CodeBall 2018

Rules

Version 1.1.0

December 2018 — January 2019

Contents

Chapter 1

About CodeBall 2018 world

1.1 General concept of the game and the rules of the tournament

This competition gives you an opportunity to test your programming skills, by creating an artificial intelligence (strategy) controlling a team of robots in a special world (you can learn about details of the CodeBall 2018 world in later sections). In different stages of the contest your team will consist of 2 or 3 robots, and using nitro may or may not be available. All robots have same parameters and are spheres, initial location is guaranteed to be symmetrical. There is also a ball in the game.

In each game you are to compete against another player's strategy. You team's goal — is to score the ball into opponent's net and defend your own. Team who has scored most goals is the winner. Game can also be tied if both teams scored same number of goals.

The championship is held in several stages

The tournament is held in several stages preceded by a qualification in the Sandbox. Sandbox is a competition that takes place throughout the championship. The player has a certain rating value – an indicator of how successful his strategy is involved in games within each stage.

The initial value of the rating in the Sandbox is 1200. At the end of the game this value can both be increased and decreased. At the same time victory over a weak (with a low rating) opponent gives a small increase, also the defeat from a strong opponent slightly decreases your rating. Over time the rating in the Sandbox becomes more and more inert, which makes it possible to decrease the impact of random long series of victories or defeats on the participant's place, but at the same time makes it difficult to change his position with a significant improvement in strategy. To cancel such effect the participant can reset the variability of the rating to the initial state when sending a new strategy, including the corresponding option. If the new strategy is adopted, the rating system of the participant will fall dramatically after the next game in the Sandbox, however, further participation in games will quickly recover and even become higher if your strategy has really become more effective. It is not recommended to use this option with minor, incremental improvements to your strategy, as well as in cases where a new strategy insufficiently tested and the effect of changes in it is not known reliably.

The initial value of the rating at each main stage of the tournament is 0. For each game the participant receives a certain number of rating points depending on the occupied place (a system similar to that used in the championship "Formula-1"). If two or more participants share some place, then the total number of rating points for this place and for the following `number_of_such_members` − 1 of places is shared equally among these participants. For example, if two participants share the first place, then each of them will receive half of the rating points number for the first and second places. When sharing rounding always takes place in a smaller direction. More detailed information about the stages of the tournament will be provided in the announcements on the project website.

First all participants can participate only in the games that take place in the Sandbox. Players can send their strategies to the Sandbox, and the last one taken from them is taken by the system for participation in qualifying games. Each player participates in approximately one qualifying game for an hour. The jury reserves the right to change this interval based on the throughput of the testing system, but for the majority of participants it remains constant. There are a number of criteria by which the interval of participation in qualifying games can be increased for a specific player. For every N-th full week that has elapsed since the player sent the last strategy, the interval of participation for this player is increased by N basic test intervals. Only the strategies adopted by the system are taken into account. An additional penalty which is equal to 20% from the basic testing interval is charged in the Sandbox for each strategy "crash" in 10 last games. More details about the causes of the strategy "crashing" can be found in the following sections. The player's participation interval in the Sandbox can not become bigger than a day.

Games in the Sandbox are held according to a set of rules corresponding to the rules of an accidental past stage of the tournament or to the rules of the next (current) stage. At the same time, the closer the rating value of the two players rating within the Sandbox, the more likely that they will be in the one game. The Sandbox starts before the start of the first stage of the tournament and ends after some time after the final stage (see the schedule of stages to clarify the details). In addition, the Sandbox is frozen during the stages of the tournament. Following the results of the games in the Sandbox there is a selection for participation in Round 1, which will involve 1080 of participants with the highest rating at the beginning of this stage of the tournament (if the rating is equal, priority is given to the player who previously sent the latest version of his strategy), as well as an additional selection to the next stages of the tournament, including the Finals.

Tournament stages:

- In **Round 1** you will learn the rules of the game and master the control of the robots. You are given 2 robots, same as your opponent. The task is — score goals! It's simple. Round 1, as all further stages, consists of two parts, between which there will be a short break (with the renewal of the Sandbox work), which allows to improve its strategy. The last strategy sent by the player before the beginning of this part is selected for the games in each part. Games are conducted in waves. In each wave, each player participates exactly in one game. The number of waves in each part is determined by the capabilities of the testing system, but it is guaranteed that it will not be less than ten. 300 highest rated participants will be held in Round 2. Also in Round 2 there will be an additional selection of 60 participants with the highest rating in the Sandbox (at the moment of Round 2 beginning) among those who did not passed according to the results of Round 1.
- In **Round 2** you have to improve your robot's skills. Now they can use nitro. Also, nitro packs spawn on the map that refill robot's nitro. The task is further complicated that after summarizing the Round 1, the part of the weak strategies will be eliminated and you will have to confront stronger opponents. According to the results of Round 2 of the best 50 strategies will reach the Finals. Also in the Finals there will be an additional selection of 10 participants with the highest rating in the Sandbox (at the beginning of the Finals) from those who did not go through the main tournament.
- **Finals** is the most important stage. After the selection, held following the results of the first two stages, the strongest participants will be remained. Also, not each team has 3 robots. The system of holding the Finals has its own peculiarities. The stage is still divided into two parts, but they will no longer consist of waves. In each part of the stage, games will be played between all pairs of Finals participants. If the time and capabilities of the testing system permit, the operation will be repeated.

All finalists are ranked according to the non-increase in the rating after the end of the Finals. If the ratings are equal, a higher place is taken by that finalist, whose strategy, which was part of the Final, was sent out earlier. Prizes for the Final are distributed based on the occupied place after this ordering.

After the completion of the Sandbox, all its participants, except for the Finals winners, are ranked according to the non-increase in the rating. If the ratings are equal a higher place is taken by the participant who sent the latest version of his strategy earlier. Prizes for the Sandbox are distributed on the basis of occupied place after this ordering.

1.2 Description of the game world

The game world is three dimensional, and is limited by the arena, that is a rounded box with both teams' nets.

All robots and the ball have a spherical shape. Neither robots nr the ball can leave the arena. X axis is horizontal, parallel to the middle of the arena, Y axis is vertical, directed up, Z axis is horizontal, directed from your net to your oppenent's net.

In the beginning of each game, and after each scored ball, robots are placed on their half of the arena. The coordinates are transformed for your strategy in such a way that your strategy always "thinks", that your half of the fiels is located in half-space with coordinate $z < 0$. Robots are located randomly on the arena's floor, on the same distance from the ball. Your robot's location is symmetrical to the opponent's robots about center of the arena (point $(0, 0)$). The ball is placed at location $(0, y, 0)$, where y — is equiprobably selected from `BALL_RADIUS` to $4 \times$ `BALL_RADIUS`.

Time in the game is discrete and is measured in "ticks". At the beginning of each tick, the game simulator transmits the world state data to the participants' strategies, receives control alarms from them and updates the state of the world in accordance with these alarms and the limitations of the world. Then makes calculation of the change of the world and objects in it for this tick, and the process is repeated again with the updated data. The maximum duration of any game is equal to 20000 ticks, but the game can be terminated prematurely if all strategies "have crashed".

The "crashed" strategy can no longer control the robots. The strategy is considered as "crashed" in the following cases:

- The process in which the strategy is started has unexpectedly terminated, or an error has occurred in the protocol of interaction between the strategy And game server.
- The strategy exceeded one (any) of the time constraints assigned to it. Strategy for one tick is allocated not more than 20 seconds of real time. But in sum for the whole game the strategy process is given

  $$20 \times <duration\_of\_game\_in\_ticks> + 20000 \tag{1.1}$$

  milliseconds of real time. The formula takes into account the maximum duration of the game. The time limit remains the same, even if the actual duration of the game is different from this value. All time limits apply not only to the participant code, but on the interaction of the client-shell strategy with the game simulator.
- The strategy exceeded the memory limit. At any point in time the strategy process should not consume more than 256 MB of RAM.

1.3 Arena shape

Arena is a box with rounded corners, and nets for each team. Robots and the ball can be inside the nets.

Arena's dimensions are described by the `Arena` object, with is in the `arena` fiels of the `Rules` (more in 3).

Arena parameters:

```
width: 60
height: 20
depth: 80
bottom_radius: 3
top_radius: 7
corner_radius: 13
```

```
goal_top_radius: 3
goal_width: 30
goal_depth: 10
goal_height: 10
goal_side_radius: 1
```

Schematic view of the arena:

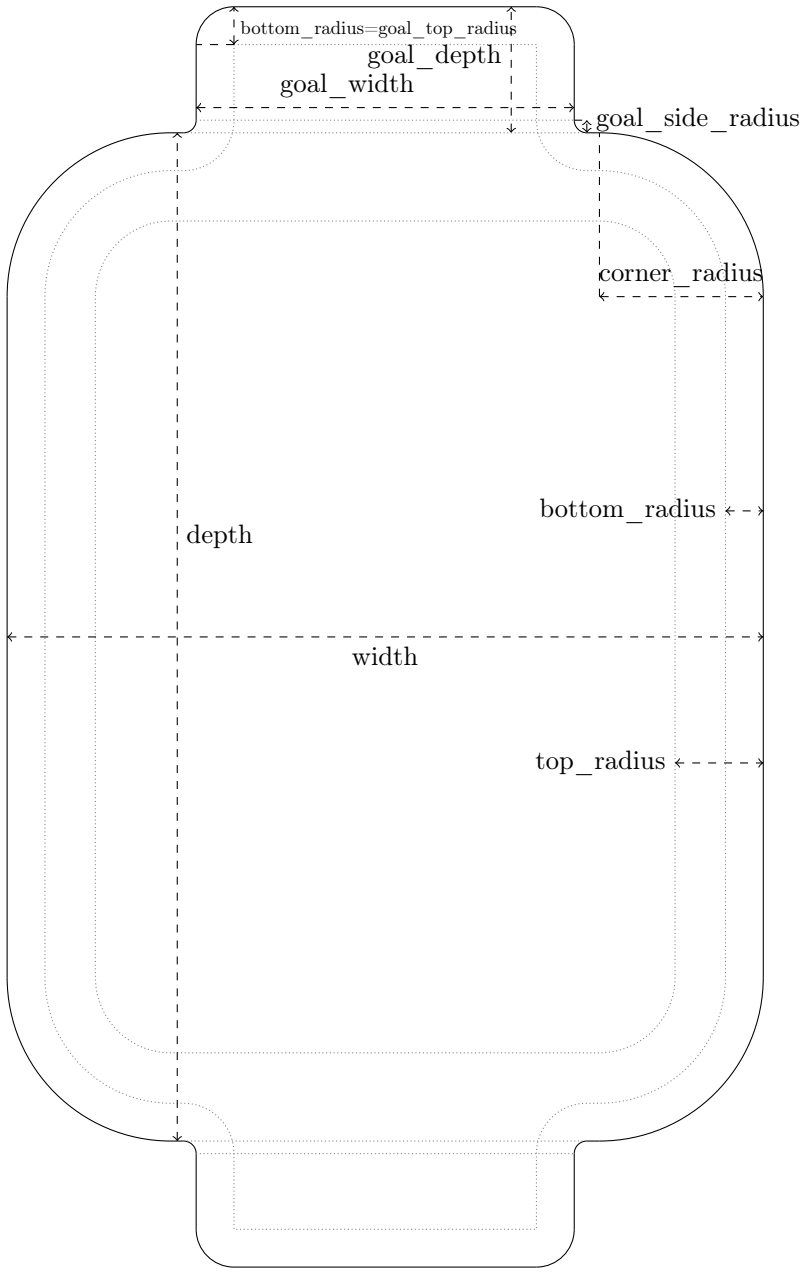

Figure 1.1: Top-down view

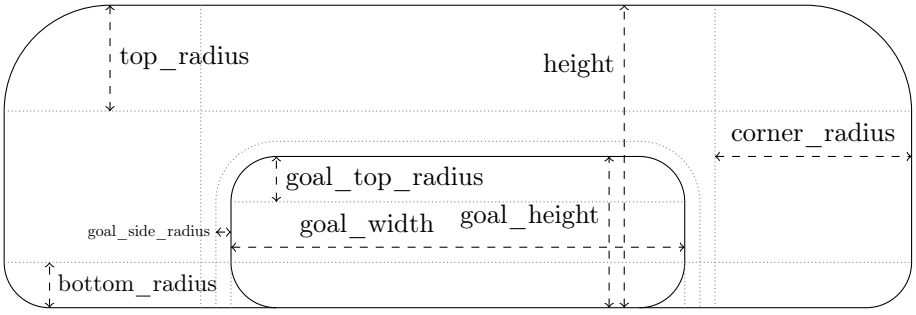

Figure 1.2: View on the net

1.4 Ball and robots

Ball is a sphere with radius BALL_RADIUS and mass BALL_MASS. Coefficient of restitution[1] when the ball collides with the arena is BALL_ARENA_E, so ball loses some speed upon collision.

In the beginning the ball is located in the center of the arena on a random height from the segment [BALL_RADIUS..4 × BALL_RADIUS].

The ball is considered scored if it completely crosses the net line:

$$abs(ball.z) > arena.depth/2 + ball.radius \tag{1.2}$$

After a scored goal simulation will be reset after RESET_TICKS ticks.

Robots are also shperes (with mass ROBOT_MASS), but there radius may vary from ROBOT_MIN_RADIUS to ROBOT_MAX_RADIUS, and is changed while jumping. The greater jump speed is, the greater the radius. Exact formula:

$$\begin{aligned} &radius = \texttt{ROBOT\_MIN\_RADIUS} + \\ &(\texttt{ROBOT\_MAX\_RADIUS} - \texttt{ROBOT\_MIN\_RADIUS}) \times \frac{jump\_speed}{\texttt{ROBOT\_MAX\_JUMP\_SPEED}} \end{aligned} \tag{1.3}$$

Small increase of the radius while jumping gives robots opportunity to jump off the walls.

Also, while jumping, the simulator thinks, that robot's radius is changing at a constant speed equal to the set jump speed (even although the radius is actually calculated using the above formula). This is how jumping is performed. This way, besides jumping itself, jump speed also has an effect on the strength of hitting the ball.

Coefficient of restitution when robot collides with arena is 0, so this kind of collision is absolutely inelastic. When robot collides with the ball, or with other robots, coefficient of restitution is equiprobably selected from MIN_HIT_E to MAX_HIT_E.

1.5 Nitro

Beginning with round 2, robots will be allowed to use nitro. Nitro gives robot acceleration in any direction. Maximal robot's nitro amount is MAX_NITRO_AMOUNT, one point of nitro changes robot's speed by NITRO_POINT_VELOCITY_CHANGE. Using nitro acceleration can't be greater than ROBOT_NITRO_ACCELERATION.

Nitro can be refilled by picking up nitro packs. Nitro packs are spheres with radius NITRO_PACK_RADIUS, and are considered picked up if distance betweeb robot's center and nitro pack's center is less than sum of their radii. Each nitro pack restores nitro back to full (to 100). After picked up, the pack will be respawned in the same place after NITRO_RESPAWN_TICKS ticks. There are 4 nitro packs total, in points $(\pm\, x, 1, \pm z)$.

In the beginning of the game and after each goal every robot's nitro is equal to START_NITRO_AMOUNT.

In first round nitro is always 0, and there are no nitro packs.

1.6 Control

Controlling robot means setting target velocity vector, jump speed, and a flag whether nitro should be used.

[1] The coefficient of restitution is the ratio of the final to initial relative velocity between two objects after they collide.

Without nitro, if robot touches arena, his velocity will be changing towards target velocity, projected to the touch plane. Acceleration can not be greater than `ROBOT_ACCELERATION`. Also, the more vertical the surface, the less acceleration is. (also, if robot touches arena with its upper half, acceleration is zero). Thus, when touching the floor, acceleration is maximal, but touching walls or ceiling, it is zero.

When using nitro, velocity also changes towards target velocity, but touching the arena has no effect.

Jump speed can be set from 0 (don't jump) to `ROBOT_MAX_JUMP_SPEED` (jump with max speed). While jumping, robot's radius is increased slightly, giving him the ability to jump off walls. Also, while jumping, simulator considers radius to be changing with speed equal to jump speed. So, jumping is performed because when colliding with the floor, relative velocity in the collition point is equal to jump speed. This also means than you can control strength of hitting the ball.

Chapter 2

Simulation

CodeBall simulation is based on a simple physics model using impulse method. There is no friction is the system, and all collisions are resolved consequentially in random order.

Simulator's pseudo-code:

```
function collide_entities(a: Entity, b: Entity):
    let delta_position = b.position - a.position
    let distance = length(delta_position)
    let penetration = a.radius + b.radius - distance
    if penetration > 0:
        let k_a = (1 / a.mass) / ((1 / a.mass) + (1 / b.mass))
        let k_b = (1 / b.mass) / ((1 / a.mass) + (1 / b.mass))
        let normal = normalize(delta_position)
        a.position -= normal * penetration * k_a
        b.position += normal * penetration * k_b
        let delta_velocity = dot(b.velocity - a.velocity, normal)
            - b.radius_change_speed - a.radius_change_speed
        if delta_velocity < 0:
            let impulse = (1 + random(MIN_HIT_E, MAX_HIT_E)) * delta_velocity * normal
            a.velocity += impulse * k_a
            b.velocity -= impulse * k_b

function collide_with_arena(e: Entity):
    let distance, normal = dan_to_arena(e.position)
    let penetration = e.radius - distance
    if penetration > 0:
        e.position += penetration * normal
        let velocity = dot(e.velocity, normal) - e.radius_change_speed
        if velocity < 0:
            e.velocity -= (1 + e.arena_e) * velocity * normal
            return normal
    return None

function move(e: Entity, delta_time: float):
    e.velocity = clamp(e.velocity, MAX_ENTITY_SPEED)
    e.position += e.velocity * delta_time
    e.position.y -= GRAVITY * delta_time * delta_time / 2
    e.velocity.y -= GRAVITY * delta_time
```

```
function update(delta_time: float):
    shuffle(robots)

    for robot in robots:
        if robot.touch:
            let target_velocity = clamp(
                robot.action.target_velocity,
                ROBOT_MAX_GROUND_SPEED)
            target_velocity -= robot.touch_normal
                * dot(robot.touch_normal, target_velocity)
            let target_velocity_change = target_velocity - robot.velocity
            if length(target_velocity_change) > 0:
                let acceleration = ROBOT_ACCELERATION * max(0, robot.touch_normal.y)
                robot.velocity += clamp(
                    normalize(target_velocity_change) * acceleration * delta_time,
                    length(target_velocity_change))

        if robot.action.use_nitro:
            let target_velocity_change = clamp(
                robot.action.target_velocity - robot.velocity,
                robot.nitro * NITRO_POINT_VELOCITY_CHANGE)
            if length(target_velocity_change) > 0:
                let acceleration = normalize(target_velocity_change)
                    * ROBOT_NITRO_ACCELERATION
                let velocity_change = clamp(
                    acceleration * delta_time,
                    length(target_velocity_change))
                robot.velocity += velocity_change
                robot.nitro -= length(velocity_change)
                    / NITRO_POINT_VELOCITY_CHANGE

        move(robot, delta_time)
        robot.radius = ROBOT_MIN_RADIUS + (ROBOT_MAX_RADIUS - ROBOT_MIN_RADIUS)
                    * robot.action.jump_speed / ROBOT_MAX_JUMP_SPEED
        robot.radius_change_speed = robot.action.jump_speed

    move(ball, delta_time)

    for i in 0 .. length(robots) - 1:
        for j in 0 .. i - 1:
            collide_entities(robots[i], robots[j])

    for robot in robots:
        collide_entities(robot, ball)
        collision_normal = collide_with_arena(robot)
        if collision_normal is None:
            robot.touch = false
        else:
            robot.touch = true
            robot.touch_normal = collision_normal
    collide_with_arena(ball)

    if abs(ball.position.z) > arena.depth / 2 + ball.radius:
        goal_scored()

    for robot in robots:
```

```
        if robot.nitro == MAX_NITRO_AMOUNT:
            continue
        for pack in nitro_packs:
            if not pack.alive:
                continue
            if length(robot.position - pack.position) <= robot.radius + pack.radius:
                robot.nitro = MAX_NITRO_AMOUNT
                pack.alive = false
                pack.respawn_ticks = NITRO_PACK_RESPAWN_TICKS

function tick():
    let delta_time = 1 / TICKS_PER_SECOND
    for _ in 0 .. MICROTICKS_PER_TICK - 1:
        update(delta_time / MICROTICKS_PER_TICK)

    for pack in nitro_packs:
        if pack.alive:
            continue
        pack.respawn_ticks -= 1
        if pack.respawn_ticks == 0:
            pack.alive = true
```

2.1 Finding nearest point to the arena

```
function dan_to_plane(point: Vec3D, point_on_plane: Vec3D, plane_normal: Vec3D):
    return {
        distance: dot(point - point_on_plane, plane_normal)
        normal: plane_normal
    }

function dan_to_sphere_inner(point: Vec3D, sphere_center: Vec3D, sphere_radius: Float):
    return {
        distance: sphere_radius - length(point - sphere_center)
        normal: normalize(sphere_center - point)
    }

function dan_to_sphere_outer(point: Vec3D, sphere_center: Vec3D, sphere_radius: Float):
    return {
        distance: length(point - sphere_center) - sphere_radius
        normal: normalize(point - sphere_center)
    }
}

function dan_to_arena_quarter(point: Vec3D):
    // Ground
    let dan = dan_to_plane(point, (0, 0, 0), (0, 1, 0))

    // Ceiling
    dan = min(dan, dan_to_plane(point, (0, arena.height, 0), (0, -1, 0)))

    // Side x
    dan = min(dan, dan_to_plane(point, (arena.width / 2, 0, 0), (-1, 0, 0)))

    // Side z (goal)
```

```
dan = min(dan, dan_to_plane(
        point,
        (0, 0, (arena.depth / 2) + arena.goal_depth),
        (0, 0, -1)))

// Side z
let v = (point.x, point.y) - (
        (arena.goal_width / 2) - arena.goal_top_radius,
        arena.goal_height - arena.goal_top_radius)
if point.x >= (arena.goal_width / 2) + arena.goal_side_radius
        or point.y >= arena.goal_height + arena.goal_side_radius
        or (
            v.x > 0
            and v.y > 0
            and length(v) >= arena.goal_top_radius + arena.goal_side_radius):
    dan = min(dan, dan_to_plane(point, (0, 0, arena.depth / 2), (0, 0, -1)))

// Side x & ceiling (goal)
if point.z >= (arena.depth / 2) + arena.goal_side_radius:
    // x
    dan = min(dan, dan_to_plane(
            point,
            (arena.goal_width / 2, 0, 0),
            (-1, 0, 0)))
    // y
    dan = min(dan, dan_to_plane(point, (0, arena.goal_height, 0), (0, -1, 0)))

// Goal back corners
assert arena.bottom_radius == arena.goal_top_radius
if point.z > (arena.depth / 2) + arena.goal_depth - arena.bottom_radius:
    dan = min(dan, dan_to_sphere_inner(
            point,
            (
                clamp(
                    point.x,
                    arena.bottom_radius - (arena.goal_width / 2),
                    (arena.goal_width / 2) - arena.bottom_radius,
                ),
                clamp(
                    point.y,
                    arena.bottom_radius,
                    arena.goal_height - arena.goal_top_radius,
                ),
                (arena.depth / 2) + arena.goal_depth - arena.bottom_radius),
            arena.bottom_radius))

// Corner
if point.x > (arena.width / 2) - arena.corner_radius
        and point.z > (arena.depth / 2) - arena.corner_radius:
    dan = min(dan, dan_to_sphere_inner(
            point,
            (
                (arena.width / 2) - arena.corner_radius,
                point.y,
                (arena.depth / 2) - arena.corner_radius
            ),
```

```
            arena.corner_radius))

// Goal outer corner
if point.z < (arena.depth / 2) + arena.goal_side_radius:
    // Side x
    if point.x < (arena.goal_width / 2) + arena.goal_side_radius:
        dan = min(dan, dan_to_sphere_outer(
                point,
                (
                    (arena.goal_width / 2) + arena.goal_side_radius,
                    point.y,
                    (arena.depth / 2) + arena.goal_side_radius
                ),
                arena.goal_side_radius))
    // Ceiling
    if point.y < arena.goal_height + arena.goal_side_radius:
        dan = min(dan, dan_to_sphere_outer(
                point,
                (
                    point.x,
                    arena.goal_height + arena.goal_side_radius,
                    (arena.depth / 2) + arena.goal_side_radius
                ),
                arena.goal_side_radius))
    // Top corner
    let o = (
        (arena.goal_width / 2) - arena.goal_top_radius,
        arena.goal_height - arena.goal_top_radius
    )
    let v = (point.x, point.y) - o
    if v.x > 0 and v.y > 0:
        let o = o + normalize(v) * (arena.goal_top_radius + arena.goal_side_radius)
        dan = min(dan, dan_to_sphere_outer(
                point,
                (o.x, o.y, (arena.depth / 2) + arena.goal_side_radius),
                arena.goal_side_radius))

// Goal inside top corners
if point.z > (arena.depth / 2) + arena.goal_side_radius
        and point.y > arena.goal_height - arena.goal_top_radius:
    // Side x
    if point.x > (arena.goal_width / 2) - arena.goal_top_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    (arena.goal_width / 2) - arena.goal_top_radius,
                    arena.goal_height - arena.goal_top_radius,
                    point.z
                ),
                arena.goal_top_radius))
    // Side z
    if point.z > (arena.depth / 2) + arena.goal_depth - arena.goal_top_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    point.x,
```

```
                    arena.goal_height - arena.goal_top_radius,
                    (arena.depth / 2) + arena.goal_depth - arena.goal_top_radius
                ),
                arena.goal_top_radius))

// Bottom corners
if point.y < arena.bottom_radius:
    // Side x
    if point.x > (arena.width / 2) - arena.bottom_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    (arena.width / 2) - arena.bottom_radius,
                    arena.bottom_radius,
                    point.z
                ),
                arena.bottom_radius))
    // Side z
    if point.z > (arena.depth / 2) - arena.bottom_radius
            and point.x >= (arena.goal_width / 2) + arena.goal_side_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    point.x,
                    arena.bottom_radius,
                    (arena.depth / 2) - arena.bottom_radius
                ),
                arena.bottom_radius))
    // Side z (goal)
    if point.z > (arena.depth / 2) + arena.goal_depth - arena.bottom_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    point.x,
                    arena.bottom_radius,
                    (arena.depth / 2) + arena.goal_depth - arena.bottom_radius
                ),
                arena.bottom_radius))
    // Goal outer corner
    let o = (
        (arena.goal_width / 2) + arena.goal_side_radius,
        (arena.depth / 2) + arena.goal_side_radius
    )
    let v = (point.x, point.z) - o
    if v.x < 0 and v.y < 0
            and length(v) < arena.goal_side_radius + arena.bottom_radius:
        let o = o + normalize(v) * (arena.goal_side_radius + arena.bottom_radius)
        dan = min(dan, dan_to_sphere_inner(
                point,
                (o.x, arena.bottom_radius, o.y),
                arena.bottom_radius))
    // Side x (goal)
    if point.z >= (arena.depth / 2) + arena.goal_side_radius
            and point.x > (arena.goal_width / 2) - arena.bottom_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
```

```
                (
                    (arena.goal_width / 2) - arena.bottom_radius,
                    arena.bottom_radius,
                    point.z
                ),
                arena.bottom_radius))
    // Corner
    if point.x > (arena.width / 2) - arena.corner_radius
            and point.z > (arena.depth / 2) - arena.corner_radius:
        let corner_o = (
            (arena.width / 2) - arena.corner_radius,
            (arena.depth / 2) - arena.corner_radius
        )
        let n = (point.x, point.z) - corner_o
        let dist = n.len()
        if dist > arena.corner_radius - arena.bottom_radius:
            let n = n / dist
            let o2 = corner_o + n * (arena.corner_radius - arena.bottom_radius)
            dan = min(dan, dan_to_sphere_inner(
                    point,
                    (o2.x, arena.bottom_radius, o2.y),
                    arena.bottom_radius))

// Ceiling corners
if point.y > arena.height - arena.top_radius:
    // Side x
    if point.x > (arena.width / 2) - arena.top_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    (arena.width / 2) - arena.top_radius,
                    arena.height - arena.top_radius,
                    point.z,
                ),
                arena.top_radius))
    // Side z
    if point.z > (arena.depth / 2) - arena.top_radius:
        dan = min(dan, dan_to_sphere_inner(
                point,
                (
                    point.x,
                    arena.height - arena.top_radius,
                    (arena.depth / 2) - arena.top_radius,
                )
                arena.top_radius))

    // Corner
    if point.x > (arena.width / 2) - arena.corner_radius
            and point.z > (arena.depth / 2) - arena.corner_radius:
        let corner_o = (
            (arena.width / 2) - arena.corner_radius,
            (arena.depth / 2) - arena.corner_radius
        )
        let dv = (point.x, point.z) - corner_o
        if length(dv) > arena.corner_radius - arena.top_radius:
            let n = normalize(dv)
```

```
                let o2 = corner_o + n * (arena.corner_radius - arena.top_radius)
                dan = min(dan, dan_to_sphere_inner(
                        point,
                        (o2.x, arena.height - arena.top_radius, o2.y),
                        arena.top_radius))

    return dan
}

function dan_to_arena(point: Vec3D):
    let negate_x = point.x < 0
    let negate_z = point.z < 0
    if negate_x:
        point.x = -point.x
    if negate_z:
        point.z = -point.z
    let result = dan_to_arena_quarter(point)
    if negate_x:
        result.normal.x = -result.normal.x
    if negate_z:
        result.normal.z = -result.normal.z
    return result
```

2.2 Constants

These constants are also available in the `Rules` object (see 3).

Please note that the rate constants and accelerations are given in the system $\frac{distance\ unit}{second}$ и $\frac{distance\ unit}{second^2}$.

```
ROBOT_MIN_RADIUS = 1
ROBOT_MAX_RADIUS = 1.05
ROBOT_MAX_JUMP_SPEED = 15
ROBOT_ACCELERATION = 100
ROBOT_NITRO_ACCELERATION = 30
ROBOT_MAX_GROUND_SPEED = 30
ROBOT_ARENA_E = 0
ROBOT_RADIUS = 1
ROBOT_MASS = 2
TICKS_PER_SECOND = 60
MICROTICKS_PER_TICK = 100
RESET_TICKS = 2 * TICKS_PER_SECOND
BALL_ARENA_E = 0.7
BALL_RADIUS = 2
BALL_MASS = 1
MIN_HIT_E = 0.4
MAX_HIT_E = 0.5
MAX_ENTITY_SPEED = 100
MAX_NITRO_AMOUNT = 100
START_NITRO_AMOUNT = 50
NITRO_POINT_VELOCITY_CHANGE = 0.6
NITRO_PACK_X = 20
NITRO_PACK_Y = 1
NITRO_PACK_Z = 30
NITRO_PACK_RADIUS = 0.5
```

```
NITRO_PACK_AMOUNT = 100
NITRO_PACK_RESPAWN_TICKS = 10 * TICKS_PER_SECOND
GRAVITY = 30
```

Chapter 3

API description

3.1 MyStrategy

In language pack for your programming language you can find file named`MyStrategy/my_strategy.<ext>`. This file contains class `MyStrategy` with `act` method, where your strategy's logic should be implementeed.

This method will be called each tick, separately for each of your robots.

```
class MyStrategy:
      method act(me: Robot, rules: Rules, game: Game, action: Action):
            // Your implementation
```

`act` parameters:

- `me` — your robot, for who the action is being determined
- `rules` — object, containing rules of the game (does not change between ticks)
- `game` — object, containing information about current state of the game (changes between ticks)
- `action` — object describing robot's action. By changing it's fields you set up action for your robot.

3.2 Action

Object describing robot's action.

```
class Action:
      target_velocity_x: Float      // X coordinate of target velocity
      target_velocity_y: Float      // Y coordinate of target velocity
      target_velocity_z: Float      // Z coordinate of target velocity
      jump_speed: Float             // Jump speed
      use_nitro: Bool               // Whether nitro should be used
```

3.3 Arena

Object describing arena dimensions (look 1.3).

```
class Arena:
      width: Float
      height: Float
      depth: Float
      bottom_radius: Float
      top_radius: Float
      corner_radius: Float
      goal_top_radius: Float
      goal_width: Float
      goal_height: Float
      goal_depth: Float
      goal_side_radius: Float
```

3.4 Ball

Object describing the ball.

```
class Ball:
      x: Float                    // Current coordinates of the ball's center
      y: Float
      z: Float
      velocity_x: Float           // Current ball's velocity
      velocity_y: Float
      velocity_z: Float
      radius: Float               // Ball's radius
```

3.5 Game

Object containing information about current game state.

```
class Game:
      current_tick: Int           // Current tick index
      players: Player[]           // List of players
      robots: Robot[]             // List of robots
      nitro_packs: NitroPack[]    // List of nitro packs
      ball: Ball                  // The ball
```

3.6 NitroPack

Object describing a nitro pack.

```
class NitroPack:
      id: Int
      x: Float                    // Coordinates of the center of the pack
      y: Float
      z: Float
      radius: Float               // Radius
      respawn_ticks: Int?         // Number of ticks until respawn
                                        (or null, if not picked up yet)
```

3.7 Player

Object describing player (strategy).

```
class Player:
      id: Int
      me: Bool                      // true, if this is you
      strategy_crashed: Bool
      score: Int                    // Current score
```

3.8 Robot

Object describing a robot.

```
class Robot:
      id: Int
      player_id: Int                // Controlling player's id
      is_teammate: Bool             // true, if this robot is controlled by you
      x: Float                      // Current coordinates of robot's center
      y: Float
      z: Float
      velocity_x: Float             // Current velocity
      velocity_y: Float
      velocity_z: Float
      radius: Float                 // Current radius
      nitro_amount: Float           // Current nitro amount
      touch: bool                   // true, if robot currently touches arena
      touch_normal_x: Float         // Normal to the touch surface
      touch_normal_y: Float               (or null, is no touching)
      touch_normal_z: Float
```

3.9 Rules

Object describing rules (persistent game properties). Also contains constants used in simulation

```
class Rules:
      max_tick_count: Int           // Maximal game length in ticks
      arena: Arena                  // Arena dimensions
      team_size: Int                // Number of robots in each team

      // Constants used in simulation
      ROBOT_MIN_RADIUS: Float
      ROBOT_MAX_RADIUS: Float
      ROBOT_MAX_JUMP_SPEED: Float
      ROBOT_ACCELERATION: Float
      ROBOT_NITRO_ACCELERATION: Float
      ROBOT_MAX_GROUND_SPEED: Float
      ROBOT_ARENA_E: Float
      ROBOT_RADIUS: Float
      ROBOT_MASS: Float
```

```
TICKS_PER_SECOND: Int
MICROTICKS_PER_TICK: Int
RESET_TICKS: Int
BALL_ARENA_E: Float
BALL_RADIUS: Float
BALL_MASS: Float
MIN_HIT_E: Float
MAX_HIT_E: Float
MAX_ENTITY_SPEED: Float
MAX_NITRO_AMOUNT: Float
START_NITRO_AMOUNT: Float
NITRO_POINT_VELOCITY_CHANGE: Float
NITRO_PACK_X: Float
NITRO_PACK_Y: Float
NITRO_PACK_Z: Float
NITRO_PACK_RADIUS: Float
NITRO_PACK_AMOUNT: Float
NITRO_PACK_RESPAWN_TICKS: Int
GRAVITY: Float
```