technische universität
dortmund

Final Report of Project Group 590

Soccer Playing Humanoid Robots

Christian Albrecht, Manuel Barbi, Karen Bieling,
Torsten Böttinger, Marco Dürr, Lennart Downar,
Marcel Eilers, Julian Hannich, Janine Hemmers,
Fabian Rensen, Heiner Walter, Florian Ziegler

April 6, 2016

Supervisors:
Dr. Lars Hildebrand
Dipl.-Inf. Oliver Urbann
Dipl.-Inf. Ingmar Schwarz

Technical University of Dortmund
Robotics Research Institute
http://www.irf.tu-dortmund.de

Contents

1 Introduction

The field of humanoid robotic soccer is a vast combination of multiple disciplines. As such it combines fields like computer vision, machine learning and control theory. Additionally, this takes place in a resource constrained environment, as a robot's computing power is limited and therefore several otherwise well known and working approaches cannot be used or have to be altered to suit the capabilities of a robot system.

Another often forgotten but very important aspect is the self organization and structuring of a project group. Managing large software projects in a productive manner should not be underestimated and can only be done with good workflow structures and good documentation. By combining current interdisciplinary research as well as practical programming experience, a project group in this environment thus allows a wholesome view into the daily workings of a researcher.

RoboCup is an international initiative to foster the development of intelligent robots through competitions. One of many leagues at *RoboCup* is the *Standard Platform League* (SPL)[1]. The currently used robot platform here is the *NAO* robot developed by *Aldebaran Robotics*[2]. This way, every team has the same hardware and it is a competition of the best code and algorithms.

As a team, *Nao Devils Dortmund* are major participants (a team being qualified for the main competition as well as for challenges) at *RoboCup* events, such as the German Open or *RoboCup* itself. Those competitions allow us to evaluate and test our work, as well as to compare it with the results of some of the best teams in the world. This inter-cultural exchange with researchers from all over the world is very fruitful for our own development and adds another virtue to the project group. Especially for students, this is usually a first glimpse into applied research on an international level.

1.1 Goals of this Project Group

Every year a project group of 8-12 students is formed to work on and improve the code for TU Dortmund's robot soccer team *Nao Devils Dortmund*. In this report we are documenting the results of project group 590, which started its work in the summer term 2015 and finished at the end of the winter term 2015/16.

For this project group, the development of a coaching robot, which analyzes games and communicates with the field players has been set as the group's main goal.

[1] HomepageoftheStandardPlatformLeague:http://www.tzi.de/spl/bin/view/Website/WebHome
[2] HompageofAldebaranRobotics:https://www.aldebaran.com/en

Figure 1.1: A Robocup SPL game.

To this end some minimal requirements have been defined:

- The ball has to be detected on the field while the coaching robot is in a sitting position on a table next to the field.
- The coaching robot should be able to communicate the ball position to field players (at least on which side of the field the ball is found) in human-readable form.
- The coaching robot should be able to use the ball position to analyze the game, which should result in strategic decisions such as a defensive behavior or offensive behavior.
- At least in a simulated game, the coaching robot should prove to be an advantage as compared to a team without a coaching robot.

In the following chapters the progress towards these goals is documented, as well as other project improvements which have been done during the project group.

2 Developing a Coaching Robot

Lennart Downar, Marco Dürr, Christian Albrecht

In recent years a coaching robot has been allowed for SPL Games. The idea is that this robot is allowed to observe the game and give strategic advice to the team's field players. To prevent the coaching robot from turning into an external vision system or remote controller, the coaching robot does not communicate directly with the field players. Instead messages in human-readable form will be sent to the *GameController*(a computer controlling the timing and penalties of the game), which in turn forwards the messages to the field players. The general rules for a coach robot are specified in detail by the SPL technical committee in the rule book[1].

The coaching robot observes the game from a table next to the field. However, the positions are not fixed and thus can differ from one game to another. Furthermore, the coaching robot is allowed to sit on a chair or other sitting support while observing the game (e.g. see Figure 2.1). To enable the coaching robot to map the playing field onto its camera image, it has to be calibrated. For this, the distance of the robot from the field borders and the height at which it is placed must be known and entered into a configuration file.

[1]SPL Soccer Rules 2015: http://www.tzi.de/spl/bin/view/Website/Downloads

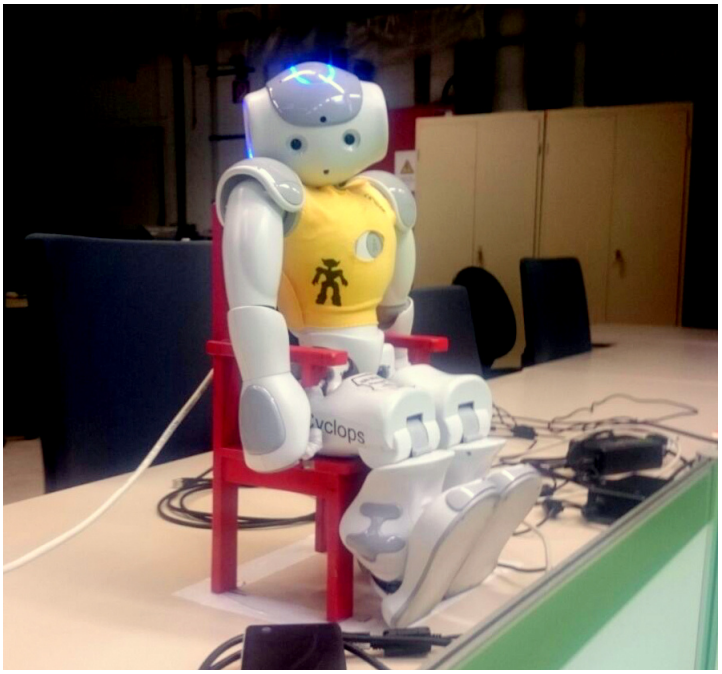


Figure 2.1: Image of the coaching robot observing a game.

2.1 Infrastructure

To centralize the analysis of a game and the sending of messages, the module *CoachControl* was developed. In its current form this module uses the the current position of the ball (provided by the ball detection, cf. Chapter 10). Based on where the ball is currently located on the field, different messages can be sent. All this happens inside the method *evaluateGame()*, which is executed by *CoachControl* at every time frame.

To reduce the amount of computation for the coaching robot, a specific configuration folder *Coach* was created, which contains adapted configuration files that deactivate modules only relevant for field players (e.g. modules providing information for walking or arm contact), and activate the coaching robot behavior.

2.1.1 Communication

To implement communication via Ethernet, the *GameHandler* module had to be adjusted such that it can be created with a custom IP address. This was previously automatically set to the current WiFi address. Additionally three macros, `COACH_COM`, `START_COACH_COM` and `SEND_COACH_MESSAGE`, were defined inside the *GameHandler* class. These essentially create a new *GameHandler* object that connects via Ethernet to the *GameController* and allows sending messages to the *GameController*.

The current rule set, as of February 2016, specifies that the communication has to be performed via broadcast over a UDP socket on port 3839. A standardized package, *SPLCoachMessage*, has been defined by the *SPL Technical Committee*[2], which sets the limit of the maximal length of the message sent by the coaching robot to 81 bytes. Additionally the header has to contain a version number, the team number to which the message is directed, and a message sequence number. Messages sent by the coaching robot are displayed on the *GameController*. However, messages will only be accepted every ten seconds, independent of the sending frequency. A received and displayed coaching robot message can be seen in Figure 2.2.

2.1.2 Message System

During the implementation of the communication for the coaching robot, a message system was developed, which allows multiple messages to be grouped and stored. For future use of the coaching robot, this could be useful if several modules provide different messages, and, depending on the timing or current game situation one message has to be chosen. Additionally, it would then be necessary to extend the message system with a priority mechanism, as multiple events and situations may be detected as candidates to be sent. The importance of information depends on its age and its content.

The collection of messages is organized by the class named *MessageSystem* in classes called *Group*s. Each *Group* object managed by the *MessageSystem* contains a set of *Message*

[2]Current Technical Committee: http://www.tzi.de/spl/bin/view/Website/Committees2016

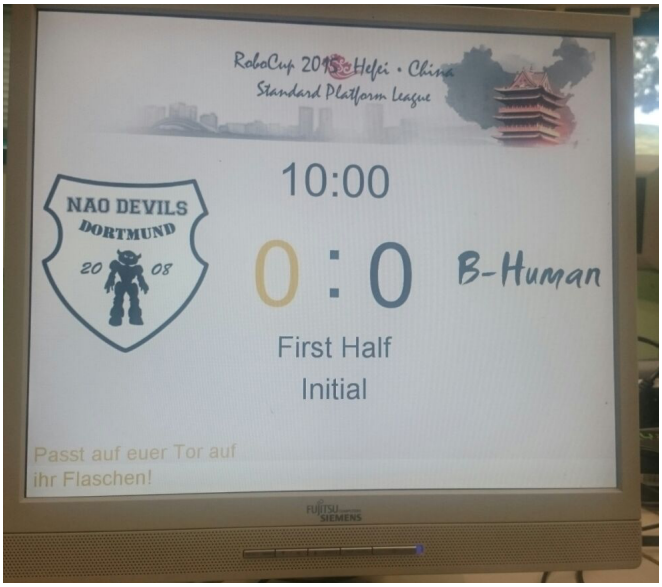


Figure 2.2: Image of the *GameController* displaying a message from the coaching robot.

objects dedicated to a special game event or situation. The *Message* class is just a container for the actual message, but it provides the ability to dismember the message into chunks of a predefined maximum size. This is useful if the message size exceeds the given maximum size of 81 bytes.

2.2 Behavior

The behavior of the robots is written in *CABSL*, a state machine language[3]. For the coaching robot a new behavior had to be written, as it has to act and behave differently as compared to a field player. Initially the state machine starts in state *sit* in which the robot performs a specific special action to relax all joints except the head[4]. If the action is completed the behavior switches to the *ballLost* state in which the *CoachSearchForBall* state machine is called. Here, the head motion to find the ball is executed. To avoid blurring it was decided to keep the head in fixed positions instead of focusing the ball in the center of the vision. Based on test results a minimum of three different head motions is necessary to cover the whole field. This is why the head can take 3 positions, right, middle and left, as shown in Figure 2.3. Out of all states a common transition to *foundBall* is performed if the robot recognizes a ball based on previous calculations. After finding the ball the behavior switches into the state *ballSeen* which checks if the ball is seen for a given time range. The coaching robot remains in this state as long as the condition holds. Otherwise the observation circle starts again by changing into the *ballLost* state.

Figure 2.3 is a visualization of the state machine of the coaching robot behavior. It consists of two different options named *Coach* and *CoachSearchForBall*. An initial state is that in

[3]For further information see: https://www.b-human.de/downloads/publications/2015/CodeRelease2015.pdf

[4]Otherwise no head motion is possible.

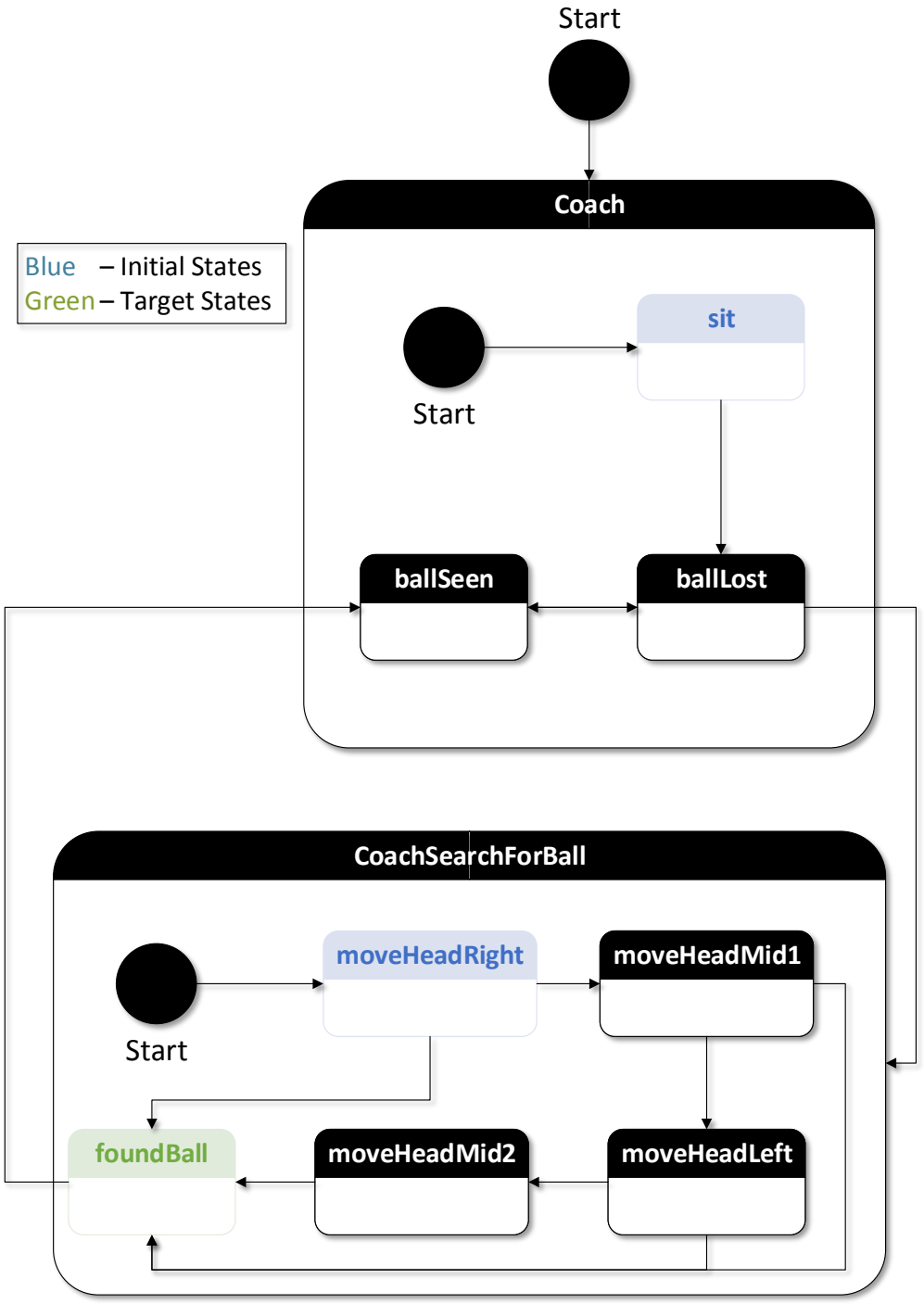


Figure 2.3: State machine visualization of the coaching robot behavior.

which a given state machine starts. If a state machine arrives in a *target state*, the state machine that called this state machine can be notified and act accordingly.

2.3 Conclusion

After *RoboCup*2015 the general rules were changed. Instead of a red ball, a ball closer to real soccer balls (black and white) will be used for future competitions. From this arose the need for a new ball detection. Details concerning the newly developed ball detection will be discussed in Chapter 10. Additionally a robot detection was developed, to identify players on the field and give strategic decisions (for details see Chapter 6). In the following chapters the details needed to implement the robot and ball detection is documented.

Future Work

The general infrastructure to utilize a coaching robot has been provided. In the future field players should make use of the information provided by the coaching robot. This has not been done at the moment as the approaches taken to detect balls have been proven to not be robust.

Improvements for the behavior would mainly concern adjustments of the head motion. A major issue is that the head motion always starts from the right side. To improve this, it would be necessary to track the ball's direction and move the head according to the side at which the ball leaves the field of vision. Another improvement related to the head motion could be an auto calibration, as currently the head angle needs to be adjusted manually with respect to the coaching robot position.

Furthermore, it would be interesting to include the GOL Framework [12] together with the coaching robot to provide not only direct information, but also strategic decisions. We plan to implement and test this at the *EuropeanOpen 2016*[5].

[5]Homepage of the EuropeanOpen 2016: https://www.robocupeuropeanopen.org/

3 Edge Detection Using Streaming SIMD Extensions

Fabian Rensen

Edge detection is an important and widely used concept for extracting features of an image and is used as a base step for further algorithms (i. e. *Hough Transform*). The idea was to implement an edge detector using the *Sobel Operator*, since this is a common used filter that considers the horizontal as well as the vertical direction of an edge and uses a three by three neighborhood to approximate the gradient of the image. The goal was to speed up the calculation of the *Sobel Operator*, due to the fact that it took too long to calculate the edge image in every cognition frame of the robots (30 fps). On the other hand this approach is not designed as a replacement for our current vision system, but as a base step for the vision system of the coach robot. Since the coach robot is only allowed to send a message every 10 seconds the evaluation of every frame is not required.

The *Streaming SIMD Extensions* (*SSE*) are a command set developed for Intel processors[1]. *SIMD* stands for *Single Instruction, Multiple Data*, i.e. a single operation is calculated on multiple data at the same time. *SSE* defines new assembler commands and is used to parallelize similar calculations, that need to be processed on a large amount of data. Since edge detection operators like the *Sobel Operator* require the computation of a 2-dimensional convolution, the goal was to speed up this calculation using *SSE*. For further reading on *SSE* instructions, see the *Intel 64 and IA-32 Architectures Software Developer's Manual* [14].

3.1 SSE in Detail

To operate on multiple data at the same time a new type of processor registers had to be developed. Therefore *SSE* introduces eight new registers named *XMM0* to *XMM7*. Unlike the previous command set standard for parallelized calculations called the *Multi Media Extension* (*MMX*), these registers can hold floating point numbers. Each of the eight registers has a size of 128-bit[2] and is able to store one of the following combinations of different data types:

- 2 floating point numbers (each 64-bit, double precision)
- 4 floating point numbers (each 32-bit, single precision)

[1] *AMD* processors also support most of the *SSE* versions

[2] On recent Intel processors that support the *Advanced Vector Extensions* 256-bit registers are available, but the current *NAO V5* and its Intel Atom processor only have the 128-bit *SSE* registers.

- 4 integers (each 32-bit, signed or unsigned)
- 8 integers (each 16-bit, signed or unsigned)
- 16 integers (each 8-bit, signed or unsigned)

SEE also defines a set of standard commands, that are used to do calculations with the registers. These are:

- Load and store
- Pack and Unpack
- Logical operations (i.e. AND, NAND, OR, XOR)
- Arithmetical operations (i.e. bit shifting, addition, average, maximum, minimum, multiplication, division, subtraction)
- Comparison operations (i.e. equal, greater, less, greater equal, etc.)

For a full list of available commands refer to [14, 15].

It has to be noted, that not all operations are available for every register type, e.g. there is no bit shift operation for 8-bit values. Furthermore some of theses operations may cause an overflow, e.g. addition. For most of the operations that may cause an overflow there is a second, almost identical operation. This second version will trigger a calculation using (unsigned) saturation arithmetic if an overflow occurs.

Since the *SSE* commands are a set of assembler commands, various compilers also bring a set of *SSE* intrinsics. To make use of these functions one of the standard headers (Table 3.1) has to be included. The headers include all of the previous versions' headers and hence only the highest usable header has to be included. For a full include graph refer to [5] (Clang documentation). The highest supported *SSE* version of the *NAO* version 5

Table 3.1: Compiler intrinsics and corresponding C++ headers.

Header file	Version
<mmintrin.h>	MMX
<xmmintrin.h>	SSE
<emmintrin.h>	SSE2
<pmmintrin.h>	SSE3
<tmmintrin.h>	**SSSE3**
<smmintrin.h>	SSE4.1
<nmmintrin.h>	SSE4.2
<ammintrin.h>	SSE4A
<wmmintrin.h>	AES
<immintrin.h>	AVX
<x86intrin.h>	All available intrinsics (Clang and GCC, x86)
<intrin.h>	All available intrinsics (MSVC)

is *SSSE3*. With these intrinsic functions every assembler command defined by *SSE* can be executed within C++ code without using the `__asm` command. Most of the common used compilers (GCC, Clang, MSVC) will perform optimization steps while translating the code into the corresponding *SSE* commands. Therefore the resulting assembler code might differ from the C++ code in terms of optimized order or added commands.

3.2 Sobel Operator

The edge detection method used in this approach is the *Sobel Operator*. This operator approximates the gradient (i. e. the first derivative) of the image intensity by calculating the 2-dimensional discrete convolution of a filter mask and the image intensity values. There exist various other filters using different sized or different valued filter masks, e. g. the *Prewitt Operator*. The advantage of the *Sobel Operator* is that it also takes the direction of the edges into account and overall presents a rather robust method. The filter masks used by the *Sobel Operator* are 3 by 3 matrices as shown in Equations (3.1) (horizontal) and (3.2) (vertical) [22, p. 21].

$$M_x = \begin{pmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{pmatrix} \tag{3.1}$$

$$M_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{pmatrix} \tag{3.2}$$

For each pixel the sum of the intensity values multiplied by the corresponding values of the filter masks are calculated. Let I denote the 2-dimensional image. The 2-dimensional convolution can be written as

$$G_x = M_x * I \tag{3.3}$$

$$G_y = M_y * I \tag{3.4}$$

If the two results for the horizontal and vertical directions are combined, the gradient magnitude (i. e. the resulting intensity) can be calculated:

$$G = \sqrt{G_x^2 + G_y^2} \tag{3.5}$$

Furthermore the direction of the gradient (not used in this approach) is:

$$\Theta = \operatorname{atan2}(G_y, G_x) \tag{3.6}$$

3.3 Using SSE to Calculate the Sobel Operator

Using *SSE* to calculate the *Sobel Operator* requires loading the image into the *SSE* registers and computing the convolution. In this approach 14 results are calculated at the same time. This cannot be done without approximation, because the values itself have a size of 8 bit and hence sums of two or more values may result in an overflow. The following sections describe the process of calculating the *Sobel Operator* with *SSE*.

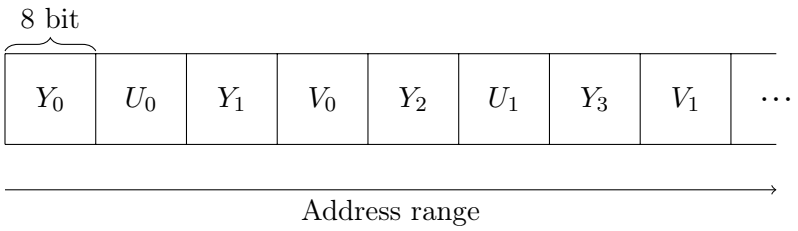


Figure 3.1: The robot's image in the main memory.

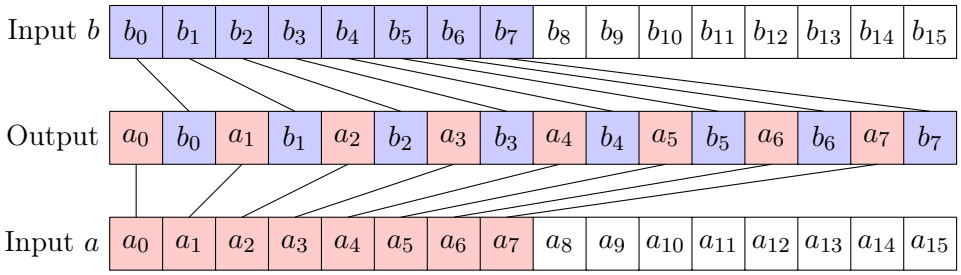


Figure 3.2: Unpack lower halves for 8-bit values (`_mm_unpacklo_epi8(a,b)`).

3.3.1 Loading the Image Using SSE

One major problem is loading the image from the main memory into the *SSE* registers. The robot's image is captured in *YUV* color space [28]. In addition, every two pixels share the color information, but each pixel has a separate intensity value. This standard is called *YUV422* [33]. Figure 3.1 shows how the robot's image is stored in the main memory. Since the *Sobel Operator* only needs the intensity values, these should be the only values loaded into the *SSE* registers. Loading the intensity values one by one would cause a copy of every value, because the load operation of *SSE* takes 128 consecutive bits in memory and stores these in a *SSE* register. For these situations *SSE* defines *unpack* and *pack* operations. An *unpack* operation takes two registers as input and will copy half of the first input and half of the second input into a new register. Depending on the command, it will store either the two first halves or second halves in the new register. Furthermore the values will be copied alternating, i. e. the first value will be from input one, the second from input two and so on. Figure 3.2 shows how the *unpack* operation works for 8-bit interpreted registers. These operations can now be used to deinterlace the YUV values, i. e. the intensity values are stored consecutively and in the desired order. This is done by loading two successive chunks of 128 bit image data into two *SSE* registers and then performing four lower and four higher *unpack* operations. The results are stored in two registers, one containing the intensity values and the other one containing the color information. Figure 3.3 shows the register contents during each *unpack* step.

3.3.2 Calculating the Convolution

With the task of loading the intensity values into *SSE* registers accomplished, the calculations have to be done subsequently. As shown before 16 values are loaded into one register. Hence, an approximation step needs to be done to prevent overflow, because the intensity

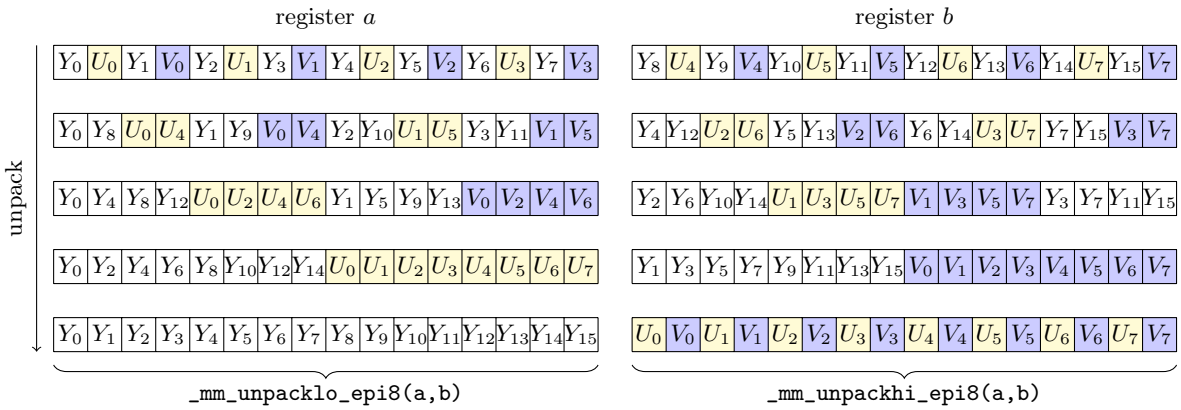


Figure 3.3: Deinterlacing of YUV422 pixels using *SSE unpack* operations.

values range from 0 to 255 (unsigned 8-bit). Considering one of the sums (horizontal in this case) that needs to be calculated for every pixel $P_{i,j}$ at the position (i, j)

$$\begin{aligned} G_x(P_{i,j}) = {} & P_{i+1,j-1} + 2 \cdot P_{i+1,j} + P_{i+1,j+1} \\ & -P_{i-1,j-1} - 2 \cdot P_{i-1,j} - P_{i-1,j+1} \end{aligned} \tag{3.7}$$

it follows that these sums range from $-4 \cdot 255$ to $4 \cdot 255$. Since only the absolute value of the sum is needed (i. e. only the gradient magnitude is computed), this sum can be split up into a negative and a positive part. These two partial sums then range from 0 to $4 \cdot 255$, which makes dividing the initial mask by 4 sufficient to prevent overflow, i. e. Equations (3.1) and (3.2) become

$$M'_x = \frac{1}{4} M_x = \begin{pmatrix} -\frac{1}{4} & 0 & +\frac{1}{4} \\ -\frac{1}{2} & 0 & +\frac{1}{2} \\ -\frac{1}{4} & 0 & +\frac{1}{4} \end{pmatrix} \tag{3.8}$$

$$M'_y = \frac{1}{4} M_y = \begin{pmatrix} -\frac{1}{4} & -\frac{1}{2} & -\frac{1}{4} \\ 0 & 0 & 0 \\ +\frac{1}{2} & +\frac{1}{2} & +\frac{1}{4} \end{pmatrix} \tag{3.9}$$

Another issue is the random access of the different values, that is used in the non-*SSE* approach. As it can be seen in Equation (3.7) the computation for one pixel requires the access of the eight neighbored values. Computations in *SSE* can only be done strictly vertical, that means if an addition is performed every two first values, every two second values and so on will be added. This can be resolved by loading three rows of data and then shifting these rows to the right according to the convolution. Figure 3.4 shows how this is done in the horizontal case. Note that in *SSE* a bit shift for every value or a bit shift for the whole register can be performed. In this case we need to shift the whole register by the number of bits of one value (i. e. 8-bit). P_1 is the first element of the non shifted register, P_2 is the first element of the register shifted by 1 and P_3 is the first element of the register shifted by 2. If the registers shifted by 2 are multiplied by 2 and all registers are added together the resulting register will contain the desired sums. See Figure 3.5 for further explanation on how the sums are calculated with the bit shifted registers. A similar procedure using different shifts is necessary for the vertical sums.

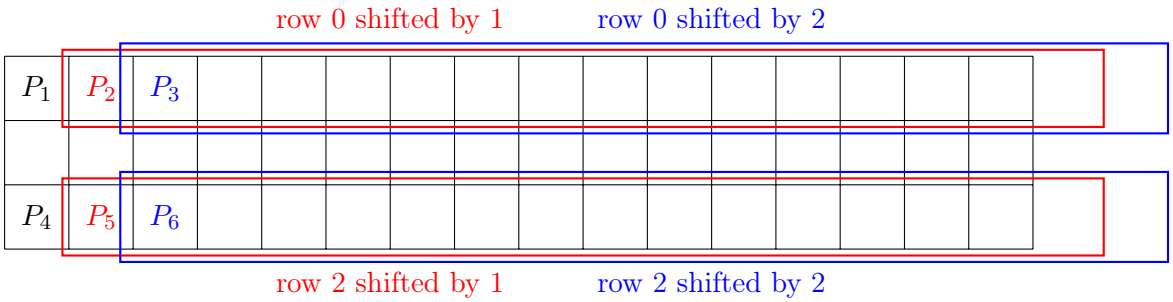


Figure 3.4: Row shifting used for calculating the convolution, horizontal sum. P_1 to P_6: Pixel intensity values loaded into *SSE* registers.

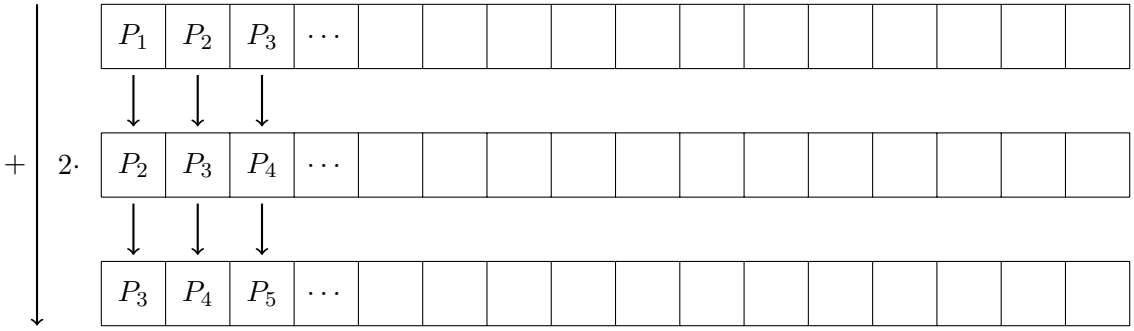


Figure 3.5: The three resulting registers from bit shifting the top row can now be used to calculate the correct sum without using random access.

The last step is to calculate the magnitude of the gradient. Since this requires two power and a square root operation, these exact operations can not be done with integer intrinsics. Furthermore this would create another overflow issue and these are more complex operations than addition or bit shifting. That is why an approximation is used again (see [11]):

$$\sqrt{G_x^2 + G_y^2} \approx \alpha \cdot \max(G_x, G_y) + \beta \min(G_x, G_y) \tag{3.10}$$

Choosing $\alpha = 1$ and $\beta = 0.25$ gives a largest error of approximatively 11.61% and a mean error of approximatively 0.65%. There are values for α and β that yield better results (see [11] for experimental results), but by choosing 1 and 0.25 one register does not need to be altered at all and the other one is merely bit shifted by two. It follows that the magnitude estimation replaces two power and one square root operation with one maximum, one minimum and one bit shift operation, while still being close to the exact value.

3.4 Results and Outlook

From a theoretical point of view the *SSE* approach should speed up the *Sobel Operator* by more than 14 times compared to the sequential approach. This follows from the simultaneously computation of 14 results and the approximation of the magnitude. Furthermore the classic approach does not make use of pointer arithmetics, but instead uses a getter-function to get the pixel values. Table 3.2 shows the timing results for the upper camera

Table 3.2: Experimental timing results of the *Sobel Operator* on a NAO V5 after approximately 1 min.

Approach	Minimum (ms)	Maximum (ms)	Average (ms)	Frequency Avg (fps)
Classic	370.157	379.786	375.283	2.435
8-bit *SSE*	9.490	11.166	10.234	29.654

image (1280×960 intensity values) of the robot evaluated with the stopwatch function of our framework's simulator.

It can be seen, that the speedup is indeed larger than 14 times. With the *SSE* approach it is now possible to calculate an edge image in every cognition frame of our framework. Nevertheless this can not (at this point of time) replace or enhance our existing vision system, which detects the ball, the goal, the field lines etc. in approximately 20 ms without using the *Sobel Operator*.

The method used in this approach could easily be transferred to more recent processors that use newer *SSE* versions with larger registers. If for example a processor with 256-bit registers could be used the speedup may be doubled. Another idea would be to pre-calculate a matrix containing the magnitude computation results. This would replace the magnitude estimation with a simple lookup in the table, but would require more space in the main memory.

4 Field Line Detection using Progressive Probabilistic Hough Transform

Manuel Barbi

A reliable field line detection is crucial to determine the location of important elements within a field representation and thus to create a global overview of what is happening upon the field. *Hough Transform* [16] is a promising candidate for this application, since it yields practical results. However, *Standard Hough Transform* is far too time consuming to be applicable within some kind of time constraints, as present in a running game. Therefore we try to utilize a more efficient version of the algorithm.

In this section we discuss the functional principle of the *Progressive Probabilistic Hough Transform* as described in [23], customizations we performed to make it more suitable to our needs and preparation of the results.

4.1 Progressive Probabilistic Hough Transform

The *Hough Transform* is a popular method to extract geometrical primitives like straight lines or circles from a given digital image if it is possible to specifiy a parametrical description. The basic idea of *Hough Transform* is to map the input data using a suitable transformation into a parameter space (see Figure 4.1), where local maxima indicates the occurrence of the specified feature. An accumulator is used in order to determine peaks in point's votings.

However, the *Standard Hough Transform* is very expensive in execution time since it has to consider a multitude of pixels to detect features. *Probabilistic* or *Monte Carlo* versions of *Hough Transform* algorithm exploit the circumstance that only a fraction of the points

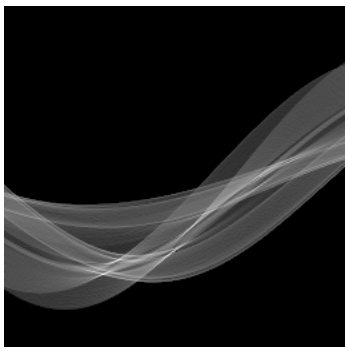


Figure 4.1: Representation of a parameterized Hough space.

voting is necessary to detect lines reliably. The *Progressive Probabilistic Hough Transform*, as described in [23], minimizes the computation time by randomly selecting points for voting. If a value within the accumulator exceeds a certain threshold, it may be considered as an indicator for a detected feature.

The *Progressive Probabilistic Hough Transform* algorithm proceeds as follows:

```
1: while not input image mask is empty do
2:    Update the accumulator with a randomly selected pixel from the input image.
3:    Remove the selected pixel from the input image mask.
4:    if highest modified peak in the accumulator is exceeding the voting threshold then
5:       Find longest segment of pixels specified by the peak not exceeding a certain gap.
6:       Remove the pixels in the segment from the image input mask.
7:       Unvote all pixels from the accumulator that has voted previously.
8:       if the line segment is longer then the minimum specified length then
9:          Add line segment into the output list.
10:      end if
11:   end if
12: end while
```

Algorithm 4.1: Progressive Probabilistic Hough Transform algorithm.

4.2 Specific Application

For the *Hough Transform* to generate useful results, it has to be executed on an edge image, rather than the raw image. In our case a *Sobel* processed image is used, as mentioned within Chapter 3. Since the *Sobel* image, depending on the conditions of illumination, may contain a considerable amount of noise, we apply a lowcut filter that discards pixels with a brightness value lower than a dynamic threshold. The lowcut threshold ϕ_{lowcut} is calculated by detecting both the highest γ_{max} and the lowest brightness value γ_{min} within the *Sobel* image and then define a portion c, to get a relative value in between:

$$\phi_{lowcut} := \gamma_{min} + c(\gamma_{max} - \gamma_{min}), c \in [0, 1] \tag{4.1}$$

We restricted the number of points, considered within the *Hough Transform* even further, by applying the field color based field detection as described in Chapter 5, to eliminate pixels outside the field (see Figure 4.2). The list of points regarded within the *Hough Transform* was implemented as a ring buffer with a fixed size. Thus points in the lower section of the image may override points from the upper border which usually are negligible anyway. However, we obtain a deterministic and acceptable upper bound for execution time.

Moreover, we were able to reduce the execution time by diminishing the number of angles that are examined within the *Hough Tranform*. However, eventually we abondened this approach since the lines within the image did not fit the few angles close enough and therefore resulting in various line segments for the same line, which were hard to merge appropriately afterwards.

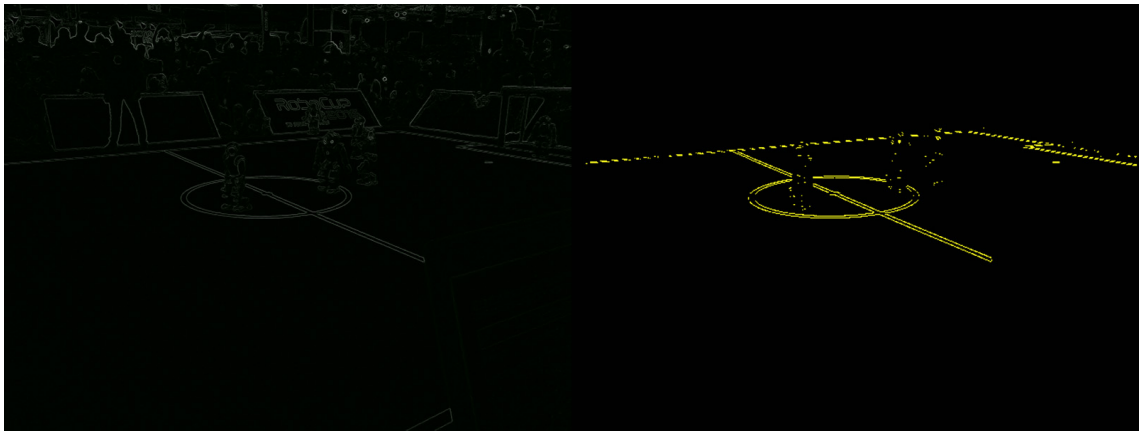


Figure 4.2: Sobel image without and with lowcut filter and field detection.

We provide some parameters to fine tune the line detection after compile time like minimum line length, maximum line gap and a probability threshold for the detected lines. Our goal was to identify values which are suitable for most situations. In practice it turned out to be difficult to find universal values. Due to differing conditions of illumination, various levels of contrast and moving obstacles blocking the view it's nearly impossible to find values that fit every situation. Furthermore, an adjustment of the parameters may cause unwanted side effects. Too short minimum line length can result in detected line segments on the center circle, too long ones exclude short parts of partial covered field lines and too high values for maximum line gap may cause detection of lines beyond the field.

4.3 Merge Algorithm for Line Segments

Usually field lines are not recognized as one contiguous line but interrupted by obstacles or seperated into various segments due to angle mismatch. The relative low camera resolution enhances this effect even further. To address this problem, we introduce a simple and efficient greedy algorithm to merge these line segments back into one line.

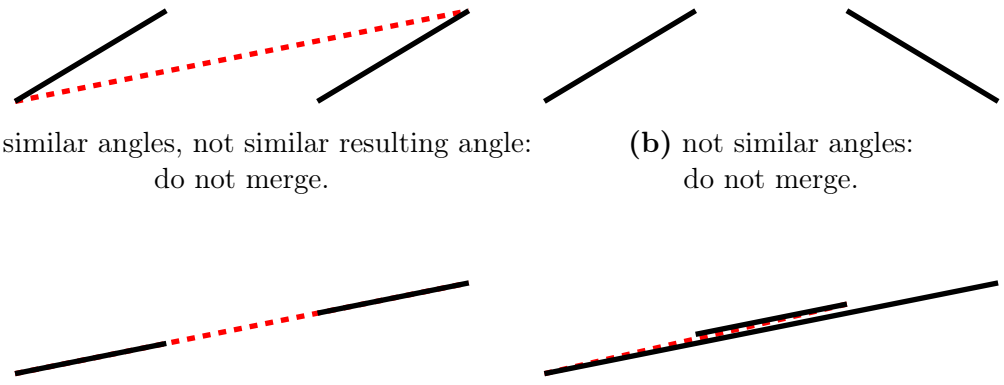
After executing the *Hough Transform*, we obtain a list of detected line segments in no specific order. It is desirable to look at each segment only once and thereupon to be able to decide whether or not to merge it with other segments already looked at. The key idea is to manage a list of already merged lines as starting points for line segments to come. That means for each line segment the algorithm iterates over the output list which is initially empty and either merges the line segment with the first found match or creates a new starting point when no match was found at all (see Algorithm 4.2).

In order to decide, whether two line segments belong together, we assume that two line segments can be merged regardless of their distance if both line segments have a very similar angle and if the merged line as well results in a similar angle (see Figure 4.3). In practice this approach proved to be sufficiently efficient and yields useful results. However, it is crucial to find a suitable condition to determine the similarity of two angles. Too small angle tolerances leads to rermaining fragments of lines, however, too big ones result in diagonal merging of the two parallel edges of a field line.

Input: Line[] in, Line[] out *// out is empty and has the same capacity as in*

```
1:  count = 0;
2:  for n = 0 to in.length − 1 do
3:      match = false;
4:      for i = 0 to count − 1 do
5:          Line merged = merge(in[n], out[i]);
6:          if merged and out[i] have a similar angle then
7:              out[i] = merged;
8:              match = true;
9:              break;
10:         end if
11:     end for
12:     if not match then
13:         out[count] = in[n];
14:         count = count + 1;
15:     end if
16: end for
```

Algorithm 4.2: Greedy algorithm for merging cohesive line segments.



(a) similar angles, not similar resulting angle: do not merge.

(b) not similar angles: do not merge.

(c) similar angles, similar resulting angle: merge.

(d) similar angles, similar resulting angle: merge.

Figure 4.3: Various cases of line segments.

Figure 4.4: Detected line segments over various frames.

4.4 Results

In order to detect at least four enclosing line segments of one field half, we performed numerous tests to fine tune the parameters of the *Hough Transform* under various conditions. The results are very fluctuant and usually the lines are not detected with full length or just one of the two parallel edges limiting a field line is recognized (see Figure 4.4). Due to insuffiecient contrast, backmost lines are often not regognized at all. Usually the Coach robot is not positioned exactly at the center line but shifted right or left. Thus the center line is not shown on the camera image when the Coach robot observes the goal area of the corresponding field half. Although our method yields fluctuant yet usable results we decided to abondon this approach for automatic field localization, since there are too many disturbing factors which make a reliable performance impractical.

5 Field Detection Based on HSI-Color Space

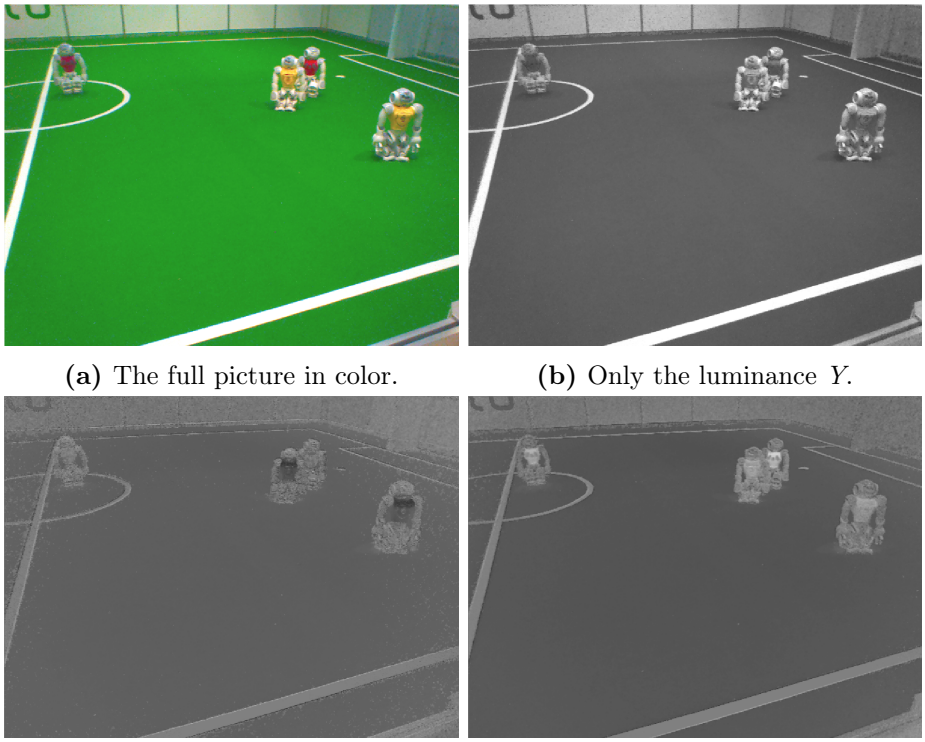
Julian Hannich, Marcel Eilers

One of the goals of this year's project group is to implement the vision of the new coaching robot. Part of the vision is to detect the field and the ball position. These information are taken from the upper camera of the coaching robot.

When working with an image, it is always useful to define some limitations of the scanned pixels. Scans over the complete image are usually very time consuming. For this reason, there should be a limitation in the picture to find the ball or other robots. Granted on the actual situation it is easy to exclude parts of the image of the field robot by simple calculations like the horizon. Depending on the head angle of the field robot, almost half of the picture can be excluded. Since the current implementation for the field robot does not need to know the outline accurately, the field outline is sufficient for the moment.

The following problems had to be faced while implementing the field detection for the coaching robot. Working with the coaching robot we require robust information, where the field starts and where it ends. There is a calculation, returning an approximate field outline already, but it is not accuratly enough for the coaching robot. It is using the position and angle of the joints to estimate the horizon. The algorithms for the coaching robot based on the fact, that the robot does not know his precise position. In the meantime, the position of the robot is measured before using the coach in a match. Even with this information, a robust field detection can save a lot of computing time on some algorithms. While the field robot can act on the assumption that there is field on the lower end of the image, the coaching robot cannot. Tests with the coaching robot showed, that depending on where the robot is placed, the table is seen on the lower end of the image. Also differences in the field brightness are much more evident for the coaching robot than for the field robot. This problem is generated by the higher position and the resulting wider view of the field. Due the wider view of the camera, its not comparable with the lower camera on the field robot which has a high angle position, too.

The field detection is based on the recognized field color. Calculating the field color is important, because many algorithms are depending on it. As mentioned, the higher position of the coaching robot makes it more difficult to find the best field color. The results of our current field players's color detection were not satisfying when tested on the coaching robot in some situations. Through testing we found that switching to a different color space, like the HSI-color space, was beneficial to detecting the soccer field's color. The reasons for this will be explained in this section.



(a) The full picture in color. **(b)** Only the luminance Y.

(c) Only the Cb value. **(d)** Only the Cr value.

Figure 5.1: The pictures show an actual view from the coaching robot with YUV-color space.

5.1 Field Color Detection with HSI-Color Space

There are several color spaces, each with its own advantages and disadvantages depending on the application. The old field color detection uses the $YCbCr$-color space.[1] The output of the camera of the robot [28] is a similar color space, called *YUV422*. The *YUV422*-color space uses three different components to define a pixel and is the standard in analogous TV-broadcast [4]. $YCbCr$-color space is used in digital TV-broadcast. As both using the same components, the conversion is easy. The first value, the Y value, represents the luminance of the pixel. With this value it is possible to create a black and white picture. To get color into the image, the values Cb and Cr are used (chromatic components). The chromatic components are created by color differences. The different values are displayed in the following picture. All three components combined create the colored image which can be seen in Figure 5.1.

As the YUV-color space has two components defining the color, it is difficult to calculate the best fitting color for the field color detection, because it needs to consider the relation of each component to each other.

From the higher view of the coaching robot, tests showed that the differences in the field color are too high to calculate it robustly. Furthermore, when scanning for other colors like the red of the old ball or some jersey colors, it can become difficult to set this color

[1] The approach from 2012 can be seen in Ingmar Schwarz, "Kalibrierungsfreie echtzeitfähige Bildverarbeitung im Kontext des RoboCup" (Diploma thesis, TU Dortmund, 2012)

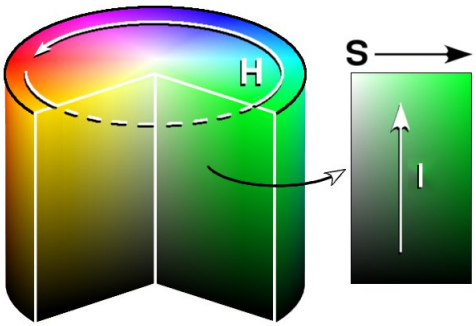
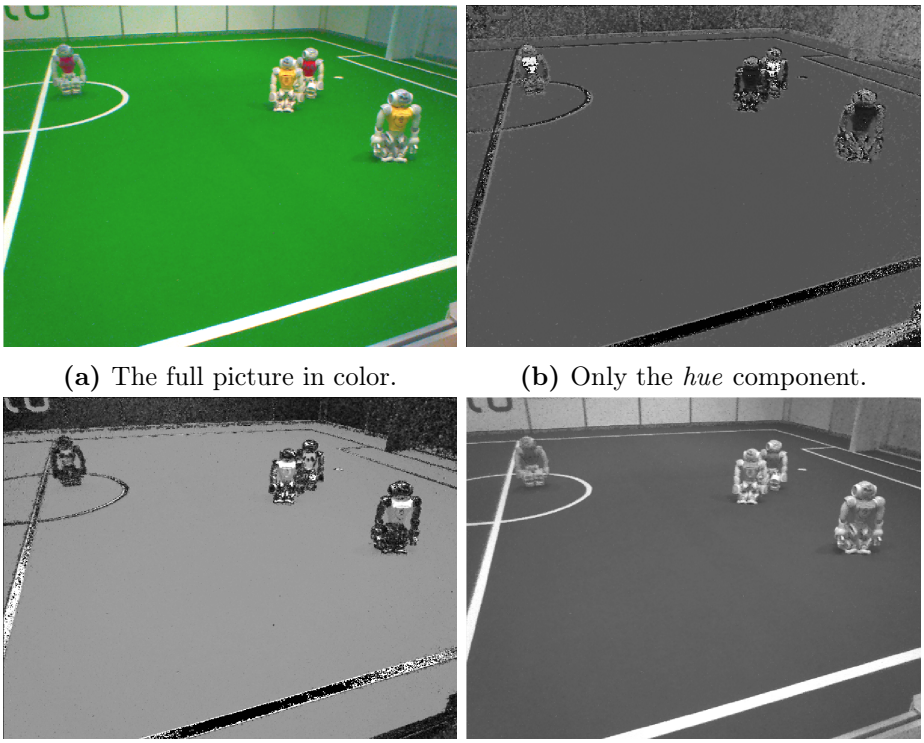


Figure 5.2: *HSI* color space cylinder [37] (Graphic modified from *HSV* to *HSI*).



(a) The full picture in color.

(b) Only the *hue* component.

(c) Only the *saturation* component.

(d) Only the *intensity* component.

Figure 5.3: The pictures show actual views from the coaching robot with *HSI*-color space.

as the two chromatic components are related to each other. To find different colors easily, we changed the color space to the *HSI*-color space. With this color space it is possible to find colors in a picture with only one value. The Figure 5.2 shows the function of the *HSI*-color space.

The component for the color is represented by the *hue H* value. It is represented by an angle, e.g. zero degrees represents the color red. The other components are the *saturation S* and the *intensity I*. Depending on the implementation, these components are represented by a percentage. The Figure 5.3 shows an image taken with *HSI*-color space, which is decomposed into its components.

Figure 5.4: Test of the calculated field color with multiple histograms.

It is difficult to find a field outline clearly in the component representing the hue, because the background fence is recognized as a nearly green color. To differentiate it from the field, the saturation must be considered, because there is a clear line in the saturation.

This is why not only one component should be used in finding the best field color. We tested several approaches which showed different results. In the following, we will discuss some of them which were implemented and tested on real images.

5.1.1 Using Simple Histograms to Detect the Field Color

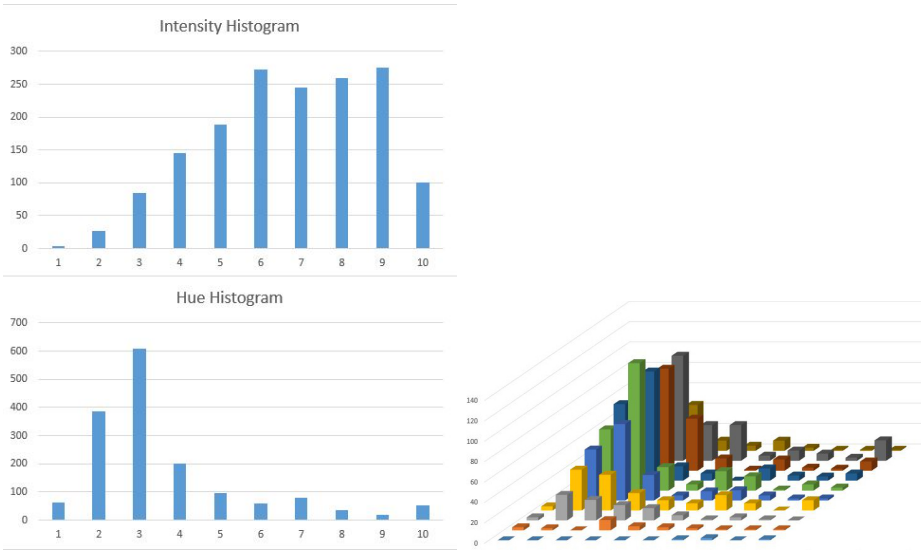
The first approach was to use an algorithm similar to the old field color detection. This approach was good in some situations and had some improvements compared to the old algorithm. One example of the output of this algorithm can be seen in Figure 5.4.

In Figure 5.4 the whole image is scanned after the field color detection. Each pixel, which is recognized as field color is marked red. As it can be seen, the result is very good and only a few points in the background are recognized as field by mistake. To achieve this result, we analyzed the histograms of the hue and saturation component.

As in this situation it is easy to detect the field color, it should not be used as a benchmark. In situations with a worse field illumination this simple approach returns a poor result, since it got some limitations.

5.1.2 Using Multi Dimensional Histograms to Detect the Field Color

The second approach was to use multi dimensional histograms. First, we tried to look at two components of the pixels at once. Using this, we got a three dimensional histogram showing the distribution of pixels with two components. The advantage of this approach was, to look at two components at the same time instead of only one. Therefore the expectation was to decrease the amount of false-positives. When using only one component at once, we are losing alot information like if a pixel is green and highly saturated. In a simple histogram, we can only see that there is a green pixel, but we have no informations about the saturation. An example of this histogram is shown in Figure 5.5.



(a) 2D Histograms. **(b)** The associated 3D histogram.

Figure 5.5: Simplified histograms taken from a picture by the coach.

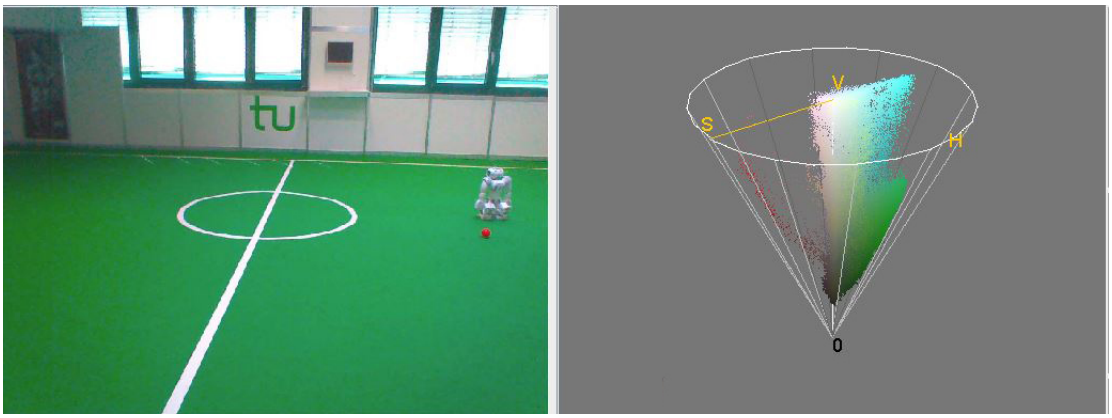


Figure 5.6: Test image and resulting distribution of pixels in 3D color space using 3D Color Inspector [3].

A simple algorithm was implemented to find the maximum value in the 3D histogram. If surrounding values were high enough they were considered as field color too. We wanted to decrease the amount of false-positive pixels which were recognized as field but at the end, the result was improved over the previous algorithms.

So we tested another approach looking at all three components at the same time. With this, we got a four dimensional histogram. In Figure 5.6 the distribution of *HSI* pixels in the 3D color space is shown. We used the software *3D Color Inspector*[2] for showing us the distribution and histograms. Using the same approach as previously, we get the histogram in Figure 5.7.

[2]3D Color Inspector can be downloaded on the website http://rsb.info.nih.gov/ij/plugins/color-inspector.html

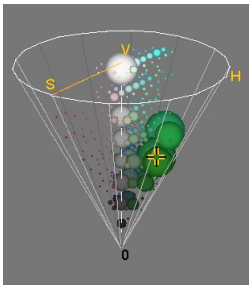
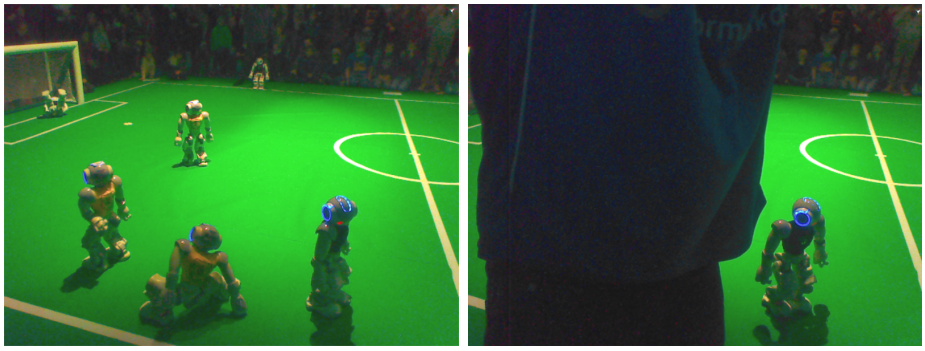


Figure 5.7: 4D histogram using 3D Color Inspector [3].



(a) Big difference in field color.

(b) Picture with a referee in between the sight of the coaching robot.

Figure 5.8: Two sample images from the tournament in munich 2015.

The result of using this approach was not satisfying. There were almost no false-positives anymore, but most of the field was not recognized as field anymore. So this approach was discarded after some tests.

5.1.3 Field Color Detection without Histograms

The biggest problem of the previous approaches appeared, when there were no dominant colors. In some situations, the field color is more a gradient instead of one color. When the light is not consistent, the field color becomes hard to detect. An example of such an situation is shown in Figure 5.8a. Another problem were the false-positives, when a referee was walking between coaching robot and field. With the histogram approach, the field was recognized on the shirt of the referee because it was the dominant color. The second problem is shown on the right side of Figure 5.8b.

To resolve these problems, we implemented a new algorithm which works without using any histogram. This approach uses scanlines to detect the field color. As shown in Figure 5.9 several scanlines are created from the center of the image. The color of the debug drawing indicates the pixel used. Green and yellow lines show pixels which are used to detect the field color. Red lines are not used, as they are too small. The algorithm searches for areas with only small differences between each scanned pixel. If a large difference was found, a new line starts. With some parameters which are easily adjusted before a match the scan can be limited on several pixels. For example the limitation can be defined by

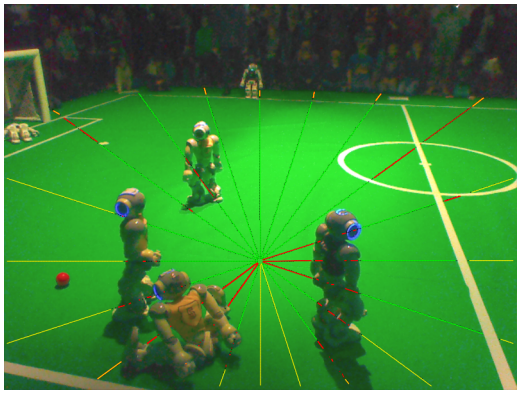


Figure 5.9: Scanlines from the center of the image to detect field color.

Figure 5.10: Green points indicating found field color.

pixels with a hue in a range from 70 to 100 (which is a yellow-green color). All other colors will be skipped. When a line is long enough, all of the pixels are used to create the field color. The maximum and minimum values of each component will be put together to get the field color.

5.1.4 Evaluation

The evaluation returned excellent results in almost every situation. In bad lighting situations and even when the referee comes into sight, the field was detected really good, as shown in Figure 5.10. The only problems are the reflecting surfaces of the robots. On some conditions, there is no difference between the color of the robot and the field color on the edges. This has to be concerned about the ball and robot detection. To solve this problem the algorithms could use scanlines to detect differences between the consecutive pixels. Restricting the field color detection to decrease the pixels found on the robots could lead to a bad detection of the field outline.

Tests on the situation in Figure 5.10 have shown that the new field detection recognizes over 90% correct pixels as field color. In most conditions it detects almost 100% of it. In average less than 1% of the found field pixel were false positives. The current approach only detected around 80% of the field or less.

Table 5.1: Runtimes of the field color detections.

Time	Field color detection ($YCbCr$)	Field color detection (HSI)
Minimum	0.109ms	0.351ms
Maximum	0.243ms	0.592ms
Average	0.149ms	0.384ms

The runtime is higher than the old field color detection as Table 5.1 shows. As the new field color detection is yet only be used on the coaching robot, the additional runtime should be nothing to concerned about.

5.1.5 Future Work

As the new field color detection is only an additional part of the vision, the next step would be to completely convert the framework to the *HSI*-color space. At the moment many functions are still using the *YCbCr*-color space. Because of the number of changes, it will need a lot of time changing the framework. The field detection and the ball and robot detection shown in a next chapter are using the *HSI*-color space already and are working well. The *HSI*-color space has some advantages over the *YCbCr*-color space like the easier representation of the color. The disadvantage of the *HSI*-color space is the conversion. The robot returns the picture in *YCbCr*-color space inherently. To use the *HSI*-color space, the pixels have to be converted beforehand which is currently achieved by a local look up table.

5.2 Field Detection

With the field color known, we need to find a precise hull around the field. For this use we created a simple algorithm, which determines the field outline by scanlines. It returns the coordinates of a convex hull so other algorithms can use them without difficulties.

5.2.1 Idea

In addition to using the field color with the *HSI*-color space we implemented an algorithm to find the correct field outline. This algorithm searches the endpoints of the field in the image. It is implemented with a simple scoring system. A comparable algorithm was found in [35]. It scans points vertically from bottom to top. If the pixel is recognized as field color, the score will be increased and if the pixel is no field color, the score will be decreased. The maximum score will be saved together with the Y-value. After scanning the whole picture vertically, the best score will be chosen as the end of the field. Figure 5.11 shows a scene with debug drawings demonstrating the algorithm. Every line represents a vertical scan.

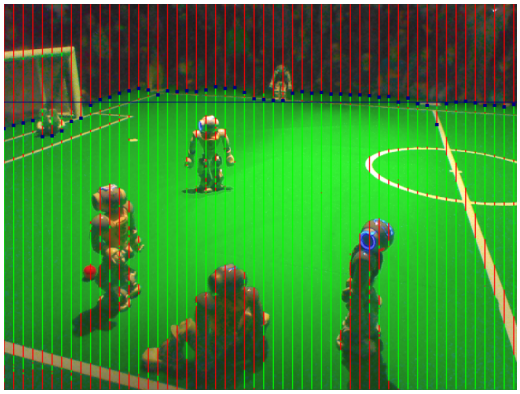


Figure 5.11: Debugdrawings of the field detection.

These scans skip some pixels to save computing time. Green lines represent found field color and an increase of the score, red lines detemine found no field color and a decrease fo the score. This way, the algorithm finds the correct field outline most of the time. Even if some field color is recognized outside of the field, the algorithm mostly finds the correct border. If the algorithm finds an outlier, it is removed by matching the previous, current and next boundary. This way, most of the outliers are removed and the next algorithm can start creating a convex hull.

5.2.2 Calculate the Convex Hull

To determine the area of the field, a convex hull of the pixels that have the field color is needed. The algorithm which is used to solve this problem is called *Graham Scan*. The idea is to construct a star-shaped polygon from the set of points and then to transform it into a convex polygon [18].

First of all the point with the minimal Y-coordinate is identified. If the result is ambiguous the point with the lowest X-coordinate is choosen as initial point. Starting from this point the angles between the x-axis and the lines that connect the initial point with all other points are calculated. The points will be sorted by this angle and connected in this order to a polygon. If there are points with the same angle, they are additionally sorted by their distance to the initial point. The result is a closed polygonal line consisting of all points [9].

The second step is to traverse this path and remove the points which are not part of the convex hull. The first two points are always part of the convex hull so that the first point that needs to be checked is the third point on the path. There are always three points that are kept in focus. In the investigation of a point C, his successor N and its predecessor P are considered. If these three points form a left turn, which means that they are ordered counterclockwise, point C is rejected. Point N remains the same, while the points C and P become their previous points. If the three points form a right turn, point C is kept and the next point will be checked. Figure 5.13 shows an example with some steps of the method [9].

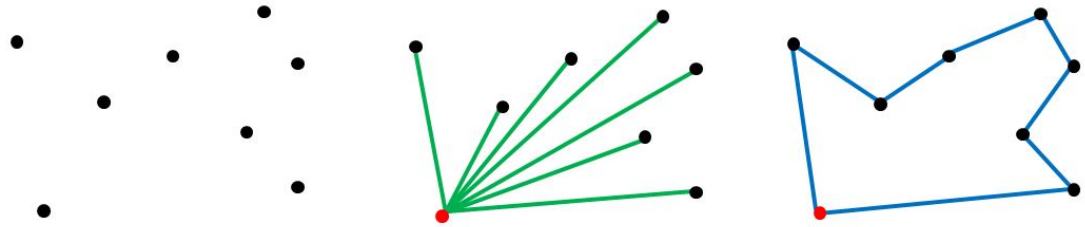


Figure 5.12: Construct the star-shaped polygon from the given points.

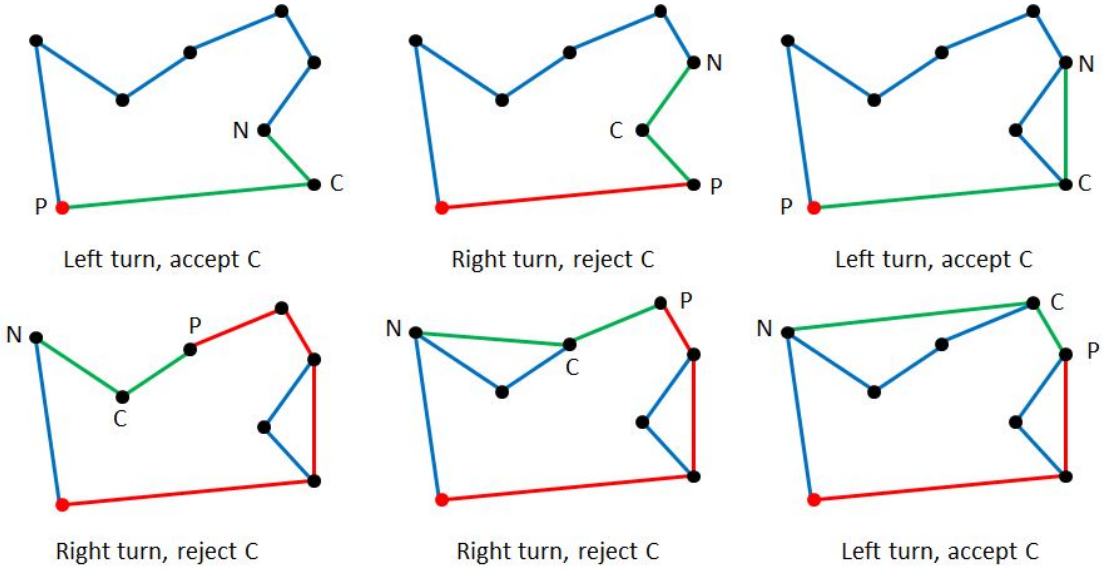


Figure 5.13: Traverse the graph and remove concave points.

After calculating the convex hull we want to know whether a given point is inside or outside the convex hull. The used algorithm also works for polygons that are not convex. In order to perform the check, an imaginary horizontal line from this point to the picture's right edge is drawn. Afterwards the number of intersections of the line and the contour of the polygon are counted. In case that the polygon is a convex hull, the result can only be zero or one. If the count of intersections is odd the point is inside the convex hull. If the point lies on a border line, it is qualified as a point lying inside the polygon. Otherwise the points lies outside the convex hull.

5.2.3 Evaluation

The complete field detection takes around 0.4ms depending on the number of vertical scans and the count of skipped pixels. At the moment, there are 54 vertical scans to detect the field outline. The number of vertical scans can be halved with only a little quality loss at the corners. Also the number of skipped pixels can be adjusted by a parameter. Adjusting

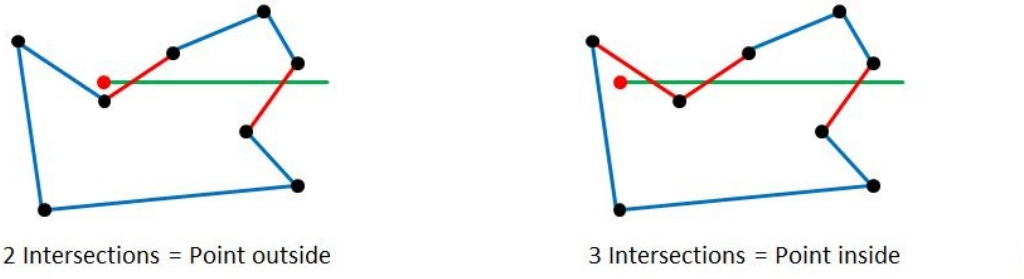


Figure 5.14: Examples of one point lying inside the polygon and one point lying outside.

Figure 5.15: Found convex hull around the field.

all this can decrease the computing time by around 0.2ms with still acceptable results. Figure 5.15 shows the output of the algorithms. The coordinates of the convex hull are saved as a representation named *FieldOutline* which can be used by other modules.

5.2.4 Conclusion

Using this convex hull can be beneficial. If the coaching robot knows the correct outline of the field, many pixels are excluded of the scan. For example, while searching for the ball, many pixels can be sorted out because they are outside the field. So it is a good way to save time required for computing. Another advantage is that several other useful algorithms, which need a good and robust field outline, can now be implemented. For example, an algorithm for detecting the goal posts, which amongst other things needs the field outline to identify the goal posts[3].

[3]For further reading on the approach to identify goal posts with the field outline see Pablo Cano, Yoshiro Tsutsumi, Constanza Villegas and Javier Ruiz-del-Solar, "Robust Detection of White Goals" (Paper, Universidad de Chile, 2015)

6 Robot Detection Based on Colors

Marcel Eilers

The approach of robot detection described in this section relies heavily on the correct determination of the playing field. It uses the results of the field detection, which are given in form of a convex polygon containing all detected field points in order to limit the search space. The area of the field is defined as the convex hull around the pixels that have the calculated field color. There are two essential conditions for a pixel in order to belong to a robot. First it must lie within the convex hull and secondly it must not be qualified as field color. Besides the robots also lines and the ball do fulfill these criteria. Because the lines have nearly the same color as the robots, all parts of the robot except the jerseys need to be filtered. The result is a set of pixels, all belonging to the jerseys of the robots. The next aim is to assign these pixels to the different robots in the picture. Finally the robots need to be categorized as two teams. The distinction is executed by means of the jersey color.

6.1 Method

The basis for finding the jerseys of the robots is an array that has the same size as the original picture. If one pixel is in the convex hull and does not have field color, there is a 1 at the corresponding position in the array, otherwise there is a 0. *DBSCAN* Density-Based Spatial Clustering of Applications with Noise (*DBSCAN*) [10] is an algorithm for clustering a set of points without knowledge of the number of clusters. The algorithm puts points that are close together in one cluster. Here the original *DBSCAN* algorithm is expanded. Before one point is added to a cluster, the algorithm checks if the point has a potential jersey color. Pixels belonging to field lines can be recognized by a high saturation value in the HSI color model. The result of the *DBSCAN* algorithm is a 2D-array in which the clusters are stored that contain the positions of the pixels with jersey color.

The Figure 6.1 shows an example about how the algorithm works. While the white pixels represent the zeroes in the array, the black pixels represent the ones, that should be clustered. Decisive for the assignment to a cluster is the neighborhood of the pixels. The currently checked pixel is marked with a red square. Here the eight surrounding pixels are choosen as the neighborhood, which is visualized with the red dashed rectangle. First step is to count the black neighbours. If this number is higher than a certain value, the current pixel is part of a new cluster. In addition to that, the black neighbour pixels will also be checked with the same routine. That means that if the count of their black colored neighbours is also higher than the threshold, they are added to the same cluster. If there are no other pixels in the queue that need to be checked, the cluster is completed as you can see in the third part of the figure. The next black pixel in the picture is then assigned

to a new cluster and the same procedure starts once again. The result can be seen in the last part of the figure.

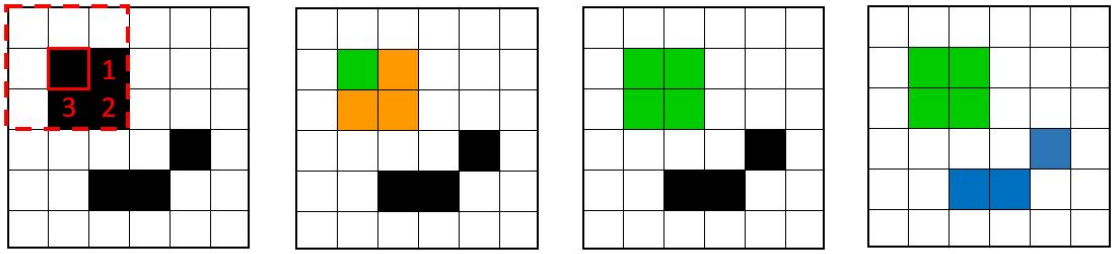


Figure 6.1: *DBSCAN* with a neighborhood of eight pixels.

The same initial situation is shown in the Figure 6.2, but in contrast to the previous picture the procedure with a neighborhood of 24 pixels is shown. As you can see, now the three pixels on the right bottom are neighbour pixels of one of the pixels of the first cluster. The result is that they are also assigned to the first cluster, which means that there is only one object found in the array whereas there were found two within the neighborhood of eight pixels.

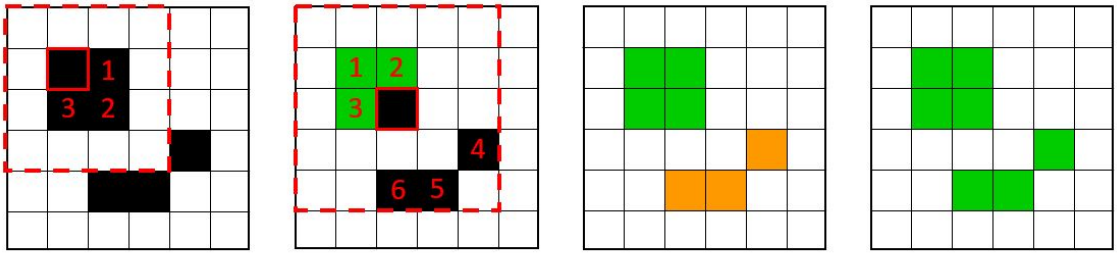


Figure 6.2: *DBSCAN* with a neighbouhood of 24 pixels.

After determining the pixels that belong to the jerseys and grouping them into clusters, the next task is to categorize them in two teams. The aim is to assign all the remaining pixels in two clusters just by their H-value in HSI color space. This problem will be solved by the *k-Means* algorithm. The result are two centroids that characterize the two main colors of the jerseys.

With this knowledge the results of the *DBSCAN* algorithm can be analyzed more detailed. Every pixel in one cluster can be assigned either to one or the other jersey color. After the assignment the number of pixels associated with a color can be determined, respectively. Each cluster is assigned to the color that constitutes the majority. Finally there are clusters that belong to the one team and clusters that belong to the other team. Because every cluster represents a robot, it's position in the picture and it's team membership is known. The result can be seen in the left part of Figure 6.3. The black and the white rectangles mark the two teams. To calculate the exact positions of the robots in field coordinates we need to know the position of their feet. Therefore the rectangle is analysed rowwise, starting in middle and moving on downwards. The Y-coordinate of the feet is defined as number of the first row in which all pixels have field color, as you can see in the right part of Figure 6.3. The corresponding X-coordinate is the center of the surrounding rectangle.

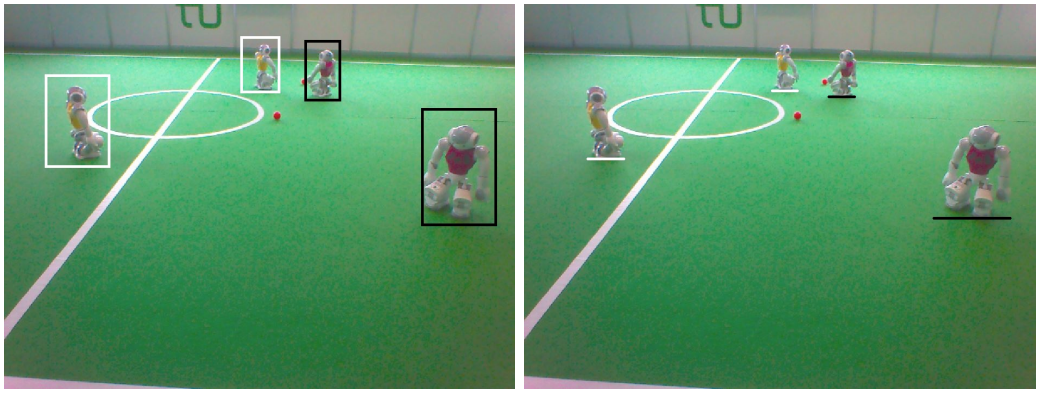


Figure 6.3: Result of the robot detection and the assignment to the teams.

6.2 Evaluation

If the algorithm finds just the clusters that match the positions of the jerseys, the assignment to the teams works in most of the times. But often there are also some other clusters found, that do not depend on the jerseys. These false positives are not filtered out and additionally influence the result of the *k-Means* clustering so that even correctly found clusters are assigned to the wrong teams. Furthermore, if one of the referees runs through the picture, the algorithm produces false positives.

7 Robot Detection of the Coaching Robot with Feature Points

Torsten Böttinger

Currently the fieldplayer's approach of robot detection uses a scan line method that detects interruptions within the field's green and treats players as obstacles. However, the coaching robot*'s* purpose is not only to recognize obstacles, but to analyze the semantics connected with them. It is interesting to find out, how many opponent players are located in the own half of the field for the players to decide how to behave. For example when they are located in the opponent's half of the field and are in possession of the ball, the coaching robot should be able to analyze whether it is opportune to perform a distant shot. For this task he has to recognize the own and the opponent players quite exactly. In this chapter a method is discussed how to handle this detection.

The chosen approach uses *feature points*, which are points within the image that are interesting to analyze. These *feature points* should be located mainly at the robots in the field the coaching robot is looking at. The *feature points* get clustered and the centroids of each cluster are used as representatives for each robot.

In the following sections we initially discuss the theoretical background of the *Feature Point Detection* and the clustering procedure, which is *k-means* in this particular case. Afterwards, we describe how these two methods are merged and used with some further improvements. Finally, we present some results from the real robot.

7.1 Features from Accelerated Segment Test (FAST)

The *FAST Feature Detection*, as published in [30] and [31], is explicitly build for realtime analysis e. g. of videos, whereas the *Harris Descriptor* or the *SIFT Descriptor* are significantly slower [30]. This serves as motivation to use the *FAST Detector* in the *NAOs'* context. The OpenCV implementation[1] is used.

7.2 The FAST Detector

The segment test considers points around a corner candidate p (see Figure 7.1). The detector checks whether there are n pixels that are contiguous and have an intensity I_n which is higher than the intensity of the corner candidate I_p. Let I_p be the brightness of

[1] The implementation can be downloaded under http://www.opencv.org/

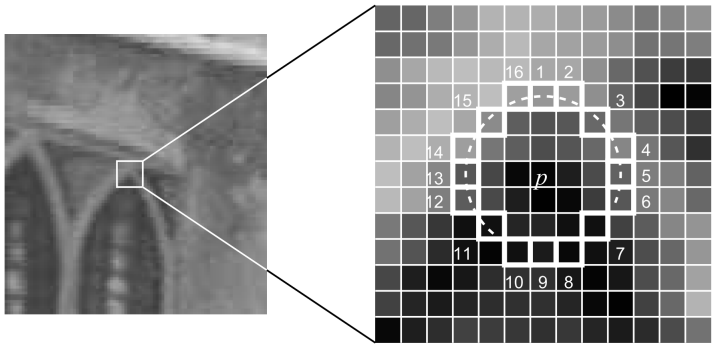


Figure 7.1: An example of the 16-neighborhood of the FAST detector.

the corner candidate p and I_n the brightness of one of the circle points p_n. t is a chosen threshold.

$$I_n \leq I_p - t \text{ (darker)} \tag{7.1}$$

$$I_n \geq I_p + t \text{ (brighter)} \tag{7.2}$$

$$I_p + t \leq I_n \geq I_p - t \text{ (similar)} \tag{7.3}$$

As it can be seen, a higher threshold t yields less *feature points*. Various neighborhoods are available to experiment with. Implemented are an 8-neighborhood with $n_8 = 5$, a 12-neighborhood with $n_{12} = 7$ and a 16-neighborhood with $n_{16} = 9$. Using a 16-neighborhood, it is very efficient to check whether a point is *feature point*, since only four points in the compass direction have to be tested.

7.2.1 Non-Max Suppression

For non-max suppression a score function is needed, which weights found *feature points*. The chosen function computes the sum of the absolute difference between the intensity of the corner point p and the intensity in the arc around p. With this score function it is possible to remove corner pixels, which are adjacent to corner pixels with a higher score.

7.2.2 K-Means

k-means is a problem originated in the cluster analysis. The goal is to approach a set of points with k centers. Given an integer k and a set of n data points X, the purpose is to find a set of centers C which minimizes the following potential function.

$$\phi(X, C) = \sum_{x \in X} \min_{c_i \in C} ||x - c_i||^2 \tag{7.4}$$

K-Means According to Lloyd

Lloyd's method to solve this problem is a heuristic algorithm [21] that is based on the local optimization of the center of mass:

1. Arbitrarily choose initial k centers. E.g. choose k points uniformly random from the data set.

2. Every point is assigned to that cluster respectively center which increases the potential function ϕ least.

3. The new centroids are chosen as center of mass of the clusters: $c_i = \frac{1}{|C_i|} \sum_{x \in S_i} x \; with \; i \in \{1, 2, ..., k\}$

4. Steps 2-3 are repeated until ϕ does not change anymore.

K-Means++

The k-means according to Lloyd cannot provide any performance guarantees. The result might be arbitrarily bad amongst others due to the seeding step. *k-means++* [2] improves this by seeding the centers depending on previous chosen centers using the following function:

$$D(x)^2 = \min_{c_i \in C} ||x - c_i||^2 \tag{7.5}$$

The k-means++ replaces the first step in Lloyd's algorithm:

1. The first center c_1 is chosen randomly from the set of all points. An uniform distribution of the points is taken as basis.

2. Seed center c_i by choosing $x \in X$ with a probability of $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$.

3. Repeat step 2 until k centers are chosen.

4. Steps 2-4 of the classic k-means algorithm of Lloyd are executed.

The expected value of this algorithm guarantees a value of the result's potential function that is at most $8(ln(k) + 2)$ times higher than the potential function of the optimal result [2].

7.3 Robot Detection

The OpenCV implementation of the FAST-Detector was imported into the *NAO's* framework and is located in a new module. The implementation uses every intensity value the camera provides. The result is a list of key-points that is ordered by the horizontal value of the point.

7.3.1 Methods to Eliminate Unnecessary Feature Points

The subsequent methods yield a result in which most of the *feature points* are located at the robots within the image.

The first implementation of the detector produced many points outside the field. These points have no significance for us because we are looking for robots on the field. To eliminate *feature points* outside the field the available field detection in Chapter 5 was utilized. To avoid an extra iteration over the result key-point list the check was integrated at the end of FAST algorithm each time before a feature-point is added to the result list.

After deleting all points outside the field there are some distracting points within the field, for example on fieldlines or around the goal. As these points all are sparse they can be eliminated by checking whether a small point density is within these regions. The algorithm checks if within a square around each point with side length d are at least n_{min} *feature points*. To optimize the iterative search through all key-points, the order of the point is used.

In different light situations, the number of feature varies and the mechanism to eliminate unnecessary points does not work properly. In this case an adjustment of the threshold solves the problem. For situations with too many points the threshold should be increased. In all other cases the threshold should be decreased.

7.3.2 Clustering of the Feature Points

The found *feature points* are clustered with k-means++ and the emerging centers are used for detection of robots. However, the results have to be processed further. For example clusters which does not reach a specified number of *feature points* are deleted.

In some cases two robots were combined into one cluster, although they were standing notably apart from each other. Since the resulting centers of these clusters are in regions without any *feature points*, this case can be identified by checking whether there are enough *feature points* within a certain distance around the centers. If such a cluster was found, the k-means++ is executed for the points within the clusters with $k = 2$. This is method 1.

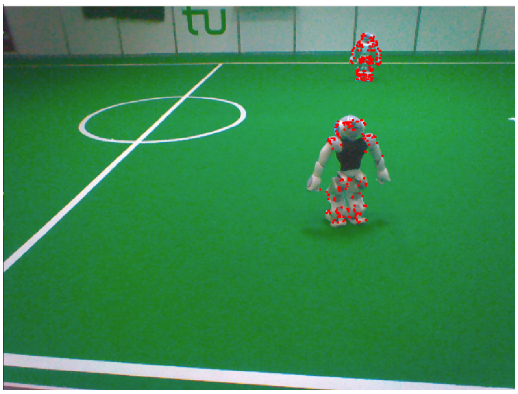
The previously introduced method 1 only handles clusters with a center in a region without any *feature points* around. However, we would like to obtain clusters which have the shape of a robot. Method 2 extends the k-means++ to provide information about height and width of the clusters. When the cluster has a wrong aspect ratio, the k-means++ can be executed again for all *feature points* within the relevant cluster with $k = 2$.

The resulting implementation is able to find robots as can be seen in the Figure 7.3. However, in many cases the clusters include more than one robot due to the recursive depth of the k-means++ executions of only one.

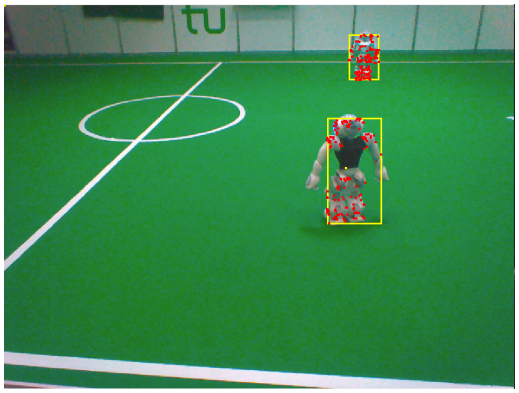
7.3.3 Timing Analysis

The runtime of this algorithm depends on the found *feature points*. With the methods above many irrelevant points are removed and a runtime for *feature-point* calculation plus *k-means* of 60 to 100 ms is reached.

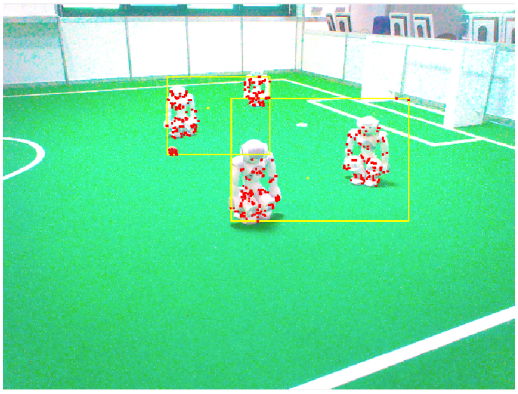
(a) FAST used on the upper picture of the *NAO* (threshold = 20).



(b) Sporadic *feature points* are removed (threshold=20).

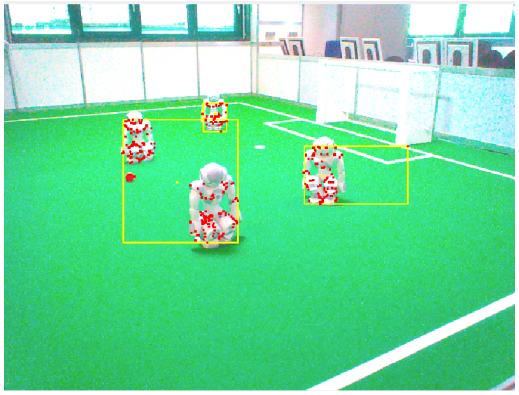


(c) k-means alone used in the robot picture works (threshold = 20).

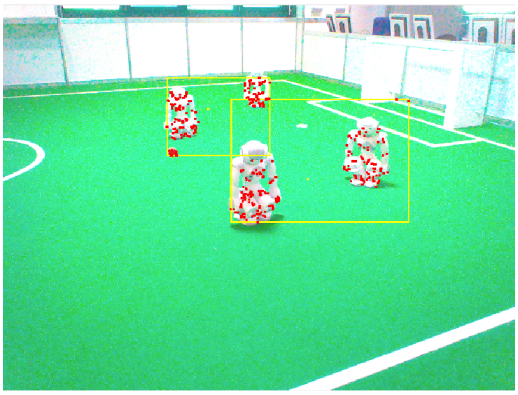


(d) k-means alone does not work well in this situation (threshold=30).

Figure 7.2: Different steps and situations of this algorithm.



(a) The clusters are handled with the method 1 (threshold = 30).



(b) The clusters are handled with the method 2 (threshold = 30).

Figure 7.3: Methods 1 and 2 of cluster postprocessing on real images.

8 Line Segment Classification

Christian Albrecht

This chapter describes an approach to classify the preprocessed line segments provided by the *Hough Transform* into the categories left, right, top, bottom and middle. However each class label can at maximum be assigned once. Furthermore the focus lies on the correct detection of the middle line segment because this allows (if the position of the ball is known) to decide whether the ball is located downfield or upfield. The knowledge of the approximate position of the ball on the field is necessary for the coaching robotrobot in order to derive a strategy.

8.1 Classification Algorithm

In the first step of the algorithm all line segments are classified in the categories horizontal and vertical. Initially each line segment is transformed from picture coordinates into an unknown coordinate system with help of the camera matrix. Afterwards the gradient angle is calculated. The transformation is necessary because it provides the same gradient angle independent from the orientation of the head in the yaw component. The class of a line segment $\mathbf{l}$ is then determined with the following rules:

$$-\frac{\pi}{4} < \mathbf{l} <= \frac{\pi}{4} \Longrightarrow \textbf{vertical} \tag{8.1}$$

$$\frac{\pi}{4} < \mathbf{l} <= \frac{3\pi}{4} \Longrightarrow \textbf{horizontal} \tag{8.2}$$

$$\frac{3\pi}{4} < \mathbf{l} <= \pi \Longrightarrow \textbf{vertical} \tag{8.3}$$

$$-\pi < \mathbf{l} <= -\frac{3\pi}{4} \Longrightarrow \textbf{vertical} \tag{8.4}$$

$$-\frac{3\pi}{4} < \mathbf{l} <= -\frac{\pi}{4} \Longrightarrow \textbf{horizontal} \tag{8.5}$$

Next the horizontal lines are analyzed more detailed because this case is much easier due to the fact that at most two lines have to be classified. In case that only one line has been extracted by the *Hough Transform*, it is considered as the top line when the Y-coordinate is lower than one half of the height of the image. Otherwise it is classified as the bottom line. If two lines have been detected, the algorithm labels the line segment with the smallest Y-coordinate as top line and the other one as bottom line, respectively.

In the next step the algorithm tries to determine the middle line. Therefore the intersection point of every vertical line segment and the bottom and top line (if available) is calculated.

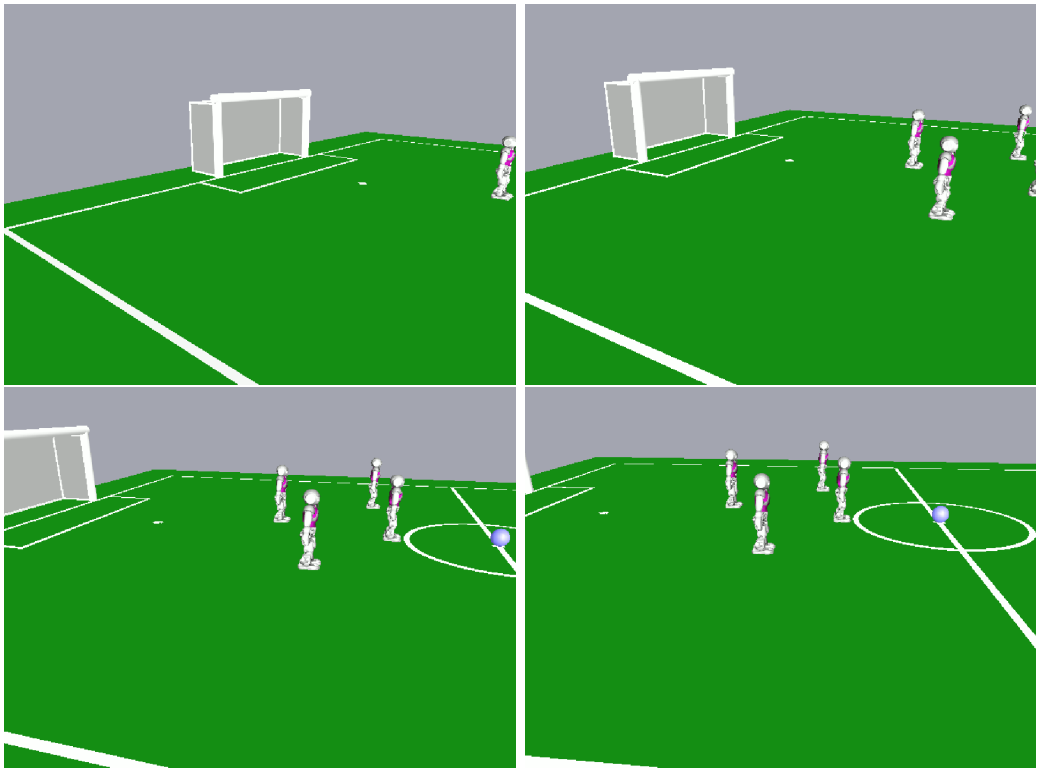


Figure 8.1: Transformation of the left field line while the head of the coaching robotrotates from left to right. The X-coordinate of upper point of the left field line segment is always greater than the one of the lower point.

If the point lies on the horizontal line segment and if it is sufficiently far away from the start and end point (distance of 0.005 and 0.995 in relation to the length of the line) the vertical line segment is considered as middle line and the further investigation is canceled.

Afterwards all remaining vertical line segments are examined and classified. At the start the vertical line segments are sorted from left to right. The approach exploits the observation that in case of the left line the X-coordinate of the upper point is greater than the X-coordinate of the lower point if the rotation of the head is limited (see Figure 8.1). The same applies for the right line. Here however the X-coordinate of the upper point smaller. In case that only one vertical line segment is available it is classified by applying the explained concept. If there are two or more vertical line segments the strategy varies. At first the distance between the parallel line segments is calculated. If the distance is smaller than a threshold (135 pixels worked fine) then one of these line segments is neither the left nor the right field line. Therefore just the first line segment has to be examined. If the upper point's X-coordinate is greater than the one of the lower point this line segment is classified as left line and all other line segments receive no classification. Otherwise if the X-coordinate of the upper point is smaller then the last line segment is labeled as right line.

In order to eliminate the classification result that the left and right field line are detected in the same frame an additional test is performed. If the robot looks to the left (yaw angle of the head greater than zero) then the label of the right field line is changed to middle.

Otherwise if the robot looks to the right and the left and right field line are detected simultaneously, the left line is labeled as middle line.

8.2 Evaluation

For evaluation purposes the algorithm has been tested in a simulated scene because the results when the classification is executed on the robot do not vary much. This scene has been specifically developed for testing and evaluating the code dedicated to the coaching robot. Like in a real robot soccer game the coaching robotsits on a Table which is located parallel to the field on level of the middle line. The coaching robotis positioned a bit to the left of the middle line. While sitting the robot performs a sweeping motion. The motion stops for one second if the head reaches the most left and right angle as well as in the middle. Besides the coaching robotthere are four field players in the scene. The evaluation of one entire sweep motion which took 1617 simulation steps which corresponds to 1617 evaluated pictures shows (compare Table 8.1) that the assignment of horizontal lines works very good. Whereas the classification of vertical lines is slightly worse. This is due to the fact that there are three types of vertical lines. An interesting observation is that most of the errors were made during the motion. During the evaluation two types of errors of the classification of the vertical lines could be detected. In the first case the vertical line of the penalty box has been classified as the left or right field line. In the other case the label of an outline and the middle line has been interchanged. A detail that the Table 8.1 does not show is that there have been some frames at which a line has been visible but no label has been assigned.

Table 8.1: Classification results of a sweep motion.

	Correct class. in %	False class. in %	Total number
top	100.0	0	621
bottom	100.0	0	1510
left	96.4	3.6	328
right	92.0	8.0	688
middle	95.3	4.7	1179

Overall the classification algorithm is a stable possibility to determine the position of the ball on the field, in terms of it is located on the left or right side of the field.

9 Calculating the Image to Field Coordinates Projection Using Homographies

Fabian Rensen

In this chapter a solution to the problem of finding the transformation between image coordinates and field coordinates is presented. Since the coaching robot does not sit or stand on the soccer field, the existing camera matrix of the *NDevils* framework can not be used. This is why a different approach is needed for the coaching robot.

9.1 Homographies

Consider a 2D to 2D transformation in homogeneous coordinates

$$\begin{pmatrix} \mathbf{x}' \\ 1 \end{pmatrix} = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{a}^T & 1 \end{bmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \tag{9.1}$$

where A is a 3×3 matrix and $\mathbf{x}$, $\mathbf{x}'$ and $\mathbf{t}$ are 2D vectors in Equation (9.1).

The types of transformations can be divided into classes [27]. The first class is a *euclidean transformation*, for which equation (9.1) is limited by three degrees of freedom

$$\begin{pmatrix} \mathbf{x}' \\ 1 \end{pmatrix} = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \tag{9.2}$$

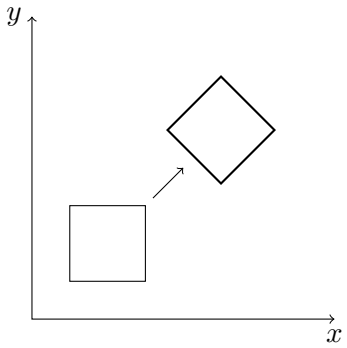
R describes a 2D rotation matrix (one degree of freedom) and $\mathbf{t}$ describes a 2D translational vector. This type of transformation preserves lengths and angles, i. e. the shape will only be translated and rotated. A related type of transformation is the similarity transformation, which additionally allows the shape to be scaled by a factor s

$$\begin{pmatrix} \mathbf{x}' \\ 1 \end{pmatrix} = s \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \tag{9.3}$$

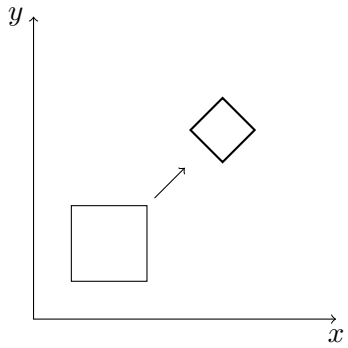
Thus preserving the *ratios* of angles and lengths and resulting in four degrees of freedom. Dropping the constraint of R being an orthogonal matrix, i. e. a rotation matrix, the class of *affine* transformations is found.

$$\begin{pmatrix} \mathbf{x}' \\ 1 \end{pmatrix} = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \tag{9.4}$$

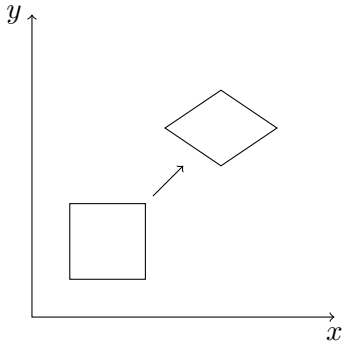
This type of transformations also includes shears and squeezes and hence affine transformations preserve neither angle ratios nor length ratios, but preserve *collinearity*. This is



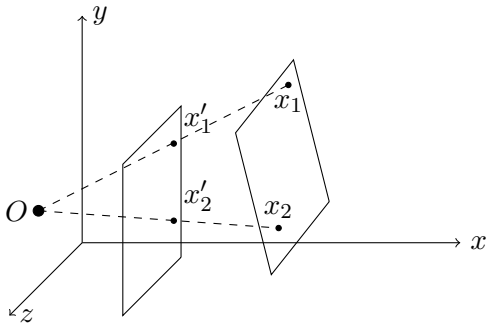
(a) Euclidean transformation, lengths and angles are preserved.



(b) Similarity transformation, length and angle ratios are preserved.



(c) Affine transformation, collinearity is preserved.



(d) Projective transformation.

Figure 9.1: Different classes of transformations.

why ratios of distances between points on the same line are preserved. Because there are no constraints of the matrix A, affine transformations have six degrees of freedom. The last class of transformations is called *projective* transformation or *homography*, which can be thought of as a camera observing a plane in 3D space. The transformation from the image plane to the object plane is not affine, since the object plane may be tilted in 3D space. Therefore eight parameters are necessary to describe the transformation

$$\begin{pmatrix} \mathbf{x}' \\ x'_3 \end{pmatrix} = H \begin{pmatrix} \mathbf{x} \\ x_3 \end{pmatrix} \tag{9.5}$$

Since the matrix H may be scaled by an arbitrary factor s one element, i.e. h_{33}, may be set to 1 and hence there are only eight instead of nine degrees of freedom. Figure 9.1 is illustrating the different types of transformations.

9.2 Calculating a Homography via Point Correspondence

To calculate a *homography* a minimum of point correspondences are sufficient. Considering the homography H [27]

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = H \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{9.6}$$

and the pinhole camera model with focal length $f = 1$

$$x = f\frac{x_1}{x_3} = \frac{x_1}{x_3} \tag{9.7}$$

$$y = f\frac{x_2}{x_3} = \frac{x_2}{x_3} \tag{9.8}$$

each point to point correspondence will result in two equations [27]:

$$x' = \frac{x'_1}{x'_3} = \frac{h_{11}x + h_{12}y + h_{33}}{h_{31}x + h_{32}y + h_{33}} \tag{9.9}$$

$$y' = \frac{x'_2}{x'_3} = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \tag{9.10}$$

These equations can be used to set up a linear system of equations for solving the parameters h [27]:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 & -y'_1 \\ & & & & & \vdots & & & \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_nx_n & -x'_ny_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_nx_n & -y'_ny_n & -y'_n \end{bmatrix} \mathbf{h} = \mathbf{0} \tag{9.11}$$

$\mathbf{h}$ are the parameters h_{11} to h_{33} arranged in a column vector and n is the total number of point correspondences. It follows that the (one dimensional, for four point correspondences) null space of this matrix needs to be calculated. A common way to solve the null space of a matrix is via *singular value decomposition*, which is included in most mathematical libraries, i. e. *Eigen*. Let $USV^T = \text{svd}(B)$ be a singular value composition of the matrix B. Then the last m columns of the matrix V corresponding to the m singular values equal to zero will span the null space of the matrix B [20]. Another advantage of the *singular value decomposition* is that it will also solve the matrix in a least squares meaning [20], if more than four point correspondences are given, i. e. the matrix in Equation (9.11) consists of more equations than needed.

9.3 Results in the NDevils Framework

In a test environment the results of the above method were accurate enough to provide the desired image to field coordinates functions. The problem is identifying four point correspondences. If the coach turns its head to left or to the right to look at one of the

soccer field halves four point correspondences may be found via detecting the middle line, the two sidelines and the goal line. Then the four intersections of these lines represent the two "T"-intersections and two field corners. It is however very unlikely that all of the four lines are identified in every frame due to a highly dynamic environment and many obstacles. Furthermore if the coach robot looks towards the middle of the field the two goal lines may not be detected and hence the four corners of the field can not be matched. Including the intersections of the center circle with the middle line is not possible either, because of the four points three are not allowed to be collinear, otherwise the matrix in Equation (9.11) would loose rank. That is why we decided to measure the coaches position and use the framework's existing matrices and functions to compute the transformation. This solution is not ideal, but considering the problems mentioned above it is a working alternative, at least until we find a way to detect four point correspondences more frequently. In that approach we measure the coach's hip position and use the existing hip-to-camera transformation matrix to provide a transformation from image to field coordinates.

10 Ball Detection with Clustering and Hough Transform

Marcel Eilers, Torsten Böttinger

The rule updates for the *RoboCup2016* in Leipzig include a new ball. The new ball is a soft foam ball with black and white soccer ball print [34]. The ball has a diameter of 10 cm. For a ball detection it is not possible to search for an orange color as used for the orange ball.

Our approach uses the shape of a ball. With a *Hough Transform* for circles we can find possible balls in an image. One disadvantage of the *Hough Transform* is the long runtime to compute it. Therefore we use a clustering mechanism to find interesting spots in the image and utilize the *Hough Transform* only on a relatively small patch of the image.

10.1 Clustering Algorithm to Find Potential Balls

The first step of our approch is to cluster the pixels that are in the convex hull but do not have the previously calculated field color. This initial situation is visualized in the upper right part of Figure 10.1 where these pixels are colored red. Detailed information about the computation of the field color and the convex hull can be found in Chapter 5.

To cluster the pixel we use the *DBSCAN* algorithm that is also utilized to find the robots. The algorithm is explained in detail in the Chapter 6. The result is a list of clusters with the corresponding pixels as you can see the lower left part of Figure 10.1. Because the coaching robot always has roughly the same distance to the field, the potential size of the ball can be severely limited. Therefore clusters with very many and very few pixel can be filtered out, which finally leads to a result that can be seen in the lower right part of Figure 10.1.

10.2 Eccentricity of the Clusters

After calculating the potential balls with the clustering algorithm, the next step is to find out which cluster is the one that most likely contains the ball. The basic criterion of this decision is the eccentricity of the clusters. In this case the eccentricity is a measure for the roundness of the clusters. It is defined as:

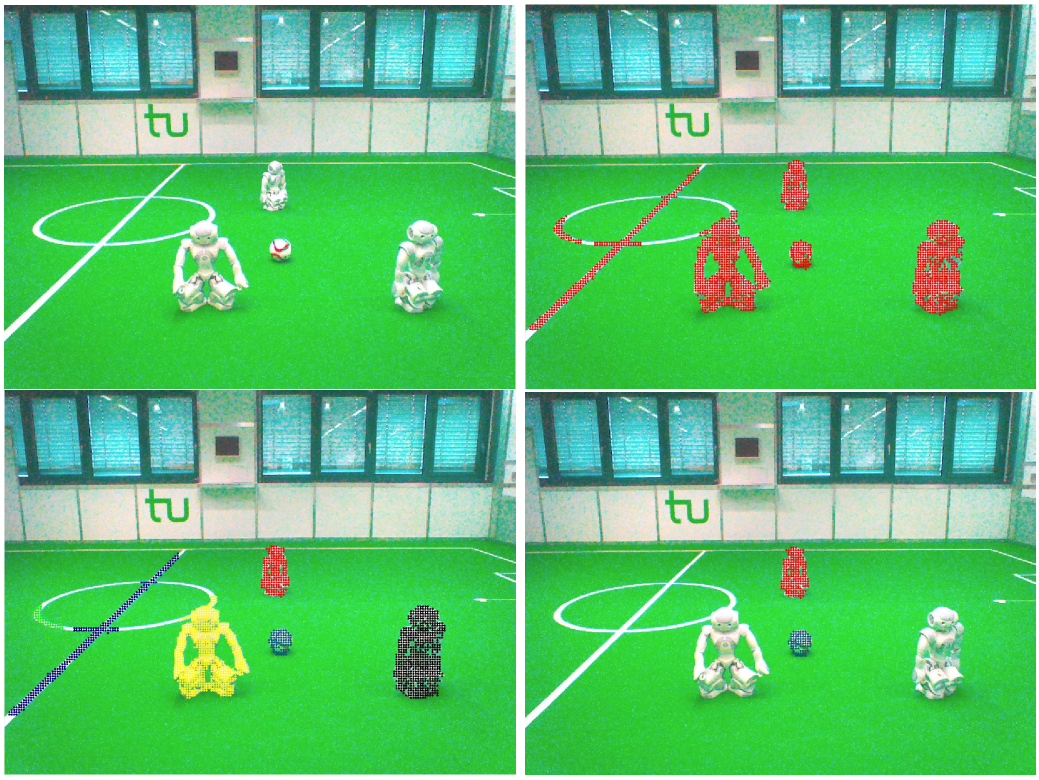


Figure 10.1: Use of the *DBSCAN* algorithm to find the ball.

$$e = \sqrt{1 - \frac{\lambda_2}{\lambda_1}} \tag{10.1}$$

λ_1 and λ_2 are the eigenvalues of the cluster, which are calculated as follows:

$$\lambda_i = \frac{\mu'_{20} + \mu'_{02}}{2} \pm \frac{\sqrt{4\mu'^2_{11} + (\mu'_{20} - \mu'_{02})^2}}{2} \tag{10.2}$$

To compute them, the normalized central image moments μ'_{pq} need to be known:

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y) \tag{10.3}$$

$$\mu'_{pq} = \mu_{pq} / \mu_{00} \tag{10.4}$$

x and y are the coordinates of the points of one cluster and $\bar{x}$ and $\bar{y}$ are the parameters of the centroid. A small difference between the two eigenvalues leads to a small excentricity. Cluster with a small eccentricity are rounder than cluster with a higher one [13].

That is why we search for the cluster with the smallest eccentricity. If this eccentricity is smaller than a certain value, the provisional ball position is defined as the centroid of the corresponding cluster. Subsequently the result is verified with the *Hough Transform* for circles.

10.3 Hough Transform for Circles

Previously we have seen the *Hough Transform* for lines in Chapter 4. We adopt this approach for circles. For each edge pixel on the lowcutted sobel image we vote up each pixel which could be a possible center of a circle with specified radius.

In contrast to the solution above we do not use a probabilistic variant of the *Hough Transform*. Also the accumulator needs to contain more information like x, y coordinate and the radius of the possible centers. This circumstance leads to a three-dimensional accumalator.

10.3.1 Quality check

To determine ball or no-ball we introduce a quality check for the result:

$$\text{quality} = \frac{\text{count}_{\text{hits}}}{\text{possibleHits}} \tag{10.5}$$

and the condition to decide ball or no-ball is

$$\text{quality} < \text{quality}_{\text{threshhold}} \tag{10.6}$$

In the concrete implementation we choose $\text{quality}_{\text{threshhold}} = 0.4$. This is a result of testing on the robots.

In many cases there were results in the center circle of the field and these results were obviously on the field. So we introduced a check for field color to this algorithm. We check the amount of fieldcolor inside this circle and and if more than half of the pixels are fieldcolor we discard this result.

10.3.2 Radius Ranges

The radiuses we chose to check (from 2 to 15 pixels) are examined from real situations of the robot's camera. The smallest ball on an image we measured had an diameter of around 6 pixels. Additionally the image is halved so that in the upper area we check from 2 to 11 pixels and in the lower area from 7 to 15 pixels. These measures cause a better performance and better results because when the ball is located directly in front of the coaching robot small circles cannot be a ball.

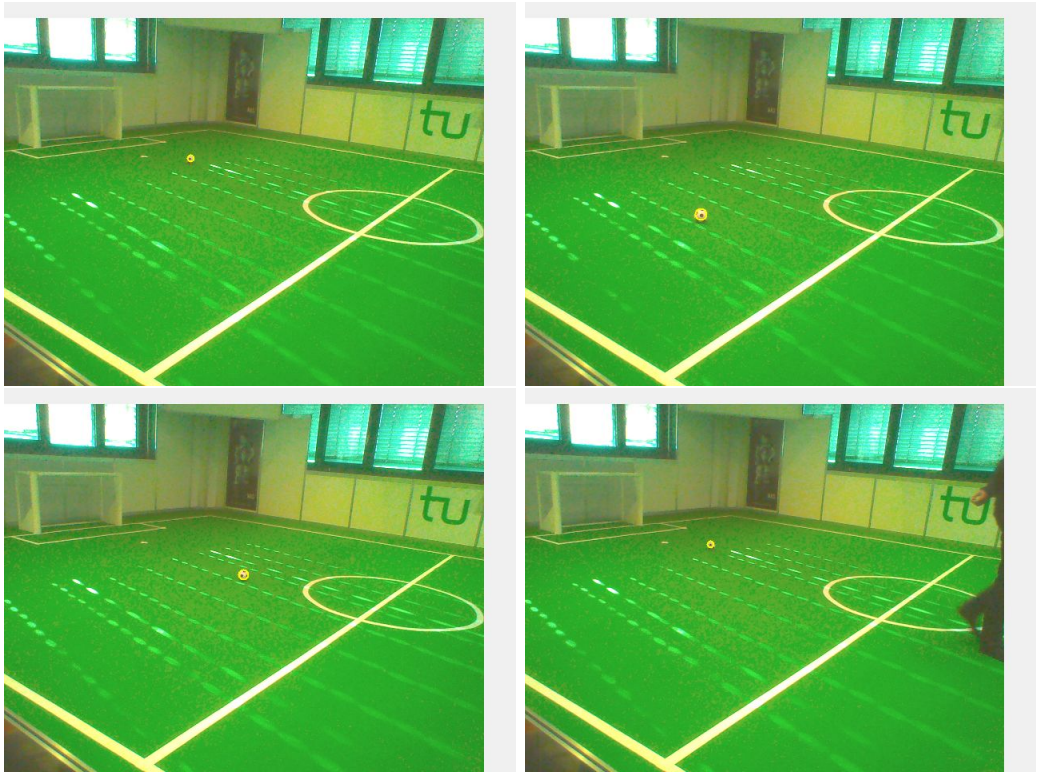


Figure 10.2: Situations with just the ball on the field (the yellow circle shows where the algorithm assumes the ball).

10.4 Evaluation

This approach can recognize the ball in all positions on the field even if it is just a few pixels small like in Figure 10.2.

The search for ball with the clustering algorithm and the analysis of the eccentricities is successful if the ball is surrounded with pixel that have fieldcolor. This situation is shown in the upper left part of Figure 10.3. Problems emerge when the ball crosses one of the field lines or rolls very close to a robot, which is shown in the upper right part. The reason for these problems is that the cluster of the ball and the cluster of a line or a robot form a single cluster. The red cluster in the lower left part is obviously not very round so that the ball cannot be found correctly with this approach. This difficulty should be solved with the hough transfom, as shown in the lower right part.

The runtime of the *Hough Transform* is primarly dependend of the amount of found edge pixels in the image and the count of the considered radiuses. The amount of the radiuses is quite static like explained above. The amount of edge pixels diversify from situation to situation.

The *Hough Transform* for the whole field the robot sees has a runtime of around 600 to 1000 ms. The assignment of the clustering algorithm and the consequential smaller amount of considered edge pixels leads to a runtime of around 100 ms. That shows that

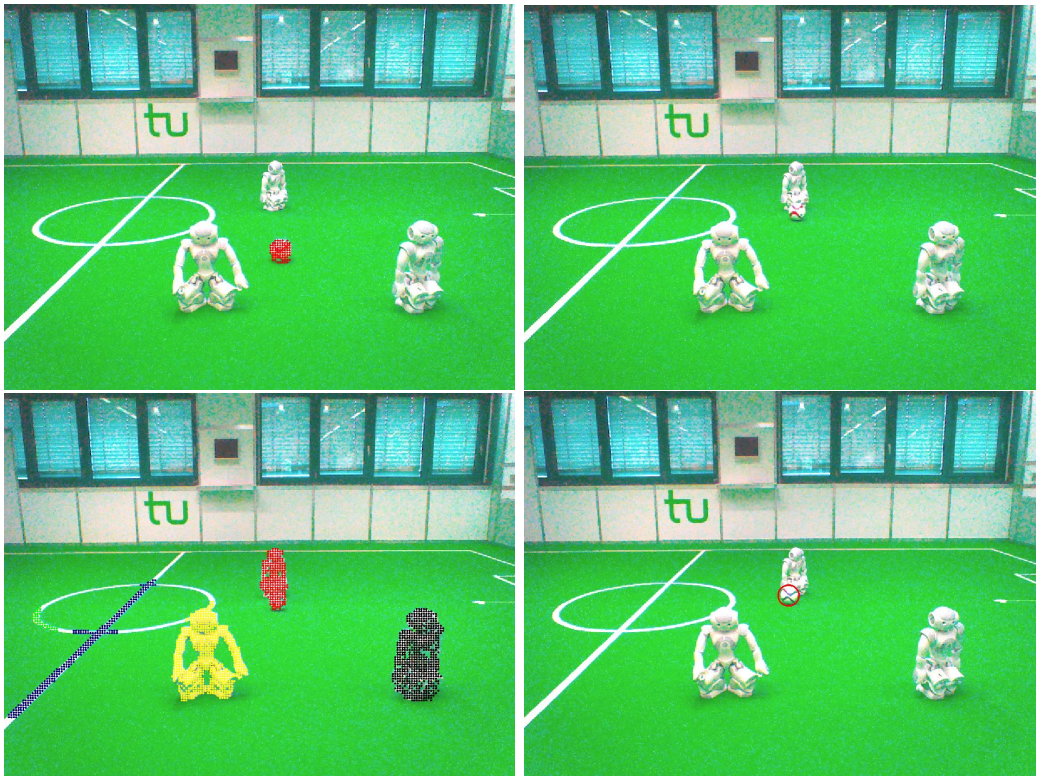


Figure 10.3: Results of the ball detection with a ball that is slightly different to the ball description of the official rules.

this recognition is working for the coaching robot which has a lot more time to analyze the situations than the $NAOs$ on the field.

11 Estimate Stability of Robots Using an Artificial Neural Network

Florian Ziegler

In this chapter we discuss a way for finding a criterion to decide whether a robot is currently in a stable position or not. We use an *Artificial Neural Network* (*ANN*) to solve this task.[1] The idea is to find a criterion that will allow us to predict, if a robot will fall down in the future so that we can introduce appropriate countermeasures like making him stand still for a moment. The current solution reacts only, if the robots position exceeds a specified angle which works not well at all. We hope to make the robot fall down less while having a minimum of false predictions at the same time.

11.1 Introduction to Artificial Neural Networks

An *ANN* is a technique from the area of machine learning where we try to learn some behaviour from a given set of data. That means, given an instantiation of the so called *input variables* (or *features*) $X_1, ..., X_N$ resp. the *input vector* $X = (X_1, ..., X_N)$, we try to make predictions about the *output variables* $Y_1, ..., Y_M$ resp. about the *output vector* $Y = (Y_1, ..., Y_M)$. If Y is a scalar, we will call it an *output variable* as well. $x_1, ..., x_N$ and $y_1, ..., y_M$ are the instances of those variables and the tuple $(x_1, ..., x_N, y_1, ..., y_M)$ is called an *example*. The real world data is described by the function $Y = y(X_1, ..., X_N)$ that could allow us to predict Y without any errors. In praxis, we will always have some error $\varepsilon \neq 0$, because we derive our predictions from a subset of the real world data. We try to approximate Y by the function

$$\hat{Y} = \hat{y}(X_1, ..., X_N) = y(X_1, ..., X_N) + \varepsilon, \tag{11.1}$$

where $|\varepsilon|$ should be minimized such that $\hat{Y} \approx Y$. *Supervised learning* methods get values for the *input variables* as well as the expected values for the *output variables* to learn Y. This set of input and associated *output variables* is called the *training set*. It is a set of known *examples*.

Because of the fact that we want to learn whether a robot is stable or not, we have a *two-class classification problem*. In this case, $\hat{Y}$ can be seen as the certainty with which our learning method has classified an *example*. We could define $f(\hat{Y})$ as a *classifier* with

$$f(\hat{Y}) := \begin{cases} 1 \ (\hat{=} \text{ robot is stable}) & : \hat{Y} \geq \theta, \\ 0 \ (\hat{=} \text{ robot is unstable}) & : \hat{Y} < \theta, \end{cases} \tag{11.2}$$

[1] For further reading on *ANNs* see [1] or [24] for example.

where θ is a constant.

Our options to minimize $|\epsilon|$ are to record as much *example* data as possible and to choose the right machine learning method with the right parameters. In our case, we used an *ANN* for the learning task. The *ANN* consists of one or more of the so called *perceptrons* [24]. A *perceptron* is used to linearly separate data by a hyperplane to perform a classification. To classify one instance of *input variables* $x = (x_1, ..., x_N)$, the *perceptron* calculates a weighted sum of its components first, added by an additional weight w_0 which results in a more flexible model:

$$\hat{y}_{\text{ANN}}(x) := \sum_{i=1}^{N} w_i x_i + w_0. \tag{11.3}$$

After that, the resulting value will be normalized e.g. by a *sigmoid function* $\text{sig} : \mathbb{R} \to (0, 1)$ and we get $\hat{y}(x) = \text{sig}(\hat{y}_{\text{ANN}}(x))$. Learning data with a *perceptron* means to optimize its weights such that its classification performance will be maximized.

One *perceptron* can classify linear separable data. If $y(X)$ is more complex, we need to use a configuration of multiple *perceptrons*. This leads to the *multilayer perceptron* (*MLP*) which is the type of *ANN* that we use in our solution. An *MLP* consists of one or more layers each consisting of one or more *units*. In a *three-layer MLP* we have the *input layer*, the *hidden layer* and the *output layer*. The units of the *input layer* (the *input units*) are represented by the *input variables* and an additional value that is always 1 (which is needed for adding up w_0). The *hidden layer* consists of l_1 *perceptrons* (in this context called the *hidden units*) each connected with each input unit. This means that each hidden unit has exactly $l_1 = N + 1$ weights. Finally, the output units are l_2 *perceptrons* each connected with each hidden unit. l_2 is the number of classes we have in our classification problem. In our case $l_2 = 2$. An example is shown in Figure 11.1. A *three-layer MLP* is able to approximate all functions which is why it fits our needs.

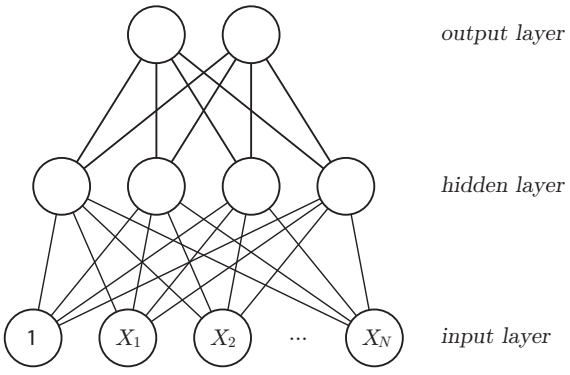


Figure 11.1: An exemplary *three-layer MLP* with input $X = (X_1, ..., X_N)$. If we take one node of the *hidden layer* for example, it has $N + 1$ edges, each leading to one input node. Those edges represent the weights $w_0, ..., w_N$ while the edge for w_0 is connected with the node labeled with 1. This corresponds to equation (11.3).

11.2 Feature Selection

In this section we describe which types of data we will choose for the *input vector* $X = (X_1, ..., X_n)$. We tried out two different solutions for this task. The first approach was the usage of sensor data of the robot without recognizing historical information. Secondly, the historical data was added, so that we got a sequence of sensor data values of the *input vector*.

11.2.1 Non-Sequenced Data

As an input we use the sensor data of the robot which means that our *input variables* are the gyroscope for the x- and y-axis, the acceleration values in x, y and z direction and the eight force resistive sensors on the feet. We don't use the gyroscope sensor for the z-axis because the rotation around the z-axis is irrelevant for determining if the walk of the robot is stable. The classification function $f(Y)$ of the *output variable* Y is 1, if the robot is in a stable position, otherwise its value is 0. We get the information whether the robot is stable or not from the class `FallDownState` in the NDevils framework. That class gives us the information, if the robot is upright, falling, lying on its back or lying on its front. We will refer the related variable that holds this information as the fall down state. Y is 1, if the robot is in the upright position, otherwise Y is 0. We can log this data which results in a CSV file in which the *examples* are temporally sorted.

According to this definition of Y we get

$$f(Y) = \begin{cases} 1 \ (\hat{=} \text{ robot is stable}) & : \text{ robot is upright,} \\ 0 \ (\hat{=} \text{ robot is unstable}) & : \text{ else} \end{cases} \tag{11.4}$$

and we can set $f(\hat{Y}) = f(Y)$ without any problems. The *ANN* won't give us any new information. So learning from this data is not enough. We want to predict, if a robot will fall down even before the `FallDownState` gives us the information that it is falling. This is why we have to set the value of Y for the ith *example* to false, if the $(i + \alpha)$th fall down state is not "upright" even if the ith fall down state is "upright". α is a positive integer. We also have to do that for the $(i + 1)$th, $(i + 2)$th, ..., $(i + \alpha - 1)$th *example*. For a better understanding, see Table 11.1.

The value of α should be large enough for the robot to react on an upcoming fall timely. On the other hand a high value of α correlates with a worse classification performance. In our first experiments we have chosen $\alpha = 10$ which shows that it is too small for making predictions early enough. We first wanted to examine, if it is possible to train a *classifier* that has an acceptable classification performance with such a low value for α. After that we have increased α.

11.2.2 Sequenced Data

We can either use the *features* as described above or we can see the *input vector* X as a time series or sequence of the recorded sensor data. Let $x_i(t_k)$ be the value of the sensor data

Table 11.1: An example of how $f(\hat{Y})$ could be derived from the fall down state. Here $\alpha = 2$. Note that in the real data there will be longer sequences of *examples* with a fall down state that has the value "falling" or "lying on back". For reasons of space we have simplified the data.

is stable ($\hat{=} f(\hat{Y})$)	fall down state
true	upright
false	upright
false	upright
false	falling
false	falling
false	lying on back
false	lying on back
false	lying on back
true	upright

variable X_i that was recorded at time t_k, where $i \in \{1, ..., N\}$ and $k \in \mathbb{N}$. t_k is the number of frames passed before $x_i(t_k)$ was recorded. With this definition we can set one instantiation of the *input vector* as $x_{\text{seq}} := (x_i(t_k), ..., x_i(t_{k+N}))$ and $X_{\text{seq}} := (X_i(t_k), ..., X_i(t_{k+N}))$ as the *input vector* for sequenced data. If the temporal distance $\Delta t = t_{k+1} - t_k = 1$, we would use an *input variable* for each frame between t_k and t_{k+N}. While not changing the number of *input variables*, we sample a larger interval by increasing Δt, but also leaving out information, because we are throwing away the values in between the samples.

This way, we can take temporal data into account. Because of the fact that we make use of an *MLP*, we can't use cycles between the units. The downside of this method is that we have to focus on one type of sensor data for one *MLP*. On the other hand it is no problem to combine multiple *MLPs* each using different sensor data variables. Furthermore we will show in section 11.5 that we only need the gyroscope value for the y-axis to obtain satisfiable results.

Let Y_{seq} be the *output variable* for sequenced data and y_{seq} its instantiation. How should we choose Y_{seq} in the *training set* for a given *example*, resp. the classification function $f(Y_{\text{seq}})$? In the non-sequenced data case, we calculated each y of the training data set by $f(y) = f(\text{sig}(y_{\text{ANN}}(x)))$ according to equation (11.3). Now we have to calculate $f(y_{\text{seq}})$ based on instantiations of multiple *output variables*, since we have to take multiple *examples* into account. Let $y(x_1, ..., x_n)(t_k), ..., y(x_1, ..., x_n)(t_{k+N})$ or shortly $y(t_k), ..., y(t_{k+N})$ be the instantiations of *output variables* of the *example set* for the non-sequenced data case at the times t_k, ..., t_{k+N}. For the $f(y_{\text{seq}})$ that we want to calculate, we could say that if at least one of the classifications $f(y(t_k)), ..., f(y(t_{k+N}))$ would result in the value 1, we set $f(y_{\text{seq}}) = 1$ and otherwise $f(y_{\text{seq}}) = 0$. Thus we define

$$f(y_{\text{seq}}) := f(y_{\text{seq}}(y(t_k), ..., y(t_{k+N}))) := \begin{cases} 1 & : f(y(t_k)) + ... + f(y(t_{k+N})) \geq 1, \\ 0 & : \text{else.} \end{cases} \tag{11.5}$$

Since the computation time of the neural network will rise with the number of *input variables*, we have to reduce the dimension of the data, if we want to consider very long time series or if our CPU is not strong enough. We have chosen $N = 100$ and divided

the time series in ten equally sized interconnected regions. The arithmetic means of the elements of each region are our *input variables* which results in a new filtered time series of ten elements such that we have an *input vector* of length ten. In Figure 11.2 we can see that it makes sense to choose a time series of length 100, since the plot of the gyroscope variable has a very characteristic form in the interval of length 100 just before the robot starts to fall down.

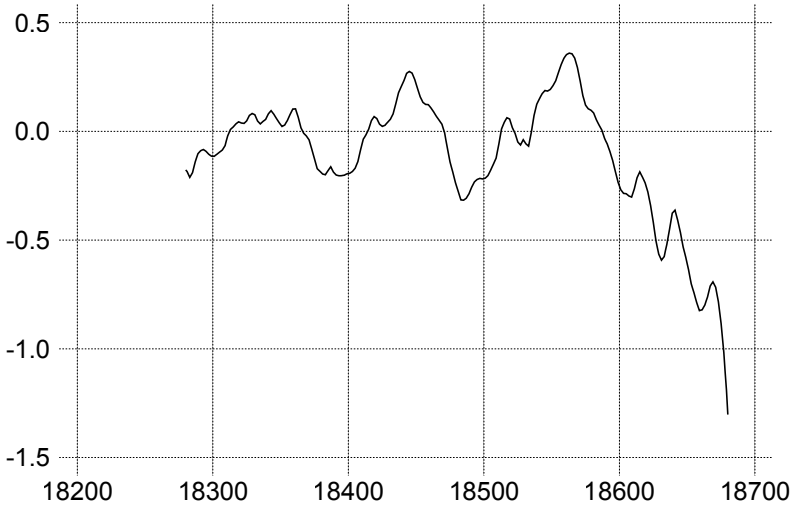


Figure 11.2: An example of how the gyroscope value of the y-axes behaves before the robot falls down. The x-axes shows the frames and the y-axis is the gyroscope value. The local maxima (minima) become higher (lower) right before the fall which is represented by the rapidly declining curve. For stability detection, the interval between 18500 and 18600 (with the highest local maximum) could be sufficient for predicting a fall.

11.3 Choosing the Classifier

For training the *classifier* we used the software *RapidMiner (Studio)*[2]. In RapidMiner we can use different machine learning methods like *MLPs*, *SVMs*, etc., export the trained model and evaluate the results. While training an *MLP*, we can use the following parameters for the learning algorithm that learns the weights of the *perceptrons*:

Training cycles Specifies how many iteration steps are used to train the *MLP*. In each iteration step the weights will be adjusted.

Learning rate Sets the magnitude with which the weights will be changed.

Momentum Adds a fraction of the previous weight update to the current one. This helps to avoid that the *MLP* sticks at a local maximum.

Decay A boolean value that decreases the learning rate over time, iff set to true.

[2] *RapidMiner Studio* can be downloaded on the website https://rapidminer.com/. We used the version 6.4.

We trained our networks with the values 1500, 0.2, 0.2 and "false" for the parameters training cycles, learning rate, momentum and decay (in this order). The results of the evaluation of the trained *MLP* that uses non-sequenced data can be found in Table 11.2. In a two-class classification problem the *precision* is the number of correct classifications (as true or false) in relation to all *examples*, while the *recall* is the number of correct classifications (as true or false) divided by those *examples* that should be classified as true resp. false.

Table 11.2: Confusion matrices for three evaluations of the learning processes. The first component of the first line of the matrix represents the number of *examples* that were classified as false (unstable) correctly. The second component of the second row represents those *examples* that were classified as true (stable) correctly. The other two components show the number of incorrect classifications.

(a) Used all sensor data for training and $\alpha = 10$.

	actual false	actual true	class *precision*
pred. false	36761	911	97.58%
pred. true	1435	185795	99.23%
class *recall*	96.24%	99.51%	

(b) Used all sensor data except the foot sensors for training and $\alpha = 10$.

	actual false	actual true	class *precision*
pred. false	36186	1063	97.15%
pred. true	2010	185634	98.93%
class *recall*	94.74%	99.43%	

(c) Used all sensor data for training and $\alpha = 50$.

	actual false	actual true	class *precision*
pred. false	33805	2330	93.55%
pred. true	4391	184376	97.67%
class *recall*	88.50%	98.75%	

The *precision* and *recall* of the network that uses non-sequenced data allows the conclusion that we will have a small number of false positives and false negatives. In the case of using sequenced data for training the network, we got the results shown in Table 11.4.

Table 11.4: Confusion matrix for the evaluation of the learning process using the mean of ten equally sized and connected regions taken from a time series consisting of 100 gyroscope values (for the y-axes), and $\alpha = 100$.

	actual false	actual true	class *precision*
pred. false	21972	3073	87.73%
pred. true	20959	185244	89.84%
class *recall*	51.18%	98.37%	

We can see, that the *recall* is much lower in comparison to the *recall* shown in Table 11.2. The reason for such a bad *recall* could be (apart from the increased α) that—according to equation (11.5)—we have set $f(y_{\text{seq}}) = 1$, iff at least one of the classifications $f(y(t_k)), ..., f(y(t_{k+N}))$ would result in the value 1 (as explained in section 11.2). The *recall* will decrease for a greater value of N, such that a robot will often be classified as stable, even if he is in fact not stable. One could experiment with other methods of declaring a time series $(x_i(t_k), ..., x_i(t_{k+N}))$ as stable, i.e. setting $f(y_{\text{seq}}) = 1$, iff at least half of the $f(y(t_k)), ..., f(y(t_{k+N}))$ have the value 1. But our experiments have shown that we even get satisfiable results and the robot will detect instabilities. The bad *recall* value will probably only cause an insignificant lost of reaction time.

11.4 Implementation in the NDevils Framework

At first, we have to get the model that was trained with RapidMiner. RapidMiner can export a trained model as an XML file. We implemented classes in the NDevils framework to read in this file and to instantiate a neural net model written in C++ based on the read information. This procedure works only for a *three-layer MLP* with two output units. The XML file has to be put in the Configurations/ClassifierModels folder.

The model of the neural network is used in the module `NeuralNetStabilityDetector`. It takes the sensor data by requiring the representation class `SensorData` and provides and modifies `StabilityInfo`. The `StabilityInfo` class has a boolean variable `isStable` that is true, iff the robot is stable. This variable will be set by the `NeuralNetStabilityDetector` by taking the current sensor data and feeding it to the model of the neural network. As a result we get a value between 0 and 1 which is our $\hat{Y}$. Next we set the variable `isStable` to true, iff the resulting value of the neural network is greater or equal than a given threshold θ. This variable corresponds to $f(\hat{Y})$, where $f(\hat{Y}) = 1 \Leftrightarrow$ `isStable` = true and $f(\hat{Y}) = 0 \Leftrightarrow$ `isStable` = false.

It is also possible to use multiple neural networks at once. The results of each network will be calculated individually. The decision if the actual walk of the robot is stable or not will be made by a vote. Each result (1 or 0) will be summed up and must be greater than or equal to a specified constant to get $f(\hat{Y}) = 1$. Otherwise $f(\hat{Y}) = 0$. Furthermore the path of each network has to be set.

To consider previous results without using a time series we can set the *memory* $\beta \in [0, 1]$ of the network. Let $\hat{y}(t_k)$ be the current output without considering previous results and $\hat{y}(t_{k-1})$ the output for the *example* before. Then we calculate

$$\hat{y}^* = \frac{\hat{y}(t_k) + \beta\hat{y}(t_{k-1})}{\beta + 1} \tag{11.6}$$

as the final output. Note that $\hat{y}^* = \hat{y}(t_k)$, if $\beta = 0$ which means that the *MLP* has no *memory* and is not considering previous values. If $\beta = 1$, $\hat{y}^*$ will be the arithmetic mean of $\hat{y}(t_k)$ and $\hat{y}(t_{k-1})$, so that the *memory* is maximal.

Finally the decision whether the robot should stop its movement because of a forthcoming fall is made in the class `RequestTranslator` by checking the `isStable` variable. Because

of noisy data we need to check the `isStable` variable k times. If it is false k times in a row, we can make the robot stop its movements. This method has a similar effect on the output compared to the approach described in equation (11.6). But it could be more useful for getting rid of outliers, while the aim of the method described first is to take past behavior into account.

11.5 Experiments

The purpose of the experiments is to show that the use of our solution for predicting falls of robots brings us time advantages in comparison to the use of the previous solution. The duration of each experiment was set to five minutes in which the robot was walking straight forward on the field. If he was not able to walk any further (which means that he has reached a wall), we had to turn him around such that he could continue walking. Whenever he fell down, we picked him up immediately. Then we counted how often he fell down and how often he stopped moving in response to the neural network. The problem with this experimental setup is that you can't exactly compare the time that is needed to pick up a robot in comparison to the duration of stopping motion. Anyhow we get an idea of how good our solution works.

11.5.1 Recording the Data

The neural networks that we have evaluated are using various types of data which means that we used two methods for recording it. In the following, we describe those methods:

1. Let the robot walk around and push him occasionally such that he will become unstable and fall down afterwards. The data includes 50 falls on the back and 50 falls on the front. We will refer to this method as *data record method 1*.

2. Let the robot walk around with an unstable walk that makes him fall down often without help. Take only those *examples* of the data that represent unsteady walks. Then record more data without making the walk artificially unstable. Combine this data set with the previously extracted *examples*. This way the robot will learn a stable walk with unchanged walking parameters and will fall down often enough in the data recording phase. We will refer to this method as *data record method 2*.

The impacts of those methods to the classification will be discussed in the following sections.

11.5.2 MLPs Without Time Series

At first we will discuss the results for the neural networks that are using the sensor data as *features* without using a temporal history. The networks were trained with *data record method 1* which turns out to lead to a lot of false positives when running on a robot. While the robot will recognize a manual triggered instability by pushing, he mostly won't

be able to recognize that he is in an unstable position in other cases and will fall down as a consequence. It is likely that the data that was recorded by pushing the robot away differs too much from the actual behaviour of the robot when he is in an unstable position. While those *MLPs* are not suitable for detecting falls that were triggered by unstable walking, they could be used to detect instabilities resulting by uneven ground like artificial lawn for example.

It would also make sense to use data method 2 together with an *MLP* that doesn't use time series which we have skipped for now, because the use of sequenced data was more promising for our task.

11.5.3 MLPs With Time Series

For those networks that are using time series for the training, we used *data record method 2*. We used different realizations of θ (the threshold of the network) and β (the *memory* of the network). The experiments were carried out for an unstable walking robot for which we changed the walking parameters and for the default walk, meaning that the walking parameters were not changed. The latter we will refer to as normal walk. Each experiment was carried out up to five times. The results are listed in Table 11.5.

Table 11.5: Three experimental setups in which the number of falls and the number of predictions of falls where summed up. Additionally, we calculated the arithmetic mean.

(a) Threshold of 0.0075 and *memory* of 0.667.

		1	2	3	4	5	total	mean
unstable	# falls	3	4	2	4	3	16	3.200
	# predictions	35	33	36	32	35	171	34.200
normal	# falls	0	-	-	-	-	0	0.000
	# predictions	13	-	-	-	-	13	13.000

(b) Threshold of 0.005 and *memory* of 0.9.

		1	2	3	4	5	total	mean
unstable	# falls	5	10	4	-	-	19	6.333
	# predictions	30	25	28	-	-	83	27.667
normal	# falls	5	4	1	3	-	13	3.250
	# predictions	10	7	5	7	-	29	7.250

(c) Neural net deactivated.

		1	2	3	4	5	total	mean
unstable	# falls	30	30	26	27	23	136	27.200
normal	# falls	1	0	3	3	-	7	1.750

The experiments in Table 11.6a and 11.6b differ in the values of the threshold θ and the *memory* β. The configuration $\theta = 0.0075$ and $\beta = 0.667$ leads to an *MLP* that is more sensible in detecting instabilities than for $\theta = 0.005$ and $\beta = 0.9$. In the first experiment, shown in Table 11.6a, most of the falls were detected. However, the number of predictions

is significantly larger compared to the falls shown in Table 11.6c (where the neural network is not active). This allows the conclusion of a great number of false negatives (meaning that the *MLP* predicted a fall but the robot would not have fallen down). For this reason, we carried out only one experiment for the normal walk, since we first wanted to invest our time to evaluate the other *MLPs*. It is noteworthy that also in the experiments shown in Table 11.6b a large number of falls and predictions occurred. Following this, we have a large variance in the number of falls. Furthermore, the robot can even be unstable without falling down afterwards. Those events would probably be detected by the neural network but not shown in the statistics in Table 11.6c where the network is deactivated. For a better evaluation it would also be helpful to run more experiments in the future. From the results shown in Table 11.6b we could conclude that we have less false positives at the cost of more undetected falls.

The experiments show that with $\theta = 0.005$ and $\beta = 0.9$ most of the falls could be predicted while having (probably) a small number of false negatives in the case of an unstable walk. For the normal walk the variance in the number of falls is too high to reach a firm conclusion. Note that the threshold value for the *MLP* is really low which leads to poorly separable classes. One reason for this could be again that we have set $y = 1$, iff for any x_i, $i \in \{1, ..., N\}$, the corresponding y^j will be 1 (see section 11.2).

11.6 Conclusion

Recapitulating, we have trained and implemented a neural network to learn a criterion for the instability of walks of robots to predict falls. For this task we used a *three-layer MLP* with different types of *input variables*. We got satisfying results by using a time series of gyroscope values as the input for the *MLP*.

For a more detailed evaluation we have to carry out more experiments, especially on multiple robots, since we used only one robot up to now. An evaluation of a running game with our solution would be helpful, too. Some methods for improving our neural network could be:

- Recording more data from multiple robots.
- Changing parameters of the neural network during training of the data.
- Changing parameters of our implementation of the neural network, i.e. the threshold θ or the *memory* β.
- Experimenting with other criterions for defining y in $(x_1, ..., x_n, y)$ where $(x_1, ..., x_n)$ represents a time series.
- Using multiple neural networks.

For the last point, we have done some rudimentary tests which have shown us, that it could be rewarding to continue experimenting in this direction. It could also be interesting to implement an *ANN* that runs on a robot and learns while he is playing to constantly update the learned model.

12 Modeling of Ball Dynamics and Localization

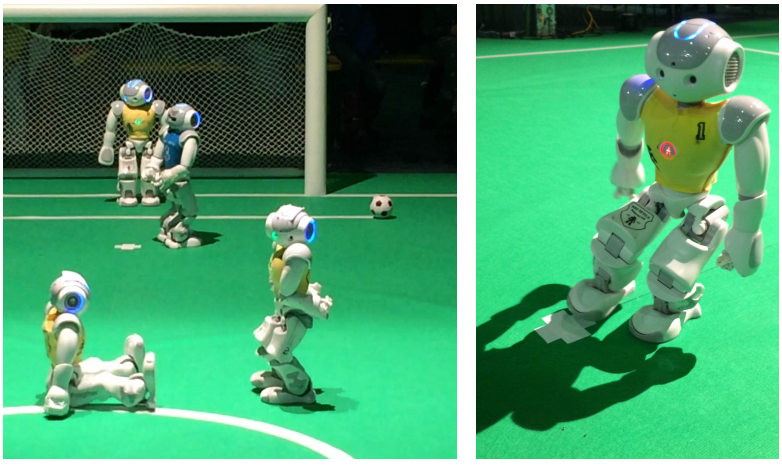
Heiner Walter

In this chapter we describe our modifications to the world state modeling. We had to improved the filtering of the ball position in reaction to the more difficult detection of a largely white ball. This is explained in Chapter 12.1. We also implemented another approach for self localization which is described in Chapter 12.2.

12.1 Modeling of Ball Dynamics Based on Uncertain Measurements

Starting from the season 2016, the red ball used in the SPL previously is replaced by a more natural looking soccerball. As the new ball is white as most other objects on the soccer field, the ball detection gets more difficult. In consequence we receive more false positives from the vision module. Especially the penalty cross is often confused with the ball (see Figure 12.1b).

For this reason, the existing ball model consisting of a single *Kalman Filter* is no longer suitable for predicting the ball dynamics. In order to make the prediction possible again,



(a) The game gets stuck as no robot recognizes the ball.

(b) The robot sees a ball in the penalty cross.

Figure 12.1: The white ball makes perception more difficult.

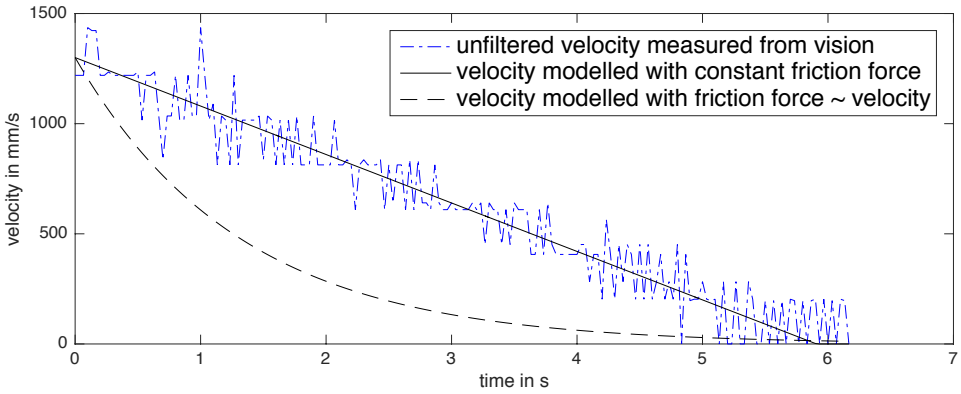


Figure 12.2: Measured ball velocity curve and estimation using two different friction models. The solid line visualizes a constant friction force model ($\mathbf{F}_r = \mathbf{c}$) and the dashed line visualizes a force model depending linearly on the velocity ($\mathbf{F}_r = c \cdot \dot{\mathbf{x}}$). The constant model approximates the measured velocity far better than the linear model.

we have developed a new *Multi Hypothesis Ball Model* which can cope with jumping ball position measurements.

12.1.1 Multi Hypothesis Ball Model

The new ball model approach is based on the assumption that different objects on the soccer field can be detected as balls. For this reason, multiple potential ball positions can be tracked by a set of multiple *Kalman Filters*. Nevertheless it is necessary to receive less false positives than true positives, to be able to distinguish between the actual ball hypothesis and the false ones.

Each ball hypothesis is modelled by a linear *Kalman Filter* which tracks position and velocity (state vector: $\mathbf{x} = (x, y, \dot{x}, \dot{y})^\top$). The associated system matrix is defined as

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{12.1}$$

where ΔT describes the time offset between two predictions or corrections. This matrix causes the position prediction by integrating the velocity $\mathbf{v} = (\dot{x}, \dot{y})^\top$. The velocity is predicted by an additional friction model described in the following section.

12.1.2 Ball Dynamics

While a soccerball is rolling without any obstacle in its way, the acceleration can be modelled by the rolling resistance. The rolling resistance $\mathbf{F}_r$ is influenced by the weight of the ball and the material characteristics of carpet and ball [6]. It is especially independent from the velocity as Figure 12.2 illustrates.

The solid line results from a constant acceleration of $a_{\text{const}} = -220\frac{mm}{s^2}$ along the velocity direction. This friction model approximates the measured velocity quite well (cumulated error: $1384\frac{mm}{s}$). The dashed line visualizes an acceleration $\mathbf{a}_{\text{lin}} = -0.75\frac{1}{s} \cdot \mathbf{v}$ which depends linearly on the velocity and differs significantly from the measurements (cumulated error: $65730\frac{mm}{s}$).

Until now the linear model $\mathbf{a}_{\text{lin}}$ is used in the *NDevils* framework although it is an inappropriate approximation of the velocity. The new *Multi Hypothesis Ball Model* uses the more precise constant model a_{const}.

12.1.3 Conclusion

The reimplemented ball modelling improves two important factors of the ball state estimation. Firstly it is able to deal with a jumping ball perception and secondly the improved friction model allows a more reliable velocity prediction.

For a complete evaluation and comparison with the old model we need more test data. For the acquisition of evaluation data we will use the ground truth system described in Chapter 13 as soon as it is reliable. For this reason we cannot made a final assessment of the new ball modelling approach at this point.

Furthermore, the implementation is not yet fully completed. Sudden changes of the ball velocity have too little impact on the ball model after it has been correct for a long time. This should be solved in future work by adjusting the covariance matrix and considering kick motions. A filtering of the remote ball percepts received from teammates is also still missing and some known bugs have to be fixed.

12.2 Localization of Soccer Playing Humanoid Robots

For mobile robots it is indispensable to know where they are on the field. A robot solves this task by processing odometry data and information about its environment which are gathered by sensors. These sensors usually provide noisy signals which have to be filtered. Also the odometry is usually inaccurate. For this application several different filtering approaches exist. One of them is the *Monte Carlo Localization* (MCL) which estimates the true pose of the robot by using a particle filter. One advantage of a particle filter is the ability to filter non linear states, which is required for the movement of a robot. The next Section (12.2.1) gives a short overview of the *Monte Carlo Localization*.

The most common way to use MCL is to scan the environment with a laser range finder [32] and compare the resulting distances with a known map. This technique is well studied, thus there are many publications on how to apply it (e. g. [36]). But in case of the humanoid robot *NAO* from *Aldebaran Robotics* [29] there is no sufficient accurate distance sensor available. So we had to modify the common way [36] to update a sample of the particle filter. This procedure is described in Section 12.2.4.

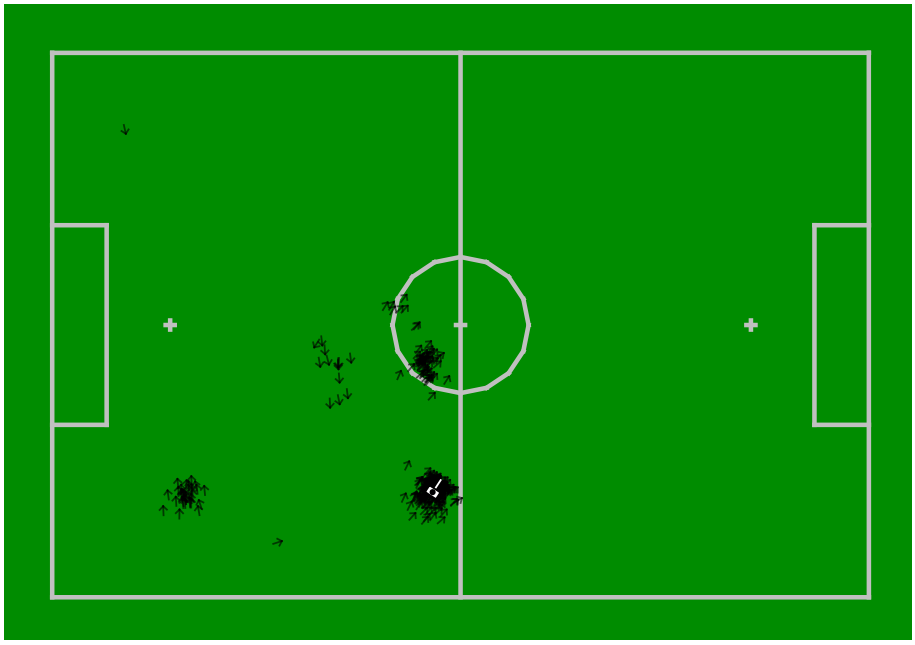


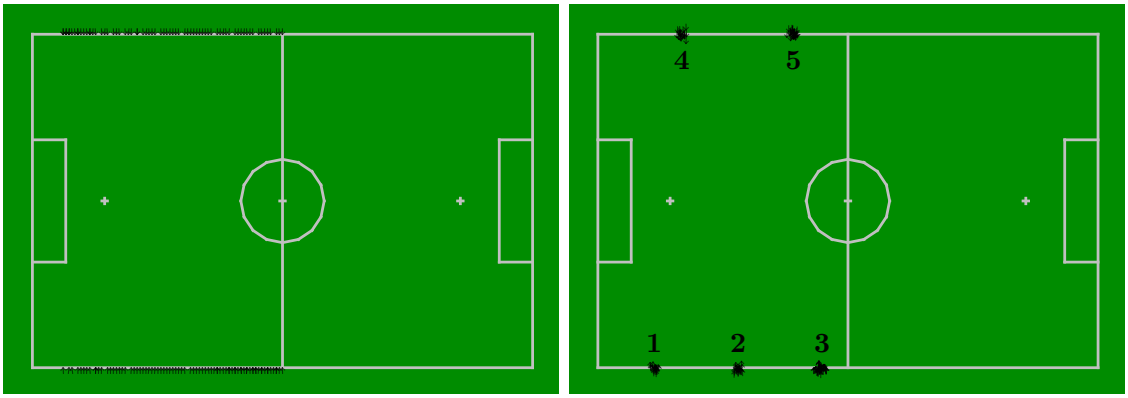
Figure 12.3: An exemplary set of samples (arrows) representing robot poses on a soccer field. The filtered robot pose (white rectangle) is drawn at the centroid of the region with the highest weight.

12.2.1 Monte Carlo Localization

The *Monte Carlo Localization* estimates the pose of a mobile robot based on a particle filter. This filter generates a set of particles or samples which represents possible poses. Each sample consists of a pose and a weight. The weight of a sample is the probability that this samples pose is identical to the actual robots pose. The probability of an entire region to contain the actual robot pose ascends with the number of samples in this region. The resulting robot pose is extracted from the set of samples by finding the region with the highest total weight of all samples in this region. See an exemplary sample set in Figure 12.3.

The particle filter is initialized with a random set of samples which covers all possible poses. The larger the area is, where the robot can be positioned, the more samples are needed to cover this area with an adequate density.

In each iteration the samples pass two update steps. First they are moved according to the odometry shift since the last iteration (motion update). Thereafter the weights are calculated for each sample by evaluating the sensor information (sensor update). After these update steps each sample contains its current probability in the form of a weight. Concerning the whole sample set these weights form a probability distribution. During the re-sampling a new set of samples is drawn out of the old one according to this probability distribution. This causes samples with a higher probability or weight to occur more frequently in the new sample set than less probable samples [36].



(a) Initialization on the sidelines.

(b) Initialization for each player on a unique position. The number at each sample cluster indicates the number of the corresponding player.

Figure 12.4: Different approaches to initialize the sample set of the particle filter.

12.2.2 Initialization

The soccer playing robot can be located everywhere on the soccer field. But at the beginning of a match the robots start at particular positions. In the Standard Platform League each robot has to start on a sideline of the own half. So the initial sample distribution can be limited to these lines. Thus the samples are spread with a normal distribution along the sidelines (see Figure 12.4a).

For further precision each players initial position can be restricted to one specific position. In that case the samples are spread for each player around its unique initial position (see Figure 12.4b). This leads to an instant correct localization, because the particle filter does not need some iterations to decide from where on the sidelines the robot has started.

12.2.3 Motion Update

The motion update applies the odometry change since the last iteration on each sample. For consideration of the inaccuracy of the odometry each sample is additionally moved by an random amount such that the sample set diverges during motion update. This procedure is taken over from [36]. Please refer to chapter 5.4 from this book for further details on the odometry model.

12.2.4 Sensor Update

One challenge of implementing MCL for soccer playing humanoid robots is the calculation of weights from sensor information (sensor update). There is no standard procedure, as for robots with a laser range-finder and a map of their environment (see [36]), which can be used to compute the weight of a sample by evaluating a picture taken by the robots camera.

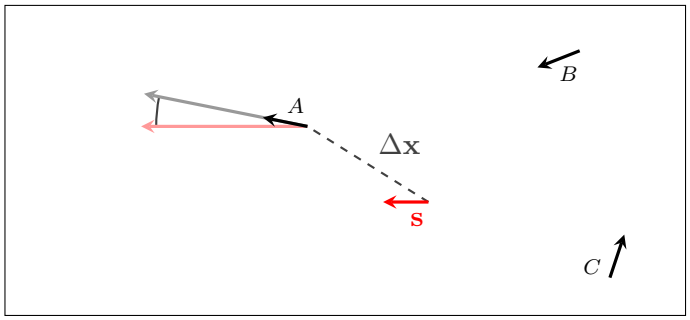


Figure 12.5: Comparing a sample $\mathbf{s}$ with multiple pose hypotheses (A, B, C). The translational difference $\Delta\mathbf{x}$ and the rotational difference $\Delta\alpha$ from the sample to the nearest hypothesis A is drawn into the plot.

Our framework already extracts some landmarks (lines, goals and penalty crosses) from the picture and provides positions of these landmarks relative to the robot. From these relative landmark positions we generate hypotheses where the robot could be located (see Figure 12.5). Based on these hypotheses we had to develop a heuristic for calculating weights. Therefore the best matching hypothesis is chosen with respect to translational difference $\Delta\mathbf{x}$ and rotational difference $\Delta\alpha$ (hypothesis A in the example from Figure 12.5). From the differences $\Delta\mathbf{x}$ and $\Delta\alpha$ between the pose sample and the pose hypothesis, two weights $w(\Delta\mathbf{x})$ and $w(\Delta\alpha)$ are calculated by the modified Gaussian distribution

$$w(\Delta) = \frac{\text{gaussian}(\Delta, \sigma)}{\text{gaussian}(0, \sigma)} \cdot (1 - w_{\text{min}}) + w_{\text{min}} \ . \tag{12.2}$$

A plot of this function is shown in Figure 12.6. Both single weights $w(\Delta\mathbf{x})$ and $w(\Delta\alpha)$ are combined by the weighted mean

$$w(\Delta\mathbf{x}, \Delta\alpha) = \frac{w_{\text{trans}} \cdot w(\Delta\mathbf{x}) + w_{\text{rot}} \cdot w(\Delta\alpha)}{w_{\text{trans}} + w_{\text{rot}}} \ . \tag{12.3}$$

The parameters σ, w_{min}, w_{trans} and w_{rot} have to be adjusted for an optimal localization result, where σ and w_{min} depend on the quality of the landmark (some landmarks are recognized with more precision than others) and w_{trans} and w_{rot} are global parameters.

12.2.5 Sample Clustering

When the samples are divided into multiple clusters (see Figure 12.7a) the overall robot pose cannot be computed from all samples. This would not result in an overall pose inside the best cluster (as shown in Figure 12.3). Thus the sample set has to be clustered.

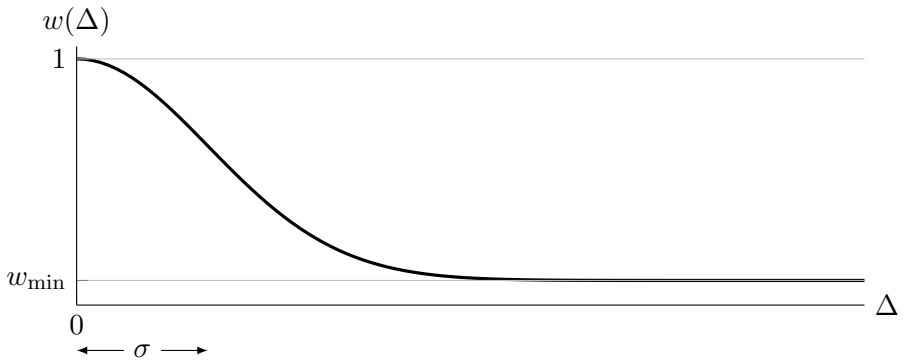
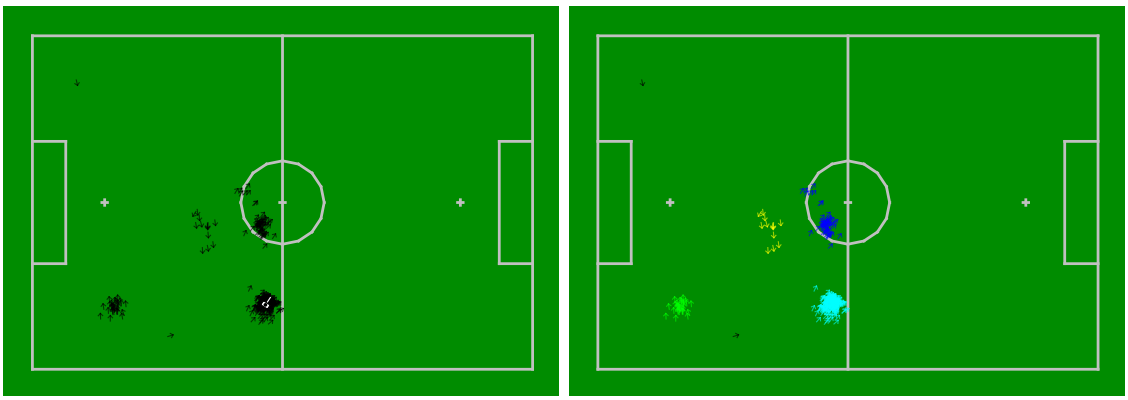


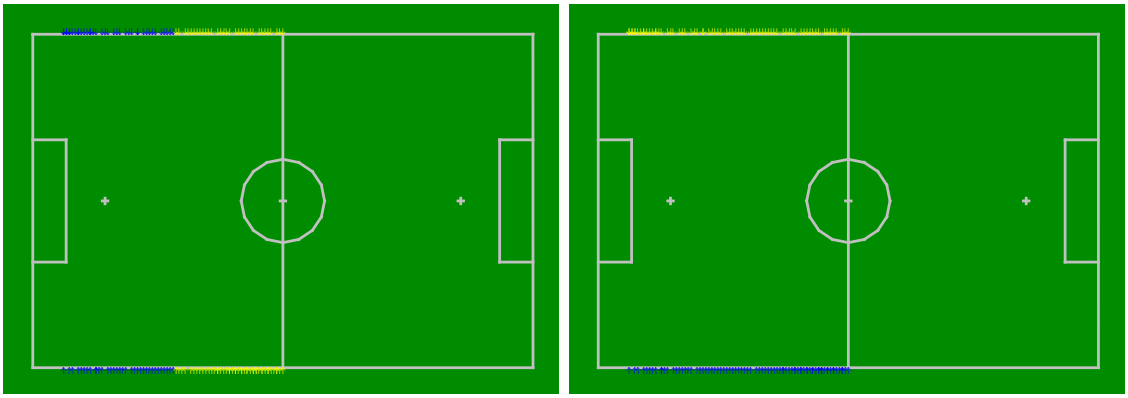
Figure 12.6: Heuristic for calculating sample weights by a modified Gaussian distribution (see Equation 12.2).



(a) Unclustered samples (same constellation as in Figure 12.3).

(b) Samples clustered by *DBSCAN*. Each color represents a cluster. Black samples are discarded.

Figure 12.7: An exemplary set of pose samples in multiple clusters.



(a) Initial sample set clustered with *k-means*. **(b)** Initial sample set clustered with *DBSCAN*.

Figure 12.8: An initial sample set clustered with *k-means* and *DBSCAN*. The clusters from *k-means* are useless.

12.2.6 DBSCAN

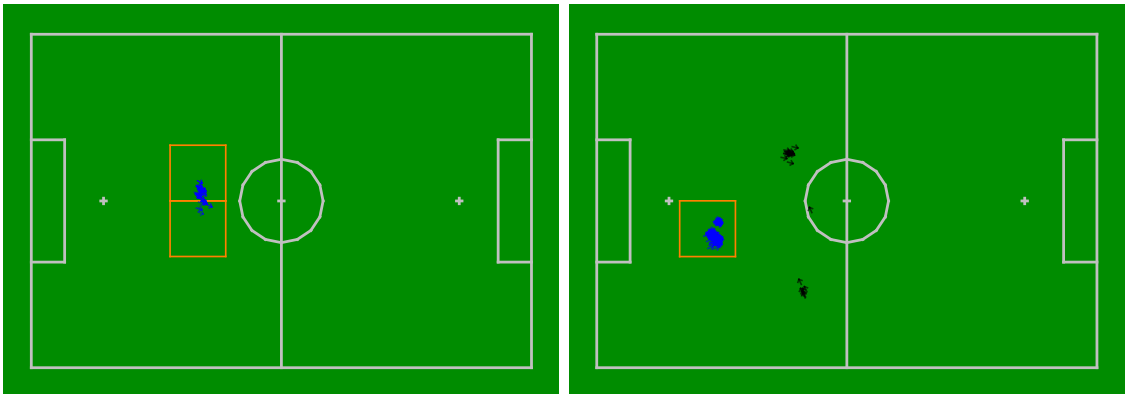
We have tested different clustering algorithms and we have come to the conclusion that *DBSCAN* [10] is most suitable. *DBSCAN* was already described in Chapter 6. It combines samples within a high density area to a single cluster. Isolated samples do not get assigned to any cluster. The samples in Figure 12.7b are clustered by *DBSCAN*.

12.2.7 K-Means

The clustering algorithm *k-means* (see Chapter 7.2.2) fails in sample constellations where the sample clusters are clearly separated, if the initial centroids are unfavorably chosen (see Figure 12.8a). In this exemplary sample constellation the clusters from *k-means* lead to the same problem as an unclustered sample set. The centroid of each cluster is located far outside of the dense regions. In contrast *DBSCAN* generates two clusters which contain in each case one of the sidelines where the sample density is high (cf. Figure 12.8b).

12.2.8 Grid Clustering

Another evaluated clustering algorithm is a grid clustering where the whole soccer field is divided into a grid of quadratic areas [19]. Each sample is assigned to one of these grid cells. Afterwards the cell with the highest overall weight is used as the origin for the main cluster. From that cell all neighboring cells with sufficient high weights are added to the cluster (see Figure 12.9). Hereby the main cluster is found if the grid size is appropriate for the cluster size. For closely together located clusters a small grid size is required which results in a more costly computation. To trace multiple clusters would also increase the computational effort.



(a) One cluster.

(b) Multiple clusters but only the largest cluster was found.

Figure 12.9: Two exemplary sample sets clustered by grid clustering.

12.2.9 Result

The current state of the implementation of MCL for soccer playing humanoid robots, as explained above, works well for situations where enough percepts are received to evaluate the samples with sufficient precision. For example this is the case when a single robot plays on a field with adequate lighting conditions.

However under difficult conditions the MCL approach is yet less accurate on the *NAO* robot than the old approach which is based on multiple *Kalman Filters*. Especially the resolution of symmetry is problematic. There could be further improvements on the new approach in the future so that it will probably achieve more precise results. Furthermore, a comprehensive evaluation, which will be done using the ground truth system described in Chapter 13, is necessary.

13 Cost-Effective Ground Truth System For Localization Evaluation

Janine Hemmers

Ground truth systems are important for mobile robots to evaluate different kinds of algorithms, such as self-localization. Therefore the ground truth system needs to work more exact than the algorithm that needs to be evaluated.

As already discussed in Chapter 12.2, a most precise self-localization is essential for soccer playing robots. There are different approaches for a proper self-localization and their precision differs in various situations. It is important to measure how precise a self-localization approach and it implementation works in a specific situation. Therefore the exact location of each robot needs to be measured and compared to the calculated position of this robot with the chosen self-localization approach. If this data is collected over time, it can be evaluated through different statistical methods. This leads to an evaluation of the chosen self-localization approach and makes it comparable to other implementations or approaches.

To compare the new self-localization approach, described in Chapter 12.2, to our old approach and later on, to an improved version of either approach, we started developing a ground truth system.

13.1 Different Ground Truth System Approaches

Since a precise localization is one of the most important parts for a high-quality soccer game, a lot of teams already have a ground truth system with different approaches like [25] and [26].

In [25] an approach for a ground truth system is proposed, where the sensor data for a single robot is logged and its motions are captured with a motion capturing system, as is done for instance in animated movies. Therefore 15 infra-red cameras are positioned around a standard SPL soccer field ($9 \times 6m^2$). Additionally special markers need to be applied to the robot, which can be tracked by the infra-red cameras. To capture motions of humans usually 42 markers are used, but because of the size of the robot compared to a human, only eight markers are used. Even if the results with this approach are promising, it would not be suitable for our purposes.

[26] pursues a very different approach, which needs less special equipment and, in contrast to the system proposed in [25], is able to perceive the position of more than one robot. For

this approach four *Kinect*[1] sensors are positioned around a testing field. The testing field used has the dimensions $4 \times 3m^2$ and therefore is smaller than a quarter of a SPL soccer field. With the *Kinect* sensors it is possible to use and evaluate the depth information and to calculate the positions of the robots with this information.

Both approaches need special, partly expensive equipment. Also neither approach can be used to monitor a whole soccer game, with 10 robots on the field. But to evaluate the self-localization in a proper way, it is important to test it under the same conditions as in a real competition. Other robots, obstacles and humans on the field might influence the self-localization in a way, that it works very precise if the robot is all alone on the field, but not in other cases. Therefore we decided to develop a ground truth system approach, where it is possible to monitor the whole field with additionally low cost equipment.

The goal is to monitor the field from the top with four webcams that are positioned at the ceiling. The robots will be marked with different colors on their head and shoulders, so that it is possible to distinguish them from each other and to perceive their orientation even with the head tilted or turned. The advantages of this approach are the interchangeable low cost cameras, of which only four are needed to monitor the whole field.

13.2 New Approach and Tool Development

To test the approach first, only one low-budget webcam, *Logitech B910 HD*[2], is available. This webcam has 78° wide angle field of vision, the ability to stream high definition videos with a resolution of 720 x 1280 at 30 frames per second. With this camera it is possible to see a quarter of a standard SPL field, which makes it possible to observe the whole field as planned with four cameras.

To calibrate the cameras and to evaluate the camera streams, we developed an application with *Qt*[3], because of the different available libraries and functionality, for instance *OpenCV*[4]. After the user included the four cameras, he needs to calibrate them. When all cameras are calibrated, the tool can be used to monitor a soccer game. It streams the videos of the different cameras, calculates the position and orientation of each robot and sends the evaluation to the robots.

To calibrate the camera we use homography as is described in Chapter 9.2 and Chapter 9.1. After loading a specific camera, the user needs to match four points on the camera image to four points on the field (see Figure 13.1). The calculation of the homography matrix and warping the image with this homography matrix is done with the *OpenCV* library.

OpenCV can also be used for color detection of the coloured marks on the robots. Since it is a commonly used library for image processing, there are already different algorithms for color detection available. The chosen approach for the color detection is very interchangeable inside the developed tool. For testing purposes we used parts of the ¨Multi Object

[1] *Microsoft Kinect* http://www.xbox.com/de-DE/xbox-one/accessories/kinect-for-xbox-one
[2] Logitech B910 HD http://support.logitech.com/en_us/product/b910-hd-webcam
[3] *Qt* http://www.qt.io/
[4] *OpenCV* http://opencv.org/

Figure 13.1: Screenshot of the developed tool after calibrating the camera. Left window shows the camera stream with the marked points, right window shows the calculated warped camera image, with the corresponding points, overlay an image of the field.

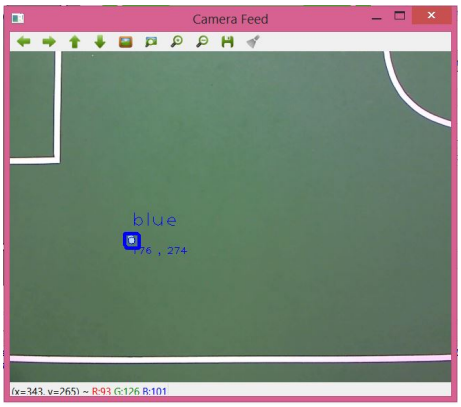


Figure 13.2: Screenshot of the developed tool while detecting a blue marker on the field.

Tracking Based On Color¨-algorithm by Ahmad Kaifi [17]. Although we did not test it with color-marked robots yet, it provides very promising results even for very small marks, as you can see in Figure 13.2. If the algorithm is robust enough must be tested.

13.3 Calculation from Image to Field Coordinates

The position of a robot on the field, as seen in an image, is basically determined by the robots height and the distance between the ground and the camera, and can be calculated with simple intercept theorem. Figure 13.3 shows the important parameters. The parameter h is the measured height of the camera. For the calculation the distance from ground to camera lens is needed. Since the distance between the camera chassis to its lens is only very small, the variance will be very small too and therefore can be neglected. The parameter r is the height of a robot. It is important to measure it while the robot is playing, because their playing stance is crouched.

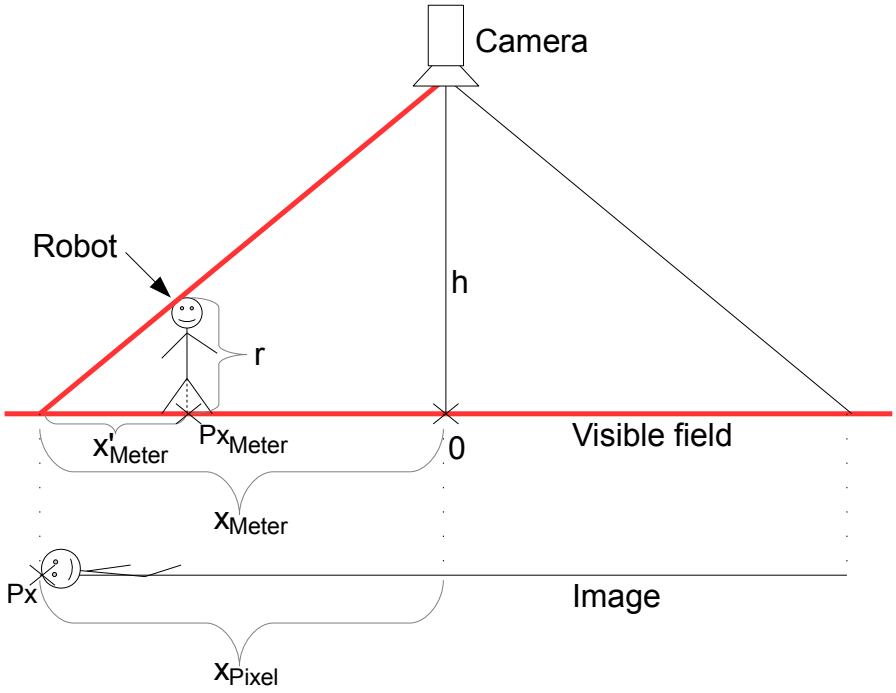


Figure 13.3: Important parameters for the conversion of image coordinates to field coordinates.

The position of the robot needs to be calculated separately for both axis. Both calculations are analogue and therefore we will only consider the calculation of the robots X-position. Because of the camera calibration at the beginning, we know what part of the field we see on the image and what measurement it has. Therefore we can convert the image scale to the real dimension.

We consider the sides of the image to be our X-axis and Y-axis whereas the origin is in the middle of the image. The same applies to the part of the field we can see in the image. We define x_{Pixel} to be half of the image side, measured in pixel, corresponding to the X-axis, which means it is the highest value X can be. The same applies again to x_{Meter}, measured in meters, and the part of the field we see in the image. The size of the part of the field you can see in x_{Pixel} of the image match x_{Meter} on the real field.

If we detect the colored mark on top of the head of a robot in an image, we need to calculate were the robots feet are in real dimensions. Therefore we pretend the head would lie directly on the ground where we see it in the image. So if we calculate the robots head position from image dimensions to real dimensions, there is a variable distance between this position and the position of the robots feet. The difference from head to feet of a robot in a taken image, depends on the distance between the robot and the origin. For calculating the X-position of the robots feet in real dimensions only the distance on the x-axis to the origin is important. If the robot is located at the origin of the X-axis, the X-position of its feet in real dimensions will be the same than the X-position of its head converted from image dimensions to field dimensions, as described above. If the robot is located at the largest distance to the origin of the X-axis Px_{Meter}, where its head is still visible in the

image, like Figure 13.3 shows, the difference from the X-position of its head Px, converted from image dimensions to real dimensions, to the X-position of its feet, in real dimensions, is at its maximum. x_{Meter} describes exactly this difference at this specific position. We assume that this difference is linear, so if the robot is half way between the origin of the X-axis and Px_{Meter}, the variation of the x-position of its head and the X-position of its feet, both converted to real dimensions, will halve. So with Equation (13.1) it is possible to calculate the difference for each position on the X-axis if x'_{Meter} is known.

$$\text{dif}_x = \frac{Px}{x_{\text{Pixel}}} \cdot x'_{\text{Meter}} \tag{13.1}$$

To calculate x'_{Meter} we use the intercept theorem. The red lines in Figure 13.3 show the needed intercepting lines. The theorem states that the ratio between r and h is the same as the ratio between x'_{Meter} and x_{Meter}. Equation (13.2) shows the rearrangement of the corresponding equation.

$$\frac{x'_{\text{Meter}}}{x_{\text{Meter}}} = \frac{r}{h} \Rightarrow x'_{\text{Meter}} = \frac{r \cdot x_{\text{Meter}}}{h} \tag{13.2}$$

To get the x-position of the robot in real dimensions, we first need to convert the X-position of the robots head Px_{Head} as seen in the image to real dimensions (see Equation (13.3)).

$$Px_{\text{Head}} = \frac{Px}{x_{\text{Pixel}}} \cdot x_{\text{Meter}} \tag{13.3}$$

Than we can subtract the calculated difference to its feet from the X-position of its head and finally get the X-position of the robot in real dimensions (see Equation (13.4)).

$$Px_{Meter} = Px_{Head} - \text{dif}_x \tag{13.4}$$

The Y-position of the robot in real dimensions can be obtained the same way.

To get the field coordinates of the robot, we can once again use the camera calibration. We know exactly what part of the field we see in the images and we now where the robot is positioned on this field part, therefore the calculation of the field coordinates is trivial.

Of course this approach does only work if r and h are perfectly parallel. In most situations this will not be the case, because the robots wiggle while they move and it is not always assured that the camera mounted that precise. But we assume that this will only lead to small, acceptable discrepancies in the calculation.

13.4 Intermediate Result

We proposed a new approach for cost effective ground truth systems. The whole approach is very promising, but it still needs proper testing. We developed a tool, that is already capable of calibrating different cameras. It can detect colors and the conversion of the robot position from image to field coordinates is already integrated. We need to define a proper set of marks for the robots and implement the logic behind it into the tool. If that is working, we than can properly test our approach.

14 Technical Challenges

Karen Bieling, Marco Dürr, Janine Hemmers

In the following chapter, we describe the development of the technical challenges we participated in at the RoboCup 2015. Details for the setup and procedure of all challenges can be found in [8, P. 2 f.].

Additional parts like the penalty shootout which can be considered as a small challenge and were developed in the year 2015 will be explained in this chapter as well.

14.1 Many Carpets Challenge

The *Many Carpets Challenge* was one out of three technical challenges of the *Standard Platform League* (SPL) at *RoboCup* 2015. A robot had to walk on an unknown carpet without any seams or lines and should score a goal. The carpet had to be significantly different in texture and fiber height from the current one used in the SPL. Purpose of the challenge was to improve the motion skills in sense of playing on a more realistic underground such as a real green.

14.1.1 Rules and Procedure

The competition took place on three different carpets which were chosen by the Technical Committee. According to the rules, only the robot, one ball and one goal were placed on the field. The positions of these objects were unannounced and chosen immediately before the competition started on each carpet. Hence each team was given the same setup to ensure equal chances. The initial game state for the robot is *set*. If the game state switches from *set* to *playing*, the robot should move to the ball and score a goal within 1 minute. The challenge was judged by the normal SPL game rules. If any infringement occurs or a goal is scored the competition for the current carpet is finished.

In Figure 14.1 you can see a possible experimental setup for all objects which need to be placed in the *Many Carpets Challenge*. The robot R_1 starts near the orange colored ball and is not pointed straight forward to the goal. It follows that the robot needs to turn around the ball. This non optimal position is chosen to get a better impression about the walking skills on the different grounds.

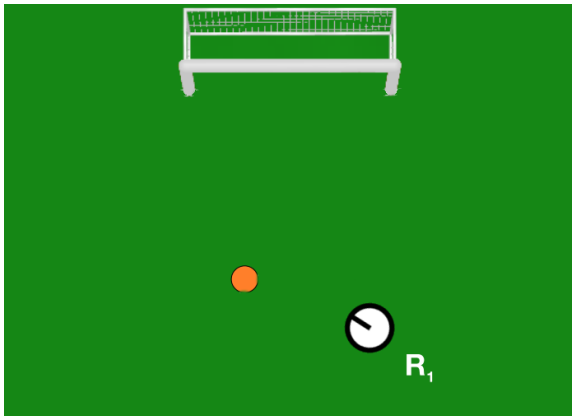


Figure 14.1: Example start position and setup for the Many Carpets Challenge.

14.1.2 Adjusted Behavior Model

Several changes had to be made on our team's existing behavior in order to participate in the challenge. Major issues in this challenge, besides the walking on a different floor, are the missing field lines and the different sized carpet. This is why the robot cannot localize itself in the same way as it does on a standard SPL field. Hence the only object used for localization is the goal or the goal posts respectively. The second reason why we need a modified behavior is because the whole experimental setup is simplified, no other players, no obstacles to avoid and no team communication.

The modified behavior basically consists of three options. The initial option is *StartMCC* which waits for pressing the chest button once. After that the robot should stand up and the behavior starts the button interface, the head control functions and the LED control functions, which are used similar to normal tournament games. As last procedure the behavior changes to the option *BodyControlMCC*. In this option the robot will wait for the *GameController* to set the game status from *set* to *playing* and starts another option called *PlayMCC*. In all other states e.g. *ready* or *penalized* the robot will just stand up straight.

The option *PlayMCC*, shown in Figure 14.2, holds the main part of the decisions the robot does. The initial state is *positioning* which is used to check if the robot has lost track of the ball. In case of a ball loss the robot walks in a defined direction to get the ball back into the focus of his vision. If the robot sees the ball he moves towards it until a certain distance is reached. After the action of moving to the ball is complete he should rotate around the ball until the robot sees two goal posts. By seeing two goal posts the robot can figure out which of them is the right and the left one. This is the information it needs to get himself in the optimal position to kick the ball into the goal. After the kick is done the state machine returns to the state *positioning* again.

14.1.3 Adjusted Walking Parameters

To optimize the walk of the robot only a few changes had to be made to the walking parameters. The material properties of the carpet result in relative difference in height of

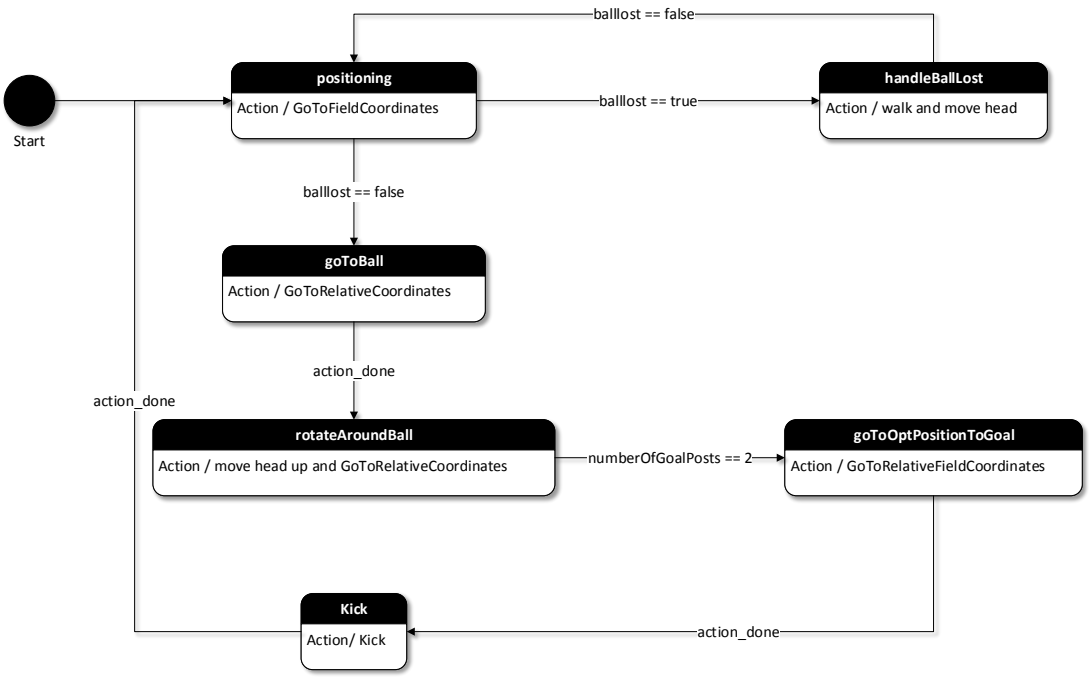


Figure 14.2: Visualization of the *PlayMCC* option.

the robot feet. As such the robot is unable to gain a firm balance. In order to imitate the human gate, the center of mass of the robot is lowered. Additionally, a side step movement was added to stabilize the robot.

14.1.4 Results

First of all the developed behavior was tested in a simulation. A qualitative analysis of the results of the simulation gave a positive indication. Even if a worse case scenario is considered, where the robot starts the challenge with its back pointing towards the goal, the task in terms of scoring a goal was accomplished.

For testing the walk, the robot needs to be started on the real field because the simulation cannot reproduce the explicit properties of the carpet. In that case, the robot exhibited sub-optimal stability while performing the walking task preventing it from accomplishing the task in an acceptable time limit. However, a positive observation was that the robot did not fall over during its attempt to achieve stability.

Another problem was the goal perception. In the simulation, the identification of goalposts was successful even while the robot was in motion. In comparison to this, on the real field, the identification of the goalposts was unreliable even when the robot was set in a stable position. In a specific situation even the wall was recognized as a goalpost. This analysis

leads to the conclusion that high priority must be given to optimize the goal perception module or define another concept to identify the goal.

14.2 Corner Kick Challenge

The corner kick challenge was one of three technical challenges in the *Standard Platform League* at this years *RoboCup*. Purpose of this challenge was to encourage passing skills, team play and perception of opponent robots.

For the setup a ball is placed at one corner of the field. The first robot, henceforth referred to as corner robot, is placed on the intersection of the sideline with the virtual extension of the penalty area's front line. The second robot, henceforth referred to as field robot, is placed one meter behind the penalty cross, which is the standard penalty kick position. Figure 14.3 shows a detailed overview of the setup. Both robots will be positioned by the Technical Committee, therefore teams may not choose which of their robots will be put on which position.

The aim of this challenge is to achieve as many goals as possible in 3 minutes. A goal is only achieved, if both robots have touched the ball and the corner robot was the first robot which touched the ball.

After each goal the robots are put back to their initial positions (see Figure 14.3) and a non moving opponent robot is put on the field as an obstacle. After scoring four goals the setup remains constant so no additional robots are placed on the field. The positions of the opponent robots are decided by the Technical Committee and were the same for every team. Details for the setup and procedure of the corner kick challenge can be found in [8, P. 2 f.].

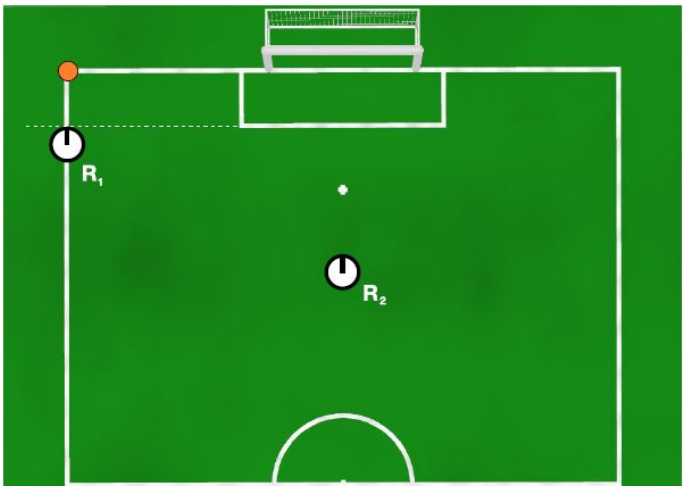


Figure 14.3: Setup for the corner kick challenge. Robot R_1 is referred to as corner robot. Robot R_2 is referred to as field robot.

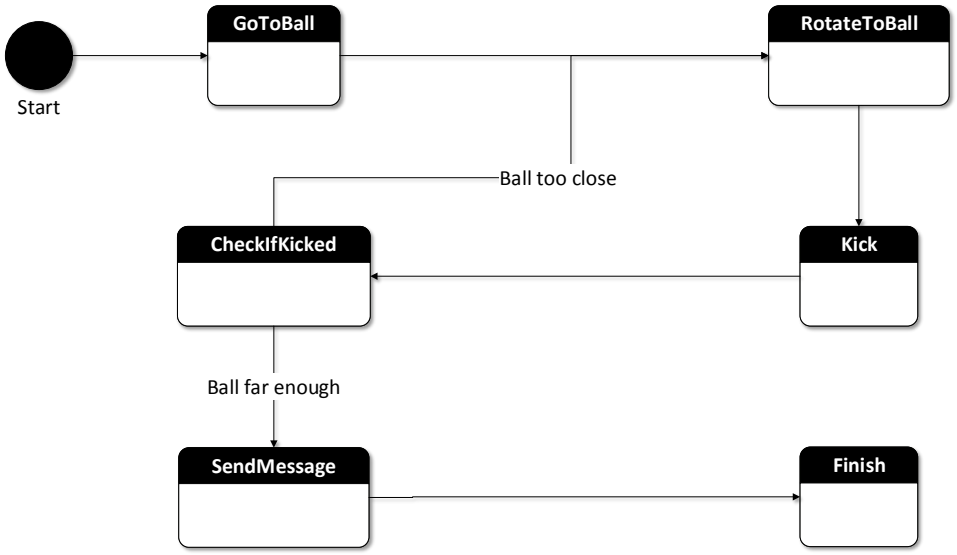


Figure 14.4: State machine for the behavior of the corner robot. Blank transitions toggle, if the robot is ready with a given set of movements and special actions.

14.2.1 Corner Robot

Figure 14.4 shows the state machine for the behavior of the corner robot. After it decided to be the corner robot it goes to the ball in the corner and faces it, leaving enough distance so that it does not hit the ball while passing it. The right position is calculated from the corner coordinates and an x and y offset. Once standing behind the ball the corner robot rotates such that it can pass the ball straight to the field robot which has moved forward in the meantime. Performing the kick includes walking to a optimal kick position and performing a kick motion.

If the ball was not hit well and still stays in the outer part of the field ($> 2m$ from the central axis) the corner robot performs a second kick such that the field robot does not have to walk such a long distance. After that the corner robot stops moving and tells its teammate that its action is completed. Therefore just one player, the corner robot, needs to decide, whether the ball has been kicked far enough.

14.2.2 Field Robot

Because of the distance to the ball, the robot at position R_2 (see Figure 14.3) knows it is the field robot and starts the field robot behavior (see Figure 14.5).

The field robot positions itself closer to the goal and to the corner robot, while it waits for the corner robot to communicate that the ball was kicked. When the field robot reaches a

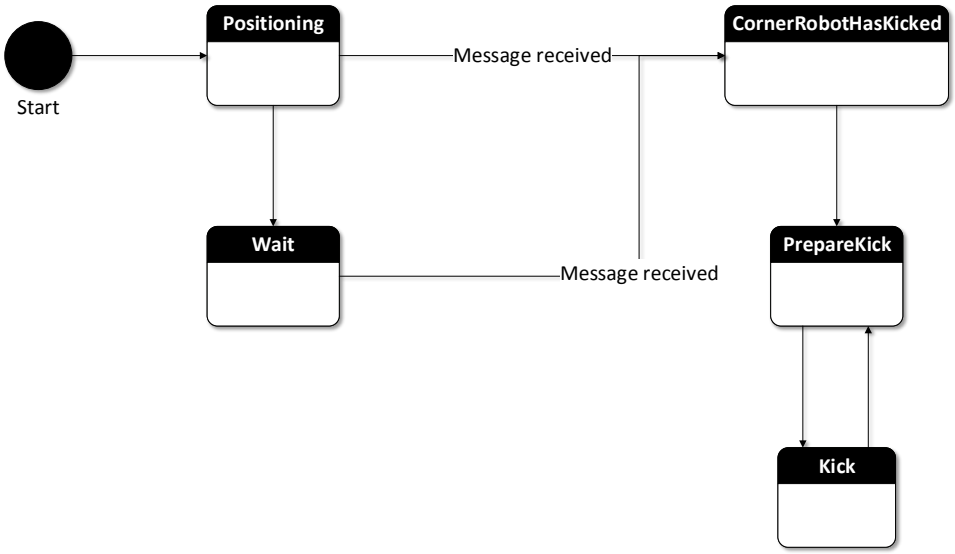
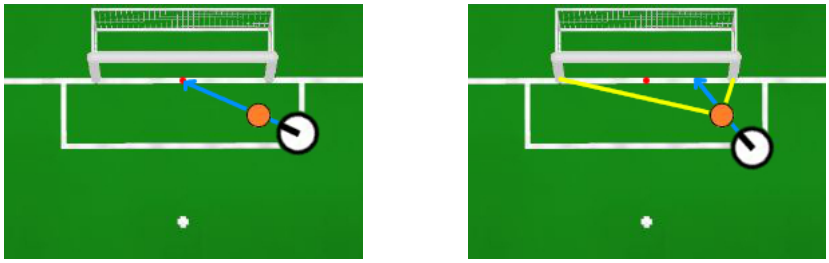


Figure 14.5: State machine for the behavior of the field robot. Blank transitions toggle, if the robot is ready with a given set of movements and special actions.



(a) Aim for the center of the goal.

(b) Aim for the angle bisectors between the two goal posts.

Figure 14.6: Different approaches to aim for the goal.

final position, which is thought to be an optimal position, it stops its movement and just waits for the corner robot to kick the ball.

After the corner robot has communicated that the ball was kicked, the field robot goes to the ball and prepares the kick. It has to position itself behind the ball, so that it is able to kick the ball and score a goal. It would be possible to just aim at the center of the goal, but there are specific situations, e.g. if the ball is close to a goal post (see Figure 14.6a), where the angle is too sharp. In these cases, there is a high probability that the ball will miss the goal and leave the field. If this happens, the ball would be placed in the corner again, but the robots would stay where they are. This would mean a huge disadvantage for us. Therefore the robot kicks the ball along the angle bisectors between the two goal posts (see Figure 14.6b).

14.2.3 Team Communication

The greater the distance to an object is, the bigger gets the uncertainty of the robot's perceived distance. Therefore relying only on the vision when deciding whether the corner robot has already kicked or not is dangerous for the field robot. It could lead to the worst case, where the robot thinks that the corner robot already kicked the ball, even if it is still lying in the corner. Then this robot would walk towards the ball an would try to kick the ball into the goal from this position. With this attempt the robot would possibly miss the goal and loose valuable time. For that reason the corner robot sends the information that the ball was kicked via WiFi. Also it decides if the ball was kicked far enough. In this case should be only a small variation in the measurement of the distance to the ball, because the corner robot is very close to the ball.

14.2.4 Corner Kick without Communication

Communication between the teammates needs WiFi. We expierenced a lot of cases were WiFi is impaired on large events due to a lot of WiFi traffic. Impaired WiFi can mean that the field robot has to find out by itself whether the ball has been kicked or not. The distance to the center axis from which on it starts approaching the ball is $2.5m$, $0.5m$ larger than the corner robot uses for a valid kick. Because of the overlapping, there is no area where none of the robots go to the ball and also it ensures that even if there is an inaccurate measurement of the distance to the ball, at least one robot goes to the ball.

14.2.5 Results

We achieved the second place in this years technical challenges. The results of the corner kick challenge itself are not as expected. Sadly our corner robot had a broken hip joint. At least both robots touched the ball once and the corner robot was the first who touched the ball. We did not score any goals, but with both robots touching the ball at least once, we achieved more than most of the other teams. If there should be a similar challenge in the future, our approach can certainly be reused.

14.3 Penalty Kick Shoot-out

To determine the outcome of a game that ended in a draw, a penalty kick shoot-out is performed. Like in human soccer, the team with most of five attempts wins the shoot-out. Each team's robot either has to play the role of the *striker* or the *goal keeper*. Therefore, two different behaviors have to be implemented. The major rules which have to be considered are shortly described. The detailed rules can be found in [7, P. 21 ff.].

The *keeper* is placed on the goal line in the center of the goal. It is not allowed to touch the ball outside the penalty area. The *striker* is placed one meter behind the penalty cross, looking towards the ball. It has one minute to kick the ball, but is not allowed to touch it twice or more.

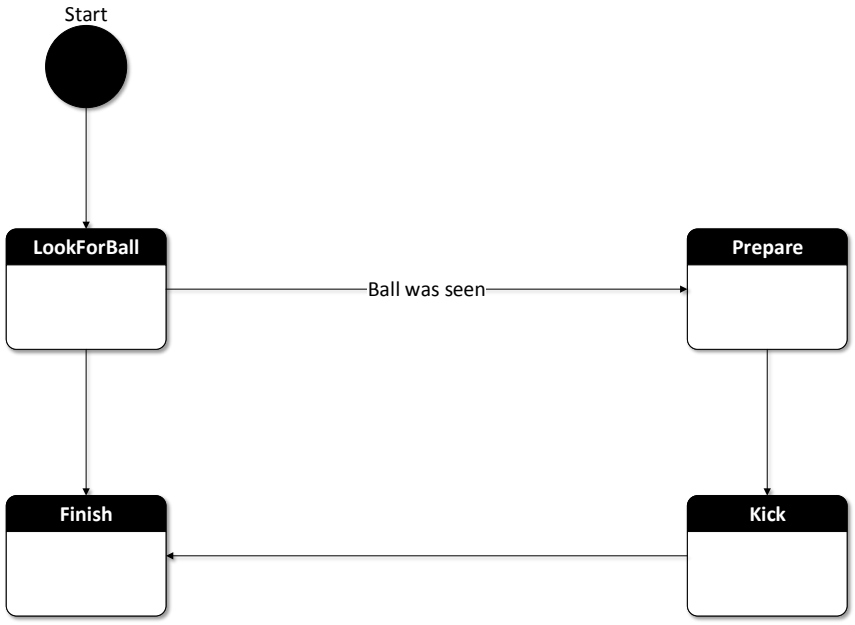


Figure 14.7: State machine of the behavior for the *penalty striker*. Blank transitions toggle, if the robot is ready with a given set of movements and special actions.

Like a lot of other teams we never had a special behavior for the penalty *striker* or *keeper*. If it was necessary we used the behavior for a normal match for our *goal keeper* or *striker*. Therefore we implemented a new behavior for the *penalty striker*, additionally using a new kick, as well as a new behavior for the *penalty keeper*, which also uses a new special action. Implementation details are considered in the next section.

14.3.1 Striker

Figure 14.7 shows the state machine of the behavior for the *penalty striker*. After finding the ball in the state *LookForBall* it walks towards the ball as described below, performs a hard kick and finally stops moving.

In the state *Prepare* the *striker* simply walks towards the ball, so that it is ready to kick it with a given angle towards the goal. Since it is not possible to find out which goal is the opponent goal, due to the symmetry of the field, the *striker* might try to kick into the wrong goal. This can easily found out by asking the robot on which side it sees himself. If it is wrong, the optimal kick position has to be rotated by 180°.

14.3.2 Keeper

The *penalty keeper* waits until the ball is moving or will reach its own Y-axis in a given amount of time. Depending on where the ball will meet the *keeper's* Y-axis it performs a

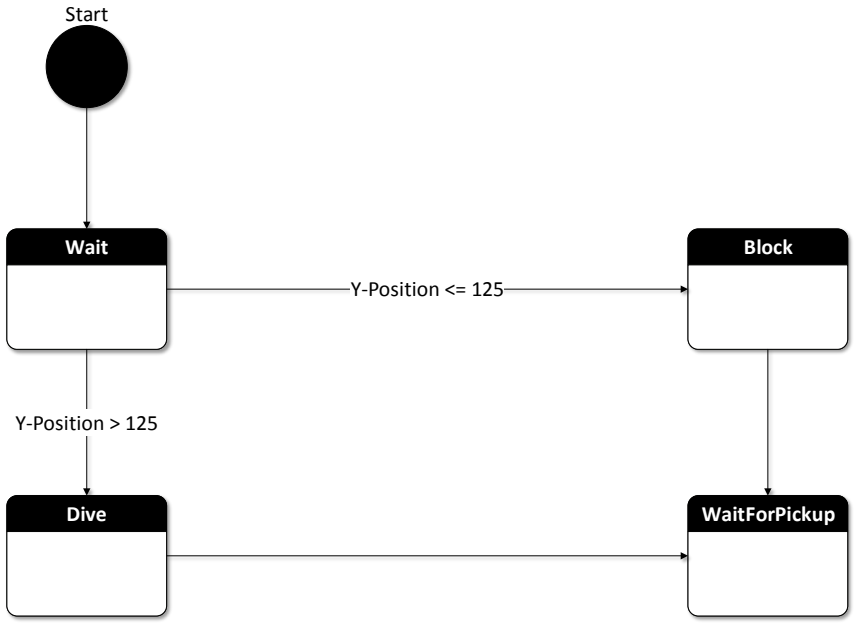


Figure 14.8: State machine of the behavior for the *penalty keeper*. Blank transitions toggle, if the robot is ready with a given set of movements and special actions.

dive or a block. After performing one of these special actions the *penalty keeper* will not stand up, but instead ease all of its joints and waits for a pickup. This way the *keeper* is able to perform actions from which it does not need to be able to stand up from on its own. See Figure 14.8 for a detailed overview.

Therefore we implemented a new way to block the ball. The *keeper* will fall into a pose, where it is stretching its legs to every side and balances with its hands in front of it on the ground. The dive action stayed the same like in a normal match because it provided great results in the past.

14.3.3 Results

We have implemented a functioning behavior for the *penalty striker* and for the *penalty keeper*, which can now be used at competitions. Future work on this topic should include the optimization of our penalty kick shoot-out as well as extensive tests in competition like conditions.

15 Joystick Control

Heiner Walter

We have developed a module within the *NDevils* framework that allows us to control a *NAO* robot using a joystick or gamepad. The following chapter explains the important steps required in order to receive commands from a gamepad and transfer them to the motion control of the robot. It also describes how to configure the module *JoystickControl* for use with a specific gamepad.

15.1 Compiling Kernel Modules

We decided to plug the gamepad directly into the USB port of the *NAO* which is located under a cover at the back of its head. Another approach is to connecting the gamepad to a PC and transfering the commands via WiFi to the robot. This technique is already implemented in our framework, but it is more impractical because of its lack of flexibility and the additionally required hardware.

By default it is not possible to access an USB gamepad from the *NAO*. The Linux drivers which are required for joystick support are missing. Therefore the drivers must be compiled for the currently used Linux kernel. The source code of *Aldebaran's* Linux kernel which is used on the *NAOs* can be downloaded from *GitHub*[1]. Our robots currently use version 2.1.4.13[2]. This is the version number given by *Aldebaran* and its not the same as the Linux kernel version.

The easiest way to compile kernel modules for the *NAO* is to use the *NAOqi VirtualBox* image provided by *Aldebaran*[3]. The virtual machine can be accessed from the host system by `ssh` and `scp`. This makes work easier and opens up the possibility to copy files between host and virtual machine. The *VirtualBox* image is configured to be accessible via IP `localhost` and port number `2222`[4]. For connecting via `ssh` from Linux or OS X hosts use the command

```
ssh -p 2222 nao@localhost # username and password are 'nao'
```

[1] Source of *Aldebaran's* Linux kernel can be downloaded on the website https://github.com/aldebaran/linux-aldebaran.

[2] Source of the kernel version we are currently using can be downloaded on the website https://github.com/aldebaran/linux-aldebaran/tree/release-2.1.x/atom.

[3] The *NAOqi OS VirtualBox* image can be downloaded on the *Aldebaran* developer resources download page: https://community.aldebaran.com/en/resources/software/.

[4] Tutorial on setting up the *NAOqi OS* virtual machine can be found on the website http://doc.aldebaran.com/2-1/dev/tools/vm-setup.html.

For sending or retrieving data via `scp` use

```
scp -P 2222 <path on host> nao@localhost:<path on virtual machine>
scp -P 2222 nao@localhost:<path on virtual machine> <path on host>
```

After downloading the kernel source it has to be configured. It is important to compile the kernel with the configuration running on the *NAOs*. For that reason the kernel config file has to be copied from a *NAO*. It is there located in the directory `/boot/` and is named `config-<kernel version>`. In our case the config file is currently named `/boot/config-2.6.33.9-rt31-aldebaran-rt`. After this config file was copied to the virtual machine the kernel source can be configured using a *ncurses* based configuration assistant[5]. Executing the command

```
make O="../linux-aldebaran-build" menuconfig
```

in the source directory runs the configuration assistant and sets the output directory (`O=...`). In this assistant the config file must be loaded at first. Then all modules are selected which are already part of the *NAO* kernel. For using gamepads the following modules must be selected in addition:

- For *DirectInput* gamepads:
  - `joydev`
- For *XInput* gamepads:
  - `xpad`
- If force feedback should be used (additional):
  - `evdev`
  - `ff-memless`
  - `xpad` with additional force feedback support

All this modules can be found in the menuconfig assistant at `Device Drivers - Input device support`. The module `xpad` can be found in the further directory `Joysticks/Gamepads`. Select them as modules (`<M>`) not as build-in (`[*]`).

After selecting all modules save the configuration, leave the assistant and compile the modules using

```
make O="../linux-aldebaran-build" modules
```

[5] A tutorial on compiling a Linux kernel can be found on the website https://wiki.ubuntuusers.de/Kernel/Kompilierung/.

into the output directory. The compiled modules `joydev.ko`, `ff-memless.ko`, `evdev.ko` and `xpad.ko` can be found in `<outputdir>/drivers/input` or `<outputdir>/drivers/input/joystick`. They must be copied into the respective directory

```
/lib/modules/<kernel version>/kernel/drivers/input # or
/lib/modules/<kernel version>/kernel/drivers/input/joystick
```

on the *NAO* to be automatically loaded by the operating system. The `<kernel version>` can be figured out by the command `uname -r`. The new modules are loaded by the system either after a reboot or by executing the command `depmod -A`[6].

15.2 Accessing Joystick Input

When a gamepad is properly connected to the *NAO* and all required joystick drivers are loaded, joystick events can be read from the file descriptor `/dev/input/js0`. These events are triggered by axis movements and presses of buttons. Each event contains a timestamp, the event type (button or axis), the button or axis number and the new value.

15.3 Configuring Framework Module JoystickControl

For controlling the robots movement we have developed the module *JoystickControl* within our *NDevils* framework. It reads the incoming joystick events and executes actions which was assigned to the events.

The mapping of buttons and axes to actions is realized by two configuration files. The first config file `JoystickControlParameters.cfg` defines parameters related to the robots movements (e. g. maximum joint angle, maximum velocity). This file also determines the model of the currently used gamepad.

For each gamepad model an individual instance of the second config file can be created. This file stores the actual mapping of buttons and axes which differs depending on the gamepad model. Which gamepad specific config file to use is set by the first config file. By default the gamepad specific configuration file `Joystick_Default.cfg` is selected.

[6] A tutorial on Linux kernel modules can be found on the website https://wiki.ubuntuusers.de/Kernelmodule/.

```
# Open ssh session:
ssh -p 2222 nao@localhost
# Clone source repository:
# Replace branch 'release-2.1.x/atom' by the required version.
git clone -b release-2.1.x/atom https://github.com/aldebaran/linux-aldebaran.git

# Copy NAO kernel config file:
# First it has to be copied from a NAO where it is located at
# /boot/config-2.6.33.9-rt31-aldebaran-rt (file name depends on version)
# Then copy it from host to vitual machine:
scp -P 2222 <path to config file> nao@localhost:~/config

# Configure kernel source on virtual machine:
# Create output directory.
mkdir linux-aldebaran-build
# Go to source directory.
cd linux-aldebaran
# Run configuration assistant and set output directory. Within the
# assistant load the config file from NAO and add desired kernel modules.
make O="../linux-aldebaran-build" menuconfig
# Compile all kernel modules selected by the configuration.
make O="../linux-aldebaran-build" modules

# Copy compiled modules from
~/linux-aldebaran-build/drivers/input/joydev.ko
~/linux-aldebaran-build/drivers/input/ff-memless.ko
~/linux-aldebaran-build/drivers/input/evdev.ko
~/linux-aldebaran-build/drivers/input/joystick/xpad.ko
# to
/lib/modules/<kernel version>/kernel/drivers/input/joydev.ko
/lib/modules/<kernel version>/kernel/drivers/input/ff-memless.ko
/lib/modules/<kernel version>/kernel/drivers/input/evdev.ko
/lib/modules/<kernel version>/kernel/drivers/input/joystick/xpad.ko
# on the NAO.
```

Algorithm 15.1: Compiling kernel modules in *NAOqi* virtual machine.

16 NaoDeployer2

Christian Albrecht

This chapter describes the development of the *NaoDeployer* tool which is used by the *NDevils* team of the TU Dortmund in order to deploy their software. In the first Section (Section 16.1), the creation of such a tool and the revision of the first iteration is motivated briefly. Subsequently, the system architecture as well as some related design decisions are explained in Section 16.2. Section 16.3 outlines the basic concept of testing that is used in the project. Finally, an outlook on the future development process is given (Section 16.4).

16.1 Motivation

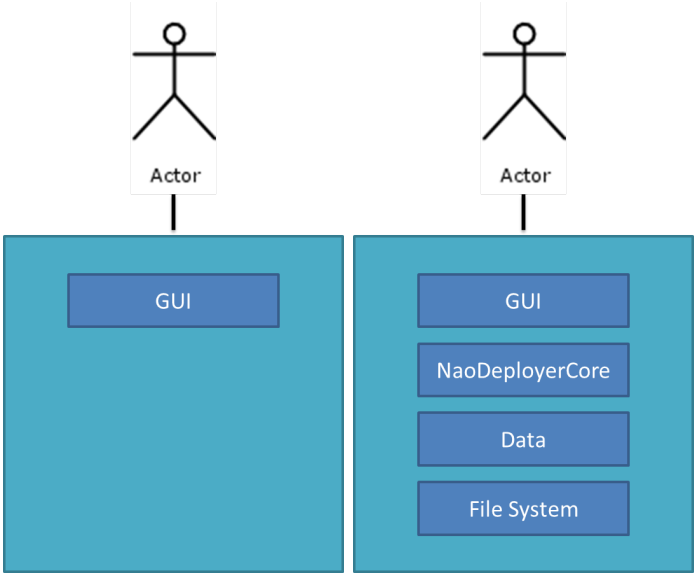
The *NaoDeployer* arose from the urgent need of a reliable tool to transfer the *NDevils* framework comfortably to the *NAO* and to configure the robot. Technically, the second iteration of the *NaoDeployer* is a complete revision of the first one. The only exception is the soccer field. As in the first version it acts as the central GUI (graphical user interface) element which allows the user to assign a player and a team number to a robot. There were many decisive reasons for an extensive revision of the old tool:

1. The old tool supports only a fix number of robots
2. No option to save and load predefined sessions. After closing the tool, everything has to be made from scratch
3. In order to stop the deployment process, the user has to restart the software
4. Porting to other platforms like Linux and MAC is rather questionable because of the dependency to Microsoft's .NET framework
5. Important properties like WiFi or LAN IP addresses of the robots are not displayed in the GUI

16.2 System architecture

The *NaoDeployer2* is implemented in the C++ programming language. Additionally the Qt[1] framework and the QtPropertyBrowser[2] are used to develop a cross-platform application.

[1]Link to the official website: https://www.qt.io/developers
[2]Link to the repository: https://github.com/commontk/QtPropertyBrowser



(a) Architecture of the *NaoDeployer*.

(b) Architecture of the *NaoDeployer2*.

Figure 16.1: Comparison between the system architecture of the *NaoDeployer* and the *NaoDeployer2*.

In order to eliminate the shortcomings mentioned in Section 16.1, the system architecture had to be changed drastically. The principle reason for the changes is the visualization of robot and deployment related properties which is space consuming, especially in a software design which consists only of a user interface like the first iteration of the *NaoDeployer*. To provide the ability to visualize and modify informations in a space-saving fashion an explorer and an object inspector are used. Therefore a data layer had to be introduced in order to separate the data from the user interface. For this purpose the *Robot* and *DeploymentSetting* class have been designed. Their main task is to encapsulate object related information. Because the data contained by both classes should be visualized and altered through instances of the *Explorer* and *PropertyBrowser* classes, they have to inherit from a common base class. The *QObject* class is predestined for this task because it allows the use of the Qt property system which enables access to an object's properties without knowing the exact class interface. From this it follows that only *QObject*s header file has to be included by the *Explorer* and *PropertyBrowser* which simultaneously reduces the project dependencies.

With regard to the implementation of an undo/redo mechanism, the modification of data objects is executed by an instance of the *NaoDeployerCore* class which acts as an controller between the data layer and the graphical user interface. Alternatively the modification could be executed directly by the *PropertyBrowser*. The disadvantage of this design is, that it enforces the use of the *PropertyBrowser* and makes it impossible to exchange it without code duplication. Because the whole state of the *NaoDeployer2* can be loaded and saved from a permanent storage device, the deepest layer of the system architecture is the file system (compare Figure 16.1). All deployment related informations are saved to files with the ending *.ndc* in the xml format. In contrast session overlapping information

Figure 16.2: The MainWindow contains all other graphical control elements.

such as the working directory are stored with the aid of a *QSettings* object in an *.ini*-file at the user scope.

16.2.1 MainWindow

The *MainWindow* is the superordinate GUI element. It contains all other graphical control elements (in the following also called widgets) which are needed to fully configure the deployment and to execute it. Furthermore it is responsible for connecting the signals and slots of the single GUI elements in order to guarantee an error-free cooperation between them. To understand the work flow and the basic tasks of the single widgets a minimal deployment process is described briefly step by step:

1. Add a *Robot* with the *Add robot* button on the *Explorer*-widget
2. Add a *DeploymentSetting* with the *Add setting* button on the *Deployment settings*-widget
3. Assign a position to a robot via the *Soccerfield*-widget
4. Select a *DeploymentSetting* for the *Robot* on the *Deployment*-widget
5. Execute the deployment by pressing the *Deploy* button on the *Deployment*-widget

Besides the described tasks the *MainWindow* offers several services which can be accessed through the menu bar. These services are not directly connected to the preparation or execution of a deployment but they offer the following operations:

- Load and save configuration from/to the file system
- Determine a file which should be loaded at the start
- Changing the path to the *NDevils* framework

16.2.2 NaoDeployerCore

The *NaoDeployerCore* class is the heart of the *NaoDeployer2*. It encapsulates all the basic functionality that this tool offers and manages all the data. Additionally the *NaoDeployerCore* provides projected related informations. Mostly this data can be used to initialize a *QSettings* object and access the data associated with it. Generally the work flow is as follows, the user alters properties or decides to execute a deployment or to start/stop the framework by interacting with the GUI. This emits a signal which notifies the *NaoDeployerCore* who executes the desired task and emits a signal on his part, that some changes are made. Thereupon all widgets listening to this signal can perform an update. Mostly all central actions like adding, removing a *Robot* or *DeploymentSetting* object or altering their properties are encapsulated in classes so called *commands*. These classes inherit directly from QUndoCommand. After their initialization they are pushed on an instance of a *QUndoStack* class object. With this approach the user can step forward and backward through the action history. In order to perform the deployment or to interact with the *NAO* a ssh connection has to be established. The *NaoDeployer2* realizes this communication by calling external bash-scripts.

16.2.3 Explorer and PropertyBrowser

The *Explorer* and the *PropertyBrowser* are two widgets which do only work in cooperation. Essentially the basic task of the *Explorer* is to offer a graphical interface in order to make a selection from a given set of *QObject*s. The *PropertyBrowser*, which is an instance of the *QtProperyBorwser* class placed on a *QWidget*, takes this very object and displays its properties in a table.

16.2.4 Deployment-widget

The responsibilities of the *Deployment-widget* is bipartite. First of all it is responsible for assigning a *DeploymentSetting* to a robot. Moreover it provides the ability to execute scripts in order to interact with the *NAO*. At the moment, this includes starting and stopping the *NDevils* framework as well as *NAOqi*[3] (the operating system installed on the *NAO*) and the execution of the deployment. In order to make this functionality available

[3]Link to the official website: http://doc.aldebaran.com/2-1/naoqi/

in a space-saving way the function as well as the caption of a button changes in its logical counter part after pressing it.

16.3 Testing

In order to guarantee that changes do not effect the software negatively the *NaoDeployer2* comes with a testing environment which allows to implement and execute unit and integration tests. For this purpose two additional project files have been created, the *NaoDeployer2_lib.pro* and the *NaoDeployer2_test.pro* project. The former project is responsible for exporting the single classes used in the *NaoDeployer2* project as a static library. The second project links against this library and thus attains access to the individual classes. In order to implement a test, a new class which inherits from QObject has to be created. According to Qt's test documentation the test logic is implemented in the private slots of the class. The initialization and configurations of the test scenario can either be done in the constructor of the test class or in the slot *initTestCase()*. Similarly, the clean up can be executed in the destructor or the slot *cleanupTestCase()*. The execution of the single tests is activated in the projects main function. This is done by initializing the desired test objects and passing them to the function *QTest::qExec()*. The framework will run all given tests and returns a statistic of the results.

16.4 Outlook

For future releases many features are planned. Most of them are concerned with the surveillance of robot parameters like temperature, running processes, information about the active configuration, charge of battery, connectivity (LAN/WiFi) and primary memory as well as CPU usage. With regard to the coach robot who is currently not allowed to have an active WiFi interface[4], it is desirable to enable or disable this functionality comfortably through the graphical user interface of the *NaoDeployer*.

[4]Soccer Rules 2015, http://www.tzi.de/spl/bin/view/Website/Downloads

17 Behavior View

Karen Bieling

While developing and debugging a behavior for the robots, it is utile to have a graphic representation of the behavior. Therefore two different visualizations has been implemented, one in the simulator which can be used for debugging issues, and one offline visualizer to develop a behavior.

17.1 Behavior View in the Simulator

To show the current state of the behavior the behavior view was extended, such that not only the name of the current behavior, but also a graphical view of the current option (as a state-machine), highlighting the current state, the state which the robot was in before and the path it took. A robot can be in several options, that all should be shown simultaneously. To get a clearer view, the options where rearranged so it can be switched between the options at the highest level with tabs. Figure 17.1 shows such a layout:

In the top (1) you can chose the top level of the options. The name of the current option (2) and a history of the last 5 states (3) are shown below. The current state is also highlighted in the state machine (dark-read, 4), the former in light-read (6). To avoid too large diagrams, only the conditions of the transitions which triggered the current state (5) and those which could trigger the next one are displayed.

Further information that are shown are the parameter names of an option and marking a target-state with a doubled border.

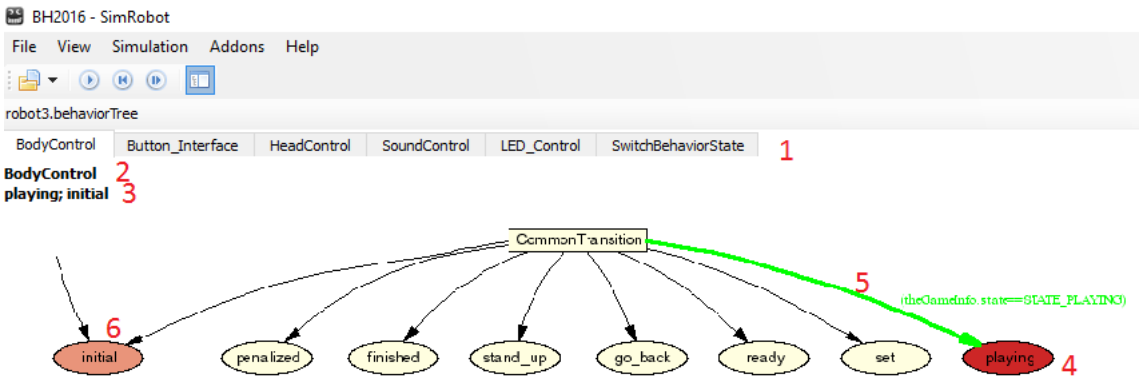


Figure 17.1: Graphical representation of the option *BodyControl* in the simulator

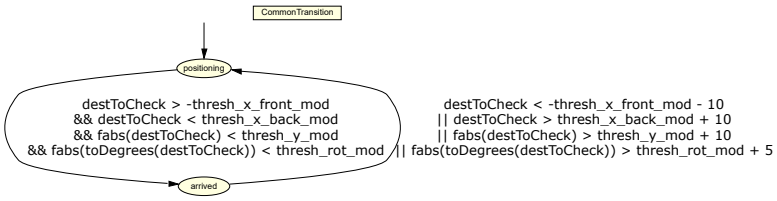


Figure 17.2: Graphical representation of the option *GoToFieldCoordinates* created by the offline *Behavior Visualizer*

Implementation

To get the information that is displayed in the view the file `cabsl.h` was extended, such that it sends an extended debug-message. It now also includes the type of a state (initial, target etc.), the parameters of an option, and the history of the states. Not all information can be managed by the *CABSL-Macros*, therefore parts of the options are parsed to find out which conditions of transitions trigger which state. This is part of the class `BehaviorInfo`. The third part which is changed is the view itself. It is based on the current *Behavior View* to keep the tree structure. The graphics are created in the *graphviz*-format.

17.2 Offline Behavior Visualizer

Sometimes it might be necessary to analyze the resulting state-machine of a developed behavior without running the simulator. Therefore an offline *Behavior Visualizer* has been implemented. It creates a state-machine from the *CABSL-Code*. A resulting state-machine of the option *GoToFieldCoordinates* is shown in Figure 17.2.

17.2.1 GUI

Figure 17.3 shows the GUI of the visualizer. In *Source* the *CABSL-File* which shall be visualized is chosen, either by using the *Browse...*-Button or by directly pulling the file via drag and drop into the textfield.

Destination is a folder where the output-picture is saved. Selecting *Create* will generate a *dot*-file - if the syntax of your *CABSL-file* is correct. Otherwise the error (including file and linenumber) is shown in the status-box below.

When selected *Open Destination Folder* the folder where you can find your created image is opened. The image is in the *png*-format, its name is the same as your option.

17.2.2 Implementation

The Behavior Visualizer was created as an Xtext-Project, a java-based Eclipse-Plugin which you can use to create a *Domain Specific Language (DSL)*. It also offers the oppor-

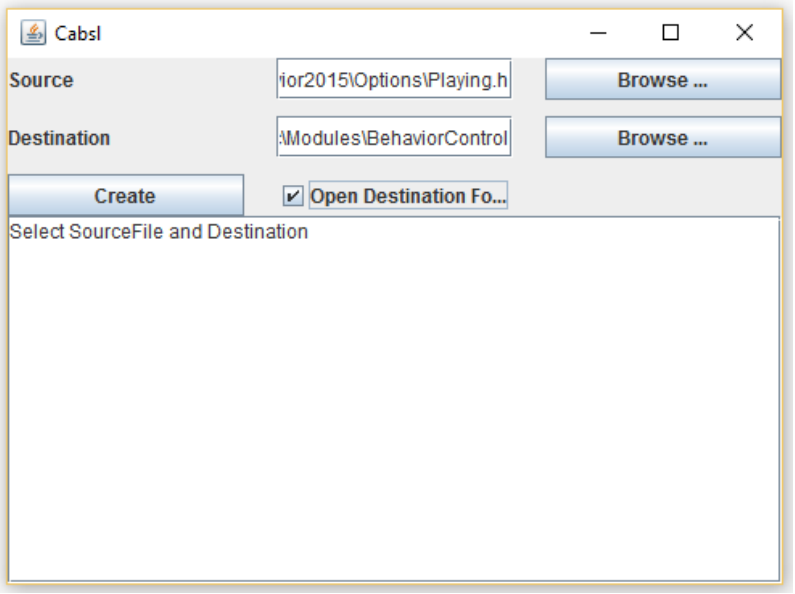


Figure 17.3: GUI of the Behavior Visualizer.

tunity to implement a generator to generate new code from source-code written in your defined language. Here this opportunity is used to create the *dot*-file from which the final picture of the state-machine is derived.

It is possible to create a CABSL-Editor as an Eclipse-Plugin which offers Syntax-Highlighting, an Outline and Auto-Completion for easy development of new CABSL-Options.

18 Conclusion

In this report we have shown the development of the robot's framework regarding several aspects. A rough categorization for different parts of our work is

1. Development for the coaching robot
2. General missing features or improvements to the existing framework
3. Necessary new features for *RoboCup* 2015

The results are summarized in the following sections.

18.1 Development of the Coaching Robot

As mentioned in Chapter 1.1, the main task of this project group is to develop a coaching robot. This is why most of the approaches presented in this report are concerned with the coaching robot. Since the coaching robot is only allowed to send a message every 10 seconds, runtime of the used algorithms is not as important as for a field player. As a result we decided on multiple separate approaches for different tasks, instead of one algorithm that solves similar tasks at once, i. e. multiple vision systems for field lines, robot detection etc. Furthermore we decided to use existing methods and algorithms as a base, instead of solely heuristic approaches (which are commonly used to fulfill the requirements regarding run time).

A common approach to detect objects is to filter the image first with a certain mask, that removes noise and/or preserves important features, e. g. edges. Since the computation of an edge image is time consuming, we developed a faster solution using *Streaming SIMD Extensions*. This allows the calculation of a full-sized edge image in approximately 11 milliseconds. This edge image is particularly useful for detecting field lines, thus we decided to use it in our field line detection as input for an algorithm based on the *Hough Transform*. The detection of the lines itself indicates a satisfying result as does the merging and the classification of those lines. One goal of this line detection procedure was to use it as a basis for the image to field coordinates transformation. Due to limitations of the field of view of the camera and the position of the coach this approach was not capable of providing four point-to-point correspondences between image points and points in the soccer field model consistently. As the calculation of a *homography* requires four point correspondences (of which three points are not collinear, see Chapter 9) we had to think of another method. In the end we decided to measure the coach's position and use existing matrices to calculate the transformation from image to field coordinates. The results of this approach were satisfactory to decide on which side of the field the ball is. In fact this transformation

gives a result with only a small error, but since the coach is not allowed to communicate any kind of numbers the exact position of object is only important in internal calculations.

Moreover, the coaching robot needs to detect the field borders. Since this is heavily dependent on the field color detection we first improved the existing field color detection. This was mainly done by converting the existing algorithm's color space into *HSI*-color space. After that, a grid search was performed to remove outliers and compute the correct outline of the field. Using a *Graham Scan* the convex hull of the field is then calculated. Most of our approaches use the field-detection to rule out pixels that are not inside the soccer field to enhance performance of the used algorithms. One of these methods is the robot detection based on colors. We are able to detect robots and their team membership with this method. A *DBSCAN* clustering is performed to distinguish between the robots. Furthermore a second algorithm using the *FAST* feature detector was developed, which does not directly depend on the field-detection. In contrast to the color based approach this uses *k-means* as a clustering algorithm. Both strategies are capable of detecting the robots, but the *FAST* based approach has issues distinguishing between the robots if they are too close together. The various algorithms in our line detection pipeline, i.e. the operator and the linear hough transform, also benefit heavily from using the field detection in terms of performance.

The updated *Standard Platform League* rules for 2016 also feature a new, white ball, which can not be recognized by a color based approach as used before. This is why we decided to use a hybrid approach using a combination of clustering and hough transform methods. With this algorithm we are able to detect the ball based on the circular shape. Problems show if the ball is very close to a robot or lies on a field line crossing. The runtime is about 100ms, which is sufficient for the coaching robot.

Equally important was the implementation of a message system. This includes basic text messaging via *UDP* to the *GameController* and grouping of stored messages. The message system implements the standardized package type *SPLCoachMessage*, which is defined by the SPL rules.

A new behavior for the coaching robotwas developed, in which the robot will search the ball at all times. If it finds the ball, it will track the ball (with his head only, since other movements are not allowed) until it is lost again. This way, we can communicate an approximate position (numbers are not allowed in coach messages) of the ball to the other players in our team. To decide where the ball is, apart from detecting the actual ball, the coach camera to field coordinates transformation mentioned in Chapter 9.3 is used.

18.2 Improvements to the Existing Framework

One major issue is our current localization, which is neither well documented nor fully functional. Hence a complete rewriting was the best decision from our point of view. Additionally, we replaced the current approach, which is based on multiple *Kalman Filters*, by an approach using a *Particle Filter*.

Another important step was to create a new or adopt a kick engine from another soccer team. We successfully ported *B-Human's kick engine* as well as the *KickView* module for the simulator. These changes were made prior to *RoboCup* 2015, in which we already used the ported engine. Although our behavior still needs a few refinements to better decide when to use which kick, the long distance kick we created using B-Human's kick engine tremendously helped in the technical challenges as well as in critical situations.

We also wanted to improve the robot's behavior in situations, in which the robot is likely to fall down. The idea was to detect if a robot is falling down at a point in time where a full stop would prevent the robot from falling. Hence we implemented an *Artificial Neural Network* to classify the sensor data of the robot as "likely to fall down" or "unlikely to fall down". This is still a work in progress, since this approach needs a lot of test data and further testing on different robots. Nevertheless, the experimental results showed that the Neural Network approach is capable of accomplishing this task.

The Any Ball Challenge as well as the rule changes considering the ball demanded the development of a new ball model. Hence, we implemented a *Multi Hypothesis Ball Model*, which is superior to the current ball model considering both jumping ball perceptions and a more reliable velocity prediction.

Our current framework already features a variety of tools, nevertheless we wanted to update some of the older tools and add new tools that enhance the workflow of our framework.

Since our current deployment tool, the *NaoDeployer*, was lacking some features and was written in *C#* (only compiled for Windows), we decided to rewrite this tool completely. The *NaoDeployer2* (see Chapter 16) is based on *Qt* and brings features like session saving, cross-platform compatibility and a cleaner interface than the current *NaoDeployer*.

We furthermore wanted to enhance the visualization possibilities for *CABSL*-Code. This is why the behavior view of the framework's simulator was extended by a graphical view of a state-machine representing the current option, which is updated live when connected to a real robot. The behavior view can also be created without having a real robot by analyzing the code with our newly created tool.

In addition we started working on a ground truth system. The software already features calibration of cameras, color detection and image to field coordinate transformations tested with a *Logitech USB Webcam*.

18.3 New Features for RoboCup 2015

Since there are usually different technical challenges at the RoboCup, some alterations have to be done to compete in those challenges. At *RoboCup* 2015 these challenges were: The *Many Carpets Challenge*, the *Any Ball Challenge* and the *Corner Kick Challenge*. Most of the alternations only had to be done for the *Many Carpets Challenge* and *Corner Kick Challenge*, since our ball detection is not based on colors. These changes were mainly achieved with a modified behavior or a completely new behavior respectively. Our existing walk engine only needed a few adjusted parameters to make the robot move on the different surfaced carpets in the *Many Carpets Challenge*. In every challenge our robots were able

to a least touch the ball and we scored the first place in the *Any Ball Challenge*. Overall these modifications were working as intended and finally we were able to finish in second place.

Likewise a new behavior was successfully created for the penalty kick shootout. Unfortunately, our team did not compete in a penalty kick shootout at *RoboCup* 2015. Hence, there are no tests results of this behavior in a real competition.

Bibliography

[1] Ethem Alpaydin. *Introduction to Machine Learning*. 2nd ed. Cambridge, Massachusetts: The MIT Press, 2010.

[2] David Arthur and Sergei Vassilvitskii. "K-Means++: The Advantages of Careful Seeding". In: *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '07. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[3] Kai Uwe Barthel. *3D Color Inspector*. Sept. 12, 2015. URL: http://rsb.info.nih.gov/ij/plugins/color-inspector.html (visited on 09/12/2015).

[4] Wilhelm Burger and Mark James Burge. *Digitale Bildverarbeitung. Eine algorithmische Einführung mit Java*. German. Springer Berlin Heidelberg, 2015.

[5] *Clang Mainline Doxygen Documentation*. Sept. 4, 2015. URL: http://clang.llvm.org/doxygen/index.html (visited on 09/04/2015).

[6] Samuel Kelly Clark and Richard N Dodge. *A Handbook for the Rolling Resistance of Pneumatic Tires*. Ann Arbor: The University of Michigan, 1979.

[7] RoboCup Technical Committee. *RoboCup Standard Platform League (NAO) Rule Book*. 2015. URL: http://www.informatik.uni-bremen.de/spl/pub/Website/Downloads/Rules2015.pdf.

[8] RoboCup Technical Committee. *RoboCup Standard Platform League (NAO) Technical Challenges*. 2015. URL: http://www.tzi.de/spl/pub/Website/Downloads/Challenges2015.pdf.

[9] Thomas H. Cormen et al. *Introduction to Algorithms*. The MIT Press, 2001.

[10] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Kdd*. Vol. 96. 34. 1996, pp. 226–231.

[11] Grant Griffin. *Magnitude Estimator*. Oct. 2, 1999. URL: http://www.dspguru.com/dsp/tricks/magnitude-estimator (visited on 09/03/2015).

[12] Matthias Hofmann and Florian Gürster. "GOL-A Language to Define Tactics in Robot Soccer". In: *Proceedings of the 10th Workshop on Humanoid Soccer Robots, in conjunction with the IEEE-RAS International Conference on Humanoid Robots*. 2015.

[13] Ming-Kuei Hu. "Visual Pattern Recognition by Moment Invariants". In: *IRE Transactions on Information Theory* (1962), pp. 179–187.

[14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*. Intel. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf (visited on 09/03/2015).

[15] Intel. *Intel Intrinsics Guide*. Sept. 3, 2015. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (visited on 09/03/2015).

[16] Illingworth J. and Kittler J. "A Survey of the Hough Transform". In: *Computer Vision, Graphics, and Image Processing*. Vol. 44. 1988, pp. 87–116.

[17] Ahmad Kaifi. *Multi Object Tracking Based on Color*. URL: http://akaifi.github.io/MultiObjectTrackingBasedOnColor/.

[18] Hans Werner Lang. *Algorithmen in Java*. Oldenbourg Verlag, 2006.

[19] Wei-keng Liao, Ying Liu, and Alok Choudhary. "A Grid-Based Clustering Algorithm Using Adaptive Mesh Refinement". In: *7th Workshop on Mining Scientific and Engineering Datasets of SIAM International Conference on Data Mining*. 2004, pp. 61–69.

[20] Jörg Liesen and Volker Mehrmann. *Lineare Algebra. Ein Lehrbuch über die Theorie mit Blick auf die Praxis*. German. 2nd ed. Wiesbaden: Springer Fachmedien, 2015.

[21] Stuart P. Lloyd. "Least Squares Quantization in PCM". In: *IEEE Transaction on Information Theory* (1982).

[22] D Luo. *Pattern Recognition and Image Processing*. Amsterdam: Elsevier, 1998.

[23] J. Matas, C. Galambos, and J.V. Kittler. "Robust Detection of Lines Using the Progressive Probabilistic Hough Transform". In: *Computer Vision and Image Understanding*. Vol. 78. 2000, pp. 119–137.

[24] Tom M. Mitchel. *Machine Learning*. New York: McGraw-Hill, 1997.

[25] Tim Niemüller et al. "Providing Ground-Truth Data for the NAO Robot Platform". In: *RoboCup 2010: Robot Soccer World Cup XIV*. Springer, 2011, pp. 133–144.

[26] Andrea Pennisi et al. "Ground Truth Acquisition of Humanoid Soccer Robot Behaviour". In: *RoboCup 2013: Robot World Cup XVII*. Springer, 2013, pp. 560–567.

[27] Ian Reid. *Lecture Notes in Computational Geometry*. 2003. URL: http://www.robots.ox.ac.uk/~ian/Teaching/CompGeom/.

[28] Aldebaran Robotics. *NAO Camera*. Sept. 10, 2015. URL: http://doc.aldebaran.com/2-1/family/robots/video_robot.html (visited on 09/10/2015).

[29] Aldebaran Robotics. *NAO Robot*. Sept. 7, 2015. URL: https://www.aldebaran.com/en/humanoid-robot/nao-robot (visited on 09/07/2015).

[30] Edward Rosten and Tom Drummond. "Machine Learning for High-Speed Corner Detection". In: *European Conference on Computer Vision*. 2006.

[31] Edward Rosten, Reid Porter, and Tom Drummond. "Faster and better: A Machine Learning Approach to Corner Detection". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* (2009).

[32] Andreas Strack, Alexander Ferrein, and Gerhard Lakemeyer. "Laser-Based Localization with Sparse Landmarks". In: *RoboCup 2005: Robot Soccer World Cup IX*. Ed. by Ansgar Bredenfeld et al. Vol. 4020. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 569–576.

[33] Gary Sullivan and Stephen Estrop. *Recommended 8-Bit YUV Formats for Video Rendering*. Microsoft. 2008. URL: https://msdn.microsoft.com/de-de/library/windows/desktop/dd206750(v=vs.85).aspx (visited on 03/11/2016).

[34] *Summary of Major Rule Changes for 2016*. URL: http://www.tzi.de/spl/bin/view/Website/MajorRule2016 (visited on 03/17/2015).

[35] Reinhardt Thomas. "Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball". PhD thesis. Hochschule für Technik, Wirtschaft und Kultur Leipzig, 2011.

[36] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. Cambridge, Massachusetts: The MIT Press, 2005.

[37] Wapcaplet. *HSI Cylinder*. Sept. 12, 2015. URL: https://de.wikipedia.org/wiki/HSV-Farbraum#/media/File:HSV_cylinder.jpg (visited on 09/12/2015).