ELSIROS

Release 0.0.1

Azer Babaev, Egor Davydenko, Ilya Ryakin, Vladimir Litvinenko, A

Sep 21, 2021

CONTENTS:

CHAPTER

ONE

ELSIROS INTRO

Soccer game of Humanoid robots is becoming popular between teams in education and academic society. This type of competitions generates many scientific research points. For students in high schools, colleges or universities in general there are 3 obstacles preventing them to set up and maintain a humanoid robot soccer team:

1. extremely high price of humanoid robots,
2. too complex algorithmic problems to be solved at initial steps of setting robot into game,
3. small number of teams in one geographical location – means every game is possible only together with travelling to far distance with relatively high travel charges.

All 3 obstacles can be eliminated if teams start their Humanoid Soccer experience from ELSIROS. Following advantages are obtained with using ELSIROS:

1. ELSIROS is free of charge and open source.
2. Most difficult parts of humanoid robot development which are beyond of school or college program like Inverse Kinematics, walking engine, path planning, detecting of ball and obstacles and localization are provided ready-made in source codes. Example of playing strategy which proved to be a leading strategy at games of real robots is provided for study and improvement.
3. In order to participate in competitions or challenges it is not necessary to travel. Participating teams can compile their source code into executable binary code which is safe against source code leaking and upload to referees server.
4. Teams don't suffer from backlashes, mis-tunings, mis-calibrations because models are tuned, calibrated and free of backlashes.
5. Strategy modules developed by teams will be ready to be used on real robots which teams may decide to build or to buy from market in future.
6. It is not necessarily powerful servers for running training games, simulation can run even at laptop.

ELSIROS is created by Humanoid Robot Soccer team "Starkit" from MIPT (Moscow) after winning Robocup World Championship 2021. ELSIROS is free of charge and open-source platform pretending from one side to help new research groups to enter into Humanoid Soccer Competitions world, from other side to host virtual games. ELSIROS comprises of following components:

1. Webots simulator (to be downloaded from vendors' site);
2. Primary robot model, which is a virtual model of existing physical robot – a winner of international humanoid soccer challenges in 2019, 2020 and 2021 used by team "Robokit";
3. Soccer simulation environment for simulator;
4. Autonomous/Human referee program and game controller;
5. Robot controller software pack capable to play games;

Teams can participate in competitions with Robokit robot and use it for study of basics of programming of strategy of soccer game for humanoid robots. This is convenient instrument for study of Artificial Intelligence in schools, colleges and universities. Programming of robots is supported with Python 3 language. But in case of necessity also C, C++, Java or MATLAB languages can be used for your convenience

Structure of robot controlling software is built for 4 level of robot developers: Beginner, Medium, Advanced and Expert level. Beginner level developers can access to programming of strategy.py file with purpose to change current robot behavior in game play. Initially supplied source code represents strategy of game used by leading Russian team at National championship 2021. Video of this game can be found under link below:

```
https://youtu.be/AmfKpkL2MUc
```

Medium level developers can try to improve launcher.py module. This module stands for detecting game state, player state and team state, managing players role and their starting positions. Advanced developers can try to modify other modules of source code which are responsible for inverse kinematics, motion, localization, robots' path planning.

Expert level developers are allowed to compose their own model of robot and use their own controller software with or without using source code included into ELSIROS open-source package. In order to be admitted to competition program team providing virtual model of their own robot design have to be qualified. Main requirement to robot model is that virtual model and real robot must have the same technical performance in all details. Special requirements to robots' PROTO which appear from simulation environment can be sent after special request.

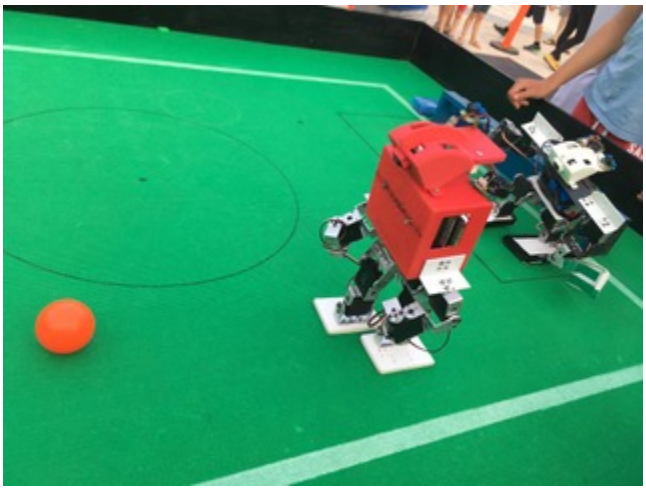
If you are a mentor of potential Humanoid Robot Soccer team you can easily start-up with your team. It is necessary simply download and install ELSIROS in your computer. There is executable version for Windows 10 or source code for Linux. Please follow instructions and sample code will play soccer just after installation. Please use your favorite Python 3 IDE for improving players' source code and you are ready to Virtual Humanoid Robot Soccer challenges in simulation. You can train your games at your laptop computer.

CHAPTER

TWO

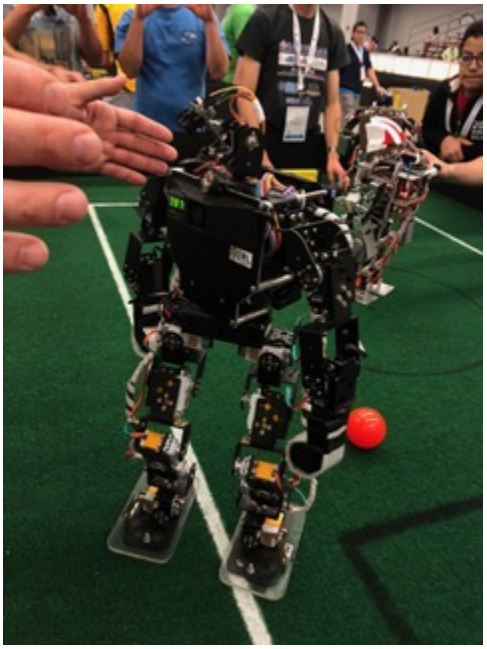
HISTORY OF JUNIOR HUMANOID SOCCER GAMES

Known early examples of humanoid robot soccer built and programmed by junior students for Robocup refers to year 2016.



Some attepmts to play soccer by humanoid robot built by team from Israel were made at 2016 - 2018

Teams from Israel and Italy have played first International match as demo game at Robocup-2018 in Montreal



In parallel 2 teams from Tomsk and from Moscow have played first game at Robofinist tournament in St.Petersburg at 2018

Same 2 teams have repeated demo game in Moscow at 2019

First full function tournament was held in Tomsk-2019 with 3 teams participating in Russian Nathinonal Robocup


ый Российский этап чемпионата

Since 2019 the game was included into program of Robocup Asia - Pacific.

2.1 The field

The field is made from 3mm carpet with total size of 3 x 4 m,

with goals made from plumbing tubes, colored to yellow ant blue color.

2.2 The ball

The ball is orange color sponge ball with 80mm diameter.

2.3 The Robots

Currently 2 types of robots mostly used by teams:

2.3.1 Bioloid type with added camera head and computer

2.3.2 Robokit - a robot designed in MIPT with using Kondo servomotors from Japan

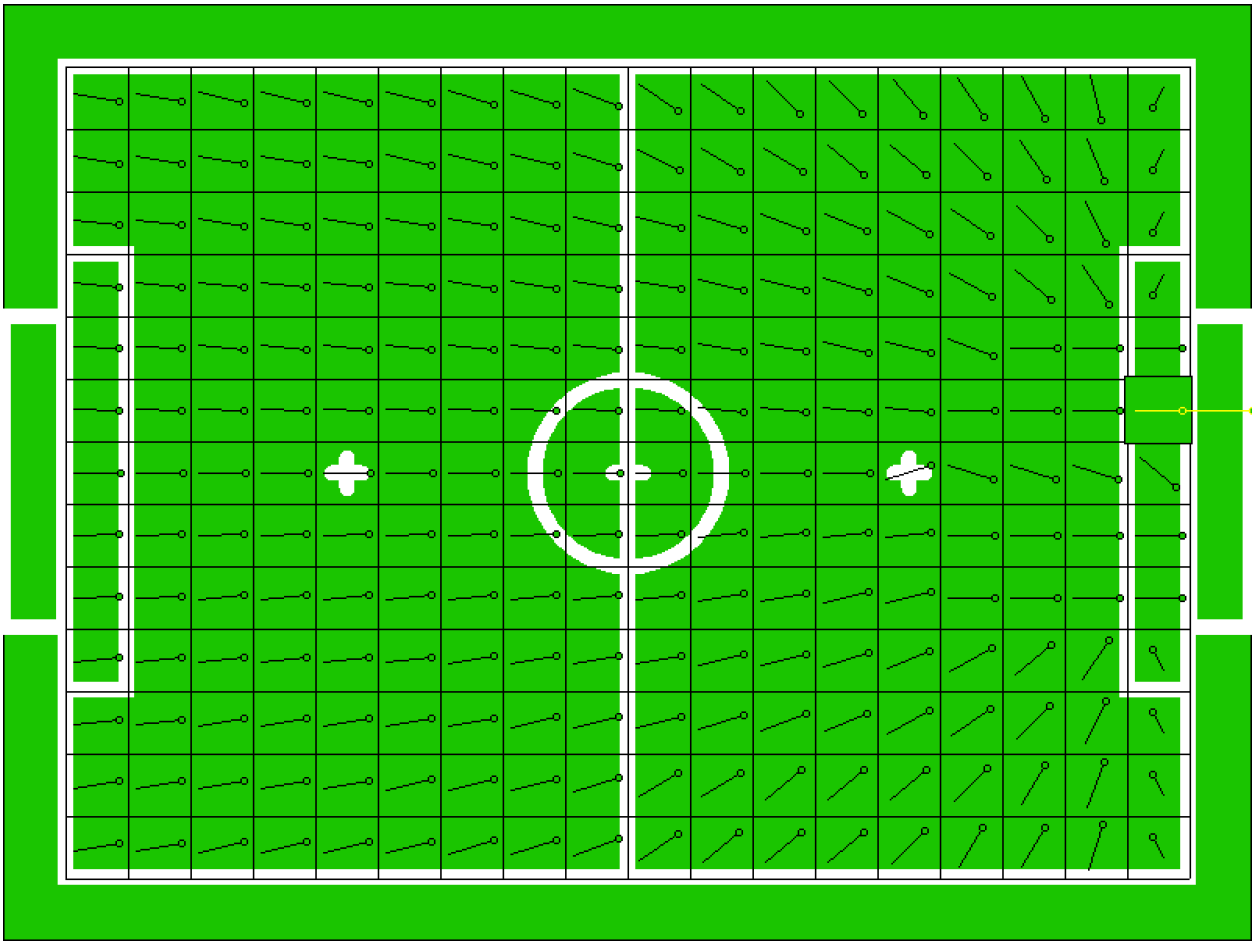
Last robot which performed best results in soccer game being champion of Asia-Pacific games of 2019 and 2020 is used as prototype for virtual model of standart robot in ELSIROS platform.

CHAPTER

THREE

ROBOT PROGRAMMING HINTS

1. It is not advised to play games between completely equal teams. In most cases you can observe non-interesting game development with rare goals and useless struggling. Therefore, we include 2 styles of playing algorithms: normal and old style which was practiced in year 2020.
2. Real Robots use OpenMV H7 smart camera as vision sensor and onboard computing module. This is single core controller with Micropython bare metal programming. This means that controller is not capable to provide walking engine and camera vision simultaneously. In order to update information about ball position and self-localization robot has to stop into stabile stand-up position and move head to various positions to observe surroundings. During head moving ball position can be red in case if ball is in visible sector of camera. Camera have 46 degrees of aperture.
3. During observation of surroundings at stand-up position of robot camera catches additional information about robots' localization from objects like goal posts, field marking and green field border. The more pictures are taken the better accuracy of localization.
4. Robot can detect obstacles from vision sensor. Obstacle avoidance algorithm is included into path planning, but it is not perfect in all aspects. There is not implementation of kicking ball strategy with accounting possible obstacles at ball path. Detected and updated data about obstacles are stored in list self.glob.obstacles
5. Communication between team members is legal by rules through UDP messages. Communication is not implemented in current game strategy, but it is allowed to be developed by teams. Communication inside team can help to organize team play. ELSIROS API provides functionality for messaging between team members.
6. Coordinate system of field for purposes of strategy is different from absolute coordinate system of field. For purpose of strategy own goals are located at part of field with negative X coordinate, opponents' goals are located at positive X coordinate. Positive Y coordinate is at left flank of attack, negative Y coordinate is at right flank of attack. Yaw heading is zero if it is directed from center of own goals to center of opponents' goals. Yaw is changed from 0 to math.pi with turning to left from zero direction. Yaw is changed from 0 to -math.pi with turning from zero direction to right. Z coordinate is directed to up with zero on floor.
7. Normal 'forward' player uses predefined strategy formulated by vector matrix. Matrix is coded in file strategy_data.json This file is readable and editable as well as normal text file. There is a dictionary with one key 'strategy_data'. Value of key 'strategy_data' is a list with default number of elements 234. Each element of list represents rectangular sector of soccer field with size 20cmX20cm. For each sector there assigned a vector representing yaw direction of shooting when ball is positioned in this sector. Power of shot is coded by attenuation value: 1 – standard power, 2 – power reduced 2 times, 3- power reduced 3 times. Each element of list is coded as follows: [column, row, power, yaw]. Soccer field is split to sectors in 13 rows and 18 columns. Column 0 is near own goals, column 17 is near opposed goals. Row 0 is in positive Y coordinate, row 12 is in negative Y coordinate.



1. During game player can take 4 roles: ‘forward’, ‘goalkeeper’, ‘penalty_Shooter’, ‘penlaty_Goalkeeper’. For each role strategy code is different. Launcher module chooses role of player to launch depending on number of player and secondary game state. In case if number of player is 1 then appointed role will be ‘goalkeeper’ or ‘penalty_Goalkeeper’. In case if number of player is other than 1 then appointed role will be ‘forward’ or ‘penalty_Shooter’. In case if secondary game state is ‘STATE_PENALTYSHOOT’ then player with number 1 will be appointed as ‘penlaty_Goalkeeper’ and player with other number will be appointed to role ‘penalty_Shooter’. Default public player controller strategy appoints player role ‘goalkeeper’ to player with number 1 and ‘forward’ to player with number other than 1 in all other secondary game states. Teams can modify strategy and use various roles depending on secondary game state. According to current game controller there could be following secondary game states: STATE_NORMAL=0, STATE_PENALTYSHOOT=1, STATE_OVERTIME=2, STATE_TIMEOUT=3, STATE_DIRECT_FREEKICK=4, STATE_INDIRECT_FREEKICK=5, STATE_PENALTYKICK=6, STATE_CORNERKICK=7, STATE_GOALKICK=8, STATE_THROWIN=9, DROPBALL=128, UNKNOWN=255

CHAPTER

FOUR

RULES

RoboCup Junior Humanoid Soccer Rules and Setup

for the 2021 virtual competitions in simulation

August 29th, 2021

Compiled by Azer Babaev
(based on the 2007 version of the rules Humanoid Soccer)

1 The Field of Play

The competitions take place on a rectangular field, which contains two goals, field lines, six restart markers, as shown in Fig. 1.

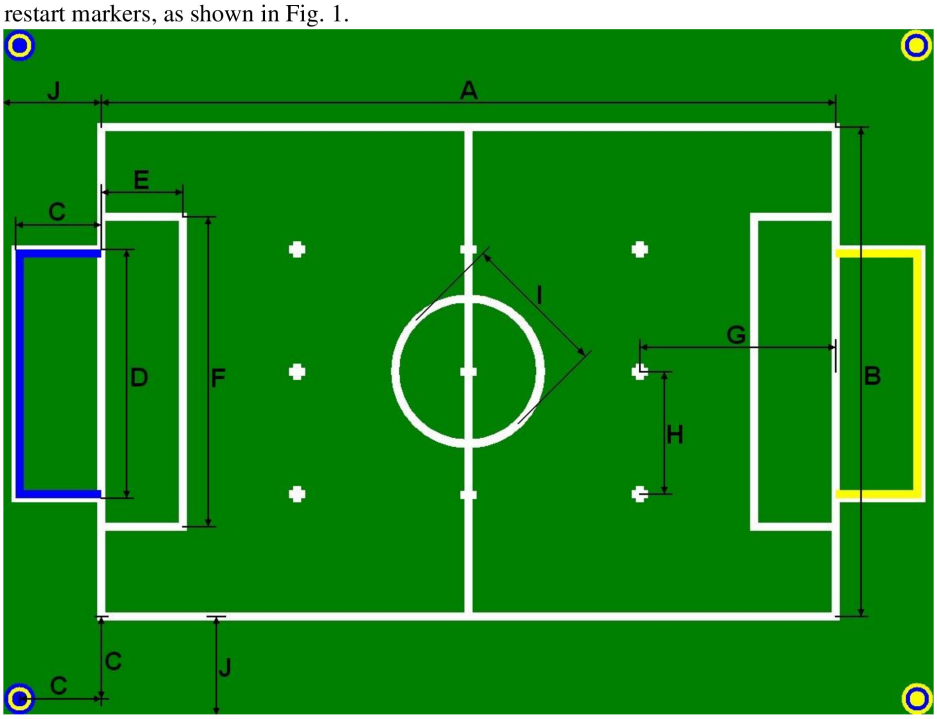


Figure 1: Soccer field.

Table 1: Soccer field sizes, in cm.

A	Field length	**360**	
B	Field width	**260**	
C	Goal length	**15**	
D	Goal width	**100**	
E	Goal area length	**20**	
F	Goal area width	**140**	
G	Penalty kick distance	**90**	
H	Restart marker width	**65**	
I	Center circle diameter	**60**	
J	Border strip width (min.)	**20**	

1.1 Playing Surface

The field is covered with green carpet. The white lines are 5cm wide. Line segments of 10cm length are used to mark penalty positions, restart positions, and the kick-off position. The longer outer field lines are called touch lines, whereas the shorter outer field lines are called RoboCup Junior Humanoid Soccer Rules and Setup

goal lines. The six restart positions are located half-way between the axis connecting the two goals and the touch lines, three on each side: in the middle and between the penalty positions and the touch lines. The field is surrounded by a border strip, which is also covered with green carpet. The world outside the border strip is undefined.

1.2 Goals

A goal is placed in the middle of each goal line. One of the goals is colored yellow at the three inner walls. The other goal is colored blue. The goals have a horizontal goal bar at a height of 60cm. The outer goal walls, as well as the goal posts and the goal bar are colored white.

1.3 Corner Poles

Not applicable for simulation

1.4 Lighting

The field is illuminated homogenously by artificial white light sources placed at a height greater than 2.5m above the field. The brightness of the lighting is between 600lux and 1200lux. The brightness within the field does not vary more than 300lux.

1.5 People area

Not applicable for simulation

2 The Ball

1. Orange plastic sponge ball (8 cm diameter, 26g),

3 The Number of Players

A match is played by two teams, each consisting of not more than two players, one of whom may be designated as goalkeeper. A match may not start if either team consists of less than one player.

3.1 Incapable players

Players not capable of play (e.g. players not walking on two legs, players not able to stand) are not permitted to participate in the game. They must be removed from the field. It is up to the referee to judge whether a player is incapable of play. A field player that is not able to get back into a standing or walking posture from a fall within 20 seconds receives a 30 seconds removal penalty. If the ball is within a radius of 0.3 m around the goal keeper inside the goal area, the goal keeper has to show active attempts to move the ball out of this radius by walking towards the ball or moving the ball. If no attempt is shown for 20 seconds, the goal keeper is considered to be an inactive player and receives a 30 seconds removal penalty. A player that stays outside of the field for 20 seconds is considered as an incapable player and receives a 30 seconds removal penalty.
Removal Penalty.

Time penalties of 30 seconds for players are called by the referee. A field player or goal keeper suffering a time penalty will be automatically removed from the field and is only allowed to re-enter the field from the team's own half of the field close to the penalty mark as indicated by the referee. The referee chooses the touch line further away from the ball if there is still an empty spot available. The first spot for a penalized robot on the touch line is on the same height of the penalty mark. A valid position must be at least 30 cm away from the goal line and center line. The referee always positions the robot on the penalty spot closest to the penalty mark. If two positions are available that are equally close, the referee chooses the position that is further away from the ball. When placed, the robot joints are reset to their initial position and their velocities is set to 0.
After the robot has been placed at the position indicated by the referee and with both feet entirely outside the field of play the 30 seconds penalty start counting.
The Game Controller will:

- Penalize the robot as soon as the referee calls the penalty.
- Marks the penalty time counting down as soon as the robot is placed on the penalty position outside the field

The penalty is automatically removed after 30 seconds of penalty have expired.

3.2 Substitutions
Not applicable for simulation

3.3 Temporal Absence
Not applicable for simulation

4 The Design of the Robots
Robots participating in the Humanoid League competitions must have a human-like body plan, as shown in Fig. 2. They must consist of two legs, two arms, and one head, which are attached to a trunk. The robots must be able to stand upright on their feet and to walk on their legs. The only allowed modes of locomotion are bipedal walking and running. -

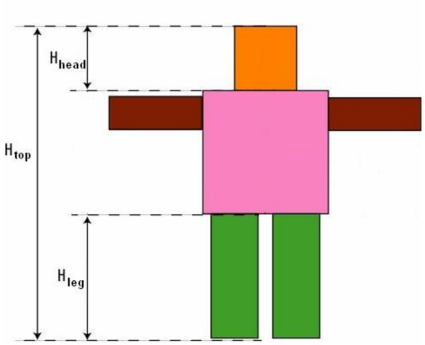


Figure 2: Humanoid robot body plan.

4.1 Robot Height
4.1.1: The height H of a robot is determined as follows:

H = min(Htop, 2.2 · H_{com}), where H_{top} denotes the height of the robot when standing upright and H_{com} denotes the height of the robot's center of mass, measured in upright posture. **4.1.2:** Based on H, the following size restrictions apply:

• 30cm ≤ H ≤ 50cm , ○

4.2 Size Restrictions
All robots participating in the Humanoid League must comply with the following restrictions:

1. Each foot must fit into a rectangle of area $H^2/24$.
2. Considering the rectangle enclosing the convex hull of the foot, the ratio between the longest side of the rectangle and the shortest one, shall not exceed 2.5
3. The robot must fit into a cylinder of diameter 0.55*H.
4. If the arms are maximally stretched in horizontal direction, their extension must be less than 1.2 · H.
5. The robot does not possess a configuration where it is extended longer than 1.5 · H.

6. The length of the legs H_{leg}, including the feet, satisfies $0.35 \cdot H \leq H_{leg} \leq 0.7 \cdot H$.
7. The height of the head H_{head}, including the neck, satisfies $0.05 \cdot H \leq H_{head} \leq 0.25 \cdot H$.

4.3 Sensors

Teams participating in the Humanoid League competitions are encouraged to equip their robots with sensors that have an equivalent in humans. These sensors should be placed at a position roughly equivalent to the human senses. In particular,

1. Any active sensor (emitting light, sound, or electromagnetic waves into the environment in order to measure reflections) is not allowed.
2. External sensors, such as cameras and microphones, may not be placed in the legs or arms of the robots. They should be placed in the robot's head.
3. Touch sensors, force sensors, and temperature sensors may be placed at any position on the robot.
4. Sensors inside the robot may measure all quantities of interest, including (but not limited to) voltages, currents, forces, movements, accelerations, and rotational speeds. They can be at any position inside the robot.

4.4 Communication and Control

4.4.1: Robots participating in the competitions must act autonomously while a competition is running.

4.4.2: Robots may communicate only via the network channel provided by the organizers. The total bandwidth of the robots belonging to one team may not exceed 1MBaud.

4.4.3: Robots of a team may communicate with each other at any time during a game.

4.5 Colors and Markers

4.5.1: Robots must be mostly black. Less than 10% of the total body surface may have a higher reflectance, e.g. gray or white. Less than 1% of the total body surface may be colored. Any color used for the field (green, yellow, blue) or the ball (orange) should be avoided.

4.5.2: The robots must be marked with team markers, attached to the trunk. These markers are colored magenta for one team and cyan for the other team. At least 6cm×6cm
of the team markers should be visible from any side.

4.5.3: The robots of a team must be uniquely identifiable. They should be marked with numbers or names. Each player must have number on its breast and on its back. Player number 1 must be a goalkeeper at beginning of game.

4.6 Safety

4.6.1: Not applicable for simulation

4.6.2: Not applicable for simulation

4.7 Robustness

Robots participating in the Humanoid League competitions must be constructed in a robust way. They must maintain structural integrity during contact with the field, the ball, or other

players. Their sensing systems must be able to tolerate significant levels of noise and disturbance caused by other players, the referees, robot handlers, and the audience.

4.8 Conformity of robot model

Model of robot used for simulation must be in conformity with technical performances of real robots owned by team. In case if team doesn't own any real robot, team can use any robot model which were provided for public usage by community of participating teams which was approved for usage. Any new robot model must be approved for usage by Technical Committee.

5 The Referee

The authority of the referee
Each match is controlled by an autonomous referee who has full authority to enforce the Laws of the Game in connection with the match to which they have been appointed. Decisions will be made to the best of the referee's ability according to the Laws of the Game and the spirit of the game and will be based on the programming of the referee who has the discretion to take appropriate action within the framework of the Laws of the Game.
The games are overseen by the Technical Committee of the league, who ensures that the players and simulated environment is according to the laws of the game, and who may sanction unsupportive behavior by teams.
Powers and duties
The autonomous Referee:
- enforces the Laws of the Game;
- controls the match;
- acts as timekeeper and keeps a record of the match;
- stops, suspends or abandons the match, at their discretion, for any infringements of the Laws;
- punishes the more serious offence when a player commits more than one offence at the same time;
- takes disciplinary action against players guilty of cautionable and sending-off offences. They are not obliged to take this action immediately but must do so when the ball next goes out of play;
- indicates the restart of the match after it has been stopped;
- provides the appropriate authorities with a match report, which includes information on any disciplinary action taken against players and/or team officials and any other incidents that occurred before, during or after the match;
- indicates when the whole of the ball leaves the field of play;
- indicates whether, at penalty kicks, the goalkeeper moves off the goal line before the ball is kicked and if the ball crosses the line;
- communicates its decisions directly to the GameController.

The Technical Committee:
- ensures that any ball used meets the requirements of Law;
- ensures that the players' equipment meets the requirements of Law;
- stops, suspends or abandons the match because of outside interference of any kind;
- takes action against team officials who fail to conduct themselves in a responsible manner and may, at their discretion, expel them from the field of play and its immediate surrounds.

Decisions of the referee

The decisions of the referee regarding facts connected with play, including whether or not a goal is scored and the result of the match, are final.

6 The Assistant Referees

Not applicable for simulation

7 The Duration of the Match

7.1 Periods

7.1.1: The match lasts two equal periods of 5 minutes.

7.1.2: Allowance is made in either period for all time lost through, e.g. substitution(s), timeouts, and wasting time. The allowance for time lost is at the discretion of the Human referee.

7.1.3: In the knock-out games of a tournament two further equal periods of 3 minutes each are played if the game is not decided after the regular playing time. If during regular playing time none of the two teams in a knock-out match was able to kick the ball to reach their respective opponent's goal the extra time is skipped and the game immediately continues by the five alternating penalty kick trials (cf. Section 14).

7.2 Timeouts

suspended

8 The Start and Restart of Play

8.1 Preliminaries

8.1.1: Teams have to upload their robots' proto file at least 1 week before game, controller software and their "team.json" file at least 20 minutes prior to the scheduled kick-off time. In case if team is going to use standard robots' proto file "Robokit1.proto" then robots' proto will be used from referees' computer. A decision about which team which goal will attack in the first half of the match, which team takes the kick-off to start the match will be taken by referee computer through random algorithm. In the second half of the match the teams change ends and attack the opposite goals. Kick-off of ball in second half of match is passed to team other than kick-off team of first half. Human handler of referees' computer downloads teams' controller software to "controller" directory, copy "team.json" file into "referee" directory. Human handler edit "game.json' file in order to input data about teams file, teams color, teams starting side and initial kick-off team. All other files in referee's computer must be same as in Elsiros distribution package in order to provide same game environment as in teams' home computers.

8.1.2: Color of team markers for each match in tournament have to be selected by random algorithm at beginning of tournament.

8.1.3: A match must start at the scheduled time. In exceptional situations only, the human referee may re-adjust the time for starting the game in accordance with both team leaders. All robots of a team are started (and stopped) by receiving a signal from Game Controller.

8.2 Kick-off

8.2.1: A kick-off is a way of starting or restarting play at the start of the match, after a goal has been scored, at the start of the second half of the match, at the start of each period of extra time, where applicable. After a team scores a goal, the kick-off is taken by the other team.
8.2.2: A goal may not be scored directly from the kick-off. Either the ball must move 20cm from the kick-off point or must be touched by another player before being kicked towards the goal.
If the ball is kicked directly towards the goal the kick-off is awarded to the opposing team.

8.2.3: The procedure for kick-off is as follows:

- All players are in their own half of the field.
- The opponents of the team taking the kick-off are outside the center circle until the ball is in play.
- The ball is stationary on the center mark.
- The referee gives a signal.
- The ball is in play when it is touched or 10 seconds elapsed after the signal.

8.2.4: Robots being able to autonomously reposition themselves can take any position on the field that is consistent with above requirements. Robots not able to autonomously reposition themselves will be positioned to ready starting position by autonomous referee.

8.2.5: The game state can turn to following states:

STATE_INITIAL: Auto referee places players to border starting positions.

STATE_READY: will be eliminated if no one team shows capability to reposition themselves. In case if one team is capable to reposition to ready starting position then this sate lasts 45 seconds allowing to team to take position.

STATE_SET : Auto referee places players to ready starting position or penalize players which have taken illegal starting position. Legal starting position is located in any point at own half of field of team outside of central circle and outside of goal area.

STATE_PLAYING : state when players allowed to play game.

STATE_FINISHED: state when all players are switched off, goals are not scored

8.3 Dropped Ball

8.3.1: A dropped ball is a way of restarting the match after a temporary stoppage which becomes necessary, while the ball is in play, for any reason not mentioned elsewhere in the rules. In particular, the referee may call a game-stuck situation if there is no progress of the game for 60s.

8.3.2: The game is continued at the center mark. A goal can be scored directly from a dropped ball. The procedure for dropped ball is the same as for kick-off, except that the robots of both teams must be outside the center circle (or at or behind restart positions if positioned manually). The ball is in play immediately after the referee gives the signal.

8.3.3: If a player moves too close to the ball before the referee gives the signal, a kick-off is awarded to the opponent team.

9 The Ball In and Out of Play

9.1: The ball is out of play when it has wholly crossed the goal line or touch line whether on the ground or in the air or when play has been stopped by the referee.

9.2: The ball is in play at all other times, including when it rebounds from a goalpost, crossbar, corner pole and remains in the field of play.

10 The Method of Scoring

10.1: A goal is scored when the whole projection of the ball on floor passes over the goal line, between the goalposts and under the crossbar, provided that no infringement of the rules has been committed previously by the team scoring the goal.

10.2: The team scoring the greater number of goals during a match is the winner. If both teams score an equal number of goals, or if no goals are scored, the match is drawn.

10.3: For knock-out matches ending in a draw after regular time, extra time, penalty kicks, and scoring times will be used to determine the winner of a match.

10.4: An abandoned match is replayed unless the league organization committee decides otherwise. If a team chooses to forfeit a match, the result will be 10:0 against the team that forfeited. Teams may choose to forfeit games at any stage prior to the end of the game.

11 Offside

The offside rule is not applied.

12 Fouls and Misconduct

12.1 Ball Manipulation

Not applied

12.2 Physical Contact

Contact between robot players is guided by the following principles:

1. Physical contact between players of different teams should be minimized.
2. If physical contact is unavoidable, the faster moving robot should make efforts to minimize the impact. The goal keeper enjoys special protection inside its goal area. The attacking player always has to avoid contact with the goalie.
3. Extended physical contact should be avoided. Both robots should make efforts to terminate contact, if the contact time exceeds 1s.
4. If entangled robots fail to untangle themselves, the referee might decide to penalize one of robots or both robots together. In this case robot approaching from far distance must be considered as “guilty”, and robot located nearer to ball must be considered as “innocent”.

12.3 Attack and Defense

12.3.1: Not more than one robot of each team should be inside the goal or the goal area at any time. If more than one robot of the defending team is inside its goal or goal area for more than 10s, this will be considered illegal defense. If more than one robot of the attacking team is inside the opponent's goal or goal area for more than 10s, this will be considered illegal attack.

12.3.2: The referee may delay the call of illegal defense or illegal attack if the robots make serious attempts to leave the goal area or if they are hindered from leaving the goal area by robots of the opponent team.

12.4 Indirect Free Kick

Not applied

12.5 Yellow and Red Cards

12.5.1: A player is cautioned and shown the yellow card if he commits any of the following offenses:

1. is guilty of unsporting behavior,
2. persistently infringes the rules,
3. delays the restart of play,
4. fails to respect the required distance when play is restarted with a free kick.

12.5.2: A player is sent off the field and shown the red card if he commits any of the following offenses:

1. is guilty of serious foul play,
2. is guilty of violent conduct,
3. receives a second caution in the same match.

13 The Free Kicks

Not applied

14 The Penalty Kick

14.1: A goal may be scored directly from a penalty kick.

14.2: The player taking the penalty kick is placed at a distance of at least $1.5 \cdot H$ from the penalty mark.

14.3: The defending goalkeeper is placed in upright position on the middle of his goal line, facing the kicker. It must remain upright between the goalposts until the ball has been touched by the kicker.

14.4: No other players are allowed on the field.

14.5: When both players are ready, the ball is placed randomly within 20cm of the penalty mark.

14.6: After the referee gives the start signal, the striker has 60s to kick the ball once or multiple times. After this time, the trial ends if the movement of the ball obviously does not result in a goal. Otherwise, the trial is extended until the ball stops.

14.7: The striker is not allowed to touch the ball during this extension. The striker is also not allowed to touch the ball after the ball has been touched by the goalie.

14.8: The goalie is not allowed to move forward or to fall until the ball is touched by the striking robot.

14.9: Both robots are not allowed to touch or cross the line around the goal area.

14.10: If the goalie robot violates the rules in any way, the referee will let the trial continue. If the striker robot scores a goal, then the goal counts. If the striker does not score a goal, the trial is retaken. If the goalie violates the rules after causing two restarts, a technical goal is awarded to the striker.

14.11: If the striker violates the rules in any way, the referee will let the trial continue. If the striker robot is unable to score a goal, the trial ends. If the striker scored, the trial is retaken without counting the goal. If the striker violates the rules after causing two restarts, the trial will end with 'no goal'.

14.12: Both teams conduct five alternating trials.

- If after the first five trials none of the teams was able to kick the ball to the goal line then the winner is determined by flipping a coin.
- If there is still a draw in knock-out games, the alternating trials continue up to five more times, until one teams leads after an equal number of trials.
- If there is still a draw in knock-out games, the alternating trials continue up to five more times without goalies, until either one striker is able to score and the other striker fails to score or both strikers score. In the latter case, the goal is awarded to the striker that needed the shortest time for scoring.
- If there is still a draw in knock-out games, the winner is determined by flipping a coin.

15 The Throw-In

A throw-in is necessary if the ball leaves the field of play, by fully crossing a touch line or a goal line (outside the goal posts or above the cross bar) either on the ground or in the air. Without stopping play, one of the assistant referees places the ball at one of the three restart points that are on the same side, where the ball left the field.

- The ball is placed at the restart point closest to a goal, if a player of the team defending this goal was touched last by the ball before it went out on the same half of the field.
- The ball is placed at the restart point on the middle line in all other cases.

If a robot obstructs the restart point, the ball is placed at the next empty spot found by moving from the restart position towards the closer touch line.

16 The Goal Kick

The goal kick is performed without stopping play according to the throw-in procedure.

17 The Corner Kick

The corner kick is performed without stopping play according to the throw-in procedure.

18 deleted

19 The Technical Challenge

suspended

20 The Competitions and Trophies

20.1 Setup and Inspections

The competitions in the Humanoid League are preceded by a setup and inspection period of at least 1 week. During this time, every robot's proto presented by teams will be inspected by the league organizing committee for compliance with the design rules detailed in Section 4. All robot protos must be designed in strict technical conformity to real robots owned by teams. If team decided to use standard robot's proto then inspection is not needed.

20.2 Referee Duty

Each team must name at least one person who is familiar with the rules and who might be assigned for referee duties by the league organizing committee.

20.3 Competitions

20.3.1: The competitions consist of Soccer Games
20.3.2: Soccer Games are organized in one or more round robins and playoffs. For the first round robin, the teams are assigned to groups at random. All teams of a group play once against each other. The round robin games may end in a draw. In this case, both teams receive one point. Otherwise, the winning team receives three points and the not winning team receives zero points.
20.3.3: After games of a round robin have been played, the teams of a group are ranked based on (in decreasing priority):

1. the number of earned points,
2. the goal-difference,
3. the absolute number of goals,
4. the result of a direct match,
5. the time needed to score a penalty kick into an empty goal (up to five alternating attempts to score, until at least one team scored),
6. the drawing of a lot.

20.3.4: At least two teams of every group will enter the next round robin or the playoffs.

20.3.5: The game plan needs to be announced prior to the random assignment of teams to groups.

Appendix A

The trend in rule evolution for the next years

In this section to make explicit the trends to be followed in the rules in the next year in order to improve the scientific level of the robots developed by the RoboCup teams.

In case if virtual simulation games attracts many participants then they can be developed to bigger size of field and bigger number of players in order to implement team play strategies.

Acknowledgements

These rules evolved from version of the RoboCup Humanoid League rules designed for 2007 competition by Emanuele Menegatti. The 2006 version of the rules was edited by Sven Behnke, who did a terrific job improving this rule document. The 2005 version was edited by Norbert Michael Mayer. Other input came from the FIFA laws of the game and the rules of the RoboCup MiddleSize and Four-legged leagues.

The rules were discussed within the technical and organizing committees of the league and also on the Humanoid list. Among others, Minoru Asada, Jacky Baltes, Hans-Dieter Burkhard, Davide Faconti, Rodrigo Guerra, Bernhard Hengst, Hiroshi Ishiguro, Damien Kee, Yue Pik Kong, Pasan Kulvanit, Norbert Michael Mayer, Emanuele Menegatti, Chew Chee Meng, Masaki Ogino, Maziar Palhang, Thomas Röfer, Philippe Schober, Naoki Shibatani, Oskar von Stryk, Ashfaque Ur-Rahman, Gerald Weratschnig, Shuzo Yumoto, and Changjiu Zhou contributed to the discussion.

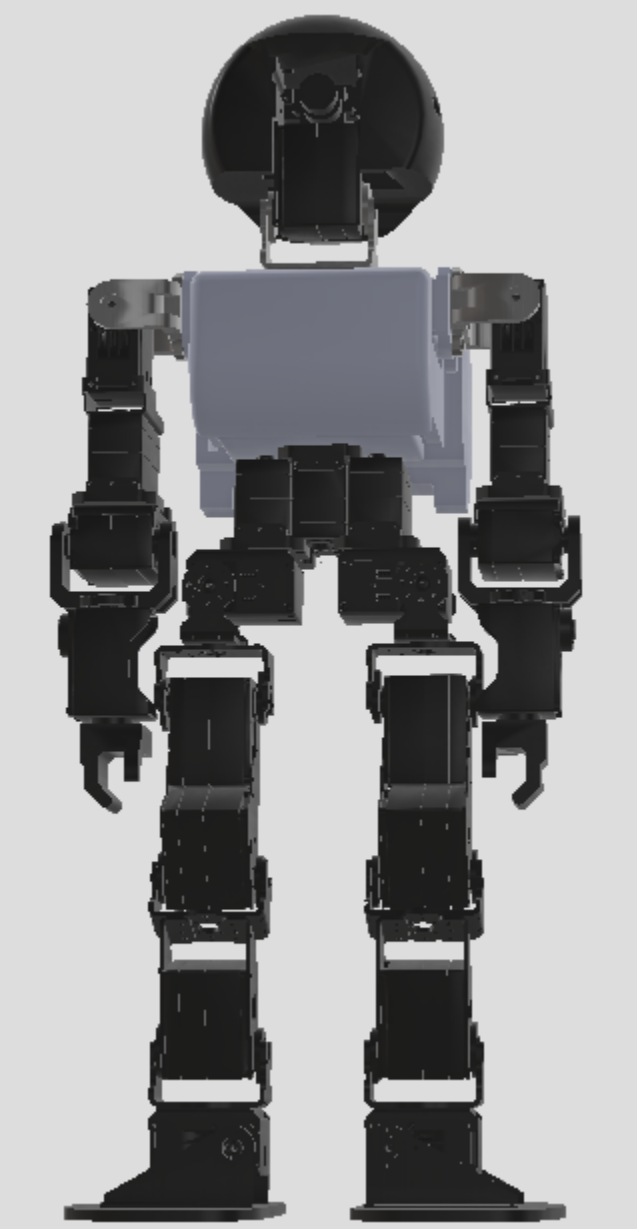
The rules were updated for virtual games in simulation with borrowing some parts from Virtual RoboCup Soccer Humanoid League Laws of the Game 2020/2021. The following members of the technical committee for 2021 were responsible for creating the first version of the rules for the virtual Humanoid RoboCup league: Jacky Baltes, Reinaldo Bianchi, Reinhard Gerndt, Wang Hao, Ludovic Hofer, Maike Paetzel and Soroush Sadeghnejad.

CHAPTER

FIVE

ROBOT DESIGN

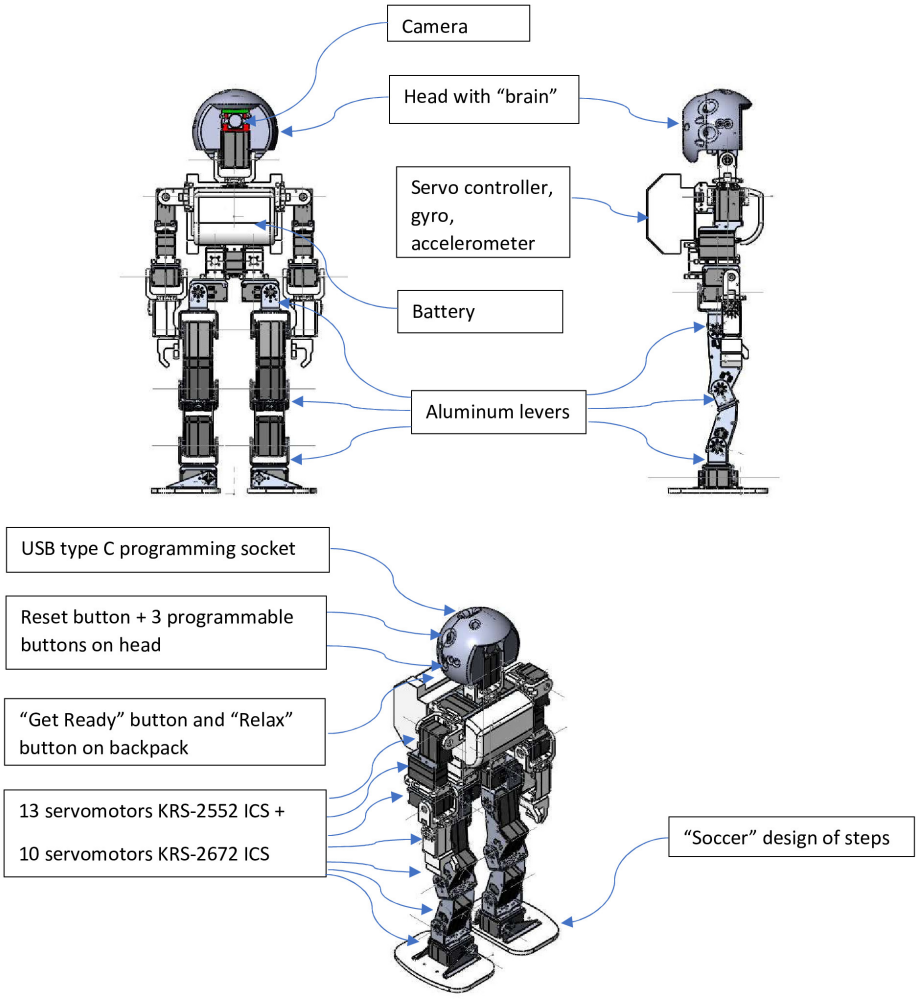
Standard robot proto used for simulation is Robokit1.proto. It is designed as virtual copy of real robot. In order to represent real robot all technical features of virtual model were copied from technical description of real robot including servo torque, servo speed, servo mass, location of camera, camera resolution, aperture of camera lens, number and location of accelerometers, weight distribution throughout of body.



Virtual model has 2 IMUs named “imu_head” and “imu_body”. Real robot is capable to recognize coordinate of itself at soccer field with accuracy ±15cm, it is capable to recognize obstacles and ball. Recognition of distance and course to ball and obstacles is made through machine vision algorithms. Robot detectds distance and course to object with accuracy mainly dependent on camera accuracy. Good calibrated robot detects course to object with accuracy ± 0.01 Radian. Accuracy of distance recognition non-linearly depends on distance to object, with best accuracy at minimum distances which is ± 5mm. Distance measurement accuracy at distance 1m is ± 25mm, at distance 2m is ± 100mm High accuracy of distance and course measurement are provided due to equipment and smart algotihms used for direct measurements. Localization on soccer field is provided by measurements of cource and distence to goal posts, field marking, green field border, odometry, measurement of IMU. All localization data is accumulated with historical data and processed by patricle filter algorithm in order to achieve better accuracy than measurement of individual object. Simulation procedure in Webots is orgaised in way when simulation of physics and motion is provided together with camera image scanning. Procedure of image scanning and transfer to outside from Webots greatly reduces simulation speed. In order to keep simulation speed at acceptable rate it was decided to skip procedure of scanning images by camera in simulation and replace it with direct request- report procedure. Following data are reported to Robokit1 robot from simulation by request: distance and course from robot to ball, distance and course from robot to other players, self

coordinate and orientation on field. Requested data before reporting to robot are mixed with random values in order to provide same accuracy which usually real robot detects from camera image. Above procedure is called “blurrer”. Above procedure provide increasing is simulation speed up to 10 times. This know-how of ELSIROS provides games to be played at laptop computer with nearly realtime speed.

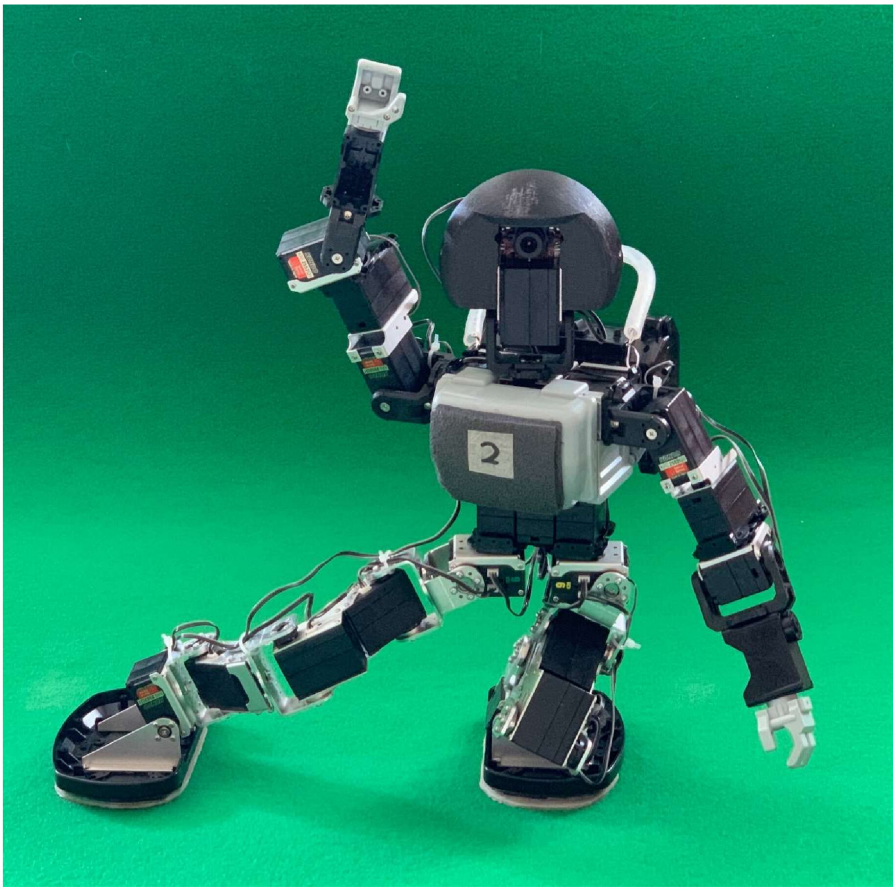
Below is technical description of real Robokit robot.



Robokit Robot

ROBOKIT Robot Specification:

- Height 45 cm
- Weight 1.9 kg
- Battery voltage 12 V
- 23 DOF: 13 servomotors KRS-2552 ICS +10 servomotors KRS-2672 ICS
- Main controller: OpenMV H7 with 32-Bit Arm Cortex-M7 operating at 400MHz with 1Mb SRAM
- Motion controller: Kondo RCB-4HV
- Programming language: Micropython.
- Sensors: OV7725 640x480 camera, 6D digital IMU BNO055, 2D analogue Gyro, 3D analogue Accelerometer



CHAPTER

SIX

API

6.1 SAMPLE_TEAM.Soccer.Motion

6.1.1 Subpackages

SAMPLE_TEAM.Soccer.Motion.motion_slots

6.1.2 Submodules

SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq

Module Contents

Functions

uprint(*text)	
normalize_rotation(yaw)	
steps(motion, x1, y1, u1, x2, y2, u2)	
ball_Approach(motion, local, glob, ball_coord)	

SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.**uprint**(**text*)

SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.**normalize_rotation**(*yaw*)

SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.**steps**(*motion*, *x1*, *y1*, *u1*, *x2*, *y2*, *u2*)

SAMPLE_TEAM.Soccer.Motion.ball_Approach_Steps_Seq.**ball_Approach**(*motion*, *local*, *glob*, *ball_coord*)

SAMPLE_TEAM.Soccer.Motion.ball_Approach_calc

Module Contents

Functions

uprint(*text)
ball_Approach_Calc(glob, ball_coord)

SAMPLE_TEAM.Soccer.Motion.ball_Approach_calc.**uprint**(**text*)

SAMPLE_TEAM.Soccer.Motion.ball_Approach_calc.**ball_Approach_Calc**(*glob*, *ball_coord*)

SAMPLE_TEAM.Soccer.Motion.class_Motion

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.
This module contains walking engine

Module Contents

Classes

Motion1

class SAMPLE_TEAM.Soccer.Motion.class_Motion.**Motion1**(*glob*)

imu_body_yaw(*self*)

norm_yaw(*self*, *yaw*)

quaternion_to_euler_angle(*self*, *quaternion*)

play_Soft_Motion_Slot(*self*, *name=''*)

computeAlphaForWalk(*self*, *sizes*, *limAlpha*, *hands_on=True*)

activation(*self*)

walk_Initial_Pose(*self*)

walk_Cycle(*self*, *stepLength*, *sideLength*, *rotation*, *cycle*, *number_Of_Cycles*)

walk_Final_Pose(*self*)

kick(*self*, *first_Leg_Is_Right_Leg*, *small=False*)

refresh_Orientation(*self*)

SAMPLE_TEAM.Soccer.Motion.class_Motion_Webots_PB

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.
The module is designed to provide communication from motion controller to simulation

Module Contents

Classes

Motion_sim

class SAMPLE_TEAM.Soccer.Motion.class_Motion_Webots_PB.**Motion_sim**(*glob*, *robot*, *gcreceiver*, *pause*, *logger*)

Bases: *SAMPLE_TEAM.Soccer.Motion.class_Motion_real.Motion_real*

game_time(*self*)

game_time_ms(*self*)

pause_in_ms(*self*, *time_in_ms*)

sim_Trigger(*self*, *time*)

wait_for_step(*self*, *step*)

imu_activation(*self*)

read_head_imu_euler_angle(*self*)

read_imu_body_yaw(*self*)

falling_Test(*self*)

send_angles_to_servos(*self*, *angles*, *use_step_correction=False*)

move_head(*self*, *pan*, *tilt*)

simulateMotion(*self*, *number=0*, *name=''*)

sim_Get_Ball_Position(*self*)

sim_Get_Obstacles(*self*)

sim_Get_Robot_Position(*self*)

sim_Start(*self*)

sim_Progress(*self*, *simTime*)

SAMPLE_TEAM.Soccer.Motion.class_Motion_real

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.

The module is a part of motion generating functions

Module Contents

Classes

Motion_real

class SAMPLE_TEAM.Soccer.Motion.class_Motion_real.**Motion_real**(*glob*)

Bases: *SAMPLE_TEAM.Soccer.Motion.class_Motion.Motion1*

seek_Ball_In_Pose(*self*, *fast_Reaction_On*, *penalty_Goalkeeper=False*, *with_Localization=True*)

watch_Ball_In_Pose(*self*, *penalty_Goalkeeper=False*)

seek_Ball_In_Frame(*self*, *with_Localization=True*)

detect_Ball_Speed(*self*, *with_Localization=False*)

see_ball_confirmation(*self*)

turn_To_Course(*self*, *course*, *accurate=False*)

head_Up(*self*)

head_Return(*self*, *old_neck_pan*, *old_neck_tilt*)

localisation_Motion(*self*)

normalize_rotation(*self*, *yaw*)

near_distance_omni_motion(*self*, *dist_mm*, *napravl*)

near_distance_ball_approach_and_kick(*self*, *kick_direction*, *strong_kick=False*, *small_kick=False*)

far_distance_ball_approach(*self*, *ball_coord*)

far_distance_plan_approach(*self*, *ball_coord*, *target_yaw*, *stop_Over=False*)

SAMPLE_TEAM.Soccer.Motion.compute_Alpha_v3

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. Module can be used for Inverted Kinematics for legs of Robokit-1 robot. Advantage of module against other IK implementations is fast and repeatable calculation benchmark. Result is achieved due to mixed analytic/numerical calculation method. Module is designed for 6 DOF robot leg. From 6 angles one angle is calculated using numerical iterations, other 5 angles are obtained through polynom roots formula calculation. This way prowides fast benchmark and repeatability. Algorithm being implemented in C language with integration into firnware of OpenMV is capable to calculated angles for robot legs within time less than 1ms. Multiple IK solutions are filtered through applying of angle limits within calculation. This yields less time for calculation. usage: create class Alpha instance and call method compute_Alpha_v3 with arguments. Returns list of 0, 1 or 2 lists of servo angles. List of 0 elements means that IK was not solved. List of 1 list means 1 possible solition is detected. List of 2 lists means that plurality of solutions was not filtered by provided arguments.

Module Contents

Classes

Alpha

Attributes

a5

class SAMPLE_TEAM.Soccer.Motion.compute_Alpha_v3.**Alpha**

compute_Alpha_v3(*self*, *xt*, *yt*, *zt*, *x*, *y*, *z*, *w*, *sizes*, *limAlpha*)

usage: list: angles = self.compute_Alpha_v3(float: xt, float: yt, float: zt, float: x, float: y, float: z, float: w, list: sizes, list: limAlpha)

angles: list of floats angles in radians of servos which provide target positioning and orientation of robots' foot

xt: target x coordinate of foots' center point yt: target y coordinate of foots' center point zt: target z coordinate of foots' center point x: x coordinate of vector of orientation of foot y: y coordinate of vector of orientation of foot z: z coordinate of vector of orientation of foot w: rotation in radians of foot around vector of orientation sizes: list of sizes defining distances between servo axles in biped implementation limAlpha: list of limits [minimum, maximum] of servomotors measured in number of encoder ticks

of Kondo series 2500 servomotors.

Target coordinates are measured in local robot coordinate system XYZ with ENU orientation. [0,0,0] point of coordinate system is linked to pelvis of robot. Foot orientation vector has length 1. Base of vector is at bottom of foot and tip of vector is directed down when foot is on floor.

SAMPLE_TEAM.Soccer.Motion.compute_Alpha_v3.**a5 = 21.5**

SAMPLE_TEAM.Soccer.Motion.path_planning

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev. module can be used for optimized path planing of Robokit-1 robot. usage: create class PathPlan type object instance and call method path_calc_optimum. Optionally module can be launched stand alone for purpose of tuning and observing result of path planing. Being launched stand alone module draws soccer field with player (white circle), ball (orange circle), obstacles (black circles). Circles are movable by mouse dragging. After each stop of mouse new path is drawing.

Module Contents

Classes

PathPlan	Plans optimized path of humanoid robot from start coordinate to target coordinate.
Glob	

Attributes

goalPostRadius
ballRadius
uprightRobotRadius
roundAboutRadiusIncrement

SAMPLE_TEAM.Soccer.Motion.path_planning.**goalPostRadius = 0.15**

SAMPLE_TEAM.Soccer.Motion.path_planning.**ballRadius = 0.1**

SAMPLE_TEAM.Soccer.Motion.path_planning.**uprightRobotRadius = 0.2**

SAMPLE_TEAM.Soccer.Motion.path_planning.**roundAboutRadiusIncrement = 0.15**

class SAMPLE_TEAM.Soccer.Motion.path_planning.**PathPlan**(*glob*)

Plans optimized path of humanoid robot from start coordinate to target coordinate. Coordinates are taken together with orientation. Path is composed from initial arc, final arc and connecting line. Connecting line must be tangent to arcs. In case of obstacles on path line additional arc is added in order to go around obstacle. Only one obstacle can be avoided reliably. Avoiding of second obstacle is not guaranteed. Therefore there are used evaluations of prices of variants of path. The Path with cheaper price is returned. Collision with obstacle in far distance is cheaper than collision with obstacle in near distance. During Path heuristic various radiuses of arcs are considered. Arc with zero radius means turning without changing coordinate.

coord2yaw(*self*, *x*, *y*)

intersection_line_segment_and_line_segment(*self*, *x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*)

Checks if 2 line segments have common point. :returns: True - if there is common point

False - if not.

x = x1 + (x2 - x1) * t1 t1 - paramentric coordinate y = y1 + (y2 - y1) * t1 x = x3 + (x4 - x3) * t2 t2 - paramentric coordinate y = y3 + (y4 - y3) * t2 x1 + (x2 - x1) * t1 = x3 + (x4 - x3) * t2 y1 + (y2 - y1) * t1 = y3 + (y4 - y3) * t2 t1 = (x3 + (x4 - x3) * t2 - x1) / (x2 - x1) t1 = (y3 + (y4 - y3) * t2 - y1) / (y2 - y1) y1 + (y2 - y1) * (x3 + (x4 - x3) * t2 - x1) / (x2 - x1) = y3 + (y4 - y3) * t2 (y2 - y1) * (x4 - x3)/ (x2 - x1) * t2 - (y4 - y3) * t2 = y3 - y1 - (y2 - y1) * (x3 - x1) / (x2 - x1) t2 = (y3 - y1 - (y2 - y1) * (x3 - x1) / (x2 - x1)) /((y2 - y1) * (x4 - x3)/ (x2 - x1) - (y4 - y3)) if t1 == 0:

t2 = (y1 - y3)/ (y4 - y3) t2 = (x1 - x3)/ (x4 - x3)

intersection_line_segment_and_circle(*self*, *x1*, *y1*, *x2*, *y2*, *xc*, *yc*, *R*)

Checks if line segment and circle have common points. :returns: True - if there is common point

False - if not.

x = x1 + (x2 - x1) * t t - paramentric coordinate y = y1 + (y2 - y1) * t R**2 = (x - xc)**2 + (y - yc)**2 (x1 + (x2 - x1) * t - xc)**2 + (y1 + (y2 - y1) * t - yc)**2 - R**2 = 0 ((x2 - x1) * t)**2 + (x1 - xc)**2 + 2 * (x2 - x1) * (x1 - xc) * t + ((y2 - y1) * t)**2 + (y1 - yc)**2 + 2 * (y2 - y1) * (y1 - yc) * t - R**2 = 0 ((x2 - x1)**2 + (y2 - y1)**2) * t**2 + (2 * (x2 - x1) * (x1 - xc) + 2 * (y2 - y1) * (y1 - yc)) * t + (x1 - xc)**2 + (y1 - yc)**2 - R**2 = 0 a * t**2 + b * t + c = 0 a = (x2 - x1)**2 + (y2 - y1)**2 b = 2 * (x2 - x1) * (x1 - xc) + 2 * (y2 - y1) * (y1 - yc) c = (x1 - xc)**2 + (y1 - yc)**2 - R**2

intersection_circle_segment_and_circle(*self*, *x1*, *y1*, *x2*, *y2*, *x0*, *y0*, *CW*, *xc*, *yc*, *R*)

norm_yaw(*self*, *yaw*)

delta_yaw(*self*, *start_yaw*, *dest_yaw*, *CW*)

path_calc_optimum(*self*, *start_coord*, *target_coord*)

Returns optimized humanoid robot path. usage:

list: dest, list: centers, int: number_Of_Cycles = self.path_calc_optimum(list: start_coord, list: target_coord) dest: list of destination point coordinates. Each coordinate is list or tuple of floats [x,y].

Each coordinate is starting or end point of path segment. Path comprises of following segments: circle segment, line segment, n*(circle segment, line segment), circle segment. Where n - iterable.

centers: list of coordinates of circle centers of circle segments of path. Each coordinate is list or tuple of floats [x,y]. number_Of_Cycles: integer which represents price of path. In case if value is >100 then collision with second obstacle on path

is not verified.

start_coord: list or tuple of floats [x, y, yaw] target_coord: list or tuple of floats [x, y, yaw]

path_calc(*self*, *start_coord*, *target_coord*)

arc_path_external(*self*, *x1*, *y1*, *yaw1*, *x2*, *y2*, *yaw2*)

check_Obstacle(*self*, *xp1*, *yp1*, *xp2*, *yp2*)

check_Limits(*self*, *x1*, *y1*, *x2*, *y2*, *xp1*, *yp1*, *xp2*, *yp2*, *xc1*, *yc1*, *CW1*, *xc2*, *yc2*, *CW2*, *dest*)

check_Price(*self*, *x1*, *y1*, *x2*, *y2*, *xp1*, *yp1*, *xp2*, *yp2*, *xc1*, *yc1*, *CW1*, *xc2*, *yc2*, *CW2*, *dest*, *centers*)

number_Of_Cycles_count(*self*, *dest*, *centers*, *yaw1*, *yaw2*)

external_tangent_line(*self*, *start*, *R1*, *R2*, *x1*, *y1*, *xc1*, *yc1*, *xc2*, *yc2*, *CW*)

arc_path_internal(*self*, *x1*, *y1*, *yaw1*, *x2*, *y2*, *yaw2*)

internal_tangent_line(*self*, *start*, *R1*, *R2*, *x1*, *y1*, *xc1*, *yc1*, *xc2*, *yc2*, *CW*)

square_equation(*self*, *a*, *b*, *c*)

class SAMPLE_TEAM.Soccer.Motion.path_planning.**Glob**

import_strategy_data(*self*, *current_work_directory*)

6.2 SAMPLE_TEAM.Soccer.Localisation

6.2.1 Submodules

SAMPLE_TEAM.Soccer.Localisation.class_Glob

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.

This module is used to store variables which are used in many classes

Module Contents

Classes

Glob

class SAMPLE_TEAM.Soccer.Localisation.class_Glob.**Glob**(*simulation*, *current_work_directory*)

import_strategy_data(*self*, *current_work_directory*)

SAMPLE_TEAM.Soccer.Localisation.class_Local

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of A. Babaev.

This module is assisting localization

Module Contents

Classes

Local

class SAMPLE_TEAM.Soccer.Localisation.class_Local.**Local**(*logger*, *motion*, *glob*, *coord_odometry=[0.0, 0.0, 0.0]*)

coordinate_fall_reset(*self*)

coordinate_trust_estimation(*self*)

normalize_yaw(*self*, *yaw*)

correct_yaw_in_pf(*self*)

coordinate_record(*self*)

localisation_Complete(*self*)

group_obstacles(*self*)

read_Localization_marks(*self*)

6.3 SAMPLE_TEAM.Soccer.strategy

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. The module is designed for strategy of soccer game for forward and goalkeeper.

6.3.1 Module Contents

Classes

GoalKeeper	class GoalKeeper is designed to define goalkeeper's play according to style developed by
Forward	The class Forward is designed for definition of strategy of play for 'forward' role of player
Forward_Vector_Matrix	The class Forward_Vector_Matrix is designed for definition of strategy of play for 'forward' role of player
Player	class Player is designed for implementation of main cycle of player.

class SAMPLE_TEAM.Soccer.strategy.**GoalKeeper**(*logger*, *motion*, *local*, *glob*)

class GoalKeeper is designed to define goalkeeper's play according to style developed by Matvei Ivaschenko - a student of Phystech Lyceum in 2020. Idea of style was in dividing of home half of shoccer field to 8 sectors according to distance from home goals. When ball is in 4 sectors closest to goals A1, A2, A3, A4 goalkeeper attacks ball with purpose transfer it to side of opponent. When ball is in 4 sectors B1, B2, B3, B4 which are in longer distance from goals goalkeeper just slide to better position from current without attempt to attack ball. In case if ball didn't go longer than 1m after kick of goalkeeper, he will undertake another attempt up to 10 times in total. In case if ball goes to distance longer than 1m or ball can't be seen by goalkeeper then goalkeeper returns to center of goals.

turn_Face_To_Guest(*self*)

The method is designed to define kick direction and load it into self.direction_To_Guest direction is measured in radians of yaw. After definition of kick direction robot turns to this direction. In case if robots' own coordinate self.glob.pf_coord shows lication on own half of field i.e. self.glob.pf_coord[0] < 0 then direction of shooting is 0 if robots' x coordinate > 0.8 and abs(y coordinate) > 0.6 then direction of kick is to center of opponents' goals. if robots' x < 1.5 and abs(y) < 0.25 kick direction will be to corner of opponents' goals, with left or right corner is defined randomly. in all other positions of robot kick direction is defined as ditection to target point with coordinates x = 0, y = 2.8

goto_Center(*self*)

Goalkeeper returns to duty position 0.4m in front of own goals. before returning robot checks trustability of localization. If localization is poor then robot undertake special motions by head and by turning to goals with purpose to improve localization. In case if distance to duty position is more than 0.5m then far_distance_plan_approach will be used, else near_distance_omni_motion will be used. after returning to duty position robot turns to kick direction, for which yaw=0 in front of own goals.

find_Ball(*self*)

Before using motion method seek_Ball_In_Pose goalkeeper define usage of method in quick mode or in accurate mode. In case if localization is trustable quick mode is used means fast_Reaction_On=True. seek_Ball_In_Pose method moves head of robot to 15 positions covering all visible areas in front and in sides of robot. this way seeking of ball is not single task. Robot improves localization through ob-

serving localization markers on obtained pictures. In case if fast_Reaction_On=True then observation of surroundings will be interrupted as soon as ball appear in visible sector. Speed of ball is also detected.

scenario_A1(*self*, *dist*, *napravl*)

This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents' goals before seeking ball. usage:

None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

scenario_A2(*self*, *dist*, *napravl*)

This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from math.pi/4 to math.pi/2. Supposed that goalkeeper stands on duty position faced to opponents' goals before seeking ball. usage:

None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

scenario_A3(*self*, *dist*, *napravl*)

This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from 0 to -math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents' goals before seeking ball. usage:

None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

scenario_A4(*self*, *dist*, *napravl*)

This method is activated if goalkeeper finds ball at distance less than 0.7m and relative direction from -math.pi/4 to -math.pi/2. Supposed that goalkeeper stands on duty position faced to opponents' goals before seeking ball. usage:

None: self.scenario_A1(float:dist, float: napravl) dist - distance to ball from goalkeeper in meters napravl - relative direction to ball from goalkeeper in radians

method undertake 10 attempts to kick off ball to opponents side. In case of successful attempt - ball goes 1m away from goalkeeper - goalkeeper returns to duty position in front of own goals. Otherwise attempts are continued up to 10 times.

scenario_B1(*self*)

This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents' goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls' Y coordinate. In case if balls' Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

scenario_B2(*self*)

This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced

to opponents' goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls' Y coordinate. In case if balls' Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

scenario_B3(*self*)

This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents' goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls' Y coordinate. In case if balls' Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

scenario_B4(*self*)

This method is activated if goalkeeper finds ball at distance more than 0.7m and less than half of length of field and relative direction from 0 to math.pi/4. Supposed that goalkeeper stands on duty position faced to opponents' goals before seeking ball. method undertake to slide robot sideways to same Y coordinate as balls' Y coordinate. In case if balls' Y coordinate abs value is more than 0.4m robots maximum Y coordinate abs value will be 0.4m After sliding sideways robot undertake turning to 0 direction

class SAMPLE_TEAM.Soccer.strategy.**Forward**(*logger*, *motion*, *local*, *glob*)

The class Forward is designed for definition of strategy of play for 'forward' role of player in year 2020. usage:

Forward(object: motion, object: lical, object: glob)

dir_To_Guest(*self*)

The method is designed to define kick direction and load it into self.direction_To_Guest, direction is measured in radians of yaw. In case if robots' own coordinate self.glob.pf_coord shows lication on own half of field i.e. self.glob.pf_coord[0] < 0 then direction of shooting is 0 if robots' x coordinate > 0.8 and abs(y coordinate) > 0.6 then direction of kick is to center of opponents' goals. if robots' x < 1.5 and abs(y) < 0.25 kick direction will be to corner of opponents' goals, with left or right corner is defined randomly. in all other positions of robot kick direction is defined as ditection to target point with coordinates x = 0, y = 2.8 returns float: self.direction_To_Guest

turn_Face_To_Guest(*self*)

class SAMPLE_TEAM.Soccer.strategy.**Forward_Vector_Matrix**(*logger*, *motion*, *local*, *glob*)

The class Forward_Vector_Matrix is designed for definition of strategy of play for 'forward' role of player in year 2021. Matrix is coded in file strategy_data.json This file is readable and editable as well as normal text file. There is a dictionary with one key "strategy_data". Value of key "strategy_data" is a list with default number of elements 234. Each element of list represents rectangular sector of soccer field with size 20cmX20cm. For each sector there assigned a vector representing yaw direction of shooting when ball is positioned in this sector. Power of shot is coded by attenuation value: 1 – standard power, 2 – power reduced 2 times, 3- power reduced 3 times. Each element of list is coded as follows: [column, row, power, yaw]. Soccer field is split to sectors in 13 rows and 18 columns. Column 0 is near own goals, column 17 is near opposed goals. Row 0 is in positive Y coordinate, row 12 is in negative Y coordinate. Strategy data is imported from strategy_data.json file into self.glob.strategy_data list. usage:

Forward_Vector_Matrix(object: motion, object: local, object: glob)

dir_To_Guest(*self*)

The method is designed to define kick direction and load it into self.direction_To_Guest. Direction is measured in radians of yaw. usage:

int: row, int: col = self.dir_To_Guest() row, col - row and column of matrix attributing rectangular sector of field where ball coorinate self.glob.ball_coord fits.

turn_Face_To_Guest(*self*)

class SAMPLE_TEAM.Soccer.strategy.**Player**(*logger*, *role*, *second_pressed_button*, *glob*, *motion*, *local*)

class Player is designed for implementation of main cycle of player. Real robot have 3 programmable buttons. Combination of button pressing can transmit to programm pressed button code from 1 to 9. At initial button

pressing role of player is selected. With second pressed button optional playing mode is selected depending on role. For 'forward' and 'forward_old_style' role second_pressed_button can take value 1 or value 4. With value 1 player starts game as kick-off player, with value 4 player stars as non-kick-off player, which means player starts moving 10 seconds later. For 'run_test' role second_pressed_button can take values from 1 or value 9 with following optional modes: 1 - 10 cycle steps walk forward 2 - 20 cycle side step walk to right 3 - 20 cycle side step walk to left 4 - 20 cycle steps walk forward 5 - 20 cycle steps with rotation to right side 6 - 20 cycle steps with rotation to left side 9 - 20 cycle steps of spot walk All modes of run test are used with purpose to calibrate walking. After calibration is completed results of calibration must be input to file Sim_params.json. Motion module is used calibration data for planning motions and odometry correction into localization. usage:

Player(str: role, int: second_pressed_button, object: glob, object: motion, object: local)

play_game(*self*)

rotation_test_main_cycle(*self*, *pressed_button*)

run_test_main_cycle(*self*, *pressed_button*)

For 'run_test' role second_pressed_button can take values from 1 or value 9 with following optional modes: 1 - 10 cycle steps walk forward 2 - 20 cycle side step walk to right 3 - 20 cycle side step walk to left 4 - 20 cycle steps walk forward 5 - 20 cycle steps with rotation to right side 6 - 20 cycle steps with rotation to left side 9 - 20 cycle steps of spot walk All modes of run test are used with purpose to calibrate walking. After calibration is completed results of calibration must be input to file Sim_params.json. Motion module is used calibration data for planning motions and odometry correction into localization. usage:

self.run_test_main_cycle(int: pressed_button)

sidestep_test_main_cycle(*self*, *pressed_button*)

norm_yaw(*self*, *yaw*)

This module normalizes yaw according to internal rule: -pi <= yaw <= pi usage:

float: yaw = self.norm_yaw(float: yaw) yaw - orientation on horizontal surface in radians,

zero value orientation is directed along X axis

forward_main_cycle(*self*, *pressed_button*)

Main cycle method for 'forward' role of player. usage:

self.forward_main_cycle(int: pressed_button)

forward_old_style_main_cycle(*self*, *pressed_button*)

Main cycle method for 'forward_old_style' role of player. usage:

self.forward_main_cycle(int: pressed_button)

goalkeeper_main_cycle(*self*)

goalkeeper main cycle method is based on vector matrix strategy. Goalkeeper doesn't leave goals too far. Supposed that goalkeeper starts game at point on middle of goal line. After 10 seconds from game start goalkeeper moves to duty position which depends on detected ball position. In case if ball appears in dangetous position goalkeeper attcks ball.

goalkeeper_old_style_main_cycle(*self*)

main cycle for old style goalkeeper strategy

penalty_Shooter_main_cycle(*self*)

main cycle for penalty striker

penalty_Goalkeeper_main_cycle(*self*)

dance_main_cycle(*self*)

6.4 SAMPLE_TEAM.launcher_pb

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. The module is designed for strategy of soccer game by forward and goalkeeper.

6.4.1 Module Contents

Functions

init_gcreceiver(team, player, is_goalkeeper)	The function creates and object receiver of Game Controller messages. Game Controller messages are broadcasted to
player_super_cycle(falling, team_id, robot_color, player_number, SIMULATION, current_work_directory, robot, pause, logger)	The function is called player_super_cycle because during game player can change several roles. Each role

SAMPLE_TEAM.launcher_pb.**init_gcreceiver**(*team*, *player*, *is_goalkeeper*)

The function creates and object receiver of Game Controller messages. Game Controller messages are broadcasted to teams and to referee. Format of messages can be seen in module gamestate.py. Messages from Game Controller contains Robot info, Team info and Game state info. usage of function:

object: receiver = init_gcreceiver(int: team, int: player, bool: is_goalkeeper)

team - number of team id. For junior competitions it is recommended to use unique id for team in range 60 - 127

player - number of player displayed at his trunk is_goalkeeper - True if player is appointed to play role of goalkeeper

SAMPLE_TEAM.launcher_pb.**player_super_cycle**(*falling*, *team_id*, *robot_color*, *player_number*, *SIMULATION*, *current_work_directory*, *robot*, *pause*, *logger*)

The function is called player_super_cycle because during game player can change several roles. Each role appointed to player put it into cycle connected to playing it's role. Cycles of roles are defined in strategy.py module. player_super_cycle is cycle of cycles. For example player playing role of 'forward' can change role to 'penalty_shooter' after main times and extra times of game finished. In some situations you may decide to switch roles between forward player and goalkeeper. Usage:

player_super_cycle(object: falling, int: team_id, str: robot_color, int: player_number, int: SIMULATION, Path_object: current_work_directory, object: robot, object: pause)

falling - class object which contains int: falling.Flag which is used to deliver information about falling from low level logic to high level logic. falling.Flag can take 0 - nothing happend, 1 -falling on stomach, -1 - falling face up, 2 - falling to left, -2 - falling to right, 3 - exit from playing fase

team_id - can take value from 60 to 127 robot_color - can be 'red' or 'blue' player_number - can be from 1 to 5, with 1 to be assigned to goalkeeper SIMULATION - used for definition of simulation enviroment. value 4 is used for Webots simulation,

value 2 is used for playing in real robot

current_work_directory - is Path type object robot - object of class which is used for communication between robot model in simulation and controller

program. In case of external controller program 'ProtoBuf' communication manager is used. 'ProtoBuf' - is protocol developed by Google.

pause - object of class Pause which contains pause.Flag boolean variable. It is used to transfer pressing pause button on player's dashboard event to player's high level logic.

6.5 SAMPLE_TEAM.main_pb

The module is designed by team Robokit of Phystech Lyceum and team Starkit of MIPT under mentorship of Azer Babaev. The module is designed for creating players' dashboard and alternating between team game with Game Controller or individual play without Game Controller.

6.5.1 Module Contents

Classes

Log
Falling
Pause
RedirectText
Main_Panel

Functions

main_procedure()
main()

Attributes

LOGGING_LEVEL
SIMULATION
current_work_directory
game_data
team_1_data
team_2_data

continues on next page

Table 16 – continued from previous page

pause

SAMPLE_TEAM.main_pb.**LOGGING_LEVEL**

SAMPLE_TEAM.main_pb.**SIMULATION = 4**

SAMPLE_TEAM.main_pb.**current_work_directory**

SAMPLE_TEAM.main_pb.**game_data**

SAMPLE_TEAM.main_pb.**team_1_data**

SAMPLE_TEAM.main_pb.**team_2_data**

class SAMPLE_TEAM.main_pb.**Log**(*filename*)

get_file_handler(*self*)

get_stream_handler(*self*)

get_logger(*self*, *name*)

class SAMPLE_TEAM.main_pb.**Falling**

class SAMPLE_TEAM.main_pb.**Pause**

SAMPLE_TEAM.main_pb.**pause**

SAMPLE_TEAM.main_pb.**main_procedure**()

class SAMPLE_TEAM.main_pb.**RedirectText**(*aWxTextCtrl*)
Bases: object

write(*self*, *string*)

class SAMPLE_TEAM.main_pb.**Main_Panel**(**args*, ***kwargs*)
Bases: wx.Frame

main_procedure(*self*)

InitUI(*self*)

ShowMessage1(*self*, *event*)

ShowMessage2(*self*, *event*)

SAMPLE_TEAM.main_pb.**main**()

6.6 communication_manager_robokit

Class that provides communication with simulator Webots.

6.6.1 Module Contents

Classes

CommunicationManager

class communication_manager_robokit.**CommunicationManager**(*maxsize=1*, *host='127.0.0.1'*, *port=10001*, *logger=logging*, *team_color='RED'*, *player_number=1*, *time_step=15*)

enable_sensors(*self*, *sensors*) → None

__get_sensor(*self*, *name*) → dict

__send_message(*self*)

__update_history(*self*, *message*)

__procces_object(*self*, *name*)

time_sleep(*self*, *t*) → None
Emulate sleep according to simulation time.

Parameters **t** (*float*) – time

get_imu_body(*self*) → dict
Provide last measurement from imu located in body. Can be empty if 'imu body' sensor is not enabled or webots does not sent us any measurement. Also contains simulation time of measurement.

Returns {"position": [roll, pitch, yaw]}

Return type dict

get_imu_head(*self*) → dict
Provide last measurement from imu located in head. Can be empty if 'imu_head' sensor is not enabled or webots does not send us any measurement. Also contains simulation time of measurement.

Returns {"position": [roll, pitch, yaw], "time": time}

Return type dict

get_localization(*self*) → dict
Provide blurred position of the robot on the field and confidence in this position ('consistency' - where 1 fully confident and 0 - have no confidence). Can be empty if 'gps_body' sensor is not enabled or webots does not send us any measurement. Also contains simulation time of measurement.

Returns {"position": [x, y, consistency], "time": time}

Return type dict

get_ball(*self*) → dict
Provide blurred position of the ball relative to the robot. Can be empty if: 1. 'recognition', 'gps_body' or 'imu_body' sensors are not enabled 2. webots did not send us any measurement. 3. robot does not stand upright position 4. ball is not in the camera field of view (fov)

Also contains simulation time of measurement.

Returns {"position": [x, y], "time": time}

Return type dict

get_opponents(*self*) → list
Provide blurred positions of the opponents relative to the robot. Can be empty if:

1. 'recognition', 'gps_body' or 'imu_body' sensors are not enabled
2. webots did not send us any measurement.
3. robot does not stand upright position
4. opponent is not in the camera field of view (fov)

Also contains simulation time of measurement.

Returns [{"position": [x1, y1], "time": time}, {"position": [x2, y2], "time": time}]

Return type list

get_mates(*self*) → dict
Provide blurred position of the mate relative to the robot. Can be empty if:

1. 'recognition', 'gps_body' or 'imu_body' sensors are not enabled
2. webots did not send us any measurement.
3. robot does not stand upright position
4. mate is not in the camera field of view (fov)

Also contains simulation time of measurement.

Returns {"position": [x, y], "time": time}

Return type list

get_time(*self*) → float
Provide latest observed simulation time.

Returns simulation time

Return type float

send_servos(*self*, *data*) → None
Add to message queue dict with listed servo names and angles in radians. List of posible servos: ["right_ankle_roll", "right_ankle_pitch", "right_knee", "right_hip_pitch", "right_hip_roll", "right_hip_yaw", "right_elbow_pitch", "right_shoulder_twirl", "right_shoulder_roll", "right_shoulder_pitch", "pelvis_yaw", "left_ankle_roll", "left_ankle_pitch", "left_knee", "left_hip_pitch", "left_hip_roll", "left_hip_yaw", "left_elbow_pitch", "left_shoulder_twirl", "left_shoulder_roll", "left_shoulder_pitch", "head_yaw", "head_pitch"]

Parameters **data** (*dict*) – {servo_name: servo_angle, ... }

run(*self*)
Infinity cycle of sending and receiving messages. Should be launched in sepparet thread. Communication manager launch this func itself in constructor

6.7 blurrer

6.7.1 Module Contents

Classes

Blurrer	Simulate localization and vision noize.

class blurrer.**Blurrer**(*object_angle_noize=0.0*, *object_distance_noize=0.0*, *observation_bonus=0.0*, *step_cost=0.0*, *constant_loc_noize=0.0*, *loc_noize_meters=0.0*)

Simulate localization and vision noize. Params is placed in the blurrer.json file. :param object_angle_noize: Noize for angle in radians.

Blurrer will uniformly random value from -object_angle_noize to object_angle_noize and add it to the ground truth course. Defaults to 0.

Parameters

- **object_distance_noize** (*float, optional*) – Noize for distance in percents divided by 100. Blurrer will uniformly random value from -object_distance_noize to object_distance_noize and multiply difference of 1 and this value with ground truth distance. Defaults to 0..
- **observation_bonus** (*float, optional*) – Blurrer will increase the consistency for every good observation (successfuly processed image). Defaults to 0..
- **step_cost** (*float, optional*) – Blurrer will decrease the consistency for every simulation step. Defaults to 0..
- **constant_loc_noize** (*float, optional*) – Constant localization noize. Defaults to 0..
- **loc_noize_meters** (*float, optional*) – Multiplier for consistency, in meters. Defaults to 0. Defaults to 0..

load_json(*self*, *filename*)

course(*self*, *angle*)

distance(*self*, *distance*)

objects(*self*, *course=course*, *distance=distance*)

loc(*self*, *x*, *y*)

coord(*self*, *p*)

step(*self*)

observation(*self*)

update_consistency(*self*, *value*)

6.8 message_manager

lass operates with protobuff messages. used to create and parse messages.

6.8.1 Module Contents

Classes

MessageManager

class message_manager.**MessageManager**(*logger=logging*, *head_buffer_size=4*)

get_size(*self*)

Returns Value of heder byte buffer.

Return type int

static create_requests_message()
Create Empty protobuf class instance for request message.

Returns Empty protobuf class instance.

Return type messages_pb2

static create_answer_message()
Create Empty protobuf class instance for answer message.

Returns Empty protobuf class instance.

Return type messages_pb2

static build_request_from_file(*path*)
Parsing data from message file to protobuf message instance.

Parameters **path** (*string*) – path to filename.txt with message.

Returns protobuf class instance of filled message from file.

Return type messages_pb2

build_request_positions(*self*, *positions*)
reating an instance of the protobuff class and fills it with the values of the actuators

Parameters **positions** (*dict*) – key - servo name and values - position.

Returns protobuf class instance of filled message with servos.

Return type messages_pb2

static generate_message(*message*)
Generate bytes string for sending message.

Parameters

- **message** (*messages_pb2*) – protobuf class instance of filled
- **message.** –

Returns bytes string of message.

Return type bytes

`message_from_file`(*self*, *path*)
Function process the protobuff message. Measurement values of sensors, messages from player.exe and webots. Received messages are placed in the dictionary :param path: path to filename.txt with default message. :type path: [string]

Returns protobuf class instance, with values from file.

Return type messages_pb2

`get_answer_size`(*self*, *content_size*)
alculating message size from header bytes

Parameters `content_size` (*bytes*) – Byte size of answer message.

Returns Size of answer message.

Return type int

`add_initial_request`(*self*, *sensor_name*, *sensor_time*)
Generate bytes string for sending message.

Parameters

- `sensor_name` (*string*) – protobuf class instance of filled
- `message.` –

Returns bytes string of message.

Return type bytes

`build_initial_request`(*self*)
Generate bytes string for initialization message.

Returns bytes string of message.

Return type bytes

`parse_answer_message`(*self*, *data*)
Parsing answer message from byte array to dict with measurements

Parameters `data` (*[type]*) – [description]

Returns [description]

Return type [type]

`static parse_message`(*message*) → dict
Function process the protobuff message. Measurement values of sensors, messages from player.exe and webots. Received messages are placed in the dictionary :param message: protobuf class instance :type message: messages_pb2 :param of new message with filled or unfilled.:

Returns dict with keys of names sensors

Return type dict

PYTHON MODULE INDEX

b

c

m

s

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

W