HULKs Team Research Report 2017

Darshana Adikari, Georg Felbinger, Arne Hasselbring, Yuria Konda,
René Kost, Pascal Loth, Lasse Peters, Nicolas Riebesel, Thomas Schattschneider,
Felix Warmuth, Felix Wege

RobotING@TUHH e. V.
Hamburg University of Technology, Schloßmühlendamm 30,
21073 Hamburg, Germany

Contents

Chapter 1

Introduction

The HULKs are the RoboCup SPL team of the Hamburg University of Technology. The team was formed in April 2013 and currently consists mostly of graduate students.

Members of the HULKs originate from various fields of studies. Therefore our research interests are widely spread among several disciplines, reaching from the design of our own framework to the development of dynamic motion control. During the season of 2017 we managed to significantly improve our performance. Resulting from this, we reached the quater finals of the RoboCup 2017. As part of the team B-HULKs we won the mixed team tournament together with B-Human. In 2017 we successfully accomplished our season goal by not causing any *global game stuck*. This was mainly due to improvements made in several areas regarding the knowledge and handling of the ball state. Namely, a new ball detector based on an artificial neural network, a team wide ball state estimator as well as a coordinated ball search were introduced.

This document serves as partial fulfillment of the pre-qualification requirements for RoboCup 2018. For this purpose, it is accompanied by a version of the code that has been used at RoboCup 2017.

Chapter 2

Running the Code Release on a Robot

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

Currently Linux is the only operating system officially supported by our code base. It is possible to build the code on Microsoft Windows but there might be some compatibility issues. The following packages are required to build the code for replay (see 2.4) on a Linux system:

> 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` for NAO compilation (see 2.4) 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 needed:

> 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

To download this year's code release one can simply clone our git repository from GitHub:

Listing 2.2.1

```
git clone https://github.com/HULKs/HULKsCodeRelease
```

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-6.3.0-2.tar.bz2`
- `sysroot-6.3.0-2.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. And the last target enables to feed a prepared dataset into the code to be able to deterministically test our code.

There are three different build types:

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

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.

2.4.1 SimRobot

Our repository comes with its own version of SimRobot which one need 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 to use with SimRobot by executing the following command:

Listing 2.4.3

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

To start SimRobot, simply run:

Listing 2.4.4

```
cd tools/SimRobot/build
./SimRobot ../Scenes/HULKs2017.ros2
```

2.4.2 Replay

To compile the code for replay (see 2.4):

Listing 2.4.5

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

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-6.3.0-2.tar.bz2` and `ctc-linux64-atom-2.1.4.13.zip`:

Listing 2.4.6

```
cd ~/naotoolchain
tar xf ctc-linux64-hulks-6.3.0-2.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
```

Next, 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

It may be noted that our team number has been replaced with placeholders in all scripts and configuration files comming with the code release. In order to deploy the code on a NAO, the following files need to be modified by replacing the place holders:

Listing 2.5.1

```
  scripts/files/net
  scripts/lib/numberToIP.sh
  scripts/gammaray (line 120, insert teamname here)
```

In order to setup a NAO, first place a symlink inside the toolchain directly pointing at the `sysroot-6.3.0-2.tar.bz2`. 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 later be used to identify a robot on the network.

Listing 2.5.2

```
ln -s sysroot-6.3.0-2.tar.bz2 toolchain/sysroot.tar.bz2
./scripts/gammaray -w -a <NAOIP> NAONUMBER
```

After the script successfully terminated, the NAO should restart itself and be ready to run the code. The robot's hostname changes to "naoNAONUMBER", 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 "tuhhnaoNAONUMBER" and restart the hulks-service.

2.7 Debug Tools

For debugging there is the OFA (One For All, see 3.5 for technical details) tool. It can be found in: `tools/ofa`. To start OFA run:

Listing 2.7.1

```
cd tools/ofa
./gradlew runOfa
```

After starting OFA, connect to it via your web browser at: `http://localhost:8000`. On the front page, connection to a NAO or to SimRobot can be chosen.

There is a plus button to add panels in the bottom right corner. For viewing a live image from the robot, add an image panel with the desired image key.

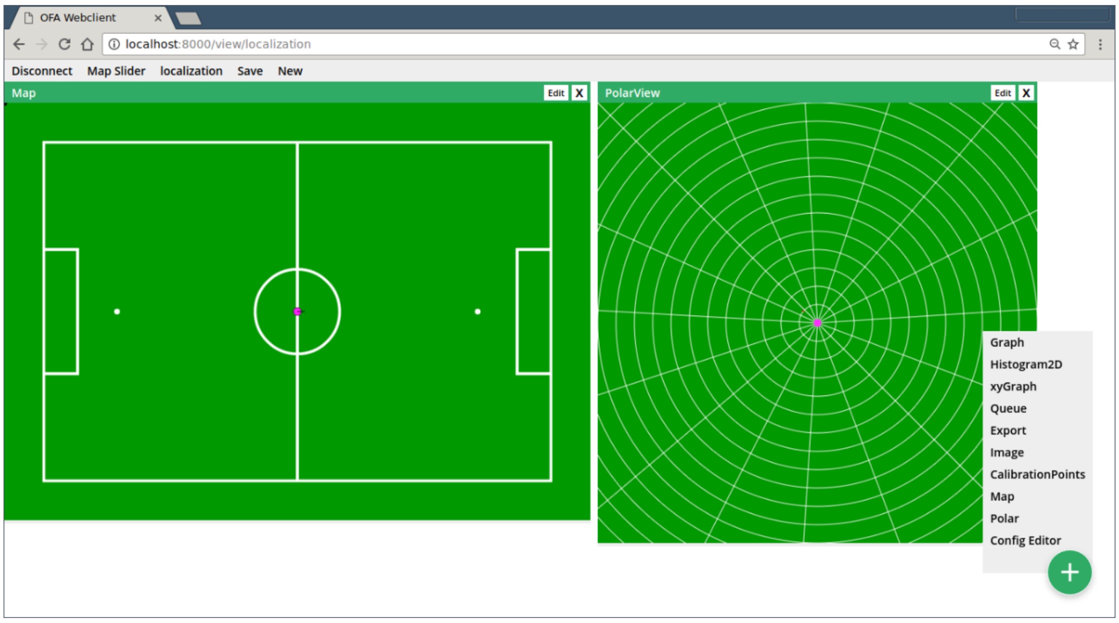

Figure 2.1: This is an example view of our debugging application running in Chrome.

There is also a feature to save and load arrangements of panels. They are called views. They can be saved and loaded in the top control bar.

A sample of the localization view can be seen in 2.1.

Chapter 3

Framework

3.1 Module Architecture

The largest part of the algorithms for robot control is organized as independent units called modules. The connection of these modules can be modeled as a bipartite data flow graph made of modules and the data types they exchange. Data types are stored in a database for each module manager[1]. 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 in 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 [7], 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.

3.2 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. 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.

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

The exchanged data types are determined by each module manager at program startup. All needed data types that are not produced in the 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.

3.3 SimRobot

SimRobot[2] 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.

3.4 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[3]. 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.

3.5 Debugging and Configuration

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 Node.js application called OFA (one for all) that connects to the NAO via a TCP socket and provides a web interface. You can see the application in action in figure 2.1. Data can be displayed in several types of views:

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

[3] Device Communication Manager

Graph views are plots of a numeric value over time. Multiple variables can be plotted color-coded in the same panel. The value range can be determined automatically and the number of samples can be adjusted.

Histogram2D is a heatmap of a two-dimensional histogram.

xyGraph displays a vector of two numbers as a point in a 2D coordinate system. As the vectors are buffered, their trajectory can be observed, too.

Queue can show textual information such as log prints.

Export creates CSV files with sequences of numbers that can be used for later analysis.

Image views display JPEG compressed images that are sent by the NAO.

CalibrationPoints is an extension of an image view in which it is possible to click points that can be sent to the robot. This is intended for camera calibration purposes.

Map is a view to display the pose of the robot on the field as well as particles of the particle filter including relative observations to the particles. Additionally the robot's target and the ball can be shown.

Polar views are used to display robot relative positions such as projected ball, line or goal post observations.

A specific setup of views can be stored and loaded as file to allow for fast access to often used view combinations.

Besides getting information from the NAO for debugging and evaluation purposes there are also several parameters that influence our software. We have a configuration system that loads parameters from JSON files when given the name of a class. It will look in multiple directories that are e. g. specific to a NAO head, NAO body or a location such as RoboCup 2017 SPL_A. Values that are given in more specific files will override values from generic files. It is also possible to change parameters at runtime via the aforementioned web based debug tool. Furthermore it is possible to map single parameters to sliders of a midi-controller. Receiving new values can cause a callback to be run so that precalculations based on these parameters can be redone.

Chapter 4

Brain

4.1 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. 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 and a bishop. Roughly, the roles have the following tasks:

1. Keeper: The goalkeeper.
2. Defender: Defensively positioned robots within own half.
3. Striker: The robot playing the ball towards the opponent goal.
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.

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.

4.1.1 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. [7, chapter 6.2.1]) which was advantageous for the Mixed Team Tournament.

The player with the number one is always assigned the keeper role. The striker is selected based on its estimated time to reach the ball. All remaining field players are assigned to the defender, supporter and bishop roles, depending on their positions and the number of them. The decision whether a bishop or a supporter is assigned is based on the position of the ball: If it is already far in the opponent's half, a supporter is more useful since no passes to the front are possible. On the other hand, if the ball is in the own half, it is likely that an upfield pass will happen.

4.1.2 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. As described in [8, section 4.3] there is special handling for mixed team games in which the regular position assgienment could lead to disadvantageous situations after kickoff.

4.2 Localization

For the purpose of self-localization we are using a particle filter as described in the last year's research report [5]. While this method provided sufficient accuracy and robustness to shoot our first goal at RoboCup 2016, its performance still led to frequent divergence of the pose estimation. Further investigation in 2017 has shown that these incidents were mostly caused by *false positives* in the line percepts. To improve the robustness of our localization strategy, we thus focused on measures to prefilter the input of the particle filter.

Analyzing the error characteristics has shown that most *false positives* originated from misperceptions near white-colored obstacles like goal posts and other robots. Furthermore, an accurate projection of the perceived lines often failed due to significant estimation errors of the camera pose. While working on these issues, the following measures have proven to be most effective:

Limited projection distance Lines that are projected over a large distance are dropped, as they are often inaccurately percepted. The distance threshold for this measurement rejection is chosen based on the currently performed body motion. Thus, we allow a higher projection distance if the robot is believed to be stable, e. g. if it is standing.

Modeling projection uncertainty The propagation of uncertainty of the projection was modeled approximately, thus reducing the influence of distant lines with higher uncertainty.

Modeling camera matrix validity When the angular velocity of the upper body is high, the camera matrix is assumed to be invalid. Thus, based on a metric calculated from gyroscope readings, all measurements for high angular velocities are rejected.

Tests at several events in 2017 have shown that these measures significantly improve the performance of the filter. Even though the aforementioned prefiltering leads to a large number of measurements being rejected, the overall filter performance increased. Additionally, an increased accuracy of odometry estimation was achieved by utilizing the IMU as described in 6.5, allowing us to compensate for the reduced number of measurements.

4.3 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 a 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.

4.4 Team Ball

The team ball is a combination of the local ball estimate and the communicated ball estimates of the teammates. 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 balls 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 conatain the local estimate, the cluster with the most confident ball [1] is selected.

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 goabesitztl.

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

[1] The confidence of a ball is assumed anti proportinal to its perception distance.

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.

4.5 Ball Search

At the end of 2016, our ball detection had low detection rates for balls that were located more than two meters away from the robot. In order to perform better in games, it was necessary to be able to keep track of the ball position even if it was too far away to be actively detected by the vision module itself. We therefore decided to create a module that is able to return a discrete probability distribution for the ball's position at any given moment, allowing the robot to relocate the ball after it went out of sight.

4.5.1 Ball Search Probability Map

We first introduced a module named `BallSearchProbabilityMap` (just *probMap* or *map* in the following) that generates a probability map on each robot. This map is a discrete probability distribution that is implemented as a matrix of probability cells (ProbCells or cells). 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:

$$\frac{1}{c+8}\begin{bmatrix}1 & 1 & 1\\ 1 & c & 1\\ 1 & 1 & 1\end{bmatrix} \tag{4.1}$$

  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.

[2]The field of view for the team members is calculated using their position and head yaw (reported via the SPL message).

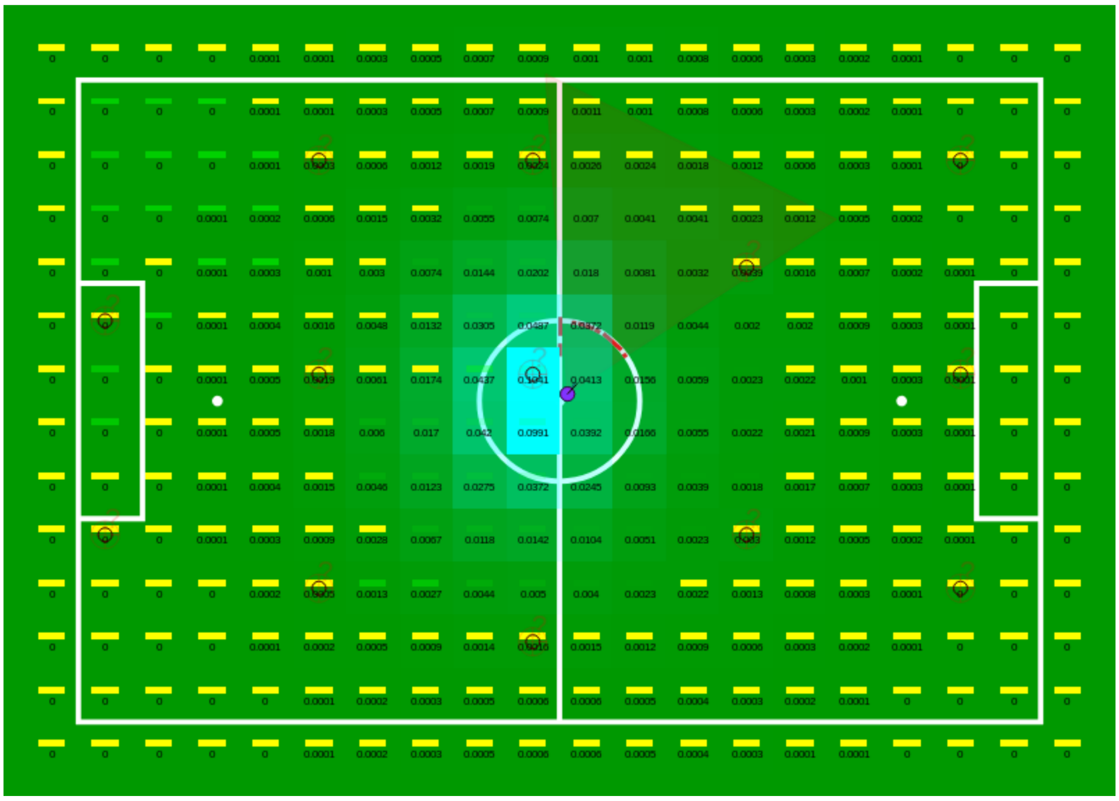
Figure 4.1: This is a visualization of the ball search map. The cell's weight is printed as a number to the map. A high weight is highlighted by blue color. The age is represented by the green to red colored bars. Also, there are markers that indicate the 'search position' candidates.

- 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.

4.5.2 Ball Search Position Provider

Secondly we introduced a so-called BallSearchPositionProvider. For every robot, it assigns positions on the field to explore. If the ball is completely lost (meaning that no ball has been seen for t seconds), all robots will retrieve the most probable locations for a ball from the map. These locations are then assigned to all active team players and shared via a SPL message, making sure that every robot gets its own position to search.

The locations sent by the player with the lowest number will be accepted as the search positions for everyone. Each robot will then go to its assigned position.

By doing this, the map will be automatically updated by the movement of all robots. New search positions are assigned if a robot's position was successfully explored or if another position became significantly more important. This behavior will (nearly) always lead to the detection of the ball since all areas of the field will be explored after some time. Since a probability distribution forms the foundation for the ball search, it usually doesn't take long to find the ball again.

4.5.3 Final words

The introduction of this ball search module 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.
- All robots should accept the search locations from the "oldest" robot (meaning the robot that was not penalized for the longest period of time, or a similar metric), instead of the player with the lowest number.
- The map is not initialized with a ball in the middle of the field.
- The map assumes that a ball can not leave the field.

4.6 Head Motion Behavior

In previous years the logic of head motion selection was solely based on the binary state of a ball being found or not. If the ball state was believed to be known, the robot would track the ball. Otherwise a periodic motion of the head yaw was performed to further explore the vicinity of the robot. While this primitive head motion logic was adequate for a ball model that is only based on local measurements, it can cause undesirable behavior for ball estimates extracted from the global team ball model. In many cases, tracking the ball is not sensible, if feasible at all. Due to this fact the head motion behavior was extended to also take into account the workspace of the head, the occlusion caused by the shoulders, as well as the limited visual range. By this means, a *look around* motion is triggered, if during ball tracking the ball is believed not to be visible anyway. Implementing this strategy allowed the robots to explore their surroundings more thoroughly, not being forced to track the — now well perceived — ball position provided by a team mate. Introducing these changes to the head motion behavior thus indirectly improved the accuracy of the self-localization, since more field features could be perceived. Future work will focus on further improving the head motion behavior towards the idea of active vision. For example, it would be desirable to chose the line of sight based on a cost function that models the feature density accessible for each potential set of head angles.

4.7 Motion Planning

Motion planning is responsible for determining the trajectory of future robot poses — namely its translation and rotiation — in order to execute more abstract requests provided by the behavior modules. It aims to create a desirable reference trajectory that moves the robot toward a specified destination while avoiding obstacles. A new *MotionPlanner*-module was developed in the season of 2017 to support multiple modes for walking on the one hand, as well as allowing to walk at a specific speed on the other hand.
The reworked motion planning resulted in vastly improved robot behavior especially around the ball, as the robot now more carefully tries to avoid ball collisions while circumventing it. Another new feature is the ability for the robot to dribble the ball aggressively, which improved the team's offensive skills significantly.
As a result of the new motion planning structure, future developments in this area have been enabled. This means that the current motion planning pipeline can now be extended with 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.

4.7.1 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.

4.7.2 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.

4.7.3 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 way-point position is constructed as shown in figure 4.2.

DRIBBLE 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.

4.8 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 4.7.3 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

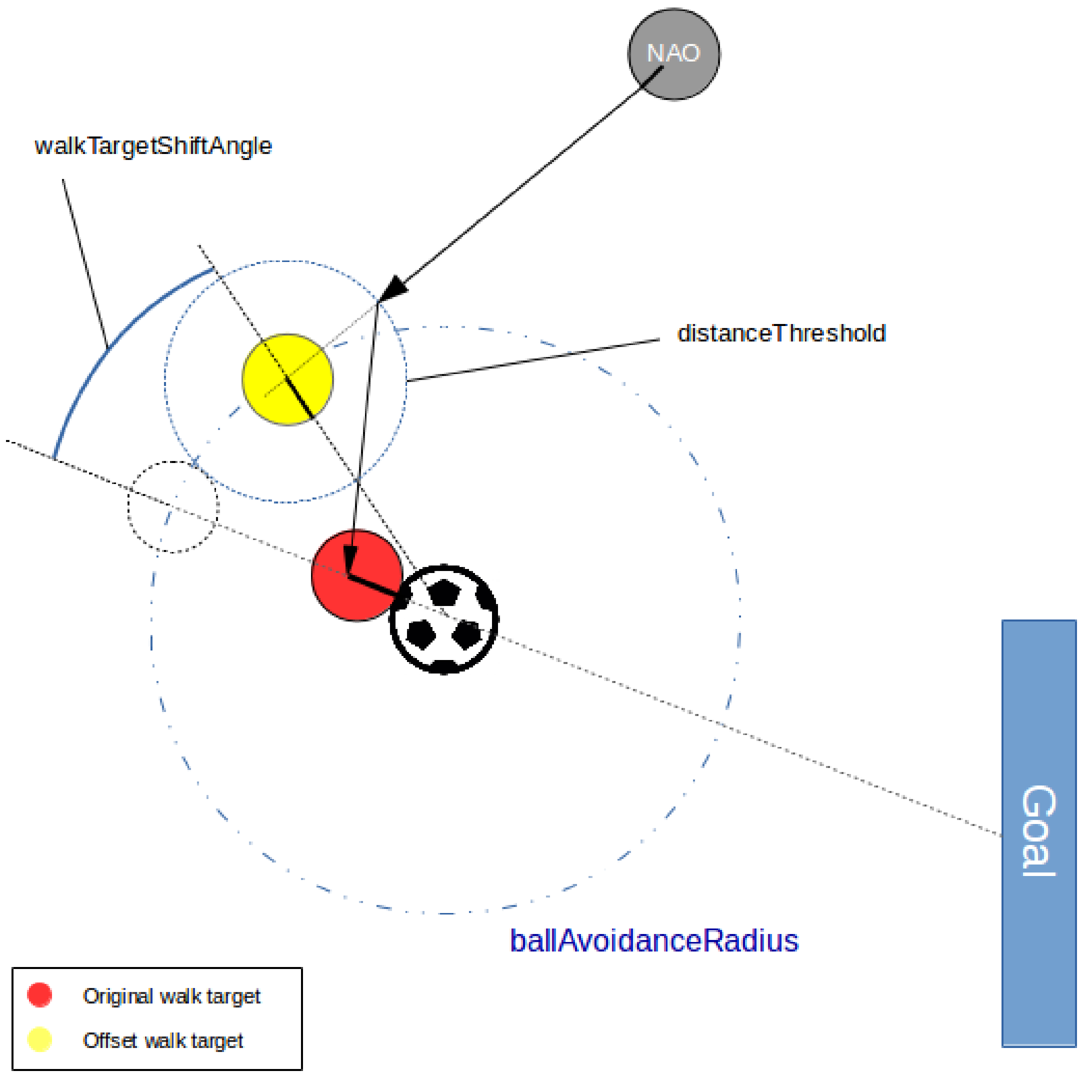

Figure 4.2: The WALK_BEHIND_BALL walking mode creates a waypoint near the ball first (shown in yellow) by pulling away the original walking destination (shown in red) to the opposite direction of where the enemy goal is, and then rotating it towards the robot's current position. The robot's final trajectory is indicated by the black arrows.

right or left or sits down based on the predicted ball destination. Since these motions are rather destructive for the robot, they are reserved for the penalty shootout.

The improved penalty goalkeeper was of particular importance for 2017's penalty shootout challenge. Additionally, these improvements contributed in winning the final game of the mixed team challenge as B-HULKs.

Chapter 5

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 and segmentation 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, i. e. the area where the carpet ends. The only objects that are currently detected are the white lines on the field and the ball.

5.1 Camera Calibration

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

While intrinsic calibration is needed less frequently, extrinsic calibration 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.

5.1.1 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, $\mathrm{R}_{\mathrm{ext}}(\alpha, \beta, \gamma)$. The transformation described by *camera2HeadUncalib* matrix is different between the two cameras. The matrices are supplied by SoftBank Robotics where further details are also available [11]. The formation of kinematic matrices from a ground point to the camera is as follows.

$$\begin{aligned}
\text{camera2Head}(\alpha,\beta,\gamma) &= \text{camera2HeadUncalib} \times \mathrm{R}_{\text{ext}}(\alpha,\beta,\gamma) && (5.1)\\
\text{camera2Ground}(\mathrm{t},\alpha,\beta,\gamma) &= \text{torso2Ground}(\mathrm{t}) \\
&\quad \times \text{head2Torso}(\mathrm{t}) \\
&\quad \times \text{camera2Head}(\alpha,\beta,\gamma) && (5.2)\\
\text{ground2Camera}(\mathrm{t},\alpha,\beta,\gamma) &= \text{camera2Ground}(\mathrm{t},\alpha,\beta,\gamma)^{-1} && (5.3)
\end{aligned}$$

The matrices containing parameter t indicate their value may change over time (ie: 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 3.4 (which runs vision module) 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.

5.1.2 Extrinsic Calibration

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 5.4 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, γ. The existing calibration values are supplied as the initial guess to converge faster and to reduce risk of stopping at a local minimum.

$$\mathit{Residual} = \text{ProjectedPoints}(\alpha,\beta,\gamma) - \mathit{DetectedPoints} \qquad (5.4)$$

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 5.1. The user is able to identify any potential problems and re-try calibration if necessary.

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

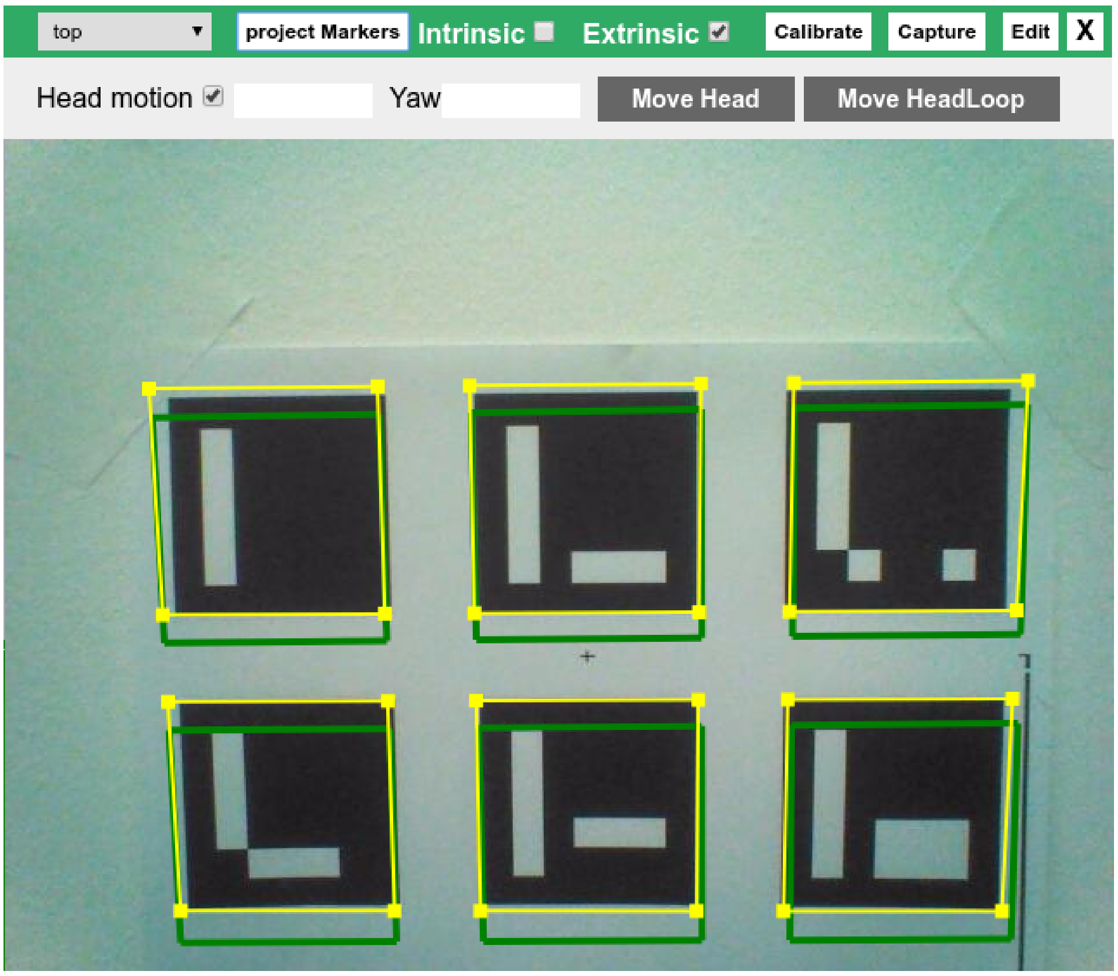

Figure 5.1: Calibration tool view in OFA 2.7 after completing calibration. The green and yellow squares correspond to the projection of markers before and after calibration process respectively.

5.1.3 Intrinsic Calibration

The first three steps of extrinsic calibration are performed first. Then, pose estimations of AR markers are used and the marker points (corners) get projected into the image plane. Then the problem of minimizing errors between projected and detected points were solved using the same Levenberg-Marquardt solver.

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.

5.1.4 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:

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 5.1.
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).

In addition to the above verification processes, unit testing was introduced to detect errors in mathematical calculations of the JavaScript libraries that were implemented by ourselves (due to unavailability of well performing non-linear solvers). Once the verification of the calibration 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.

5.1.5 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. Some outliers were observed where the captured images and kinematic matrices were time desynchronized in general. Motion blur and other communication issues also contributed.

- 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 not better than the traditional method and sometimes the results were erroneous. Possible factors were availability of feature points, quality of pose estimation. However this was sufficient for emergency calibration in event such as replacement or change of a camera during competitions.
- Automated extrinsic calibration of both cameras generally took less than 2 minutes. Additional time may be needed for secondary verification.
- Automated intrinsic calibration also took less than 2 minutes.

This automated camera calibration process was employed starting from Iran Open 2017, German Open 2017 and RoboCup 2017. The main goal of performing extrinsic calibration of all robots prior to each game was made possible due to the automation resulting up to twenty fold speedup. This contributed for better performance of robots during every game and workload reduction for team members.

Future work will focus on performing automatic calibration including the joints and eliminate the need for a relatively cumbersome mechanical fixture. It was also decided to switch to Python for the next iteration and implement the entire process on the NAO.

5.2 Robot Projection

It often happens that body parts appear in the image, such as shoulders or knees. 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.

5.3 Field Color Detection

For determining 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. Within each step, the mean of the cluster is shifted towards the mean of the corresponding colors sampled. A sampled color is part of the cluster if the distance in the Cb-Cr plane is lower than a parameterized threshold and the Y value is lower than a parameterized multiple of the cluster's mean Y. In order to avoid huge jumps of the cluster, e. g. when the robot is facing a wall or another very nearby robot, the shift of the cluster mean is limited, and remains unchanged in these cases.

5.4 Image Segmentation

The image is segmented only along equidistant vertical scanlines. A one-dimensional edge detection algorithm determines 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.

5.5 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, i. e. not belonging 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.

5.6 Line Detection

The line detection takes image regions that start with a rising edge and end with a falling edge and where the gradients of both edges point towards each other. For each of those regions its middle point is added to a set of line points.

Through these points up to five RANSAC lines are fitted. Each line is checked for large gaps between line points and split apart. If a resulting line segment contains too few points it is discarded.

5.7 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

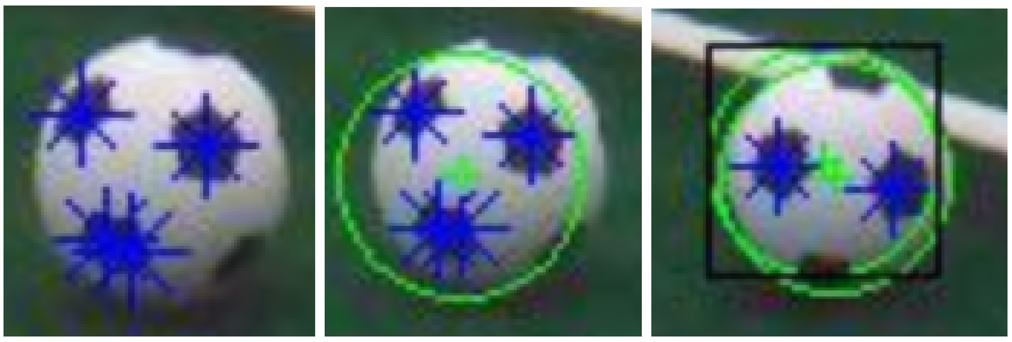
(a) (b) (c)

Figure 5.2: Visualization of ball candidates [2, pp. 17–18]. 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).

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 [9], 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 [3]. 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 many hyperparameters for the structural setup of the networks. This method has been developed in a project thesis [2] and further details can be found there.

5.7.1 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 Figure 5.2.

5.7.2 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 6

Motion

6.1 Walking Engine

The Walking Engine generates the joint angles, that assemble the bipedal gait. The basic algorithm used at RoboCup 2017 is identical to the one of previous years [5]. However, several parts of the walking engine have been reworked, in order to increase robustness and functional scope. Hereafter the key changes are presented.

6.1.1 Ankle Controller

Our team is using a model based control strategy to generate a bipedal gait for the NAO. While this strategy worked fairly well for the old carpet used at previous RoboCups, the 2016 version of our algorithm was not capable of ensuring stability on the new, artificial turf. The main effect causing instability arose from the excitation of oscillatory motions of the upper body. The measures presented in last years team research report [5] were no longer sufficient to suppress harmful amplitudes of this vibration. In order to regain robustness, we decided to augment our control strategy by an additional proportional controller applied to the ankle pitch. This ankle controller utilizes gyro feedback as described in [1] in order to damp undesirable angular vibration. As a result, we have managed to slightly increase the robustness of our walking engine, despite the more challenging conditions arising from the new carpet.

6.1.2 Transitions

In the previous years we used a static stand pose, which was generated according to a configuration file. However, having a stand pose different from the desired initial pose at walk start time, requires additional interpolation between those two poses. In order to reduce the time needed for transition from standing to walking, we decided to generate the stand pose directly from the walking engine. By means the position of the center of mass during standing matches the desired initial position at walk start time, and thus dynamically adapts to different walking parameters. Furthermore, this slight change also allowed to get rid of adverse transient effects, induced by the previous transitions phase.

6.1.3 In-Walk-Kick

Competitive interaction with the ball requires a short reaction time. Thus it is desirable to reduce the time needed to execute a kick. In order to come at this issue, we enhanced the walking engine by adding the option to perform an in-walk-kick. Such a kick is generated by superposition of the swinging foot trajectory with an additional kicking component. Such kicking trajectories are designed not to change the original target of the step. Therefore the induced offset declines to zero at both beginning and end of each step. Effective timing of such in-walk-kicks currently is problem, that is not solved to our full content. Future work will thus focus on incorporating an in-walk-kick request also in the planning of preceding steps.

6.1.4 Step Planner

The step planner is the component of the Walking Engine that determines the reference pose of the next footstep, given the current step and the reference trajectory that is set by the motion planner (cf. section 4.7). It is based on the idea of modeling this problem as a geometric constraint satisfaction problem. To simplify this task, the problem is split into the computation of the rotational and translational components of the step.

For the rotational component, the application of constraints means the intersection of angle intervals. There is no special handling for the discontinuity of angles at $-\pi/\pi$ which is justified by the fact that the step size never comes close to these values. Whenever there is no intersection between the previous and the new interval, the boundary of the previous interval that is closest to the new interval is chosen as the final result. The following intersections are applied in order to get the rotational step:

1. The configured maximum velocity must not be exceeded, resulting in an interval $[-\theta_{\max}, \theta_{\max}]$.

2. The velocity must not change more than a configurable amount, meaning intersection with the interval $[\theta_{\text{previous}} - \Delta\theta_{\max}, \theta_{\text{previous}} + \Delta\theta_{\max}]$. Here θ_{previous} is the rotation of the previous step and $\Delta\theta_{\max}$ is the maximum change of the step rotation per step.

3. Braking must still be possible without the need to change the step size more than $\Delta\theta_{\max}$ in any future step. The number of steps k that is needed to bring the rotational velocity to zero without changing the step rotation more than $\Delta\theta_{\max}$ per step is determined by the formula

$$k = \left\lceil \frac{1}{2}\left(1 + \sqrt{1 + 8\frac{|\theta_{\text{target}}|}{\Delta\theta_{\max}}}\right) \right\rceil - 1. \tag{6.1}$$

Given this number of steps, the maximum rotation of the next step can be calculated directly:

$$\theta_{\text{brake}} = \frac{|\theta_{\text{target}}|}{k} + \frac{1}{2}\Delta\theta_{\max}(k-1) \tag{6.2}$$

with θ_{target} being the target rotation that should be achieved. The intersection is then performed with the interval $[-\theta_{\text{brake}}, \theta_{\text{brake}}]$.

4. To obtain the actual rotation θ_{final} for the next step, the interval is intersected with the number θ_{target}. This means that if θ_{target} lies inside the previous interval, the target will be reached directly. Otherwise, the largest possible rotation step in the direction of the target rotation is planned.

The translational part of the step is determined by the intersection of two circles and a line. One of the circles is centered around the origin of the robot coordinate system at the start of the planned step. The other circle is centered around the point where the next coordinate system origin would be if the same step as before is executed. The line describes the direction towards which the robot should walk. Thus, the current state of the planning can be represented by the radius of the centered circle r_{centered}, the center of the outer circle $\mathbf{x}_{\text{outer}}$ and the radius of the outer circle r_{outer}. The same rule as for the rotation is applied: If the addition of any new constraint makes the problem unsolvable, the point from the previous region that is closest to the intersected region is chosen is the final result.

1. r_{centered} is set to the maximum configured step size.

2. $\mathbf{x}_{\text{outer}}$ and r_{outer} are initialized to the position of the previous step and the configured maximum step size change, respectively. This avoids too large accelerations.

3. r_{centered} is multiplied by $1-\frac{\theta_{\text{final}}}{\theta_{\max}}$. The idea is that large rotational and translational steps at the same time lead to instability of the walk.

4. r_{centered} is reduced (if needed) to allow for braking by applying Equation 6.2 to the distance to the target.

5. The desired target is incorporated by intersecting the previous geometric object (i. e. the intersection of centered and outer circle) with the half-line originating in the center of the circle going in the direction of the target. The resulting line segment is represented by the radii r_1, r_2 of first and last point on the segment. Two easy special cases can be handled early: If the centered circle is completely inside the outer circle, the line segment goes from the center of the centered circle to its boundary, meaning $r_1 = 0$ and $r_2 = r_{\text{centered}}$. If, on the other hand, the centered circle completely encloses the outer circle, only the intersections of the direction half-line with the outer circle are relevant, which yield r_1 and r_2. In case there are no intersections, the point on the outer circle that is closest to the half-line is already chosen as the final step position.

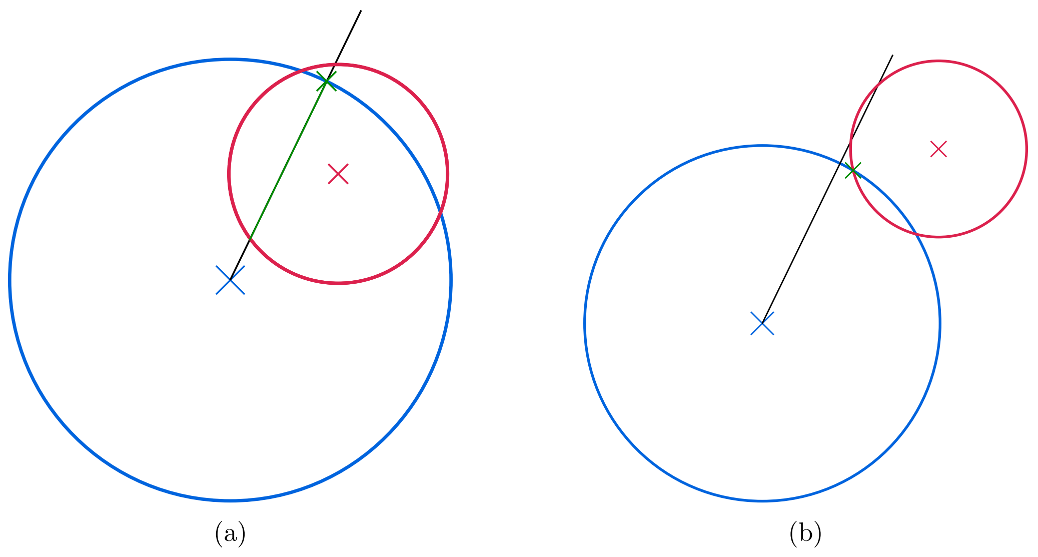

Figure 6.1: Visualization of the last phase of the step planner. The centered circle and the outer circle are depicted in blue and red, respectively. The green cross marks the step position that is selected. a) A situation in which all constraints can be fulfilled. The allowable part of the half-line is marked green. b) The intersection of the two circles and the half-line is empty. The closest point from the intersection of the two circles is chosen.

> However, in general, the boundaries of the outer and the centered circle intersect in at least one point. The intersections of the half-line and the outer circle are computed first. If the inner intersection is inside the centered circle, r_1 and r_2 are well defined. To obtain the final step, the distance to the target pose is clipped to the interval $[r_1, r_2]$ and used to scale the direction to the target. There is also special handling for cases in which some these intersections do not exist. Figure 6.1 illustrates two possible situations.

Finally, if $\frac{\theta_{\text{final}}}{\theta_{\text{max}}} + \frac{l_{\text{final}}}{l_{\text{max}}}$ is greater than one — which can happen if the step length can not be reduced far enough — the rotational step is reduced. This might reduce the rotational step more than $\Delta\theta_{\text{max}}$ but it has shown to be no issue in practice. The only way to do this entirely correct would be to use an ellipsoid for combined rotation and translation.

6.2 Adaptable Kick

At present, we use kick motions that are generated from motion files. As described in [6] we have worked on implementing and testing a more sophisticated kick motion. For the Motion Planner frequently wastes time on precise positioning and aligning behind the ball prior to a kick, we strived to design a kick motion that is capable of adjusting

its trajectory to the ball position. The desired adaptability of the motion was achieved by using Dynamic Movement Primitives. A more detailed description of the approach and its implementation can be found in [13].
The time lost during positioning and aligning was significantly reduced with the most recent iteration of the Motion Planner. Nonetheless, an adaptable kick motion still provides advantages when compared to motion files. In particular, it can more easily be adjusted to different artificial turf conditions or robots with varying hardware conditions because it comes with a set of tunable parameters, whereas tuning joint angles in motion files is rather unintuitive.

6.3 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 we found that such incidents were strongly correlated with dmesg events of reconnecting USB-devices as depicted below.

Listing 6.3.1 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

$$\theta_{\mathrm{max}} = \theta_{\mathrm{omax}} + \frac{1}{2}\left(\theta_{\mathrm{imax}} - \theta_{\mathrm{omax}}\right)\left[1 + \cos\frac{\pi}{\varphi_{\mathrm{r}}\varphi_{\mathrm{p}}}\right]. \tag{6.3}$$

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 pitch.

6.4 Sonar Filter

The sonars 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 [10]. Below 0.2 m the sensor saturates, thus not being able to provide any reliable distance measurements. Therefore, we reject all measurements exeeding 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.

6.5 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 bodies 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 perfomed 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 [12] was implemented. This algorithm utilizes quaternions for internal state representation, thus evading the aforementioned singularty issues. The implementation was validated with a dataset provided by [4], 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 occured during RoboCup 2017.

Bibliography

[1] Faber, F., Behnke, S.: Stochastic Optimization of Bipedal Walking using Gyro Feedback and Phase Resetting. In: 7th IEEE-RAS International Conference on Humanoid Robots. pp. 203–209 (2007)

[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] 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`

[4] 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)

[5] Riebesel, N., Hasselbring, A., Peters, L., Poppinga, F.: Team Research Report 2016 (2016), `https://www.hulks.de/_files/TRR_2016.pdf`

[6] Riebesel, N., Hasselbring, A., Peters, L., Wege, F., Kahlefendt, C.: Team Description Paper 2017 (2017), `https://www.hulks.de/_files/TDP_2017.pdf`

[7] 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`

[8] Röfer, T., Laue, T., Hasselbring, A., Richter-Klug, J., Röhrig, E.: B-Human 2017 – Team Tactics and Robot Skills in the Standard Platform League. In: RoboCup 2017: Robot World Cup XXI. Lecture Notes in Artificial Intelligence, Springer (2018)

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

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

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

[12] 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)

[13] Wege, F.: Developement and Implementation of a Dynamic Kick for the NAO Robotic System (2017), https://www.hulks.de/_files/BA_Felix_Wege.pdf

All links were last followed on January 30, 2018.