

Berlin United - Nao Team Humboldt
Team Report 2019
DRAFT v0.1

Heinrich Mellmann, Benjamin Schlotter, Steffen Kaden, Philipp Strobel, Thomas Krause, Etienne Couque-Castelnovo, Claas-Norman Ritter, Robert Martin

Adaptive Systeme, Institut für Informatik, Humboldt-Universität zu Berlin, Unter den Linden 6, 10099 Berlin, Germany

https://naoth.de

nao-team@informatik.hu-berlin.de

22nd January 2020

Contents

1 Introduction **3**
- 1.1 Team . . . 3
- 1.2 Summary . . . 4

2 Architecture **5**
- 2.1 NaoSMAL . . . 5
- 2.2 Platform Interface . . . 6
- 2.3 Module framework . . . 7
- 2.4 Serialization . . . 10

3 Debugging and Tools **13**
- 3.1 Concepts . . . 13
- 3.2 RobotControl . . . 14
- 3.3 Robot Setup and Deployment . . . 22
- 3.4 Logging . . . 25
- 3.5 Simulation . . . 27

4 Visual Perception **30**
- 4.1 Green Detection . . . 30
- 4.2 ScanLineEdgelDetector . . . 34
- 4.3 FieldDetector . . . 34
- 4.4 LineGraphProvider . . . 35
- 4.5 RansacLineDetector . . . 36
- 4.6 GoalFeatureDetector . . . 36
- 4.7 GoalDetector . . . 37
- 4.8 Black&White Ball Detection . . . 38

5 Modeling **45**
- 5.1 Camera Matrix Calibration . . . 45
- 5.2 Probabilistic Compass . . . 46
- 5.3 Multi-Hypothesis-Extended-Kalman-Filter Ball Model . . . 47

5.4 Multi-Hypothesis Goal Model (MHGM) 48
5.5 Simulation Based Selection of Actions 49
5.6 Arm Collision Detection 55
5.7 Time synchronization 56
5.8 IMU 57

6 Motion Control **58**
6.1 Walk 59
6.2 Energy Efficient Stand 62

7 Behavior **64**
7.1 Team Strategy 65
7.2 Role Change 66
7.3 Teamball 69

Chapter 1

Introduction

This document gives an overview over the current state of the software base developed by the *Berlin United - Nao Team Humboldt* to control a humanoid robot in the context of the robot soccer competition *RoboCup*. The release of the code base accompanying this report and the corresponding technical documentation can be found under the following links:

Docu: https://github.com/BerlinUnited/NaoTH/wiki

Code: https://github.com/BerlinUnited/NaoTH

1.1 Team

The research group *NaoTH* is part of the research lab for Adaptive Systems at Humboldt-Universität zu Berlin headed by Prof. Verena Hafner. At the current state the core team consists of about 12 students of Bachelor, Master/Diploma, and PhD levels. Besides the direct participation at the RoboCup competitions NaoTH is involved in teaching at the university, public engagement and building of the RoboCup community.

NaoTH was founded at the end of 2007 at the AI research lab headed by Prof. Hans-Dieter Burkhard. As a direct successor of the *Aibo Team Humboldt* which was active in the Four Legged League of the RoboCup as a part of the *GermanTeam* until 2008. GermanTeam won the world championship three times during its existence.

NaoTH participated yearly at the RoboCup competitions since the first SPL competition in 2008 in Suzhou. The most recent achievements include 3rd place in the Outdoor Competition at the RoboCup world championship in 2016 and 2nd place in the Mixed Team Competition as part of the team DoBerMan at the RoboCup 2017.

In 2010 and 2011 we also competed in the 3D Simulation league with the same code as used for the SPL. In the 3D Simulation, we won the German Open and the AutCup competitions and achieved the 2nd place at the RoboCup World Championship 2010 in Singapore.

In 2011 we formed a conjoint team *Berlin United* with the team FUmanoids from Berlin, which participated in the KidSize League. The collaboration included extensive exchange of experience, sharing of code and organization of joint workshops. In 2017 the team FUmanoids ceased to exist. Since then *NaoTH* remains the only member of Berlin United and continues to compete under this name.

In 2017 and 2018 *NaoTH* competed together with the team NaoDevils as joint team *DoBerMan* at the RoboCup SPL Mixed Team Competition and achieved 2nd place in both years. In 2017 *NaoTH* won the 2nd place in the challenger shield and in 2018 reached the quarterfinals in the champions cup of the main competition.

In 2019 we won the SPL Technical Challenge and together with B-Human we won the Mixed Team Competition.

1.2 Summary

Our general research fields include agent-oriented techniques and Machine Learning with applications in Cognitive Robotics. Currently, we mainly focus on the following topics:

- Software architecture for autonomous agents (section 2)
- Narrowing the gap between simulated and real robots (section 3.5)
- Dynamic motion generation (section 6)
- World modeling (section 5)

Chapter 2

Architecture

An appropriate architecture is the base of each successful software project. It enables a group of developers to work at the same project and to organize the solutions for their particular research questions. From this point of view, the artificial intelligence and/or robotics related research projects are usually more complicated than commercial product development, since the actual result of the project is often not clear. Since we use this project also in education, a clear organization of the software is necessary to achieve a fast familiarization with the software. Our software architecture is organized with the main focus on modularity, easy usage, transparency and convenient testing capabilities.

In the following subsections we describe the design and the implementation of different parts of the architecture. A detailed description of the principles we used can also be found in [11].

2.1 NaoSMAL

In our architecture we don't use the NAOqi API directly but use our own so-called NaoSMAL (Nao Shared Memory Abstraction Layer) NAOqi-module. This calls the DCM API of NAOqi[1] and makes it accessible for other processes via a shared memory interface. Thus we can implement our own code as a completely separated executable that has no dependencies to the NAOqi framework. The benefits are a safer operation of the Nao on code crashes (NaoSMAL will continue to run and ensures the robot will go in a stable position), faster redeploy of our binary without restarting NAOqi and a faster compilation since we have fewer dependencies.

[1] http://doc.aldebaran.com/1-14/naoqi/sensors/dcm-api.html

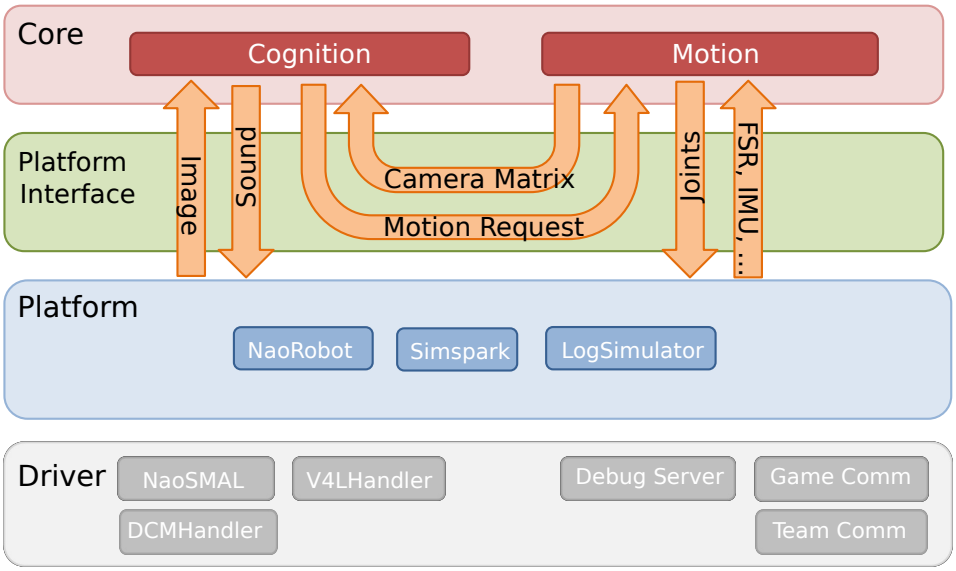


Figure 2.1: Platform Interface is responsible for data transfer and execution of the Cognition and Motion processes.

2.2 Platform Interface

In order to integrate different platforms, our project is divided into two parts: a platform independent one and platform specific one. The platform specific part contains code which is applied to the particular robot platform. We support the Nao hardware platform, the SimSpark simulator[2] and a logfile based simulator. While the platform specific part is a technical abstraction layer the platform independent part is responsible for implementing the actual algorithms. Both parts are connected by the *platform interface*, which transfers data between the platform independent and specific part (see Fig. 2.1).

[2] http://simspark.sourceforge.net/

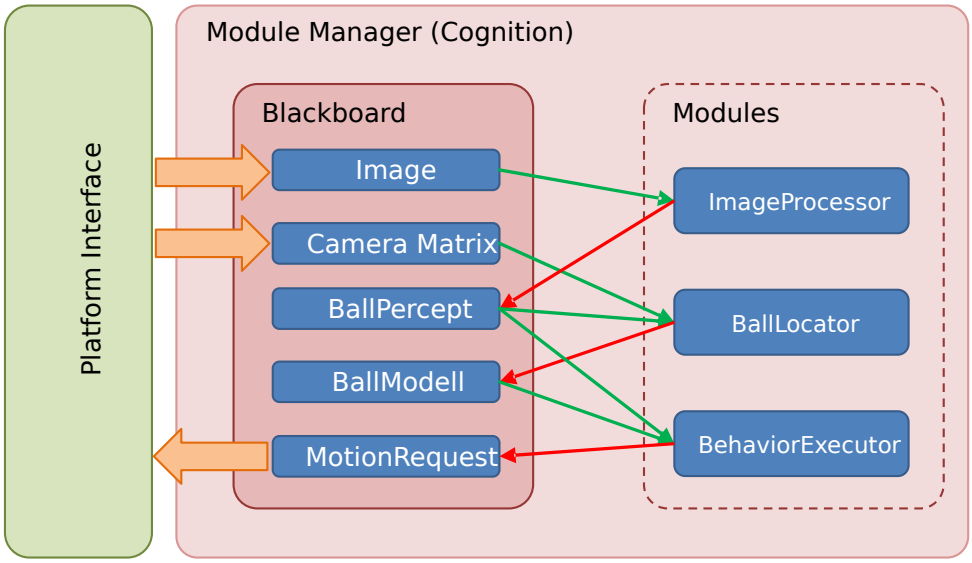


Figure 2.2: Overview about the different components of the module framework.

2.3 Module framework

Our module framework is based on a *blackboard architecture*. The framework consists of the following basic components:

Representation (objects carrying data and simple manipulation functions),

Blackboard (container storing representations as information units),

Module (executable unit, has access to the blackboard and can read and write representations),

Module Manager (manage the execution of the modules).

Figure 2.2 describes the interaction between these components. A module may *require* a representation, in this case it has read-only access to it. A module *provides* a representation, if it has write access. In our design we consider only sequential execution of the modules, thus there is there is no necessity

for handling concurrent access to the blackboard. We decide which representation is required or provided during compilation time. Different modules can implement similar functionality and provide the same representations. You can configure which of the modules should be executed at runtime and it is also possible to dynamically change this for debugging purposes.

2.3.1 Example module

A module is a C++ class which inherits a base class which is created with the help of some macros defining the interface of the the module.

```
#ifndef _MyModule_H
#define _MyModule_H

#include <ModuleFramework/Module.h>
#include <Representations/DataA.h>
#include <Representations/DataB.h>

BEGIN_DECLARE_MODULE(MyModule)
 REQUIRE(DataA)
 PROVIDE(DataB)
END_DECLARE_MODULE(MyModule)

class MyModule: public MyModuleBase
{
public:
 MyModule();
 ~MyModule();

 virtual void execute();
};

#endif  /* _MyModule_H */
```

Listing 2.1: MyModule.h

The `MyModule` class inherits the `MyModuleBase` class which was defined with the `BEGIN_DECLARE_MODULE` macro. Each representation which is needed by the module is either declared as provided or required with the corresponding macro. After declaring a representation it is accessible with a getter function, which has the name of the representation prefixed with "get', e.g. `getDataA()` for the representation `DataA`. The actual implementation of the functionality of a module must be in the `execute()` function.

```
#include "MyModule.h"
```

```
MyModule::MyModule()
{
  // initialize some stuff here
}

MyModule::~MyModule()
{
  // clean some stuff here
}

void MyModule::execute()
{
  // do some stuff here
  getDataB().x = getDataA().y + 1;
}
```

Listing 2.2: MyModule.cpp

A representation can be any C++ class, it does not need to inherit any special parent class.

```
class DataA
{
public:
  DataA(){}

  int y;
};

class DataB
{
public:
  DataB(){}

  int x;
};
```

Listing 2.3: DataA.h/DataB.h

A module must be registered in the cognition process by including it in the file `NaoTHSoccer/Source/Core/Cognition/Cognition.cpp`.

```
#include "Modules/Experiment/MyModule/MyModule.h"
```

In the `init` method add the line:

```
REGISTER_MODULE(MyModule);
```

The order of registration defines the order of execution of the modules.

2.4 Serialization

As described in the Section 2.3 the core of the program is structured in modules which are responsible for different tasks like image processing, world modeling etc.. The modules communicate with each other through the *blackboard* by writing their results to *representations*. The *representations* are mainly pieces of data and have no significant functionality. These representation can be made *serializable*, which is mainly used in two cases: logging to a file and sending over the network for debug or monitoring reasons.

The backbone of the serialization framework is formed by the Google *Protocol Buffers*[3] (protobuf) library. For a representation to be serialized (which is described by a C++ class) an according protobuf message is defined. Please refer to the documentation page of protobuf for more details on this part. The serialization procedure is then performed in two steps: first the data is copied from the object which is to be serialized to the according message object; in the second step the message object is serialized by a protobuf serializer to a byte stream. The deserialization process works in reverse order. The second step is entirely done by the protobuf library. The copy procedure in the first step, however, has to be defined explicitly. This procedure is described in the `serialize()` and `deserialize()` functions of the template class `Serializer` which has to be specified for each representation to be serialized.

The following listings illustrate the whole code necessary for serialization of a representation. Listing 2.4 shows the header file *MyRepresentation.h* containing the declaration of the representation class `DataA` and the according specialization of the serializer `Serializer<DataA>`. Listing 2.5 contains the probobuf message for `DataA`. Listing 2.6 illustrates the implementation of the serialization functions in the file *MyRepresentation.cpp*.

```
#include <Tools/DataStructures/Serializer.h>

class DataA
{
public:
  DataA()
    :
    y(0),
    time(0.0)
  {}

  int y;
  double time;
```

[3] https://developers.google.com/protocol-buffers

```
};

namespace naoth {
template<>
class Serializer<DataA>
{
public:
  static void serialize(const DataA& representation, std::
     ostream& stream);
  static void deserialize(std::istream& stream, DataA&
     representation);
};
}
```

Listing 2.4: MyRepresentation.h

```
package mymessages;

message DataA {
  required double time = 1;
  required int32 y = 2;
}
```

Listing 2.5: messages.proto

```
#include "MyRepresentation.h"
#include "Messages/mymessages.pb.h"
#include <google/protobuf/io/zero_copy_stream_impl.h>

using namespace naoth;

void Serializer<DataA>::serialize(const DataA& data, std::
    ostream& stream)
{
  // create a new message
  messages::DataA msg;

  // copy data from the representation to the message
  msg.set_y(data.y);
  msg.set_time(data.time);

  // serialize the message to stream
  google::protobuf::io::OstreamOutputStream buf(&stream);
  msg.SerializeToZeroCopyStream(&buf);
}
```

```
void Serializer<DataA>::deserialize(std::istream& stream,
    DataA& data)
{
  // create a new message
  messages::DataA msg;

  // decerialize the message from stream
  google::protobuf::io::IstreamInputStream buf(&stream);
  msg.ParseFromZeroCopyStream(&buf);

  // copy data from the message to the data
  data.y = msg.y();
  data.time = msg.time();
}
```

Listing 2.6: MyRepresentation.cpp

Chapter 3

Debugging and Tools

In order to develop a complex software for a mobile robot, we require means for high-level debugging and monitoring (e. g., visualization of the robot's posture or its position on the field). Since we do not exactly know which kind of algorithms will be debugged, there are two aspects of high importance: accessibility at runtime and flexibility. The accessibility of the debug construct is realized based on our communication framework. Thus, they can be accessed at runtime by using visualization software like RobotControl, as shown in Figure 3.1.

3.1 Concepts

Some of the ideas were evolved from the GT-Architecture [12]. The following list illustrates some of the debug concepts:

debug request (activates/deactivates code parts),

modify allows modification of a value (in particular local variables)

stopwatch measures the execution time

parameter list allows to monitor and modify lists of parameters

drawings allows visualization in 2D/3D; thereby it can be drawn into the image or on the field (2D/3D)

plot allows visualization of values over time

As already mentioned, these concepts can be placed at any position in the code and can be accessed at runtime. Similar to the module architecture,

the debug concepts are hidden by macros to allow simple usage and to be able to deactivate the debug code at compilation time, if necessary.

In order to use a debug request in the code you have to register it once with the DEBUG_REQUEST_REGISTER macro:

```
DEBUG_REQUEST_REGISTER("My:Debug:Request", "Description of
    the debug request", true);
```

After that, you can use the DEBUG_REQUEST macro to wrap code that should be only executed when the debug request is active.

```
DEBUG_REQUEST("My:Debug:Request",
    std::cout << "This code is not executed normally" << std
        ::endl;
    ++c;
);
```

MODIFY works in a similar way, but does not need any registration. By, for example, wrapping a variable and defining an identifier, this variable can be changed later from RobotControl.

```
double yaw  = 0;
MODIFY("BasicTestBehavior:head:headYaw_deg",yaw);
```

In addition to these means for individual debugging, there are some more for general monitoring purposes: the whole content of the blackboard, the dependencies between the modules and representations, and execution times of each single module. The Figure 3.1 illustrates visualizations of the debug concepts. In particular a field view, 3D view, behavior tree, plot and the table of debug requests are shown.

3.2 RobotControl

The various debugging possibilities are organized in different dialogs. In this year we started to update the UI by switching to JavaFX for some dialogs. The following list consists of our most used RobotControl Dialogs.

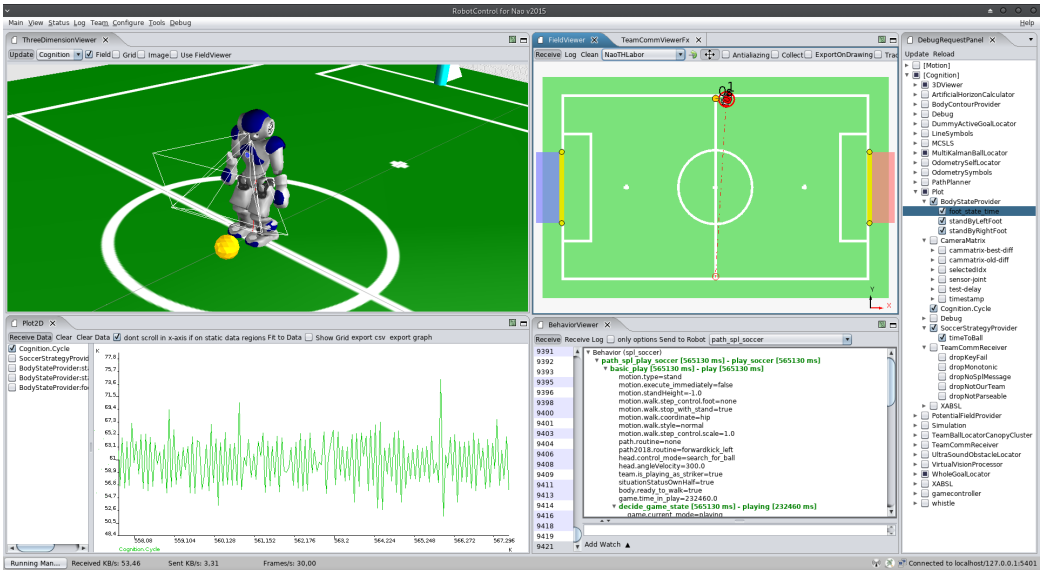
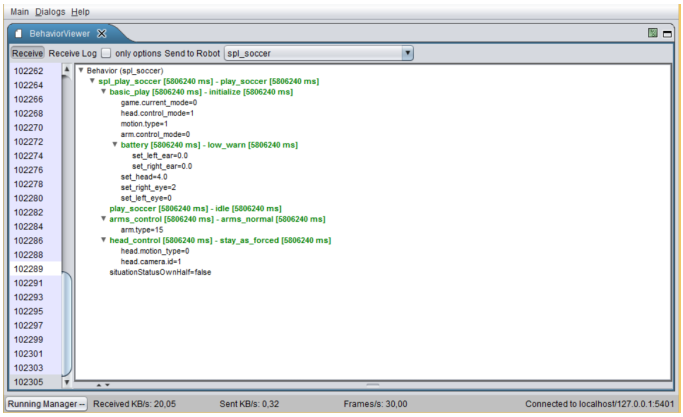


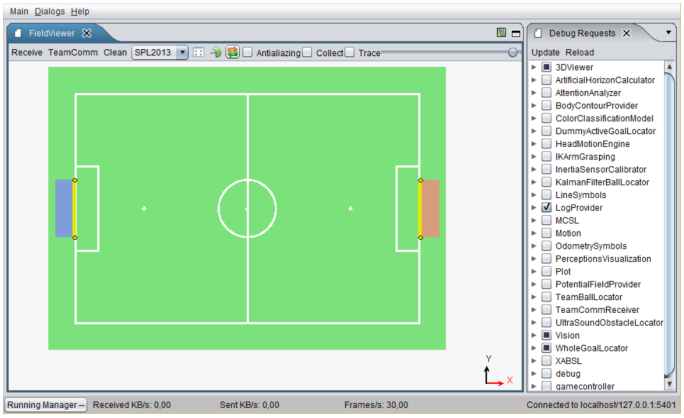
Figure 3.1: The RobotControl program contains different dialogs. The 3DViewer (top left) is used to visualize the current state of the robot; the Value Plotter dialog (bottom left) plots some data; the Field Viewer dialog (top center) draws the field view; the Behavior dialog (bottom center) shows the behavior tree; the Debug Request Center dialog (right) is for enabling/disabling debug requests.

Behavior Viewer



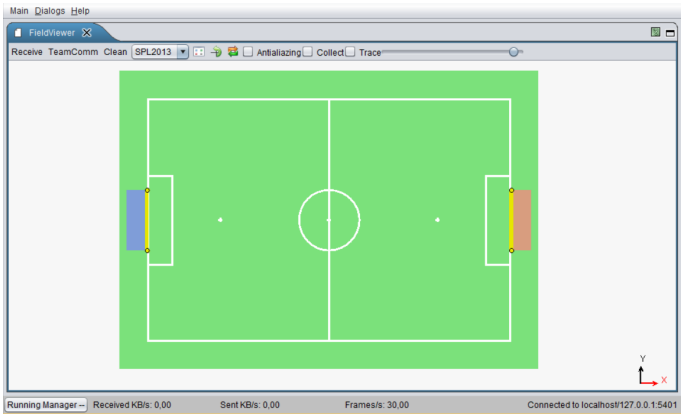
Shows the behavior tree for the current behavior. The compiled XABSL behavior needs to be sent to the robot first and then an agent can be selected to be executed. With 'Add Watch' you can track XABSL input and output symbols.

Debug Requests



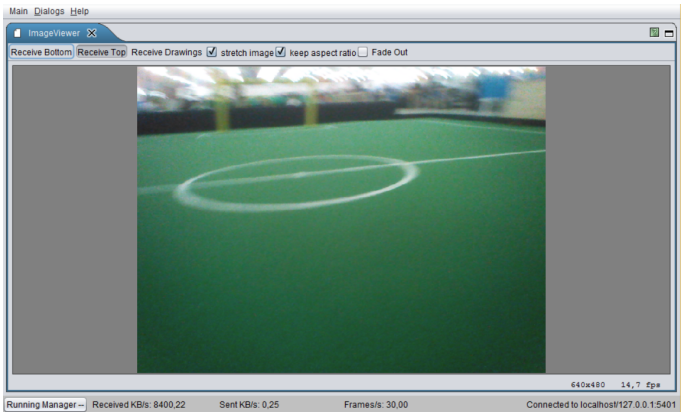
(De-)activates the debug request code. Usually a debug request draws something on the field viewer or on the camera images. For further information about individual debug requests, have a look at the source code.

Field Viewer



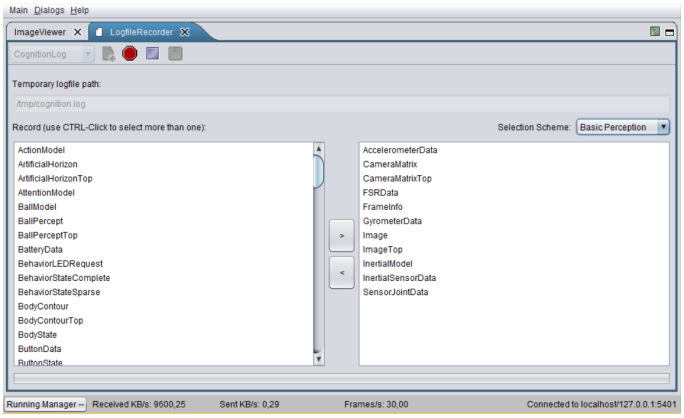
There are views for different field sizes and a local view. Certain debug requests draw on these views. For example, you could draw the robots' positions on the field by activating the corresponding debug request.

Image Viewer



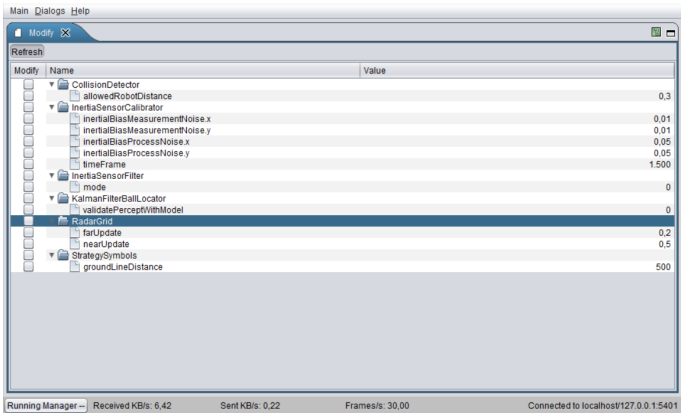
Can show the top and bottom images. There are debug requests that draw on the camera images, if they are active.

Logfile Recorder



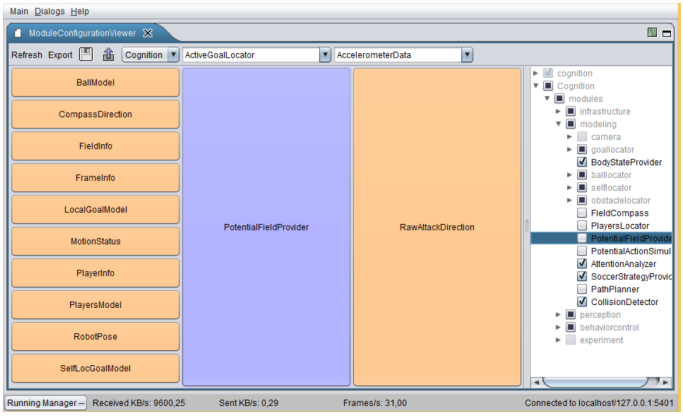
Records a log file on a robot with manually selected representations.

Modify



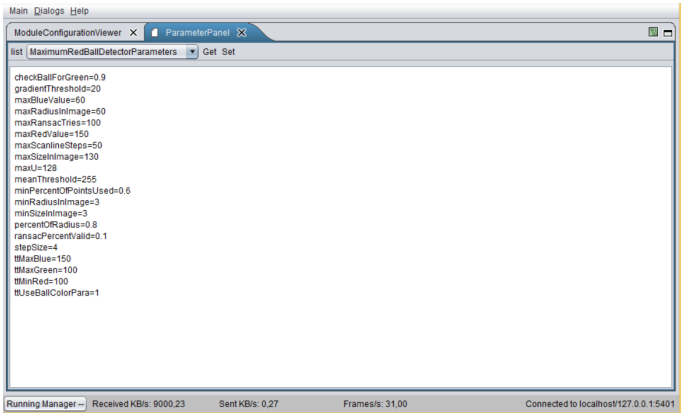
The Modify macro allows changing values of variables declared within this macro at runtime.

Module Configuration Viewer



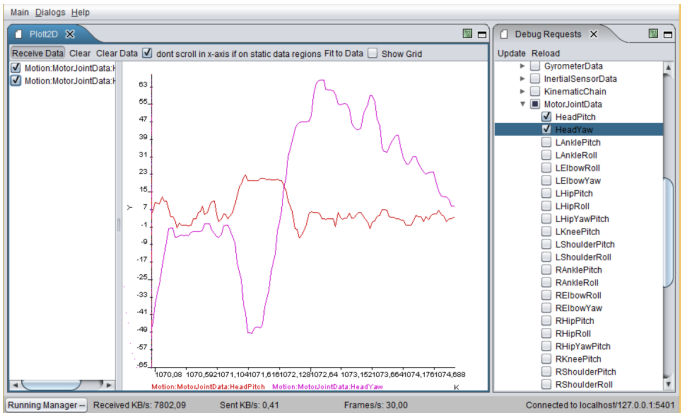
Shows which modules are currently (de-)activated. Also indicates, which other modules are required (left) and provided (right) by each module.

Parameter Panel



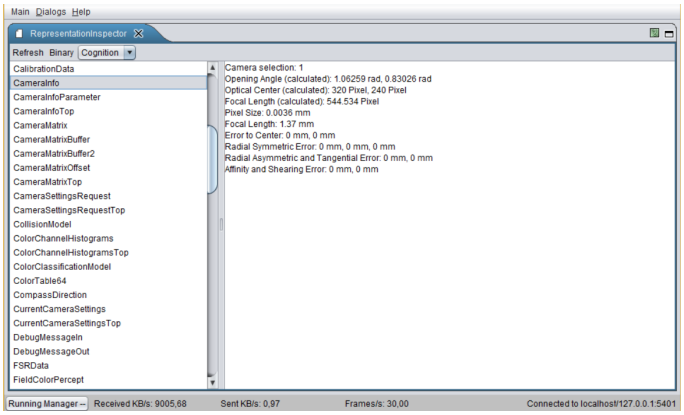
Shows parameters defined in our configuration files. It is possible to change the values at runtime. The variables must be registered as parameters in the code.

Plot 2D



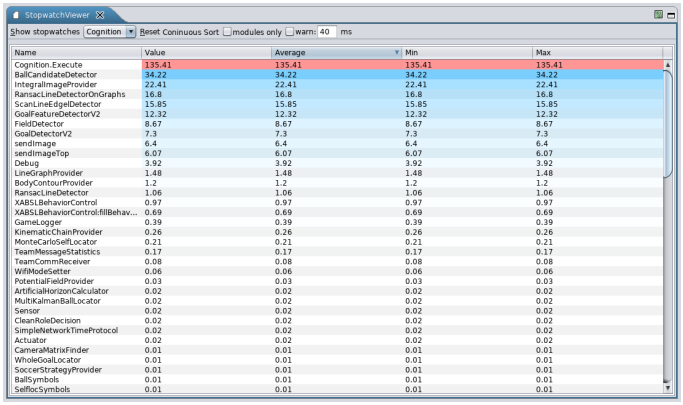
Shows plots activated by plot debug requests.

Representation Inspector



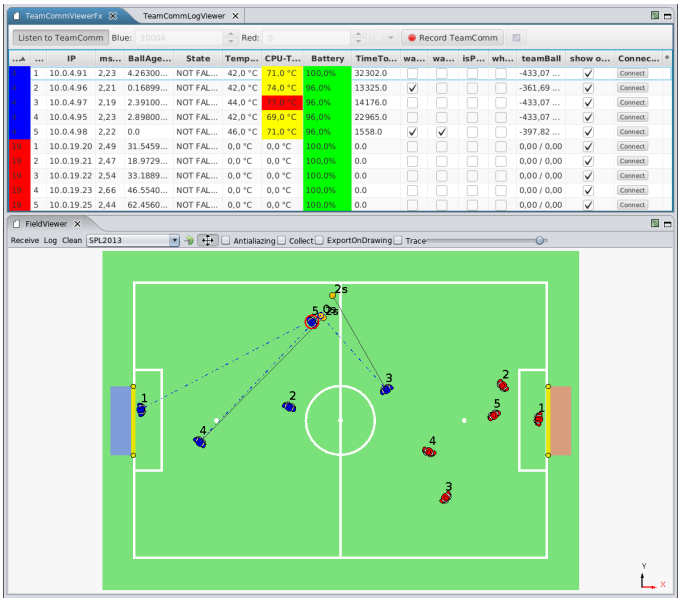
Shows the data that is written to the blackboard by each representation.

Stopwatch Viewer



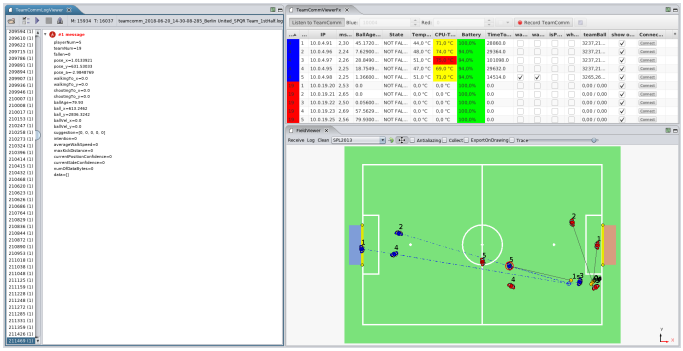
Shows the execution time for each module.

Team Communication Viewer



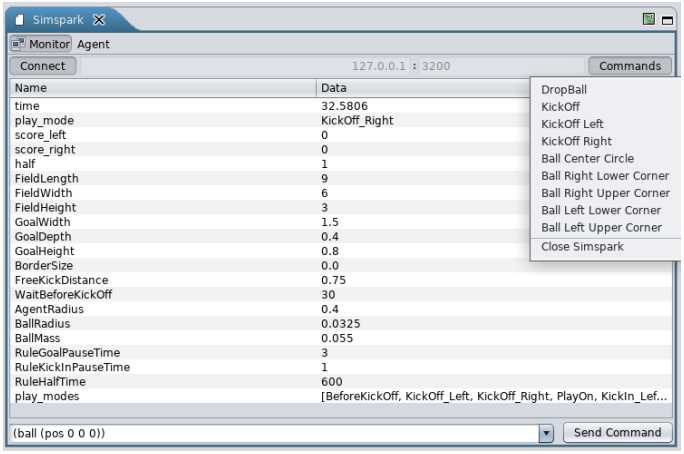
Shows all connected robots and possible all of their provided status information via TeamComm (e.g. ip address, battery charge, temperature, etc.). This dialog is mainly used during competitions to get a quick overview of the robots health status, their communicated data (e.g. ball, whistle detection, etc.) and their role decisions. Positions of the robots, role decision and seen ball for each robot are visualized in the FieldViewer 3.2. It's also useful for debugging the team communication to get a general overview of the role decision and its transitions. This year, the dialog was re-implemented using JavaFX.

Team Communication Logviewer



The *Team Communication Logviewer* allows to re-play team communication logfiles previously recorded with the *Team Communication Viewer*. It's also able to re-play logs from the GameController and export them as JSON.

Simspark



This dialog shows the state of the connected simspark instance. It also provides some predefined commands and a textfield for user-defined commands to send to simspark. Furthermore it also provides the teamcommunication of the simulated robots for other dialogs in RobotControl (e.g. *Team Communication Viewer*).

External Tools



Some „external tools" where modified in order to load them into RobotControl via reflection. Its convenient to have all needed tools in the application we're using most. Currently we

have the NaoSCP tool (see 3.3.2 NaoSCP), for the setup & deployment of a robot, and a modified version of RoboViz, for visualization of a SimSpark simulation, integrated in RobotControl.

3.3 Robot Setup and Deployment

In this section we give an overview over our deployment and robot setup procedure. Changing the configuration of the robot, e.g., deployment of the binary, network setup etc., is a critical point during both, development and competition. To minimize the chance of error we developed a set of procedures and tools.

3.3.1 Deployment Procedure

Currently we have two different deployment procedures:

- deployment via usb flash drive
- deployment via network

The general procedure consists of two steps:

1. assemble deployment directory containing all files and configurations to be deployed as well as a corresponding deployment shell script;
2. copy this directory to the robot and run the deployment script;

This division minimizes the chance of mistakes and allows for easier debugging, i.e., if something went wrong, the error is either in the locally assembled deployment directory or has occurred during the deployment on the robot - both can be inspected separately. At the same time this locally assembled deployment directory serves as a binary backup, which can be very useful during the competition, e.g., if something turned out to be wrong with the new version just before the game and one needs to switch back to the binary from the last game.

The beginning and the end of the deployment procedure are indicated by different sounds. This way the state of the robot and the progress of the deployment can be easily monitored, this is especially helpful when setting up a whole team before a game.

3.3.2 NaoSCP

NaoSCP is a setup and deployment tool. It primarily has three tasks: (1) initialize a new robot, e.g., copy libraries and scripts, (2) set the network configuration and (3) deploy naoth binary and configurations to the robot. All these tasks can be done on a command line as well, the main aims for designing *NaoSCP* were simplification of the deployment process, ensured backup of deployed binaries and reduction of the chance of mistakes during setup in critical situations, e.g., before a game at the world championship.

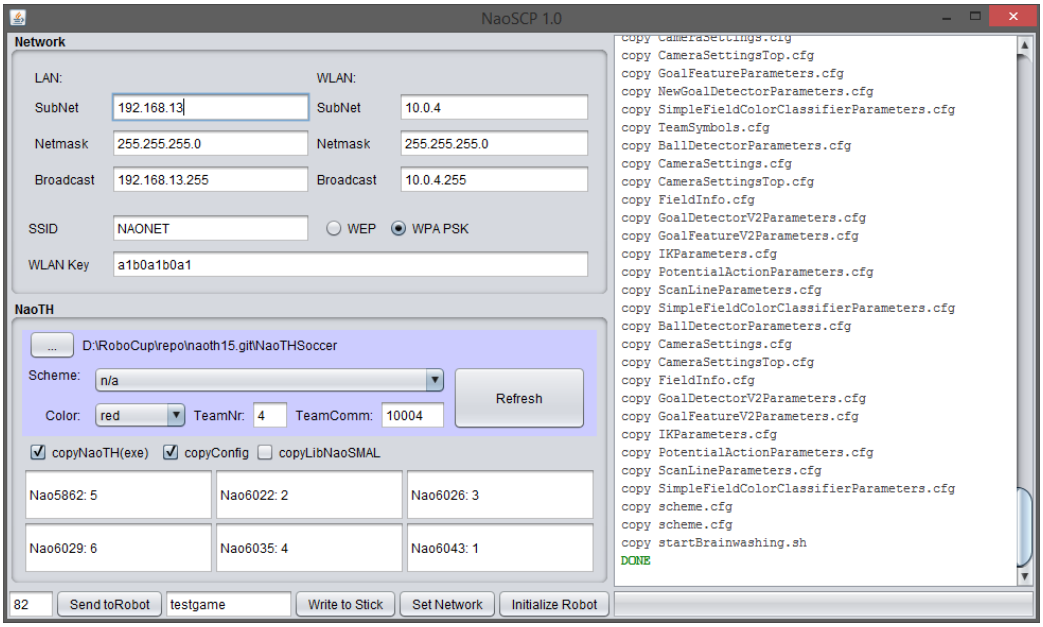


Figure 3.2: NaoSCP user interface. The log panel on the right displays status of the deployment process. The left side contains the panels for the configuration of the deployment / setup process: *Network* configures the network setup; *NaoTH* is used to adjust the configuration relevant for the deployed binary, e.g., player numbers. The buttons in the left bottom tool bar trigger particular deployment and setup actions like writing the network configuration to the robot or copying a new binary to a deployment USB flash drive.

Usage Remarks

The following describes the particular components of the NaoSCP user interface as illustrated in the Figure 3.2.

Log Window (right) shows information regarding the progress of the deployment process, e.g., copied files, connection errors and such.

Network configuration (top left) is used to setup the LAN and WLAN;

NaoTH dialog (left) configures the deployment of the game binary and contains things like the path to the source where the binaries can be found, used configuration scheme and player numbers for each robot based on their IP address;

Action toolbar (bottom left) contains the buttons for the four different deployment / setup actions: *Send to Robot* deploys the complied code

and configuration to a particular robot via network. The text field left of the button defines the last byte of the ip address of the target robot. The network configuration from the dialog *Network* is used to determine the complete address. In this particular example the LAN target address would be 192.168.13.82. Thereby LAN is tried first and in case of failure WLAN is tried; *Write to stick* writes the deployment directory to a USB flash drive. If the flash drive already contains a deployment directory, a backup version of it is created. The text field left to the button holds an optional tag, which is used to organize the backups on the flash drive; *Set Network* configures the robots network according the the settings in the dialog *Network*; *Initialize Robot* will initialize a new robot, e.g., after a factory reset. This action will copy additional libs, configure the NaoQi modules, necessary starting scripts for binaries and for automatic mounting and running of USB flash drives. Additionally the network is configured and the binary is deployed like previously described;

3.3.3 USB flash drive

A deployment flash drive can be created manually or (as described above) via the NaoSCP tool. As the minimum requirement there should be an executable shell script named "startBrainwashing.sh". When the flash drive with a shell script (and a deployment directory) is connected to the robot, it is mounted automatically to a defined directory and the setup script is executed. The script is responsible for copying the particular binary and/or configuration files. The begin and end of the deployment procedure are indicated by different sounds. This way is preferred when deploying software on several robots, e.g., setting up a team before a game.

Flash drive variants

Currently we have different kinds of USB flash drives to accomplish different kinds of deployment or collection tasks.

Deploying As described above the deployment flash drive is used to copy new binaries and/or configuration files to the robot.

Collecting log files The "collect log files" flash drive is used to copy the recorded log file from the robot. This is primarily used after games to be able to analyze possible misbehavior of the robot. Otherwise, if the robot is shut down, the log files would be lost since we record the logfiles in main memory.

In addition some informations about the robot and the currently executed binary are collected and written to the “nao.info“ file. With that information we're documenting the state of the robot/binary of the log file and giving a quick overview on later log file examination. Finally we're collecting all “heard“ whistles for later fine-tuning of our whistledetector.

Setting network The network flash drive is used to set the network configuration of the robot (like IP address, etc.). Especially for the Wi-Fi configuration this method is useful to quickly set up all need robots before a game.

3.4 Logging

Analysis and evaluation of the algorithms running on the robot is a big challenge.

Our team has a long history in logging (our logs from 2010 are still readable and useful). Through the years we developed a comprehensive infrastructure for recording log files on individual robots during the games as well as tools for synchronizing these log files with videos of the game and analyzing them.

A log file is recorded by a robot in its local file system, and is collected later through network or USB (cf. Section 3.3.3). Currently we use two different modes for recording such log files - automatic game log and log of manually selected representations.

The game log is only recorded when the robot is in a *playing* state. It is recorded with the cognition pace, i.e., each time a new image arrives (each 33ms), and contains mainly the behavior state as well as additional information needed in a particular situation. For example during the RoboCup 2016 we recorded the best four ball candidate patches from each image in each frame to create a database of realistic samples for ball detection (cf. Section 4.8.2). The game logs recorded by individual robots can be synchronized with the video of the game, which can be very useful to analyze and find bugs in the behavior patterns of the robot and ultimately in the team behavior.

Figure 3.3 illustrates the manual synchronization interface. To simplify the synchronization procedure changes in the game state, e.g., from *ready* to *play*, are automatically extracted from the log file. The operator can select a suitable event from a drop-down list. To synchronize both, the operator then needs to find the corresponding time in the video. Good events for synchronization are often changes from *init* to *ready* - in the video one can

clearly see when the robots receive the signal from the game controller and start to move.

The manual logs can be configured and triggered through the according control dialog in the *RobotControl* as shown in the Section 3.2. This is used to record specific data for debugging or analysis in isolated experiments. A good example are log files containing full images of particular situations for general image processing, which can be only recorded for a short period of time due to the large size.

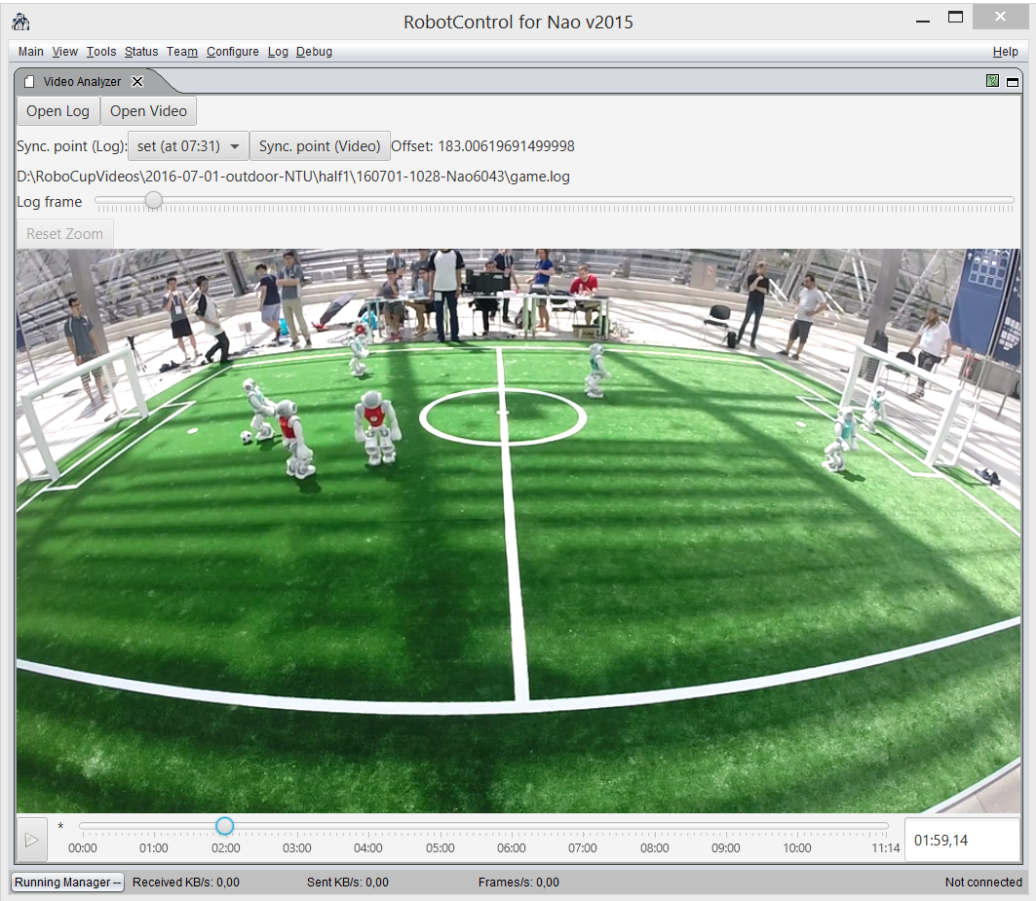


Figure 3.3: Synchronization Interface for individual log files and videos of a game.

3.4.1 Logfile Format

As described in the Section 2.3 the state of the robot is stored in *representations* on the *blackboard*. Any of these representations can be recorded to

a log file if it has a designated serializer as described in Section 2.4. The format by which the representations are stored in a logfile is pretty straight forward. Each representation is stored in a package of the following form:

```
<frame number><representation name><size><data>
```

Where <frame number> is 4 byte long, <representation name> is a string terminated with a zero character '\0' and <size> is 4 bit long length of the following binary <data>. A logfile is simply a sequence of such packages. Important to note is that when several different representations are recorded in each frame they are written in the same way, so the final log looks something like this:

```
1 FrameInfo ...
1 ImageTop ...
1 CameraMatrix ...
2 FrameInfo ...
2 ImageTop ...
2 CameraMatrix ...
...
```

Annotate and Evaluate Logs Annotation interface was created and used to annotate different kick actions executed by our robots in the videos recorded during the games at the RoboCup in 2015. The kick events were automatically extracted from the log files recorded by the individual robots and aligned with the video. Thus the human annotator can simply click through the particular events and inspect them in a short time. The results were used to evaluate the performance of the kick decision algorithm and were published in [10]. Figure 3.4 illustrates an example of a labeling session for the first half of the game with the team *NaoDevils* at the RoboCup 2015. The following two links lead to an online demo and to the public repository with the code of the labeling interface:
https://www2.informatik.hu-berlin.de/~naoth/videolabeling/index.php
https://github.com/BerlinUnited/VideoLogLabeling

3.5 Simulation

As a common experience, there are big gaps between simulation and reality in robotics, especially with regards to basic physics with consequences for low level skills in motion and perception. There are some researchers who have already tried to narrow this gap, but there are only few successful results so far. We investigate the relationships and possibilities for methods and code

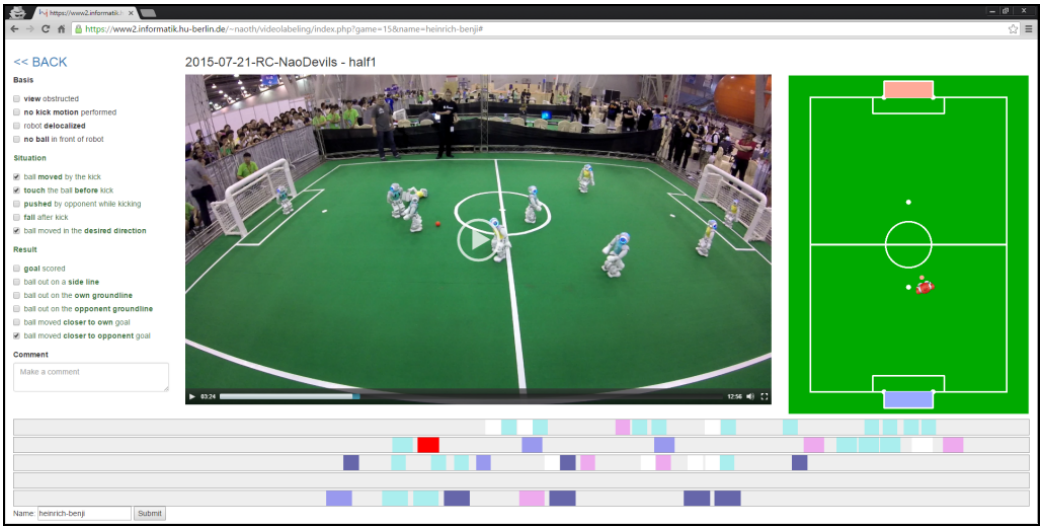


Figure 3.4: Labeling interface used to annotate kick events regarding their quality. At the bottom are time lines for each of the robots. Different actions are represented by buttons on the time line with different colors. On the right the robots estimated state is visualized, i.e., estimation of its position, ball model and obstacles. On the left are three categories of binary labels describing the quality of the action.

transferring. Consequences can lead to better simulation tools, especially in the 3D Simulation League. At the moment, we use the SimSpark simulator from the 3D Simulation League with the common core of our program, see Figure 3.5. As already stated, therewith, we want to foster the cooperation between the two leagues and to improve both of them.

When compared to real Nao robots, some devices are missing in the SimSpark, such as LEDs and sound speakers. On one hand, we extended the standard version of SimSpark by adding missing devices like camera, accelerometer, to simulate the real robot. On the other hand, we can use a virtual vision sensor which is used in 3D simulation league instead of our image processing module. This allows us to perform isolated experiments on low level (e. g., image processing) and also on high level (e. g., team behavior). Also we developed a common architecture [11], and published a simple framework allowing for an easy start in the Simulation 3D league.

Our plan is to analyze data from sensors/actuators in simulation and from real robots at first and then to apply machine learning methods to improve the available model or build a good implicit model from the data of real robot. Particularly, we plan to:

- improve the simulated model of the robot in SimSpark,

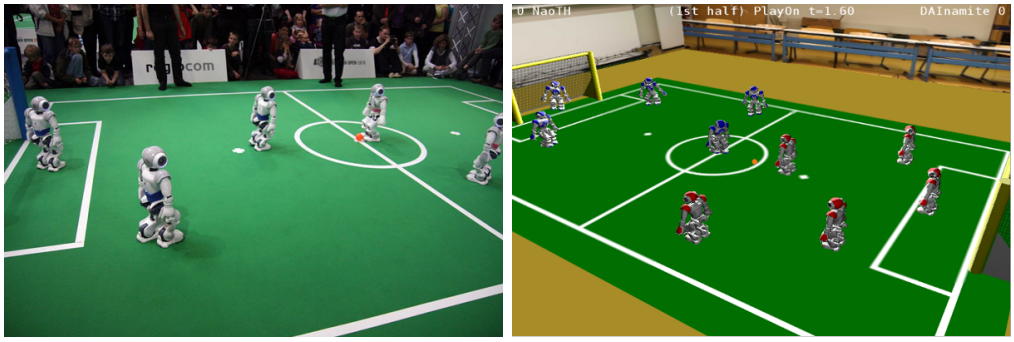


Figure 3.5: NAO robots run in Standard Platform League (left) and 3D Simulation League (right).

- publish the architecture and a version of SimSpark which can be used for simulation in SPL,
- transfer results from simulation to the real robot (e. g., team behavior, navigation with potential field).

So far, we have developed walking gaits through evolutionary techniques in a simulated environment [6, 5]. Reinforcement Learning was used for the development of dribbling skills in the 2D simulation [14], while Case Based Reasoning was used for strategic behavior [4, 2]. BDI-techniques have been investigated for behavior control, e. g., in [1, 3].

Chapter 4

Visual Perception

Visual perception is the primary way for the NAO robot of perceiving its environment. To reduce computational complexity our vision is based on a reliable field color detection. This color information is used to estimate the boundaries of the visible field region in the image. Which is done while scanning for line edgels - oriented jumps in the brightness channel. Detection of other objects - lines, ball, goals - is performed within this field region. Lines are modeled as a graph of the aforementioned edgels. Ball detection consists mainly of two steps - key points (ball candidates) are detected using *integral image* and *difference of gaussians*, which are then classified by a trained cascade classifier. Goal post detection uses scan lines along the horizon.

Figure 4.8 shows the dependency graph for the vision modules[1] and the representations they provide, which were used at the RoboCup 2016 competition. In the following we describe some of the important modules in more detail.

4.1 Green Detection

This section describes a new approach to classify the field color which has not been used at the RoboCup 2015.[2] This constitutes the first step in the attempt for a automatic field color detection. Thereby we analyze the structure of the color space perceived by the robot NAO and propose a simple yet powerful model for separation of the color regions, whereby green color is of a particular interest.

To illustrate our findings we utilize a sequence of images from recorded

[1]For definition of Modules and Representations see Section 2.

[2]For the first time this approach has been presented in November 2015 at the RoHOW workshop in Hamburg.

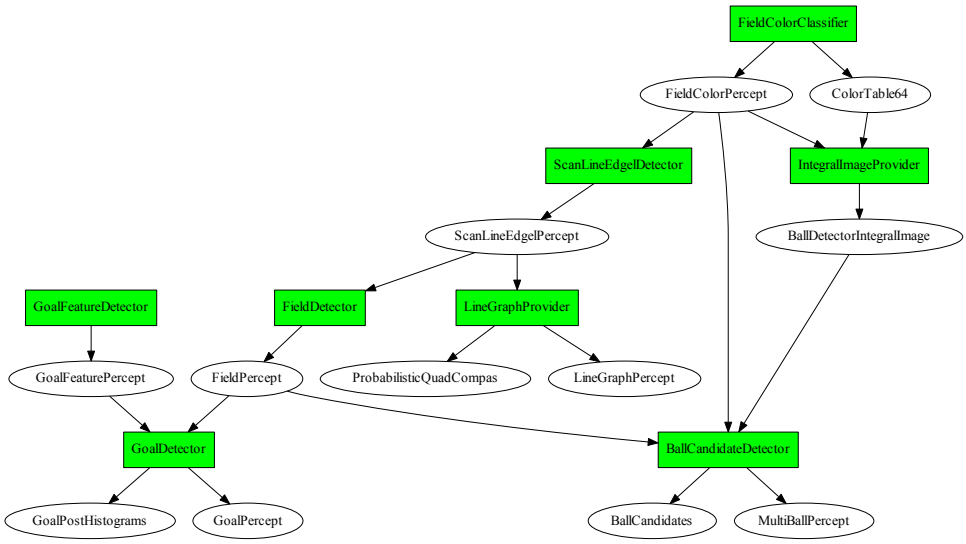


Figure 4.1: Overview over the vision system. Green boxes illustrate modules and round nodes visualize the representations with arrows indicating the provide-require relationships between them. An outgoing arrow from a module A to a representation R means A provides R; an incoming arrow from R to A means R is required by A.

by a robot during the Iran Open 2015. Figure 4.2 (left) shows a representative image from this sequence. To analyze the coverage of the color space we calculate two color histograms over the whole image sequence. In the Figure 4.3 (left) you can see the uv-histogram, which is basically a projection of the yuv-space onto the uv-plane. The light green points indicate the frequency of a particular uv-value (the brighter the more). One can clearly recognize three different clusters: white and gray colors in the center; green cluster oriented towards the origin; and a smaller cluster of blue pixels in the direction of the u-axis which originate from the boundaries around the field. For the second histogram we choose a projection plane along the y-axis and orthogonal to the uv-plane which is illustrated in the Figure 4.3 (left) by the red line. This plane is chosen in a way to illuminate the relation between the gray cluster in the center and the green cluster. Figure 4.3 (middle) illustrates the resulting histogram. Here we clearly see the gray and the green cluster.



Figure 4.2: (left) Example image from the Iran Open 2015. (right) Pixels classified as green are marked green; pixels with too low chroma marked red;

From these two histograms we can make following observations: all colors seem to be concentrically organized around the central brightness axis , i.e., gray axis $(128, 128, y)$, which corresponds to the general definition of the yuv-space; the colors seem to be pressed closer to the gray axis the darker they are. In particular all colors below a certain y-threshold seem to be collapsed to the gray axis. So we can safely claim that for a pixel (y, u, v) always holds $y = 0 \Rightarrow u, v = 128$. On the contrary the spectrum of colors gets wider with the rising brightness. Speculatively one could think that the actual space of available colors is a hsi-cone fitted into the yuv-cube. The collapse of the colors towards the gray axis might be explained by an underlying noise reduction procedure of the camera.

Based on these observations we can divide the classification in two steps: (1) separate the pixels which do not carry enough color information, i.e., these which are too close to the gray axis. Figure 4.3 (middle) illustrates a simple segmentation of the gray pixels with a cone around the center axis illustrated by the red lines; (2) classify the color in the projection onto the uv-plane. Figure 4.3 (right) shows the uv-histogram without the gray pixels. Red lines illustrate the separated uv-segment which is classified as green. This way we ensure independence from brightness. The equation 4.3 illustrates the three conditions necessary for a pixel (y, u, v) to be classified as green. The five parameter are $b_o \in [0, 255]$ the back cut off threshold, $b_m, b_M \in [0, 128]$ with $b_m < b_M$ the minimal and the maximal radius of the gray cone, and finally $a_m, a_M \in [-\pi, \pi]$ defining the green segment in the uv-plane.

$$(u-128)^2 + (v-128)^2 > \max\left(b_m, b_m + (b_M - b_m) \cdot \frac{y - b_o}{255 - b_o}\right) \tag{4.1}$$

$$\text{atan2}(u-128, v-128) > a_m \tag{4.2}$$

$$\text{atan2}(u-128, v-128) < a_M \tag{4.3}$$

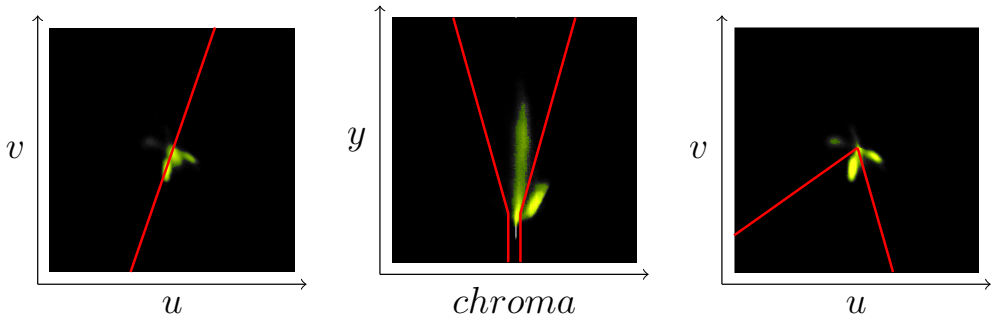


Figure 4.3: (left) UV-histogram for a log file taken at the Iran Open 2015. Red line illustrates the projection plane along the green region for the Y-Chroma histogram (middle); (middle) Y-Chroma-histogram along the projection plane illustrated in (left) figure. Red lines illustrate the gray-cone, i.e., area with not enough color information to be classified as a particular color; (right) UV-Histogram without pixel falling into the gray-cone as illustrated in the (middle) figure. Red lines illustrate the segment to be classified as green.

The classification itself doesn't require an explicit calculation of histograms. At the current state it's a static classification depending on five parameters to define the separation regions for the gray and green colors. These parameters can be easily adjusted by inspecting the histograms as shown in the Figure 4.3 and have proven to be quite robust to local light variation.

The structure of the color space depends of course largely on the adjustments of the white balance. We suspect a deviation from a perfect white balance adjustment results in a tilt of the gray cluster towards blue region if it's to cool and towards red if it's too warm. The tilt towards blue can be seen in the Figure 4.3 (middle). This might be a queue for an automatic white balance procedure which would ensure an optimal separation between colored and gray pixels. The green region shifts around the center depending on the general lighting conditions, color temperature of the carpet and of

course white balance. In the current example the green tends rather towards the blue region. Tracking these shifts might be the way for a fully automatic green color classifier which would be able to cover the variety of the shades to enable a robot to play outside.

4.2 ScanLineEdgelDetector



Figure 4.4: With top to down scanlines [green lines] the edges of possible field lines [black lines] including their orientation are detected (left) and the last field colored points are assumed as endpoints of the field [green circles] (right).

With this module we detect field line border points and estimate some points of the field border. To do this, we use scanlines, but only vertical ones. Along every scanline jumps are detected in the Y channel, using a 1D-Prewitt-Filter. A point of the field lines border is located at the maximum of the response of that filter. We estimate with two 3x3-Sobel-Filters (horizontal and vertical) the orientation of the line. With the result of the field color classification we detect along every scanline a point, which marks the border of the field.

4.3 FieldDetector

With the field border points, estimated with the *ScanLineEdgelDetector*, we calculate for each image a polygon, which is representing the border of the field in the image.

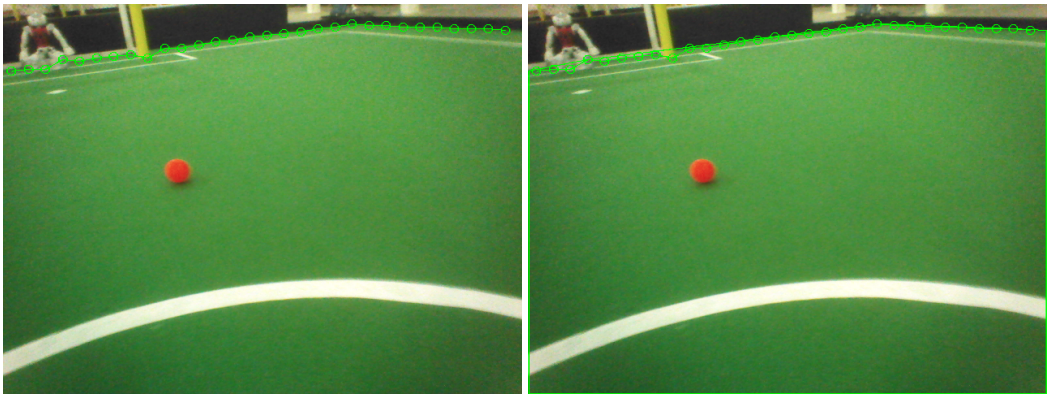


Figure 4.5: The endpoints provided by the *ScanLineEdgelDetector* (left) are used to calculate the field border (right).

4.4 LineGraphProvider

This module clusters neighbouring line border points, detected by *ScanLineEdgelDetector*.



4.5 RansacLineDetector

This module detects lines on the directed points generated by the LineGraph-Provider using the Random sample consensus (RANSAC) method. Our implementation can be summarized by the following 4 steps.

1. We pick two random directed points with a similar orientation to form a model of the Line. This will be a line candidate.

2. We count the inliers and outliers of the model. A point is marked as an inlier, if it is close to the line and has a similar direction. We accumulate the error of the distance for later validation.

3. If a model has a sufficient amount of inliers, it may be considered as a valid line. We repeat the above steps for a number of iterations and continue with the best model according to the amount of inliers and the smallest error in distance.

4. Finally, we determine the endpoints of the line. We choose them as the maximum and minimum x and y values of the inliers. If the length of the line is sufficient, a field line is returned.

In order to detect the remaining field lines we repeat this procedure on the remaining outliers until no lines are found.

4.6 GoalFeatureDetector

This module is the first step of the goal post detection procedure. To detect the goal posts we scan along the horizontal scan lines parallel to the artificial horizon estimated in *ArtificialHorizonProvider*. Similar to the detection of the field line described in Section 4.2 we detect edgels characterized by the jumps in the pixel brightness. These edgels are combined pairwise to goal features, which are essentially horizontal line segments with rising and falling brightness at the end points. Figure 4.6 illustrates the scan lines as well as detected edgels (left) and resulting goal post features (right).

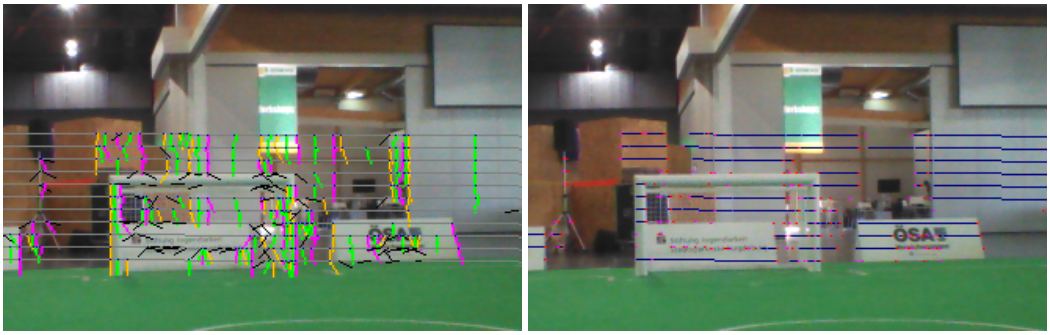


Figure 4.6: The scan lines [grey lines] above and below the estimated horizon are used to detect the goal post border points and the orientation of the corresponding edges [colored and black segments] (left). The results are features of possible goal posts [blue line segments with red dots] (right).

4.7 GoalDetector

The *GoalDetector* clusters the features found by the *GoalFeatureDetector*. The main idea here is, that features, which represent a goal post, must be located underneath of each other. We begin with the scan line with the lowest y coordinate and go through all detected features. Then the features of the next scan lines (next higher y coordinate) are checked against these features. Features of all scan lines, which are located underneath of each other, are collected into one cluster. Each of these clusters represents a possible goal post.

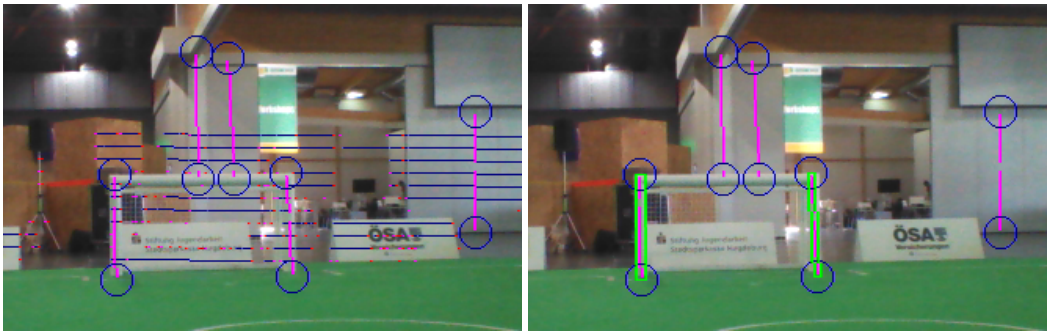


Figure 4.7: Goal features detected as described in 4.6 are clustered to form candidates for the goal posts (left). These candidates are evaluated regarding expected dimensions as well as their relation to the field. The candidates fulfilling all necessary criteria are selected as goal post percepts (right green boxes).

From the features of a cluster, the orientation of the possible goal post is estimated and used to scan up and down along the estimated goal post. This is done to find the foot and the top point of that goal post. A goal post is seen as valid, if its foot point is inside of the field polygon as described in the Section 4.3. Using the kinematic chain the foot point is projected into the relative coordinates of the robot. Based on this estimated position the expected dimensions of the post are projected back into the image. To be accepted as a goal post percept a candidate cluster has to satisfy those dimensions, i.e., the deviation should not exceed certain thresholds. The Figure 4.7 illustrates the clustering step and the evaluation of the candidate clusters. Although there seem to be a considerable amount of false features, both posts of the goal are detected correctly.

4.8 Black&White Ball Detection

In 2015 the standard ball used in competitions changed to a black&white foam ball as illustrated in Figure 4.8. Detection of such a ball in the SPL setup poses a considerable challenge. In this section we describe our strategy for detecting a black&white ball in a single image and the lessons learned.

Given an image from the robot camera the whole approach is basically divided into two steps: finding a small set of suitable candidates for a ball by a fast heuristic algorithm, and classifying the candidates afterwards with a more precise method.



Figure 4.8: Examples of the black&white as seen by the robot.

4.8.1 Candidate Search – Perspective Key Points Detection

Properties of the ball as an object that can be assumed as known include

- it has a fixed size;
- it has a round shape (symmetrical shape);

- it is black and white (so it does not contain other colors);
- the pattern is symmetric;

These properties have some implications on the appearance of the ball in a camera image:

- knowing the pose of the camera we can estimate the balls size in the image;
- it mostly does not contain other colors than black and white (chroma is low). Note: beware of reflections;
- looking only at the color distribution we can assume it's rotation invariant;

Having the limited computational resources in mind, we are looking now for following three things:

1. a simple object representation for a ball in image;
2. an effective and easy to calculate measure quantifying the likelihood such object to represent an actual ball;
3. a tractable algorithm to find a finite set of local minimas of this measure over a given image (these minimal elements we then call candidates);

Representation: In our case we define a *ball candidate* (more general a *key point*) as a square region in the image which is likely to contain a ball. Such a key point $c = (x, y, r)$ can be described by its position in the image (x, y) and its side radius (half side length) r. Basically we represent a circular object by the outer square (bounding box) enclosing the circle. This leads to a vast number of possible candidates in a single image and a necessity for an efficient heuristic algorithm to find the most likely ones.

Measure: Intuitively described, a *good* key point is much brighter inside than on its outer border. For a key point $c = (x, y, r)$ we define its *intensity value* $I(c)$ by

$$I(c) := \frac{1}{4r^2} \sum_{i,j=x-r}^{x+r} Y(i, j) \tag{4.4}$$

where $Y(j,i)$ is the Y-channel value of the image at the pixel (i,j). For the intensity of the outer border around c with the width $\delta > 0$ holds $I(c_\delta) - I(c)$, with $c_\delta := (x, y, r \cdot (1+\delta))$. Now we can formulate the measure for c by

$$V(c) := I(c) - (I(c_\delta) - I(c)) = 2 \cdot I(c) - I(c_\delta) \tag{4.5}$$

This measure function can be calculated very effectively using integral images. Figure 4.9 (left) illustrates the measure function for the valid pixels of the image.

Finding the local maxima: To save resources the search is performed only within the estimated field region (cf. Section 4.3). For a given point $p = (i,j)$ in image we estimate the radius $r_b(i,j)$ that the ball would have at this point in the image using the camera matrix. This estimated radius is used to calculate the measure function at this point: $V(p) := V((i,j,r_b(i,j)))$. Currently we only consider points at which the ball would be completely inside the image. The following algorithm illustrates how the local maxima of this function are estimated:

Algorithm 1: Find local maxima.

```
Data: Set of possible key points A
Result: List of locally maximal key points K
K ← ∅;
for p ∈ A do
    insert ← true;
    for q ∈ K do
        if overlaps(p,q) then
            if V(p) > V(q) then
                K ← K \ {q};
            else
                insert ← false;
            end
        end
    end
    if insert then
        K ← K ∪ {p};
    end
end
```

The basic idea is to keep only the key points with higher value than any other overlapping key points, i.e., the one with the highest value in it's neighborhood. To generate the list of *possible key points* we iterate over the image and generate key points for each pixel. Of course a number of heuristics is

used to make the process tractable. In particular we only consider every 4th pixel and only if it is within the estimated field region. The list of local maxima is also limited to 5 elements - the list is kept sorted and any additional key points with lower value are discarded. Figure 4.9 (right) and Figure 4.10 illustrate the detected best 5 local maxima of the measure function.

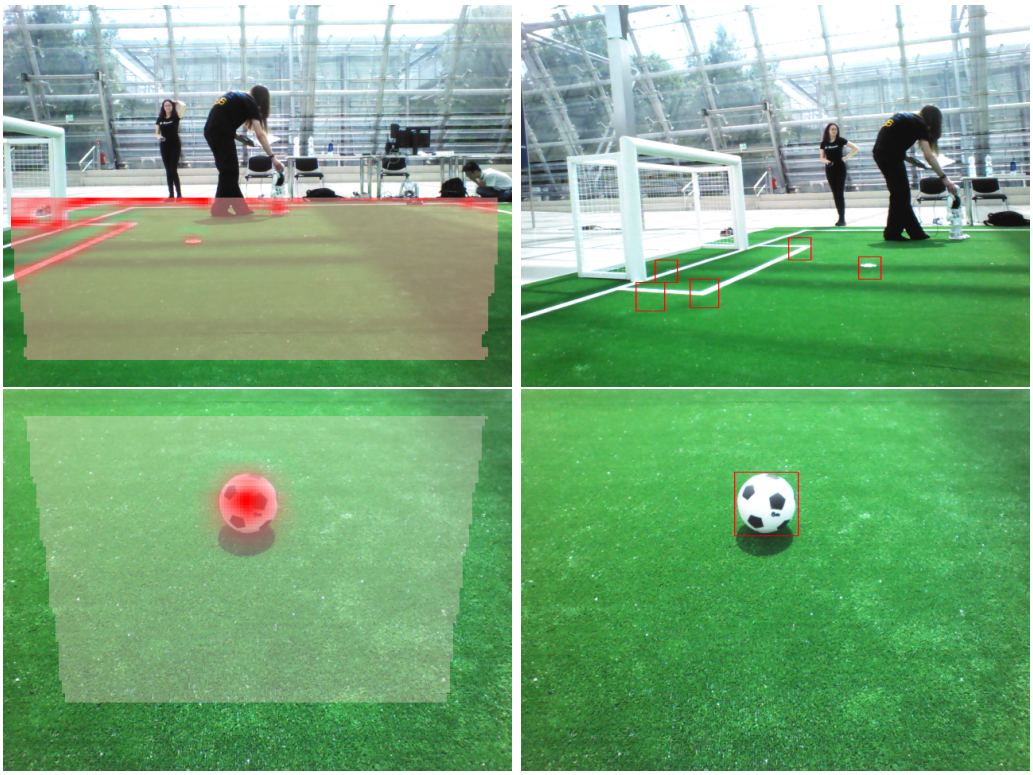


Figure 4.9: Illustration of the value function for finding the ball key points. Semitransparent overlay illustrates the searched area. Intensity of the red color shows the value function.

4.8.2 Classification

In the previous section we discussed how the number of possible ball candidates can be efficiently reduced. At the current state we consider about 5 candidates per image. In general these candidates look very ball alike, e.g., hands or feet of the robots, and cannot be easily separated. To solve this we follow the approach of supervised learning, which mainly involves collecting positive and negative samples, and training a classifier. In the following we outline our approach for fast collection of the sample data and give some brief

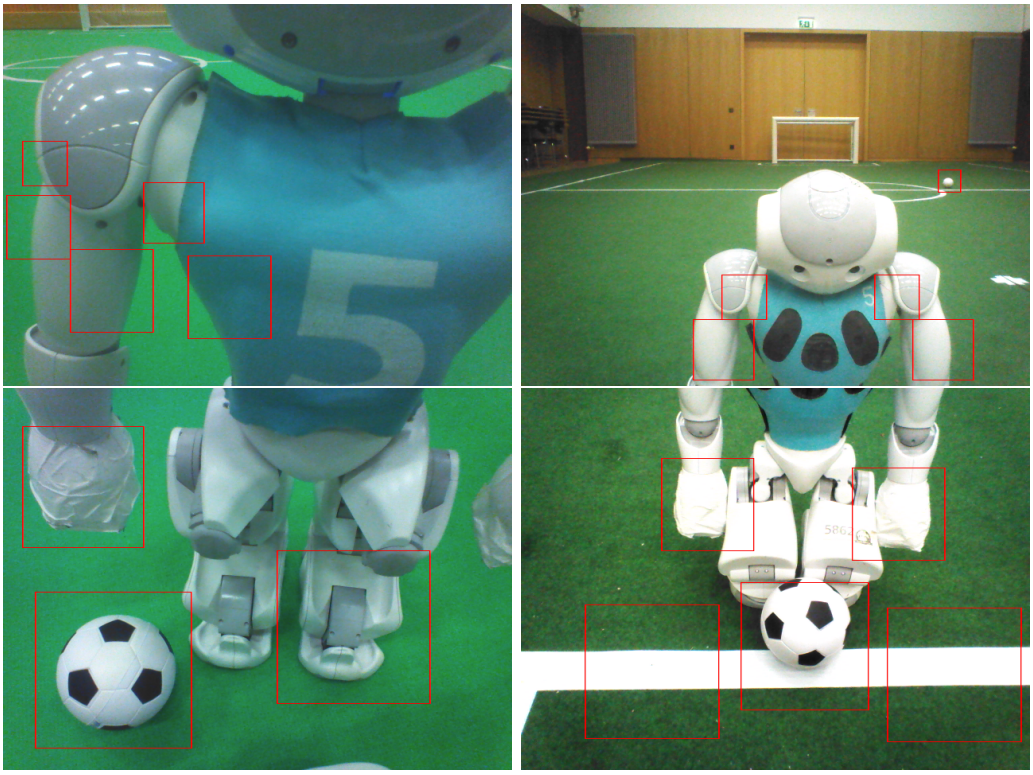


Figure 4.10: Illustration of detected key points in the situations where the ball is close to the robot or overlapping with the line.

remarks about our experience with the OpenCV Cascade Classifier which we used during the RoboCup competition in 2016.

Sample Data Generation

Collecting sample data basically involves two steps: collecting images from game situations and labeling the ones containing the ball. This can be a very tedious and time consuming task. To simplify and accelerate the process we collect only the candidate patches calculated by the algorithm in Section 4.8.1. This allows to collect the data during a competition game at full frame rate. This however produces a vast amount of data to be labeled. Because the patches are quite small (we experimented with sizes between $12px \times 12px$ and $24px \times 24px$), a large number of patches can be presented and reviewed at once. Figure 4.11 illustrates two examples of the labeling interface. Patches are presented in pages consisting of a 10×10 matrix. Labels can be applied or removed simply by clicking with the mouse at a particular patch.

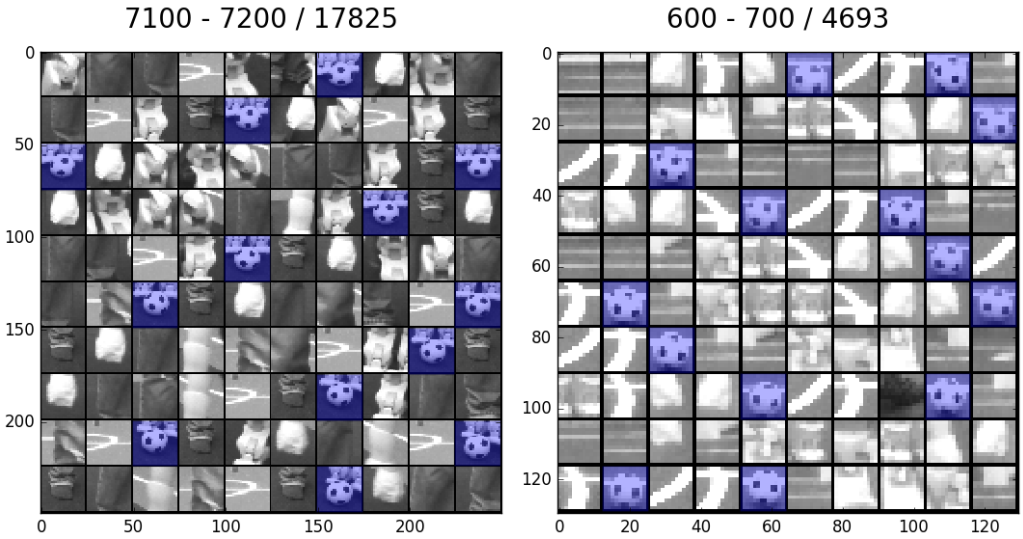


Figure 4.11: Examples of the labeling interface. Patches are presented in pages consisting of a 10×10 matrix. Labels are applied or removed by a mouse click at a particular patch.

Classification with Convolutional Neural Networks

In the last step the generated ball candidates (patches) are classified. At the current point we distinguish two classes: `ball` and `noball`. As a classifier we use a Convolutional Neural Network (CNN). The weights of the network are trained in MatLab with the help of the *Neural Network Toolbox*. The resulting network is exported with our custom export function `createCppFile.m` to optimized hard coded implementation of the network in C++. The generated files are included in the project and directly used by the ball detector. All corresponding functions can be found in the directory `Utils\MatlabDeepLearning`. Example of the resulting classification can be found in Figure 4.12.

4.8.3 Acknowledgment

Some of the most important parts of this ball detection procedure were inspired by very fruitful discussions with the RoboCup community. At this point we would like to thank in particular the team NaoDevils for providing their trained CNN classifier during the RoboCup 2017 competition. This classifier is included in our code base and can be used for comparison purposes. It can be found in
`...\VisualCortex\BallDetector\Classifier\DortmundCNN`.

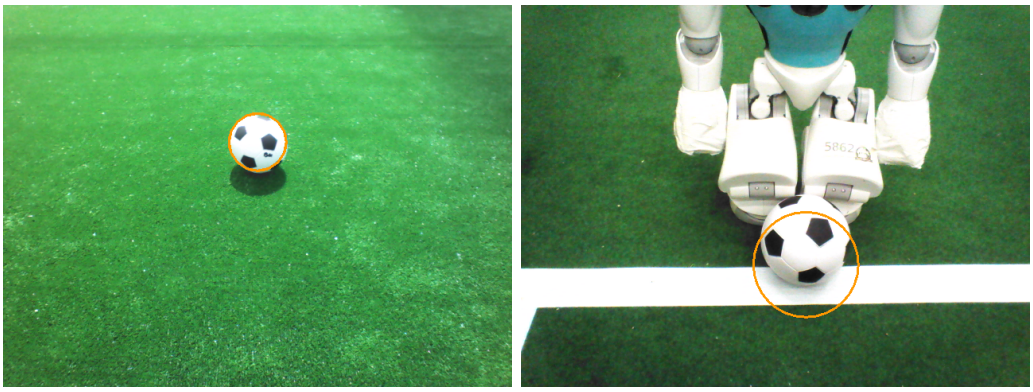


Figure 4.12: Examples of the detected ball.

Chapter 5

Modeling

In order to realize a complex and successful cooperative behavior it is necessary to have a appropriate model of the surrounding world. In our approach we focus on local models of particular aspects of the environment. In this section we present two local models: a compass and a goal model.

5.1 Camera Matrix Calibration

Camera matrix is the coordinate transformation of a camera in relation to the local coordinate system of the robot. The camera matrix is used to establish the relation between objects detected in the image and their position relative to the robot. For instance, the center of the detected ball in the image can be projected on the ground plane and so the distance to the actual ball can be estimated. In a certain way, the camera matrix stands between the basic perception and the model of the situation affecting directly the quality of self localization, ball model and such. Thus, an accurate estimation of the camera matrix is crucial for the robot's perception of its environment.

One way to estimate the camera matrix is the usage of the kinematic chain in combination with the accelerometer and gyrometer to estimate the rotation. This approach, however, can yield substantial errors, as the parameters of the kinematic chain differ between the robots due to differences in manufacturing and the effect of wearing out over time. In particular the following 11 joints have been observed developing major offsets: (body rotation pitch/roll, head rotation pitch/roll/yaw, top/bottom camera rotation pitch/roll/yaw).

In order to compensate for these errors we apply calibration offsets to these joints. To calculate the offsets we utilize the line percepts, which are provided by the LineGraphProvider described in section 4.4. The basic ap-

proach is to place the robot at a known position on the field and allow it to collect a set of line perceptions by looking around. Detected line perceptions in the image are projected on the ground plane using the camera matrix. The projected results are compared with the actual lines on the field. The resulting error can be minimized by adjusting the aforementioned offsets. It can be choosen between a simple Gauss-Newton and two Levenberg-Marquardt implementations as minimization algorithm. The current implementation supports a manual and automatic mode for the calibration procedure. Figure 5.1 illustrates the projection of the collected line perceptions before (left) and after (right) the minimization procedure.

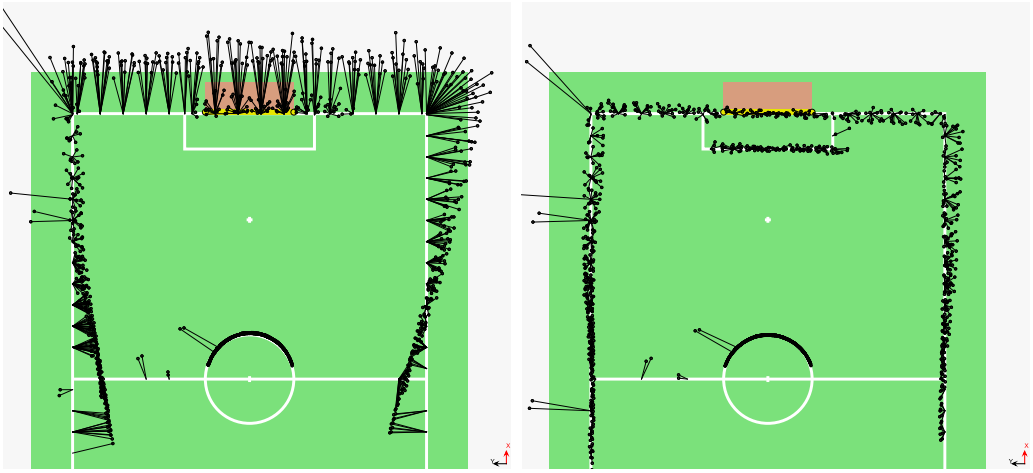


Figure 5.1: Projection of the line point perceptions before (left) and after(right) camera calibration. The assignments of the perceived line points to the field lines, which are used to calculate the error, are illustrated as black thin lines.

5.2 Probabilistic Compass

We estimate the orientation of the robot on the field based on the detected line edgels utilizing the fact, that all field lines are either orthogonal or parallel to the field. Based on the orientations of the particular projected edgels it is possible to estimate the rotation of the robot up to the π symmetry. We calculate the kernel histogram over the orientations of the particular projected edgels, i.e., edgels in the local coordinates of the robot. To utilize the symmetry of the lines we use sin as distance measure. Let $(x_i)_{i=1}^n$ be the set of edgel orientations. We calculate the likelihood $S(x)$ for the robot rotation

$x \in [-\pi, \pi)$ as shown in the equation 5.1.

$$S(x) = \sum_{i=1}^{n} \exp\left\{-\frac{\sin^2(2 \cdot (x - x_i))}{\sigma^2}\right\} \tag{5.1}$$

This compass is calculated in each frame where enough edgels have been detected. It has shown to be robust regarding outliers, e. g., when some edgels are detected in a robot. It can be directly used to update the likelihood of particles in the self locator. Figure 5.2 shows a set of edgels detected in a particular frame on the left side. On the right side the according histogram is plotted.

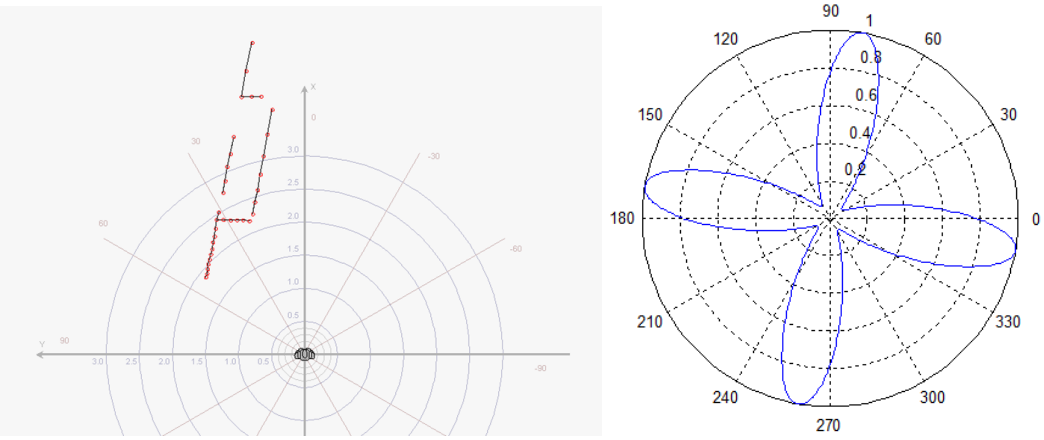


Figure 5.2: Left figure visualizes the edgel graph in local coordinates of the robot in a particular frame. Right illustrates the kernel histogram over the orientations of edgels shown left, calculated with formula 5.1.

5.3 Multi-Hypothesis-Extended-Kalman-Filter Ball Model

Although there is usually only one ball involved in a RoboCup game, there are several good reasons for being able to represent and track several ball hypotheses at the same time. The main reason however are the false-positives. Due to the change to the new black and white ball as described in the Section 4.8, the chance of a false positive became much higher. The most of the false detections appear only sporadically and do not persist over long time. Tracking several possible balls at the same time allows to effectively separate the true (persistent) detections from the false (sporadic) ones.

Each candidate (hypothesis) is tracked by an Extended Kalman filter. The Extended Kalman filter is used with linear state transition model and a nonlinear observation model. The state is defined as the location and velocity of the ball in the robot's local Cartesian coordinates while the measurement is taken as the vertical and horizontal angle in the camera image.

At first all hypotheses (ball candidates) are removed and not considered in the following steps which weren't updated for an amount of time and theirs variance in location became too high. Then the odometry of the robot since the last update is applied transparently to the states and covariances of the Extended Kalman filters so they stay in the robot's local coordinate system. Next, the current position and velocity of the ball candidates are calculated according to the linear state transition model. The friction between ball and carpet is modeled as negative acceleration in opposite direction to the current velocity and incorporated into the linear state transition model as a control vector. After that the ball candidates in the image are assigned to the hypotheses and the update is performed. Each measurement is assigned at most to one hypothesis and vice versa. If no matching hypothesis is found a new Kalman filter is created which will represent a new hypothesis. The final ball model for the behavior is the hypothesis which is the closest to the robot and is updated frequently.

5.4 Multi-Hypothesis Goal Model (MHGM)

In this section we describe a multi-hypothesis approach for modeling a soccer goal within the RoboCup context. The whole goal is rarely observed and we assume the image processing to detect separate goal posts. So we represent the goal by its corresponding posts. To reduce complexity of the shape of uncertainty we model the separate goal posts in local robot coordinates. The ambiguous goal posts are tracked by a multi-hypothesis particle filter. The actual goal model is extracted from the set of post hypotheses.

The joint uncertainty can be subdivided into *noise*, *false detections* and *ambiguity*. Each of these components is treated separately in our approach. The multi-hypothesis filter has to take care of noise and false detections, but it does not resolve the ambiguity of the goal posts. Instead, all occurring goal posts are represented by corresponding hypotheses and the ambiguity is solved on the next level when the goal model is extracted. Particle filters are great in filtering noise and are shown to be very effective for object tracking. To deal with sparse false positives we introduce a delayed initialization procedure. We assume a false positive to result in an inconsistency, i. e., it cannot be confirmed by any existing goal post hypothesis. In this case the percept

is stored in a short time buffer for later consideration. This buffer is checked for clusters, in case a significant cluster of goal post percepts accumulated during a short period of time, a new hypothesis is initialized based on this cluster. The dense false detections result in post hypotheses, which is later ignored while extracting the goal.

More detailed description of the algorithm as well as the experimental results can be found in [13].

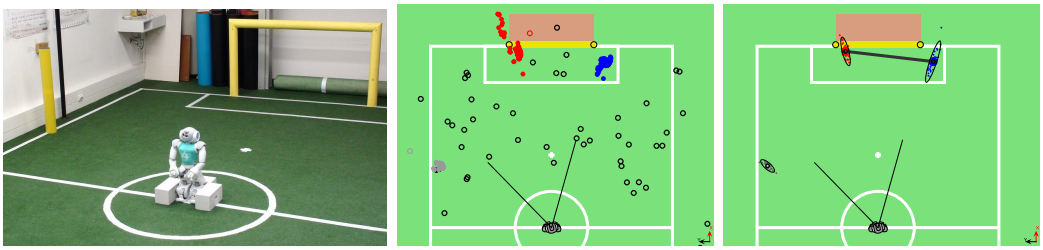


Figure 5.3: The left figure illustrates the experiment setup. The robot faces the goal and an additional goal post is placed to its left side. From the object recognition perspective, this post is identically to the *real* goal posts. The figure in the center visualizes all percepts collected during the course of the experiment. The full circles illustrate perceived goal posts, whereby their color indicates the classification by the MHGM: red - left post, blue - right post, gray - unknown post, black - none (percept buffer). The circles with holes stand for artificially generated sparse false positive perceptions. The right figure illustrates a snapshot of the state modeled by the MHGM at the end of the experiment. Drawn are the particle filter representing the goal posts with corresponding deviations as well as the extracted goal model. Similar to the figure in the center, the colors of the particles indicate the classification of the hypotheses.

5.5 Simulation Based Selection of Actions

The robot is capable of different kicks and should given a particular situation, e.g., the robot's position, the position of the ball and obstacles, determine which kick is the optimal kick to perform in this situation. A naive geometric solution which selects a kick based on the robot's direction towards the opponent goal does not account for uncertainty of the actual execution of the kick. Furthermore the distance of the kick is not considered in this approach. An improved kick selection algorithm was developed which is based on a forward simulation of the actions. Thereby each possible kick is simulated and

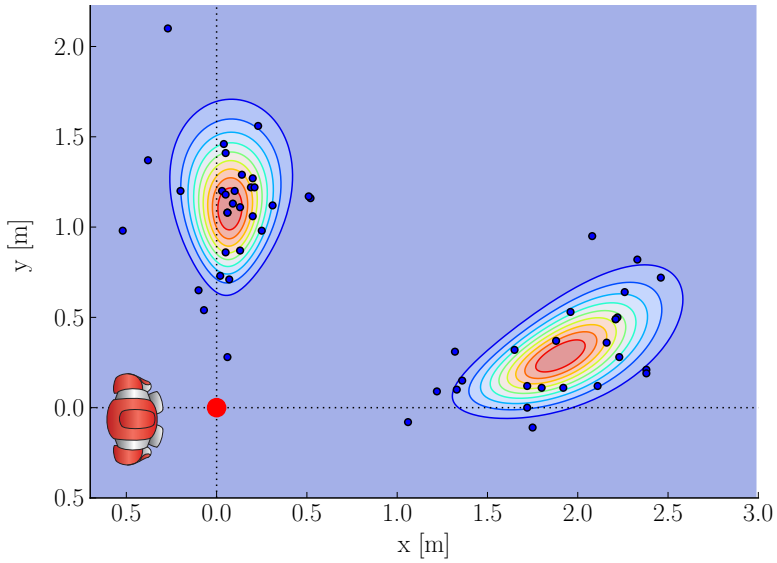


Figure 5.4: Kick action model: distributions of the possible ball positions after a sidekick and the long kick forward with the right foot. Blue dots illustrate experimental data.

the best kick is chosen based on the outcome, i.e., the position of the ball after the kick. Uncertainty and additional constraints can be integrated in a straight forward way.

5.5.1 Definition of an Action

An Action is a set of parameters which describe a probability distribution of the possible ball location after the execution of a kick. Currently there are 4 kicks, two forward kicks and two sidekicks as well as the case of turning around the ball. The probability distribution is modeled as a Gaussian distribution. The parameters which describe the distribution for one action are velocity, angle and their standard deviations.

5.5.2 Determine the parameters

To calculate the initial velocity of a kick the distance the ball rolled after a kick was measured in an experiment. By using the stopping distance formula the initial velocity of one kick can be calculated by

$$v_0 = \sqrt{d \cdot 2c_R \cdot g} \tag{5.2}$$

where v is the initial velocity of the ball. c_R the rolling resistance coefficient and g the gravitational constant. The mean of v_0 of multiple repetitions defines the initial velocity of this action. The standard deviation of the repetitions defines the standard deviation for the velocity of the kick. The parameter for angle is predefined for every action, e.g., it's zero for forward kicks and 90 degrees for left sidekick. The standard deviation for the angle is the standard deviation of the angle measurements from the previous experiment. The coefficient of friction is calibrated to a real surface from rolled distances of the ball rolling on this surface with a known initial velocity. For this, we performed multiple experiments with an inclined plane starting at different heights. From these heights we could determine the initial potential energy of the ball, which was converted to kinetic energy by rolling down the inclined plane. At the end of the inclined plane (not taking into account the friction of the inclined plane), the initial velocity of the ball could thus be determined. We then measured the distance in multiple experiments. By transposing the rolling distance formula the rolling resistance coefficient can be calculated.

$$c_R = \frac{1}{2} \cdot \frac{v_0^2}{g \cdot d} \tag{5.3}$$

where v_0 is the starting velocity, g the gravitational constant, and d the total distance the ball traveled. The mean of the calculated coefficients is used as the rolling resistance coefficient for the other calculations. In the algorithm the position of the ball after the execution of an action is needed. To calculate this, the formula is transposed to calculate the distance the ball rolls after the execution of an action:

$$d = \frac{v_0^2}{2c_R \cdot g} \tag{5.4}$$

where v is the initial velocity of the ball. c_R the rolling resistance coefficient and g the gravitational constant. Figure 5.4 shows a resulting end position cloud of a hypothetical kick. The end points are calculated by drawing a sample from both the angle and kick speed distribution and plugging these values in equation 5.4. For detail, refer to section 5.5.3.

5.5.3 The algorithm

The whole simulation is divided into three steps: simulate the consequences, evaluate the consequences and decide the best action.

Simulating the consequences

Each action is simulated a fixed number of times. The resulting ball position of one simulation for an action is referred to as particle. The positions of the particles are calculated according to the parameters of the action with applied standard deviations as shown in figure 6.3. The algorithm checks for possible collisions with the goal box and in case there are any the kick distance gets shortened appropriately. Collisions with the obstacle model are handled the same way.

Evaluation

Each particle is sorted in different categories based on where on the field it is, e.g., inside the field, inside the own Goal, outside the field. If a particle lands outside the field it is sorted in the category according where it went out, e.g., left sideline or opponent ground line. This is repeated for every particle of every Action that is defined. The algorithm then counts the number of particles of each action that is either inside the field or inside the opponent goal.

Decision

If an action has less than the defined threshold of particles either inside the field or inside the opponent goal the action is discarded. For the remaining actions the one with the most particles inside the opponent goal is calculated. If there are two actions with the most particles inside the goal the best action is determined by evaluating the particles of each actions with the potential field. The action with the smaller sum is selected. If at least one particle of an action is inside the own goal the action will not be chosen. If no action has a particle inside the opponent goal the potential field is used to rank the actions. In this case all particles from one action are evaluated by the potential field and the mean of these values is calculated. The action with the highest mean is selected and executed. If no action has enough good particles, the best action is to turn towards the opponent goal.

5.5.4 Potential field

A potential field assigns a value to each position of the ball inside the field. The values reflect the static strategy of the game and are used to compare possible ball positions in terms of their strategic value. For instance, the position a meter away in front of the opponent goal is obviously much better

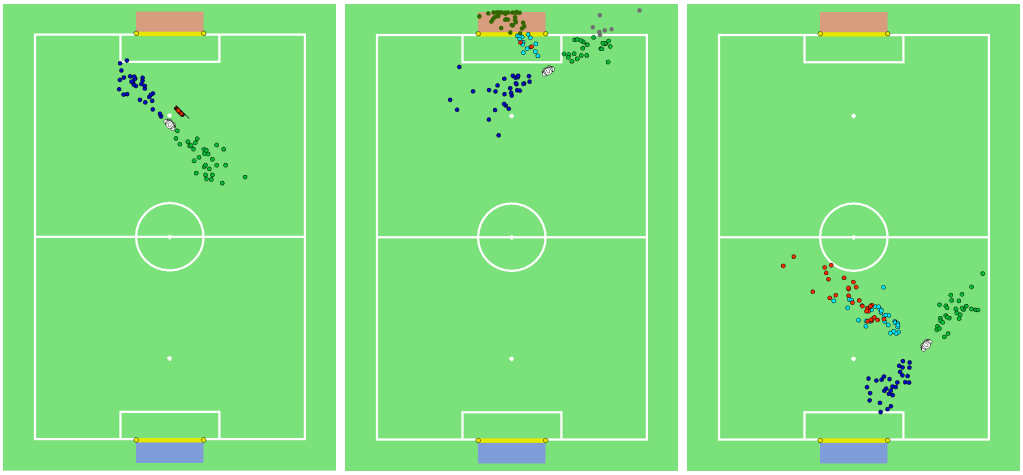


Figure 5.5: Three examples of kick simulations. Each possible kick direction is simulated with 30 samples (different colors correspond to different kicks). Left: the short and long kicks are shortened due to collision with an obstacle. Middle: long kick is selected as the best action since it has the most samples result in a goal. Right: the best action is sidekick to the right – the other kicks are more likely to end up in a dangerous position for the own goal according to the potential field.

than the one in front of the own goal. In our experiments we use the following potential field:

$$P(x) = \underbrace{x^T \cdot \nu_{\text{opp}}}_{\text{linear slope}} - \underbrace{N(x|\mu_{\text{opp}}, \Sigma_{\text{opp}})}_{\text{opponent goal attractor}} + \underbrace{N(x|\mu_{\text{own}}, \Sigma_{\text{own}})}_{\text{own goal repulsor}}, \tag{5.5}$$

where $N(\cdot|\mu, \Sigma)$ is the normal distribution with mean μ and covariance Σ. It consists of three different parts: the linear slope points from the own goal towards the opponent goal and is modeling the general direction of attack; the exponential repulsor $N(x|\mu_{\text{own}}, \Sigma_{\text{own}})$ prevents kicks towards the center in front of own goal; and $N(x|\mu_{\text{opp}}, \Sigma_{\text{opp}})$ creates an exponential attractor towards the opponent goal. The configuration used in our experiments is

$$\nu_{\text{opp}} = (-1/x_{\text{opp}}, 0)^T \tag{5.6}$$

with $x_{\text{opp}} = 4.5$ being the x-position of the opponent goal and

$$\mu_{\text{own}} = (-4.5, 0) \qquad \mu_{\text{opp}} = (4.5, 0) \tag{5.7}$$

$$\Sigma_{\text{own}} = \begin{pmatrix} 3.375^2 & 0 \\ 0 & 1.2^2 \end{pmatrix} \qquad \Sigma_{\text{opp}} = \begin{pmatrix} 2.25^2 & 0 \\ 0 & 1.2^2 \end{pmatrix} \tag{5.8}$$

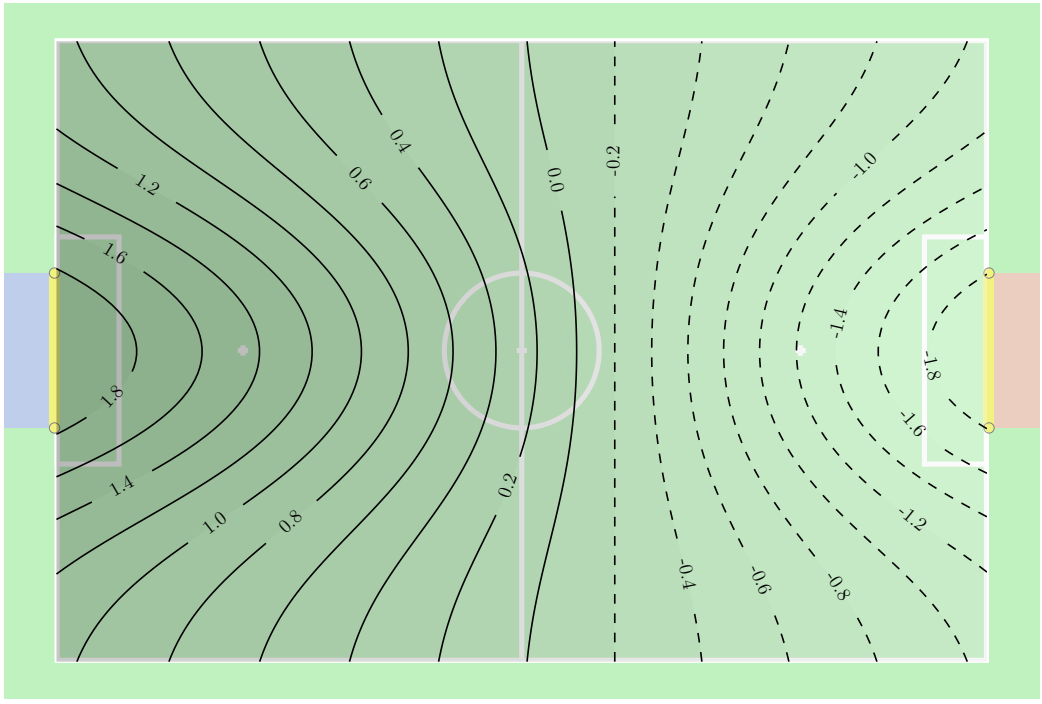


Figure 5.6: Strategic potential field evaluating ball positions. Own goal is on the left (blue).

for the repulsor and attractor respectively. All parameters are of unit m. Figure 5.6 illustrates the resulting potential field. A more detailed description of this simulation can be found in [10]. A continuation of this work was presented at the HUMANOIDS 2017 RoboCup Workshop [9].

5.6 Arm Collision Detection

In the current implementation of the stand and walk motions the arms are used for stability and energy efficiency. During the stand motion the arms are kept down along the sides to minimize energy consumption, and are moved back and forth during the walk to balance the rotational forces and stabilize the walk. On the downside, these arm poses effectively enlarge the frontal silhouette significantly increasing the chance of collision with other players or goal posts, which may destabilize the robot and lead to a fall or a penalty for illegal pushing.

The goal of collision detection is to recognize when robots arms collide with an external object. When a collision is detected, the robot reacts with an evasive movement by taking the arms back and so effectively minimizing its silhouette. This allows the robot to "sneak" through narrow spaces without falling or pushing others. The earlier a collision is recognized, the more effective will be the avoidance. However, a false detection will force the robot to take the arms back unnecessarily destabilizing it.

A collision with an arm can be identified by observing the difference in the planned joint position (MotorJointdata) and the measured joint position (SensorJointData) of the LShoulderPitch and RShoulderPitch joints. These joints have the largest leverage and are affected the most when a collision on an arm occurs.

In the current implementation we use a straight forward approach. A collision is detected when the moving average of the absolute error in the joint position rises above a fixed threshold. The moving average is calculated over a fixed window of 100 frames, i.e., 1 second. Note that there is a delay of 4 frames, i.e., 40 ms, between the joint position command and the corresponding measurement. This delay needs to be taken into account when calculating the error in the joint position. To determine the threshold a number of experiments with collisions was recorded. The threshold was calculated as a maximum of the moving average error over all experiments, with a bit of tolerance. Together with the delay induced by the moving average filter, this results in a conservative detector.

Figure 5.7 shows the accumulated absolute error for both arms. During the experiment the robot goes from stand to walk and a collision occurs at

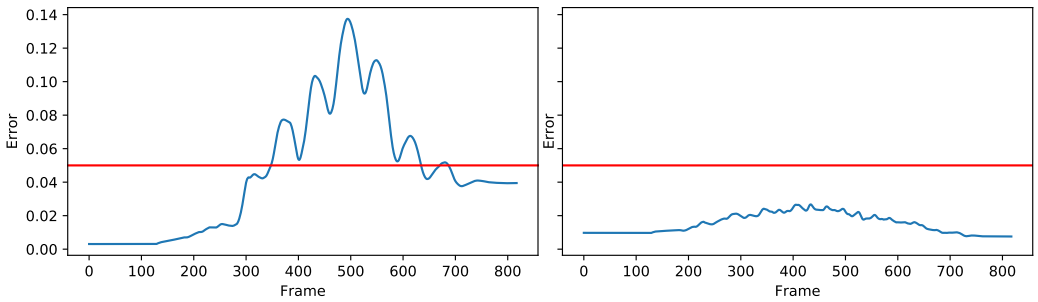


Figure 5.7: Moving average of the absolute joint error during a collision on the left arm. The error of the left shoulder pitch (LShoulderPitch) is shown on the left, and the error of right shoulder pitch (RShoulderPitch) on the right. Red line illustrates the threshold for the detection of a collision.

the left arm. It can be clearly seen that the error for the left arm rises above the threshold (red line) when the collision occurs, while the error for the right arm (the arm without collision) stays well below the threshold during the entire time.

Despite its simplicity, this approach has shown good results during the RoboCup competition in 2017 and was able to recognize most of the collisions. In the current work we focus on a more sensitive and accurate detector. To avoid false detections, the static threshold has to be chosen larger than the largest expected error which might occur with out a collision. The accuracy can be improved with a dynamic threshold. We observed a correlation between the position of the joint the average absolute error which. Thus, a threshold depending on the position of the joint might improve the results significantly.

5.7 Time synchronization

The robots communicate via wifi with the gamecontroller and with each other. The communicated information are not only used for debugging & visualization in RobotControl, their're also used in the behavior and role decision of each player. Because of the different wifi conditions in the lab, during competitions and other events, the robots couldn't be very confident how reliable the provided information are. So far the robots couldn't determine, when the data was sent and therefore how old the received data really is. For this reason it was necessary to implement a synchronization primitive in order to account for the transmission latency, which could be as fast as 20ms, but also could took up to 1s. The idea of the time synchronization

module is based on the „simple network time protocol“, which is a simplified version of NTP. In contrast to the standard SNTP, we're using extra data fields in our teamcomm message, instead of a low level network format. Further we doesn't use a server, centralized software instance, instead the robots themselves deciding to which other robot they want to synchronize with and sending them the appropriate information. That results in a decentralized simple network time protocol. The whole process is based on timestamps in ms and works as follow:

- a robot randomly selects a teammate he wants to synchronize with
- he creates a ntp request which contains the player number of the synchronizing partner, the timestamp of the last message the partner send and the timestamp the robot received the last message.
- after some time, the robot should get a similar request from its teammate and can calculate with the given information the total round-trip-time (RTT) and the latency for one direction
- the fastest transmission cycle is then used to calculate the time offset between the robot and its teammate
- the offset can then be used to provide an estimated time of the teammates for other modules

Together with the estimated time of the teammates and the timestamped message data, each robot is able to determine how „old“ the received data of its teammates are. This synchronization primitive is not as accurate as the standard NTP, but it is reasonably good and much better than a manually chosen latency or no synchronization at all. Currently only the teamball module uses the time synchronization to determine how old a seen ball of its teammates really is, but future development should use the feature more and thus make the communicated information more reliable and the thereon based calculations more accurate.

5.8 IMU

A Unscented-Kalman-Filter is used to combine the measurements of the accelerometer and gyrometer to a orientation and a gravity adjusted acceleration in the inertial frame. Quaternions are used in calculations and the orientation is represented as a rotation vector (Euler vector) in the state of the Unscented-Kalman-Filter to improve the numerical stability and avoid the problem of singularities.

Chapter 6

Motion Control

The performance of a soccer robot is highly dependent on its motion ability. Together with the ability to walk, the kicking motion is one of the most important motions in a soccer game. However, at the current state the most common approaches of implementing the kick are based on a key frame technique. Such solutions are inflexible and take a lot of time to adjust the robots position correctly in order to use a key frame motion effectively. Moreover, they are hard to integrate into the general motion flow, e.g. to change between walk and kick the robot usually has to change to a special stand position.

Fixed motions such as keyframe nets perform well in deterministic environments, but they are restrictive. More flexible motions must be able to adapt to different conditions. Four possible specifications are adaptation to control demands, e.g. required changes of speed and direction to enable omnidirectional walk, adaptation to the environment, e.g. different floors with different heights and angles, kick execution according to ball state and fluent change between walk and kick.

At the current state we have a stable version of an omnidirectional walk control and a dynamic kick, both used in our gameplay. Along with further improvements of the dynamic walk and kick motions our current research focuses in particular on *integration* of the motions, e.g. a fluent change between walk and kick.

Adaptation to changing conditions requires feedback from sensors. We experiment with the different sensors of the NAO. Especially, adaptation to the visual data, e.g. the seen ball or optical flow, is investigated. Problems arise from sensor noise and delays within the feedback loop. As a correlated project we also investigate the paradigm of local control loops, e.g. we extended the Nao with additional sensors.

6.1 Walk

The algorithm we use to accomplish a walking motion can be subdivided into four components: the path planner, the step planner, the preview controller and stabilization.

The path planner is currently of very primitive nature, acting solely as a bridge for certain movement routines between behavior and motion control. The behavior part is presented in chapter 7.

At first, the step planner determines the target position for the next step considering the walk request and various stability criteria. After that, a sequence of desired ZMPs (zero moment points) is planned for each execution cycle of that step. This sequence of ZMPs is used by the preview controller to compute the trajectory of the COM (center of mass) during the execution of the step assuming a linear inverted pendulum model. While the step is executed the foot's 3D trajectory is calculated on demand and combined with the corresponding COM pose to finally determine the target joint configuration using inverse kinematics. Figure 6.1 shows a overview of the walking engine.

6.1.1 Path Planner

The path planner is intended to calculate a collision free and optimal path from the robot towards a goal. Currently that intention is not fulfilled. Instead it is used as a bridge between behavior described in chapter 7 and motion control. The behavior requests a certain routine, e.g. that the robot walks to the ball, and the path planner executes that routine by requesting steps from motion control. The types of routines that are implemented inside the path planner are walking or kicking routines.

Currently, there are two different interfaces to walk requests, i.e. standard and step control. A standard walk request consists of a target pose (x, y, θ), a frame of reference for the target pose in form of a local coordinate with its origin in the left foot, the right foot or the hip and a flag signaling fast, normal or slow step execution. Step control extends the standard interface by adding the following parameters: the target foot, the execution time in ms, an angle d that impacts the final trajectory and $s \in [0, 1]$ which can scale the final trajectory resulting in faster execution. The flag in the standard interface maps itself to s.

Step control allows for more fine-grained control over the actual steps, e.g. allowing retraction of the kicking foot after kick execution, and is thus preferred, but the standard interface is kept for backwards compatibility as not all possible movement routines related to walking are implemented

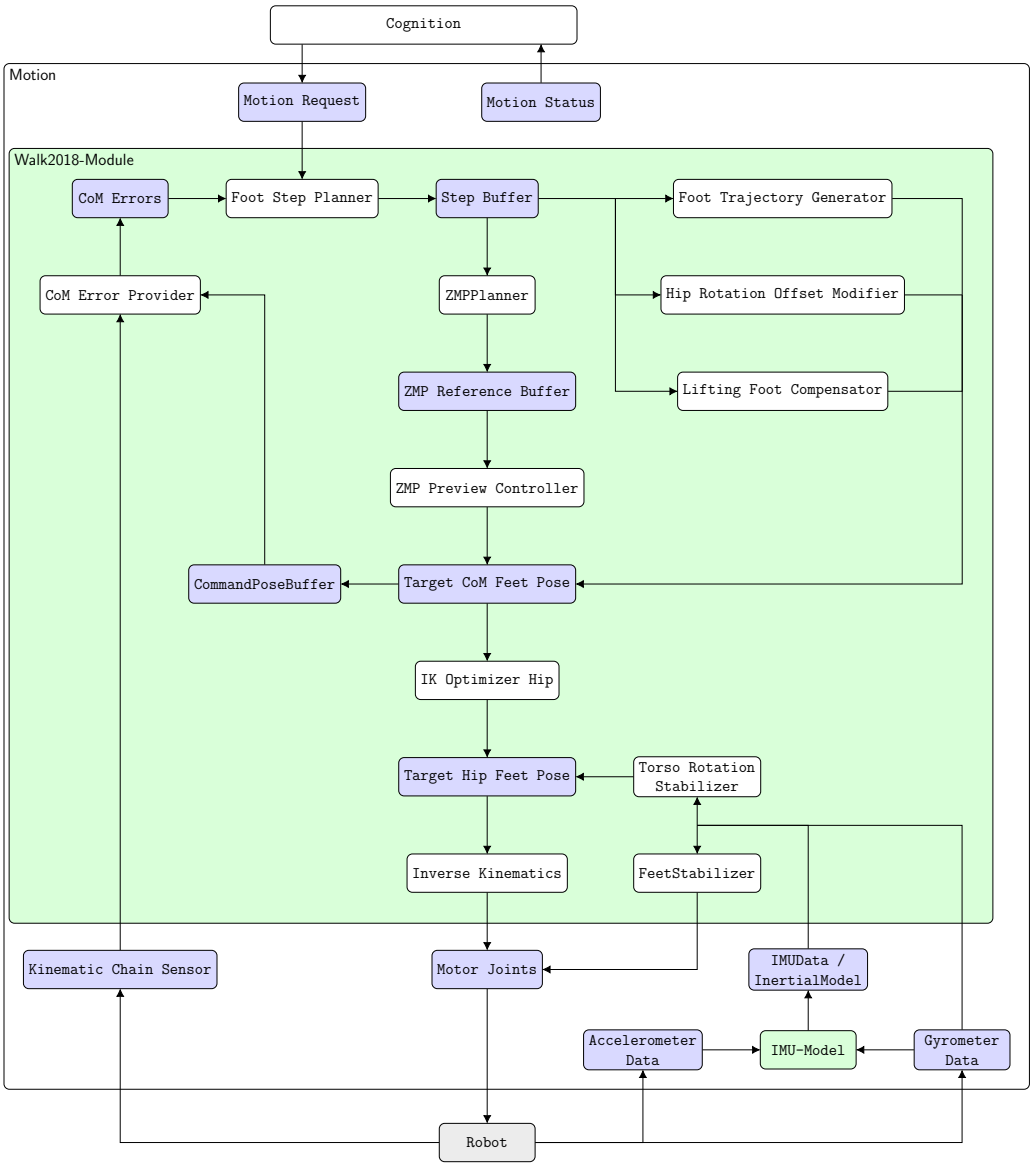


Figure 6.1: A closer look on the currently used walking engine.

inside the path planner. To allow this fine-grained control over actual steps the path planner implements a simple mechanism to ensure that a desired step is requested until actual execution of the step, unless the step becomes obsolete in the process.

6.1.2 Step Planner

The step planner calculates the next 2D positions of the feet based on the walk request inside the motion request.

The walk request and the current pose of the moving foot are transformed into the local coordinate system of the supporting foot. The supporting foot is the one that isn't executing the current step. The local coordinate system of the supporting foot is chosen for this as it is static whilst the step is being executed. This is not the case for the moving foot or the hip. The walk request is then applied to said current pose of the target foot resulting in the target pose for the step.

If the walk request was made using the standard interface the step planner is responsible for choosing the moving foot. Step control requests a specific foot explicitly.

The requested steps are restricted in regard of anatomic constraints as well as to increase the walks stability. A step is restricted elliptically in x-y-plane in general. The normal steps final dimensions are scaled by the cosine of the requested rotation. A big rotation therefore results in a small translation. In addition, the change in the step size is also restricted to increase stability. This prevents the robot to begin walking with the maximal possible step size. After applying these restrictions the step is finally added to the step buffer.

Independent of the requested steps the step planner might insert zero steps for increasing the stability of the walk. A zero step is a step in which no foot is moved.

6.1.3 Preview Control

The Preview Controller calculates the trajectory for the COM based on planed ZMPs. For estimating a stable trajectory for the COM we assume a linear inverted pendulum model with constant height. In each planning cycle of a step a target ZMP is added to the ZMP-buffer. The ZMP-buffer is used by the preview controller to calculate the target position, velocity and acceleration of the COM during a step. The following equation is used to

determine the control vector [15]:

$$u = - \underbrace{K_x x_k}_{state feedback} - K_I \underbrace{\sum_{i=0}^{k} (p_k - p_k^{ref})}_{accumulated ZMP error} - \underbrace{[f_1, f_2, \ldots, f_N]}_{preview\ gain} \underbrace{\begin{bmatrix} p_{k+1}^{ref} \\ p_{k+2}^{ref} \\ \vdots \\ p_{k+N}^{ref} \end{bmatrix}}_{future ZMP} \quad (6.1)$$

Where x_k is a vector describing the location, velocity and acceleration of the COM at time k. p_k is the ZMP and p_k^{ref} the target ZMP at time k. K_x, K_I and $f_1, \cdots, f_N$ are the parameters of the preview controller and are pre-calculated. The next target COM x_{k+1} can be calculated using a linear motion model:

$$x_{k+1} = Ax_k + ub \quad (6.2)$$

6.1.4 Stabilization

The simplified model can easily be affected by disturbances in the environment. Therefore a closed loop stabilization is required.

Different control techniques are used during step creation and execution to accomplish a stable walk.

During step creation the target step is adapted by a P-D-Controller mechanism to compensate small errors in the COM's position. Another mechanism uses the average COM-Error. If the average COM-Error exceeds a threshold an emergency stop is performed. This emergency stop is realized by zero-steps. As long as the COM-Error doesn't drop below a threshold the robot won't execute a step which is requested by a Walk-Request.

During the execution of a step three stabilization mechanisms are used. At first the height of the hip and its rotation around the x axis are adapted to compensate the moments appearing while a foot is lifted. A second stabilizer tries to keep the upper body in an upright position the whole time. And a third controller adapts the ankles according to the current orientation of the robot's body and its change in orientation.

6.2 Energy Efficient Stand

During games we have to deal with two problems regarding the hardware of the robots. The first problem is the increasing temperature of the joints, which affects the stability of walking. The second problem is the overall power consumption limiting the operational time. We observed that the

robots are standing a lot on strategic positions during the game causing high energy consumption. The core of the problem seems to be that when going to the stand pose the joints are never reached completely and so remain in a state of permanent tension. In particular this can happen when the last step before stand was not completed exactly and the feet are a bit shifted. Therefore to address both problems we try to reduce the energy consumption and temperature increase during standing.

After the robot reached the target standing pose the *measured* joint positions are used as new target joint angles to ensure that each joint really reached the target position and thus relaxing the joints. Reducing the applied stiffness on the motors will result in a reduction of the applied current and so reduce temperature increase and the energy consumption. Additionally we try to use as less stiffness as possible while maintaining a posture close to the target standing pose. To achieve that the stiffness is linearly interpolated between 0.3 and 1 depending on the joint angle error for each joint.

The knee pitch and ankle pitch joints are the joints which have to carry most of the load. It was observed that in some cases the applied current can be reduced significantly if the target position of these joints is relaxed by the minimal step size of the motors. Therefore an offset is added to the joint positions. Every second the offset of the joint with the highest current consumption is relaxed, i.e., increased (for knees) or decreased (for ankles), by the minimal step size of the motors.

Both energy saving approaches may result in a drifting of center of mass. So if the difference to the target center of mass becomes too large regarding translation and rotation the offset are reset and the standing posture is corrected again with full stiffness.

Chapter 7

Behavior

The Extensible Agent Behavior Specification Language — *XABSL* cf. [8] is a behavior description language for autonomous agents based on hierarchical finite state machines. XABSL is originally developed since 2002 by the *German Team* cf. [7]. Since then it turned out to be very successful and is used by many teams within the RoboCup community. We use XABSL to model the behavior of single robots and of the whole team in the Simulation League 3D and also in the SPL.

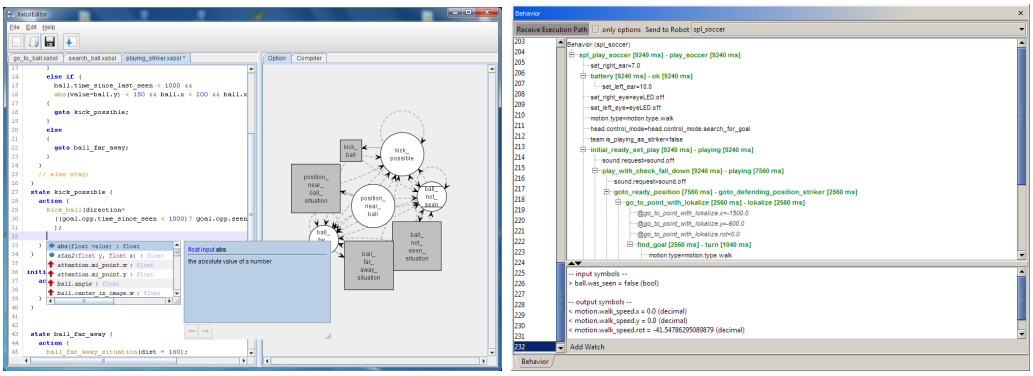


Figure 7.1: (left) XabslEditor: On the left side, you see the source code of a behavior option. On the right side the state machine of this option is visualized as a graph (right). In the main frame the execution path is shown as a tree; at the bottom, some monitored symbols can be seen, the developer can decide which symbols should be monitored; On the left side, there is a list of buffered frames, which is very useful to see how the decisions changed in the past.

In order to be platform independent, we develop our tools in Java. In particular we are working on a Java based development environment for

XABSL, named *XabslEditor*. This tool consists of a full featured editor with syntax highlighting, a graph viewer for visualization of behavior state machines and an integrated compiler. Figure 7.1 (left) illustrates the *XABSL Editor* with an open behavior file.

Another useful tool we are working on is the visualizer for the XABSL execution tree, which allows monitoring the decisions made by the robot at runtime. At the current state, this visualizer is part of our debugging and monitoring tool *RobotControl*. Figure 7.1 (right) illustrates the execution tree of the behavior shown within the visualizer.

7.1 Team Strategy

Our team strategy is based on the notion of the *active* and *passive* behavior. Active behavior defines for each robot what is to do if the robot is at the ball, while passive behavior describes what is to do if the particular robot is not at the ball. A robot in the active state, i.e., handling the ball, is also referred to as *striker*.

Essential part of the passive behavior are the *home positions*. These are fixed positions assigned to each player to which the robot eventually returns if it is in the passive state. This is important in order to keep players at the strategically important positions. At the current point the assignment of players to the passive positions is fixed based on the player number. The start and kickoff positions which robot assume during the ready phase before the game are chosen in the way to minimize the overlapping of the paths of the robots and the distance to be walked. Figure 7.2 illustrates the different formations - start positions (yellow), kickoff positions (blue) and home positions (red) - as well as the player assignment.

The task of a robot in the active state is to handle the ball, i.e., approach it and move it towards the opponent goal. We use a simulation based approach to chose the best action/kick to perform as described in the Section 5.5. In the ideal case there is exactly one player in active state (striker) at all times. We use a team communication protocol to ensure the player with the best chance to get the possession of the ball is in the active state. The details of the negotiation are described in the Section 7.2.

The passive and active behaviors are the same for all field players differing only in individual home positions, while goalie has specialized behavior. A small improvement for the goalie this year was, that the goalie tries to block pro-actively a direct goal kick of an opponent to the center of the goal. Therefore, if the goalie sees the ball and the ball is between the penalty cross and half line, the goalie position itself on the crossing between an oval around

the goal and within the penalty area and a straight line from the goal center to the ball.

Please refer to the code release for further details of the particular XABSL behaviors for different states (active, passive, goalie) etc.

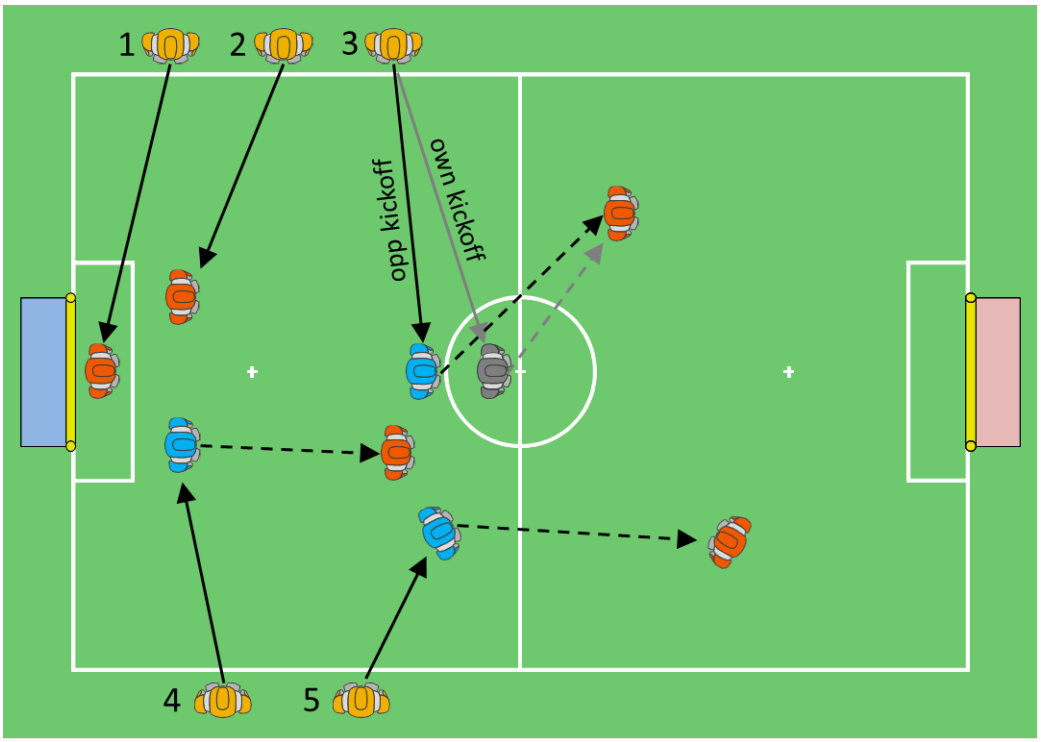


Figure 7.2: Illustration of the different strategic player position: start positions (yellow), ready positions during opponent kickoff (blue), during own kickoff (gray) and home positions (red). For players 1 and 2 all positions are the same respectively. Player 3 is the only one with a different ready position during own kickoff. Solid arrows illustrate the assignment of ready positions to the players. Dashed arrows illustrate the transitions from passive kickoff positions to the corresponding passive positions.

7.2 Role Change

The task of the role decision mechanism is to determine who is striker and who's passive. In general there should only be one robot which acts as striker. This year we loosened the „only one striker“ condition in order to handle wrong perceptions of a ball. So there can be now two striker for two

„different“ balls. All others are passive, means their're taking a predefined position based on their player number and observing the ball.

The goalie is excluded from the general role decision process in terms of how he decides to become striker. His decision is based on how close the ball is to the own goal - independent from other players. In case the goalie decides to be striker, all the other player become passive - or, if one sees a different ball than the goalie, he can get „second striker“.

The decision process which role should be used, runs every cognition cycle and is based on four data points.

1. the previous decision (wants to be striker and was striker)
2. the state of the robot (active, fallen, penalized, etc.)
3. time to the ball
4. player number

Therefore we communicate every ∼500ms the SPL standard message and the following informations:

- **is penalized**
  states if the robot is penalized and by this is “inactive“
- **is/was striker**
  describes if the robot decided to be striker and executed the striker behavior
- **wants to be striker**
  describes whether or not the robot is able to play, sees and is near the ball and therefore wants to be striker
- **time-to-ball**
  an estimation of how fast the robot can approach the ball; for that the robot takes into account his walking, turning and standup speed, if there's an obstacle between his position and, if the ball is moving, whether the ball can be intercepted by the robot.

Every robot calculates the role decision for every team member (including himself) based on the communicated information. With that we should get a consistent distribution of roles throughout the team. We have currently up to four different methods for the role decision process. In the last year we've implemented a new variation and successfully used it in this year competitions.
The decision process is executed in two steps:

1. every robot decides if he wants to be striker and announces it to his teammates as “wants to be striker“
    - the striker decision is based on whether the robot is in an “active“ state. That means, the robot isn't fallen or penalized and sees the ball.
    - moreover there shouldn't be more than two teammates which already announced to be striker and are closer to the ball
2. in the next cognition cycle the robot checks all received messages of its teammates for the ones who “wants to be striker“. The player closest to the ball becomes the first striker. The „second striker“ is the robot which „wants to be striker“, but sees a different ball than the first striker
    - hereby 'closest' means, the time the robot needs to approach the ball (see *time-to-ball* above)
3. only the first & second striker continues to announce “wants to be“ striker (see point 1.)

To prevent oscillation the last robot, who was striker, gets a time bonus in terms of “time to ball“ and should therefore always be the closest to the ball in situations where two players are approximately in same reaching distance to the ball.

In the whole decision process the goalie has a special role – he never „wants to be striker“. Instead the goalie gets always striker if the ball is near the own goal an the goalie needs to clear the ball. Then the goalie becomes striker, means he executes the striker behavior, and communicates its decision to his teammates. In consequence all teammates become passive in order to fulfill the requirement that there should be only one striker, except there is a player which sees a ball on a different position than the goalie („second striker“).

When we're assuming that all robots have nearly the same information about each other, every robot should come to the same conclusion of who should be striker and who should be passive.

The „two striker for two different balls“ approach, allowed us to more often kick the „real“ ball, even if the first striker sees a wrong ball.

As last safeguard, if there's only one player left, this player becomes striker.

7.3 Teamball

The teamball represents a hypothesis of the real ball, based on the ball observations of each player and consolidated to a single position on the field.

The teamball is then used to adjust the search behavior of passive robots and the goalie. In the past years, when the robot was passive and searched for the ball, the robot turned in one direction (randomly) and moves his head to find the ball. With the teamball the robot changes its behavior. Before the robot starts a new search rotation he turns to the teamball and looks in that direction in order to find the ball. If he doesn't see the ball, he continues to turn in the same turning direction as before, until a full round is completed. Before starting a new round, the robot takes again a look at the teamball and re-starts the described behavior. The ball search continues until the robot found the ball or another game event occurs.

In 2017 we reintroduced the teamball in order to improve our ball search behavior. Therefor each player communicated its ball model and collected the global ball position of its teammates over a certain amount of time. In each cognition cycle the median in x and y position of all collected balls where determined and used as teamball in the above described behavior. If there wasn't an update for some time, the teamball was invalidated and not used anymore.

This year, another calculation method was implemented. The „old“ teamball was sometimes far away from the „real“ ball, because of one de-localized robot or incorrect ball detection of teammates. The new method uses the canopy clustering algorithm to generate clusters of balls, communicated by teammates, and selects the cluster to which most robots contribute. The algorithm works roughly by collecting all communicated balls (global position on the field) and generating clusters of balls. The clustering uses two parameters in order to determine the „cluster membership“ of a ball. The „tight distance“ is used to select all balls which are only part of just one cluster and the „loose distance“ is used to mark balls which possibly belongs to one or more clusters. After the cluster generation, the cluster center (mean/average) with the most „members“ is used as teamball. The teamball gets invalidated, if there wasn't an update for a certain amount of time.

With the new calculation method, the teamball was more often closer to the „real“ ball and more robust against single robots providing „false“ ball information.

Bibliography

[1] Ralf Berger. Die Doppelpass-Architektur. Verhaltenssteuerung autonomer Agenten in dynamischen Umgebungen (in German). Diploma thesis, Humboldt-Universität zu Berlin, Institut für Informatik, 2006.

[2] Ralf Berger and Gregor Lämmel. Exploiting Past Experience. Case-Based Decision Support for Soccer Agents. In *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI'07)*. Springer, 2007.

[3] Hans Dieter Burkhard. Programming Bounded Rationality. In *Proceedings of the International Workshop on Monitoring, Security, and Rescue Techniques in Multiagent Systems (MSRAS 2004)*, pages 347–362. Springer, 2005.

[4] Hans-Dieter Burkhard and Ralf Berger. Cases in robotic soccer. In Michael M. Richter Rosina O. Weber, editor, *Case-Based Reasoning Research and Development, Proc. 7th International Conference on Case-Based Reasoning, ICCBR 2007*, Lecture Notes in Artificial Intelligence, pages 1–15. Springer, 2007.

[5] Daniel Hein. Simloid – evolution of biped walking using physical simulation. Diploma thesis, Humboldt-Universität zu Berlin, Institut für Informatik, 2007.

[6] Daniel Hein, Manfred Hild, and Ralf Berger. Evolution of biped walking using neural oscillators and physical simulation. In *RoboCup 2007: Robot Soccer World Cup XI*, Lecture Notes in Artificial Intelligence. Springer, 2007.

[7] M. Lötzsch, M. Risler, and M. Jüngel. Xabsl - a pragmatic approach to behavior engineering. In *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 5124–5129, Beijing, China, October 9-15 2006.

[8] Martin Lötzsch, Matthias Jüngel, Max Risler, and Thomas Krause. XABSL web site. 2006. http://www.ki.informatik.hu-berlin.de/XABSL.

[9] Heinrich Mellmann and Benjamin Schlotter. Advances on simulation based selection of actions for a humanoid soccer-robot. In *Proceedings of the 12th Workshop on Humanoid Soccer Robots, 17th IEEE-RAS International Conference on Humanoid Robots (Humanoids), Madrid, Spain.*, 2017.

[10] Heinrich Mellmann, Benjamin Schlotter, and Christian Blum. Simulation based selection of actions for a humanoid soccer-robot. In *RoboCup 2016: Robot Soccer World Cup XX*, 2016.

[11] Heinrich Mellmann, Yuan Xu, Thomas Krause, and Florian Holzhauer. Naoth software architecture for an autonomous agent. In *International Workshop on Standards and Common Platforms for Robotics (SCPR 2010)*, Darmstadt, November 2010.

[12] T. Röfer, J. Brose, D. Göhring, M. Jüngel, T. Laue, and M. Risler. GermanTeam 2007 - The German national RoboCup team. In *RoboCup 2007: Robot Soccer World Cup XI Preproceedings*. RoboCup Federation, 2007.

[13] Marcus M Scheunemann and Heinrich Mellmann. Multi-hypothesis goal modeling for a humanoid soccer robot. In *Proceedings of the 9th Workshop on Humanoid Soccer Robots, 14th IEEE-RAS International Conference on Humanoid Robots (Humanoids), Madrid, Spain.*, 2014.

[14] Vìctor Uc-Cetina. *Reinforcement Learning in Continuous State and Action Spaces*. PhD thesis, Humboldt-Universität zu Berlin, 2009.

[15] Yuan Xu. *From simulation to reality – migration of humanoid robot control*. PhD thesis, Humboldt-Universität zu Berlin, 2014.