EE4306: Distributed Autonomous Robotic Systems

Robot Soccer Final Project

L
U
K
R A
TEAM **ROS**
Y M
A A
N I
N

Lukas Brunke A0149064A

Romain Chiappinelli A0145192H

Ryan Louie A0149643X

16 April 2016

Introduction

This paper details our work on the final project of the module EE4306 Distributed Autonomous Robotic Systems. In this project we needed to control differential drive robots playing football on the Coppelia V-Rep simulation platform. The environment, scene and robot model for the simulator are provided.

We needed demonstrate three stages:

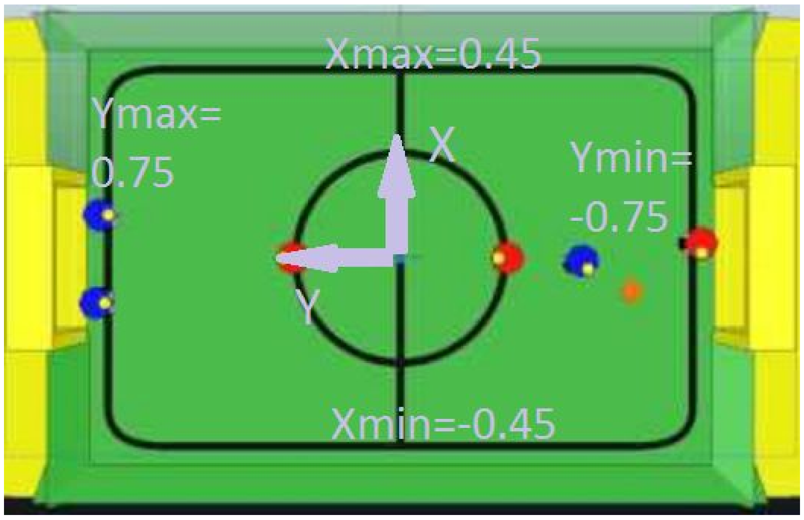
- Penalty kick (goal keeper side)
- Skills and drills
    - Dribbling
    - Passing and Shooting
- Competitions (match against another team)

The objective of this report is to detail our approach for these three stages, explain the algorithms that underly the planning, control, and coordination of robot football player, and highlight the positives and negatives of our solution.

Allowing ourselves bragging rights, we were delighted to be the only team in our class to kick the game winning shot into the goal! This result is a testament partly to the techniques we present in this paper, but also to the persistence and commitment of this (human) team of young engineers. We credit each other's flexibility to use new tools and languages to collaborate technically; iterating on ways of dividing our work while making the fruits of our efforts reusable; and maintaining a strong camaraderie for the duration of the project. So with that in mind, let us learn about the inner workings of a class-champion robot football team!

Reference Frame

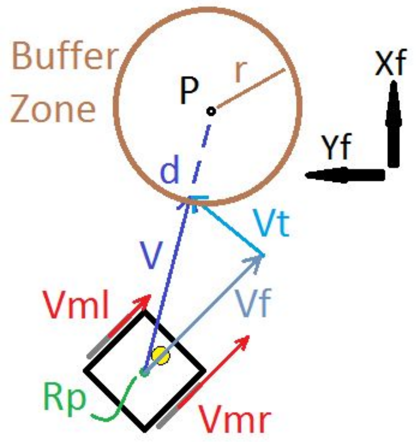
We measured the reference frame and field dimensions by moving a robot at the four corners of the football field. We obtain the following results:
X is the vertical axis pointing up, X is in the range -0.45 to 0.45;
Y is the horizontal axis pointing left, Y is in the range -0.75 to 0.75.



Base Robot Football Player Functionality

Before discussing the task specific solutions to the three project stages, we will cover the set of methods that is basic to the functionality for all differential-drive robot football players, regardless of their position as a goalie, defender, or striker.

Velocity to position: ‘V2Pos()’



This function is the first brick of our path execution. Knowing the current position and orientation of the robot we want to set the velocities of the right motor and the left motor in order to reach to position P after multiple calls of this function.

Lets define the frame of the field (Xf, Yf), and the frame of the robot (Xr along Vf and Yr along Vt). We can easily find the distance d=||P-Rp|| in the field frame. Then if d is smaller than r (the radius of the buffer zone), we consider to have reached the point P.
If not, we first normalize d and multiply it by the desired motor velocity Vm:

$$\vec{V} = Vm \cdot \vec{d}/d$$

Then we need to find v in the robot frame. The robot orientation is given by an angle α between Xr (=Vf) and Xf. Vf and Vt can be found by the relation:

$$^{R}\vec{V} = \begin{bmatrix} Vf \\ Vt \end{bmatrix} = \begin{bmatrix} -sin(\alpha) & cos(\alpha) \\ -cos(\alpha) & -sin(\alpha) \end{bmatrix} {}^{F}\vec{V}$$

Finally we can set the motor velocities left and right by this relation (given a rotation gain KR):

$$Vm_L = Vf - K_R Vt$$

$$Vm_R = Vf + K_R Vt$$

We also implement a variant of this function ‘V2PosP()’ used by the goalkeeper. This robot use a P controller and is able to go backward in order to save time during his motions.

The P controller is achieved when we calculate V from the motor velocity Vm. Instead we want this velocity to depends on the remaining distance d to the objective point P. We use a linear relationship with the proportional gain Kp. This Velocity is also bounded by Vmax:

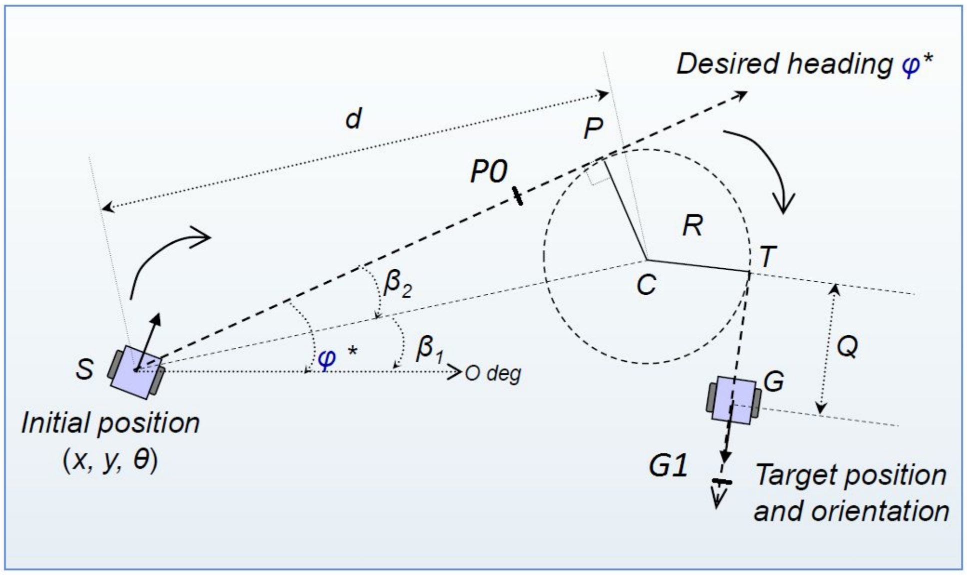
$$\vec{V} = min[K_p\vec{d}, Vmax \cdot \vec{d}/d]$$

For the backward implementation, we can achieve it by changing the sign of the translational velocity Vt if the vector V point backward of the robot (which is equivalent to say that the forward velocity Vf is smaller to 0).

$$\text{if } Vf < 0\text{:}\ \ Vt \cdot (-1)$$

Pass Path

This function create a Path to ensure reaching the position G with an orientation Φ(=φ*). This path is represented by a list of points.
We need to know the radius of curvature R, the distance Q, the initial configuration S (position and orientation) and the desired configuration G.



We start by creating the point T, then find the center C, but we do not know on which side it will be, so we calculate booth center C1 and C2 and we pick C the be the closest to S:

$$\vec{T} = \vec{G} - Q\begin{bmatrix} cos(\phi) \\ sin(\phi) \end{bmatrix} \qquad \vec{C}_{1,2} = \vec{T} \pm R\begin{bmatrix} -sin(\phi) \\ cos(\phi) \end{bmatrix}$$

If C1 is the closest, we know we are describing an anticlockwise trajectory, we set the angle γ to be -**π/2**. Else, C2 is the closest to S, so we are describing a clockwise trajectory: γ = **π/2**

Now a problem can appear is the robot is inside the circle of center C and radius R. It is not possible to find a tangente. In this case, the path will be only the two points T and G. If not, we can find the tangent point P by:

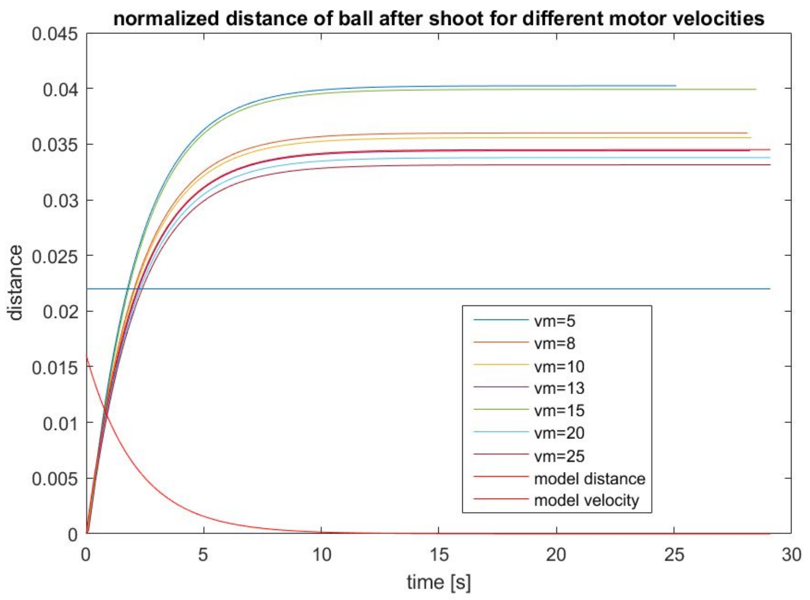
$$\vec{P} = \vec{C} + R\begin{bmatrix} cos(\beta_1 + \beta_2 + \gamma) \\ sin(\beta_1 + \beta_2 + \gamma) \end{bmatrix} \qquad \begin{aligned} \beta_1 &= atan2(\vec{C} - \vec{S}) \\ \beta_2 &= sin^{-1}\Big(\frac{R}{||\vec{S} - \vec{C}||}\Big) sgn(\gamma) \end{aligned}$$

We can finally sample some point on the circle between P and T (not described here) and put all theses points on a list which will be our trajectory.

In fact our list contains one more dimension. For each node, we have a corresponding velocity. It allows us to set the maximal velocity between the points S and P, then a slower velocity to describe the curve between P and T and finally a defined velocity Vd between T and G to kick the ball to a desired position. To determine this velocity Vd, we need to have a model for the ball...

Ball Model

The procedure to find a model, is to sample some data and find a good relationship to fit these data. To collect the data, we kick the ball at different velocities in a straight line and measure the distance of the ball with respect to its initial position. We then need to rescale the time, to ensure the kick happens a t=0 and we also divide all the distance points by the velocity of the kick (since we assume the system to be linear).



We obtain this plot ‘distance vs time’ (available in the Matlab script ‘ballModel.m’). It looks like a first order system, so we set the equation to be:

$$V_m K\left(1 - e^{-t/T}\right)$$

And from the plot we extrapol the gain K and the time constant T. K=0.0345 and T=2.15 [s].

So we learn from the model:

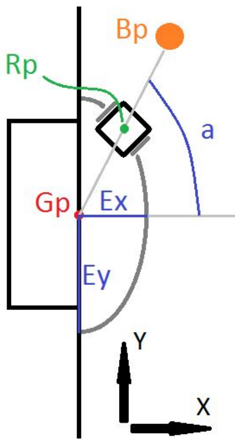
- the time for the ball to stop after a kick does not depend on the velocity of the kick but is a constant, approximately 4T=8.6 [s].
- to kick to a distance d (to pass to another robot for example), we need to set the motor velocity Vm=d/K. This relation is used for the last point on the path if we want to pass.

Penalty Shot: Goalie Strategy

We tried different methods for the goalie, the first one was based on the velocity of the ball and we keep the goalie on a straight line parallel to the goal (// to X frame axis). But the goalie got sometimes a wheel blocked on the goal since the goal is a bit lower than the field. So we opted to keep the goalie on an elliptic trajectory. This allows it to stay away from the goal and avoid to get blocked in. We also choose the goalie position to be based only on the ball position.

So the goalie should stay on this ellipse between the center of the goal Gp and the ball position Bp. To find the desired robot position, we first need to find the angle 'a':

$$a = atan2(Bp_y, Gp_x - Bp_x)$$



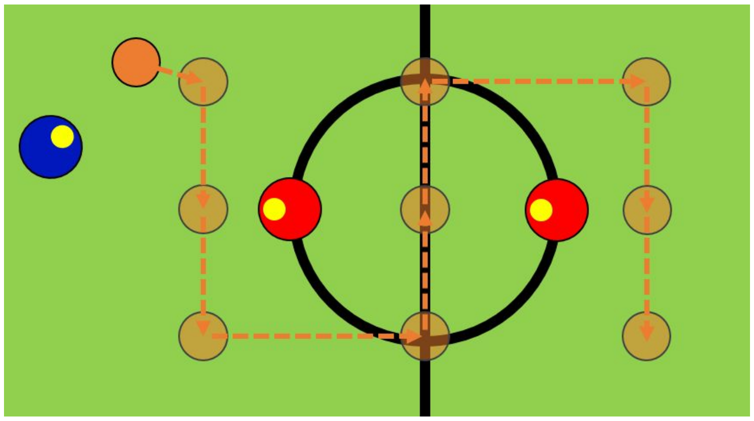
Then Rp is given by the ellipse equation:

$$\vec{Rp} = \begin{bmatrix} Gp_x + E_x cos(a) \\ E_y sin(a) \end{bmatrix}$$

We can specify which goal to keep simply by changing Gp (reusable code for red and blue team).

Then the goalie is controlled to reach the desired Rp with the P controller with backward implementation 'V2PosP'.

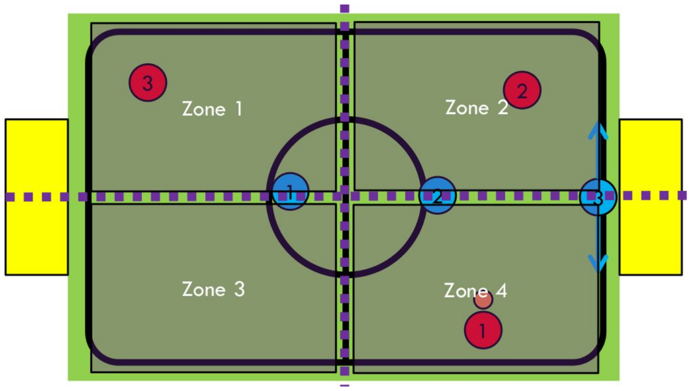
Skills and Drills: Dribbling



In order to complete the dribbling task we predefined the path the ball should take, which can be seen in the above figure. At first the ball has to be taken to the starting position of the dribbling course from its initial random position; the starting position is always the same one, no matter if the ball spawns in the top or bottom half of the field. This is done by using the function `passPath()`, which is also used for kicking the ball into all the other positions. To increase the overall accuracy of the kicks, the robot always waits until the ball has reached its rest position before it kicks the ball into the next position. Unfortunately, it is not always possible to successfully complete this drill, e.g. when the ball gets spawned at a position, which is very close to the boundaries of the field, then the robot can't kick the ball back into the field.

Skills and Drills: Zone Passing

With our dribbling strategy, the balls end location is always Zone 4. We used this predictability to our advantage in developing a hard coded zone passing strategy. Starting in Zone 4, our players make 5 passes in a counterclockwise direction. Players also have their roles preassigned. Our explicit zone order and robot assignment is described in the table below.



Zone 4	Zone 2	Zone 1	Zone 3	Zone 4	Zone 2
Bot 1	Bot 2	Bot 3	Bot 1	Bot 2	Bot 3

.
We start with an index at the beginning of the zone passing plan, which we denote the *active index* and which represents the leftmost column of the table. The *active index* keeps of 1) the *active zone* in which the ball is located and 2) the *active player* responsible for passing it to the next zone. We wait for the ball to come to rest before evaluating if it has been moved successfully to the next zone; in addition, a stationary ball is more convenient to kick than a moving one! Below is some pseudocode that describes the main loop of `zone_passer.py`:

```
active_index = 0 # initialize active index to the beginning
executing = False # start player at rest, with no path to execute
While Time Still Remaining:
        active_zone = zone_pass_plan[active_index]
        active_bot = bot_plan[active_index]
        If not executing:
                If active_index equal 5: # last index
                        kickingDestination = opposingGoalShootingTarget
                else:
                        kickingDestination = zone_pass_plan[active_index+1]
                active_bot.passPath(ballStart, kickingDestination)
        active_bot.followPath()
```

```
If ball has moved from original position and now came to rest:
    executing = True
    If ball has moved to next zone:
        active_index += 1
```

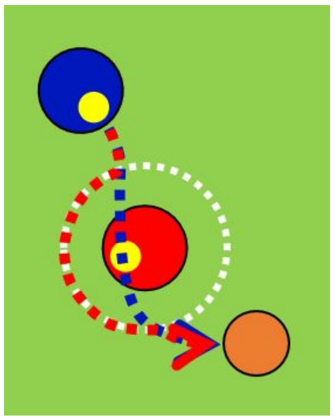
The careful reader can observe several artifacts of this design. The *active robot* responsible for kicking the ball to the next zone will be very persistent; the robot can take multiple attempts at kicking the ball until it has come to rest in the correct next zone. While persistence does work, it also means that a player might make a valid pass but continue to follow the rigid passing plan (a valid pass is defined as crossing into a zone different from the current zone and the zone before that).)Moreover, the path is calculated and executed for the active player; while there are three players on the field, a robot moves one at a time until it is their turn.

The zone passer code was effective, under the assumption that the passes were very accurate. However, due to variations in the the contact with the ball, passes had a tendency to place the ball in awkward situations for the robot:

- Ball located too near the field boundary
- Ball located behind an opposing robot, resulting in a trajectory that collides with obstacles

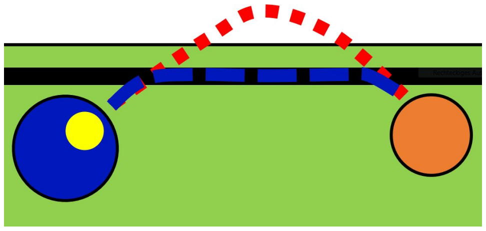
Using the vanilla versions of the pass path planner and follow path controller was not robust enough to handle these situations. It necessitated the creation of environment-aware path planning algorithms. These new methods were designed to complement our existing `passPath` and `v2Pos` methods, while providing solutions to the problem of field boundaries and obstacles during play.

Obstacle Aware Path: a solution for obstacle avoidance

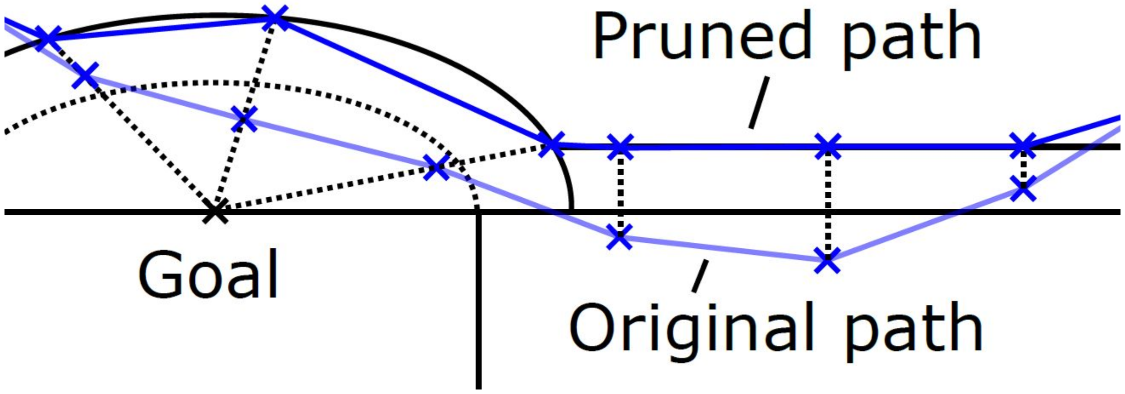


We realized that obstacle avoidance was necessary. Inspired by the static obstacle avoidance presented in class we developed a function, which would recalculate the path and make the robot go around any obstacles, which include every robot player on the field. This is done by checking if any point in the path is too close to an obstacle by calculating the distance *d* from every path point to every obstacle. Once the distance is smaller than a defined minimal distance *r*, we know that there is an obstacle in our planned path. Every point, which is too close then gets moved to a point on a circle with the radius of the minimal distance using the angle γ of the path point to the center of the circle.

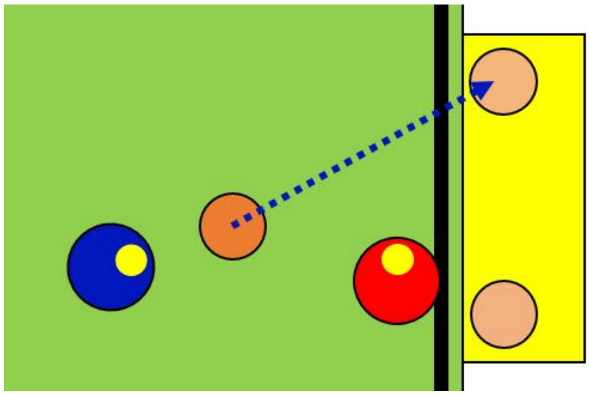
Pruned Path: a solution to field boundary issues



In some cases it is not possible to plan a valid path in order to reach the ball at a certain angle, e.g. when the path is planned outside of the boundaries of the field or too close to them. In these cases we replan the path to be a straight line parallel to the specific boundary, but with an offset, so that the robot is able to reach that position. Of course the robot is not able to kick the ball very accurately, but at least the robot does not get stuck on the wall. We also consider another special case: When the path is planned very closely to our own goal, it is possible that the goalie is considered as an obstacle. It is very likely that the obstacle avoidance plans a path around the goalie in that way, that the obstacle aware path is outside of the boundaries of the field. Therefore, pruning the path as before will just result in our robot moving into our own goalie. We avoid this by using a second extended (by the diameter of the robot) ellipsis around the goal center. Whenever a path is planned through the area of your own goal we prune the path to be on the outside of this ellipsis.



Shooting and Aiming

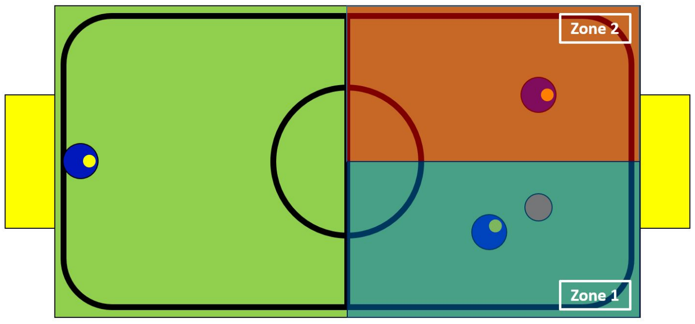


Scoring a goal is the essential part of playing soccer. In order to increase our chances of scoring we keep track of the opponent's goalie. Consequently, we are able to choose the side of the goal where we want to kick the ball in. Since it is not possible to exactly know which player of the opponent's team is going to be the goalie or if the opponent's players switch roles, we determine the opponent's goalie as the player which is closest to the center of the goal. By knowing the x-Values of the goal posts and goalie position, we determine if the upper or lower gaps are bigger. If the upper gap is bigger, we will aim at a position close to the upper goal post, but not too close so that the ball can fit. If the lower gap is bigger, we will aim at a position close to the lower goal post. Once we found the position we shoot the ball by using `passPath()`, but using the maximum velocity, which makes defending the goal more difficult for the opponent.

Match Play Competition

For the match we distinguish two basic cases: The ball is located on our side of the field and the ball is located on the opponent's side of the field. While one of the players is always behaving as a goalie, the roles of the two remaining players can vary over the course of the game, and therefore, result in an offensive and a defensive strategy.

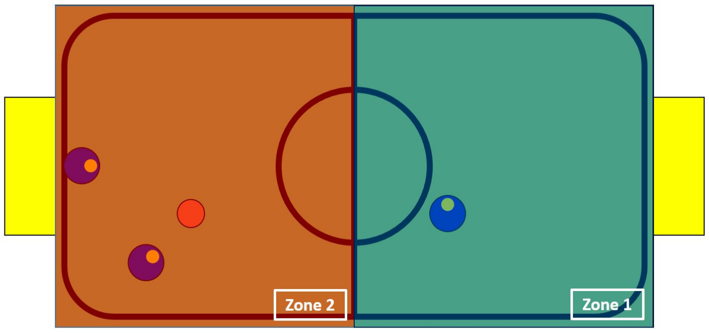
Offensive Strategy



For our offensive strategy we set our two field players to become attackers. The opponent's field side gets divided into two zones by the y-axis. The zone in which the ball is currently located becomes the active zone. According to the ball position this can change. Also there is an active attacker and a passive attacker. The robot, which is closest to the ball becomes the active attacker and executes the primary `robotCode()`, the other robot is passive and runs its `secondaryCode()`. Once the ball crosses the line of the other zone or becomes closer to another robot the active zone and player can change dynamically, respectively. This makes the offensive gameplay much more effective.

- Active attacker: is aiming the ball at the goal and tries to score. If the robot isn't successful it will try again, until it has scored.
- Passive attacker: the original `secondaryCode()` is following the y-position of the ball and is ready to take over the ball once it is closer to it. A second variant of the `secondaryCode()` makes the passive robot try to block the goalie by always moving to the position of the opponent's goalie.

Defensive Strategy



For the our three players, we set the roles to be attacker, midfielder, and goalie. Since the ball is on our side, the midfielder becomes the *active bot* and runs its primary `robotCode();` the attacker, no longer *active bot,* runs its `secondaryCode()`. The goalie continues to operate independently as normal.

- Midfielder `robotCode()`: attempt to kick the ball to the opponent's side of the field, so our strategy can turn offensive. Like some great sports player once said, "Offence is the best defence."
- Attacker `secondaryCode()`: stay on the opposite side to prevent the punishment. Our first iteration of this function was to move to a static (x,y) location on the opposing side. The final version had the robot move to a fixed y-position that prevents the punishment, but a dynamic x-position that is equal to the x-position of the ball. By following this policy, the attacker is at the closest position to the ball when it crosses the midfield line, ready to take possession and play the offensive strategy.

Conclusion

We were tasked with implementing fundamental individual and coordinated behaviours to complete three stages of robot football: 1) defending the goal during a penalty kick, 2) controlling the ball by completing dribbling and passing drills, and 3) reacting to unpredictable ball behaviour by planning different offensive and defensive behaviours in a match.

In the course of implementing these stages, our first iteration approaches often revealed unforeseen weaknesses. in the overall approach and solution. The final goalie implementation sweeps away the ball to the sides, after realising that a passive goalie only blocks but does not remove the ball from vulnerable areas; getting stuck on walls and obstacles motivated the methods like pruned path and obstacle aware path; the two winged attack strategy developed after realising that one player alone could hardly score a goal.
The development of our robot football team was not a mere implementation of an approach we envisioned from the start. We started with simple approaches, and only after battle testing them and concluding they were not up to par did we go forward with more sophisticated solutions.