Team Report and Code Release 2021

Thomas Röfer[1,2], Tim Laue[2],
Nikolai Bahr[2], Jonah Jaeger[2], Jannes Knychalla[2],
Thorben Lorenzen[2], Nele Matschull[2], Yannik Meinken[2],
Lukas Malte Monnerjahn[2], Lukas Plecher[2], Philip Reichenberg[2]

[1] Deutsches Forschungszentrum für Künstliche Intelligenz,
Cyber-Physical Systems, Bremen, Germany

[2] Universität Bremen,
Fachbereich 3 – Mathematik und Informatik, Bremen, Germany

Revision: December 28, 2021

Contents

Chapter 1

Introduction

1.1 About Us

B-Human is a joint RoboCup team of the Universität Bremen and the German Research Center for Artificial Intelligence (DFKI). The team was founded in 2006 as a team in the Humanoid League, but switched to participating in the Standard Platform League in 2009. Since then, B-Human has won eleven European competitions and has become RoboCup world champion eight times. The 2021 team consisted of the following persons:

Students: Nikolai Bahr, Jonah Jaeger, Jannes Knychalla, Thorben Lorenzen, Nele Matschull, Yannik Meinken, Lukas Plecher

Active Alumni: Andreas Baude, Lukas Malte Monnerjahn, Philip Reichenberg

Leaders: Tim Laue, Thomas Röfer

Associated Researchers: Udo Frese, Arne Hasselbring

1.2 About the Document

This document provides a survey of this year's code release[1], continuing the tradition of annual releases that was started several years ago. The description is now split into two parts. The

[1] `https://github.com/bhuman/BHumanCodeRelease/tree/coderelease2021`

more technical information can be found in our Wiki[2]. This document is more focused on the new approaches we introduced during the past two years and our specific solutions to the challenges posed by RoboCup 2021. It is not a description of the complete system. Topics that are neither described here nor in the Wiki can probably be found in the description of our 2019 code release [10].

In the last year, we participated in two competitive events. The German Open Replacement Event (GORE)[3] and the RoboCup 2021. In lack of the official RoboCup German Open in 2021, the GORE was organized by the Standard Platform League community. It was held at the Technical University of Dortmund and the University of Bremen. Due to the pandemic, no members of the other teams went to these locations, but instead mailed some of their robots to establish robot pools, so that full 5 vs 5 games could be played at both sites. Two hours before a game, each team was assigned random robots from a pool and had to remotely set up and calibrate them. Because it was not possible to play many games, a modified version of the Swiss-System Tournament was used. After 3 games with a total of 28:0 goals, we won this competition. A detailed description of the GORE is given in [7]. In 2021, the RoboCup was distributed all over the world. There were no soccer games of 5 vs 5 robots, but four different challenges. Furthermore, in two of these challenges the robots had to be deployed remotely without any connection to the robot. By winning three of those challenges and getting to the final on the last one, our team also won the RoboCup 2021.

In addition to these competitions, we also published multiple research papers at the RoboCup Symposium 2021. Our new approach for detecting the field border by using a deep neural network has been developed by Hasselbring and Baude [3]. New approaches for making our walking motions faster and more stable are presented by Reichenberg and Röfer [9]. A framework for learning a semantic segmentation on augmented simulated images is described by Blumenkamp, Baude and Laue [1]. It is worth mentioning that the two latter publications were both nominated for the RoboCup Symposium Best Paper Award.

The major changes made to our software since the last code release in 2019 are shortly enumerated in Section 1.3. In the following chapters, the four challenges of RoboCup 2021 are described with each chapter consisting of the task, the approach, and the result. The first one is about the Obstacle Avoidance Challenge, in which a robot has to score a goal by dribbling between obstacle robots in as little time as possible. In Chapter 3, we describe the Passing Challenge, in which two robots have to pass a ball back and forth between two obstacle robots. The next part is Chapter 4, which is about the 1vs1 Challenge, where two robots compete against each other in kicking balls into the opponent's half or goal. The last challenge in Chapter 5 is the Autonomous Calibration Challenge. It contains multiple calibration and localization tasks. Following these descriptions, there are three more chapters, which provide in-depth information of the aforementioned system improvements.

1.3 Major Changes Since 2019

The major changes made since RoboCup 2019 are described in the following sections:

6 Perception (Vision)
Changes made to the image processing pipeline.

[2] https://wiki.b-human.de/coderelease2021
[3] https://gore2021.netlify.app/

6.1 Field Boundary Detection Using Convolutional Neural Networks
We detect the field boundary using a CNN, thus making this processing step independent from calibration.

6.2 Lighting Independent Field Line Detection
We built a field line perception that doesn't need manual calibration and adapts to lighting changes.

7.1 Orientation Stabilization
The soles of the feet are used for stabilization of the orientation estimation.

7.2 Arm Contact Detection
We changed our arm contact detection from using the shoulder joints to using the hand positions in the robot coordinate system.

7.3 Foot Support Switch
We added a minimum pressure that the new support foot must have at a support foot switch to prevent a false detection. Furthermore, the minimum weights are automatically calibrated.

8 Motion Infrastructure
A new infrastructure for everything related to robot motion.

8.3 Walking
A new step adjustment approach and support foot sole rotation compensation to achieve higher stability.

8.3.4 Walk Kicks
A new module to generate kicks, which adjust themselves to the ball position.

8.3.7 Treading on a Robot's Foot
We detect walking on another robot's foot and act accordingly.

8.4 Fall Motions
New fall motions to fall more safely.

8.5 Keyframe Motions
We combined the two modules for getting up and executing so-called "special actions" into a single one.

Chapter 2

Obstacle Avoidance Challenge

2.1 Task

The general task of this challenge is to move the ball as quickly as possible into the opposing goal. The ball lays in the middle of the field, while the playing robot is placed on the center circle of its own half, facing the ball. Between the ball and the goal there are four obstacles in the form of turned off robots. There are three types of obstacles: One robot in the center, one slightly offset and two robots with space in-between. The robot that is slightly offset can either be offset to the left or to the right. The first obstacle stands on the middle circle opposite of the playing robot. The next obstacle is in front of the penalty area and the final obstacle on the goal area line. Each type of obstacle stands on one of the positions. Therefore there are twelve different possible obstacle configurations (three types on three positions times two because the offset robot can be either left or right), an example is depicted in Figure 2.1.

The playing robot has to bring the ball to a position that is at most 1.3 meters away from the opponent ground line before it is allowed to score a goal. If it scores a goal from another position, the ball is moved back in the center of the field. The same happens if the ball leaves the field.

If the robot touches one of the obstacles, 10 seconds are added to the timer. If the ball touches an obstacle, 5 seconds are added.

The final score is the median of three attempts. For each attempt, a new obstacle configuration is randomly chosen.

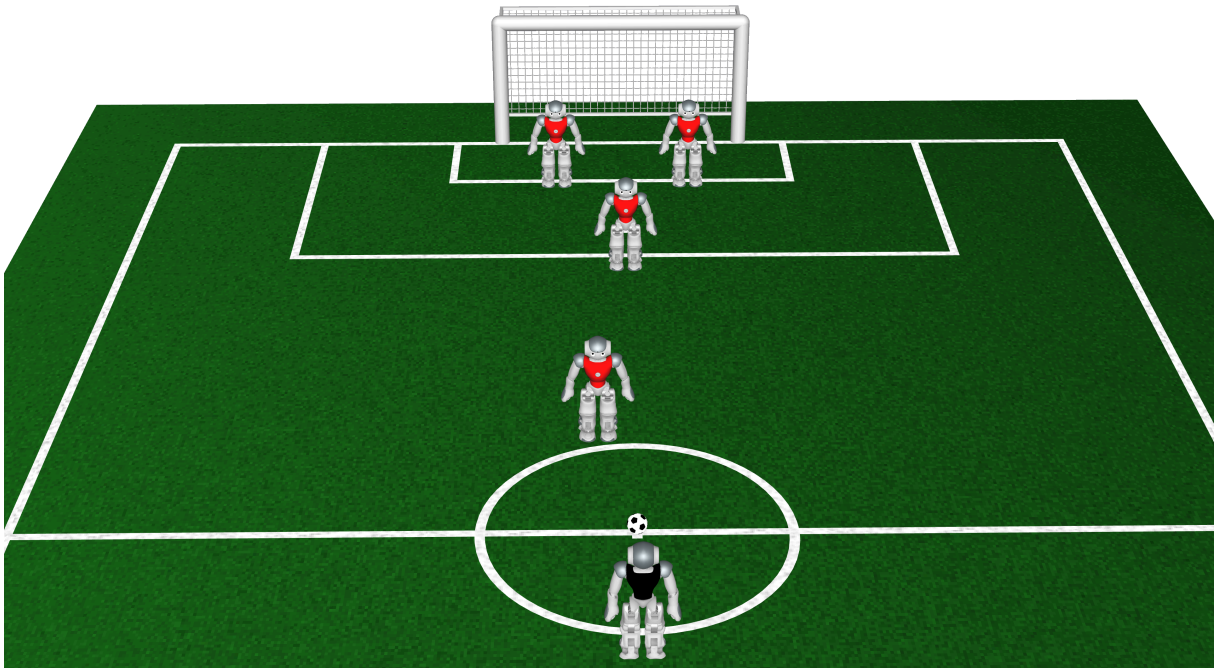


Figure 2.1: One possible obstacle configuration

2.2 Approach

In the following section we explain how we model the obstacles and plan a path which avoids them and let us score a goal consistently, fast. We describe which existing modules we could use and what actions we have taken to adapt to the challenge.

2.2.1 Modeling

To avoid the obstacles, we first have to recognize and model them. For these tasks, we already had existing and reliably working modules. Due to the specific rules of this challenge, we were able to change the modeling so that it uses additional domain-specific information. Since the obstacles do not move, we can assume that all obstacles, which are in our model but are not in our field of sight, or should be otherwise sensed, remain unchanged. According to the rules, there are only 15 positions on which an obstacle could possibly stand. Given that our self-localization is accurate enough, we can use this fact to ignore obstacles which are sensed not within a threshold distance to any of those.

Because there are only 12 possible obstacle setups, we are able to examine which of them fit to our obstacle model and therefore predict or exclude obstacles on positions that we not have seen yet. To do so, we first sort our obstacles into one of the three obstacle stages. Then we list all 6 (at this point we handle the left or right offset obstacle as the same) possible permutations. We compute a cost for each stage based on how good the sensed obstacles in this stage align with the ones of the permutation. Then we can find the permutation with the lowest overall cost. We take the stages of these permutations that cost less than a threshold for granted and remove the according obstacle type from the possibilities for the other stages as well. In this way, we can clearly specify the stages that we have seen well and for the others we only exclude the obstacle types seen elsewhere.

2.2.2 Behavior

For consistent results, we decided to walk along a path on the side around all obstacles (see Figure 2.2). Early tests showed that the penalty for touching one of the obstacles is enough for changing a good try to a bad one. Thus, we chose a safe path in the middle of the outmost obstacle and the field border. The path starts at the middle point and goes from there next to the first obstacle in the middle between the outmost possible obstacle and the field line. The other points that define the path are also on this position but next to the second/third obstacle. The last point is in the middle of the goal, but it is not used because shooting at the goal is faster. For this reason, we also reduce the kick strength when being near the 1.3 meter mark so that we do not overshoot it massively and waste time walking to the ball.

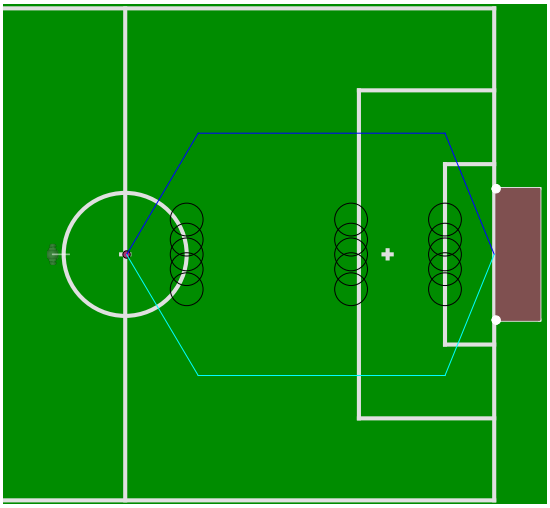


Figure 2.2: Left and right path. The circles indicate possible positions for obstacles

When the ball is close enough to the ground line so that a goal would be legal and there is an angle in which we can safely hit the goal without touching an obstacle, we shoot at the goal.

To increase the speed a bit, we execute a longer initial kick if the ball is near the center point and we already know the type of the first obstacle. This kick is longer than the normal dribble kicks so that we can walk a longer distance at higher speed afterwards because we do not need to be careful to not accidentally touch the ball. If the first obstacle are the two robots, we kick the ball through the middle, otherwise we play left or right past it.

2.3 Results

2.3.1 Attempts

In our first attempt, the robot dribbled sideways past the goal after 32 seconds so that the ball was returned to the starting point. The robot had to return and try again. This time, the robot was able to score a legal goal after a total of 89 seconds. At no time, the robot or ball came near to an obstacle. Thus, 89 seconds was our first official result.

In the second attempt, the initial kick was a bit close to the first obstacle but did not touch it. But the robot could not walk fast to the ball because it had to maneuver around the obstacle. After that, the ball was played back at the path and then over the 1.3 m line and from there it was kicked into the goal. This attempt took a total of 32 seconds.

	1. Run	2. Run	3. Run	Combined (Median)	Overall Rank
B-Human	1:29	0:32	0:27	0:32	1
HULKs	0:35	0:36	3:41	0:36	2
Berlin United – NaoTH	0:45	0:55	–	0:55	3
Nao Devils	1:30	0:42	1:00	1:00	4
UT Austin Villa	0:48	1:04	–	1:04	5
HTWK Robots	1:32	1:54	0:54	1:32	6
Bembelbots	2:42	4:06	–	4:06	7
SPQR Team	–	–	1:13	–	8
R-ZWEI KICKERS	–	–	1:59	–	9
Dutch Nao Team	–	–	–	–	10
Naova	–	–	–	–	10
NomadZ	–	–	–	–	10
rUNSWift	–	–	–	–	10
Starkit	–	–	–	–	10

Table 2.1: Ranking of the Obstacle Avoidance Challenge

In our last attempt, the initial kick placed the ball closer to the desired path and the robot was able to quickly go after it so that this attempt only lasted 27 seconds.

2.3.2 Comparison to Other Teams

Our second and third attempt were the two best of all official attempts during RoboCup 2021 and therefore our median was also the best, making us the winner of the Obstacle Avoidance Challenge. The detailed results are shown in Table 2.1.

2.3.3 Possible Improvements

At our first attempt, the problem was that the robot dribbled past the goal and had to return to the middle point for a second chance. We think that this happened because we did not implement a correct dribble behavior for the area past the last obstacle. In this case, the robot should shoot at the goal, if the angle is right but this was not the case here and the robot just dribbled forward. Ideally, in this case, we should dribble to a proper position.

Chapter 3

Passing Challenge

3.1 Task

In this local challenge, two NAOs have to complete as many passes as possible in one of three attempts. Between these attacker robots, two static obstacle robots are placed. The position of each NAO is selected randomly with three possible positions for each attacker and three possible positions for both obstacles, as shown in Figure 3.1. One attempt is over when

- five minutes are over,
- the ball hits an obstacle or
- an attacker robot leaves its red zone for more than 30 seconds.

If the ball leaves a red zone towards outside the field, the ball is set to the position where the ball went out. However, if the ball leaves the field or stops moving in the blue zone, it is set to the outermost attacker starting position (either A or I) of the red zone where the ball was initially kicked from. A referee keeps track of the challenge via a video stream. The highest amount of successful passes in one of the attempts is counted to determine the final score.

3.2 Approach

This section is about the implementation of the challenge. Firstly, our main strategy is stated. Secondly, the obstacle model is explained, and thirdly, some parts of the behavior are described in more detail.

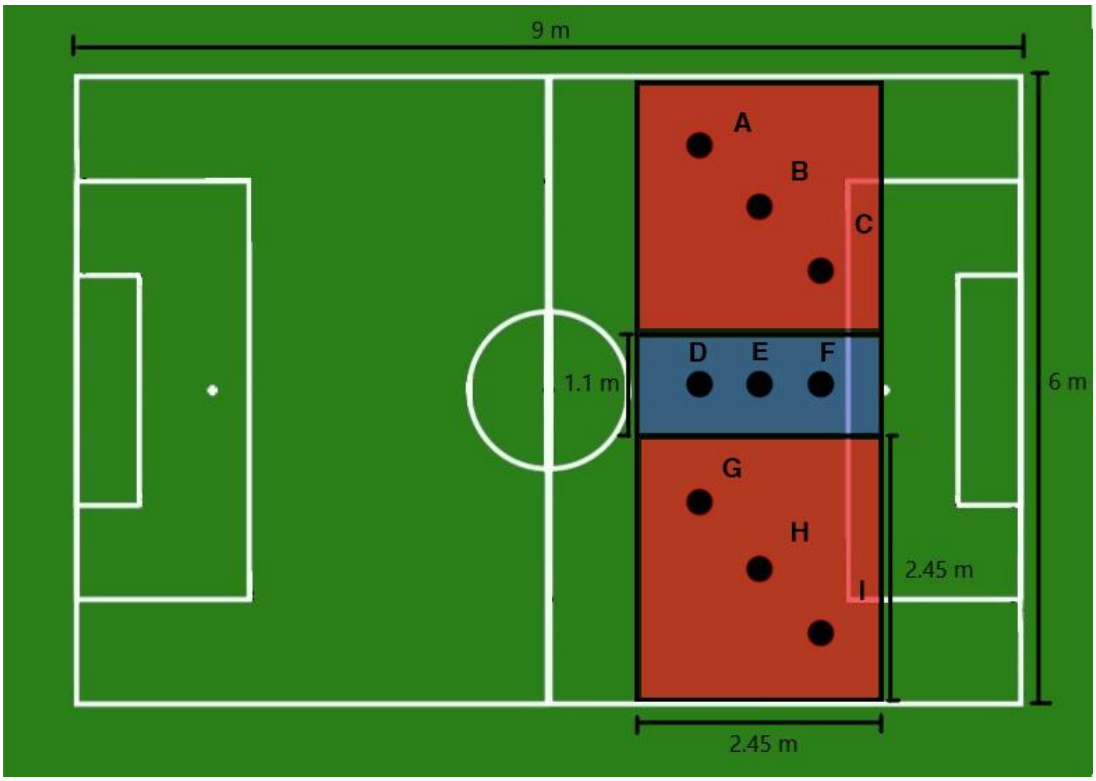


Figure 3.1: The layout of the field. The red zones define the regions where the attackers are playing the passes. The blue zone shows the zone of the obstacles which must not be used to execute valid passes. The points A, B, C, G, H and I mark the possible starting positions of the attackers, the points D, E and F show the possible obstacle positions (see [2]
).

3.2.1 Strategy

As an attempt ends when the ball touches an obstacle, the passes should be done as low-risk as possible. The best way to do so is to always use the free corridor and pass parallel to the long side of the challenge's field. Figure 3.2 illustrates that the angle range of the gap to pass through gets smaller if the passing robot is not in the free corridor. The bigger the angle range to pass through, the lower the risk of accidentally hitting an obstacle.

Therefore, the robots have to move into the area of the free corridor to get a bigger angle range. Once both robots are in the free corridor, the idea is that they only need to pass the ball back and forth.

3.2.2 Obstacle Model

The recognition of the free position in the obstacle area is fundamental to successfully run an attempt of the passing challenge. From the challenge's rules we know the coordinates of the three possible obstacle positions. By calculating the distances of the estimated positions of recognized obstacles to the three fixed positions from the rules, the fixed position with the highest minimal distance to any of the recognized obstacles is considered as the free position.

It can occur that more than one fixed position has the highest minimal distance to any obstacle higher than a specific threshold (e.g. 40 cm). This indicates either a position estimation diverging too far from the real position of the obstacles or a recognition of only one obstacle. In both

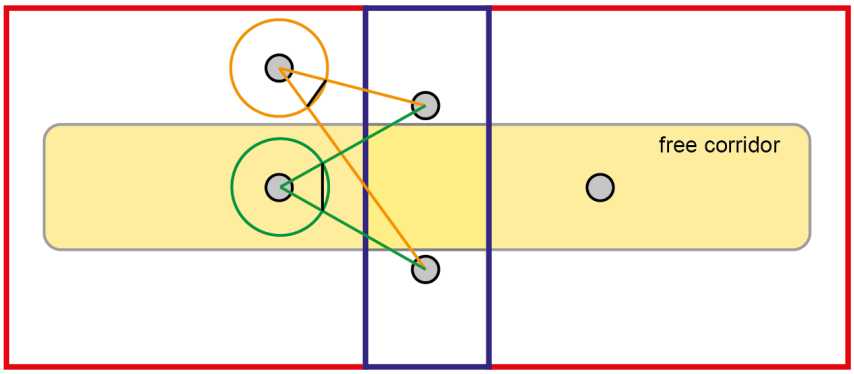


Figure 3.2: Angle ranges of two exemplary passing positions. The green one is in the free corridor, the orange one is not.

cases, such information is not precise enough, so the model is not getting updated.

As the accuracy of obstacle estimation is not perfect and positions can vary from frame to frame in some centimeters, our passing challenge obstacle model needs to get stabilized. By using a discount factor, the obstacle model takes the information of the frames of the last seconds into account. Furthermore, the obstacle model is constantly validated in order to prevent a false stable estimation of the free position.

3.2.3 Behavior

In this section, the behavior of the robots is described. The role assignment is explained in 3.2.3.1, followed by the behavior of one robot in specific situations.

3.2.3.1 Team Behavior

The player number is hard-coded to one side of the field to keep the code and the behavior less complex. This can be done without consequences, since both deployed NAOs won't need to leave their side of the field throughout the whole challenge. We decided to use the robot numbers 2 and 3. Robot 2 is on the right side of the field, in negative y-coordinates, in the zone of starting positions A, B and C. Robot 3 is on the opposing side of the field in positive y-coordinates. The two passing robots use a particular role assignment to ensure that in every situation only one robot performs a pass and the other one prepares to catch it. To achieve this, the role assignment is very straightforward. A robot considers itself a *passBallPlayer* if the ball moves to a position that is in the own half and has at least 20 cm distance to the center of the field. If this is not the case, the robot will become a *receiveBallPlayer*. The 20 cm threshold is used to lower the risk of falsely assigning passBallPlayer to the wrong or to both robots. This has the side effect that both robots will be receiveBallPlayers when the ball is expected to stop in the 40 cm wide area in the center of the field. This situation is acceptable and desirable since the ball will be reset if it stops moving there as described in section 3.1. The overall behavior in the challenge execution majorly depends on this role assignment.

3.2.3.2 Observing

Every attempt starts with observing the environment to help the robots locate themselves. This is done by turning the head left and right and perceiving the field.

In this challenge, the ball is placed in front of one of the robots. Since the robot is selected randomly, the robots look down in front of them for just a moment as soon as they are finished looking left and right, so they get a chance to detect the ball. This way, the attackers know from the beginning which role they start in and how to make the first walking movements.

While not moving, the image quality is better, which leads to a better accuracy in the obstacle estimation. Therefore, the robots get some seconds at the beginning before moving to get a stable evaluation of the obstacles. Each of those positions needs to be observed carefully to ensure a correct obstacle model. This is done by moving the attackers heads back and forth between the angles that point towards the possible obstacle positions. The sequence of the observed obstacle positions is D-E-F-E-D-E-F, meaning that they move their head from the first to the last position twice. To improve the observing conditions even further, at each of those positions the head stops moving for a short time.

Whenever the obstacle model is not stable or not valid during a started attempt, the robot stops moving and gets a new evaluation of the free position by moving to the middle attacker position, turning towards the obstacles and looking left and right. In practice, this can happen due to issues with self-localization and image recognition, for instance caused by challenging lighting conditions, but occurs rather rarely.

3.2.3.3 Move To Passing Position

As mentioned in 3.2.1, the robots should move into the free corridor to do the passes. Depending on the role assignment, the robot either needs to position in the free corridor to receive the ball or needs to dribble it in the free corridor to pass it. Figure 3.3 shows the destinations of the robots. The receiver walks to the position marked in the left field view with the blue cross and the passing robot dribbles the ball to the area marked in the right field view with the green rectangle.

3.2.3.4 Pass Ball

If the ball is in the free corridor or was dribbled into it, it can be passed to the receiver. In case the receiver is not in position to catch the ball, the passing robot waits until the receiver is close to its position. The time for which the robot waits to pass depends on the orientation of the receiver relative to the passing robot. If the receiver is roughly orientated towards the passing robot, the pass can be done earlier to save some time, because it can be assumed that the receiver can catch the ball. If the orientation of the receiver is not yet towards the passing robot, the passing robot waits to prevent a bad catch, which would probably take more time than waiting for the receiver.

3.2.3.5 Receive Ball

If the receiving robot is at the desired position, the robot waits until the other robot kicks the ball. When the ball is rolling towards the receiver, it can do sidesteps to adjust its position in order to catch the ball.

3.2.3.6 Search for Ball

While playing, the robots could lose track of the ball. This does not necessarily need to be due to a poor ball perception. As mentioned in section 3.1, the ball could be reset to attacker

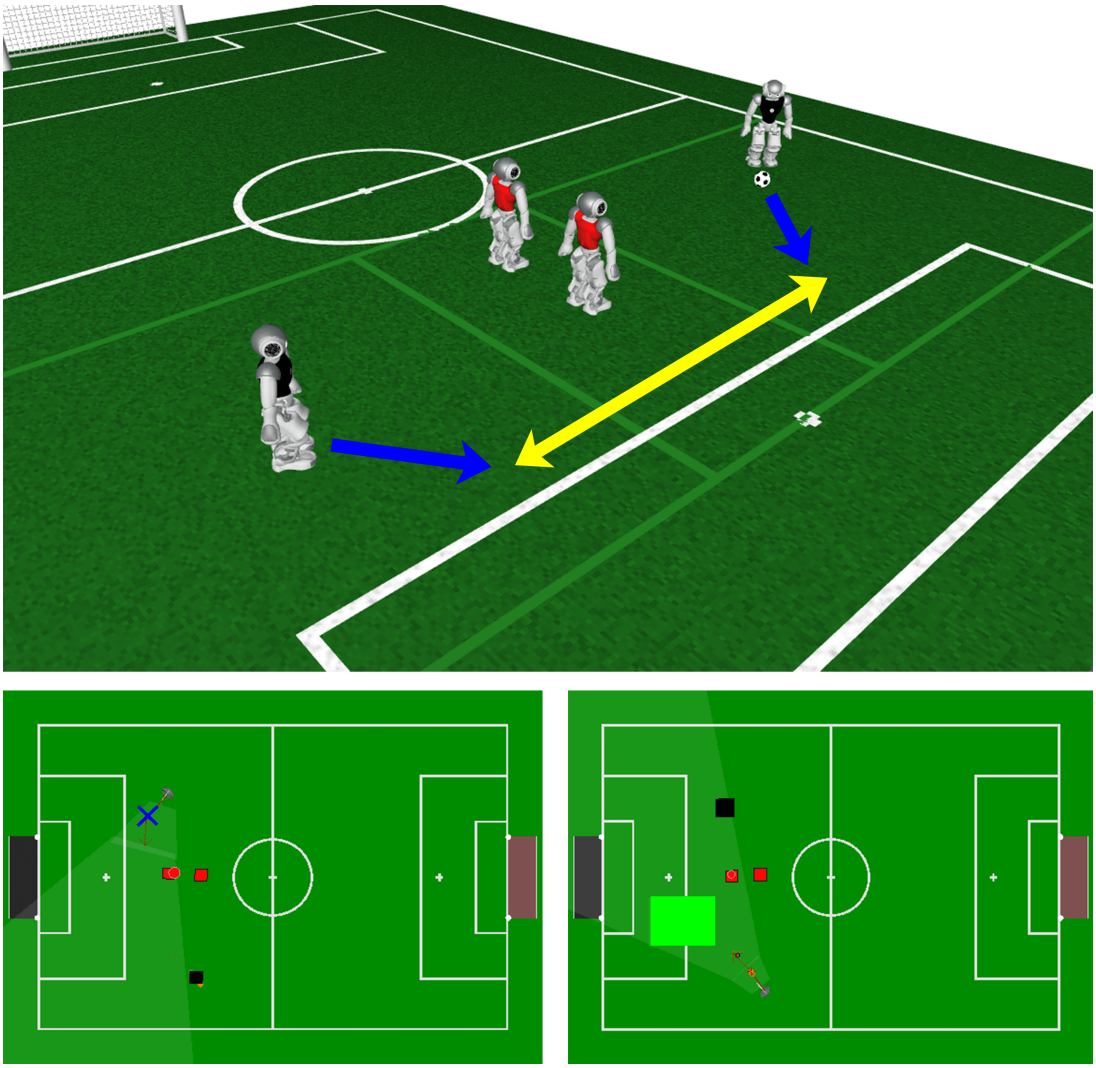


Figure 3.3: Upper: The yellow arrow represents the free corridor. The blue arrows illustrate where the robots have to move to get in passing position. Left: Field view of the receiver. Right: Field view of the passing robot.

position A or I.

When the ball was not seen by an attacker for a certain time, a passing challenge-specific ball search starts. It works with the assumption that the ball perception works fine in a relatively short radius. This makes the task easier to solve, since the ball can only lie in one of the rather small red zones.

The robot searching for the ball turns around for 360° in the direction that the ball was last seen in. If after doing this the robot still has not seen the ball, which rarely occurs, the robot first moves to the center of its red zone. Once it arrives there, it turns around itself again until it or the other attacker found the ball.

However, if the robot was standing outside the field when starting to search for the ball, it skips the first spin and goes straight to its red zone's center to turn around there. This behavior was implemented to eliminate every chance of staying outside the field for more than 30 seconds which is illegal as described in section 3.1.

3.2.3.7 High Risk Pass

As a result of an imprecise pass or a bad first touch of the receiving robot, the ball can get in a position where the robot first needs to move it back to the free corridor to kick the ball with low risk. But at the end of the challenge, this might take too long. Therefore, the passing robot can also do a high risk pass as a last chance to get one more point. A sector wheel is used to determine the best direction to pass. If the teammate is fully visible in the sector wheel, this direction has always the highest priority. If not, a fallback angle that points into the teammate's area is calculated. Then the receiver has to move to touch the ball in time.

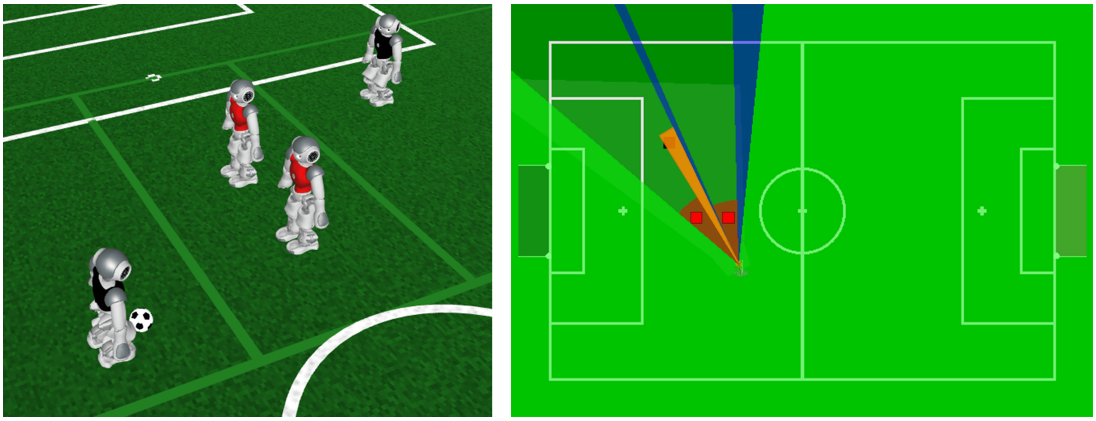


Figure 3.4: Situation of a high risk pass at the end of the challenge. The field view shows the sector wheel of passing angels. Angle colors: green - ball goes out; red - obstacle in the way; blue - free area to pass in; orange - teammate.

3.3 Results

Our strategy and all implemented features worked as intended. With a total of 19, 18 and 27 passes, we were able to place first. A complete table of all participants and their scores is shown in Table 3.1.

Both robots located themselves correctly and perceived the ball and the obstacles. The free corridor for passes was always selected correctly because of the correct obstacle model. The passBallPlayer role was assigned correctly so that the correct attacker always moved to the passing position with the ball and executed the pass, while the receiveBallPlayer went to the correct spot to wait for an incoming pass. When losing track of the ball, the robots could find it again rather fast and continued to play. All attempts were played until the time went out. Neither did the ball hit an obstacle, nor did an attacker leave the field for too long.

Future improvements could include a better transition between catching the ball and passing the ball. This would mean that the movements when receiving the ball need to be more careful to prevent accidentally hitting the ball, which might cost more time. At the same time however, the behavior including movement would need to be faster to achieve a higher score.

	1. Run	2. Run	3. Run	Combined (Best)	Overall Rank
B-Human	19	18	27	27	1
Nao Devils	2	0	8	8	2
Bembelbots	5	2	0	5	3
HTWK Robots	2	0	4	4	4
rUNSWift	0	2	1	2	5
UT Austin Villa	1	0	0	1	6
R-ZWEI KICKERS	0	0	0	0	7
SPQR Team	0	0	0	0	7

Table 3.1: The results of the RoboCup 2021 Passing Challenge

Chapter 4

1vs1 Challenge

4.1 Task

As one of the two remote challenges, it got carried out with an unknown robot on another teams' stadium. For the other remote challenge, see chapter 5.

In this challenge, there are two robots competing for the highest score. A point will get awarded by either scoring a goal or when the ball stops in the opponent half. Scoring a goal will result in replacing the ball to the scorer's goal area again, thereby it is the favorable way to score points. If the opponent robot touches the moving ball, no point will be given. Kicking over the ground line will result in a placement at the goal area. A game is split in two halftimes by switching sides after five minutes, which can be prolonged for one additional minute, if a tie occurs. At the start of a game, the balls get placed on each corner of the goal areas, four in total. Meanwhile, the robots are standing outside the field in the *READY*-State. After the game started (signalized by a whistle), they are allowed to move to their starting positions in 45 seconds. During a game, the robots are not allowed to surpass the middle line. If it happens nonetheless, it results in a timeout for the given robot.

4.2 Approach

We divided the behavior into an *offensive (or "with-ball")* and a *defensive (or "without-ball")* strategy so we can handle them separately.

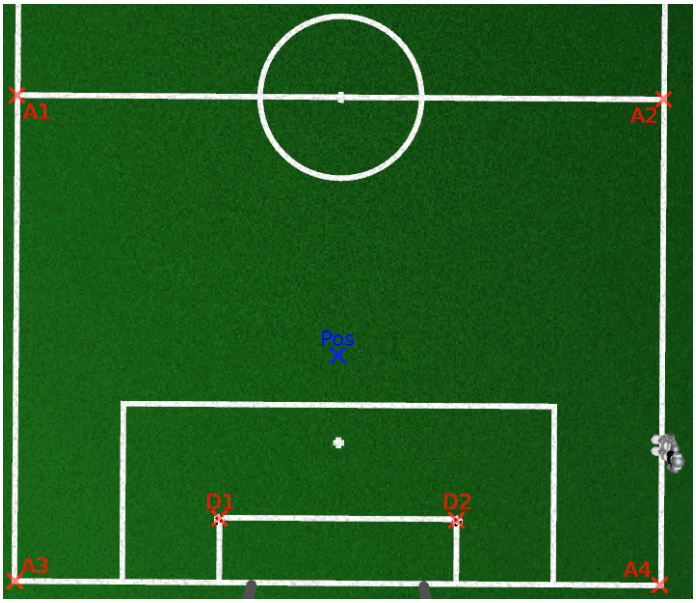


Figure 4.1: The visual anchor points (in red) and the position we are standing when looking for the ball (in blue). A1 - A4 are the primary anchor points, D1 and D2 the secondary ones

4.2.1 Offensive Strategy

The attack strategy basically consist of a simple concept with some minor changes, depending on certain situations. The basic strategy is that the robot will dribble with the ball to a certain distance to the opponent's goal and then kick towards the goal. This distance is here defined as the maximum gap between the goal and the ball where the robot is still able to score. Additionally, the dribbling speed declines as the robot comes nearer to the kicking position.

Furthermore, we added some behaviors to cope with certain special situations. Firstly, the robot does not kick when an opponent is blocking the direct way to score a goal. Our robot will wait a bit for the opponent to leave the kicking path. But if the opponent is not leaving, the robot will just do a weak kick to a free area, in which the ball will land nearest to the opponent goal line. The robot can only change his decision of the free area one time. After that, it will be forced. Secondly, right before the end of a halftime, the robot will kick from every position (we do not dribble anymore) to score the most points possible. Thirdly, we implemented a fallback strategy in case the robot is not able to score from the shooting position. If it does not score three times in a row because its kicks the ball out of the field (the necessary information is communicated), it will switch to weaker kicks to at least score some points in a different way.

4.2.2 Ball Search

Kicking a ball into the opponent's half results in a state, where the robot doesn't know the position of the next ball. To cope with that, we have an immediate and a repetitive behavior. Overall, we have four primary and two secondary visual anchors (see fig 4.1 for points) to alternate between. Directly after kicking a ball into the opponent half, the robot walks to a point near the center circle. While doing that, it is first looking to A1 and then to A2. By doing so, the robot is able to find balls laying near the center line quicker (e. g. if the opponent only dribbles the ball over the line). If the robot does not find any ball, the repetitive behavior kicks in; the robot moves towards the search position (see fig 4.1) and focuses its view on either the middle most primary, the left most primary or the right most primary visual anchors (the algorithm gets

Stage	Opponent	Score
Group-Stage	Dutch Nao Team	16.5:0
Group-Stage	SPQR Team	25.5:0
Quarterfinal	NomadZ	22.5:10
Semifinal	Nao Devils	21:15
Final	HTWK Robots	18:19.5

Table 4.1: The results of the RoboCup 1vs1 Challenge. Mind the factor of 1.5 for autonomous calibration.

through these combinations in this order). While looking towards A3 or A4, the robot aligns itself accordingly (rotation by 90/-90 degrees). If the robot doesn't find any ball, it walks back to the goalie position (between the goal posts and right before the ground line). In most of the cases, if the robot walks in the direction towards the secondary visual anchors (drop in points D1 and D2), it alternates between them. This happens, for example, after the immediate behavior after the kick. By doing so, the drop in points will get observed, and the robot spots any ball it kicked into the opponent goal as fast as possible.

If the robot found a ball during execution of the repetitive algorithm, it saves the state of the search it found the ball in. The next time the search algorithm starts, it will start its search in exactly that state. Often we kick the ball into the opponent goal, but the ball doesn't get back in time. In the worst case, the ball does get on the drop in point D2, after the robot places itself into the goal. Normally, the algorithm would recognize the ball in the last case. But if this happened before that, it will start by looking at that point and finds the ball quicker.

4.2.3 Defensive Strategy

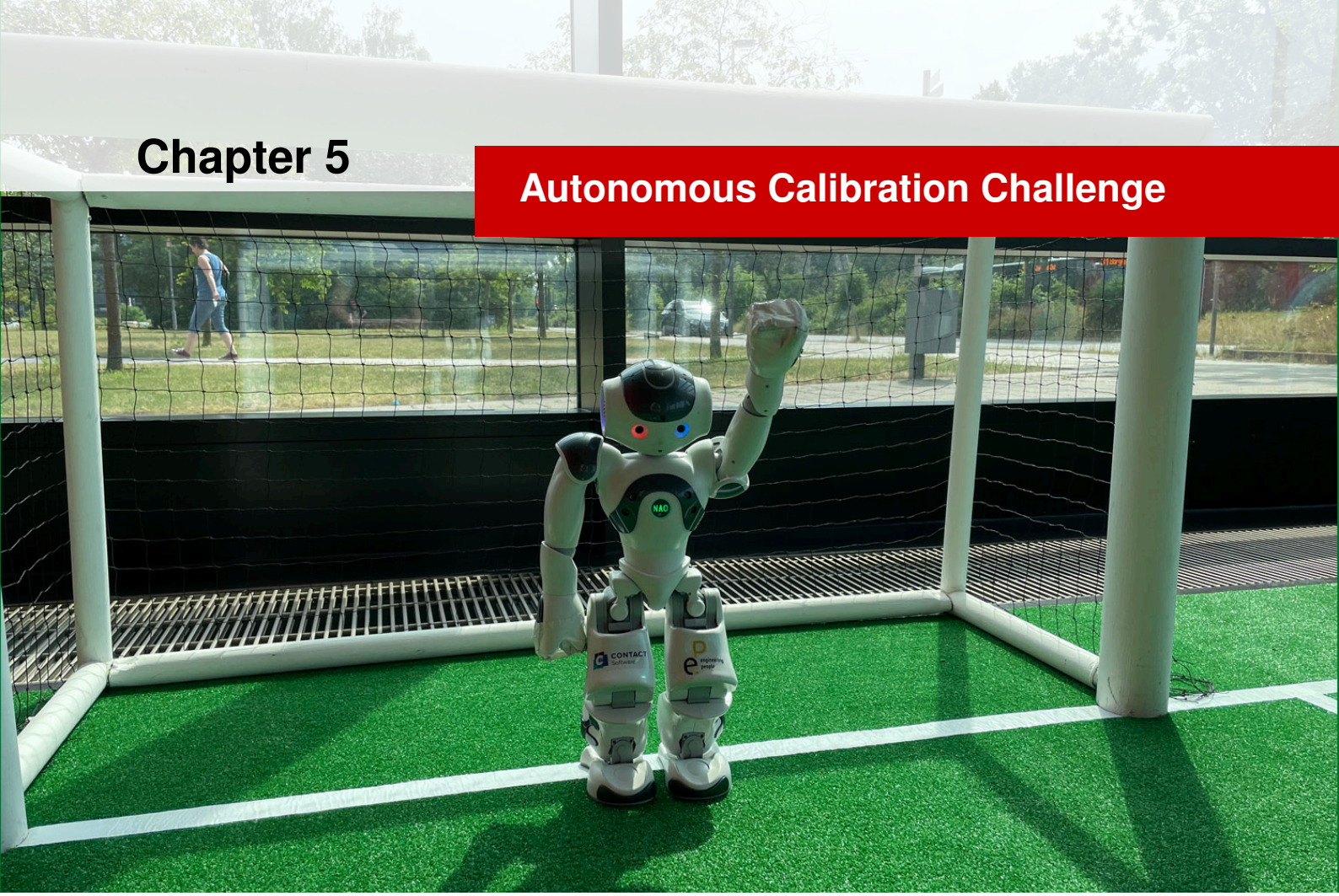
The defensive strategy consists of an active and a passive defense mode. The active defense is basically a goalkeeping strategy: If the ball rolls towards the goal and can be intersected, the robot will either stand still or walk to an appropriate point for interception. During passive defense, the robot's behavior is tailored to maximize the chance, to see an incoming ball. This is realized by looking towards near obstacles which got identified as a robot. Due to technical limitations, the robot cannot always see an opponent. In that case, the robot's view alternates between both drop in points. If a ball was seen and gets lost (again due to technical limitations), the robot will continue to look to that point. After a short period of time it continues with the normal algorithm. Although in theory it's nice to have a defensive behavior, the practical value of it is less than the one of the search for a ball. Thus, most of the time the robot is not in the defensive behavior.

4.3 Results

We won every game despite the last one. For yet debatable reasons we did not score any goal in this game. The ball always took a big curve and went just next to the goal. As described in our attacking approach, our fallback strategy came into place. This strategy was not designed to maximize the scored points, since we still dribble to the middle half and then do a weak kick instead of kicking from every position with a stronger kick. Thus, if the opponent is points ahead, we most likely cannot win with this strategy, if the opponent does not major mistakes. This was exactly the case in the final.

Chapter 5

Autonomous Calibration Challenge



5.1 Task

Along with the 1vs1 Challenge (chapter 4), the Autonomous Calibration Challenge is one of the two remote challenges, i. e. it is held at another team's venue and with an unknown robot. The challenge consists of two consecutive parts.

First, up to 10 minutes are allowed for the calibration process. During this time, the robot can move freely, as long as it does not touch a ball or leaves the field. If the robot finishes the calibration in less than 10 minutes, it may end the calibration time early.

The second part of the challenge is the evaluation phase, in which the robot has to move to specific target points and report positions of two balls that have been placed randomly before. Just as in the calibration phase, the robot may move freely, but should not touch either ball, nor leave the field area. In the up to 10 minutes that are allowed for the evaluation phase, the robot has to complete the following five tasks:

1. Report the field-relative position of the ball in the robot's own half.
2. Walk to the center of the goal in its own half, at any rotation.
3. Walk to the center circle ball kickoff spot, at any rotation.
4. Report the field-relative position of the ball in the opposition half.
5. Walk to the center of the goal in the opposition half, at any rotation.

After the robot finished the evaluation phase, the distance between the robot's actual position and the correct target positions as well as the distance between the reported ball positions and the correct ones are measured and rounded to the nearest 10 cm. These results serve as the basis for the scoring. The scoring is done by ranking the performance in the following seven categories:

1. Fastest time to complete the automatic calibration phase.
2. Fastest time to complete the evaluation phase.
3. Closest reported position of the ball in the robot's own half.
4. Closest reported position of the ball in the opposition half.
5. Closest robot position at the goal in the robot's own half.
6. Closest robot position at the center circle.
7. Closest robot position at the goal in the opposition half.

The team with the lowest total rank wins the challenge [2].

5.2 Approach

In the following, the behavior during the two phases of the Autonomous Calibration Challenge is described.

5.2.1 Calibration Phase

To be able to precisely estimate positions on the field, the robot needs to be able to project points from image space onto the field plane.

As each robot's camera is attached with a slightly different rotation, we need to correct these rotations by applying a matching opposite rotation to the image. To find these values we perform the extrinsic camera calibration on each robot.

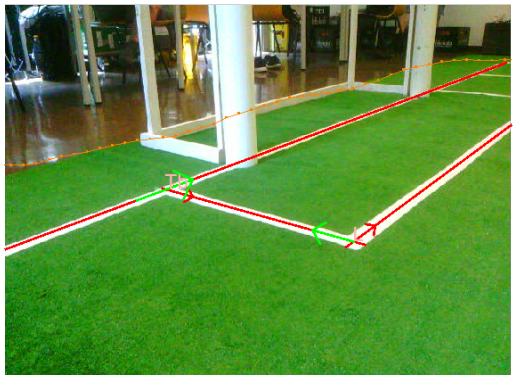


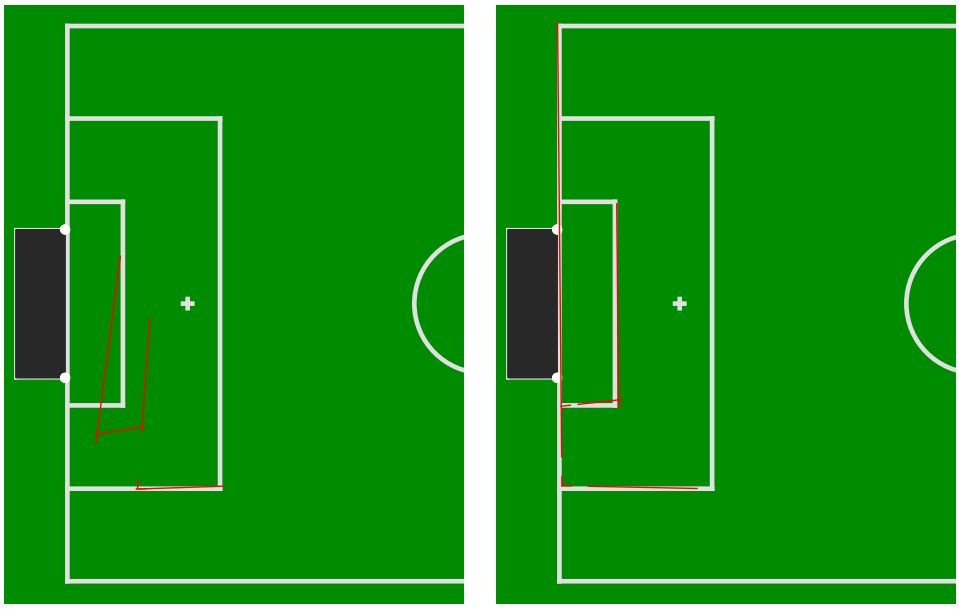
Figure 5.1: The robot's view during the calibration, taken during our attempt in Hamburg.

In the past, we used a semi-automatic process for camera calibration, in which the robot had to be placed manually on the field, collected samples and performed an optimization of the

rotation corrections. These samples were points located on detected field lines in images. The algorithm only required the field lines of the goal area to be seen in the image.

We were able to use this existing process as a baseline for our autonomous camera calibration approach, as the robot only needed to position itself roughly at the right position. On uncalibrated robots, the self-localization is still precise enough for robustly guiding the robot to the necessary positions, which are not far away from the robot's starting point.

Our remaining task mostly consisted of constructing the behavior to reliably perform the calibration. First, the robot must move autonomously to the required position. To ensure that the calibration process can be performed successfully, the robot needs to see all necessary field lines. If this is not achieved right away, the robot will turn its head and try some different angles. If the robot still misses some important line, it will move to another position within a 40 cm radius and again try to record a sample. After all the required samples are recorded, the optimal calibration parameters are calculated and saved and the autonomous calibration is finished.



(a) The projected field lines before calibration

(b) The projected field lines after calibration

Figure 5.2: Results of the autonomous camera calibration on a robot of the HULKs team during RoboCup 2021.

5.2.2 Evaluation Phase

In the second phase of the challenge, the previously done calibration is evaluated. Two essential components of our strategy are the precise positioning at targets and the ball search. These are both relevant several times as the subtasks of the evaluation phase are similar to each other.

5.2.2.1 Precise Positioning at Targets

Due to the way the scoring system was designed, we decided to focus more on precision than on speed. Therefore, after reaching the target position, the robot waits for a moment to look around and improve its localization. It follows a second positioning attempt before the position

	Dortmund	Leipzig	Sydney	Amsterdam	Hamburg	Bremen	Average	Overall Rank
B-Human	9			9	9		9.0	1
Nao Devils		11		11	15		12.3	2
UT Austin Villa	18	12	14				14.7	3
HTWK Robots	27		10			13	16.7	4
Bembelbots		19		19	20		19.3	5
Berlin United – NaoTH		26	20			16	20.7	6
rUNSWift				25	22	15	20.7	6
SPQR Team	28		28			22	26.0	8

Table 5.1: The combined ranking of the Autonomous Calibration Challenge. The columns show the combined rank the teams reached at the different venues.

is reported. When positioning itself at one of the goals, the robot will face the field to see more field features and therefore be able to estimate the own position more precisely.

5.2.2.2 Ball Search

If the robot cannot detect the ball by looking around at the initial position, it moves to the center of the currently used field half and performs a 360 degree rotation. By doing this, we already cover the majority of the area, where the ball could have been placed. If the ball still cannot be found, the robot subsequently walks to the areas that are not easily visible from the center of the field half, i. e. it walks along the edge of the field and stops regularly to look around. This search strategy is interrupted when the robot detects the ball. It then moves closer to the ball to confirm its position and make it more precise. As we already know that the next task will be to walk to the goal of the same field half, the robot chooses a route towards the ball that is as close as possible to the path towards the goal. Just like the robot corrects its own position before reporting it, it also takes a moment to get its own position as exact as possible before reporting the ball because the ball position has to be reported in global coordinates, so the self-localization has a huge impact on it. To ensure that all tasks are finished in time, we will abort the search and report the currently guessed ball position if the whole process of searching the ball takes too long.

5.3 Results

Our Autonomous Calibration Challenge took place at three different venues. We managed to achieve the lowest overall rank in each one of them (Table 5.2, Table 5.3, Table 5.4) and thus won the whole challenge (Table 5.1). The strategy to focus more on precision than on speed worked out, allowing us to achieve a rounded distance of 0 cm or 10 cm in each of the five categories regarding precision. Even though speed was only our second priority, we achieved comparatively good results and were able to reach at least the second rank in the categories *calibration time* and *evaluation time*.

Arena Dortmund	B-Human		UT Austin Villa		HTWK Robots		SPQR Team	
Calibration Time	0:40	*2*	0:00	*1*	1:00	*3*	-	*4*
Evaluation Time	1:55	*2*	1:40	*1*	-	*4*	-	*4*
Own Half Ball Error	0 cm	*1*	220 cm	*2*	-	*4*	-	*4*
Opponent Half Ball Error	0 cm	*1*	-	*4*	-	*4*	-	*4*
Own Goal Position Error	10 cm	*1*	610 cm	*2*	-	*4*	-	*4*
Center Position Error	0 cm	*1*	-	*4*	-	*4*	-	*4*
Opponent Goal Position Error	10 cm	*1*	-	*4*	-	*4*	-	*4*
Combined		*9*		*18*		*27*		*28*

Table 5.2: The results and the ranking of the Autonomous Calibration Challenge in Dortmund.

Arena Amsterdam	B-Human		Nao Devils		Bembelbots		rUNSWift	
Calibration Time	0:55	*2*	2:40	*3*	2:40	*3*	0:00	*1*
Evaluation Time	2:20	*1*	2:20	*1*	7:25	*3*	-	*4*
Own Half Ball Error	10 cm	*1*	10 cm	*1*	30 cm	*3*	-	*4*
Opponent Half Ball Error	10 cm	*2*	0 cm	*1*	30 cm	*3*	-	*4*
Own Goal Position Error	0 cm	*1*	20 cm	*2*	20 cm	*3*	-	*4*
Center Position Error	0 cm	*1*	0 cm	*1*	40 cm	*3*	-	*4*
Opponent Goal Position Error	0 cm	*1*	10 cm	*2*	10 cm	*2*	-	*4*
Combined		*9*		*11*		*19*		*25*

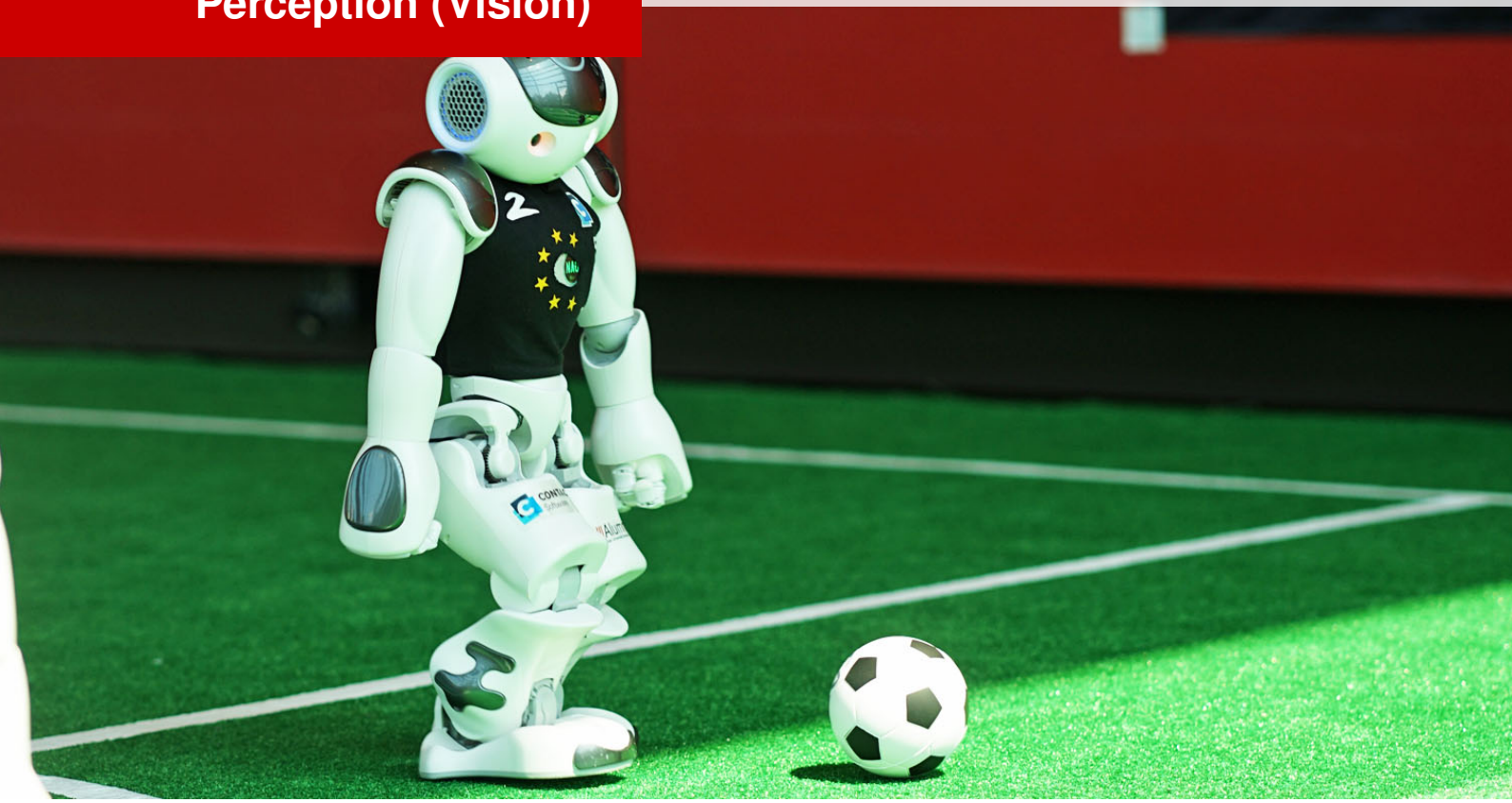
Table 5.3: The results and the ranking of the Autonomous Calibration Challenge in Amsterdam.

Arena Hamburg	B-Human		Nao Devils		Bembelbots		rUNSWift	
Calibration Time	0:40	*2*	1:45	*3*	2:55	*4*	0:05	*1*
Evaluation Time	2:10	*2*	3:00	*3*	4:05	*4*	1:35	*1*
Own Half Ball Error	0 cm	*1*	20 cm	*2*	30 cm	*3*	-	*4*
Opponent Half Ball Error	0 cm	*1*	40 cm	*3*	10 cm	*2*	-	*4*
Own Goal Position Error	10 cm	*1*	10 cm	*1*	10 cm	*1*	460	*4*
Center Position Error	0 cm	*1*	10 cm	*2*	30 cm	*3*	290 cm	*4*
Opponent Goal Position Error	0 cm	*1*	0 cm	*1*	10 cm	*3*	320 cm	*4*
Combined		*9*		*15*		*20*		*22*

Table 5.4: The results and the ranking of the Autonomous Calibration Challenge in Hamburg.

Chapter 6

Perception (Vision)



In the past our vision pipeline depended on manual calibration of several parameters to the lighting conditions of the venue. Because this takes up a lot of time during game preparation and cannot adapt to changes of the lighting conditions during the game, we rebuilt our vision pipeline to not depend on manual calibration. Mainly affected were the field boundary detection, color segmentation and field line detection, which can now work on single images without prior calibration. This reduces calibration time, makes remote setup easier and enables adaption to changing lighting conditions in game.

6.1 Field Boundary Detection Using Convolutional Neural Networks

Detecting the field boundary is one of the first steps in our vision pipeline. Conventional methods make use of a (possibly adaptive) green classifier, selection of boundary points and possibly model fitting. Our new approach is based on [11] and predicts the coordinates of the field boundary column-wise in the image using a convolutional neural network. This is combined with a method to let the network predict the uncertainty of its output, which allows to fit a line model in which columns are weighted according to the network's confidence. Experiments show that the resulting models are accurate enough in different lighting conditions as well as real-time capable. Figure 6.1 shows the detected field boundary and the predicted uncertainty. For a detailed explanation see [3].

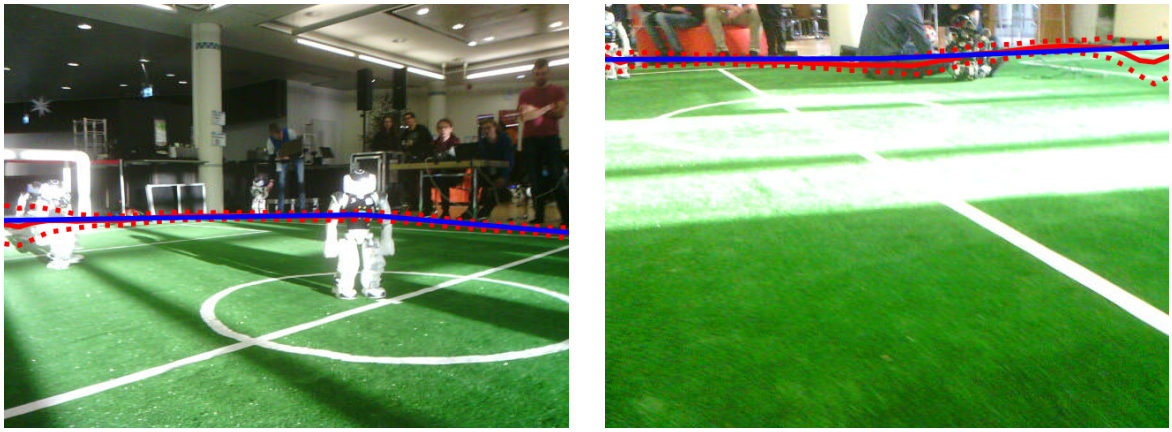


Figure 6.1: Sample images with the output of the network with uncertainty. The solid red line is the predicted mean, the dotted lines are $\pm\sigma$ intervals, and the blue line is the fitted model.

6.2 Lighting Independent Field Line Detection

The `LinePerceptor` detects field lines in the camera images. It relies on the `ColorScanLineRegions` representations which hold segmentations of horizontal and vertical lines of the image into green, white and a generic class for everything else. In the computation of this segmentation, many thresholding parameters are needed. In past years, these had to be calibrated by hand before the start of the game. Instead, we now calculate all needed parameters per image, allowing the perception to adapt to lighting changes during the game.

6.2.1 Relative Field Colors

Used multiple times throughout the lighting independent vision, the `RelativeFieldColors` representation bundles functions that classify color values into field and white using nearby image regions. Given an image region is already classified as white or field, the luminance and saturation values of a nearby region or pixel can be compared to the already classified region's color values. Considerably higher luminance paired with lower saturation compared to a field region means that the tested color value is probably white. Analogously, the test for field regions goes the other way around. Additionally, we then test if the presumed white or field region satisfies generic thresholds for luminosity, saturation and hue.

6.2.2 Color-Segmented Scan Lines

Scan lines are a means to analyze only a part of the image while retaining as much useful information as possible. This is especially important now that our color segmentation approach takes significantly more execution time. That makes it impossible to apply the color segmentation to the whole image, so instead we use it only on the scan lines.

The `ScanGrid` defines scan line positions. A scan line is simply a vertical or horizontal line of pixels in the image, so each scan line position is signified by a single x- or y-coordinate in the image plane.

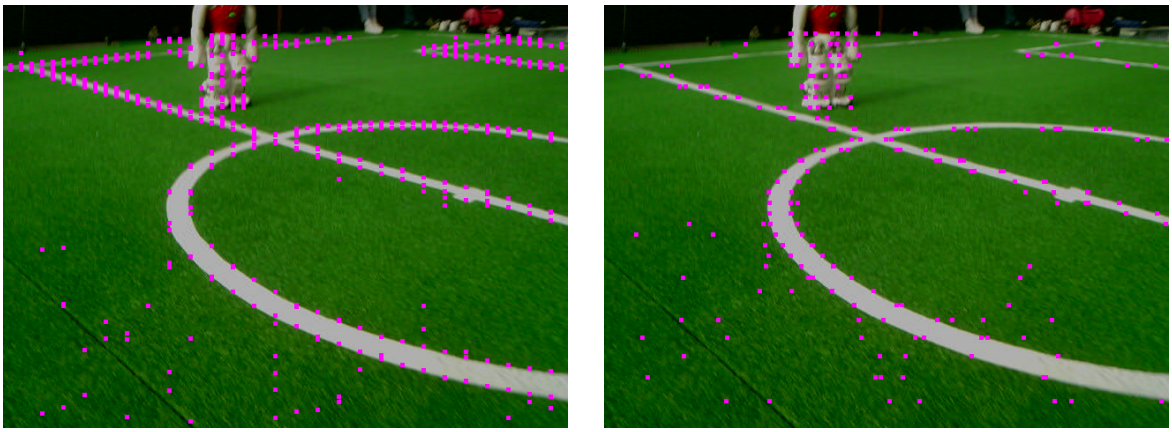
The color segmentation of the scan lines is done by the `ScanLineRegionizer` in four steps:

1. Split each scan line into homogeneous regions

2. Classify regions as field
3. Classify regions as white
4. Perform cleanup operations

Currently, this is done for the vertical and horizontal scan lines independently of each other.

For the first step, we draw samples in regular intervals on each scan line. For each sample point, we apply a Gaussian blur filter on the luminance to smooth out noise. When two consecutive sample points' smoothed luminances differ significantly, a Sobel filter is applied to the interval in between. The highest Sobel value then signifies the exact position that separates two regions of the scan line. In Figure 6.2, the resulting region transition points are marked with pink dots. Each region gets a representative YHS2 (see [10, p. 60]) color value by averaging over its pixels.



(a) Region transition points of vertical scan lines (b) Region transition points of horizontal scan lines

Figure 6.2: The pink dots indicate the positions where two scan line regions are separated.

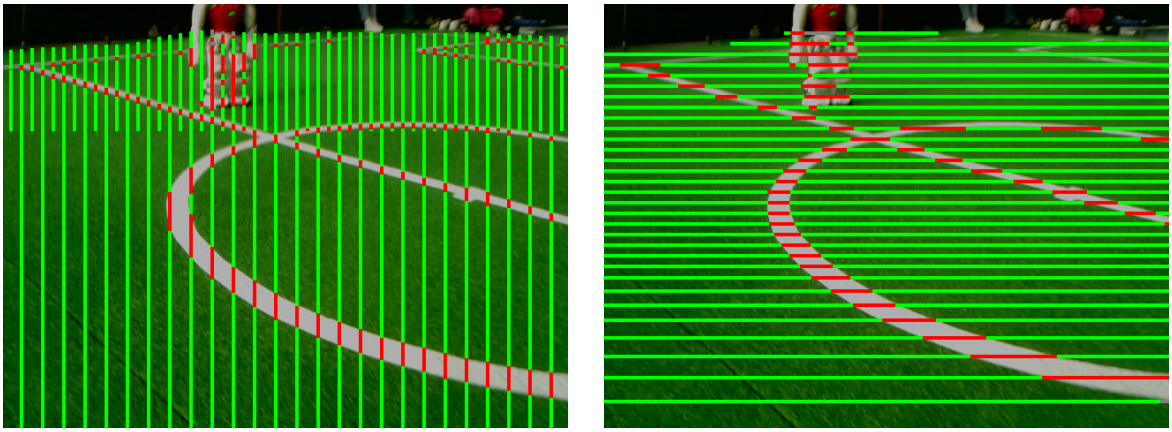
The field often appears in the image in form of rather big homogeneous patches that are within a certain color range. In order to use this characteristic, we unite neighboring regions with similar YHS2 color values. This is done efficiently using a union-find disjointed tree data structure. All united regions that span enough pixels and are in the expected color range of the field become classified as field and allow to determine the exact color range the field has in this image. Afterwards, all remaining regions are also classified as field, if they are within the ascertained field color range.

The classification of white regions utilizes simple thresholding techniques. The thresholds depend on the previously established field color and the approximated average luminance and saturation of the image. The results can be seen in Figure 6.3.

Finally, neighboring regions on the same scan line, which have the same classification, become united into one region. We also noticed that this classification procedure sometimes leaves small gaps in between a field region and a white region. Because this impairs the line detection, we fill small gaps between a field and a white region by dividing them up equally into the neighboring regions, as can be seen in Figure 6.4.

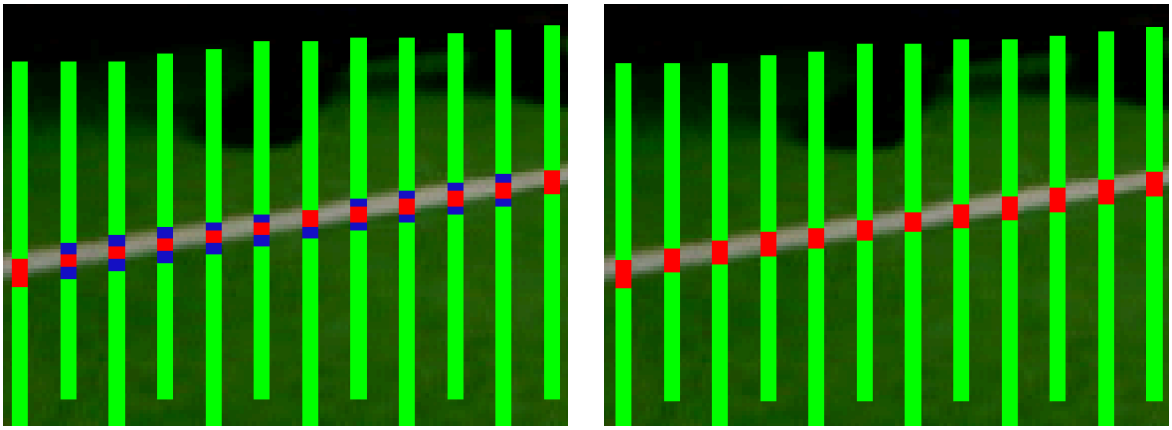
6.2.3 Line Detection

The perception of field lines by the `LinePerceptor` relies mostly on the scanline regions. In order to find horizontal lines in the image, adjacent white vertical regions that are not within a



(a) Vertical color segmented scan lines (b) Horizontal color segmented scan lines

Figure 6.3: The color segmented scan lines. For better contrast the regions classified as white are drawn in red.

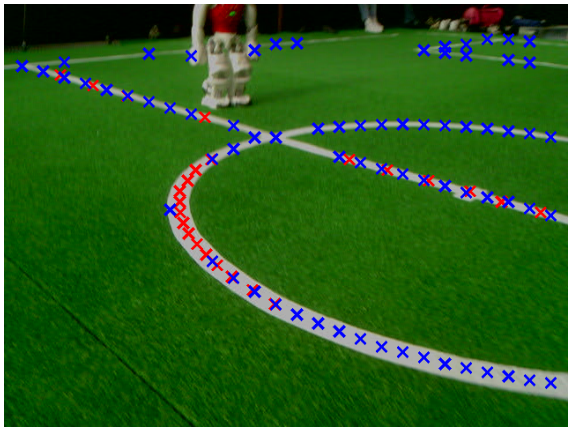


(a) Small wrongly classified regions at the transitions between field and line (b) Transition between field and line with filled in gaps

Figure 6.4: Filling in of small gaps between a field and a white region. Regions classified as white are shown in red.

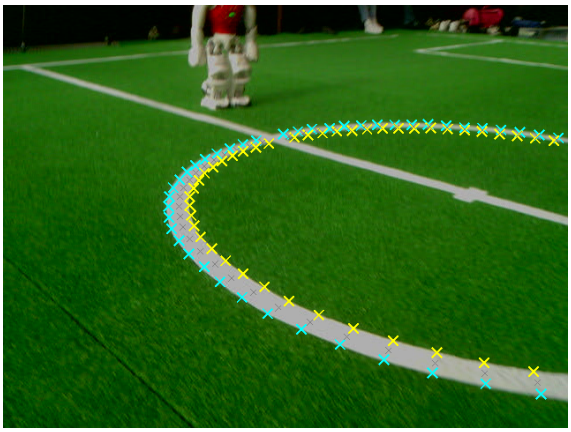
perceived obstacle [10, pp. 71-74] are combined to line segments. Correspondingly, vertical line segments are constructed from white horizontal regions. These line segments and the center points of their regions, called *line spots*, are then projected onto the field. Using linear regression of the line spots, the line segments are then merged together and extended to larger line percepts. During this step, line segments are only merged together if at least a given ratio of the resulting line consists of white pixels in the image. Figure 6.5 shows the process of finding lines and the center circle in the camera image.

We don't compute the color segmentation for the whole image any more. Instead, the pixels on the presumed line are compared to their surroundings in order to find out if the line is actually white. In regular intervals, we draw samples from a presumed line, compute two positions above and below the line for each sample and then use the `RelativeFieldColors` white test for each pair of sample and comparison position. That effectively results in testing whether the line is white and embedded in green surroundings. Figure 6.6 shows the positions of the samples and their comparison positions.

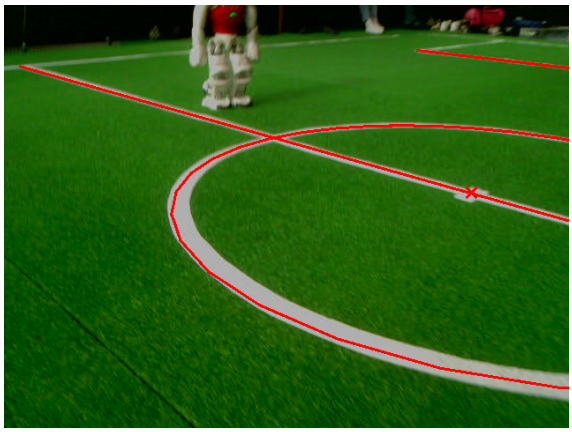


(a) Center points of the line spots. The ones found on vertical scan lines are marked in blue and finds on the horizontal scan lines are marked in red.

(b) The line segments built up from the line spots



(c) Circle candidate with points on inner and outer edge marked in yellow and cyan



(d) The final perception of field lines and the center circle

Figure 6.5: The process of finding field lines

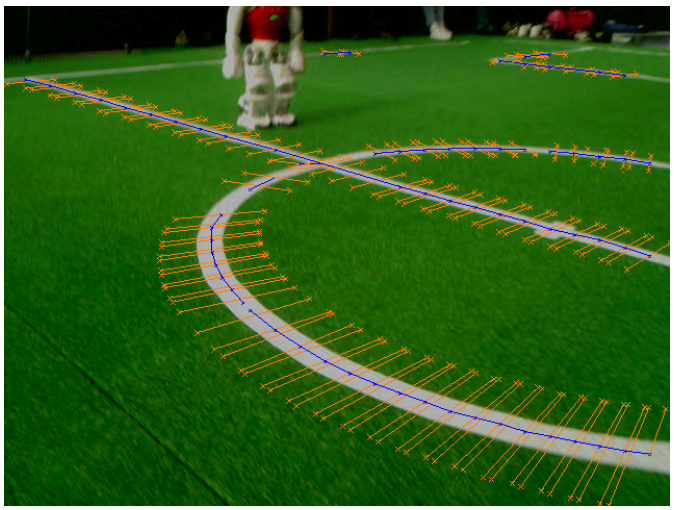


Figure 6.6: Sample points (blue) and comparison points (orange) for the white test of line segments.

Chapter 7

Proprioception (Sensing)

7.1 Inertial Sensor Data Filtering

The `InertialDataProvider` module determines the orientation of the robot's torso relative to the ground. Therefore, the calibrated IMU sensor readings (`InertialSensorData`) and the measured stance of the robot (`RobotModel`) are processed using an Unscented Kalman filter (UKF) [5].

7.1.1 Stabilizations

In the recent years, we noticed that the estimation of the orientation is often off by up to seven degrees in either the x- or y-rotation. This resulted in a phenomenon where the projected field lines in robot relative coordinates where up to ten times longer than they should be. In general, the orientation sometimes had a drift. Both combined made it quite hard to get an accurate localization and to apply any walk balancing, which was based on the orientation.

To reduce this problem, we use the planes of the feet whenever we can be sure that they have ground contact and can be seen as some kind of ground truth measurement. When a support foot switch is measured while walking, we know for a fact that both feet must have ground contact. Therefore, we calculate which of the four edges of each foot sole is the lowest in the robot coordinate system. Both points are then used as a vector, which spans from the right to the left. With this vector, we can calculate the theoretical acceleration measurement for the y- and z-axis. With these we update the UKF. As a result, the estimation for the x-rotation of the orientation is more stable and shows less drift.

Additionally, in case the foot plane rotation is calibrated[1], we use its orientation to calculate the theoretical acceleration measurement and overwrite the actual measured x-acceleration and set its deviation to a low value. This is only done once per walking step and only if the supporting foot measures pressure in the toe and the heel.

7.2 Arm Contact Recognition

Robots should detect whether they touch obstacles with their arms in order to improve close-range obstacle avoidance. When getting caught in an obstacle with an arm, there is a good chance that the robot gets turned around or falls.

We improved our previous recognition by using different thresholds for errors in the forward and side direction, which scale based on the walking speed.

7.3 Foot Pressure Filtering

The `FootSupportProvider` filters the `FsrSensorData` and calculates how much weight is supported relative of the feet to each other. This procedure is taken from the walking approach of the rUNSWift team [4].

Compared to previous versions, we now not only calibrate the highest measured weights, but the lowest too. Additionally, support foot switches are only accepted if the new support foot measures pressure above a threshold, which is automatically calibrated too. If a support foot switch would be detected without this threshold, the detection waits until the weights are high enough to set the flag for a detected switch.

We also simplified our support foot prediction. This prediction is currently only used in the simulation and is planned to be used in the near future.

[1] `https://wiki.b-human.de/Coderelease2021/getting-started/#foot-sole-rotation-calibration`

Chapter 8

Robust Motion

The B-Human motion control system provides and controls the motions needed to play soccer with a robot. There are seven different types of motions that can be requested: *walk*, *stand*, *kick*, *fall*, *getUp*, *keyframeMotion*, and *playDead*. These motions are generated by the corresponding motion engines. Additionally, there are specific engines for the head motions and arm motions. The *walk* and *stand* motions are dynamically generated by the `WalkingEngine` (cf. Section 8.3). Moreover, this module provides a special kind of kicks, the in-walk kicks (cf. Section 8.3.4). They are generated by the `WalkKickEngine` and executed by the `WalkingEngine`. All the other kicks are generated by the `KickEngine` [8] that models kicks by using Bézier splines and inverse kinematics. When a robot is unintentionally falling, the `FallEngine` prevents damage (cf. Section 8.4). A robot which is laying on the floor can get back to stand by using the `KeyframeMotionEngine` (cf. Section 8.5). This module also generates static motions. Here, a sequence of predefined joint angles is used as an input and a motion is created on that basis. It is used wherever an own engine would be unnecessarily complex. Currently, it is used mainly for ball catching motions.

Each motion engine generates joint angles. Compared to previous years, every motion is represented as a `MotionPhase` (cf. Section 8.1). At any given moment, only one motion phase exists. A motion phase provides functions, set by their corresponding engine, to calculate new joint angles but also to check if the motion phase can be stopped to start a new one. The `MotionEngine` controls the execution and generation of new motion phases and combines – based on the selected motions (cf. Section 8.2) – the arm, head, and leg joints into the `JointRequest`, which is then sent to the robot.

An extra layer for walk and kick motions are the so-called `MotionGenerators`, which are provided by their own engines. They serve as an interface between the `MotionEngine` and the

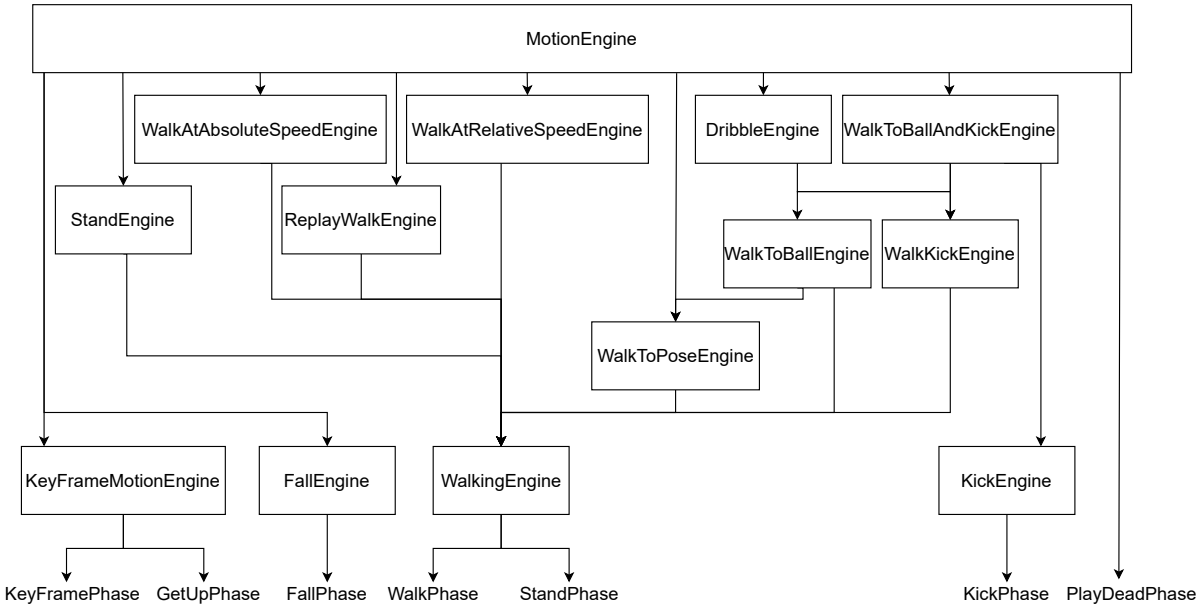


Figure 8.1: Overview of the hierarchy in the motion control

`WalkingEngine`. Some behavior parts such as walking around the ball are calculated in the thread *Motion*, given some constraints in the `MotionRequest`. If a walk or kick motion is requested and a new motion phase can start, the `MotionEngine` requests the corresponding `MotionGenerator` to generate a new motion phase. The motion control system is shown in Fig. 8.1.

8.1 Motion Phases

A motion phase provides multiple functions, overwritten by the corresponding engine:

1. *isDone* checks, if the current motion phase can be stopped and a new one can start.
2. *createNextPhase* is the constructor to create a new motion phase.
3. *calcJoints* calculates the joint positions to be requested.
4. *update* is called at the start of the `MotionEngine` to update the current motion phase and let it know that a new motion frame started.
5. *freeLimbs* gives information about if the head and the arms can be controlled by their own engine or if the motion phase sets them itself. For instance, while walking, the head is always controlled by the `HeadMotionEngine`, while the arms are only controlled by the `ArmKeyFrameEngine` if there is a request from the `ArmMotionRequest`. Otherwise, the motion phase takes over. In contrast, when the motion phase is a *keyframeMotion*, the head and arms are always provided by the motion phase.

A motion phase is completely independent of the previous one and only uses it to allow a smooth transition when generating a new motion phase. Therefore, the robot always only knows about the current motion phase and has no history about the previous ones.

8.2 Motion Selection

According to the representation `MotionRequest`, the `MotionEngine` determines which motion is executed. Here, the *isDone* function of the current motion phase is executed. If the motion phase returns `true`, then a new motion phase is created based on the `MotionRequest`. Afterwards, the head and arm joints are created and given the *calcJoints* function of the current motion phase, which in turn returns the complete joint request.

An exception to this behavior is the *fall* motion, which activates itself in the `MotionEngine` when needed and therefore ignores the *isDone* function, to achieve a reflex reaction. Thereby, it also ignores the state of the currently executed motion phase because if the robot is falling, it is unstable by default.

The information about the chosen motion for the legs is provided by the representation `MotionRequest`, for the arms by the `ArmMotionRequest` and for the head by the `HeadMotionRequest`.

8.3 Walking

The `WalkingEngine` provides the joint angles for walking and standing. The module coordinates the actual generation of the walk pattern with the execution of in-walk kicks (cf. Section 8.3.4) and it combines the odometry computed from the requested motion with the odometry computed from measured motion. It is based on the walk developed by Bernhard Hengst [4] for the team UNSW Australia/rUNSWift.

Additionally, multiple generators provide the interface to generate specific walk behavior such as dribbling, kicking or walking to specific points, while also avoiding obstacles.

8.3.1 WalkingEngine

The *WalkingEngine*, is based on three major ideas:

1. As in all walks, the body swings from left to right and back during walking to shift away the weight from the swing foot, allowing it to be lifted above the ground. In contrast to many other walks, this is achieved without any roll movements in the leg's joints (unless walking sideways), i. e. the swing foot is just lifted from the ground and set back down again, which is enough to keep the torso in a pendulum-like swinging motion.

2. As a result, there is a clearly measurable event, when the weight of the robot is transferred from the support foot to the swing foot, which then becomes the new support foot. This event is detected by the pressure sensors under the feet of the robot. At the moment this weight shift is detected, the previous step is finished and the new step begins. This means that a transition between a step and its successor is not based on a model, but on a measurement, which makes the walk quite robust to external disturbances.

3. During each step, the body is balanced in the forward direction by the support foot. The approach is very simple but effective: the lowpass-filtered measurements of the pitch gyroscope are scaled by a factor and directly added to the pitch ankle joint of the support foot. As a result, the faster the torso turns forward, the more the support foot presses against this motion.

As an addition, we introduced a step adjustment (cf. Section 8.3.6) to improve the balancing, which mainly prevents walking steps that let the robot walk in the opposite direction of its potential falling direction.

Furthermore, as a result from the new motion framework, *kick* and *getUp* motion phases can be left instantly when executing a *walk* phase. There is no transition time, instead, the robot starts directly with a walking step. Combined with the step adjustment, the transitions between the different motion phases are more robust and the robots fall less often.

8.3.2 Generating a Step

A walking step is a 2D-pose, a translation with a rotation. It describes how the robot will move, if it first places the current swing foot parallel to its supporting foot to reach the zero position and then translates and rotates corresponding to the requested step. A step is a motion phase of the type *walk*. To generate and execute a step, there exist multiple sections.

Step size request

A step is generated by one of the walking generators:

1. `StandGenerator`: generates a motion phase that lets the robot simply stand still. The walk step is a pose with only zero values.
2. `WalkAtRelativeSpeedGenerator`: generates a step based on percentages of the maximum walking speed.
3. `WalkAtAbsoluteSpeedGenerator`: generates a step based on absolute walk step sizes.
4. `WalkToPoseGenerator`: given a walking path to avoid obstacles, a step is calculated to reach the given robot-relative pose.
5. `WalkToBallGenerator`: given the relative ball position, transformed as if the robot would already have executed a zero step, and a kick pose, the robot calculates a step to reach the given kick pose. Given some constraints, the `WalkToPoseGenerator` is used to generate this step.
6. `DribbleGenerator`: given the relative ball position, transformed as if the robot would already have executed a zero step, the robot uses the `WalkKickGenerator` to generate an in-walk kick (cf. Section 8.3.4). If no kick can be generated, the `WalkToBallGenerator` is used to walk to the kick pose and generates a step.
7. `WalkKickGenerator`: the `WalkKickGenerator` provides two functions. The *canStart* checks if the requested kick and kick direction is executable for the next step, while *createPhase* generates the in-walk kick as a motion phase which is then the next executed step.
8. `WalkToBallAndKickGenerator`: given the kick type, kick direction and relative ball position, it is checked if the kick is executable. In case of an in-walk kick, the `WalkKickGenerator` is tasked for it, otherwise the `WalkToBallAndKickGenerator` does it itself. If so, the `WalkKickGenerator` or the `KickEngine` then generates the next motion phase. Otherwise, a kick pose is calculated and given to the `WalkToBallGenerator` to generate a step.

To allow some control over the step size, the `WalkingEngine` provides the two functions *generateTranslationPolygon* and *generateTranslationPolygonFast*, which are used by the generators. Given the last motion phase and the requested rotation speed, the allowed translation size for the next walking step is clipped accordingly. Here, the higher the rotation speed the lower the translation becomes. *generateTranslationPolygon* is used for the general purpose of walking from A to B, while *generateTranslationPolygonFast* is used when walking to or around the ball. Here, the translation is less reduced by the rotation speed to allow bigger steps when walking around the ball.

Initializing

The motion phase that executes the step initializes the start positions of both feet. These are the 2D-translations and one parameter for the z-rotation. The initialization results from the previous motion phase. If the previous motion phase was neither a walk nor stand phase, this is done with forward kinematics from the previously requested joint position. Otherwise, the requested end positions of the feet from the previous walk phase are used. The end positions are set by the requested step size. Afterwards, in every motion frame, the requested feet positions are interpolated between the start and end positions over the time span of the planned walk step duration, which is a static 250 ms time window.

The heights of the feet are kept static except for the swing foot, which is reduced over the first half of the walk step duration and interpolated back to zero over the second half.

Calculating base joint positions

After determining the feet translations and rotations, the corresponding joint angles are calculated based on inverse kinematics. Afterwards, the arm joint angles are set. The `MotionEngine` can force joint angles for either arm side. For a side that is not set, the `WalkingEngine` calculates a natural arm swinging motion, which moves the arms in the opposite direction of the leg on the same side. Since our robots sometimes take their arms behind their back to better avoid obstacles, the center of mass gets shifted. Therefore, a dynamic center of mass computation, which considers the impact of the arms' positions, is used to counteract the arm positions. The difference of the center of mass's projection to the ground of the calculated joint angles compared to the regular arm positions is used to tilt the torso forward with the hip pitch joints.

Step adjustment

As a new addition to our walk, we adjust the feet positions as described in cf. Section 8.3.6. This is only done when no in-walk kick is executed, as we prefer a correct kick of the ball over not falling over.

Gyro balancing

Afterwards, the lowpass-filtered measured gyroscope value is added with a factor on the ankle pitch request. The factor is different for positive and negative values because the feet are shorter at the back and stability benefits from slightly more aggressive balancing in that direction. Furthermore, the robots are balancing sideways with the ankle rolls only when they are standing.

Additionally, we added some controllers to achieve more stability in some specific situations. Here, we also only use the lowpass-filtered measured gyroscope value. While or after an in-walk kick, we add this balancing value on top of the hip pitch joints. If our walk step adjustment adjusted the swing foot by a large value forward, negative gyro values are added on top of the hip pitch joint of the supporting foot. We noticed that when a robot is tilted backwards and starts tilting forward, the hip pitch contracts and the knee pitch stretches out by a few degrees more than commanded, which lets the robot tilt more forward and potentially fall over as a result. Therefore, if our walk step adjustment adjusted the swing foot by a small value backwards or the projected center of mass was too much backward in the supporting area of the feet, we assume the robot tilted too much backward. In such a case, positive gyro values are added on top of the hip and knee pitch joints of the support foot to prevent this effect.

Foot sole rotation compensation

Last but not least we noticed a general problem with robots when they tilt forward as a result from a collision with another robot or from an uneven underground, especially when walking fast. Here, the main problem comes from the joints of the supporting foot. One of the pitch joints is stuck and cannot move against the fall direction of the robot's weight when it starts tilting forward. The other joints can keep executing the requested positions and result in the robot tilting even more forward. Therefore, the supporting foot is still parallel to the ground but the swing foot is rotated by the rotation error of the supporting foot (cf. Fig. 8.2). In the moment in which the swing foot is more forward than the support foot, it collides with the ground, resulting in an early support foot switch and the robot will tumble over shortly after. To minimize this problem, we simply use the rotation error of the support foot and add it on top of the swing foot, with an interpolation, so both feet are rotated similarly. After 75% of the walk step duration, this correction value is interpolated back to zero to ensure that the swing foot has no extra rotation at the end of the walk step. Otherwise, the robot would just keep being tilted forward and fall over even more often.

A similar effect happens in the opposite direction as one of the pitch joints might get stuck when the robot tilts too much backwards. The step adjustment might try to move the swing foot backwards which only pushes the heel in the ground. To counteract this, we use the backward orientation of the robot (cf. Section 7.1) and use this as an error that is added on top of the swing foot rotation. That way the swing sole rotates forward and is once again parallel to the ground. We interpolate the error the same as for the forward case above. In case the feet soles are calibrated[1], we use the support foot rotation instead as an error once again.

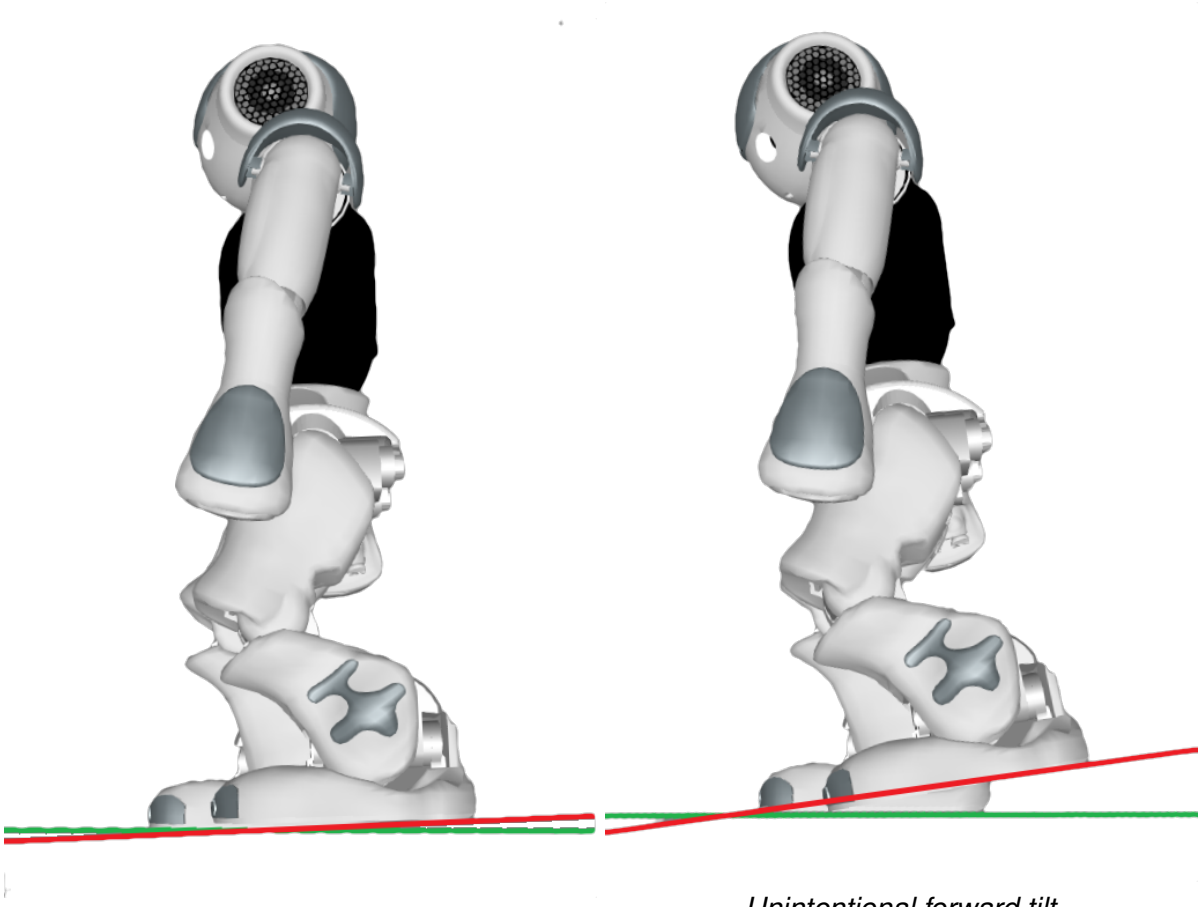
Step termination

A new step starts when the weight shifts from one leg to the other. This information comes from the representation `FootSupport`. In such a case the function *isDone* of the motion phase returns true.

8.3.3 Additional Smaller Features

If it is detected that the robot is stuck on one foot, e. g. resulting from walking against a goal post, the robot returns into a standing position. If this does not resolve the situation, an emergency step is requested, where the robot pushes the current swing leg into the ground.

[1] `https://wiki.b-human.de/Coderelease2021/getting-started/#foot-sole-rotation-calibration`



Normal

(a) The feet under normal circumstances.

Unintentional forward tilt

(b) The robot's body swings quickly forward and is already tilted a lot.

Figure 8.2: The feet rotations of a robot while walking. The right foot is the support foot, the left foot is the swing foot. The ground is shown in green. The movement direction of the swing foot is shown in red.

If the robot starts its first step, the swing height is only 50%.

For the step size clipping, a polygon is calculated in the first execution frame of the `WalkingEngine`. In case the walking speed parameters are changed in the run time, they have no influence.

When walking sideways, we shift the torso slightly in the direction of the swing foot and increase the interpolation time for the swing foot height, but keep the interpolation time of 250 ms for the translation and rotation. In previous years, we also used higher interpolation times for the translation when walking sideways, which resulted in an up to 25% slower walking speed. With these changes, the robots walk up to 270 mm/s sideways with close to no rotational deviation. At the RoboCup 2021 Passing Challenge (cf. Chapter 3), we used a commanded sideways walking speed of 300 mm/s, which resulted in an actual sideways speed of 340 mm/s. Higher sideways speeds are possible, but the use cases besides catching the ball are not enough as the joints heat up quite fast and the robot unintentionally rotates at those high speeds which results in inaccurate walking steps.

Our high stand is now also provided by the `WalkingEngine`.

The forward and sideways translations are no longer coupled by a parameter, but instead by

our inverse kinematic calculation. Both could be 100% and are only reduced, if the robot could not reach the desired position because of hardware limits.

The swing foot stretches out when the step duration was a lot longer than planned. This should prevent multiple steps afterwards that might have a longer than planned step duration.

8.3.4 In-Walk Kicks

Besides walking and standing, the `WalkingEngine` has also minor kicking capabilities. The tasks of walking and kicking are often treated separately, both solved by different approaches. In the presence of opponent robots, such a composition might waste precious time as certain transition phases between walking and kicking are necessary to ensure stability. A common sequence is to walk, stand, kick, stand, and walk again. Since direct transitions between walking and kicking modules are likely to let the robot stumble or fall over, the `WalkingEngine` can carry out simple kicks while walking.

At the moment, there are five kick types: `forward`, `forwardLong`, `sidewardsOuter`, `turnOut` and `forwardSteal`. Those kicks are described individually by the configuration file located in *Config/Robots/Default/walkKickEngine.cfg*.

A kick is configured as a quantity of relative ball positions which are converted into walking steps, all done by the `WalkKickEngine`. Therefore, every kick adjusts itself to the ball. The `WalkingEngine` then interpolates linear between these steps. For instance, if two steps are given with the first needing 40% of the step duration and the second one 60%, then the walk phase would first interpolate in the first 40% of the step duration to the first step size fully. After 40% the step size would be completely achieved and the resulting feet positions are the new starting positions. Afterwards, for the next 60% of the step duration, the walk phase interpolates to the second step. Meanwhile the swing height is calculated like in a normal walk step, without the mentioned interpolation parameters of the keyframe steps.

8.3.5 Learning Parameters Online

At past competitions, we had the problem that when the robots heated up or after they played a lot of games, they were more likely to fall. To counteract this, we added an automatic adjustment system for the gyro balancing parameters, which are used to balance with the ankle pitches. It uses the past peaks in the gyro measurements and adjusts the balance parameters by random, by increasing and decreasing them each for a few walking steps. Afterwards, the sampled gyro peaks are used to determine, which parameters resulted in a more stable walk.

In 2019, we noticed that at the moment the joints are too hot and their stiffness is reduced, the balance parameters start to increase, which furthermore results in a stable walk. With our step adjustment (cf. Section 8.3.6), this effect does no longer occur. After a few minutes of playing the parameters stabilized themselves.

8.3.6 Walk Step Adjustment

Although the two stability mechanisms of the *rUNSWift* walk, i. e. detecting the change of the support foot and gyro-based balancing, can counter a certain amount of disturbances while walking, there are situations which require to change the steps itself to prevent falls. For instance, the gyro-based balancing has its limits, in particular for preventing falls to the back, because the soles of the feet extend less to the back than to the front and therefore they cannot exert that much force against the ground to keep the body upright. In addition, the available

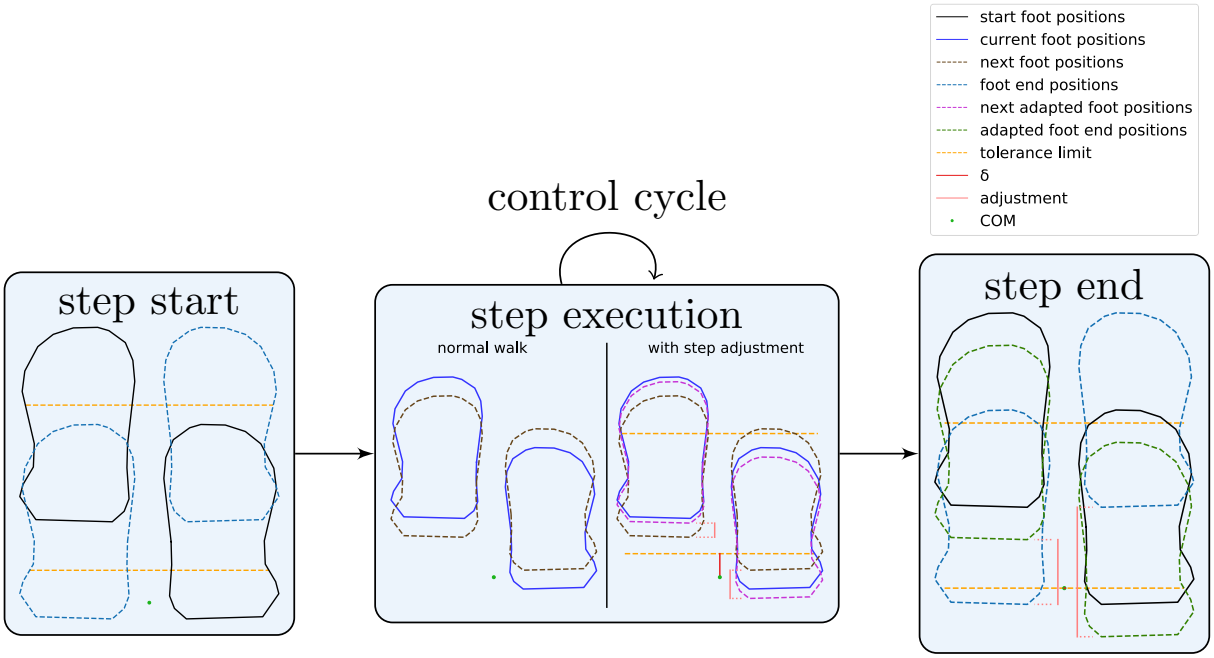


Figure 8.3: The step adjustment. The planned step is to swing the right foot forward, while the left (support) foot is moved backward. The projected center of mass is outside the tolerance area by an, in this case negative, offset δ. In each control cycle, the feet are moved closer to their planned end positions. Meanwhile, the step adjustment adds δ to the position of the swing foot and thereby moves the foot slowly backwards in the direction of the center of mass, and half of the offset is subtracted from the position of the support foot. At the end of the step, the robot effectively walked backwards for a small step. At the start of the step, the tolerance area is shown relative to the initial foot positions (dark blue). At the end of the step, it is shown relative to the new foot positions (dark green).

torque of the motors is limited, so even some swings to the front cannot be compensated although the foot might be long enough to do so. Therefore, the goal of the step adjustment is to modify the step trajectories themselves instead of just pitching the ankle joints. However, these adjustments are only applied when the robot reaches a state the original walk cannot cope with. [9]

For example in a situation in which the robot is walking forward and its body starts tilting backwards, the torso does not seem to keep up with the legs. Therefore, the step size is reduced (up to a full backward step), which allows the upper body to “catch up”. This is achieved with the approach that is shown in Figure 8.3. The center of mass (COM) of the robot is projected onto the plane of the feet. The robot is considered to be stable if the projected COM is located inside a tolerance area that is spanned by fixed regions on both feet. If the COM leaves this area, the swing foot is moved in that direction by the amount that the COM left the area and the support foot is moved in the opposite direction by half that amount, until the COM is once again inside the tolerance area. This adjustment is only applied in the sagittal direction. The projected COM is lowpass-filtered to avoid any jerky movements due to measurement noise. As shown in Figure 8.3, this can lead to a step in the opposite direction of the current walking direction. In this case, the next step will be very small, because the swing foot is already in front and the support foot is in the back. In combination, this results in two (or even more) steps on the spot that stabilize the robot, after which it can continue to walk normally.

The boundaries of the tolerance area were determined beforehand by statistically analyzing the

projected COM of one of our robots from the log files of a competition game. This tolerance area is specified in% values of the support polygon of the robot. It is assumed that the support polygon is calibrated. Otherwise, the step adjustment will adjust the feet too much in one direction and does nothing in the other direction.

To compensate for the sensor delay of up to 48 ms (i. e. four control cycles), the behavior of the COM is predicted using a LIPM [6]. Thus, the relation between the COM and the tolerance area is estimated based on the commanded joint angles and the estimated current position of the COM, reducing the reaction time significantly.

The whole step adjustment is located in the file *WalkGeneratorData*.

8.3.7 Treading on a Robot's Foot

In most robot soccer games, it is unavoidable that two robots fight for the ball. In many cases, at least one of these robots will fall over as a result of stepping on the foot of the other one. But even when robots just walk over the field, sometimes two robots will collide and one will fall.

We noticed that in most cases, the robot that fell did so because of a sudden back switch of the supporting and the swinging foot. Therefore, the believed swinging foot is actually the supporting foot, which contracts in its height. Eventually, the robot just falls from the height difference of both feet.

To counteract this problem, we try to detect such cases. If detected, the currently allowed maximum height of the swing foot, which is normally 12 mm, is set to 0. Furthermore, the next support foot switch is accepted even if not enough time has passed. If the actual next support foot switch happens at least in the first 75% of the step duration, then the next step size is partially overwritten to force a backwards step and pull both feet together in the y-axis.

In the first four motion frames of the walk phase, the detection itself calculates the height difference of the swing foot toe and the edges of the supporting foot in torso coordinates. With some threshold checks over the differences in the fourth frame and the change of difference from the first to the fourth frame, a stepping on another robot's foot is assumed.

8.4 Falling

The falling works the same as in our previous code release [10]. A fall is detected with the help of a Kalman Filter, which filters and predicts the center of mass in the support area. If the center of mass leaves the support area, then the robot will execute a fall motion based on the moving direction of the center of mass. Once the `FallEngine` activates the *fall*, it is not left until a *getUp* phase is demanded or the robot has been put up manually. To avoid a false activation, certain situations are excluded: For example, a *getUp* and *keyframeMotion* phase cannot trigger the `FallEngine`. To prevent damage to the robots, we adjusted our fall motion. If the robot falls backwards a squatting position is adopted to decrease the drop height of the head. Additionally, the arms are placed behind the back and the head is tilted to the front. If the robot falls forward, the *sitFront* motion is called, the head is placed in the neck and the arms are placed forward.

8.5 KeyframeMotionEngine

The `KeyframeMotionEngine` provides motion phases of the type *getUp* and *keyframeMotion*. Both are hardcoded motions by interpolating between predefined joint positions. The only difference is the use case of both and to let the behavior know what kind of motion phase is currently executed. Currently, all developed balancing mechanisms of the `KeyframeMotionEngine` are only used for the *getUp* motions, but could also be used for motions of the type *keyframeMotion*. All motions are defined in *Config/Robots/Default/keyframeMotionEngine.cfg*. The balancing parameters are defined in *Config/Robots/Default/keyframeMotionParameters.cfg*.

Compared to previous versions, the `KeyframeMotionEngine` is the `GetUpEngine` from our last code release [10], which now also handles the motions formerly known as `SpecialActions`.

8.6 Stability Increasement for the Kicks

We did some small quality of life changes to our kicking, which is based on [8], as we wanted to reduce the amount of needed calibration between new and old robots. We noticed that it is not hard to let the robot kick the ball far, but to make sure that the robot will not fall afterwards. Many falls result from the problem that after the ball is kicked, the kicking foot touches the ground too early and pushes the robot away, resulting in a fall. To prevent this, we measure the pressure change of both feet with the representation `FootSupport`. If a support foot switch is measured and the behavior requests to start a walk phase, we abort the kick early to start walking. The walk phase handles the stability itself afterwards. This change removed the needed calibration.

Bibliography

[1] Jan Blumenkamp, Andreas Baude, and Tim Laue. Closing the reality gap with unsupervised sim-to-real image translation. In Rachid Alami, Joydeep Biswas, Maya Cakmak, and Oliver Obst, editors, *RoboCup 2021: Robot World Cup XXIV*, volume n/a of *Lecture Notes in Artificial Intelligence*, page n/a. Springer, to appear.

[2] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) Challenges & Rule Book, 2021. Only available online: `https://cdn.robocup.org/spl/wp/2021/06/SPL-Rules-2021.pdf`.

[3] Arne Hasselbring and Andreas Baude. Soccer field boundary using convolutional neural networks. In Rachid Alami, Joydeep Biswas, Maya Cakmak, and Oliver Obst, editors, *RoboCup 2021: Robot World Cup XXIV*, volume n/a of *Lecture Notes in Artificial Intelligence*, page n/a. Springer, to appear. `https://b-human.de/downloads/publications/2021/DeepFieldBoundary.pdf`.

[4] Bernhard Hengst. rUNSWift Walk2014 report. Technical report, School of Computer Science & Engineering University of New South Wales, Sydney 2052, Australia, 2014. `http://cgi.cse.unsw.edu.au/~robocup/2014ChampionTeamPaperReports/20140930-Bernhard.Hengst-Walk2014Report.pdf`.

[5] Simon J. Julier, Jeffrey K. Uhlmann, and Hugh F. Durrant-Whyte. A New Approach for Filtering Nonlinear Systems. In *Proceedings of the American Control Conference*, volume 3, pages 1628–1632, 1995.

[6] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa. Biped walking pattern generation by using preview control of zero-moment point. In *Robotics and*

Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on, volume 2, pages 1620–1626 vol.2, Sept 2003.

[7] Tim Laue, Arne Moos, and Patrick Göttsch. Let your robot go – challenges of a decentralized remote robot competition. In *Proceedings of the 10th European Conference on Mobile Robots*, 2021.

[8] Judith Müller, Tim Laue, and Thomas Röfer. Kicking a Ball – Modeling Complex Dynamic Motions for Humanoid Robots. In Javier Ruiz del Solar, Eric Chown, and Paul G. Ploeger, editors, *RoboCup 2010: Robot Soccer World Cup XIV*, volume 6556 of *Lecture Notes in Artificial Intelligence*, pages 109–120. Springer, 2011.

[9] Philip Reichenberg and Thomas Röfer. Step adjustment for a robust humanoid walk. In Rachid Alami, Joydeep Biswas, Maya Cakmak, and Oliver Obst, editors, *RoboCup 2021: Robot World Cup XXIV*, volume n/a of *Lecture Notes in Artificial Intelligence*, page n/a. Springer, to appear.

[10] Thomas Röfer, Tim Laue, Andreas Baude, Jan Blumenkamp, Gerrit Felsch, Jan Fiedler, Arne Hasselbring, Tim Haß, Jan Oppermann, Philip Reichenberg, Nicole Schrader, and Dennis Weiß. B-Human team report and code release 2019, 2019. `https://github.com/bhuman/BHumanCodeRelease/raw/master/CodeRelease2019.pdf`.

[11] Rico Tilgner, Thomas Reinhardt, Stefan Seering, Tobias Kalbitz, Samuel Eckermann, Michael Wünsch, Florian Mewes, Tobias Jagla, Stephan Bischoff, Carolin Gümpel, Marvin Jenkel, Andreas Kluge, Tobias Wieprich, and Felix Loos. Nao-Team HTWK team research report. Technical report, Hochschule für Technik, Wirtschaft und Kultur Leipzig, 2020.