HULKs Team Research Report 2018

Darshana Adikari, Georg Felbinger, Pascal Gleske, Chris Kahlefendt, Yuria Konda, René Kost, Pascal Loth, Konrad Nölle, Lasse Peters, Nicolas Riebesel, Thomas Schattschneider, Maximilian Schmidt, Erik Schröder, Felix Warmuth, Felix Wege

RobotING@TUHH 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 reached the RoboCup 2018 semi-finals.

Our ambition for 2018 was the improvement of robot behavior in one-on-one situations. This was accomplished by rewriting the behavior modules and tuning our motion planning towards more aggressive dribbling as well as introducing a foot collision detector and refining the sonar filter. Adopting the walking engine of rUNSWift–in the version ported by B-Human—enabled us to increase our walking speed significantly. We created a new python-based tool called MATE for live data visualization, configuration and calibration.

This report serves as partial fulfillment of the pre-qualification requirements for RoboCup 2019. For this purpose, it is accompanied by a version of the code that has been used at RoboCup 2018.

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 oulines 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, and in replay mode on a Linux machine.

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 replay (see section 2.4):

> C++14 compiler (GCC $\geq$ 5.0.0, Clang $\geq$ 3.4), git, CMake, bzip2, libpng, libjpeg-turbo, zlib, fftw, 3 $\geq$ 3.3, portaudio, ccache, qt5-base, qt5-svg, glew, libxml2, ode

To build the `cross-toolchain` required to cross compile for the NAO robot the following packages are needed:

> build-essentials (gcc, make, ...), git, automake, autoconf, gperf, bison, flex, texinfo, libtool, libtool-bin, gawk, libcursesX-dev, unzip, CMake, libexpat-dev, python2.7-dev, nasm, help2man, ninja

To communicate (uploading code, configuring) with robots, the following packages are required:

> rsync, ssh, curl

It should be mentioned that our code is optimized to run on a NAO v5 since this version has a three axis gyroscope. However, it is possible to run the code on older versions of the robot with only small changes required.

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 2018
```

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

```
scripts/files/net
scripts/lib/numberToIP.sh
scripts/gammaray (line 120, insert teamname here)
home/configuration/location/default/id_map.json (explanation below)
```

The `id_map.json` need 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 this (while `####` needs to be replaced by the last 4 digits of the serial number of the robot part.).

Listing 2.2.3

```
{
"idmap.nao": [
  {
    "bodyid": "####",
    "headid": "####",
    "name": "NAMEnao01"
  },
  {
    "bodyid": "####",
    "headid": "####",
    "name": "NAMEnao02"
  }]
}
```

2.3 Building the Toolchain

To build the `hulks cross-toolchain` the dependencies listed in section 2.1 must be met. Afterwards one can run the following commands inside the repository root. A fast internet access and a few hours of spare time are recommended.

Listing 2.3.1

```
cd tools/ctc-hulks
./0-clean           && \
./1-setup           && \
./2-build-toolchain && \
./3-build-libs      && \
./4-install
```

After all scripts completed, there should be two new important files:

- `ctc-linux64-hulks-7.3.0-1.tar.bz2`
- `sysroot-7.3.0-1.tar.bz2`

2.4 Building the Robot Software

Currently, the code supports the following targets

- `nao`
- `simrobot`
- `replay`

The first target compiles the code to be able to run on a NAO with the `hulks cross-toolchain`. The second target compiles the code to be executed in SimRobot. The last target enables to feed a prepared dataset into the code to be able to deterministically test our code.

There are four different build types:

- `Debug`
- `Develop`
- `Release`
- `RelWithDebInfo`

The `Debug` type is for debugging only since optimization is turned off and the resulting executable is very slow. It should not be used for normal testing or in competitions. The `Develop` type is the normal compilation mode for developing situation. The `Release` mode is used in actual games which mainly removes assertions. The `RelWithDebInfo` type is equal to the `Release` type except it contains debug symbols. This mode is especially useful for profiling (see section 7.3). The resulting executable is quite huge so that we do not recommend using it in competitive games.

2.4.1 SimRobot

Our repository comes with its own version of SimRobot which one needs to compile first:

Listing 2.4.1

```
cd tools/SimRobot
./build_simrobot
```

After building SimRobot, the project needs to be configured with CMake. This can be done by using the setup script as follows (from repository root):

Listing 2.4.2

```
./scripts/setup simrobot
```

Then the code base can be built for SimRobot by executing the command 2.4.3.

Listing 2.4.3

```
./scripts/compile -t simrobot -b <BuildType>
```

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

Listing 2.4.4

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

2.4.2 Replay

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

Listing 2.4.5

```
./scripts/setup replay
./scripts/compile -t replay -b <BuildType>
```

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

2.4.3 NAO

The scripts need to know the locations of HULKs- and SoftBank-toolchains in order to compile the code. Therefore exctract the `ctc-linux64-hulks-7.3.0-1.tar.bz2` and `ctc-linux64-atom-2.1.4.13.zip`:

Listing 2.4.6

```
cd ~/naotoolchain
tar xf ctc-linux64-hulks-7.3.0-1.tar.bz2
unzip ctc-linux64-atom-2.1.4.13.zip
```

After that, the toolchain can be initialized inside the code repository:

Listing 2.4.7

```
./scripts/toolchain init ~/naotoolchain
```

Finally, the code can be configured and build:

Listing 2.4.8

```
./scripts/setup nao
./scripts/compile -t nao -b <BuildType>
```

2.5 Setting up a Robot

In order to setup a NAO, place a symlink inside the toolchain directly pointing at the `sysroot-7.3.0-1.tar.bz2` at first. Subsequently, the `gammaray`-script can be executed to prepare the NAO for running our code. `NAOIP` needs to be replaced by the current IP address of the robot, while the `NAONUMBER` may be an arbitrary number in the range from 1 to 240. This number will be used to identify a robot on the network afterwards.

Listing 2.5.1

```
ln -s sysroot-7.3.0-1.tar.bz2 toolchain/sysroot.tar.bz2)
./scripts/gammaray -w -a <NAOIP> <NAONUMBER>
```

Once the script terminated successfully, the NAO should restart itself and be ready to run the code. The robot's hostname changes to `nao<NAONUMBER>`, while the IP Address will change to `10.1.TEAMNUMBER.NAONUMBER + 10` during this process.

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 `upload`-script:

Listing 2.6.1

```
./scripts/upload -dr <NAONUMBER>
```

The script will upload the compiled code and configuration files to `nao<NAONUMBER>` and restart the hulks-service.

As executing these scripts is rather painful for multiple robots, we introduced scripts like `pre-` and `postgame`. A detailed description on how to use these scripts can be found in section 7.1.

2.7 Debug Tools

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 `pythonRequirements.txt` are installed.

Listing 2.7.1

```
pip install -r pythonRequirements.txt
```

To start MATE run:

Listing 2.7.2

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

After starting MATE, connect to a NAO or SimRobot session. By clicking the *New* button a new *View* can be opened. To see live images from the robot, add an image view with the desired image key. There is also a feature called layouts to save and load arrangements of views. 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 be able to use the framework (also known as `tuhhSDK`) for different robots, offline processing or simulation there exists the `robotInterface`. It has methods to get the `cameraInterface`, `audioInterface` and to control the robot. The `tuhhSDK` also provides `Debug` (see section 3.4) and `Configuration` (see section 3.5) capabilities.

3.1 Module Architecture

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

The connection of these modules can be modeled as a bipartite data flow graph made of modules and the data types they exchange. `DataTypes` are stored in a `database` for each module manager[1] as can be seen in fig. 3.2. The relation of modules and data types can be either that a module produces a data type or that it depends 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.

Each module must have a `cycle` method that executes the code that has to be run for 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 actually 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 [14], 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 capable of reading C++ can easily read the code at the cost of a bit more verbose declarations of dependencies and productions.

Modules can be enabled or disabled before the startup of `tuhhNao`. To do this every

[1] Brain and motion implement the module manager interface.

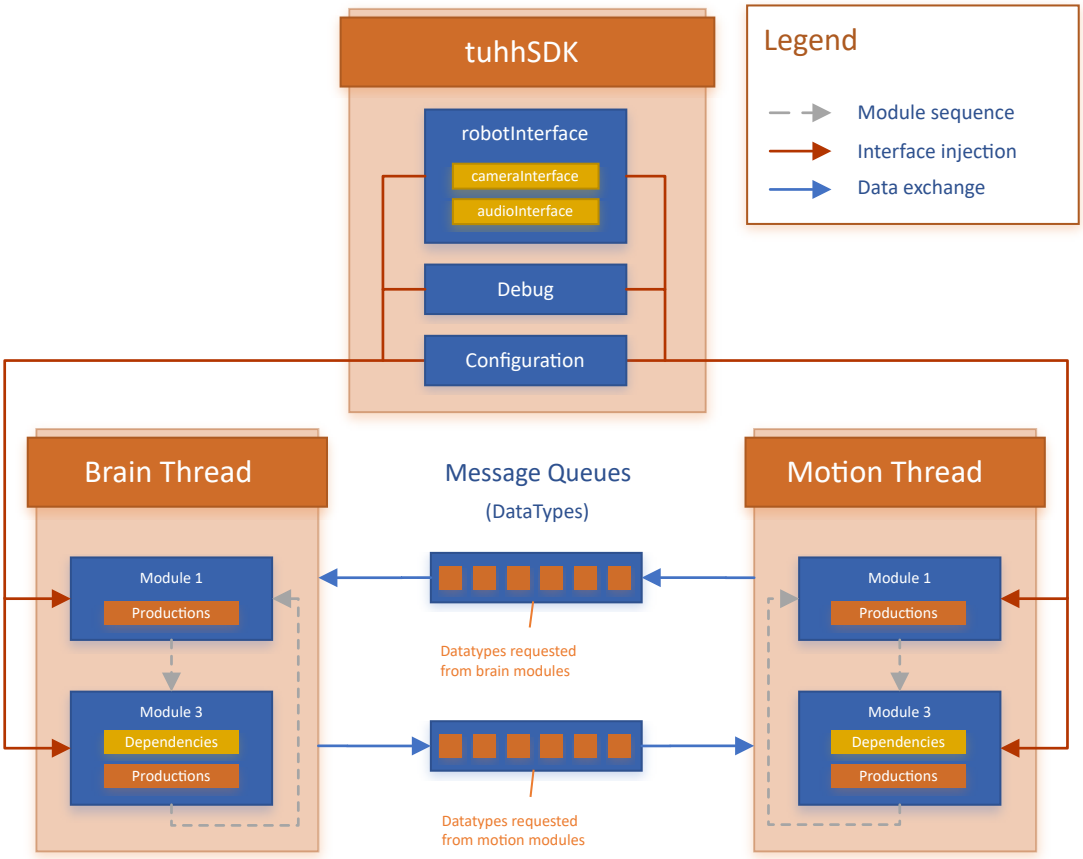


Figure 3.1: Overview over the general module architecture of our framework.

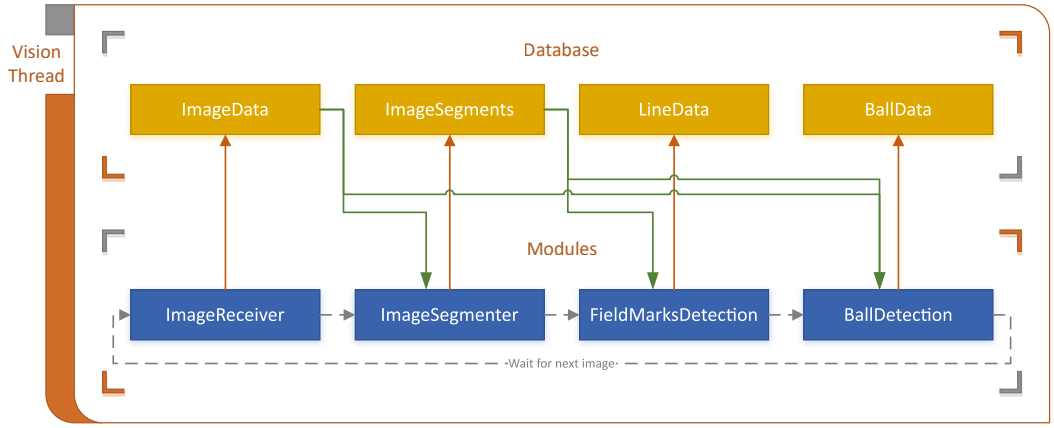


Figure 3.2: An example of a module execution order and the associated `DataTypes` stored in the `Database`. Productions are marked with orange arrows and dependencies with green ones.

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. Each setup resides in `home/configuration/location/default/`. Every module setup has to start with `moduleSetup_`. The currently active setup can be configured inside the `home/configuration/default/tuhh_autoload.json` entry: `moduleSetup`.

Every module has to be listed inside the `moduleSetup_default.json` if it should be enabled by default or not. This can be overridden inside the specialized module setup file. We already created a few module setups. 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 of robots in real-time.

3.2 Threads

Autonomous interaction with the environment requires to evaluate a variety of sensor outputs as well as updating the actuator control inputs at an adequate rate. Since the update rate of the camera images is lower than the one of the chestboard, our framework features two different threads, each of which is synchronized with the associated hardware update rate, to run the related processing algorithms. Additionally, messaging infrastructure is provided to safely share data between threads.

The motion thread processes all data provided by the DCM[2]. These are accelerometer, gyroscope and button interface sensor inputs as well as joint angle measurements and sonar data. Any data provided the DCM is updated at 100Hz, which thus is the frequency the motion thread is scheduled.

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

3.3 Messaging

Data types that are 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. It is made sure that 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. On the PC side there is a python application called MATE that connects to the NAO via a TCP socket (or to a simrobot instance via an unix socket). MATE is documented in the tools chapter section 7.2 of this document.

3.5 Configuration

We have a configuration system that loads parameters from JSON files which are named identically to the belonging module. It will look for configuration files in multiple directories that are e. g. specific to a NAO head, NAO body or a location such as RoboCup 2017 SPL_A. Parameters that are set in more specific files will override parameter values from generic files. It is also possible to change parameters at runtime via the aforementioned MATE tool. Receiving new values can cause a callback to be run so that pre-calculations based on these parameters can be redone.

[2]Device Communication Manager

If there are specific JSON files to load, their location name can be specified inside *home/configuration/location/default/sdk.json* in the parameter `location`. This name is used to load configuration files a second time from a subfolder with this name inside the *location* folder. 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.

3.6 Replay Recorder

We provide a module for recording live data called the *replay recorder*. In the past, algorithm input data was processed directly on the robot and the results were logged to the robot's storage. There were multiple drawbacks to this method. First, all the collected data may become deprecated as soon as the algorithm generating that data changes. Furthermore, if we take an image-based candidate generator as an example, it becomes impossible to infer something like the real detection rate if only the successfully generated candidate output gets stored, because that data is then missing the information about all the missed objects.

Therefore a new framework for collecting data was implemented. We developed the replay recorder module, which is capable of processing and storing all image and sensor data as well as metadata like which buttons were pressed. Within a fixed rate, the module checks whether a frame can be recorded and acts accordingly.

The collected frames can be reprocessed again by the HULKs framework by using the *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.

3.7 SimRobot

SimRobot[3] is a simulator developed at the University of Bremen and the German Research Center for Artificial Intelligence. We developed our own controller that integrates our usual code into the simulator. It is possible to simulate multiple robots at once to be able to evaluate team behaviors.

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

3.7.1 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. We attain this effect

[3] `http://www.informatik.uni-bremen.de/simrobot/index_e.htm`

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, we use the fact that the computation of the average can be implemented using bit-wise operators on larger datatypes (`int64` instead of `byte`). This reduces the loop cycles necessary to iterate over the whole image.

3.7.2 New Field Texture

To make the field less monotone, we replaced the shadow-texture with a higher resolution image which resembles a 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 4.4) and segmentation (see section 4.5) of the image to reduce the amount of data to be processed by subsequent object detection algorithms. Subsequently another module tries to identify the field border (see section 4.6), i. e. the area where the carpet ends. The objects that are currently detected are the penalty spot, the field lines, the center circle and the ball (see section 4.7, section 4.8, section 4.9 and section 4.10, respectively).

4.1 Image Data

The image data is provided through the `RobotInterface` which holds both camera objects as members. The Camera class is configured so that we receive 640×320 pixel images in the `YUYV format`. The RobotInterface then provides a method which returns the camera (top or bottom) to use in the next vision cycle. Hereby the camera with the oldest, not yet processed data gets returned.

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

As we request a YUYV image, all modules and its algorithms use native YCvCr422 pixel data without treating any Y-values as padding. However, conversion to YCbCr444 is currently needed for generating debug images. This is currently done on the robot and not in our debug tools.

4.2 Camera Calibration

The two cameras of the NAO have to be calibrated for optimal performance. Camera calibration consist of intrinsic (determining focal lengths and centres) and extrinsic (ad-

justing camera pose matrix using the kinematic chain) stages.

While intrinsic calibration is needed less frequently, 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.

While the general methodology remains the same, UI and underlying calibration logic was rewritten for the new python based debug tool (MATE) during this season. Usage of Numpy, OpenCV 3 greatly reduced implementation time compared to the calibration tool made for OFA in the previous season by using built in non-linear solvers such as Levenberg-Marquardt and intrinsic calibration functions based on [19] and others. Another change is the architecture was altered to support multiple patterns of different classes (Aruco, ChAruco, Chessboard) [4]. as a prerequisite for the joint calibration research done at the moment. However, only the ChAruco pattern is supported out of the box, other options are not tested.

4.2.1 Kinematic Chain

The kinematic transformations from the ground point (generally positioned between the feet of the robot) to the camera of the robot is 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 caused by the respective transformation matrix. For example, *camera2Ground* describes the transformation from the camera to the robot's head; *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 is different between the two cameras. The matrices are supplied by SoftBank Robotics [17]. The formation of kinematic matrices from a ground point to the camera is as follows.

$$camera2Head(\alpha, \beta, \gamma) = camera2HeadUncalib \times R_{ext}(\alpha, \beta, \gamma) \tag{4.1}$$

$$\begin{aligned} camera2Ground(\alpha, \beta, \gamma, t) = {} & torso2Ground(t) \\ & \times head2Torso(t) \\ & \times camera2Head(\alpha, \beta, \gamma) \end{aligned} \tag{4.2}$$

$$ground2Camera(\alpha, \beta, \gamma, t) = camera2Ground(\alpha, \beta, \gamma, t)^{-1} \tag{4.3}$$

The matrices containing parameter t indicate their value may change over time i. e. due to moving of the NAO. This distinction is important at the step of capturing images and kinematic matrices for a given frame. Since the cycle times of motion thread and brain thread (which runs vision modules) are different, the extra precaution of capturing images and kinematic matrices while the robot is still was taken to ensure time synchronism of images and kinematic matrices.

4.2.2 Extrinsic Calibration

The 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 for same array indices.

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.

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

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.

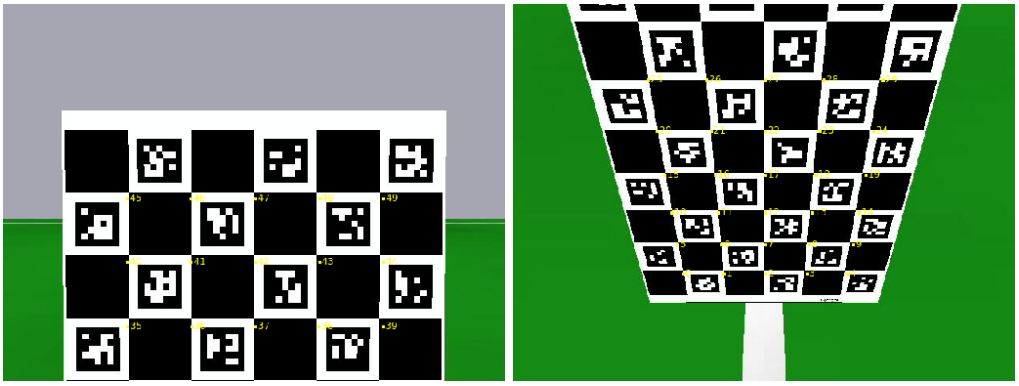
4.2.3 Intrinsic Calibration

With the inclusion of OpenCV framework, it was possible to use the well tested, mature implementation of OpenCV to perform intrinsic calibration. Therefore major amount of testing and verification for intrinsic calibration could be removed.

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.

4.2.4 Validation, Qualification

Validation of this tool is the process of ensuring accuracy of computed calibration parameters. Qualification is the step of determining that this process and tooling is fit for usage during competitions where accurate and timely calibration is crucial. The accuracy of the extrinsic calibration was verified as follows.



(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.

1. Visually verify that the projection of markers of the fixture coincide with the border of the observed markers. This facility is provided in the calibration tool UI (cf. fig. 4.1).

2. Check the penalty box projection used for manual extrinsic calibration. In event of correct calibration, the projection must coincide over the penalty box of the field. The user should verify the satisfaction of this condition.

The accuracy of the intrinsic calibration was verified solely by observation of the marker projection shown in the calibration tool. The reason for this choice was that any error in calculation or projection was directly observable for the case of intrinsic calibration while it wasn't possible for the extrinsic calibration (which required the need for two-step verification).

Similar to the last season, unit testing was introduced to verify calculations of the calibration library. Once the verification was completed, the qualification step was performed that included evaluation of software reliability, computational time and user experience. Feedback from team members were used to further improve these factors.

4.2.5 Obervations & Remarks

The following section presents the observations and findings of the verification and qualification steps. The average time values account for finding optimal calibration values for a data set consisting of approximately 300-400 correspondence pairs.

- The accuracy of extrinsic calibration was similar or better compared to manual calibration. While time synchronization was much better compared to the original iteration in JavaScript, there were inaccuracies of the calibration results at certain

instances (esp. Bottom camera) forcing us to use the penalty area based manual calibration.

- It was discovered during RoboCup 2018 that some of the cameras showed significant amount of radial distortion. This surely contributed for the errors due to the positioning of the calibration rig while it was also determined that another review of the calculations has to be performed.
- Although torso calibration was included and tested in extrinsic calibration, the results were not reliable, since the variety of body poses during capture was not enough to detect the torso posture error.
- Accuracy of intrinsic calibration was equivalent to the traditional method of using Matlab calibration toolbox [1] as OpenCV's calibration algorithms are based on that. The output also allowed to conclude if the particular camera has significant radial distortion. [11]
- Automated extrinsic calibration of both cameras generally took approximately 90ms (25 captures, excluding time for movement). This is a massive speed up compared to previous 2+ minutes. The contributing factor was usage of SciPy solvers which internally call highly optimized solver implementations in C.
- Automated intrinsic calibration took approximately 9 seconds for both cameras. This is 12 times speed increase compared to the previous year's implementation.

Testing of the tool for MATE started with Iran Open 2018 and was also used during German Open 2018 and RoboCup 2018 (to lesser extent due to reliability issues probably contributed with lens distortion). Due to the availability of proper intrinsic camera calibration, it was possible to fully calibrate the cameras although external calibration was less than reliable forcing secondary fine tuning at the field with penalty box.

At the moment research is being performed on possibility of joint calibration of the NAO as a Project Thesis. Based on the results of that, it'll be possible to determine optimal positioning of calibration patterns, etc. In addition, additional investigation has to be performed to determine the reliability issues faced during RoboCup 2018.

4.3 Robot Projection

It often happens that body parts, such as shoulders or knees, appear in the image. To avoid falsy percepts in these regions of the image, knowledge of the forward kinematics is used to project the 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 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 false candidates in the ball and line detection algorithms.

4.4 Field Color Detection

To determine the field color a derived version of k-means clustering of the pixel colors is used. 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 lower than a threshold and the Y value is lower 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 mean is limited and remains unchanged in these cases.

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 approximately 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 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 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.

4.7 Penalty Spot Detection

First the penalty spot detection is using the horizontal scanline segments only. There are fewer horizontal scanline segments 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 and too far away from the robot. The detection distance is limited to 3m in order to reduce potential false positive detections. Then the pixel size of a theoretical penalty spot at the segment 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 theoretical point of intersection is calculated which is defined as the point in which the two segments would overlap if they intersected each other in the segment's mid point. The theoretical point of intersection represents the penalty spot center point. Twelve equidistant points on an elliptical path are calculated around the center point. The main and minor axis of the ellipse can be calculated by back projections of the real penalty spot dimensions. Each of the points on the ellipse must be inside the image and have a darker luminance value than the penalty spot center. In addition, either each point must have a higher chrominance compared to the center or can 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 [10].

4.7.1 Penalty Area Detection

By combining the detected penalty spot with a line from the penalty box we can define a feature we call the penalty area. The penalty area has the advantage that it can be used to improve on the orientation estimation for our robot localization. For penalty area detection we can simply use the known dimension of the field. The penalty area is detected by looking for a line which corresponds to the known distance between the penalty point and the penalty box.

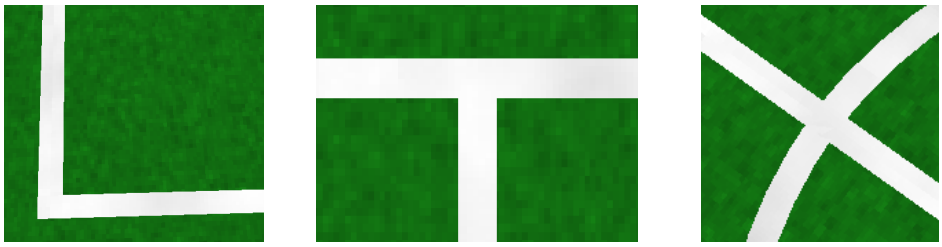
4.8 Line Detection

The line detection takes image regions that start with a rising edge and end with a falling edge. The gradients of both edges must point to each other and should be parallel. In situations where sunlight creates bright spots on the field, the real length of the region can be checked to better distinguish them from field lines. 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 larger gap, it is separated. If a resulting line segment contains too few points it is discarded.

4.8.1 Line Intersections

For localization purposes it is a good idea to create high-level features from detected lines. A single line can basically appear in almost any part of the field. By combining lines which intersect we can create features which are much rarer and therefore better for improving the localization of our robots. For these purposes we differentiate between three types of intersections as depicted in fig. 4.2.



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

Figure 4.2: Different types of line intersections.

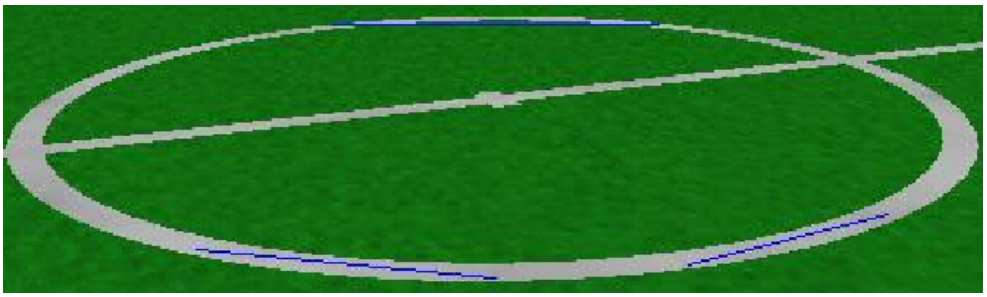
We merge detected line segments into intersections by performing the following steps:

1. Find all pairs of line segments which are more or less orthogonal to each other. For the following steps we use the exemplary pair (A, B).
2. For each of these pairs find their point of intersection p_i (does not necessarily have to lie 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. We will 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 we differentiate between the following cases:
   - $x_A > 0$ 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

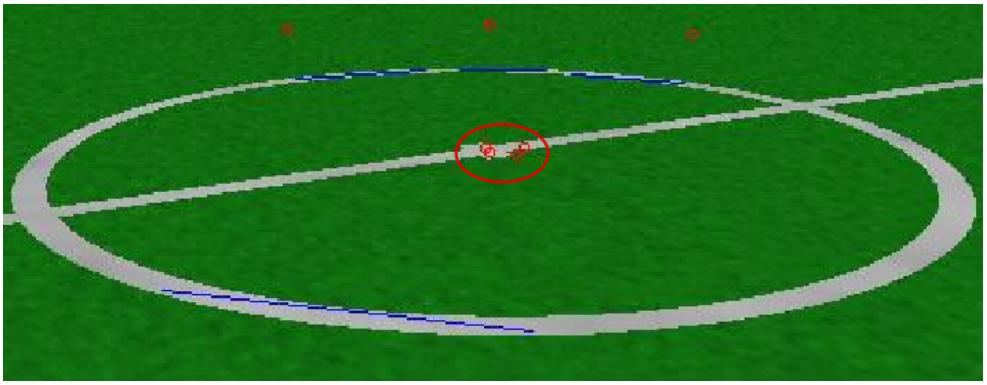
4.9 Center Circle Detection

Since we are detecting line segments within the center circle as shown in fig. 4.3a we can use these to estimate its position. To realize this we take the middle point of a detected line and orthogonally place two points with a distance of the center circle radius to the left and the right of the line. After doing this for all lines we then try to find clusters of the points we placed. Considering only clusters which contain more than a minimum of points and do not spread too much around the center of the cluster,

we then take the center of the cluster which holds the most points as our center circle estimate. One cluster found this way is shown in fig. 4.3b. In order to be able to also use the orientation of the center circle for our localization we look for a line that moves through the previously found cluster.



(a) Line segments detected in the center circle.



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

Figure 4.3: Visualization of center circle detection.

4.10 Black and White Ball Detection

As of 2016, the Black and White Ball Detection consisted of a derived version of the red ball detection, suffering from many false positives and low detection ranges. The need for a more robust ball detection motivated us to explore new possible solutions to this problem. As another purely algorithmic approach with the aim of color-independent ball detection had already been implemented previously, but has also proven to be infeasible [15], the desire arose to try out machine-learning-based solutions. Approaches based on convolutional neural networks for object detection lead to promising results in our previous work, such as the robot detection [8]. These methods save a lot of work, as no manual feature extraction is necessary. Still, there has to be a region of interest search due to the lack of computation time for inferencing a network. There are also

(a) Seed points corresponding to the center of the black patches on the ball.

(b) Merged seeds and projection of the corresponding ball radius.

(c) Reprojected ball from result of the ball filter (green circle within black rectangle).

Figure 4.4: Visualization of ball candidate seeds, merged seeds and reprojected ball [2, pp. 17–18].

many hyperparameters for the structural setup of the networks. This method has been developed in a project thesis [2] and was later published as a paper [3].

4.10.1 Candidate Generation

As the first step of the candidate generation, the algorithm determines points lying on the ball. This is done by searching the segmented image for dark regions that are surrounded by brighter pixels. These points are called seeds, being the center points of segments corresponding to the black patches of a ball.

After the calculation of all seeds in an image, nearby seeds within the range of a projected diameter of a ball get merged to candidates. The position and radius of a candidate is estimated by the mean of the corresponding seeds.

Additionally, the increased accuracy of the ball state prediction allows to use the reprojected the predicted ball position as an additional source of ball candidates. While this strategy already showed some promising results–improving the tracking performance of balls with higher velocity–it can only be successful if the prediction is accurate enough to relocate the ball within the image. Since this is often not the case, future work could focus on generating additional candidates by reprojecting sigma-point like samples from the estimated state distribution. The process of candidate generation is illustrated in fig. 4.4.

4.10.2 Training New Models

After collecting and labeling candidates, our training set consisted of 16880 positive and 23768 negative examples and the test set, collected at a testing event, of 5687 positive and 12730 negative images.

The structure of the convolutional network classifying the candidates was optimized using a genetic algorithm. The search spaces consisted of parameters like the sample size, number and size of the convolutional and hidden layers, etc. For the fitness function, a combination of the classification performance and the inference complexity was used. The classification performance was determined by the true positive and true negative rates and the inference complexity by an asymptotic approximation [2, pp. 20–24].

Chapter 5

Brain

The "Brain" part of our code base is divided into two domains: Gain of knowledge and coordinating team behavior. The former is concerned with localization (section 5.2), team ball filtering (section 5.4) and whistle detection (section 5.10) 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

In the past season, we have introduced a dynamic role assignment that assigns the playing roles during the game based on the world model. To coordinate the team behavior, roles are assigned 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 supporter, a bishop, and a replacement keeper. Roughly, the roles have the following tasks:

1. Keeper: Guard the goal.
2. Striker: The robot playing the ball towards the opponent goal.
3. Defender: Defensively positioned robots within own half. There can be up to two defenders.
4. Supporter: Stays close to the striker in case the striker loses the ball or falls down.
5. Bishop: Tries to be a favorable pass target by occupying an offensive position on the opponents half.
6. Replacement Keeper: Guard the goal while the keeper is penalized or far away.

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.

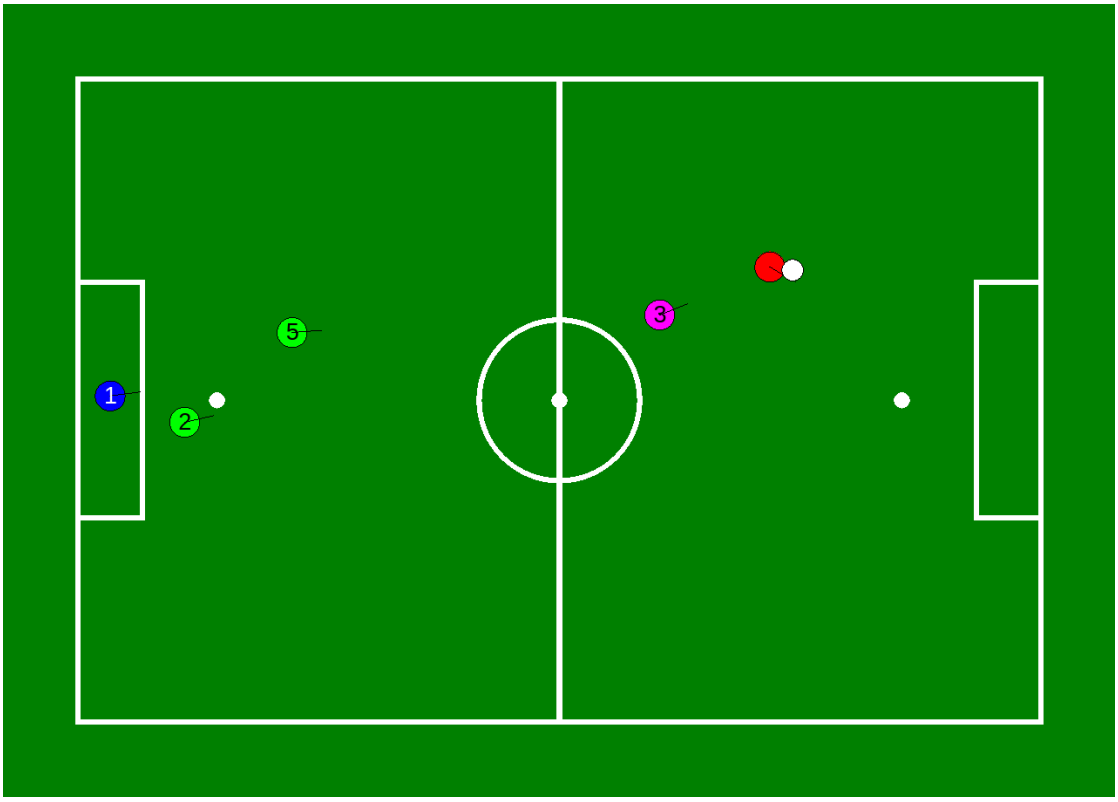


Figure 5.1: An aggressive situation. The colors indicate the role that is performed. Red is the striker, purple is the supporter, blue is the keeper, and green are the defenders. One defender stays behind the penalty spot, the other is more aggressive.

5.1.1 Role Provider

The general procedure of the role assignment is that each robot provides roles for the whole team. Each individual robot uses the role of the teammate with the lowest number that is not penalized. This approach is similar to that of B-Human (cf. [14, chapter 6.2.1]).

The roles are assigned as follows. At first the robot with the smallest estimated time to reach the ball is assigned to be the striker. Thus, there will always be a robot playing 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 the striker (cf. fig. 5.4). No striker is assigned during enemy free kicks. Second, the player with player number one becomes the keeper unless it already is the striker, 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. 5.5). After striker, keeper, and/or replacement keeper are considered there are zero to four robots left. The remaining robots are assigned–in order–to the

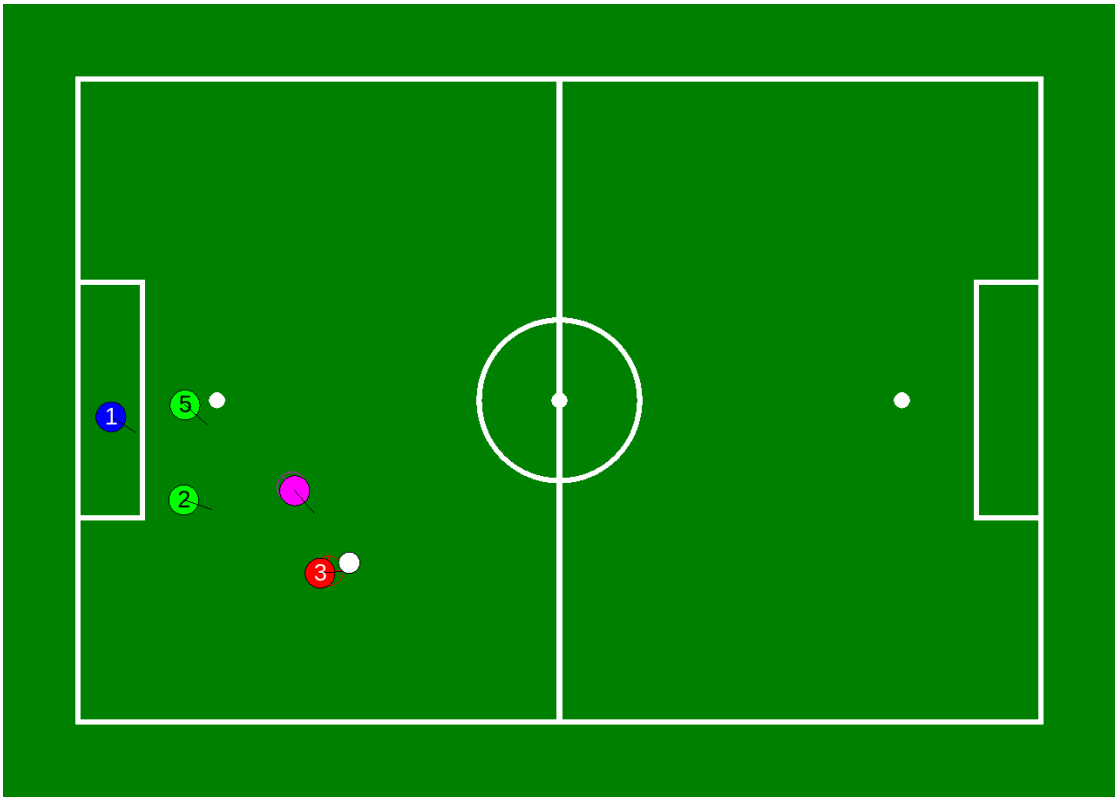


Figure 5.2: A defensive situation. Both defenders stay behind the penalty spot. Note that the supporter position is capped so that it does not interfere with the defenders.

following roles: defender, supporter/bishop, defender, and supporter/bishop, whichever has not been assigned, yet. 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 position should match the position that each robot is assigned to when the game state changes to *Playing* and roles are assigned.

5.1.3 Keeper

The keeper's responsibility is to avoid recieving 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

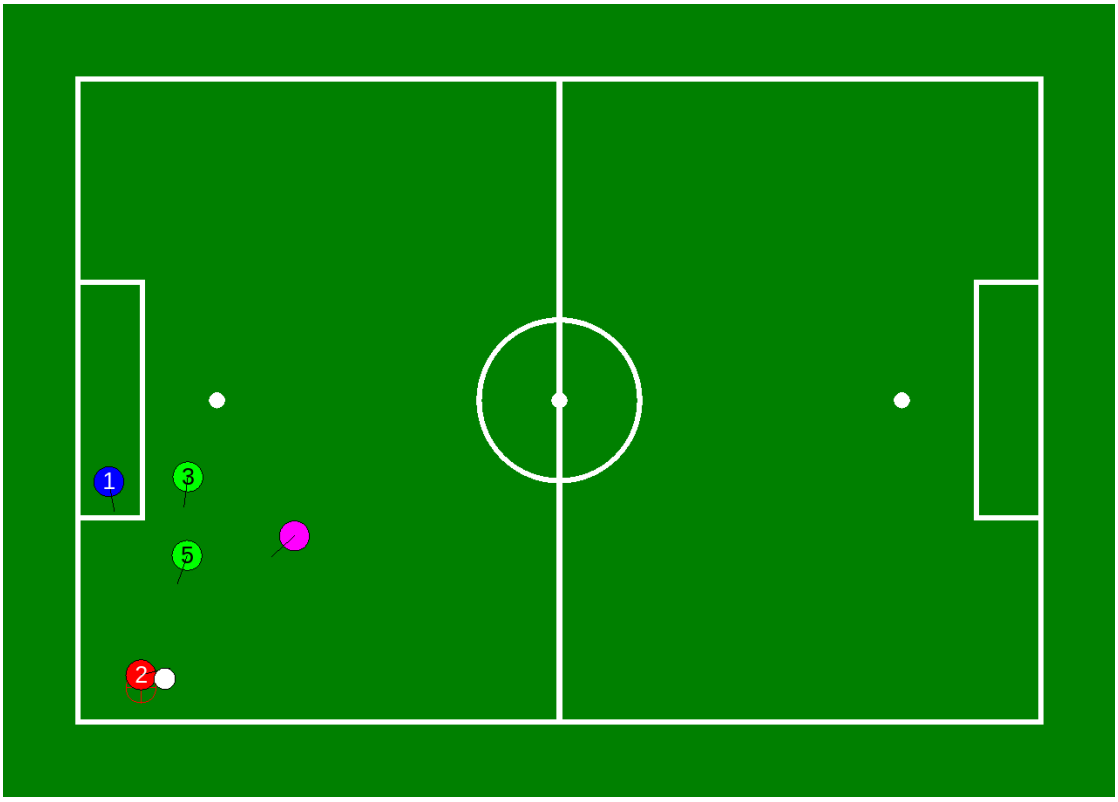


Figure 5.3: A corner situation. Note that the supporter position is capped so that it does not interfere with the defenders.

genuflect–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. Figure 5.1, fig. 5.2 and fig. 5.3 show the keeper–shown in blue–in an aggressive, defensive and corner game situation, respectively.

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.

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

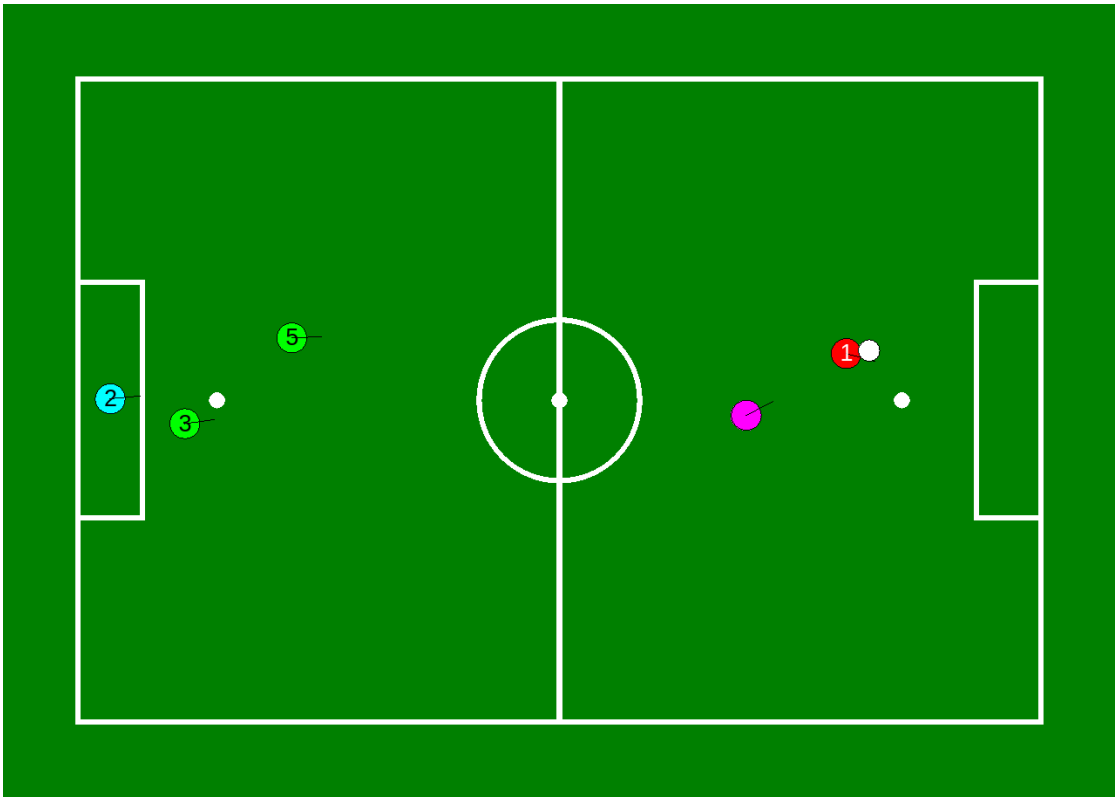


Figure 5.4: Player number one has become striker. In return, player number two took the role of replacement keeper. Defenders and supporter are assigned as usual.

blocked a sector around the desired direction is sampled in a lawn-sprinkler-like way. This is illustrated in fig. 5.7. The sector is discretized as 41 discrete directions with weights to model the kick deviation. 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 defenders assist the keeper in blocking as much of the own goal as possible. Their positions on the field 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 defenders stay right behind the penalty spot (cf. fig. 5.2). If the ball is far away one

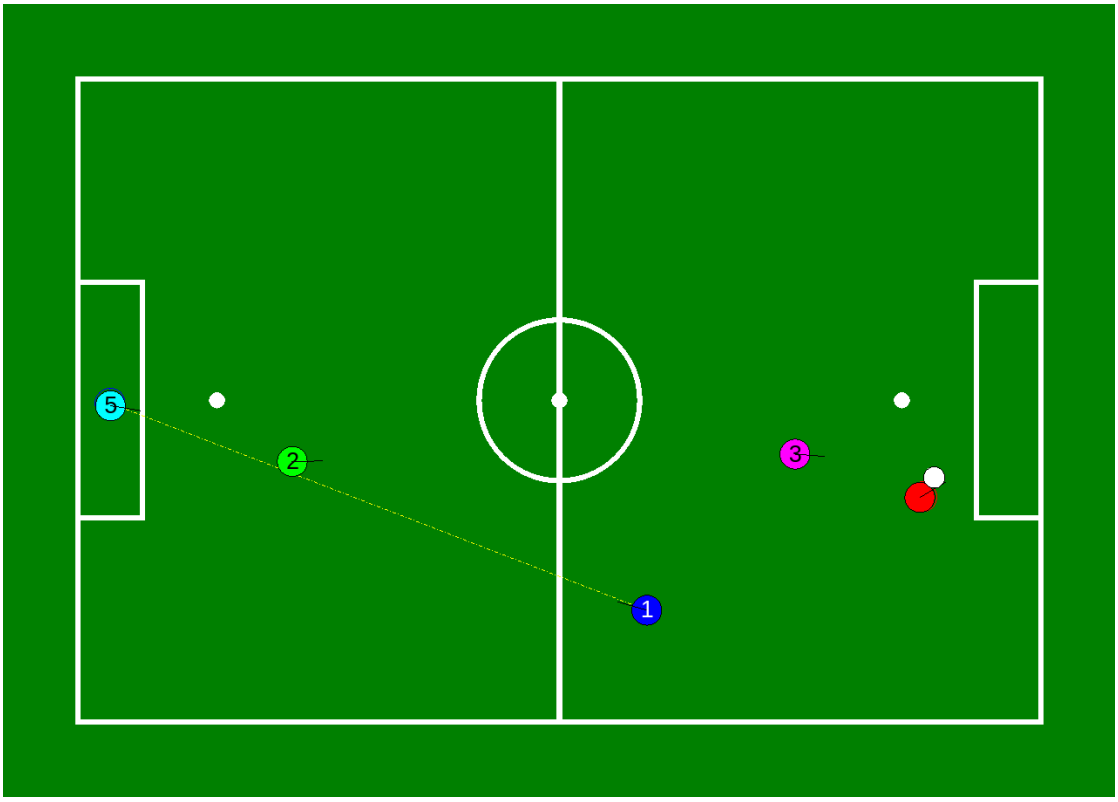


Figure 5.5: Keeper and replacement keeper exist simultaneously. The keeper is far away from the own goal, hence the replacement keeper guards it until the real keeper returns.

defender is more aggressive to cover a larger area of the field in case the ball is lost (cf. fig. 5.1). 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 defenders position to the left and right of that line. This alignment is illustrated in fig. 5.2. 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 (cf. fig. 5.3).

The defending behavior is unique in that there can be one or two defenders. If a defender is alone it generally behaves less aggressive.

5.1.6 Supporter

The supporter is designed to assist the striker by taking over if the striker loses the ball in a duel or if it falls down. To accomplish this the supporter 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. Figure 5.4 illustrats this trade-off. The supporter–shown in purple–does not stand directly on the line of sight.

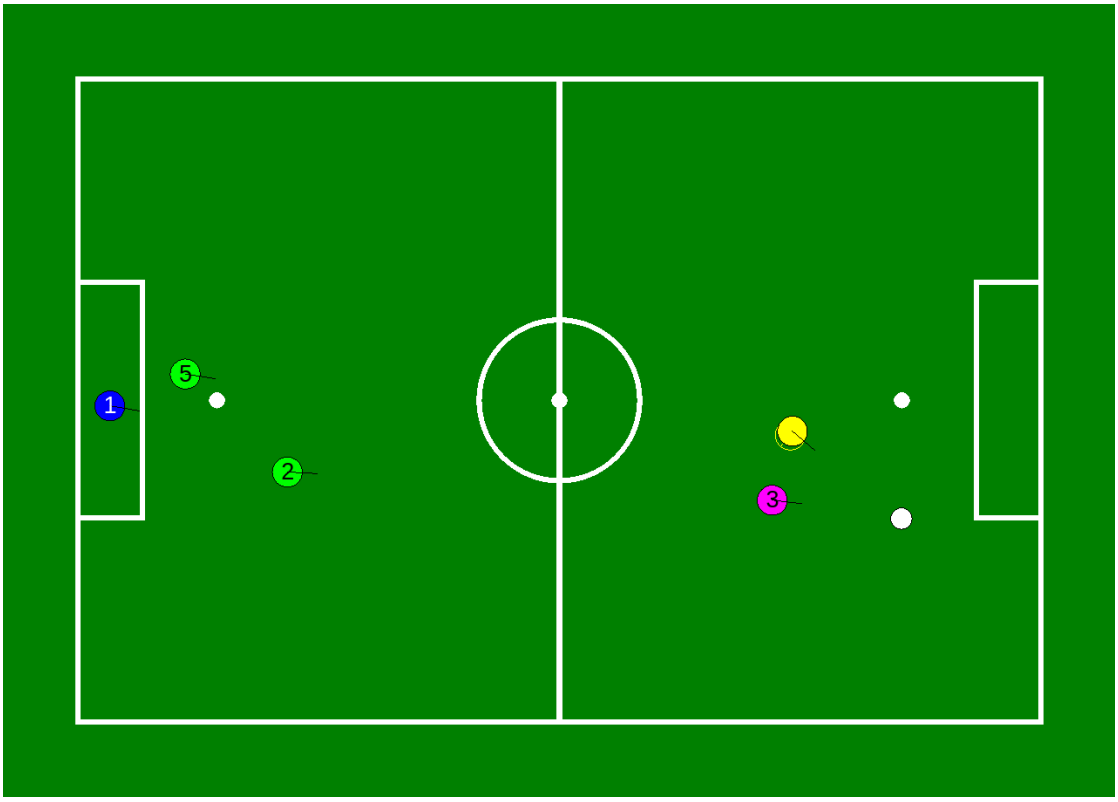


Figure 5.6: A goal free kick for the opposing team. No striker is assigned. Instead, a bishop–shown in yellow–emerges.

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. Figure 5.2 shows a defensive situation where the ball is close to the own goal. In order to avoid mutual obstruction of striker and its teammates the supporter 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 supporter. Figure 5.6 shows a bishop in defensive mode during an opponent's goal free kick.

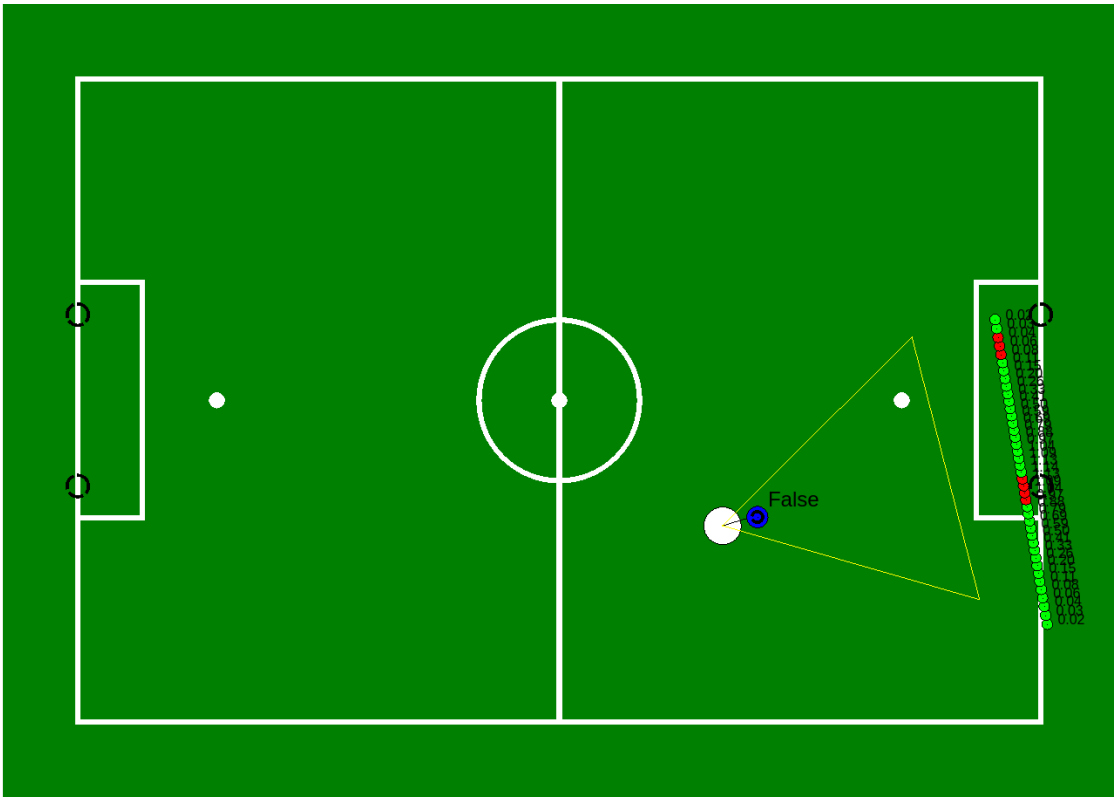


Figure 5.7: The lawn-sprinkler-like sampling of the kick deviation sector. The blue circle is the ball; the white cirle is the striker. The striker aims at the goal center. Green dots indicate lawn sprinkler directions that do not intersect with obstacles; red dots are obstructed by obstacles.

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 recieves a penalty for touching the ball with its hands [13]. To avoid accidentally touching the ball the replacement keeper must not use the genuflect 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.

Figure 5.1 and fig. 5.4 depict two similar game situations. The only difference is the fact that in the latter one player one is the striker. Consequently, player two replaces the missing keeper.

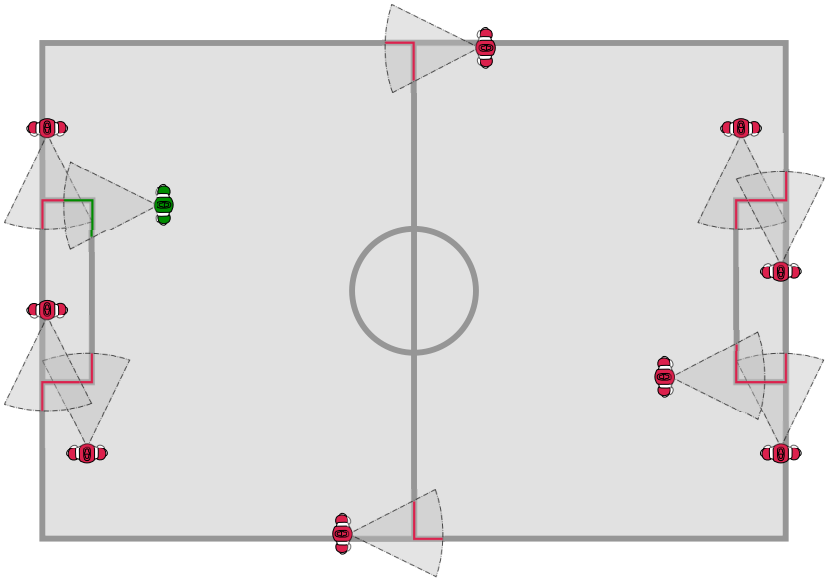


Figure 5.8: Ambiguity of measurements for feature based localization. True pose and perceptions (green) and other state that could create the same perceptions (magenta).

5.2 Localization

As of 2018 our code base features a new localization module based on an Unscented Kalman-Filter (UKF). Experience from previous seasons has shown that our former approach–based on a particle filter at it's core–was too computationally expensive. High runtime variance as well as partially bad estimation performance motivated the development of new method [12].

While a full discussion or 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 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 (see fig. 5.8). All visual features input to the localization are 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 6.10) and forward kinematics is used for prediction of the state evolution.

5.2.2 Algorithm

On the most 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 hypotheses is represented by its position and orientation (x, y, α) and can then be estimated using a separate UKF mode. Additionally, each hypotheses holds information about past measurement prediction errors.

Prediction At every cycle, we predict the state evolution of each hypotheses based on the odometry estimate. The new state estimate is computed from the last belief transformed by the estimated pose shift (see fig. 5.9). Since the predict is non-linear, we use the unscented transform to compute the predicted state distributions.

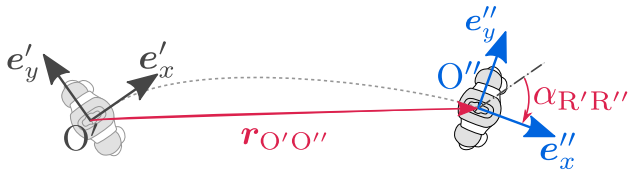


Figure 5.9: Predicting the state evolution using the odometry estimate.

Correction Measurements are used to correct the state estimate whenever the camera matrix is believed to be valid. We classify the validity of the camera matrix 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.

With every correction step in addition to the pose we also update the weighted error of predicted observations 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 it's 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 new approach clearly dominated 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 performace evaluation can be found in [12].

The improved estimation performance has enabled to stay well localization throughout most of the games. This allowed us to perform more complex team maneuvers and improved 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. While we modeled the ball dynamics as fully linear and conservative system in prior seasons, the new model was enhanced by also modeling viscose friction. Even this friction model only provides a very simplified picture of the highly non-linear ball dynamics, it has proven to be accurate enough for most of tasks. The resulting improvement of the 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 the team-mates. 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 team-mates 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 containing the local ball estimate is selected. In case the largest clusters do not contain the local estimate, the cluster with the most confident ball [1] is selected.

[1] 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 has proven to be quite robust against false positive ball detections of single robots. In particular, the ball-playing robot (i. e. the Striker) has a stable team ball, which is important for the general goal that, once a robot sees the (true) ball, it consequently plays it towards the direction of the opponent's goal.

The team ball model also integrates prior knowledge in certain game states if no ball is seen: In the *Set* state, the ball position is set to the center of the field (or the penalty spot in a penalty shootout), if it was otherwise unknown. 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 rules.

5.5 Ball Search

Whenever the ball is lost during a game there is the need of having a strategic team behavior to search for the ball. Our framework contains two modules that take care of this task. The first module is called `BallSearchMapManager`. 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 required by the second module called `BallSearchPositionProvider`. It calculates position suggestions for all active robots on the field. After it received these suggestions from every other player it determines which suggestion to trust most and calculates the desired walk target.

The following sections describe how these modules work. It should be noted that these two modules implement a ball search behavior that aims to search for a ball as fast as possible even if the ball detection is not that good. The behavior is not considered ideal when it comes to short term search.

5.5.1 BallSearchMapManager

This module is responsible for keeping track of all seen balls of every player that is neither penalized nor unsure about its self localization. Therefore it 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[2].
- 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 throw-in positions are 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. 5.10 (visualized using MATE (see section 7.2)).

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

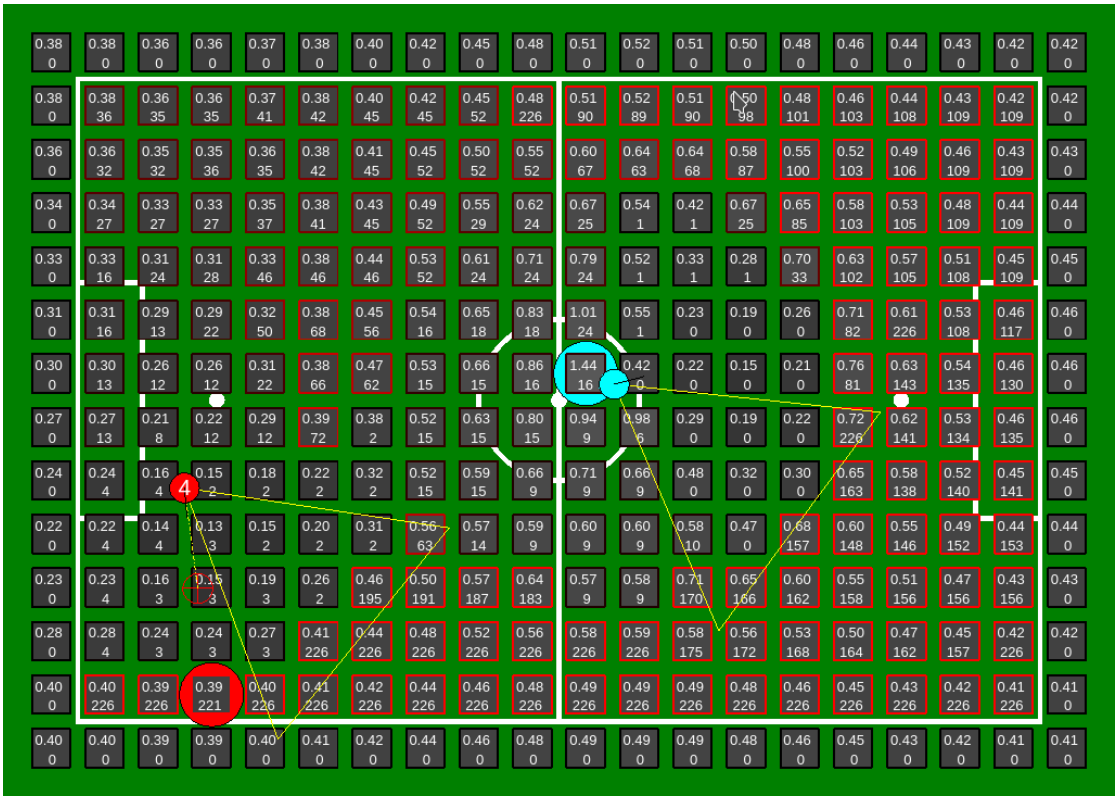


Figure 5.10: This is a visualization of the ball search map. The probability (upper value inside the cells) is represented by the brightness of the cells while the age (second value inside the cells) is represented by the red border. Also the players with their field of view and current target cell are visualized.

5.5.2 BallSearchPositionProvider

The `BallSearchPosition provider` is responsible for calculating search positions for every active robot on the field as well as agreeing on the `most wise player`[3]. This module mostly depends on the `BallSearch map` and the player positions.
The output of this module consists of the following information:

- An array of suggested positions for all robots that are available for this task.
- An array that flags all previously mentioned position suggestions as valid[4] or invalid.
- A flag that shows if this particular robot is ready to participate in the ball search. A robot that is ready is called `explorer`.
- 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. Reassigning only happens whenever a robot is dropped or added to the explorers.

Assigning search positions: After all robots are assigned to one search area they only need 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)).

[3] 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

[4] A suggestion is marked as invalid whenever the module decides to not let a player participate in searching for the ball.

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

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

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 these two ball search modules 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 a defender is added to the explorers.

5.6 Head Motion Behavior

The Head Motion Behavior is controlled by the Active Vision module. The idea behind this module is to decouple the head motion from the rest of the behavior. The Active Vision module provides a set of different modes that can be activated in the behavior. Based on the chosen mode the module will then independently decide how the head will behave. 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 preselected and significant field marks like the center circle or T-sections.

BallAndLocalization Works similar to the *Localization* mode but also takes the ball into account, so that, if possible, it is also included in the field of view.

LookForward The robot looks forward.

5.7 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. As of 2018, free kick areas in the case of a hostile free kick are the only obstacles of this type.

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 consequently to the map. While adding obstacles, we check for mergeability with obstacle 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 persisted. E.g., 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.

5.8 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-module 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 hand.

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.

A few changes were introduced in 2018 to improve motion planning. First, the robot now checks while dribbling if it is still aligned correctly with the ball towards the desired destination and repositions itself accordingly on the fly. Second, the way in which the ball is handled as an obstacle while circumventing it was improved to enable faster approaches. Additional minor tweaks in alignment and targeting calculations resulted in substantially faster and more robust dribbling with less unnecessary stopping. Future developments in this area aim to extend the motion planning additional modules for path planning in order to achieve a more sophisticated trajectory planning.

In the following, specific components of the motion planning will be explained in more detail.

5.8.1 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 destination 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.

5.8.2 Rotation

Depending on the requested walking mode, there are two ways in which the robot rotation is handled. The first option is that the robot tries to walk along a trajectory while maintaining a globally fixed orientation. The other option is that it will try to directly face the destination position until it approaches this position, where it then rotates gradually to the final orientation. The 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.

5.8.3 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. 5.11.

DRIBBLE is the most important walking mode for an attacking robot. It generally behaves like the *WALK_BEHIND_BALL* mode, but it switches to the *VELOCITY* mode once the ball waypoint was reached, after which 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.

5.9 Penalty Shootout

A new penalty striker behavior was developed and the penalty goalkeeper which was almost non-active was enhanced to react for incoming balls.

The penalty striker randomly selects a corner of the goal it will be shooting at. In addition, the enhanced motion planner (see section 5.8.3) was used to approach the ball slowly and safely, avoiding 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.

At RoboCup 2016 the standard keeper behavior was used for penalty shootouts. In advance of RoboCup 2017 new motion files were introduced that allow the robot to catch a ball rolling towards the goal.

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 to track the ball and thus can not predict the ball destination correctly. This leads to the

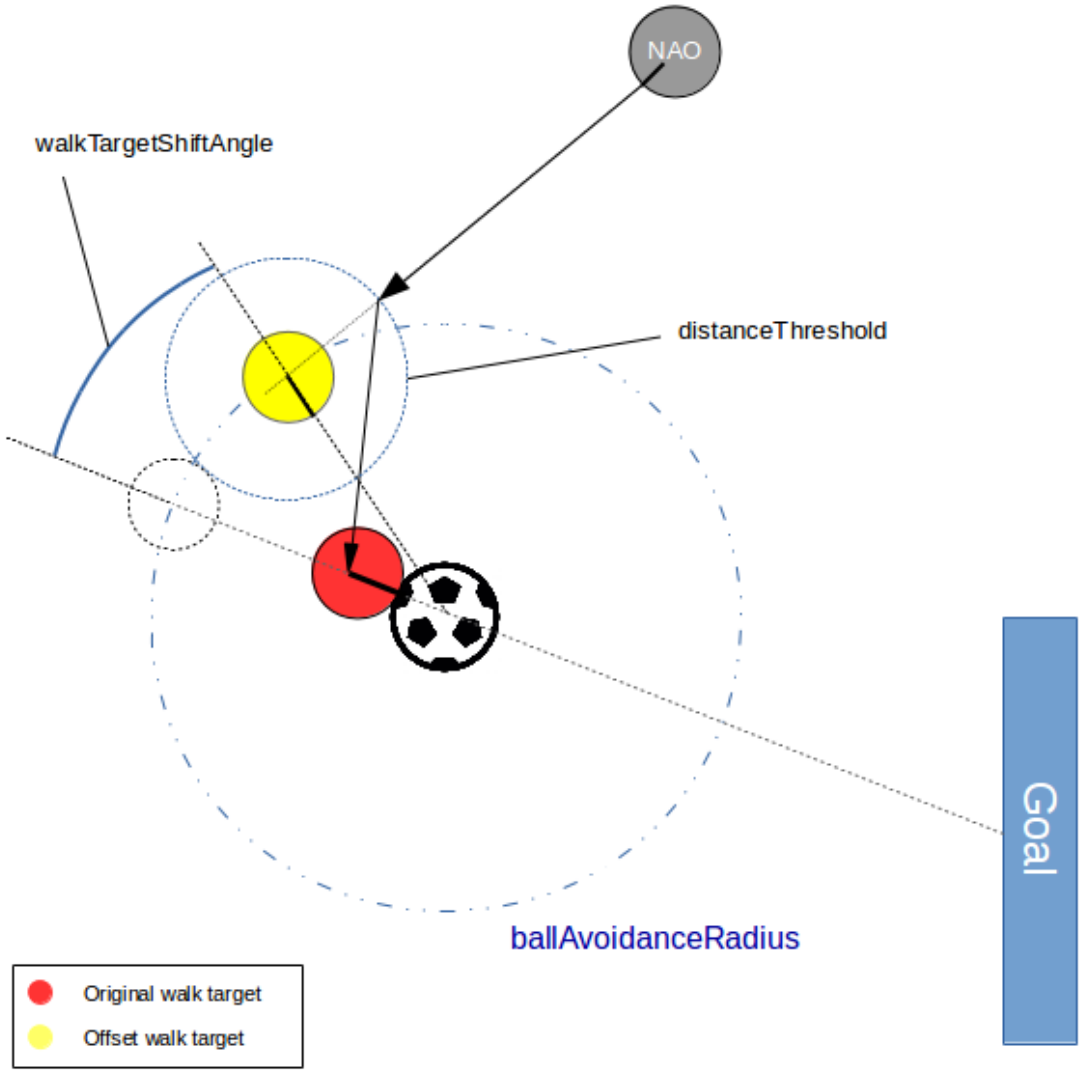


Figure 5.11: The WALK_BEHIND_BALL and DRIBBLE walking modes create a waypoint near the ball first (shown in yellow) by pulling away the original walking destination (shown in red) to the opposite direction of where the enemy goal is, and then rotating it towards the robot's current position. The robot's final trajectory is indicated by the black arrows.

problem, that the keeper does not react at all. To avoid this, some parameter changes in the ball filter are necessary. Since these motions to catch the ball are rather destructive for the robot, they are reserved for the penalty shootout.

Since last year, a penalty shootout challenge was introduced in our league to improve penalty shootout behavior.

5.10 Whistle Detection

Based on a bachelor thesis [5] a new whistle detection was implemented. The whistle detection features a dynamic detection of the whistle band in the frequency spectrum. 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 said to be found in spectrum. If a whistle is found in four consecutive spectra a whistle is considered to be detected and the game state is changed 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.

5.11 Foot Collision Detection

If we fail to detect an upcoming collision with sonar sensors (cf. section 6.9) 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 checks 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` (see section 6.4)—if at least one foot bumper is broken the module is not executed.

5.12 Free Kick Situations

In 2018 free kicks have been introduced in the RoboCup SPL [13]. Some of our modules (mainly the modules that provide actions for the playing roles) are actively reacting to free kick situations to comply with rules and use free kicks to out advantage. The modules and their reactions are listed below.

RoleProvider Removes the striker role assignment whenever the enemy performs a free kick. This way no robot tries to play the ball.

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 whenever a goal free kick is called.

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

ReplacementKeeperActionProvider Moves the replacement keeper away from it's position (if the ball too close) as he is not allowed to stay in the goal in this situation.

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

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 treat as a referee mistake thus we are not approaching the ball.

5.13 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 or shut down.

Chapter 6

Motion

This chapter describes how motions are executed in our framework. With one exception all motions are the result of a `MotionRequest` that is derived from an `ActionCommand` from the brain. The `MotionRequest` 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 send as commands to the joints via the Device Communication Manager (DCM).

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 6.11).

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 6.11.

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` 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. Playing motion files can be done with the `MotionFilePlayer`. It loads a motion file and interpolates between the specified frames while it is being played. Some basic motion such as standing up and keeper motions are realized with motion files.

6.6 Walking Engine

In 2017 we managed to improve our former walking engine to perform reasonable on the artificial turf. However, experience at competitions has shown that dexterity and maneuvering speed of this approach were insufficient to allow for competitive soccer play in the coming seasons. 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 [14]. We decided to port the `Walk2014Generator` of UNSW 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

Development of our own walking engine in previous seasons has brought forward several features and ideas that we sought to integrate to UNSW gait generator. Additionally, we applied new features to improve the robustness of the gait in tackling situations. Hereafter two of the most effective modifications are presented.

In-Walk-Kicks In order to move the ball faster and prevent the opponent to position at the ball moving the ball as quickly as possible as proven to be a good strategy. Based on this idea our team has developed kicks that can be performed in parallel to the normal gait already in 2017. In our former walking engine, these *In-Walk-Kicks* were implemented as superposition of the kick trajectory throughout a single step. *In-Walk-Kicks* were commanded by our behavior and were automatically performed if the ball was considered sufficiently close. While this approach provided reasonable performance with our old walking engine using small step sizes, this approach needed to be improved to also work with a more agile and fast-pace gait.

As of 2018 *In-Walk-Kicks* are requested by the behavior but only commanded to the walking engine through the `MotionPlanner`. This allows for better timing and special placement of such motion sequences. In order to allow for more controlled interaction with the ball *In-Walk-Kicks* now consist of two steps–a preparatory step and a kicking step. In the event of a straight front-kick, the preparatory step is used to place one foot next to the ball before striking with the other–a strategy that we observed to be successfully performed by other teams performing similar kicks in previous tournaments [7].

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 robots upper body forward and at the same time pulling the arms back closely to the robots waist. By this 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 robots projected footprint and therefore lowers the likelihood of the arms getting caught on 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 position 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 two kick types exist: a forward kick and a side kick. Both use the same interpolation scheme and only differ in their parameters. Thus, it is very easy to add new kick types or change existing ones.

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 in a way to compensate the moment about 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 Head Angle Limitation

In 2016 we had severe hardware issues with several NAOs. Some robots randomly lost stiffness in all joints, only recovering after a few moments. Further investigation on this issue have shown that such incidents were strongly correlated with dmesg events of reconnecting USB-devices as depicted below.

Listing 6.8.1 dmesg output after two incidents of spontaneous stiffness loss

```
[ 7470.750343] hub 3-0:1.0: port 2 disabled by hub (EMI?), re-enabling...
[ 7470.750357] usb 3-2: USB disconnect, address 8
[ 7470.956081] usb 3-2: new full speed USB device using uhci_hcd and
    address 9
[ 7471.500349] hub 3-0:1.0: port 2 disabled by hub (EMI?), re-enabling...
[ 7471.500365] usb 3-2: USB disconnect, address 9
[ 7471.707073] usb 3-2: new full speed USB device using uhci_hcd and
    address 10
```

The disconnecting USB device is the chest board, which is connected at the head mount of the NAO. Such *chestboard-disconnects* occurred particularly often if the head was moved far right or left while trying to apply a certain head pitch. Since these hardware issues couldn't be fixed by SoftBank Robotics, we addressed this problem by introducing a yaw dependent head pitch limit

$$\theta_{\max} = \theta_{\text{omax}} + \frac{1}{2}\left(\theta_{\text{imax}} - \theta_{\text{omax}}\right)\left[1 + \cos\frac{\pi}{\varphi_{\text{r}}\varphi_{\text{p}}}\right]. \tag{6.1}$$

θ_{omax}**:** The head pitch limit for the outer head-yaw range. Default: 11.5°.

θ_{imax}**:** The head pitch limit for the inner head-yaw range. Default: 20°.

φ_p**:** Yaw threshold, separating inner and outer head-yaw range. Default: 32.5°.

φ_r**:** The requested head yaw.

6.9 Sonar Filter

The sonar sensors allow for an estimation of the distance to near field objects. Since our code does not yet feature any vision module that allows for a reliable visual detection of obstacles like other robots, we use the sonar data to detect obstacles in the close vicinity of the torso. Since obstacle avoidance–key to complying with the pushing-rule—is solely based on the sonar data, filtering these measurement is indispensable.

Our sonar filter is a low-pass filter, 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 [16]. Below 0.2 m the sensor saturates, thus not being able to provide any reliable distance measurements. Therefore, we reject all measurements exceeding the aforementioned limits. Low-pass filtering the data helps dealing with the sensor noise. Additionally, 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. A high stability of the low pass filter, as described above, ensures the possibility to use sonar sensor data as a reliable source of obstacles in close range.

6.10 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. In 2015 work started on a sensor fusion module, utilizing measurements of the accelerometer and the gyroscope to provide a robust estimation of the torso orientation. A first version of this module was used at RoboCup 2016. While it performed reasonably well for most cases, estimating the body pose in the state space of Euler angles caused severe divergence in the case of a gimbal-lock. Therefore, a new approach based on [18] was implemented. This algorithm utilizes quaternions for internal state representation, thus evading the aforementioned singularity issues. The implementation was validated with a dataset provided by [9], originally recorded during UAV experiments.

The current implementation allows precise and robust estimation performance. Therefore it provides a reliable source of orientation for our self-localization and is one of the major reasons, why symmetric mislocalization has not occurred during RoboCup 2017 and 2018.

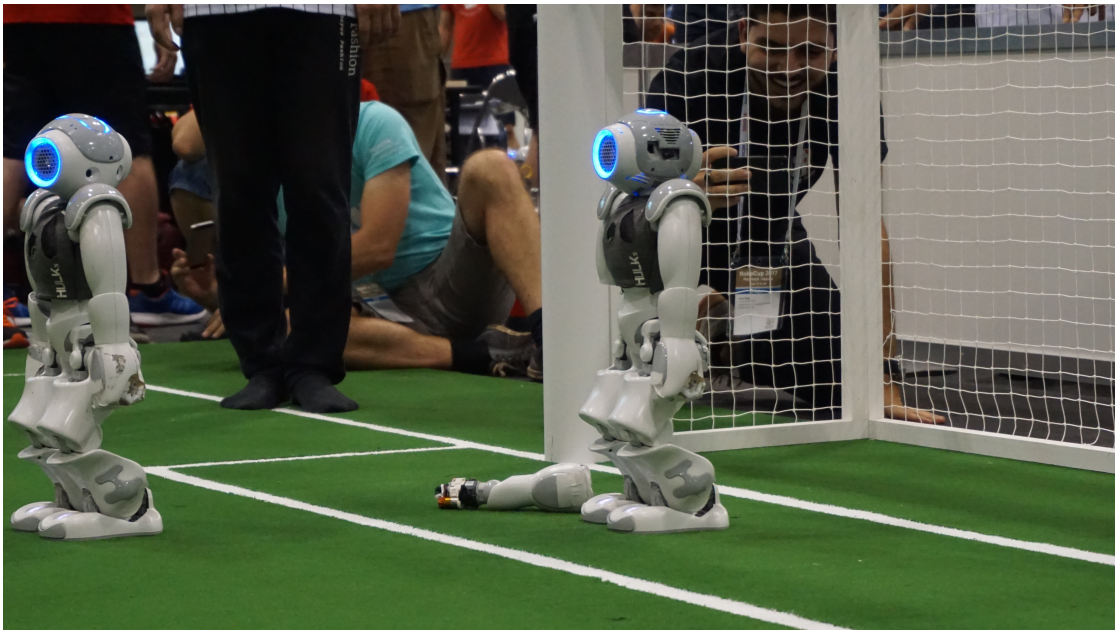


Figure 6.1: A robot that has lost its right forearm during a RoboCup 2017 game. That is of course no reason for a HULK to stop playing.

6.11 Fall Manager

The frequent observer of RoboCup SPL games will have noticed the unique crouch motion our robots execute when they are about to fall towards the front. This motion is performed by the fall manager. Similarly, a sit down motion is executed when a robot is about to fall to a side or to the back.

The crouch motion was originally designed to reduce hardware damage. We have found that it fails to accomplish this. Moreover, the fact that the crouch motion further accelerates a falling robot towards the ground leads to a more intense collision. The head might be protected from damage, but the hands, arms, and shoulders take serious damage. In fact, the crouch motion might have been responsible for the detachment of two arms during RoboCup 2017 (cf. fig. 6.1). Consequently, we have abolished the crouch motion to replace it with a different–and less arm-detaching—motion.

The new fall manager 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 collision. Neither head nor arms take damage.

6.12 Collision Prevention

With the incremental penalty times [13] accumulating fouls is a severe disadvantage. A collision can be detected with sonar sensors (cf. section 6.9) or foot bumpers (cf. section 5.11). To prevent pushing other robots the arms are taken on the back of the robot

if a collision is detected. To take the arms back the upper arms are pulled close to the body and the elbows are bent with the lower arms facing backwards. In particular, the arm swing during walking is switched off, which makes the walking somewhat unstable. To compensate this the center of mass is moved slightly forward.

Chapter 7

Tools

This chapter explains the different tooling we utilize both during development and in competition situations. During development, debug tools are necessary for visualizing data. During development, debug tools are needed for data processing and measuring CUP 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 games helps getting constant 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 *game controller controller* [1] during the game.

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. One team member branches off of the repositories master branch (called the *game branch*) and pushes it to a remote that is accessible to everyone. We then start calibrating cameras as well as finding the right vision and walking parameters which are directly pushed to the game branch.

40 minutes prior to the game, all parameters need to be pushed to the *game branch*. The one that created the game branch then starts setting up the robots (scripts called from repository root):

Listing 7.1.1

```
./scripts/changePlayerNumber 11:4 19:2 16:5 12:1 18:3
./scripts/pregame -n SPL_A 11 12 16 18 19
```

These two scripts 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 all active robots
- Clear all custom log files and replay data. This ensures that we do not run out of disk space.
- Restart `naoqi` and the hulks service
- Connect all robots to the network (e. g. `SPL_A`)

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 shoot 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.

[1] our naming proposal for the *game controller operator*

This very basic tests 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 `naoqi` and hulks services are executed correctly.

5 Minutes prior to a game all robots should be connected to the correct network. The person who deploys the code to the robots 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 script 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 one script:

Listing 7.1.2

```
./scripts/postgame -l LOG_DIR 11 12 16 18 19
```

The post-game script does the following:

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

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

7.2 MATE

In 2018 we have developed a new tool for visialization, 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 part (network communication and data management) and a front end part (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, views and windows and is located in the *ui/* directory. The *run.py* script starts a *QApplication* and a *QMainWindow*. All following widgets and views are dynamically created and shown in 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 view or element can subscribe one or more keys. The subscriber is identified with a custom string holding for example an 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 Views

All MATE-views are realized using *QDockWidgets* which enables dynamic arrangement and positioning of all views. There are six individual views implemented: Text, Image, Plot, Map, Config, CameraCalibration. A combination of various views is visualized in fig. 7.1. The *Text-view* visualizes incoming data in JSON-formatted text. The *Image-view* 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-view*. 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-view*. An example of plotting the first element of an incoming array is shown in 7.2.1.

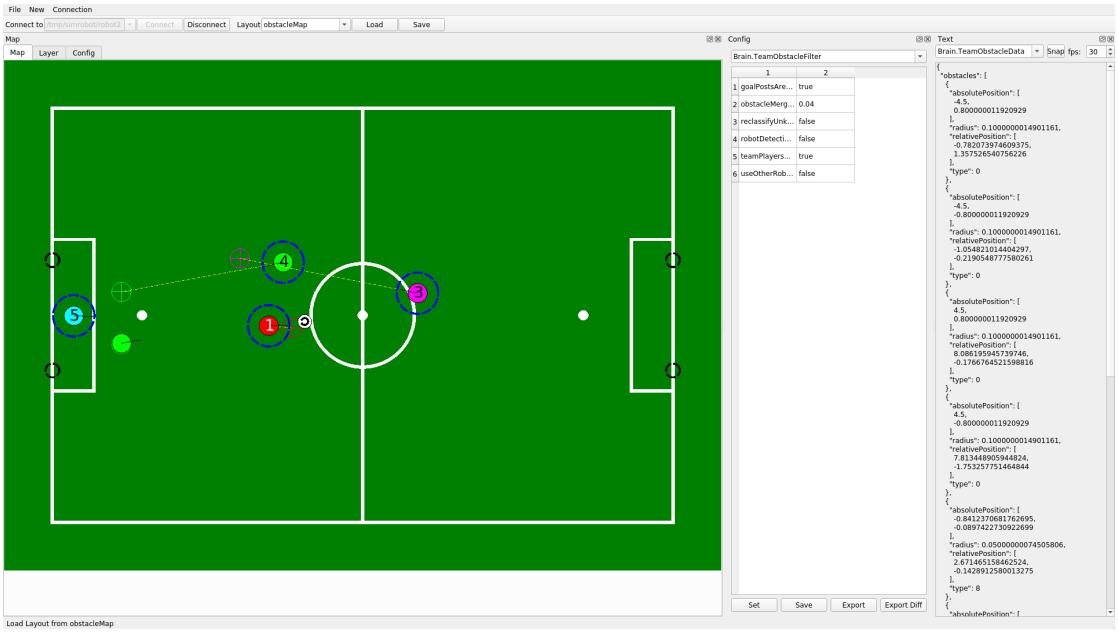


Figure 7.1: An example of combining a Map-view picturing the registered obstacles of a NAO with a config-view and a text-view.

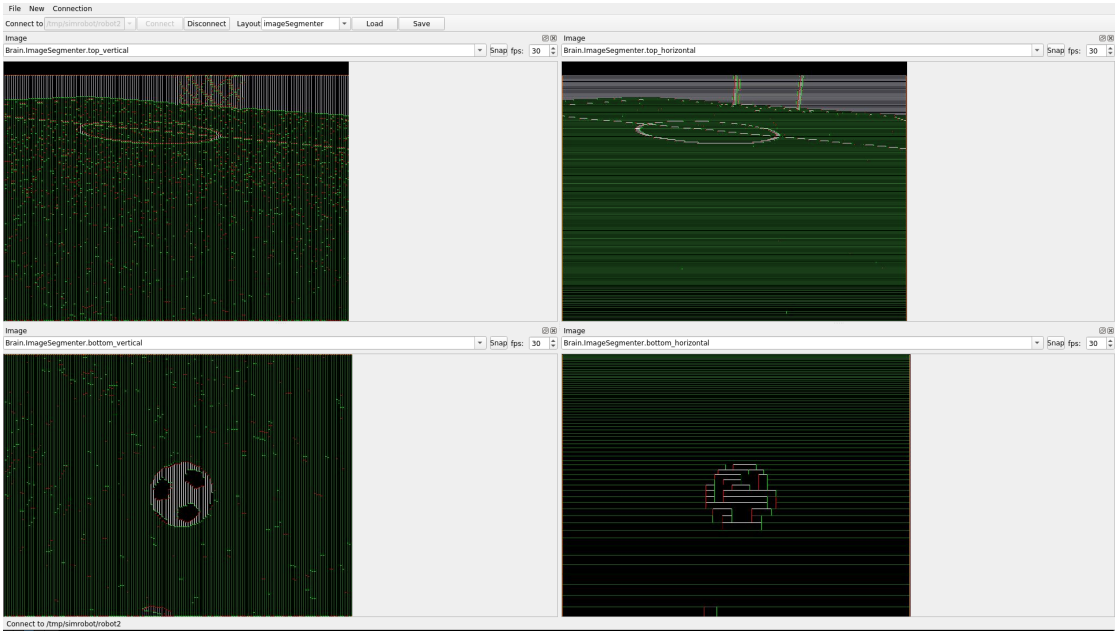


Figure 7.2: The image segmenter output in four MATE Image-views.

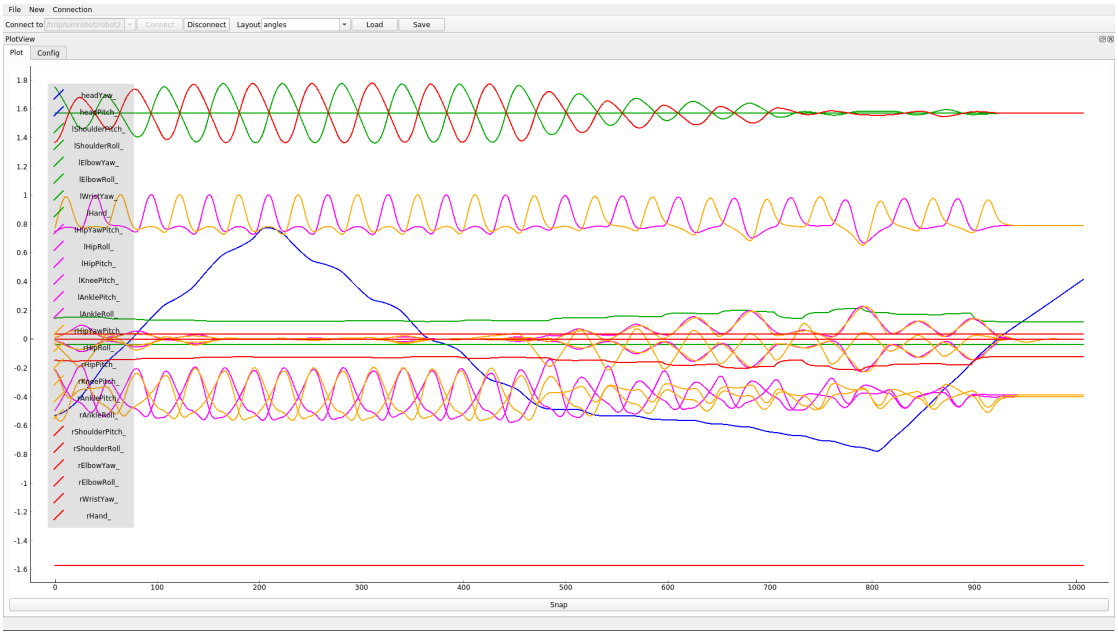


Figure 7.3: A Plot-view showing all joint angles.

Listing 7.2.1

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

7.2.4 Higher-Level Data Processing Using the Map View

The *Map-view* 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-view* with relevant layers has proven to be helpful. A *Map-view* for the ball search (cf. section 5.5) is shown in fig. 7.4; the layer configuration is shown in fig. 7.5.

7.3 Profiling

We prepared our code to work with `Intel VTune Amplifier`. It is a x86/x68_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.

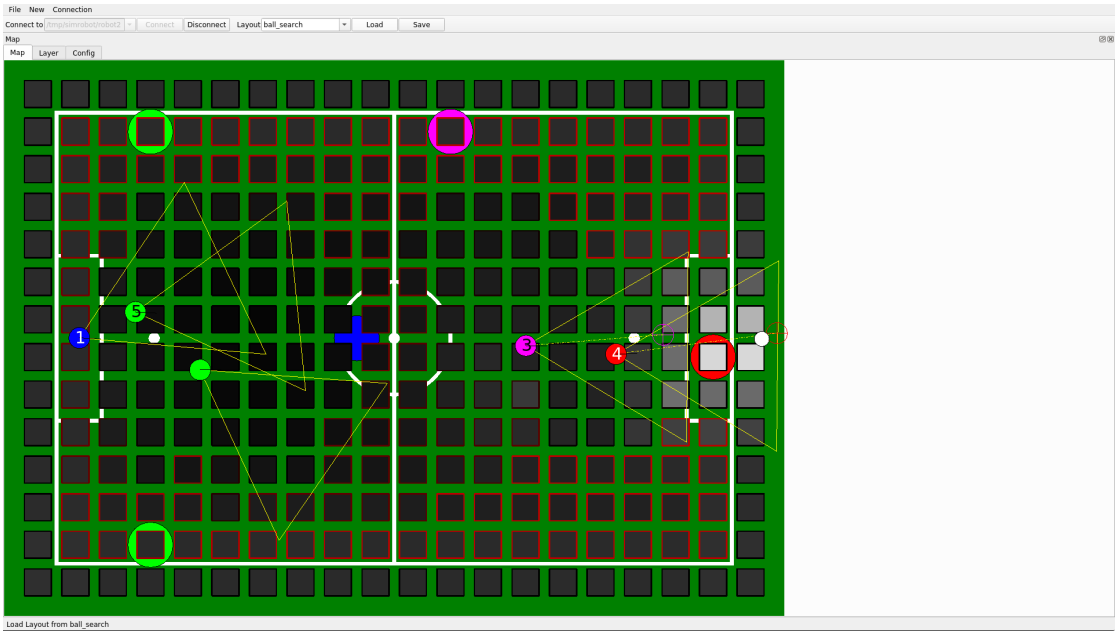


Figure 7.4: Using the Map-view to visualize the current ballsearch model.

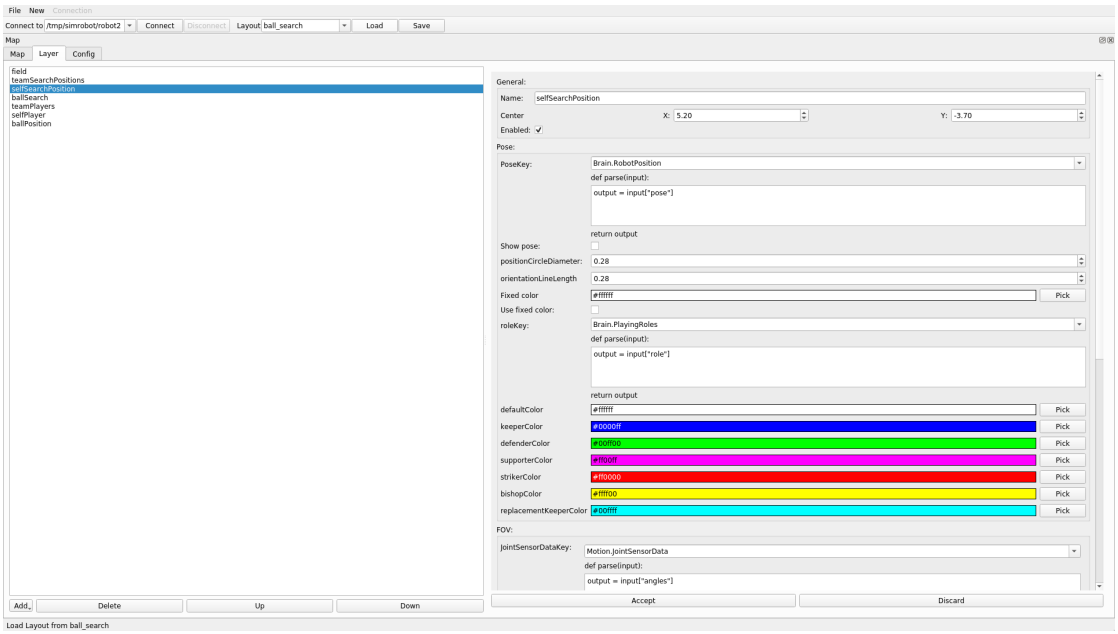


Figure 7.5: The layer configuration of a ball search visualization.

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
```

For better disk performance we use an usb drive to store temporary files during profiling. Mounting a drive to `/mnt/usb` is recommended.

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 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 -d -b Release <NAOIP>
./scripts/connect <NAONUMBER>
nao# sudo /etc/init.d/hulk stop
```

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:

Listing 7.3.4

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

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.

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 normal. 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 `dev-python/matplotlib` and `dev-python/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.

7.4 Debugging on a Robot

Sometimes it is necessary to debug program crashes or other strange behaviors of the robots. For debugging the `tuhhNao` process on the NAO, it is necessary to run it under `gdb`. The NAOqi OS only has a very old version of it installed. Our sysroot contains a newer usable version of `gdb`. Debugging `tuhhNao` with `gdb` can be achieved with 7.4.1

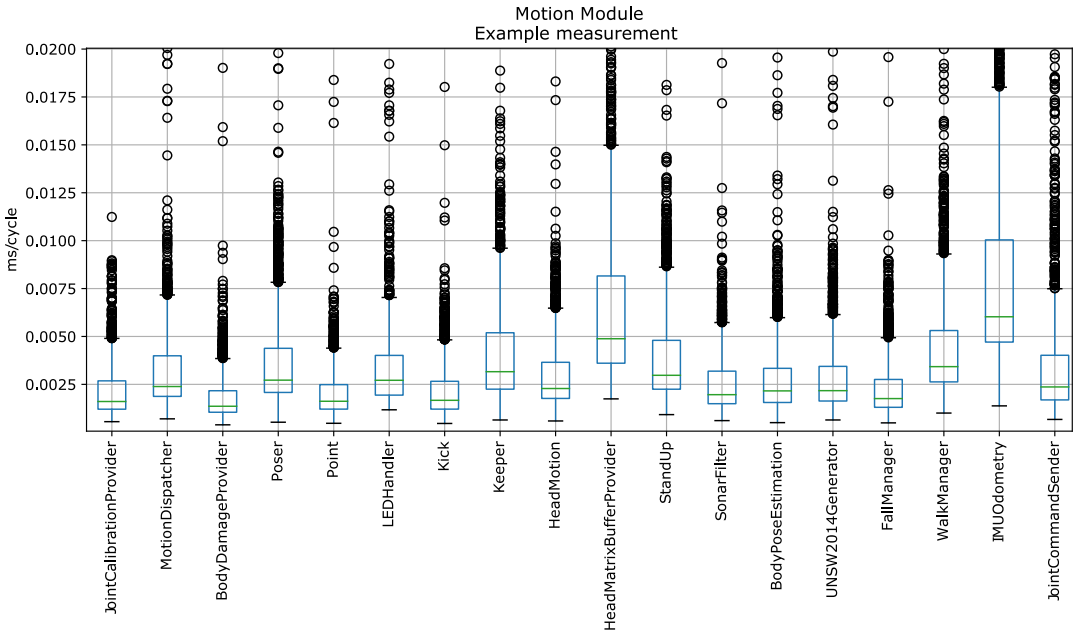


Figure 7.6: A box plot depicting the runtime of all Motion Modules. Black circles indicate outliers.

Listing 7.4.1

```
/home/nao/sysroot-7.3.0-1/usr/bin/gdb /home/nao/naoqi/bin/tuhhNao
```

Afterwards the process can be executed with `run`. 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` returns automatically to the 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 `tuhhNao` with `Debug` instead of `Release` target.

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

Bibliography

[1] Bouguet, J.Y.: Camera calibration toolbox for matlab (01 2010), http://www.vision.caltech.edu/bouguetj/calib_doc/

[2] Felbinger, G.: A Genetic Approach to Design Convolutional Neural Networks for the Purpose of a Ball Detection on the NAO Robotic System (2017), https://www.hulks.de/_files/PA_Georg-Felbinger.pdf

[3] Felbinger, G., Göttsch, P., Loth, P., Peters, L., Wege, F.: Designing Convolutional Neural Networks Using a Genetic Approach for Ball Detection. In: Proc. RoboCup 2018 Symposium (2018), to appear

[4] Garrido-Jurado, S., noz Salinas, R.M., Madrid-Cuevas, F., Marín-Jiménez, M.: Automatic generation and detection of highly reliable fiducial markers under occlusion. Pattern Recognition 47(6), 2280 – 2292 (2014), http://www.sciencedirect.com/science/article/pii/S0031320314000235

[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] Hofmann, M., Kerner, S., Schwarz, I., Tasse, S., Urbann, O.: Team Description for RoboCup 2011 (2016)

[8] Kahlefendt, C.: A Comparison and Evaluation of Neural Network-based Classification Approaches for the Purpose of a Robot Detection on the Nao Robotic System (2017), https://www.hulks.de/_files/PA_Chris-Kahlefendt.pdf

[9] Lee, G.H., Achtelik, M., Fraundorfer, F., Pollefeys, M., Siegwart, R.: A benchmarking tool for mav visual pose estimation. In: ICARCV. pp. 1541–1546 (2010)

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

[11] OpenCV contributors: Camera Calibration and 3D Reconstruction, https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html

[12] 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

[13] RoboCup Technical Committee: RoboCup Standard Platform League (NAO) Rule Book (2018), http://spl.robocup.org/wp-content/uploads/2018/04/SPL-Rules_small.pdf

[14] 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

[15] Schattschneider, T.: Shape-based ball detection in realtime on the NAO Robotic System (2015), https://www.hulks.de/_files/BA_Thomas-Schattschneider.pdf

[16] SoftBank Robotics: Sonars, http://doc.aldebaran.com/2-1/family/robots/sonar_robot.html

[17] SoftBank Robotics: Video Camera, http://doc.aldebaran.com/2-1/family/robots/video_robot.html

[18] 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)

[19] 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 November 04, 2018.