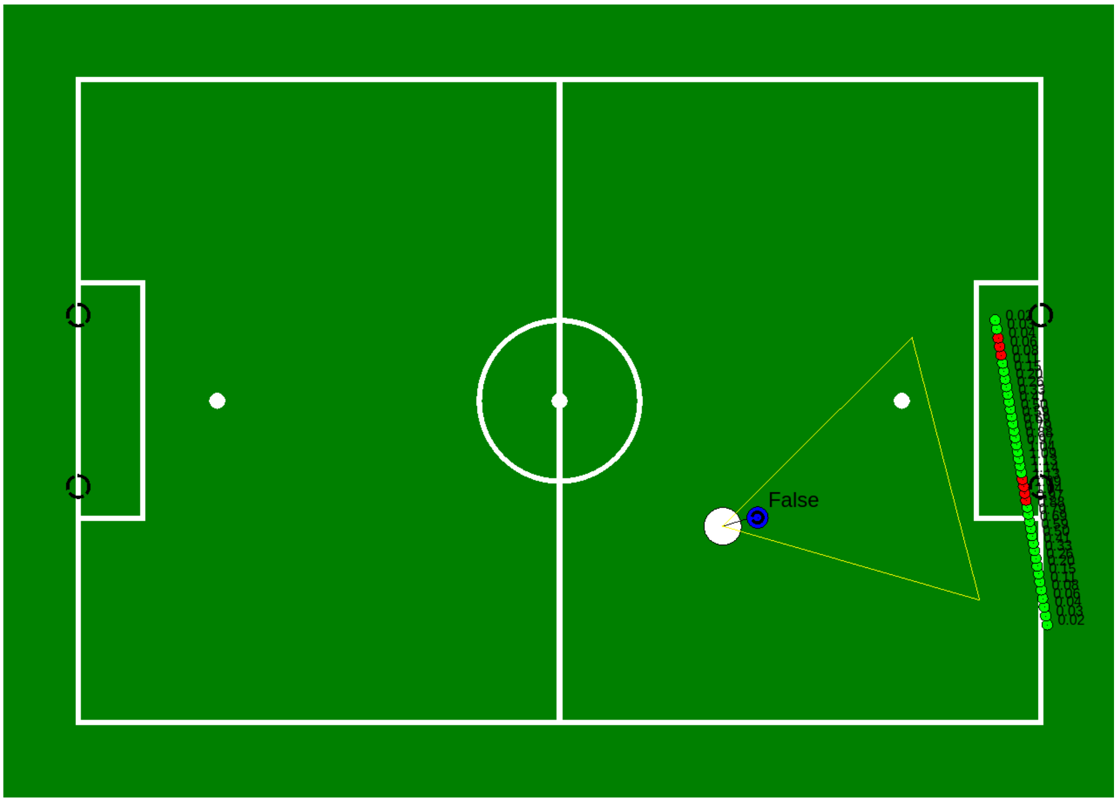


Figure 5.2: The lawn-sprinkler-like sampling of the kick deviation sector. The blue circle is the ball; the white circle is the striker. The striker aims at the goal center. Green dots indicate lawn sprinkler directions that do not intersect with obstacles; red dots are obstructed by obstacles.

If the ball is near the own goal, the striker should clear the ball as fast as possible. The direction to clear the ball is the sum of directions at several key points weighted by their distance to the current position. Depending on whether obstacles block this direction, the striker either decides to kick or dribble. To determine whether the direction is blocked a sector around the desired direction is sampled in a lawn-sprinkler-like way. This is illustrated in fig. 5.2. The sector is discretized as 41 discrete directions and are assigned weights. The weights describe the deviation and follow a Gaussian normal distribution. If the sum over all weighted directions that are not obstructed exceeds a threshold, the striker kicks, otherwise dribbling is assumed to be the better action.

If the ball is not close to any goal the situation is not critical. The best action is to dribble the ball towards the opponent's half to reach a position from which the striker can score.

5.1.5 Defender

The defender assists the keeper in blocking as much of the own goal as possible. The assignment of left and right defender is handled by the `PlayingRoleProvider`. The left/right defender is assigned so that it stays on the same side of the field as the ball, i. e., if the ball is in the left half a left defender is assigned.

The defending positions are determined by three different lines: passive, neutral and aggressive. Depending on the ball position the defending positions differ in their x-coordinate, where the x-axis points towards the opponent's goal. If the ball is close to the own goal the defender stays right behind the penalty spot. If the ball is far away, the defender is more aggressive to cover a larger area of the field in case the ball is lost. The y-coordinate is determined by the line between the own goal center and the ball. Ideally, this line of sight is blocked by the keeper. To cover the remaining area of the goal, the defender positions itself to the left or right of that line. If the ball is in a corner on our own side of the field, the positions are clipped because the line intersections can be outside the field.

5.1.6 Support Striker

The support striker is designed to assist the striker by taking over if the striker loses the ball in a duel or if it falls over. To accomplish this, the support striker always stays some distance behind the striker. The exact position is subject to a trade-off between covering as much as possible of the own goal by standing on the line of sight between ball and goal on one hand and being able to look at the ball on the other hand. The support striker, shown in purple, does not stand directly on the line of sight. Instead, its position is slightly shifted so that it is able to see the ball. This matters significantly because the team ball model works better if multiple robots see the ball (cf. section 5.4).

The penalty box and the surrounding area can become densely crowded. In order to avoid mutual obstruction of striker and its teammates, the support striker keeps a minimum distance to the goal.

5.1.7 Bishop

The bishop has two modes: an aggressive and a defensive one. In the aggressive mode the bishop lurks around close to the opponent's goal. It waits for the ball to arrive to eventually become striker and score a goal. The defensive mode is similar to the behavior of the support striker.

5.1.8 Replacement Keeper

The replacement keeper mimics the behavior of the keeper with one exception. The SPL rules specify that a robot that does not have player number one receives a penalty for touching the ball with its hands [15]. To avoid accidentally touching the ball, the replacement keeper must not use the squat motion. This is realized by a permission management that is similar to UNIX file permission. Actions are encoded by powers of two. Keeper and replacement keeper have a variable that stores their permission level as a sum of allowed actions. If the binary representation of the permission level does not include the power of two of an action, that action is prohibited.

5.1.9 Loser

The loser role can only be assigned in the first few seconds after the ball is lost. When a striker loses sight of the ball it was close to and there is no valid team ball either, the striker will become loser in the next cycle. A loser's task is to find the previously lost ball again as quickly as possible. Since the loser will only be assigned if this robot has been close to the ball before, there is a high possibility that the ball is still close by. The loser will therefore walk a few steps backwards while turning its head looking for the ball. If the ball is not found after a few seconds, it is assumed to be further away and the loser role will not be assigned again during this ball search. If the ball is found, ball search ends and the loser role is not assigned any more.

5.1.10 Searcher

The searcher is a role that can, like the loser, only be assigned during ball search. Except for those robots that obtain special roles (loser, keeper, replacement keeper, defender), all remaining robots become searcher when the ball is not seen. All available searchers will walk around the field looking for the ball. To prevent several robots from checking the same places, a ball search map is used to calculate the search positions for all searchers.

5.2 Localization

Our code base features a localization module based on an Unscented Kalman-Filter (UKF) [13].

While a full discussion of this method in all details goes beyond the scope of this paper, we will outline the high-level idea of this module and present the main results of our evaluation.

5.2.1 Inputs

As it is common in the SPL, we feed sparse field features like lines, penalty spots and the center circle as updates to our pose estimator. While this low-dimensional, feature-based approach is very data efficient, it also brings the challenge of ambiguous measurements (cf. fig. 5.3). All input to the localization module is provided by the `LandmarkFilter`, a module that pre-filters and abstracts the raw percepts produced by our vision pipeline. Additionally, an odometry estimate computed from the orientation estimation (see section 6.9) and forward kinematics are used to predict state evolution.

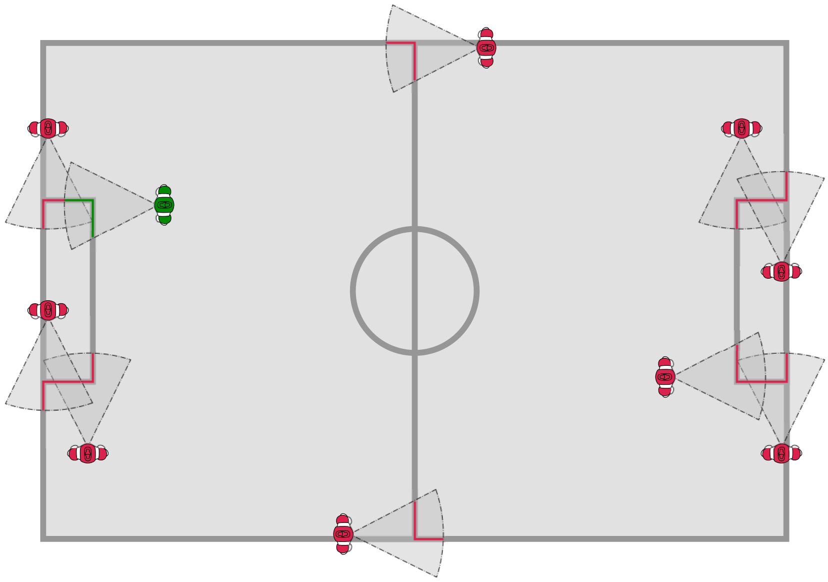


Figure 5.3: Ambiguity of measurements for feature based localization. True pose and perceptions (green) and other state that could create the same perceptions (magenta).

5.2.2 Algorithm

On an abstract level our algorithm solves the estimation task by tracking multiple hypotheses of possible robot locations. This multi-hypothesis approach allows us to approximate the multi-modal state distribution by a set of Gaussians. The state of each hypothesis is represented by its position and orientation (x, y, α) and can then be estimated using a separate UKF mode. Additionally, each hypothesis holds information about past measurement prediction errors.

Prediction At every cycle, we predict the state evolution of each hypothesis based on the odometry estimate. The new state estimate is computed from the last belief transformed by the estimated pose shift (see fig. 5.4). Since the prediction is non-linear, we use the unscented transform to compute the predicted state distributions.

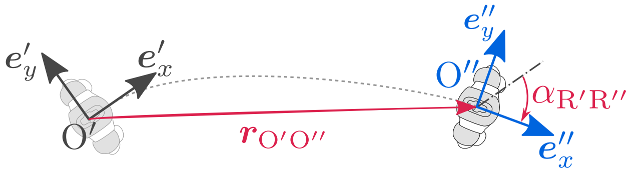


Figure 5.4: Predicting the state evolution using the odometry estimate.

Correction Measurements are used to correct the state estimate whenever the camera matrix is believed to be valid. The validity of the camera matrix is classified based on the estimated angular velocity of the camera frame. For large angular velocities of the camera frame, all measurements are rejected. Updates are either performed as linear updates as in the vanilla Kalman-Filter or, in case of non-linear observation models, by utilizing the Unscented transform.

In addition to the pose, we also update the weighted error of predicted observations with every correction step to yield a measure for the validity of every hypothesis.

Hypothesis Elimination and Selection Following the correction step, we perform one round of hypothesis elimination. In this step we eliminate hypotheses if they are significantly worse than the current best estimate. Here, the quality of each hypothesis is rated by its weighted error of predicted observations. Additionally, we merge hypotheses whose state means have converged to approximately the same state. Redundant hypotheses are deleted after the merged hypotheses are obtained by updating the neighboring hypotheses with the full state observation of the deleted ones.

After all invalid or redundant hypotheses have been deleted or merged, the hypothesis with the lowest observation error is published as the current state belief.

5.2.3 Performance

The evaluation of both localization approaches, particle filter and Unscented Kalman-Filter filter, showed that our approach clearly dominates in both estimation performance and runtime. Due to the fact that runtime constraints only allow to simulate a very limited amount of particles, this approach suffered from insufficient sampling density of the relevant state space. As a result, the particle-filter-based localization occasionally falsely eliminates relevant clusters and provides a less smooth state estimate due to sampling noise. The UKF-localization provides a more robust and accurate state estimate, while achieving a worst case runtime seven times faster than the particle-filter. Details about the performance evaluation can be found in [13].

The current estimation performance enables good localization throughout most games. This allows performing more complex team maneuvers and improves positional play.

5.3 Ball Filter

Key to an accurate estimation of the ball's position and velocity is a good model of the ball dynamics. Although the used friction model only provides a very simplified picture of the highly non-linear ball dynamics, it has proven to be accurate enough for most tasks. The resulting improvement in prediction performance in many cases allows the robot to re-localize the ball. Furthermore, it allows to obtain a straight forward estimate of the ball destination.

5.4 Team Ball

The team ball is a combination of the local ball estimate and the communicated ball estimates of other robots. It is designed in a way that behavior modules can safely rely on the team ball and do not need to decide between the own estimate and the team's estimate for themselves. Each player maintains its own buffer of balls. The estimates of the other robots and the local estimate are added to a buffer. However, a number of conditions must apply before adding a ball. More precisely, the ball filter must be confident about the ball state and the time of last perception must not be too long ago. Balls that have not been seen for a time longer than a certain threshold, will be removed from the buffer. Spacial clustering is applied to the ball data in the buffer to obtain candidates for a final ball position. The largest cluster is generally favored, but when there are several clusters of the same size, the cluster which contains the ball that has been seen for the longest time is selected. Afterwards, the best ball of the best cluster is selected. The best ball in a cluster is always the local estimate. In case the largest clusters do not contain the local estimate, the most confident ball is selected. The confidence of a ball is assumed anti-proportional to its perception distance.

Currently, there is no averaging performed to extract the ball state from the best cluster, instead only one estimate is selected. This selection proved to be quite robust against false positive ball detections of single robots. In particular, the ball-playing robot, the Striker, has a stable team ball.

The team ball model also integrates prior knowledge in certain game states if no ball is seen: If no ball is found in the SET state, the ball position is set to the center of the field or the penalty spot in a penalty shootout. Other modules can access the information whether the team ball originates from the robot itself, a teammate, is invalid, or is known by the game's state.

5.5 Ball Search

Whenever the ball is lost during a game, a strategic team behavior to search for the ball is needed. Our framework features a modified role assignment as well as two ball search specific roles to take care of this task.

Ball Search consists of two different states, namely Short Term Ball Search and Long Term Ball Search. While Short Term Ball Search starts as soon as no robot sees the ball with enough confidence, it automatically transists into Long Term Ball Search as soon as a certain configurable amount of time has passed and the ball has not been found yet. The key difference between Short and Long Term Ball Search is the roles that are being assigned while they are active. The main goal during Short Term Ball Search is, apart from finding the ball again, to prevent the opponent team from scoring a goal while our team does not see the ball. The main goal of the Long Term Ball Search is to find the ball as quickly as possible to prevent a Global Game Stuck. In order to achieve these two goals, during Short Term Ball search the following roles can be assigned: loser, keeper, replacement keeper, defender, searcher. During Long Term Ball Search, only the following roles may be assigned: keeper, replacement keeper, searcher. By eliminating the loser and defender roles, the Long Term Ball Search can assign the searcher role to more players and therefore raise the probability of finding the ball. While finding the ball is still important in Short Term Ball Search, it is likely that the opponent team is able to see the ball. In this case, is is crucial to keep the defense up by keeping a defender as well as a keeper and/or replacement keeper to prevent the opponent team from scoring a goal. If the ball has not been found for a long time, it becomes more likely that the opponent team does not know the ball position either. Then it becomes more important to prevent a Global Game Stuck and Long Term Ball Search starts.

The central playing role of the Ball Search is the searcher which is assigned to search for the ball at positions calculated by the `BallSearchMap`. It takes information like robot positions, ball position and field of view, to calculate the most probable ball position represented by a heat map.

This map is used in a module called `SearcherPositionProvider`. It calculates position suggestions for all searchers on the field. After it received these suggestions from every active player it determines which suggestion to trust most and calculates the desired walk target.

It should be noted that the implemented searcher behavior aims to search for a ball as fast as possible, even if the ball detection is not optimal. The behavior is not considered ideal when it comes to Short Term Ball Search.

5.5.1 BallSearchMap

The BallSeachMap is responsible for keeping track of all seen balls of every player that is neither penalized nor unsure about its own localization. Therefore the SearcherPositionProvider gets the following information from all players:

- robot position information
- field of view
- ball position and age
- penalty information

Additionally, the module depends on the `GameControllerState` as it implicitly contains information about the ball state.

The map manager then produces the `BallSearchMap` from this input. This map is a discrete probability distribution that is implemented as a matrix of probability cells (ProbCell). Each cell stores its current probability (weight) to contain the ball and an *age* which denotes how much time has passed since the cell was last seen by any robot. The map gets updated with every single cycle on each robot. Each update works in the following way:

- If any team member sees the ball at a given position, the corresponding ProbCell's weight will be increased.
- Every ProbCell's weight will be decreased, albeit at a slower rate, if it is inside the field of view of any team member[1].
- Each cell that was not modified by the preceding operations will get an increased age. The age of all other cells gets reset.
- Afterwards, the map gets convolved with the following convolution kernel:

$$\frac{1}{c+8}\begin{bmatrix}1 & 1 & 1\\1 & c & 1\\1 & 1 & 1\end{bmatrix} \tag{5.1}$$

  in which c is large. It is ensured, that the newly calculated value for a cell may never be lower than its previous value after the convolution is done. This prevents downvoting a cell that was never inspected by any robot.
- If the game is in READY state, the map is being reset constantly so that the center circles cells have the highest probability.
- Whenever the ball leaves the field, the kick-in position is increased in probability.
- When there is an ongoing goal free kick the two possible ball positions are heavily increased in probability.
- Lastly, the map is normalized to keep an overall probability sum of 1, since the ball is assumed to be on the field at all times.

The convolution will cause a ProbCell with a high value to slowly spread its probability to its neighbor cells. If the ball is seen continuously by any robot, the containing cell will be increased with every cycle, which negates the convolution mostly, since the map is also normalized every cycle. Updating the map in this way ensures that it keeps track of all balls seen by any team member on the field. While a ball that is seen by two robots is represented with a higher probability. The resulting map is shown in fig. 5.5 (visualized using MATE, see section 7.2).

[1]The field of view for the team members is calculated using their position and head yaw.

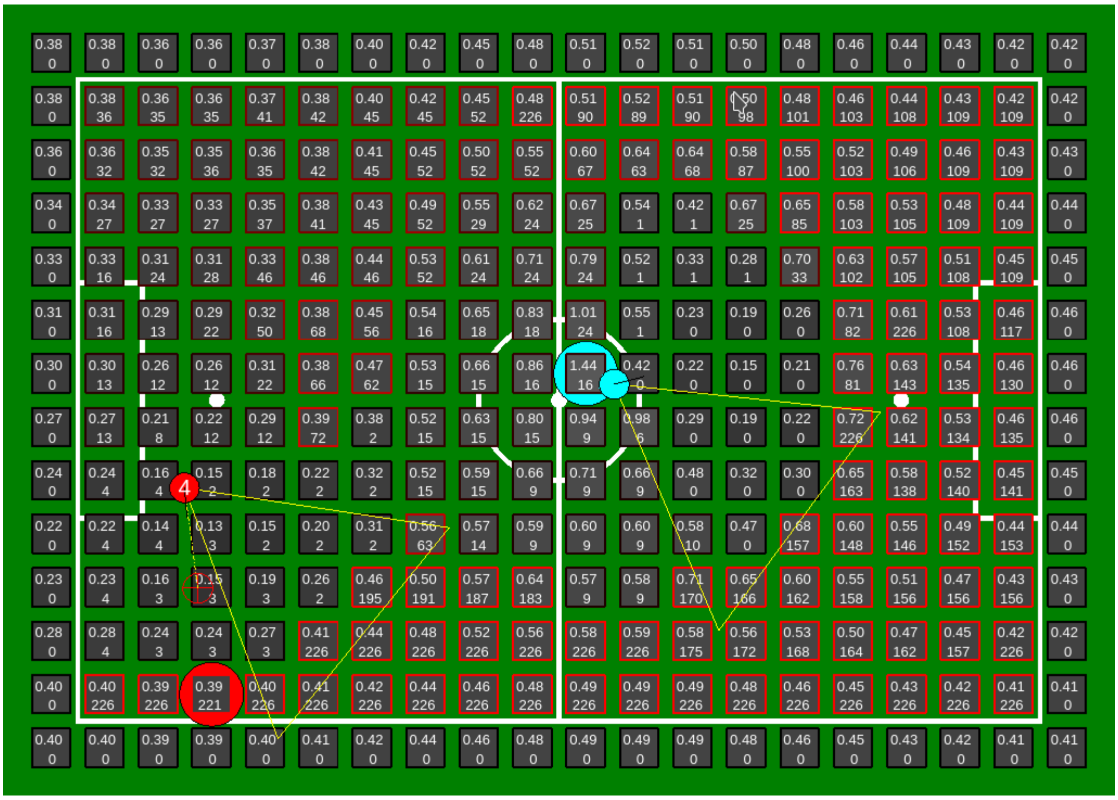


Figure 5.5: This is a visualization of the ball search map. The probability upper value inside the cells) is represented by the brightness of the cells while the age (second value inside the cells) is represented by the red border. Also the players with their field of view and current target cell are visualized.

5.5.2 SearcherPositionProvider

The `SearcherPositionProvider` is responsible for calculating search positions for every searcher as well as agreeing on the `most wise player` [2]. This module mostly depends on the `BallSearchMap` and the player positions. The output of this module consists of the following information:

- An array of suggested search positions for all searchers.
- An array that flags all previously mentioned search position suggestions as valid[3] or invalid.
- The most wise player number.
- The search position (the exact position to look at).
- The search pose (the pose to walk to for looking at the `search position`).

Gathering the listed information is done in the following steps:

Calculating the most wise player: In this step, the player with the *best* map is selected as the most wise player. *Best* means the map that was updated for the longest period of time without any interruption (like a penalty for that specific robot). As the wireless network is not reliable on competitions at all times the robot may fall back to it's own knowledge when needed.

Generating and assigning search areas: The field is divided into as many areas as there are explorers on the field. The areas are defined by an array of points. Using those points as seeds for a Voronoi diagram with euclidean distance then defines the search areas. These areas are then assigned to the robot that is closest to the cells center. Re-assigning only happens whenever a robot is dropped or added to the searchers.

Assigning search positions: After each searcher is assigned to a search area it only needs a position to explore. This is done by sending the robot to the *best* ProbCell inside the search area. The *best* cell is the one with the lowest cost to explore (see eq. (5.2)).

$$costToExplore(cell) = \frac{timeToReach(cell) + 2}{value(cell)} \tag{5.2}$$

$$\begin{aligned} value(cell) = & cell_{probability} * probabilityWeight \\ & + \min(maxAgeValueContribution, cell_{age}) \end{aligned} \tag{5.3}$$

[2] A player that holds a map that was continuously updated for the longest period of time, the decisions of this particular player are then accepted by every other player

[3] A suggestion is marked as invalid whenever a player does not have the searcher role.

Generating the own search pose: This last step sets the own search position to the value that is proposed by the most wise player (may be himself). Afterwards a suitable pose to look at the chosen position is calculated. The resulting pose can then be used by the behavior whenever needed.

5.5.3 Remarks

The introduction of the BallSearchMap allows us to almost always find the ball again after it has been lost. It also dramatically reduces the chance to cause a `global game stuck`, since the robots will always explore all areas of the field when the ball is lost. However, there are some problems that are not addressed by the current implementation:

- A robot's field of view might be blocked by an obstacle (e. g. another robot).
- The map assumes that a ball can not leave the field.
- Defense is down whenever Long Term Ball Search has started.

5.6 Head Motion Behavior

The Head Motion Behavior is controlled through `ActiveVision`. The idea behind this is to decouple the head motion from the rest of the behavior. It provides a set of different modes that can be activated in the behavior. Based on the chosen mode, a decision on how the head behavior is made. The following modes are available:

LookAround The robot moves his head from left to right.

LookAroundBall The robot will move his head from left to right but will, if possible, always keep the ball in the field of view.

BallTracker The robot will follow the ball as close as possible and keep it in the middle of the field of view.

Localization The robot looks in the direction that maximizes the number of points of interest in the field of view. The points of interest are pre-selected and significant field marks like the center circle or T-sections.

SeachForBall The robot looks around, searching for the ball. It locks onto a detected ball. This mode should only be used if no team ball is available.

LookForward The robot looks forward.

5.7 Robots Filter

The `RobotsFilter` applies simple modeling to robot detections (cf. section 4.12). For each filtered robot a Kalman-Filter is instantiated. A Kalman-Filter has two steps: prediction and update with measurement. Robot motion is predicted using a linear velocity model. While this neglects acceleration and rotation, predictions are sufficiently accurate to be associated with new measurements if the detection rate is high enough. After the prediction step, filtered robots are updated with associated robot detections. A robot detection is associated with a filtered robot if the distance between them is below a threshold. Finally, new objects are created from unassociated detections and filtered robots that have not been updated for some time are deleted.

5.8 Team Obstacle Filter

The `TeamObstacleFilter` performs the task of fusing obstacle detections from all team member's local obstacle models to obtain a combined obstacle model including all available knowledge. In our model, obstacles can have different types. On an abstract level we distinguish:

Robots Robot obstacles are known positions of friendly and hostile robots. The team affiliation and information about whether this robot is fallen are encoded in the type.

Map Obstacles Map obstacle are obstacles with a fixed global position throughout the game and are known from the map. Currently, goal posts are the only obstacles we consider of this type.

Rule Obstacles Rule obstacles are areas that have to be avoided according to the rules, e. g., free kick areas.

Ball The position of the current ball estimate as obtained by the `TeamBallFilter`.

Unknown Any other type of obstacle whose type could not be classified. This type of obstacle is generated e. g. in the event of sonar detections.

These types are used to determine mergeability of neighboring obstacles. In order to obtain a combined team model, obstacles from each player's local model are added to the map. While adding obstacles, we check for mergeability with obstacles already present in the map. Two obstacles are considered mergeable if the type is consistent and their positions are located in a small neighborhood. In the event of an obstacle merge, the more informative type is preserved. For example, merging an obstacle of type *Unknown Robot* with an obstacle of type *Hostile Robot* yields a merged obstacle of type *Hostile Robot*. Currently, the position of the merged obstacle is simply computed as the mean of the involved positions.

5.9 Motion Planning

Motion planning is responsible for determining the trajectory of future robot poses and the required translations and rotations in order to execute more abstract requests provided by the behavior modules. In doing so, it aims to create a desirable reference trajectory that moves the robot toward a specified destination while avoiding obstacles. The `MotionPlanner` supports multiple modes for walking on the one hand, as well as allowing to walk at a specific speed and into a specific direction on the other.

Our motion planning uses a straightforward vector-based approach. It is especially important when moving around the ball, where the robot will carefully try to avoid ball collisions while circumventing it. Furthermore it creates an aggressive dribbling behavior, which essentially tries to walk towards the ball in order to hit it as much and as fast as possible while maintaining correct alignment. To navigate around obstacles, an A*-based `PathPlanner` is used.

Specific components of the motion planning are explained in more detail in the following sections.

5.9.1 Translation

Determining the robot translation works by first creating a target translation vector that either points towards a pre-specified direction or to a desired position, depending on the requested walking mode. It then checks all known obstacles for any potential collision. All obstacles that lie within a threshold distance of the robot create additional displacement vectors that point away from the obstacle. A weighted superposition of the target translation vector with all the obstacle displacement vectors is then used to determine a final translation vector as an output of the module. This translation gets recalculated and reapplied every cycle, which results in the robot moving along a trajectory.

5.9.2 Rotation

Depending on the requested walking mode, there are two ways in which the robot rotation is handled. One option for the robot is to walk along a trajectory while maintaining a globally fixed orientation. Alternatively it will try to directly face the destination until approaching it, then gradually rotating to the final orientation. Gradual adaptation to the final orientation begins at a threshold distance and is done with a linear interpolation based on the remaining distance to target.

5.9.3 Walking Modes

There are several walking modes which can be requested by behavior modules. They differ in the way obstacles are handled, as well as allowing different formats for the motion request specification. The walking modes are implemented as follows:

PATH is the general mode used most of the time. The robot walks to a specified target while facing it. Obstacle avoidance is enabled in this mode.

PATH_WITH_ORIENTATION does the same as *PATH*, but in this mode the robot will directly adopt to a specified orientation.

DIRECT is the mode that ignores all obstacles and makes the robot walk directly to the destination, again facing the destination until near.

DIRECT_WITH_ORIENTATION is the same as *DIRECT*, but as in *PATH_WITH_ORIENTATION*, the robot's orientation while walking must be specified and will be adopted immediately.

WALK_BEHIND_BALL generally behaves like the *PATH* mode, but causes the robot to obtain a waypoint position close to the ball that may be reached safely, before approaching the secondary destination pose attached to the ball. The waypoint position is constructed as shown in fig. 5.6.

DRIBBLE is the most important walking mode for an attacking robot. It generally behaves like the *WALK_BEHIND_BALL* mode. Once the ball waypoint was reached, all obstacles are ignored and the robot directly walks at the ball as long as it is still facing the enemy goal.

VELOCITY is the mode in which a requested velocity vector directly specifies the desired translational and rotational velocity for the robot. Since the requested vector is not modified, all obstacles will be ignored. This mode is unused and only for debug purposes. The StepPlanner is currently unable to process this mode.

5.9.4 Path Planning

Walking towards the desired position directly is not always the optimal course of action. Often there are other robots, balls or rule-derived exclusion zones on the field, blocking the direct path. For this we use an A*-based path planning algorithm over tangent points between the current position of the robot, the desired target position and circular obstacles.

The implementation is based on a research project [22]. A notable difference is that blocked areas or obstacle shapes other than circles are currently not supported.

5.10 Penalty Shootout

The penalty striker randomly selects a corner of the goal it will be shooting at. In addition, the enhanced motion planner (see section 5.9.3) is used to approach the ball slowly and safely, minimizing the risk of the robot running into the ball by accident. This walking mode also ensures that the robot is positioned accurately to kick the ball to the designated target.

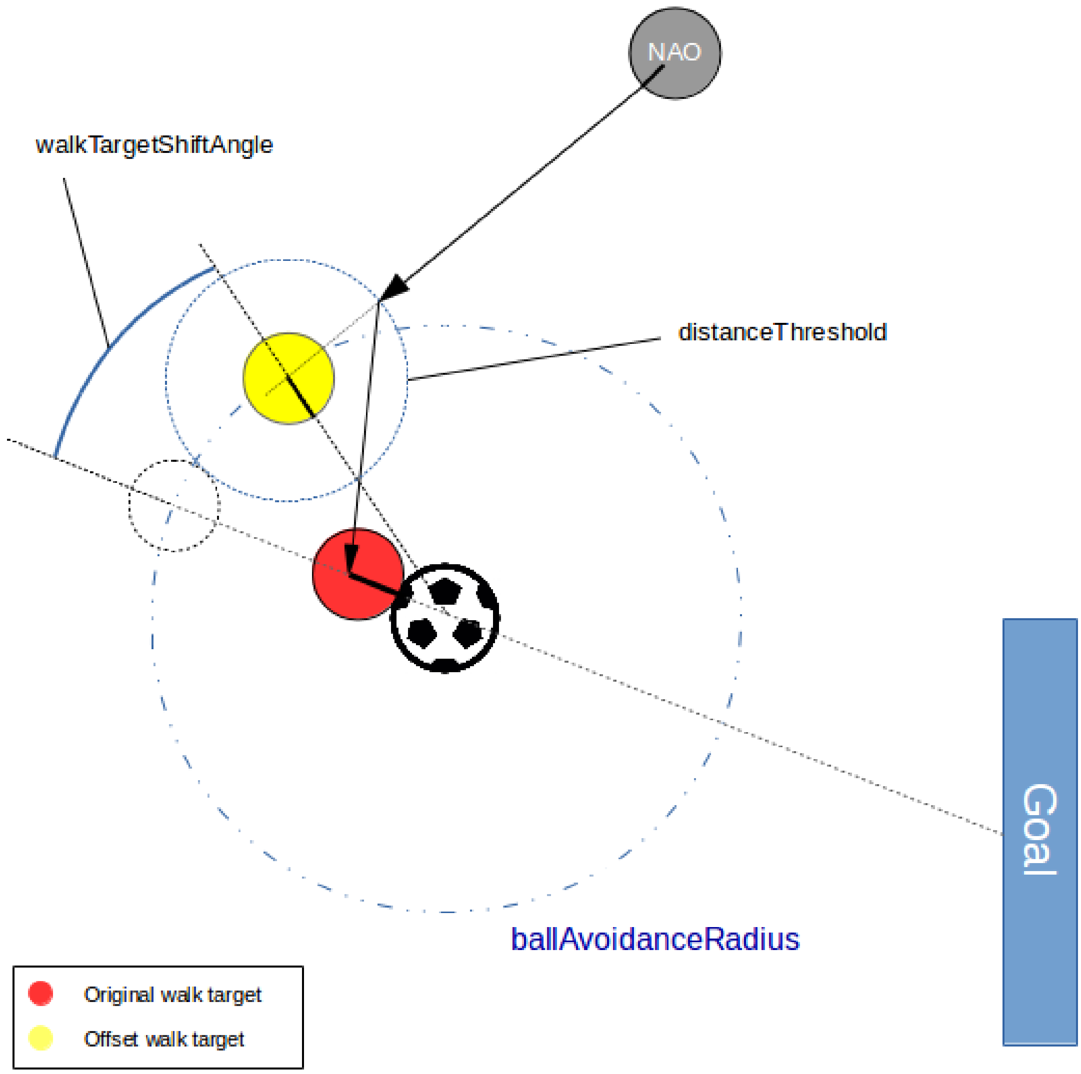


Figure 5.6: The `WALK_BEHIND_BALL` and `DRIBBLE` walking modes create a waypoint near the ball first (shown in yellow) by pulling away the original walking destination (shown in red) to the opposite direction of where the enemy goal is, and then rotating it towards the robot's current position. The robot's final trajectory is indicated by the black arrows.

During a penalty shootout, the keeper jumps either right or left or sits down based on the predicted ball destination. In bad lighting conditions the keeper has difficulties tracking the ball and thus can not predict the ball movement correctly, which leads to him not reacting at all. To circumvent this problem some parameter changes in the ball filter section 5.3 are necessary. Since these motions to catch the ball are rather destructive for the robot, they are not used in normal games.

5.11 Whistle Detection

The whistle detection features a dynamic detection of the whistle band in the frequency spectrum (cf. [5]). A Hann window is used to reduce spectral leakage.

In a predefined frequency range, in which the whistle is expected to be, the complex spectrum is divided into bands of fixed size. For each band the mean of the absolute values is compared against a threshold to narrow down the band that potentially contains a whistle. If the mean of the whistle band exceeds a threshold, a whistle is believed to be found in that spectrum. If a whistle is found in four consecutive cycles, it is considered to be detected and the game state changes to `Playing`.

The whistle detection proved to work well in the noisy environment that is the RoboCup, with the only flaw being the detection of whistles that are blown on other fields. Notably, a whistle is detected in the intro of the song *Engel* by Rammstein.

5.12 Whistle Localization

Most teams detect the whistle blown by the referee correctly. However, in the real RoboCup environment, whistles of neighboring fields are detected, which leads to unwanted behavior. To address this problem, RoboCup 2019's technical challenge featured the localization of whistle sounds. In the context of this challenge, a whistle sound source localization was implemented by computing the relative direction of a whistle source by each single robot and then combining the solutions to an absolute position by Bayesian updating.

Within the scope of a master thesis [8], the structure of the whistle localization was defined as follows: sound data of all four microphones are buffered until the whistle detection in section 5.11 finds a whistle. For obtaining the relative direction of the sound, a smaller number of samples at the beginning of the sound are chosen. In order to get the direction of the sound, the time difference of arrival (TDOA) method was selected, which is a popular approach for acoustic signal source localization. The fundamental concept of TDOA algorithms is to obtain direction information about acoustic signals by observing the time delay between separate microphones. Between multiple methods, the generalized cross-correlation with phase transform (GCC-PHAT) algorithm came out as most stable and accurate. By this, each neighboring microphone pair yields two possible direction angles which are filtered to one relative sound direction. This information is shared with the other robots and together with the pose of the robots, an absolute whistle position follows by considering the intersections of the robots' individual sound direction rays.

5.13 Foot Collision Detection

If we fail to detect an upcoming collision with sonar sensors (cf. section 6.8) we use the foot bumpers as a fallback. This is mostly used to detect NAO robots lying flat on the ground. The raw foot bumper values are checked for alternating sequences of the left and right foot bumper. If a left-right-left or right-left-right sequence occurs in a given timeframe, a collision is detected. An obstacle is created in front of the robot. The foot collision detector module respects the `HardwareDamageProvider` (cf. section 6.4), if at least one foot bumper is broken, the module is not executed.

5.14 Free Kick Situations

In 2018 free kicks have been introduced in the RoboCup SPL [14] for fouls and moving the ball behind the enemy's goal line. The rules of 2019 replaced throw-ins with kick-ins and corner kicks, extending the number of possible free kicks [15]. Some of our modules are actively reacting to free kick situations to comply with rules and use free kicks to our advantage. The modules and their reactions are listed below.

RoleProvider Favors a bishop over a support striker in certain circumstances to provide a pass target or goalhanger.

ObstacleFilter Adds an obstacle around the ball whenever the enemy performs a free kick, so that no robot may enter the forbidden area.

BallSearchMapManager Integrates implicit information about where a ball may be when a goal free kick is called.

DefendingPositionProvider Moves the defenders away when the free kick area is close to their defending positions.

ReplacementKeeperActionProvider Moves the replacement keeper away from its position if the ball too close, as he is not allowed to stay in the goal in this situation.

SupportingPositionProvider Moves the support striker between the ball and the own goal in case the enemy performs a free kick.

SetPlayStrikerActionProvider Replaces the regular StrikerActionProvider. A striker will try to score a goal during a free kick if distance and angle allow it. Otherwise it passes to an ally if there is one in a favorable position or starts dribbling towards the opponent's goal.

All of these modules also perform a validity check on the free kick situation. Whenever a goal free kick is called for the enemy team but a ball is detected in the own half of the field we ignore the game controller state as the referee clearly made a mistake. This also applies vice versa: A free kick that is called for us with a ball that was detected in the enemy half of the field is treated as a referee mistake, thus we are not approaching the ball.

5.15 Detecting Basic Referee Mistakes

As humans are not perfect, referees make mistakes. Mistakes made by the `GameController controller (GCc)`[4] can have a huge impact on the robot's performance, which is why we introduced the `RefereeMistakeIntegration`. As a sub-module of the `GameControllerAugmenter` it has the power to override the `GameControllerState` before any other module receives it.

Free Kicks The `kicking team` flag in the game controller message is set according to the actions of the GCc. During `corner free kicks` and `goal free kicks`, an error can be easily detected by checking the ball position for plausibility. We expect the ball to be in the opponents half during an enemy goal free kick and vice versa. The same logic can be applied to corner free kick situations.

Penalties During a game, the assistant referees have to carefully listen to the GCc for 10 second warning calls, signaling that a robot must be placed according to the rules for a successful return to the field. In busy situations, a robot might be unpenalized while it is still moved by an assistant referee. Our current approach checks for ground contact after a robot got unpenalized and extends a penalty, as long as the robot doesn't have safe contact with the ground.

In future versions we plan to watch for unexpected IMU readings to prevent delocalization whenever a robot is turned around after it is unpenalized.

5.16 Rainbow Eyes

To be able to tell if a robot is currently executing our code or not, we introduced the `rainbow eye` mode. In this mode, the LEDs in the eyes are all set to display different colors, forming a circular rainbow. Every n motion cycles the colors are moved one LED further resulting in a rotation.

This mode is always used in the INITIAL game state. This way the person responsible for deployment (see section 7.1) may quickly determine whether a robot was successfully deployed, crashed or shut down.

[4]Often mistakenly referred to as the `GameController operator`.

Chapter 6

Motion

This chapter describes how motions are executed in our framework. With one exception, all motions are the result of a `MotionOutput` that is derived from an `ActionCommand` from the brain. The `MotionOutput` is used in the `MotionDispatcher` to determine which motion ought to be active. The `JointCommandSender` interpolates angles and stiffnesses to execute the transitions and yields the angles and stiffnesses that are sent as commands to the joints via LoLA.

Section 6.1 briefly explains the `MotionDispatcher`. In section 6.2 the `JointCommandSender` is described. The following section 6.3 depicts the `JointCalibrationProvider`. Section 6.4 outlines the `HardwareDamageProvider`. The remainder of this chapter details different motions such as walking (section 6.6), kicks (section 6.7) and fall management (section 6.10).

6.1 Motion Dispatcher

The `MotionDispatcher` keeps track of the last motion that was active and handles the transition between motions. Each motion type has an activation value between 0 and 1. To transition from one motion into another, the activation value of the currently active motion is decreased from 1 to 0. Simultaneously, the activation value of the motion to be activated is increased from 0 to 1.

The `FallManager` is handled differently. Its activation is not controlled by the brain. Instead, it is triggered if the robot is detected to be falling. This is similar to reflexes in vertebrates which bypass the brain. Details about the fall manager motion can be found in section 6.10.

6.2 Joint Command Sender

The `JointCommandSender` uses the activations computed in the `MotionDispatcher` to interpolate the outputs of all motion modules. The outputs consist of joint angles and stiffnesses. For each joint the weighted sum of all motion module outputs is computed, where the weight is the respective motion activation. Joint calibration offsets are applied. In addition, the stiffness for each joint that is configured to be damaged is set to zero (see section 6.4).

6.3 Joint Calibration Provider

Joint offsets of each robot can be taken into account by the `JointCalibrationProvider`. This module produces the `JointCalibrationData` data type containing a set of calibration offsets for all joints. These offsets are subtracted from the measured joint angles in the `SensorDataProvider` and added to the final angle calculation in the `JointCommandSender`. The offset values are expected to be known.

6.4 Hardware Damage Provider

We can set an extensive hardware status list for each robot according to the state of each specified hardware component. It lists all joints and sensors such as sonar, foot bumpers and cameras and specifies whether they are functional or not. The `HardwareDamageProvider` processes this information and makes it available to the rest of the framework. Other modules can declare a dependency on the hardware status and react accordingly. For example, we may reduce the stiffness of broken joints to 0, so that they can then no longer be controlled by any other module (see section 6.2). Additionally, the hardware status information can be used to reduce the voting weight of a robot with broken microphones when trying to detect the start of a game.

6.5 Motion File Player

The simplest way to execute a motion is to play a motion file. These files consist of one header and several key-frames. While the header specifies the involved joints as well as the total duration of the motion, the key-frames consist of joint angles and stiffnesses with a corresponding relative duration. Motion files can be played with the `MotionFilePlayer`. It loads a motion file and interpolates between the specified frames while it is being played. Some basic motions such as standing up and keeper motions are realized with motion files.

6.6 Walking Engine

In the season of 2018 we replaced our walking module with the walking engine of UNSW [6] in the version ported by B-Human in 2017 [16]. The `Walk2014Generator` of UNSW was ported to our framework as it provides a robust and fast gait, yet has comparably low code complexity and thus can be easily augmented with additional features.

6.6.1 Modifications

Several improvements to the UNSW gait generator were made, two of which are explained in more detail here:

Step Sizes instead of velocities The UNSW gait generator is based on velocities as inputs to compute step sizes to execute. This was modified to an interface only supporting step sizes as walk requests. A new input for a step is only considered when starting a new step internally. That is, the support foot has changed successfully. The walk generator internally handles the transformations and computes the respective foot positions from that request.

Tackling Tackling situations require a stable gait that keeps the robot in balance while interacting with the ball. Thus, such situations pose one of the greatest challenges to a humanoid gait generator in RoboCup SPL.

In the event of a tackling situation we adjust the gait, leaning the robot's upper body forward and at the same time pulling the arms back closely to the robot's waist. By these means we shift the center of mass to the center of the feet, making the robot less sensitive to disturbing forces. At the same time, taking the arms close to the body reduces the robot's projected footprint and therefore lowers the likelihood of the arms colliding with other obstacles.

6.7 Kick Motion

The kick is one of two motion types that are not generated from motion files, the other being walking. Similar to motion files, the kick is generated from interpolation of joint angles. However, the joint angles are computed from desired positions of kicking foot and center of mass, relative to the support foot, at certain points in time during the kick using inverse kinematics. Parameterizing positions in cartesian coordinates, instead of playing motion files, has the advantage of being able to easily tune the kick motions. Among other things, the desired positions are parameterized to enable extensibility. In the current implementation only one kick type exists: a forward kick. As all kicks are based on an interpolation scheme, it is very easy to add new kick types or change existing ones by changing parameters.

At the start of the kick, the torso is shifted so that the robot can stand solely on its support foot. The kicking foot is lifted, swung, retracted and extended to establish ground contact again. During the kick, the arms are moved to compensate the momentum in the z-axis (the vertical axis) generated by swinging the foot. Low-pass filtered gyroscope measurements are used as feedback to improve balance. The gyroscope roll and pitch, multiplied by gains, are added to the support foot ankle roll and pitch, respectively.

6.8 Sonar Filter

The sonar sensors allow the robot to estimate the presence of and distance to obstacles in front of it. In addition to the robot detection (cf. section 4.12), we use the sonar data to detect obstacles in the close vicinity of the torso. Since the raw sensor data is really noisy, filtering these measurements is indispensable.

Our sonar filter can be configured to use either a median filter or a low-pass filter and augmented with fundamental validity checks. The NAO documentation states that a reasonable detection performance can be expected in a range from 0.2 m to 0.8 m [20]. Below 0.2 m the sensor saturates, thus not being able to provide any reliable distance measurements. Therefore, we reject all measurements violating the aforementioned limits. Median-filtering the data helps dealing with sensor noise. Additionally, when using the lowpass filter, measurements far off the current distance estimation are penalized with a lower weight to achieve rudimentary outlier rejection.

The sonar sensors behave differently from robot to robot. The high stability of the filter, as described above, ensures the sonar sensor's usability as a reliable source of obstacle data in close proximity.

6.9 Orientation Estimation

Knowing the orientation of the torso with respect to the ground is essential for many tasks in robotics. While the rotation around the roll- and pitch-axis is a key input to estimate the body's pose and stability, knowledge of the rotation around an inertial yaw-axis is a helpful reference for localization tasks.

Therefore, the code base features an IMU sensor fusion approach based on [21]. The algorithm utilizes quaternions for internal state representation. This state representation avoids numerical issues arising from singularties during large orientation offsets, thereby allowing precise and robust orientation estimation.

6.10 Fall Manager

Being able to protect the robot's structural integrity during competitive games is key to surving the RobCup group phase with a non-zero number of NAOs. Specifically, unscheduled ground contact[1] puts severe stress on humanoid hardware (cf. fig. 6.1).

For this purpose, the framework features a `FallManager`. When triggered, this module reclines the head and slightly bends the legs backwards when falling to the front. Hip and chest are first to collide with the ground, absorbing most of the energy. Thus, neither head nor arms take damange, empirically increasing the chance for robots to complete games fully assembled.

[1] Commonly known as *falling*.

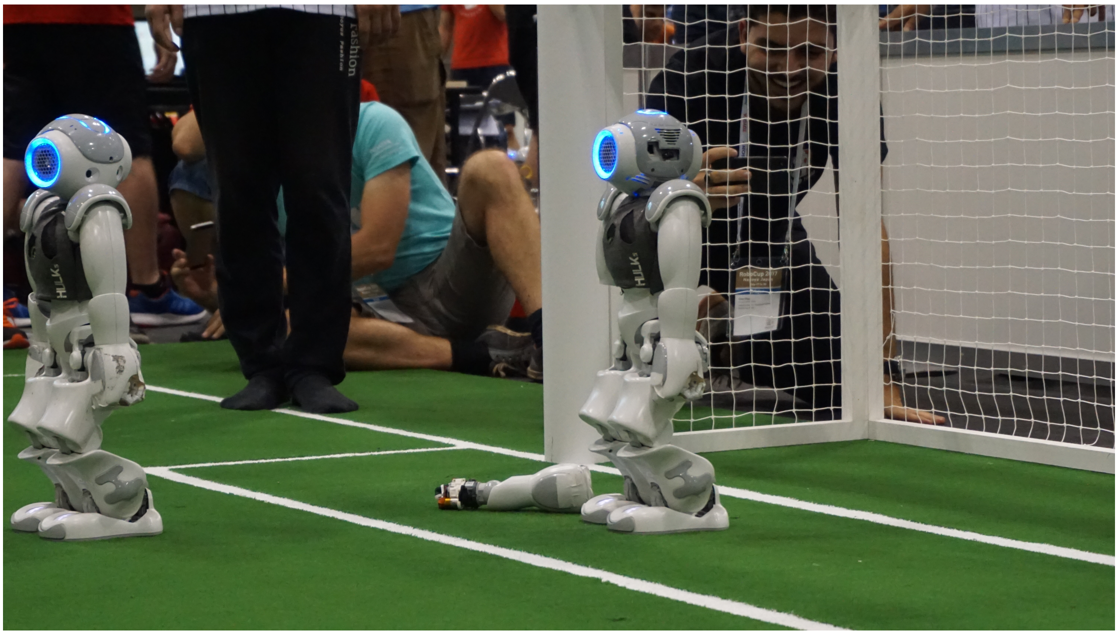


Figure 6.1: A robot that has lost its right forearm during a RoboCup 2017 game. That is of course no reason for a HULK to stop playing.

6.11 Collision Prevention

With the incremental penalty times [15], accumulating fouls is a severe disadvantage. To avoid these penalties, when a near-collisions scenario is detected with sonar sensors (cf. section 6.8) or foot bumpers (cf. section 5.13), the arms are moved behind the robot's back. However, the arm swing is deactivated in this collision avoidance mode which destabilizes walking. To compensate, the center of mass is moved slightly forward.

Chapter 7

Tools

This chapter explains the tooling we utilize both during development and in competition situations. During development, debug tools are necessary for visualizisations, data processing and measuring CPU utilization. Beyond this, structured organizational procedures of real game situations have proven to be useful.

Section 7.1 describes how team members are assigned to specific tasks before, during and after the game. Section 7.2 covers MATE, a tool for visualization, configuration, and calibration written in Python. In section 7.3 a tool to measure CPU utilization is described. Finally, section 7.4 explains how to debug directly on the NAO.

7.1 Pre- and Post-Game Process

This section describes our processes and the scripts used to prepare and finish a competitive game on RoboCup events. We figured out that having fixed processes prior to games helps getting consistent results during competitions.

7.1.1 Roles

Having persistent roles for specific tasks reduces chaos significantly. These roles are reassigned once a day at most. The roles are as follows.

Deployment Sets up the game branch and is the only person that is allowed to have a connection to the active robots.

Game log Writes down important events during games to discuss them in the post-game meeting.

Strategy Has the last word on parameter changes as well as changes to the game branch (e.g. if we want to dribble only).

Coach Assistant to the strategy guy. Exclusive interface to the head referee and `GameController controller` [1] during the game.

[1] It really should be called this way.

Assistants 6 people, each responsible for one robot (jerseys, robot placement etc.).

7.1.2 Pre-Game Process

90 minutes before a game officially starts we start preparing ourselves. The deployer branches the *game branch* off the repository's master branch and pushes it to our remote. He then starts merge-squashing camera, vision and walking parameters as needed on top.

40 minutes prior to the game, the *game branch* is fully prepared. The deployer then starts setting up the robots (scripts called from repository root):

Listing 7.1.1

```
scripts/pepsi playernumber 11:4 19:2 16:5 12:1 18:3
scripts/pepsi upload 11 12 16 18 19
scripts/pepsi wlan connect SPL_A 11 12 16 18 19
scripts/pepsi hulk restart 11 12 16 18 19
```

These commands do the following:

- Change the player numbers of all active players (robot 11 will have jersey number 4, robot 19 will have jersey number 2, ...)
- Compile for `Release`
- Upload the code to selected robots
- Clear all custom log files and replay data. This ensures that we do not run out of disk space.
- Connect all robots to the network (here: `SPL_A`)
- Restart the `hulk` service

30 Minutes before the game we have a procedure called *golden goal benchmark*. During this benchmark we start a test game at the field were the actual game will take place. During this test game we go through INITIAL, READY and SET like in normal games. After the whistle is blown in SET we measure the time our robots take to score a goal against the empty field and terminate the game immediately after we scored. We also terminate the game if it took us more than two minutes to score.

This very basic test shows us if our code works as intended. Problems in walking parameters, team behavior and communication, as well as problems in the vision pipeline, can easily be spotted in this period of time. If we observe something strange, we have some time to fix the problem without the need of a timeout.

After the *golden goal benchmark* is completed, we start our post-game procedure described in section 7.1.4.

10 Minutes before the game starts we re-upload our code to the robots (as described above) to ensure that all services are executed correctly.

5 Minutes prior to a game all robots should be connected to the correct network. The deployer gives a signal when the robots are ready to be carried to the field.

2 Minutes prior to the game all robots are placed on the field and get manually penalized by pressing the chest button.

7.1.3 Half-Time Process

All robots are taken back to the team zone in the half-time. The post-game procedure (see section 7.1.4) is executed to download all logs and replay files. At this point the person responsible for strategy may change parameters.

After the post-game commands are finished, we immediately start the pre-game procedure again and bring the robots back to the field.

7.1.4 Post-Game Process

Immediately after a game is finished, we start our post-game procedure. This process only consists of calling different subcommands of the `pepsi` tool:

Listing 7.1.2

```
scripts/pepsi logs download LOG_DIR 11 12 16 18 19
scripts/pepsi logs delete 11 12 16 18 19
scripts/pepsi hulk stop 11 12 16 18 19
scripts/pepsi wlan disconnect 11 12 16 18 19
```

These commands do the following:

- Download all logs and replay files
- Delete the logs and replay files from the robots
- Stopping the hulks service
- Disconnect the robots from the wireless network

After the post-game commands are finished, our team normally has a short meeting where the game log is being discussed and tasks are assigned.

7.2 MATE

In 2018 we developed a new tool for visualization, configuration and calibration. The tool is written in Python and is called MATE (Monitor And Test Environment).

7.2.1 Structure

The entire tool is split into a back-end (network communication and data management) and a front-end (PyQt windows and widgets). For each MATE-session there exists one NAO object holding the network back-end and higher level communication methods, e. g., to request a specific image. All networking- and communication-related sources are located in the *net/* directory. The user interface includes all widgets, panels and windows and is located in the *ui/* directory. The *run.py* script starts a *QApplication* and a *QMainWindow*. All other widgets and panels are dynamically created and shown by interaction with the MATE user interface.

7.2.2 Network Communication

MATE uses socket communication to send and retrieve data from a NAO or Simrobot session. The underlying protocol is built using the *asyncio* library. A stream connection is established utilizing respectively TCP- or UNIX-Sockets and the corresponding protocol defined in the NAO framework. Through this connection MATE subscribes *Debug-Keys* and receives the appropriate data, i. e., the associated value.

Any panel or element can subscribe one or more keys. The subscriber is identified with a custom string holding for example a uniquely generated ID. Additionally a callback function is registered. This function is used to pass the incoming data to the subscriber object.

Regarding the config protocol, MATE implements a similar subscriber hierarchy, disregarding that a config data request gets a single response.

7.2.3 Visualize Data in Panels

All MATE-panels are realized using *QDockWidgets* which enables dynamic arrangement and positioning. There are nine individual panels implemented: Aliveness, CameraCalib, Camera Register, Config, Image, ManualCamCalib, Map, Plot and Text. A combination of various panels is visualized in fig. 7.1. The `Text-panel` visualizes incoming data in JSON-formatted text. The `Image-panel` enables us to visualize live images that are rendered on the NAO (see fig. 7.2).

Numeric values can be plotted in the time domain using the *Plot-panel*. It is possible to visualize the values of multiple debug keys in fully customizable colors , e. g., sketch all joint angle values as seen in fig. 7.3. It is also possible to extract numeric values from complex data types utilizing a Python lambda function, which can be set in the configuration of the *Plot-panel*. An example of plotting the first element of an incoming array is shown in 7.2.1.

Listing 7.2.1

```
def parse(input):
    output = input[0]
    return output
```

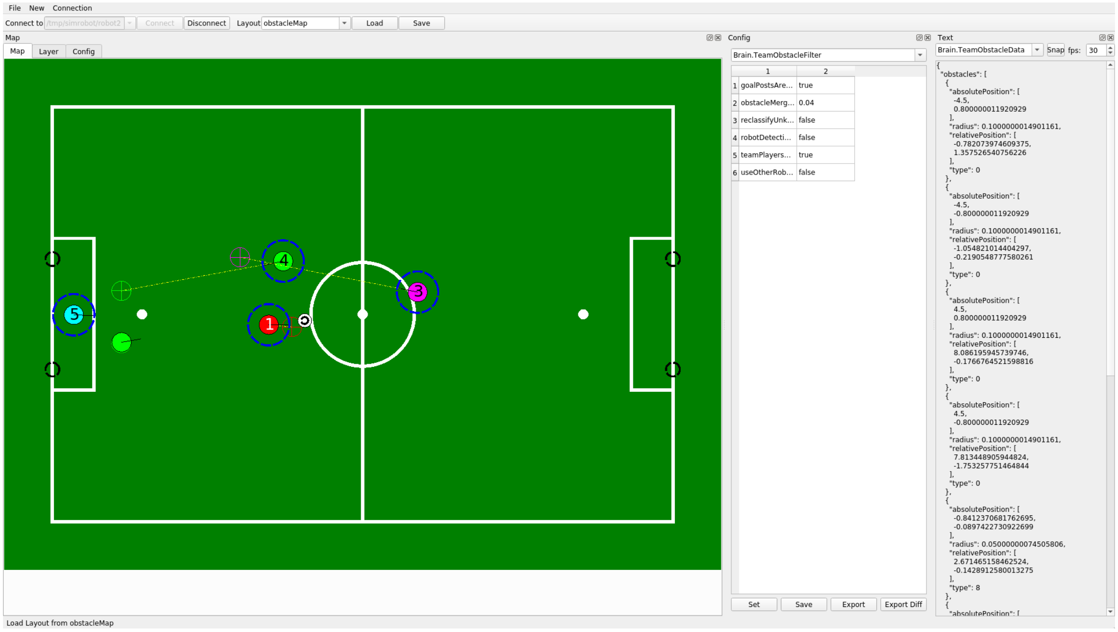


Figure 7.1: An example of combining a Map-panel picturing the registered obstacles of a NAO with a config-panel and a text-panel.

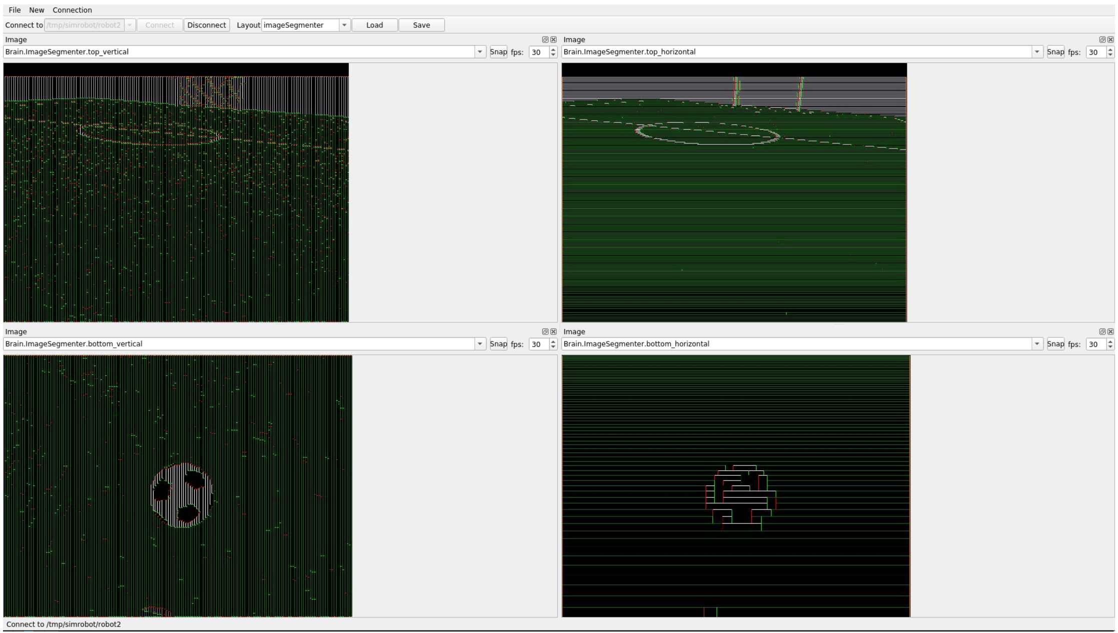


Figure 7.2: The image segmenter output in four MATE Image-panels.

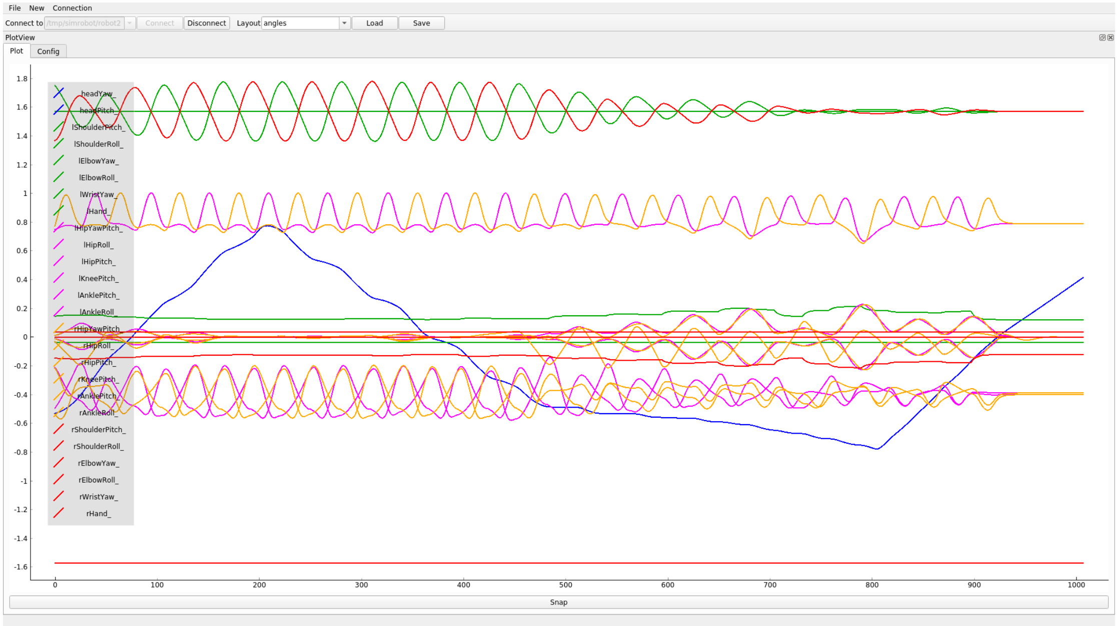


Figure 7.3: A Plot-panel showing all joint angles.

7.2.4 Higher-Level Data Processing Using the Map View

The *Map-panel* is a layered 2D top-down visualization. Implemented layers include both static and dynamic elements such as ball-position, players, playing field, obstacles, ball-search probability-map. For development and debugging of any given feature, a *Map-panel* with relevant layers has proven to be helpful. A *Map-panel* for the ball search (cf. section 5.5) is shown in fig. 7.4; the layer configuration is shown in fig. 7.5.

7.2.5 Config Export

MATE has a dedicated `Config` panel that shows config options as tables that can be edited in-place by the user. In the bar at the bottom now are two distinct `Export` buttons. The former (labeled `Export`) exports the currently displayed values as a file containing a json object you can save on your disk. The latter gives you several options. Each option exports a diff in form of a JSON against a specific set of settings. (For more information on how our approach works, see section 3.5)

New Default Does what the other export button does, creating a complete dump that can be used as a default for all robots.

Robot Default Creates a JSON containing only the changes between the currently active settings and the default for *all robots*. Can then be used as a new default for this robot, e.g. for location-independent calibration.

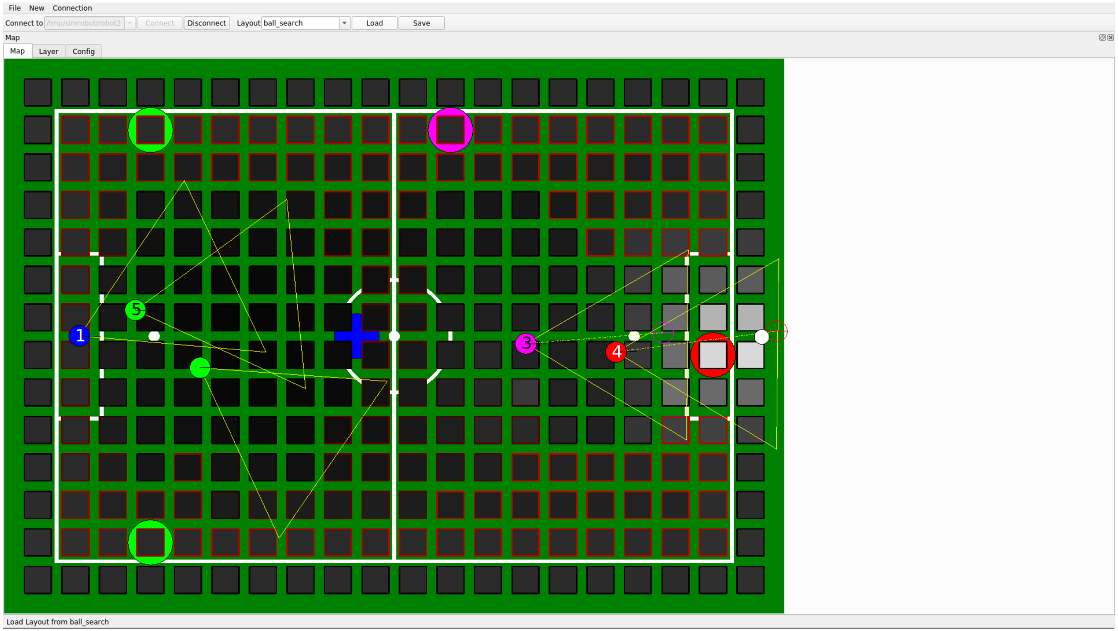


Figure 7.4: Using the Map-panel to visualize the current ballsearch model.

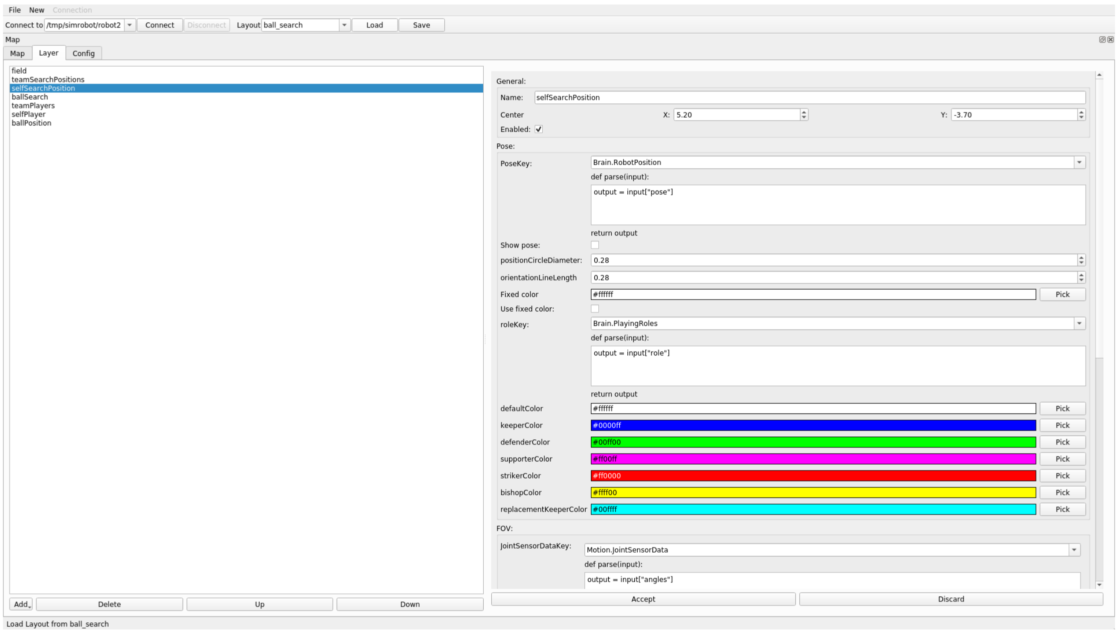


Figure 7.5: The layer configuration of a ball search visualization.

Location Default Creates a JSON containing only the changes between the currently active settings and the default for *this location* (for *all robots*). Can be used as a new default for a location (affecting *all robots*).

Robot + Location Creates a JSON containing only the changes between the currently active settings and the default settings for *this specific robot*. Can be used as a new setting for a *specific location* and a *specific robot*.

7.2.6 Motion Editor

A rudimentary motion editor exists as a MATE panel to create and modify keyframe animations. Keyframes can be captured from a connected robot, created from posefiles or by manually manipulating joint angles. Motions can be played on a connected robot via the puppet mode.

7.3 Profiling

We prepared our code to work with `Intel VTune Amplifier` [7]. It is a x86/x86_64 profiler that can be used to measure CPU utilization down to the instruction level with very low performance impact, enabling profiling during normal test games.

7.3.1 Prerequisites

VTune needs to be able to connect to the NAO without any password prompt. This can be accomplished by enabling ssh authentication via ssh keys. Adding the robot as a host to `.ssh/config` simplifies connecting via VTune:

Listing 7.3.1

```
Host ROBOT_IP
  HostName ROBOT_IP
  Port 22
  User nao
  IdentityFile /PATH/TO/IDENT_FILE
```

To be able to use VTune, CMake needs to find `libittnotify` on your system. Therefore the `NAO` compile target needs to be setup again. This should do the trick:

Listing 7.3.2

```
export VTUNE_HOME="~/intel/vtune_amplifier"
scripts/setup -t Nao
```

The output should contain a message like `Found ITTNOTIFY`. Uploading to the target robot can be done like this:

Listing 7.3.3

```
scripts/upload -b Release <NAOIP>
scripts/pepsi hulk stop <NAONUMBER>
```

7.3.2 VTune Configuration

After starting the `amplex-gui` of VTune it is possible to create a new project. The configuration should look like 7.3.4:

As `Analysis Type` we recommend to use the `Basic Hotspot` analysis. Choose a sampling interval (e. g. 1 ms) and make sure to check Analyze user tasks, events, and counters.

Listing 7.3.4

```
destination: nao@<NAOIP>
Application path: /home/nao/naoqi/bin/hulk
VTune Amplifier installation directory: /home/nao/intel
Temporary directory: /mnt/usb/tmp
```

7.3.3 Actual Profiling

After the `Start` button is pressed VTune will prepare the robot. This might take a while. When the initialization finished, the robot can be used as usual. We recommend to collect at least 180 s of profiling data to have good results.

7.3.4 Evaluation

When an analysis is finished, you can view the results inside VTune Amplifier. We also published a Python script that plots the runtime of all modules. It can be found inside the tools folder (`tools/IttNotify`) and needs `matplotlib` and `pandas` installed. A usage example can be found here:

Listing 7.3.5

```
plot_modules_from_ittnotify_data.py --supress-wait-modules \
~/intel/amplxe/projects/HULK/r000hs plot
```

It is possible to get all parameters and their description with `--help`. The resulting plot will resemble the one shown in fig. 7.6.

It should be noted that while the plot is already good for getting an overview of the modules' run-times, VTune itself makes it possible to analyze the performance in even greater detail.

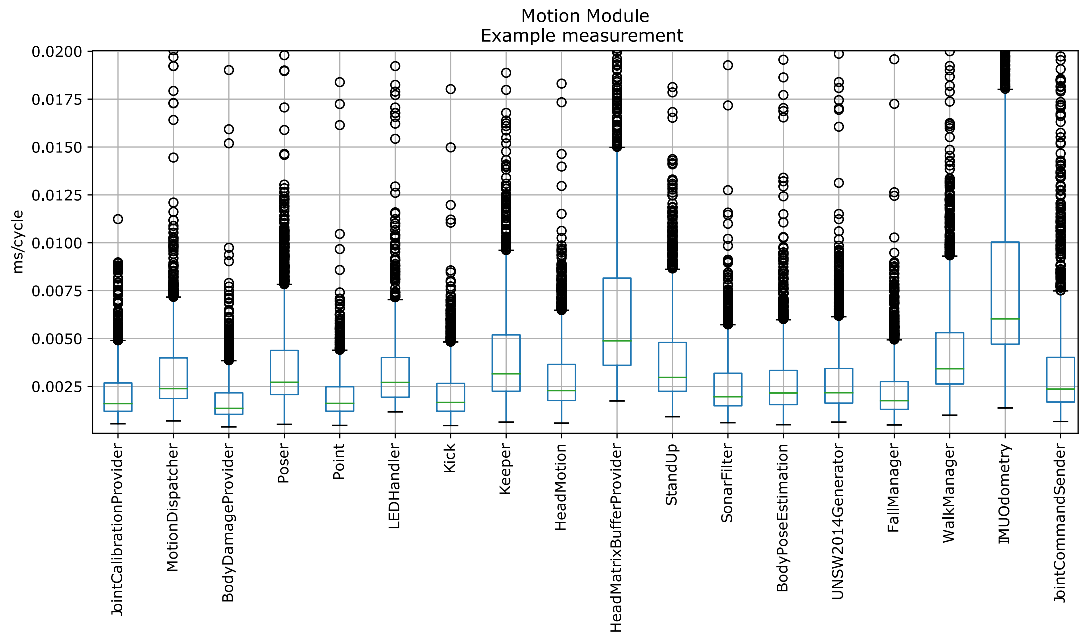


Figure 7.6: A box plot depicting the runtime of all Motion Modules. Black circles indicate outliers.

7.4 Debugging on a Robot

Sometimes it is necessary to debug program crashes or other strange behavior of the robots. For debugging the `hulk` process on the NAO, it is necessary to run it under `gdb`. Our repository contains a script for remotely debugging on the NAO. First compile and upload the `hulk` executable to the NAO. Make sure that no other instance of the `hulk` process is running already. Then use the script `scripts/remoteGdb` to get a GDB shell connected to a remotely instanciated `hulk` executable. At any time it is possible to return to `gdb`'s command line using `Ctrl-C` (this stops the execution of the process). If the process segfaults, `gdb` automatically returns to it's command line. Afterwards it is possible to type commands to probe the current state of the application. For example, to get the function the currently selected thread is inside, type `backtrace`. If `backtrace` does not list useful information, it is required to build `hulk` with a build type that includes Debug symbols, see section 2.4.

If you need more information about debugging with `gdb`, we refer you to the many existing tutorials on the topic.

Bibliography

[1] Basler, J.: Field Color Detection using Illumination Invariant Imaging in the RoboCup Standard Platform League (2018), https://hulks.de/_files/BA_Justus-Basler.pdf

[2] BHuman: Simrobot, https://github.com/bhuman/SimRobot

[3] Cyberbotics Ltd.: Webots, https://www.cyberbotics.com

[4] Furuhashi, S.: Messagepack, https://msgpack.org

[5] Hasselbring, A.: Implementierung und Evaluation einer Pfeifendetektion für den NAO-Roboter (2017), https://www.hulks.de/_files/BA_Arne-Hasselbring.pdf

[6] Hengst, B.: rUNSWift Walk2014 Report. Technical report, School of Computer Science & Engineering University of New South Wales (2014), http://cgi.cse.unsw.edu.au/~robocup/2014ChampionTeamPaperReports/20140930-Bernhard.Hengst-Walk2014Report.pdf

[7] Intel Corporation: Intel vtune profiler, https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.nyn5nv

[8] Konda, Y.: Whistle Sound Source Localization Using Multiple NAO Robotic Systems (2019)

[9] Loth, P.: Detektion von Feldmerkmalen auf dem NAO-Robotiksystem in der RoboCup Standard Platform League (2018), https://www.hulks.de/_files/PA_Pascal-Loth.pdf

[10] Microsoft Corporation: Vcpkg, https://github.com/microsoft/vcpkg

[11] Nölle, K.V., Sieck, H., Gleske, P.: Neural network architecture search for ball detection using a distributed and scalable genetic algorithm (2021), https://hulks.de/_files/Paper_Neural_Network_Architecture_Search_for_Ball_Detection_Using_a_Distributed_and_Scalable_Genetic_Algorithm.pdf

[12] OpenEmbedded: Openembedded, https://www.openembedded.org/wiki/Main_Page

[13] Peters, L.: Adaption und Vergleich von nichtlinearen Filtermethoden zur Selbstlokalisierung auf einem Feld mit dem humanoiden NAO-Robotiksystem (2017), https://www.hulks.de/_files/BA_Lasse-Peters.pdf

[14] RoboCup Technical Committee: RoboCup Standard Platform League (NAO) Rule Book (2018), http://spl.robocup.org/wp-content/uploads/downloads/Rules2018.pdf

[15] RoboCup Technical Committee: RoboCup Standard Platform League (NAO) Rule Book (2019), http://spl.robocup.org/wp-content/uploads/downloads/Rules2019.pdf

[16] Röfer, T., Laue, T., Bülter, Y., Krause, D., Kuball, J., Mühlenbrock, A., Poppinga, B., Prinzler, M., Post, L., Röhrig, E., Schröder, R., Thielke, F.: B-Human Team Report and Code Release 2017 (2017), http://www.b-human.de/downloads/publications/2017/coderelease2017.pdf

[17] Siemens AG: Kas, https://github.com/siemens/kas

[18] Siemens AG: kas-container, https://github.com/siemens/kas/blob/master/kas-container

[19] Siemens AG: Kas user guide, https://kas.readthedocs.io/en/latest/userguide.html

[20] SoftBank Robotics: Sonars, http://doc.aldebaran.com/2-8/family/nao_technical/sonar_naov6.html

[21] Valenti, R.G., Dryanovski, I., Xiao, J.: Keeping a Good Attitude: A Quaternion-Based Orientation Filter for IMUs and MARGs. Sensors 15(8), 19302–19330 (2015)

[22] Warmuth, F.: A Graph Based Path Planning Approach for the RoboCup Standard Platform League (2019), https://hulks.de/_files/PA_Felix_Warmuth.pdf

[23] Yocto Project: Yocto project (2022), https://www.yoctoproject.org

[24] Zhang, Z.: A flexible new technique for camera calibration. IEEE Transactions on Pattern Analysis and Machine Intelligence 22(11), 1330–1334 (Nov 2000)

All links were last followed on Tuesday, December 14th 2021.