Bembelbots Team Research Report for RoboCup 2019

Sina Ditzel Timm Hess Kyle Rinfreschi Jens-Michael Siegl
Felix Weiglhofer Felix Nonnengießer Benedikt Hahner Tim Schön
Jonas Dehen Ramona Fritz

January 21, 2020

Contents

1 Introduction **2**

2 JRLSoccer System Architecture **4**
2.1 Backend 4
2.2 Frontend 6

3 Cognition **7**
3.1 Object Detection Pipeline 7
3.1.1 Crossing Detection 7
3.1.2 Object Classifiers 8
3.1.3 Generation Of Synthetic Data 9
3.2 Worldmodel Creation 10
3.2.1 Teamcommunication 10
3.2.2 Self-Localization 11
3.2.3 Obstacle Detection 12
3.2.4 Ball Processing And Teamball 12
3.3 Behavior 13
3.3.1 Team Strategy 13
3.3.2 Reactive Walk 13

4 Motion Control **15**
4.1 Bodycontrol 15
4.2 Motions 16
4.2.1 Kick 16

5 Whistle Detection **17**
5.1 Base Whistle Detection 17
5.2 Directional Whistle Detection 17

6 Debugger **18**

Chapter 1

Introduction

This report describes the code of team Bembelbots developed for participation in the *Standard Platform League* at RoboCup 2019. It gives an overview of the current status of their code with focus on the changes for 2019.

Team

The RoboCup team *Bembelbots* was founded in 2009 at Goethe University Frankfurt (Main), Germany, as a group fully organized by students. As there is no robotics group at the university, the team should help students to increase their experience in robotics as well as programming skills in addition to the theoretical orientation of the computer science degree program of the university.

Figure 1.1: Team *Bembelbots* at the RoboCup German Open 2019 in Magdeburg, Germany.

Currently 11 students are working on the implementation of the framework for playing soccer. Teamleader is Jens Siegl, a student of computer science at Goethe-University. The team owns five Nao v6 and six Nao H25 v5 robots. Since 2012 the team *Bembelbots* organizes the FIAS BembelCup in line with the "Night of Science", a public event showing different aspects of science, technology, engineering and mathematic study paths offered at the Goethe University in Frankfurt. This small tournament of three to four competing teams is one of the final tests for the RoboCup, as it is temporally located nearby.

Goals

Due to the environmental changes to ball, playing field turf, and lighting, our landmark and ball detection was faulty in the last years, which affected our self-localization and made team play nearly impossible. Since we mostly mastered these challenges in preparation for RoboCup 2018 we are now focusing on how to play with team strategy such as preventing fighting for the ball in our own team. Additionally, due to missing robot detection, the team suffered severely from time penalties for illegal robot-robot interactions. Development of anti pushing measurements was another main priority.

Report Structure

The report will first outline the fundamental architecture of the Bembelbots framework. Subsequently, the cognition process from camera images to decision making, motions and motion-scheduling, whistle detection, approaches to challenges posed in 2019, and finally debugging applications will be described.

Chapter 2

JRLSoccer System Architecture

As the main goal of the team is to teach students in the field of robotics, as well as working on a complex software architecture, we focused right from the beginning on building a full framework from scratch rather than using existing architectures. Our framework is split up into four main parts: A *backend* implementation, providing data for the *frontend* and controlling communication to the robot's hardware. The frontend represents the robot's software functionality. Backend and frontend communicate using shared memory segments, in order to minimize latency and overhead. A standalone *monitor* provides system information on robot health, as well an interface to configure game setting via UDP. For debugging purposes we use several tools written in *Python* combined in our *bembelDbug* debugging suite. Figure 2.1 illustrates the overall structure of our framework.

2.1 Backend

The *backend* encapsulates all interaction with the robot hardware, such as reading sensors or setting actuators. This allowed us to switch to the new LoLa interface on Nao V6 with relative ease, as we were able to write a new LoLa-backend while running the same frontend code with very few modifications (mainly related to the different camera modules) on both Nao V5 and V6 robots. In addition to the LoLa and NaoQi backends, there is also a dummy backend which allows running the frontend for testing and debugging on regular Linux systems.
Communication between backend and frontend is done through three shared memory (SHM) segments, each passing data only in one direction: Sensor data is passed from backend to frontend, and actuator data is fed back from the frontend to the backend. On V5 robots there is an additional SHM segment for text-to-speech output through NaoQi.

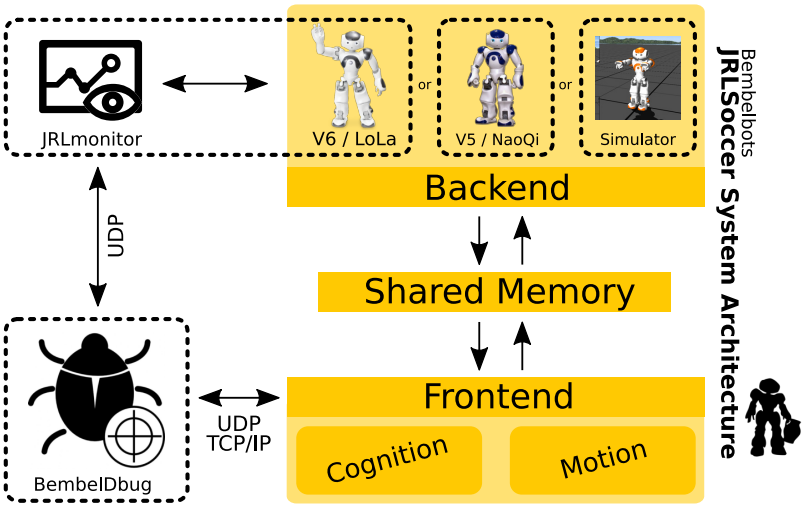

Figure 2.1: Overview of the JRLSoccer Framework.

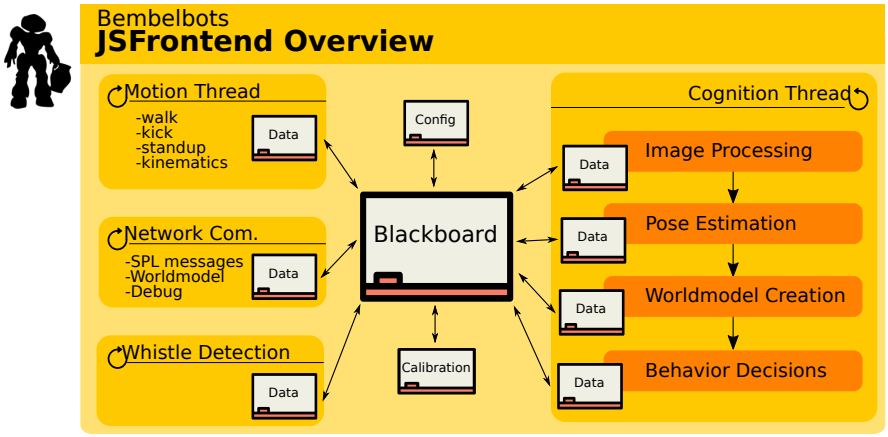

Figure 2.2: Overview of the main parts of the JSfrontend architecture with its main threads.

2.2 Frontend

The *frontend* is where all of our game logic is implemented. Being implemented as a standalone program, this allows us to test the frontend separately from the robot hardware, which is very helpful to reduce the possibility of memory leaks. The software management is organized using Git so that the whole framework can be tested using a Jenkins [1] continuous integration process after every code change.
Our framework is organized as a blackboard architecture, giving every module the possibility to share data, access information of other modules, and increase debuggability. The data in each blackboard can be viewed, analyzed, and manipulated at runtime using our debug tool *BembelDbug*, see section 6.

The frontend is split up in two major parts. The *cognition* thread is synced with the image gathering process, calculating the localization, world beliefs, and the desired behavior decisions. The *motion* thread is limited by the sensor acquisition frequency and computes the movements of the robot (see section 4 for more details). Network communication with other robots as well as the whistle detection (see section 5.1) are also placed in separate threads. These will be started and stopped on demand. An overview of our frontend structure can be found in Figure 2.2.

As basically all of the computationally intensive tasks (e.i. vision processing) are done sequentially within the *cognition* thread, the current architecture cannot utilize the full compute power of the Nao V6's quad-core CPU. For this reason we plan to move to a new architecture which will allow parallel execution of tasks in 2020.

[1] Jenkins open source automation server: `www.jenkins.io`

Chapter 3

Cognition

3.1 Object Detection Pipeline

Our robot's vision is based on the HTWK Leipzig vision pipeline release of 2018 [1]. This provides us with field color, bounds detection, as well as lines, and regions of interest for possible ball candidates. Our contribution is the refinement of line information to extract line crossing type and location as additional information for our localization, and classifiers for ball, robots, and penalty marks.

3.1.1 Crossing Detection

We use a line based approach to detect T or L shaped crossings. X shaped crossings are currently not in use. Line segments, received from the vision pipeline, are compared to each other to detect intersections at a 90 degrees angle when projected from the camera image into the robot's coordinate system. Those are considered possible candidates for T or L crossings.

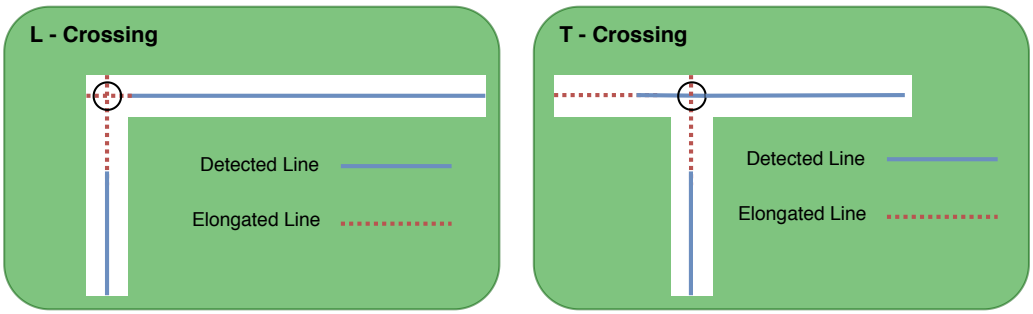

Figure 3.1: Crossings Detection Process
Crossings detection by elongating lines and checking distances to crossings intersections (circle). On the *left* both line segments got elongated to find an L-Crossing, as the line-endings are both in bounds with respect to the crossing center. On the *right* a T-Crossing is detected by elongating lines, as one line ending is out-of-bounds it cannot be labeled L-Crossing.

[1] NaoHTWK's HTWKVision GitHub repository: `github.com/NaoHTWK/HTWKVision`

The candidate line segments are extended as far as possible without passing a line-color (white) to field-color(green) gradient, in order to compare the position of lines intersection to the length of the line. If for one intersecting line the distance from point of intersection to the end of the line is grated than a given threshold, e.g. half of the line width, this crossing is considered a T crossing. Else it is considered an L crossing. For a depiction of the process, see Figure 3.1
For the robot's self-localization it is important to also estimate the crossing's orientation. Each crossing is marked with a fixed angle [2]. For L crossings the direction of the 'right' crossing's line is used, for T crossings the direction of the 't-base' line is used, to determine the angle of the crossing to the robot. This further distinguishes crossings of the same type, see Figure 3.2.

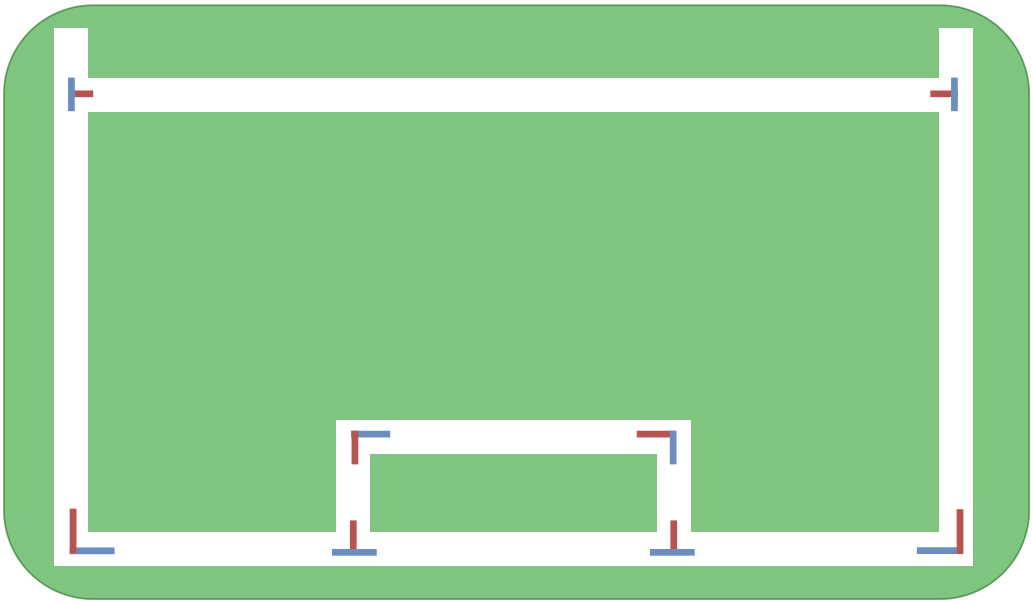

Figure 3.2: Crossings Orientation
A sketch of the crossings directions for one half of the playingfield. The red lines mark the orientation of the crossing.

3.1.2 Object Classifiers

The ball detection makes use of the regions of interest (ROIs) provided by the HTWK vision pipeline. Each ROI marks the image region of a possible ball candidate. We check the ROIs by passing them through convolutional neural network classifiers. There are two classifiers with the same architecture, one for ball classification and one for robot(feet) and penalty mark classification. As all of theses categories are frequently captured by the ROIs, we apply both classifiers on each ROI. The input to our classifier is a 12×12 grayscale image patch. The classifier's architecture is depicted in Figure 3.3. For deployment on the robot we use an SSE optimized version of the caffe framework, provided by HTWK.
Our training data is mostly synthetic with real-world data added during training for fine-tuning.

We outline the process for data-generation in section 3.1.3. The real-world images have either been taken of releases from other teams, such as SPQR, HULKs, and HTWK, which at the time only included annotations for ball, or from our own data gathering, where we increased performance of our classifier in an iterative fashion using it for semi automatic labeling of the image patches recorded by the robot to decrease human annotation effort.

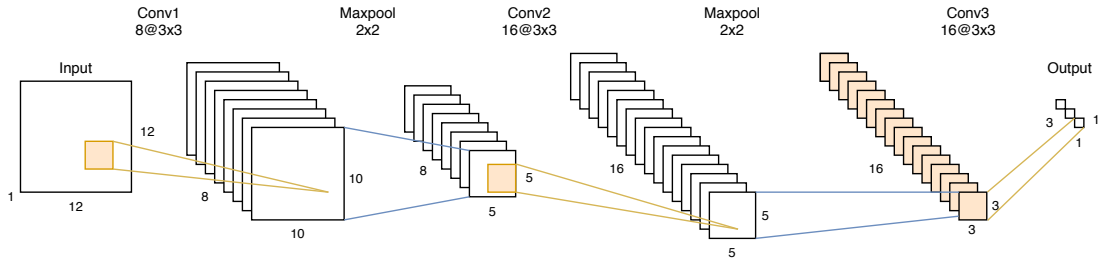

Figure 3.3: Bembelbots Classifier

3.1.3 Generation Of Synthetic Data

As of the introduction of the new black and white ball in 2016 our team switched from a purely model based ball detection approach to object classification using machine learning. By that time only little data was publicly available. Thus we started early on using *synthetic* data for training purposes. This data is available at plenty and comes with included pixel-wise annotation.

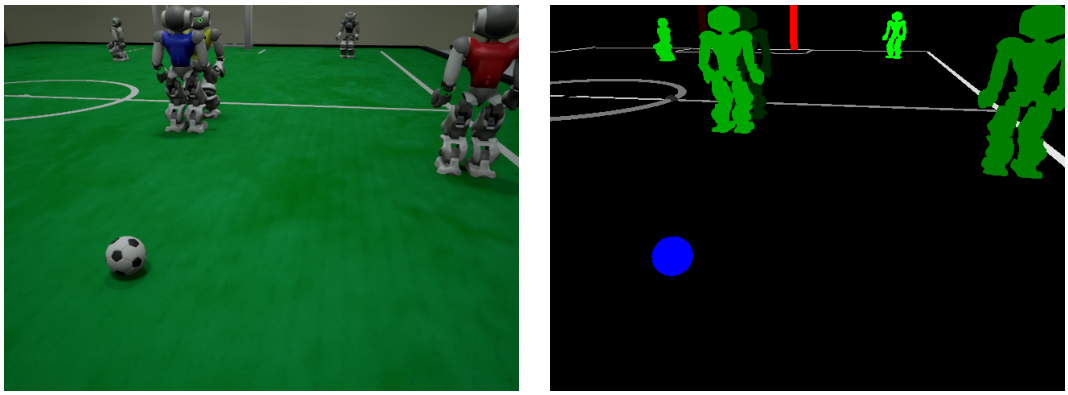

Figure 3.4: Example of synthetic data with semantic ground truth annotation.
The figure shows a single example of a simulated playing situation (**left**) together with its automatically generated pixel wise ground truth annotation (**right**).

Data generation is performed by replicating the RoboCup environment, as given by the official rule

book, in an *Unreal Engine 4* [2] scene. The choice for this particular rendering engine is due to its trade-off of being a real-time renderer, which is needed to be able to create new data fast, and at the same time providing one of the highest photo-realism standards by incorporating physically-based-rendering (PBR), keeping the domain gap from synthetic to real data as small as possible. The 3D models are self-made with exception to the Nao model which is provided by SoftbankRobotics. The scene's lighting setup is oriented on typical ceiling mounted lights as common on most RoboCup venues. The simulation allows to create SPL game scenes by a stochastic process: All robots and the ball are placed at random positions on the playingfield, where one robot has a observing camera attached. At the same time the environment, i.e. lighting and playingfield color are sampled from a range of known possible colors and intensities. This aims to diversify the generated images such that the classifier finally trained on this data has highest generalization capabilities. For an extensive explanation of the generation process, please see [5]. From the pixel-wise annotation, we cut image patches mimicking the ROIs of our detection pipeline. Each year we refine the image generator. Recently we added sprayed lines of sampleable intensity as well as a new grass shader. Also natural lighting conditions, i.e. sun light, has been added, see Figure 3.5.

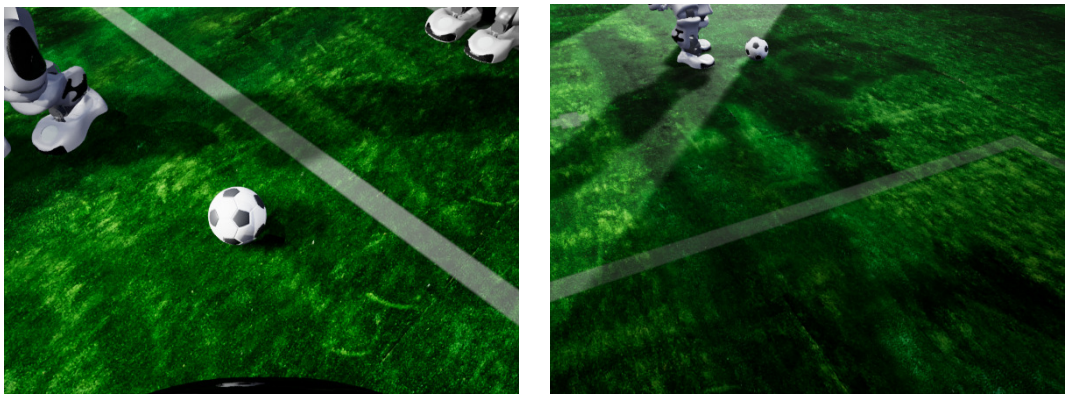

Figure 3.5: UERoboCup Visual Extention
Example of the new playingfield grass shader with sprayed lines of multiple opacities, and sun lighting.

3.2 Worldmodel Creation

The data extracted by the vision module as well as data received from teammates has to be processed and merged into a model of the world of the robot, named *Worldmodel*. It contains positions of all objects relevant to the robot, such as ball, teammates, obstacles, teamball and itself. All behavior-decisions are based on these informations, which are updated, estimated and calculated with the following modules:

3.2.1 Teamcommunication

Each robot broadcasts its estimated pose, last known ball location and additional data, via UDP in the standard SPL message format (see Figure 3.6).

[2] Unreal Engine 4 `www.unrealengine.com`

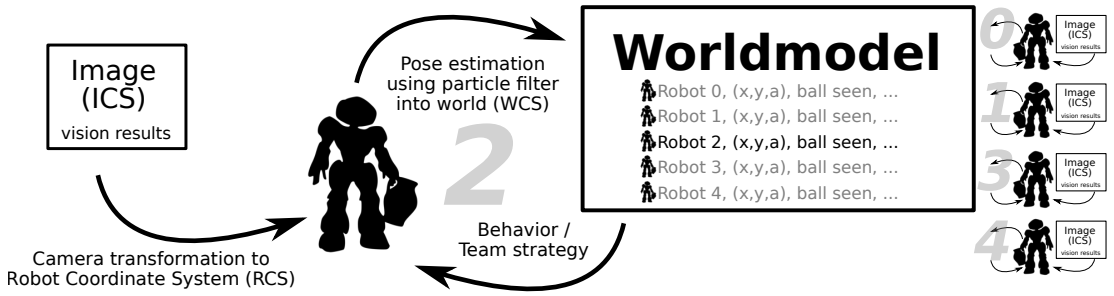

Figure 3.6: Creation of the worldmodel from own pose position process and the by receiving broadcasts from teammates.

3.2.2 Self-Localization

To estimate the robot's pose, visually perceived landmarks and odometry are fused in a particle filter, see [3], a Monte-Carlo localization (MCL) algorithm. One of the main reasons for the particle filter is the possibility of solving global localization problems, as well as the integration of *all* observed landmarks without further processing. The filter represents the belief of the robot's pose with a set of samples (particles). Each particle is a possible pose of the robot. In the initial state, after a penalty, and after manual placement, the particle filter is initialized by fixed pre-determined distributions, giving priors of the robot's pose according to the situation.
In our localization approach we use a combination of motion controls, and sensor data from gyroscopes for the motion model update. We weight the particles by matching the visually perceived lines and crossings-interceptions with a map of known field landmarks. In each measurement update the filter compares landmarks by their distance and angle, and matches them using the maximum likelihood method. We chose the systematic resampling algorithm [11], which yields a reliable localization result for low numbers of particles (less then 100). To determine the robot's pose from the particle distribution we calculate the particle closest to the mean of all particles. This offers the best compromise between accuracy and runtime performance, compared to choosing the particle with the highest weight or applying a cluster algorithm. To evaluate the filter and the different methods of resampling and position determination, we use a ceiling mounted camera, tracking LEDs attached to Nao's head. This provides us with ground truth information on the robot's movement on the field. An evaluation of a path over the playingfield is illustrated in Figure 3.7.
The particle filter only works properly if the particles cover the true robot position. If all particles lost track, e.g. because of a series of false detected landmarks, most of the time the particle filter does not find the track again and the robot is de-localized. To prevent this situation we developed a readjustment method for the particles this year: If we see a significant landmark (e.g the center circle) or combinations of those, so that there are only a few positions possible for the robot on the field, these are used as position hypotheses. We then choose the hypothesis that is closest to the particles and sample a low percentage of the particles at this position.

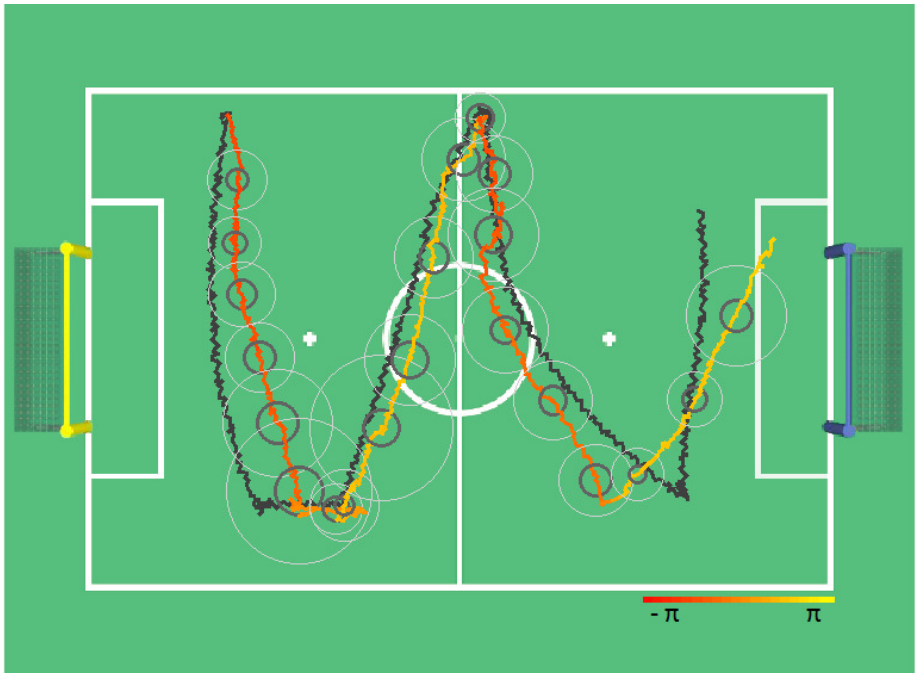

Figure 3.7: Evaluation of the particle filter
The black line represents the ground truth position of the robot. The yellow to red line shows the estimated position, where the color decodes the estimated orientation of the robot. The circles represent the distribution of the particles: Dark grey circles display the mean distance of the particles from the center, light grey circles indicate the particle position with the highest distance.

3.2.3 Obstacle Detection

We consider other robots and goal posts as obstacles that should be avoided by the robot's behavior, specifically by our reactive walk. These obstacles are detected by our vision module and by monitoring the movements of our arms while setting the stiffness to a low value, interpreting changes in joint values as collision with an obstacle. Both measurements are integrated into the Worldmodel and stored for 3 seconds. New measurements are merged with already known ones when they are close to one-another.

3.2.4 Ball Processing And Teamball

We apply an exponential filter to the estimated ball position from the vision module, due to the ball's position estimation being subject to small variations, as the ball detection is currently not able to pin point the ball center. Further, ball related informations are calculated, such as which robot is nearest to ball, teamball and if a seen ball matches the team ball. The teamball is used when multiple robots see a ball. It is estimated by clustering the balls seen by all robots of the team, choosing the cluster with the biggest size and calculating a mean position of the balls, weighted by a confidence factor $e^{-\alpha d_i}$, where d_i is the distance of the corresponding robot to the ball seen by it and α a parameter.

In case a robot is assigned goalkeeper in a penalty shoot-out, a ball motion filter is applied. It predicts the ball movements based on linear regression, to calculate if the ball will hit the goal-line left or right of the goalkeeper.

3.3 Behavior

As of recent years, we still use CABSL, the standalone C-implementation of XABSL published by B-Human. CABSL is an implementation of hierarchical finite state machines, that allows the robot to make long term and short term decisions. Since our self-localization got more reliable over the last years, and based on a new module that processes and stores all ball-related information, see 3.2.4, we have developed a dynamic team strategy, which should prevent robots from fighting for the ball within the own team. Also this year, we have been working on a reactive walking behavior, allowing our players to dynamically avoid obstacles, which is implemented outside of CABSL. This reduces CABSL to more high-level-behavior.

3.3.1 Team Strategy

We dynamically decide who should be the striking robot based on the ball distance, while in previous years the robots were following their pre-configured role assignments, which often lead to fighting for ball in the own team. The striking robot dribbles the ball into the direction of the opponent goal. If a robot is not assigned the striking robot, it still follows predefined actions:

- The goalkeeper mainly stands in his goal, with the exception of attacking the ball when it is close by.
- The defender positions itself between the goal center and the ball if the ball is inside the own team's half of the field, else it positions itself closely to the front of its teams penalty-area.
- The other offensive robots follow the striking robot to block the goal and take over the dribbling in case the striking robot falls or loses the ball.

3.3.2 Reactive Walk

In order to prevent our robots from walking into obstacles (other robots, goal posts, etc.) we implemented a reactive walking behavior. It leads to a more dynamic walking path and reduces robot collisions and therefore pushing or foul penalties. Objects present in our Worldmodel (robots, ball, and goal posts) get assigned an either positive or negative charge. In our model these charges apply an attracting or repelling force onto the navigating robot, see Figure 3.8. The strength of the force is determined by the objects charge and distance. Closer objects will therefore have a bigger impact on the robot's walking behavior than objects further away. The robot's walk target has a positive charge assigned and therefore attracts the robot. The resultant force vector is calculated by adding up all forces applied to the robot, determining the robot's walking direction. A filter is applied to smooth rapid direction changes.

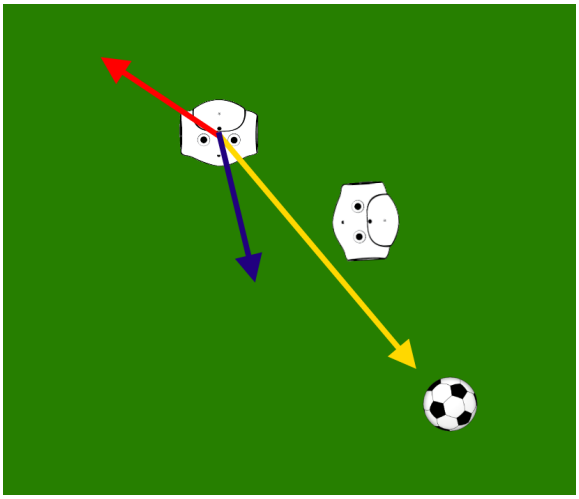

Figure 3.8: Forces determining the robots walking direction
The left robot is attracted by the ball (yellow force vector) and repelled by the other robot on the field (red force vector). Combining these forces results in the walking direction of the robot.

Chapter 4

Motion Control

4.1 Bodycontrol

The new body control and motion framework, *bodycontrol*, see [6], has a modular structure, in which all motions are separate modules, some of which run exclusively, such as whole body motions (i.e. stand up moves), and some run in parallel, such as the walk and the head looking towards a position. Apart from motions, the bodycontrol includes additional functionalities: Sensor data acquisition, center of mass (CoM) calculations, camera transformations, inertial measurement unit (IMU) filters, and setting LEDs for debugging purposes. Which all follow the same modular structure. Figure 4.1 shows the structure of the framework.
For each motion thread cycle, all active modules modify the data on a blackboard upon being called in sequence by a control shell (runner), which also handles activation and deactivation of modules as well as the communication with the backend and the cognition/behavior thread. Testing individual modules is done by faking data on the blackboard and running a single module. Alternatively the bodycontrol may run independently from the rest of the framework. In this dummy frontend the whole cognition and behavior thread is replaced with a network thread that listens to our bembelDbug network protocol, and can pass arbitrary commands to the same thread-safe command queue the behavior would use from inside the JRLSoccer framework. The modules are not interdependent to one-other for compilation, they only depend on the blackboard that needs to have the correct fields and has no dependencies on it's own. Logical dependencies are declared on registration of the modules in a single file, where each module gets a human readable ID, and optional dependencies and priorities are declared, determining the execution sequence. Compared to the standard dependency injection design pattern this design omits the need for defining interfaces. It also differs from the classical blackboard approach, as its control shell has no knowledge of the data the modules may modify. This puts more responsibility for module dependencies on the programmer, which is mitigated by the blackboard and the inclusion file, giving an complete overview over all modules, data and dependencies in a single file.
For the composition of motions we provide some utility functions. Inverse kinematics of the lower body, stabilizing functions using the arms or legs, and interpolation functions can be used to program the motions in a conceptual space closer to how we would describe them as humans. Additionally most motions have different phases or sections for which we adopt the well known finite state machine concept. These concepts allow us to define states interpolating between keyframes

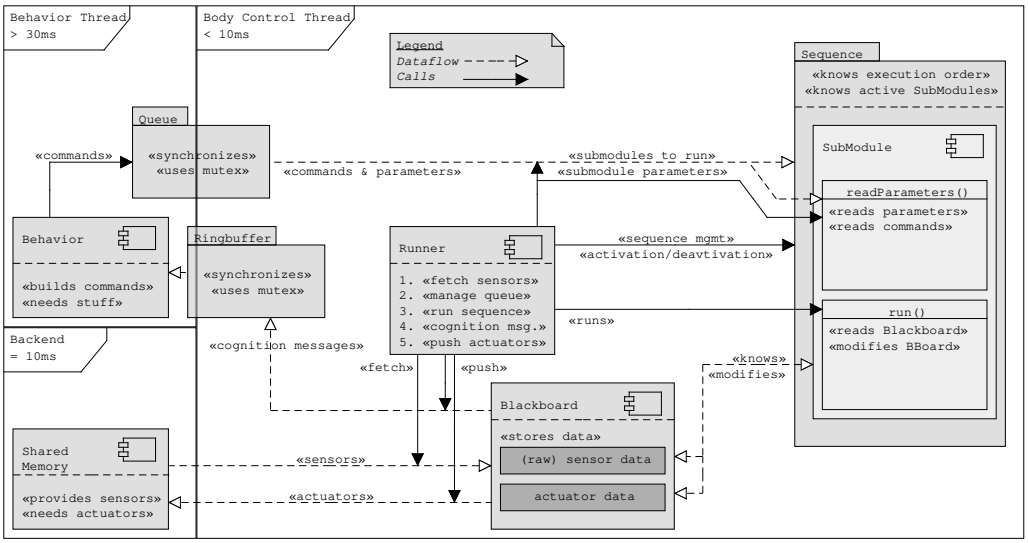

Figure 4.1: Bodycontrol and Motion Framework.

that are calculated with inverse kinematics. On motions such as a stand up move, we can check if a part of the motion succeeded and repeat parts as necessary. Through the return values of the states, a motion may signal if it is currently stable or in a critical section and should thus not be interrupted.

4.2 Motions

We use the walk modules of Nao-Team HTWK. The stand-up-motions are static motions.

4.2.1 Kick

With the fast movements and one-foot stand needed when kicking a ball, naturally come stability issues, which are further increased by external forces from other robots. We use a mostly pre-computed kick motion to minimize the computation needed when performing the kick: key-frames are defined based on center of mass and angular momentum calculations to achieve static and dynamic stability. Forceful contact can then be balanced by counter movements in ankles and shoulders, using prior calculated per key-frame balancing details (see [7]).

Chapter 5

Whistle Detection

5.1 Base Whistle Detection

Our whistle detection uses the *alsa* library to access microphones on the Nao robot. A Fast Fourier transformation is run on the acquired audio data using the library `fftw3`[1]. Once a whistle candidate is detected, in the frequency domain we check whether the frequency was above 2000 Hz for most of the signal. As the detection itself needs a lot of computational resources, the detection is enabled only in special game situations, such as the set state, and is disabled in the playing state.
Last year we published the whistle detection on our Bembelbots Github page [2]. The release also includes a small debugger to test the whistle detection on `wav` files.

5.2 Directional Whistle Detection

The directional whistle detection was developed to localize the referee after blowing the whistle. Necessary for its successful application is a well-functioning whistle detection. The moment when our whistle detection recognizes a whistle, the directional whistle detection determines the sound source location by calculating the time offset between the four microphones of the Nao robot. To achieve this, cross correlation was implemented and used on the first 60 ms of the detected whistle. To determine the microphone which is closest to the sound source, the sound volume of each microphone is evaluated. This allows us to limit the location to a single quadrant. Then the angle to the sound source is calculated by using cross correlation of only the two nearest microphones, to prevent distortion of the results.
Given the results of multiple robots, the determined angles are triangulated and the center of the resulting triangle is calculated as the estimated sound source position. We chose to use multiple robots for the localization instead of only using a single robot. In fact the usage of a single robot is possible, but needs a different method for localization, which is slower and more inaccurate.

[1] Fast fourier transformation, `www.fftw.org/`
[2] Bembelbots whistle detection: `github.com/Bembelbots/NaoWhistleDetection`

Chapter 6

Debugger

To communicate with our framework we designed our own debug protocol based on json and raw data. It uses a client-server architecture where the client is the debugger and the server is the Nao. Only one client is allowed to make changes to the Nao at a given time. Thus in order to make changes, the client has to first connect with the server. Once the connection is terminated through disconnect or timeout another client can make a new connection. We wanted to make the protocol as simple as possible. To accomplish this it was decided to use a subscriber system, where the connected client subscribes to the required data. The sending and receiving of data is done over UDP except when receiving images, which is done over TCP. Previously we interfaced with our debug protocol through a Python 2 and Qt4 program called NaoDebug. After some time we realized that the user interface was inadequate as module experts in the team required information from multiple sections concurrently. Implementing this functionality proved challenging and after some analysis we decided to start fresh. Thus a new program named *BembelDbug* based on Python 3 and Qt5 came into fruition. The idea is to have each tool, previously named section, in its own window allowing the user to decide how to place the tools for an optimal overview based on debugging requirements. This would also led to each tool being self contained. The advantage being that each module expert can write their own tool, extending the debugger without any cross dependencies. This new system has proved to be useful and has decreased setup time before games, whilst improving productivity during development on the Nao.

Bibliography

[1] Fürtig, Andreas, Holger Friedrich, and Rudolf Mester (2010): "Robust Pixel Classification for RoboCup." Farbworkshop Ilmenau

[2] Becker, Christian (2013): Linienbasierte Featuredetektion und Positionstracking durch Voronoi-Diagramme in einer symmetrischen Umgebung, Diploma thesis at Goethe University, Frankfurt

[3] Ditzel, Sina (2016): Selbstlokalisierung des Nao-Roboters im RoboCup mittels Partikel Filter, Bachelor thesis at Goethe University, Frankfurt

[4] Hess, Timm (2016): Training convolutional neural networks on virtual examples for object classification in the RoboCup-environment, Bachelor thesis at Goehte University, Frankfurt

[5] Hess T., Mundt M., Weis T., Ramesh V. (2018): Large-Scale Stochastic Scene Generation and Semantic Annotation for Deep Convolutional Neural Network Training in the RoboCup SPL. In: Akiyama H., Obst O., Sammut C., Tonidandel F. (eds) RoboCup 2017: Robot World Cup XXI. RoboCup 2017. Lecture Notes in Computer Science, vol 11175. Springer, Cham

[6] Brast, Jonathan Cyriax (2017): NAO Body Control and Motion Framework for RoboCup, Bachelor thesis at Goethe University, Frankfurt

[7] Hahner, Benedikt (2019): Dynamic Stabilisation of the Kick-Motion for Soccer Playing Humanoid Robots Based on Center of Mass Calculations, Bachelor thesis at Goethe-University, Frankfurt

[8] Zheltonozhskiy, E.: Tinydnn. https://github.com/tiny-dnn/tiny-dnn (2017). Accessed: 09.05.2017

[9] Min Lin and Qiang Chen and Shuicheng Yan(2013): Network In Network, CoRR

[10] Reinhardt, Thomas (2011): Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball, Master thesis at Hochschule für Technik, Wirtschaft und Kultur Leipzig

[11] R. Douc and O. Cappe (2005): Comparison of resampling schemes for particle filtering, ISPA 2005