RoboCup 2024 SSL Champion TIGERs Mannheim — State-of-the-Art Dribbling and Path Planning Methods

Nicolai Ommer, André Ryll, Michael Ratzel, Mark Geiger

Department of Information Technology
Baden-Württemberg Cooperative State University,
Coblitzallee 1-9, 68163 Mannheim, Germany
info@tigers-mannheim.de
https://tigers-mannheim.de

Abstract. In 2024, TIGERs Mannheim won the RoboCup Small Size League competition in Division A for the fourth time in a row.
This paper presents our latest improvements of ball control and path planning algorithms. First, the paper will focus on state-of-the-art methods for dribbling a ball and the employed control architecture. Furthermore, many implementation details and improvements to our trajectory-sampling-based path planning are depicted. This includes generating and executing trajectories, handling dynamic obstacles, and dealing with corner cases.

1 Dribbling Methods and Control

SSL robots are allowed to control the ball by actively exerting a backspin on it. They must not remove all degrees of freedom of the ball and an area of at least 80% when viewed from above needs to be outside the robot's convex hull[1]. The part of the robot which is exerting backspin is commonly referred to as a dribbling device. These devices appeared first around 2002[2], which is only 5 years after the league was founded in 1997. Dribbling devices were mainly used with straight forward or backward movement only, as they had little lateral control of the ball. Nevertheless, this allows to pull back the ball from opponents and it is essential during ball placement procedures (introduced 2017 and mandatory in Division A since 2018).

Dribbling got more advanced in 2015 when team CMDragons introduced dribbling while turning [1]. A circular path is followed to pass around a single opponent. Because of this skill, among other things, CMDragons won the world championship in 2015. The next step of dribbling device and ball control evolution happened in 2018 when team ZJUNlict demonstrated outstanding movement abilities with the ball. Their so called *break skill* was able to move

[1] cf. SSL rules section 3.2.3.

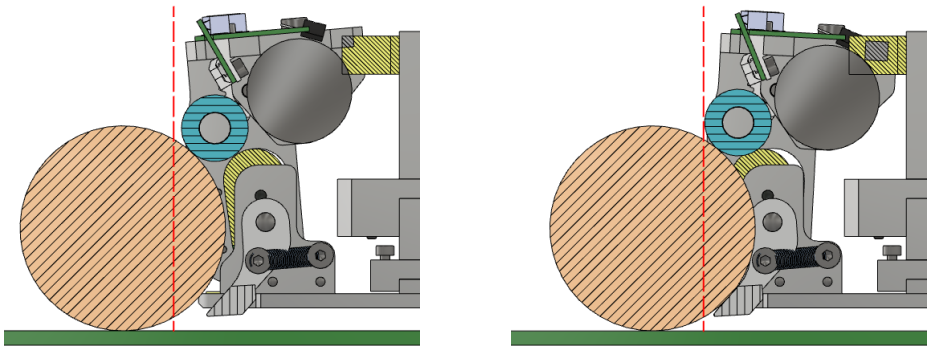
[2] Archive images of CMDragons team with dribbling device: https://www.cs.cmu.edu/~robosoccer/image-gallery/small/

with the ball on arbitrary paths, not limited to circular movement [3]. This led to two consecutive world champion titles in 2018 and 2019 for them.

Especially the achievements in recent history show the importance of a good dribbling devices and ball control to become a top team of the SSL. Section 1.1 describes our current dribbler design and two common methods to control the ball. Section 1.2 gives a detailed description of the motor controller used for successful ball handling.

1.1 Dribbler Design

Figure 1 shows the dribbler design of our current v2020 robot generation at two different states. The forward dribbler position is the default position when the robot is not in contact with the ball or when it just made contact. In this state, the ball only has two contact points: with the dribbling bar and with ground. The rearward dribbler position is limited by the chip shovel position. In this state there is an additional contact point between ball and chip shovel. Over the full range of movement the robot does not violate the coverage rule as indicated by the dashed vertical line.



(a) Forward dribbler position, inclination 5° (2-point contact).

(b) Rearward dribbler position, inclination −3° (3-point contact).

Fig. 1: Half section view of two different dribbler positions with highlighted elements being: ball (diagonal section lining), dribbling bar (horizontal section lining), chip shovel (vertical section lining), damping elements (fine diagonal section lining). The vertical dashed line indicates the ball coverage limit according to SSL rules.

High Speed and Low Speed Dribbling After the success of ZJUNlicts dribbling in 2018, we first tried to recreate their design. We observed that they are using high dribbling speeds and they employ a 3-point contact mode as seen

in Fig. 1b. In this mode, the ball is constantly pulled onto the chip shovel and drops back to ground, leading to a periodic dynamic steady state [2].

Ultimately, we failed at recreating the exact same method of dribbling. We could either have a good dribbling or a good ball reception, but not both. To achieve a 3-point contact, the dribbling bar must be at a more rearward position. In this position, we were not able to receive balls well as it hit the chip shovel during the process and bounced back from the dribbler. Then, we tried a more forward position. The problem then was to get into a 3-point contact state with high dribbling speed. If there is sufficient traction between the ball and ground and between the ball and dribbling bar we noticed a strong force pushing onto the ball. This also pushed the dribbling bar back to a 3-point contact but the required motor torque was high and reduced the ball speed. Hence, 3-point contact and high speed was not achievable.

After multiple tests with the dribbler it turned out that a low speed 2-point contact dribbling can also work very well to control the ball. A key factor is to use a torque controller for the dribbling motor (see section 1.2). By controlling the torque, we can effectively control how far the dribbler moves back and how far the ball enters into the robot. The chip shovel only acts as a final end stop and ensures the coverage rule is not violated. As torque is limited, the dribbling speed reduces as soon as the balls' friction forces exceed the torque limit. In some situations, the ball comes to a full stop. In this case we have a firm *grip* on the ball with a well controlled force.

Properties of Different Dribbling Methods The methods of dribbling described above have notable differences and have various benefits and drawbacks.

Ball Centering Achieving ball centering requires a specific spiral shape and a rotating dribbling bar. Unfortunately, our approach lacks a consistent rotating dribbling bar and lacks the required spiral shape, rendering ball centering unattainable in our case.

Dribbling Bar Wear High speeds contribute to increased wear on bearings, particularly on the dribbling bar material due to abrasion. Our deliberate use of low-speed dribbling proves advantageous in mitigating wear. Notably, we did not need to replace any dribbling bar during the 2023 World Championship.

Kicks with Backspin Kicking a ball with a backspin is a good way to improve chip kick ball control. A lobbed ball with backspin rolls less far or even rolls back. The same applies for flat kicks but in this case the behaviour is usually not desired. Fast kicks will be slowed down by a large backspin. Kicks with backspin are in general not possible with our dribbler approach.

Danger of Lost Balls In scenarios where the ball is lost during high-speed dribbling (e.g., skirmish situations), it retains considerable rotational energy, leading to unpredictable acceleration. This makes it challenging to regain ball control. Opting for low-speed dribbling proves advantageous as the ball remains closer, reducing the risk and facilitating easier ball control recovery.

Ball Control Both dribbling methods can offer good ball control after successful tuning to a given carpet. There is no common metric for dribbling performance yet which makes a scientific comparison difficult. Observations show that ZJUNlicts performance is slightly better at the moment but this may also be due to better higher-level systems (e.g., better path planning with the ball) which are not covered in this paper.

In summary, our design decisions prioritize robust dribbling and effective ball reception, accepting trade-offs in ball centering and backspin kicks.

1.2 Dribbler Motor Control

Figure 2 details the control architecture of our dribbler motor. The motor itself is a brushless DC (BLDC) motor with three hall sensors. The three motor windings are energized according to the common 6-step commutation scheme. This is the simplest mode of control for a BLDC motor and results in characteristics and mathematical model equivalent to that of a brushed DC motor. Its main drawback is a small torque ripple as the rotating magnetic field is only adjusted six time per electrical revolution.

In addition to the hall sensors there is one low-side shunt sensor for current measurements. The shunt circuit has a voltage offset to allow bidirectional measurements (accelerating and braking motor). Our dribbling motor is not equipped with an encoder.

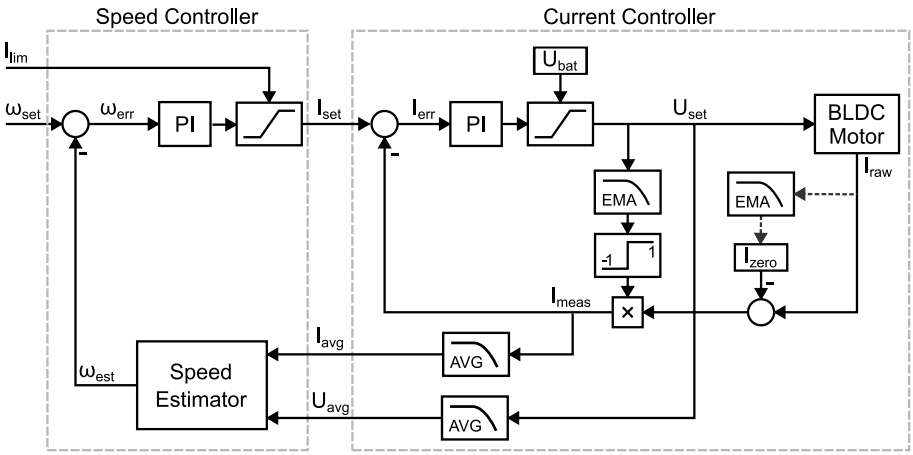


Fig. 2: Outer speed control loop (1 kHz) with estimator and inner current control loop (40 kHz) with filtering.

Architecture The motor is controlled by a cascaded architecture of an outer speed control loop and an inner current control loop. The speed loop runs at 1 kHz, the current loop at 40 kHz. The loops are implemented on a single

STSPIN32F0A microcontroller which also includes gate control logic for the inverter MOSFETs. To achieve the control frequency all computations are done with fixed-point arithmetic and without any division operations. They would be very slow, because this microcontroller does not have a floating-point unit or hardware divide instructions.

The electrical system of the motor can be modeled as

$$L \cdot \dot{I} + R \cdot I = U - K \cdot \omega \tag{1}$$

with motor inductance L, motor resistance R, back-EMF constant K, current I, speed ω, and applied voltage U.

Current Control Loop Section 1.1 emphasizes on the importance of controlling the motor torque τ for good ball handling. The motor torque is not directly measurable but it is related to the motor current I by $\tau = K \cdot I$, where K is the motor torque constant. Hence, the controller uses current as controlled unit.

The input to the current controller is a desired current I_{set} and its output is the voltage U_{set} applied to the windings via pulse-width modulation (PWM). During each PWM cycle, the current of the motor windings I_{raw} is measured. Due to the bidirectional measurement there is an offset current I_{zero} which is subtracted to get the real current reading. I_{zero} is adjusted via an exponential moving average (EMA) filter when the motor is off to account for manufacturing tolerances, aging, and temperature changes.

The motor current is measured by a single common shunt resistor to ground. This setup cannot measure the direction of current flowing within the motor windings but only its magnitude. Hence, I_{raw} will mostly be positive values, except for very short periods where current within a motor winding changes direction. For proper current control it essential to have actual positive and negative current measurements as it represents positive and negative torque.

To solve this problem, we multiply the measured current by the signum of a filtered U_{set} value. U_{set} is passed through an EMA filter to simulate the effect of the inductance. Without this filter I_{meas} would change signs immediately whenever U_{set} changes signs. This is not realistic and the reason given as follows: The step response (voltage change) of a LR circuit like the motor winding is a first-order response with a time constant of $t_c = L/R$. The step response reaches 99.3% of the target value within $5t_c$. Our motor has $L = 35\,\mu\text{H}$ and $R = 0.48\,\Omega$ resulting in $5t_c \approx 365\,\mu\text{s}$ which is larger than our control period $T_{\text{ctrl,cur}} = f_{\text{ctrl,cur}}^{-1} = 25\,\mu\text{s}$. Hence, the inductance cannot be ignored in the state estimation for this control loop.

I_{meas} and I_{set} now form the current error I_{err} for the PI controller. The PI values have been tuned to the motor to have minimum overshoot. The controller output is limited by the battery voltage U_{bat}. Although shown separately in figure 2 limiting is implemented in the controller to use anti-windup techniques for the integral part.

Speed Control Loop The input to the speed control loop is a desired speed ω_{set} and a maximum output current I_{lim} for saturation. The output is the desired current I_{set} for the inner control loop. The controller is of PI type with integrated saturation as well. The gains are tuned manually. Currently, we use a high-gain P-only setting.

A challenge for this control loop is a frequent measurement of motor speed ω. An apparent choice is the use of the hall sensors. Our motor has only one pole-pair which results in 6 pulses per revolution. An update frequency of 1 kHz requires at least 10000 rpm. This is impractical, especially since we operate at low speeds or even standstill (see section 1.1). To overcome this problem, we are using a speed estimator based on average setpoint voltage U_{avg} and average current I_{avg}. The average values are computed from block filters with 40 samples each (all samples from the current control loop within one speed control period). The motor model from equation (1) can be rearranged to

$$\omega = \left(U - R \cdot I - L \cdot \dot{I}\right) \cdot K^{-1}. \tag{2}$$

For the speed control loop we can omit the inductance because its time constant t_c is much smaller than $T_{\text{ctrl,speed}} = 1\text{ms}$. This results in

$$\omega_{est} = (U_{avg} - R \cdot I_{avg}) \cdot K^{-1}. \tag{3}$$

The speed estimator has a fixed update frequency of 1 kHz independent of the motor speed. ω_{est} and ω_{set} form ω_{err} for the speed PI controller.

It is worth noting that the speed controller output is usually saturated at I_{lim} and the actual speed does not equal the desired speed. This is intentional, as we prioritize current control over speed control to have a well defined force on the ball.

This architecture has proven to work well for controlling the ball, even at low speeds. Furthermore, it does not need an additional encoder for speed measurements. Nevertheless, a possible improvement is to use an encoder to enable field-oriented control instead of 6-step control. This will reduce torque ripple and eventually result in less vibrations during dribbling. An open task for future work is a precise controller performance analysis (e.g. bandwidth/margin, overshoot, and robustness to parameter deviations).

2 Path Planning

Path planning is a common problem not only in the SSL, but also in general robotics. A lot of work has been published about this already. In 2019, we described our novel trajectory-sampling-based path planning approach in detail [5], which we started to develop and use in 2015 already. Many teams were using an RRT-based approach back then. In 2020, team ER-Force changed their path planning from an RRT to a similar trajectory-sampling-based approach [6]. In 2023, team RobôCIn compared our solution and that of ER-Force and based their own implementation on that work [4].

Since the last publication in 2019, we have tuned the path planning implementation further. While the basic idea is still the same, much of the original implementation has changed. This chapter describes our current implementation and points out some differences that we applied since our last publication.

With our approach, we committed about 3 fouls due to collisions where our robot was significantly faster than the opponent (henceforth called crash) on average per match during RoboCup 2023[3] and only 1.8 fouls during RoboCup 2024.

2.1 Overview

The following sections describe our current implementation in detail. The code can be found in our open-source release.

Path planning is initiated by a specific skill, the *MoveToSkill*. It is performed individually and independently for each robot with the following steps:

1. Generate a list of obstacles: Each robot may have individual requirements for path planning. A supporter wants to keep a larger distance to the ball, while an attacker needs to touch it. A defender wants to respect opponents in a more defensive/aggressive way than other robots. Details on how obstacles are modeled can be found in section 2.5.

2. Create and adapt the path finder input: The input to the path finder consists of the robot's current 2D position and velocity, the robot's movement constraints, like maximum velocity and acceleration, the list of obstacles from the previous step and the target position.

3. Find a path: The path finder will return a path finder result, consisting of the trajectory to be executed and a list of collisions along its path. Collisions consist of the obstacle and the first collision time. Sections 2.2 and 2.3 describe the process of generating trajectories and finding collisions along the path and section 2.4 describes how resulting path finder results are accepted or rejected.

4. Execute the trajectory: Based on the returned path finder result, the *MoveToSkill* will either execute the trajectory, or perform a fast brake, if a collision is imminent. This is outlined in section 2.6.

2.2 Generate Trajectories

The basic approach is straightforward: First, create a trajectory from the current state to the target position. If its path is acceptable, return it. Else, iterate through a finite list of intermediate targets. For each intermediate target, create a trajectory from the current state to the intermediate target. Next, step over the resulting trajectory in discrete time steps $t_{\text{intermediate}} = 200\,ms$ and create a new trajectory consisting of two parts. The first part is the intermediate trajectory with an upper time limit of $t_{\text{intermediate}}$, the second part is a new trajectory with the initial state of the intermediate trajectory at $t_{\text{intermediate}}$ and the final target position. So far, we never created more than one intermediate target for

[3] https://ssl.robocup.org/match-statistics/

our trajectories. This was sufficient for our use cases. However, it would be an option if more fine-grained paths are needed.

The way of generating the intermediate targets has not changed much in recent years. We still have two different implementations. One that generates a finite list of random targets and one that generates a systematic list. Currently, we use the random-based implementation with five targets. Having such a small number of intermediate targets is acceptable, because we run path planning continuously every frame (about every 10 ms) and can thus eventually find a good path. Additionally, the target from the previous iteration is also added to the intermediate targets. This caches good results and stabilizes the resulting path over time.

Initially, we always calculated paths for all intermediate targets and selected the best one. Now, we choose the first acceptable path and stop. This reduces the overall runtime at the cost of not finding a potentially faster path. With this lazy approach, the order of the intermediate targets has become relevant, so for the random-based implementation, we sort the targets by the angle between current position to target and current position to intermediate target, smallest first. This favors paths pointing towards the target.

2.3 Finding Collisions Along a Trajectory

We apply a simple iterative approach, but also added several optimizations. Basically, we step over a given trajectory and for each time step t, we iterate over all obstacles to check if there is a collision.

In the initial implementation, we used fixed time steps and a binary collision result (colliding or not colliding). In the current implementation, the time steps are dynamic and the obstacles return the distance to the currently checked point. This enabled two improvements. First, the next time step can now be calculated based on how close the closest obstacle is. This can reduce the total number of collision checks significantly. Second, we can apply an extra margin between obstacle and robot. We chose a dynamic margin depending on the robot velocity v_t at the given time step, defined by

$$m = \left(\frac{\min\left(v_{\max}, v_t\right)}{v_{\max}} \right)^2 \cdot m_{\text{base}} \tag{4}$$

with $v_{\max} = 3\,\text{m/s}$ and margin $m_{\text{base}} = 0.2\,\text{m}$. With this dynamic margin a robot can drive very close to an opponent robot, when slow, but considers an appropriate safety distance when driving fast.

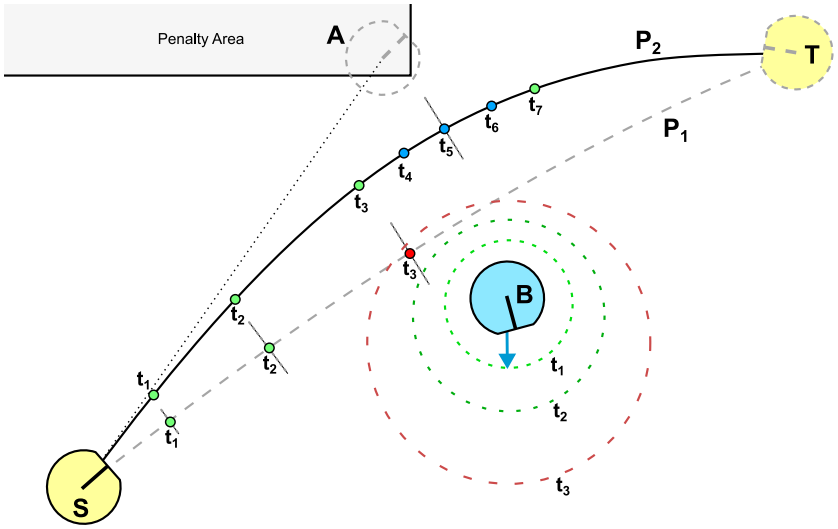


Fig. 3: The path planning situation visualizes how a collision check is performed. The path is planned from start S to target T with an intermediate target A, avoiding Obstacle B, which is moving slightly down. Path P_1 has a collision at time t_3 and path P_2 is free of collisions. The dynamic obstacle size is illustrated through the dashed circles at times t_1 to t_3, which correspond to the collision check points along the paths. The dynamic margin of the robot is depicted as an orthogonal line for some of the collision points.

In figure 3, the collision check procedure is depicted. Every dot on the paths is one collision check for one obstacle. Path P_1 is checked first and at time t_3, the collision check determines that the point is inside obstacle B and will stop. The second path P_2 is tried and this time, there are no collisions. One can also see, that the time steps t_4 to t_6 are closer to each other. The step size is smaller here, because they are near the penalty area, a close obstacle.

Another optimization that was introduced recently is the separation of moving obstacles from motionless obstacles, as they have some distinct differences. A motionless obstacle, like the penalty area, is much simpler than an opponent robot that moves in an unpredictable way. There are two main reasons to separate those two kinds of obstacles. First, motionless obstacles are always checked before moving obstacles. If there is a collision with a motionless obstacle, the path is not accepted and the moving obstacles are not checked. Second, we have different implementations for accepting a path with a collision. They are explained in the next section.

2.4 Accepting or Rejecting Path Finder Results

Matches in the Small Size League are very fast and competitive. The best path is not necessarily the fastest one without a collision. We cannot be too careful,

because then the opponent may catch the ball before us. Most obstacles are moving robots that can change their direction any time, so the situation has to be reevaluated every frame. Caring about collisions which happen one second in the future are just not worth to consider. However, this is not true in general. There are some static, motionless obstacles like the field boundary and the penalty area. On the other hand, moving obstacles can also be categorized into different behaviors. For our own robots, we know very well, where they will move along. Movement of opponent robots has to be estimated, while avoiding to be pushed or otherwise influenced by the opponents.

Our solution to those specifics are two acceptors, one for moving obstacles and one for motionless obstacles. They are applied to a path finder result and their output is binary: accepted or rejected.

Since last RoboCup (2023), we tuned the moving obstacle acceptor to more reliably intercept a ball before the opponent. Previously, our own robots got too nervous about a crash with the opponent ball interceptor and thus lost the ball against it. The optimization has been successfully evaluated during RoboCup 2024.

2.5 Modeling Obstacles

In practice, an obstacle is not just a simple geometric shape. That's why we have several different implementations for different kinds of obstacles. Some consider the current time on the trajectory to account for moving objects. For the field border and penalty area, we have specific obstacles for their geometric shape. Our own robots are modeled by their current trajectory. If they have no trajectory (because they perform some special skill, or because they are broken/off), they are instead modeled by a simple tube shape, which is constructed by position and velocity of the robot. Opponent robots have two different obstacle types that depends on the desired aggressiveness. If we want to be more aggressive, we only assume constant velocity over a short time window, otherwise we expect that robots will accelerate up to a certain maximum velocity.

For that, we calculate a circle r_{dyn} depending on time t that defines the area in which the robot could potentially move within this time based on its movement limits. Figure 3 visualizes such an opponent obstacle for multiple time steps. The radius $r_{\text{obstacle}}$and circle center $\boldsymbol{p}_{\text{obstacle}}$is calculated with equations (5)-(8).

$$t_c = \begin{cases} 0 & \text{for } t < 0 \\ t_{\max} & \text{for } t > t_{\max} \\ t & \text{otherwise} \end{cases} \tag{5}$$

$$r_{\text{dyn}} = \frac{|f_+(t_c) - f_-(t_c)|}{2} \tag{6}$$

$$\boldsymbol{p}_{\text{obstacle}} = \boldsymbol{p}_{op} + [\boldsymbol{n}_{\text{op}} \cdot (f_-(t_c) + r_{\text{dyn}})] \tag{7}$$

$$r_{\text{obstacle}} = r_{\text{op}} + r_{\text{dyn}} \tag{8}$$

f_+ and f_- are the 1D bang-bang trajectories from position zero to positive and negative infinity with the current opponent speed and its velocity and acceleration limits, $\boldsymbol{p}_{\text{op}}$ is the current opponent position, $\boldsymbol{n}_{\text{op}}$ is the current opponent normalized move direction and $r_{\text{op}} = 0.09\,\text{m}$ is the robot radius. The limits can be adapted to individual opponents, but are usually around $3\,\text{m/s}$ and $3\,\text{m/s}^2$. The maximum time $t_{\max}$ is set to about $0.5\,\text{s}$, which allows for enough lookahead to react to direction changes of opponents. A higher maximum time results in larger obstacles and thus longer paths. A small tweak is applied to this obstacle to make it less pessimistic. Instead of a circle, only a tube is created. The width of the tube is the robot width and the length is the circle radius. Additionally, for non-moving robots, the dynamic radius is set to zero, effectively reducing the obstacle to a small circle. Without this tweak, our robots would circumvent opponents widely, as they expect them to accelerate in any direction soon.

The ball is modeled by its ball trajectory. If it would simply be modeled with a tube from start to the point where it will stop, this would create a wall across the field that robots could not cross easily, so modeling the ball by time is quite important.

For ball placement, a virtual ball is modeled, as if it would be kicked to the placement position now. This also avoids the wall issue.

2.6 Executing a Trajectory

Executing the trajectory is generally straightforward. Our robots require a target position as input. We take the next (intermediate or final) target position from the trajectory and send it to the robot every frame.

However, before the trajectory is applied, the *MoveToSkill* will check if it is safe to do so. The trajectory may have a collision and reports the time until the first collision will happen. The skill will calculate if it can reduce the robot velocity to a safe amount before hitting the obstacle. If it is still safe, the path is applied. Otherwise, the skill brakes until it is safe again. The skill will use a larger acceleration for braking, than the path planning considers. This makes it more likely to actually brake in time.

Incorporating the emergency brake feature into the skill offers several advantages. Our current bang-bang trajectories lack the ability to differentiate between acceleration and braking, yet the robots demonstrate a significantly faster braking capability compared to acceleration. Consequently, addressing this within path planning proves challenging. Additionally, planning a path with a reduced brake acceleration enables us to account for inaccuracies and devise a more risky trajectory.

2.7 Summary

Since we switched from an RRT-based approach to our novel trajectory-sampling-based approach in 2015, we improved our path planning performance continuously. Recently, other teams, like ER-Force and RobôCIn, adopted our approach as well and reported good results. While the basic idea is simple, there are a

lot of options in detail. We showed how to handle different kinds of obstacles accordingly and how to assess and accept collisions in order to not being pushed too much by opponent robots.

We are still using the same bang-bang trajectories that we introduced back in 2015, mainly because they are really simple and fast and can also easily be implemented and reproduced on the robots. This held us back from switching to another version that also supports non-zero end-velocities, like other teams did.

There is at least one major disadvantage of our current approach. We have tuned it to return an acceptable path fast and optimize it over time (over multiple frames). Also, we only use one intermediate target, so longer paths might be rather rough at the beginning. For path planning itself this is usually not an issue, as the path will adapt throughout the journey. However, the exact path and thus the estimated total time can change a lot, which makes it less useful for the AI, for example to estimate which robot can intercept a ball the fastest. For that reason, we generate simple trajectories without path planning in all estimations, like for checking which robot can intercept the ball best.

References

1. Biswas, J., Cooksey, P., Klee, S., Mendoza, J., Wang, R., Zhu, D., Veloso, M.: CMDragons 2015 Extended Team Description (2015)
2. Huang, Z., Chen, L., Li, J., Wang, Y., Chen, Z., Wen, L., Gu, J., Hu, P., Xiong, R.: ZJUNlict Extended Team Description Paper for RoboCup 2019 (2019)
3. Huang, Z., Zhang, H., Guo, D., Jia, S., Fang, X., Chen, Z., Wang, Y., Hu, P., Wen, L., Chen, L., Li, Z., Xiong, R.: ZJUNlict Extended Team Description Paper for Robocup 2020 (2020)
4. Oliveira, A., Gomes, C., Silva, C., Alves, C., Souza, D., Xavier, D.: RoboCIn — Extended Team Description for RoboCup 2023 (2023)
5. Ommer, N., Ryll, A., Geiger, M.: TIGERs Mannheim — Extended Team Description for RoboCup 2019 (2019)
6. Wendler, A., Heineken, T.: ER-Force — Extended Team Description for RoboCup 2020 (2020)