Environment "Robot Soccer " Documentation

Author: Shuo Jiang (jiang.shuo@husky.neu.edu)

Introduction

Inspired by the world famous robot soccer game "RoboCup", I tried to code it as a simulator that can be used for a learning system. The code is implemented in python and provided with simple interfaces. The environment is decoupled with learning method so reader can try any algorithms on.

Scenario Description

Here we consider the game as 3 vs 3 robot team match, marked as red team and blue team. Red team is attacking and blue team is defending. The scenario looks like

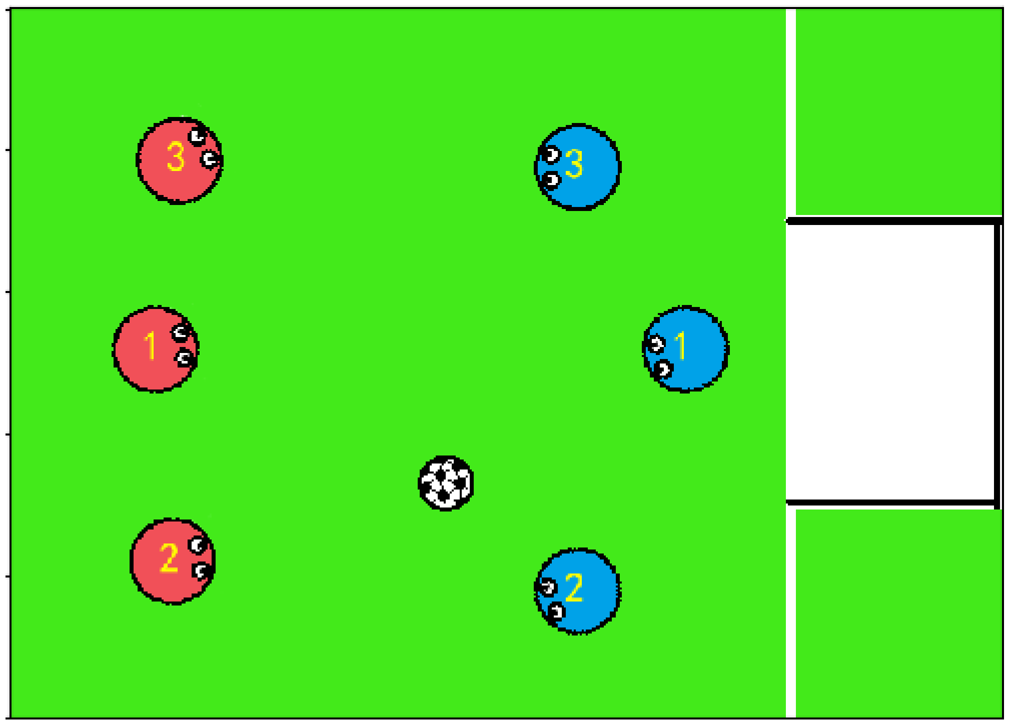


Figure. Scenario

As all readers should be familiar with soccer game. The goal of red team is to shoot the ball into the gate. The Blue team is trying to avoid it to happen. Each time the ball is in the gate. The red team will get 10 points reward and blue team will get -10 points reward.

There are some physical properties that might be different from real soccer game, which is made for simplification purpose.

1. There is no kick-out, ball goes to the boarder will be bounced off like bounced into a rigid body.

2. When a player is on the way of the ball, there is 60% chance that the ball is bounced back by the player, and 40% chance the ball will keep its way. (which means the ball passes in the middle of the two legs)

3. When the players moves, and the ball is on its way, the player will kick the ball away.

4. The motion dynamic of players and ball only keep the first order equation. Which means we only consider the position and velocity of ball and players

Coordinate system

The coordinate system is defined as

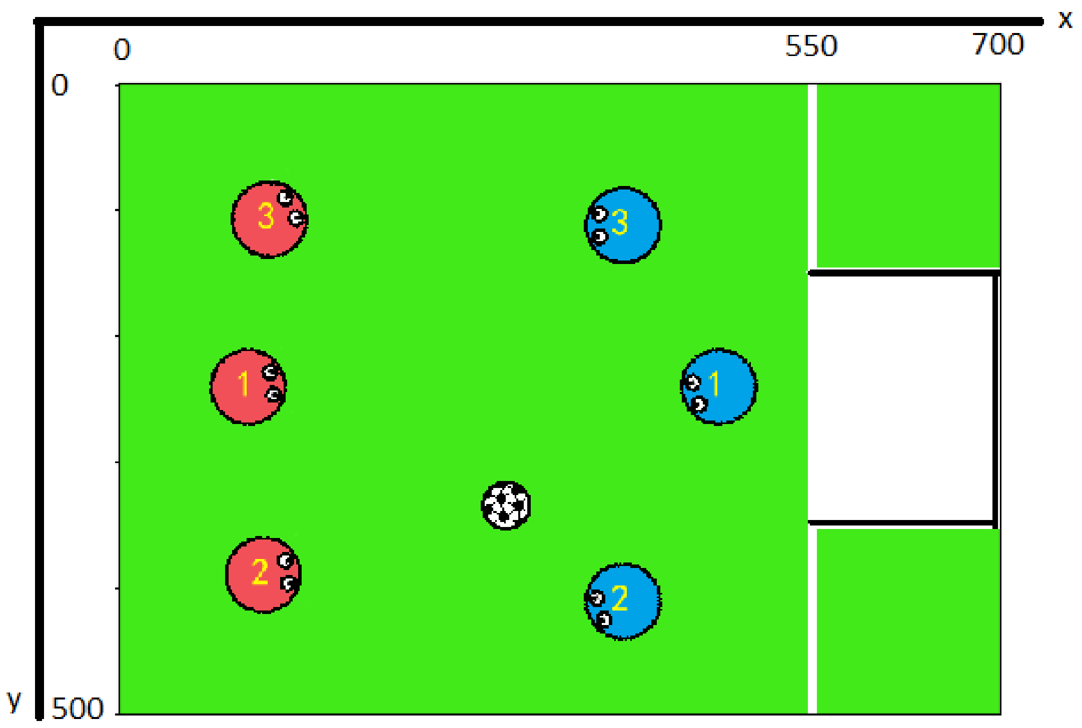


Figure. Coordinate system

So if you want to relocate the players manually, please keep the players in x [0, 550] and y [0, 500] range, otherwise you will not see the players.

Initialization

Initializing the game will relocate the ball and players at positions as Figure shown before. The ball is at center, and players surrounding it. You can choose how many players to be on the field.

In the initialization function, there is an `add_player()` function, which was called 6 times as

```
self.add_player(0)
self.add_player(0)
self.add_player(0)
self.add_player(1)
self.add_player(1)
self.add_player(1)
```

input parameter is a variable, when it equals 0, a red player will be added, when it equals 1, a blue player will be added.

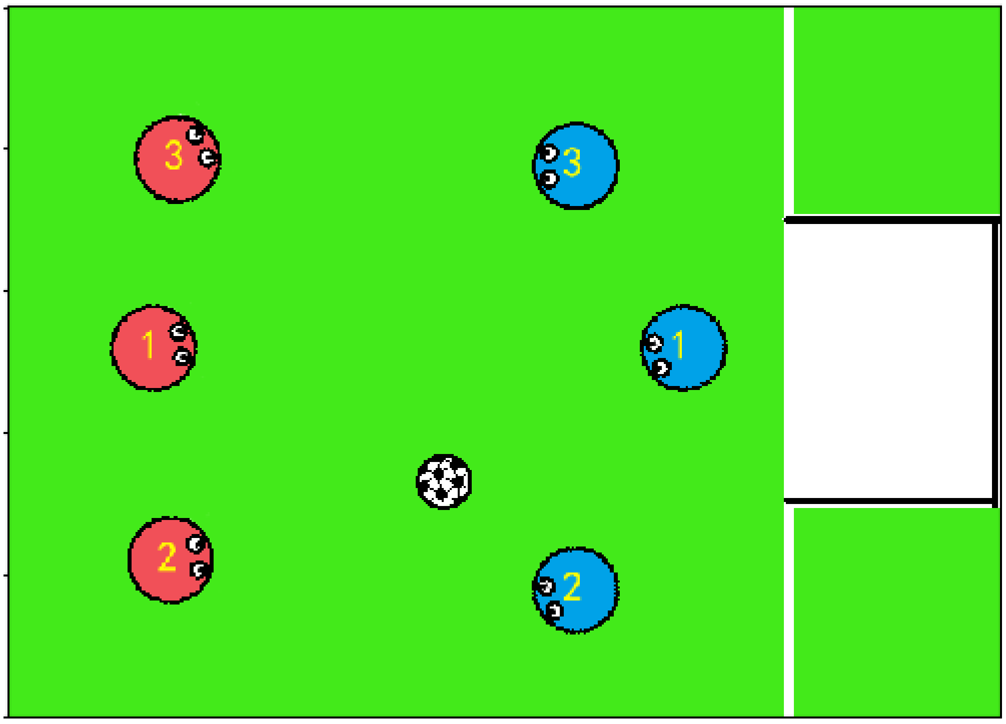


Figure. Initialization positions

Noticing that in the image above, there is a number from 1-3 on the head of each player. There will be at maximum 3 players for each team. When the first red player is added in the game, it will be located at the position numbered 1 of player, as shown in the figure. Similarly the second blue player will be located at position blue with number 2. Here is an example with 2 red players and 1 blue player initialized.

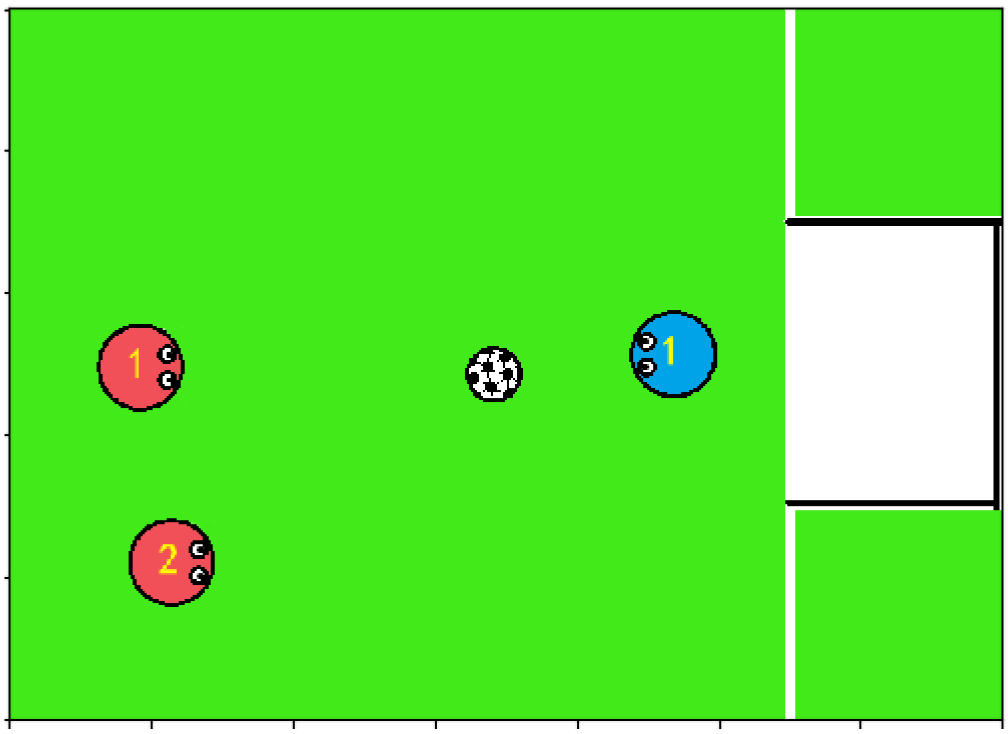


Figure. Initialization with different number

Players

The player's state is defined as the aggregation of Cartesian position in 2D, velocity, orientation and angular velocity. The position and velocity are defined in the coordinate system illustrated as before.

Orientation is defined as the angle between orientation vector and vector [1, 0], from -180 degree to 180 degree.

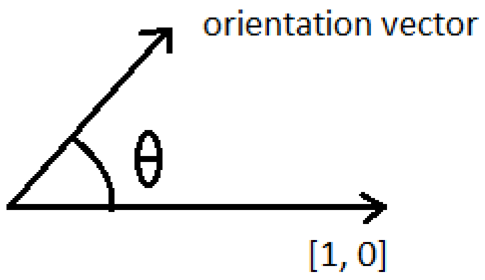


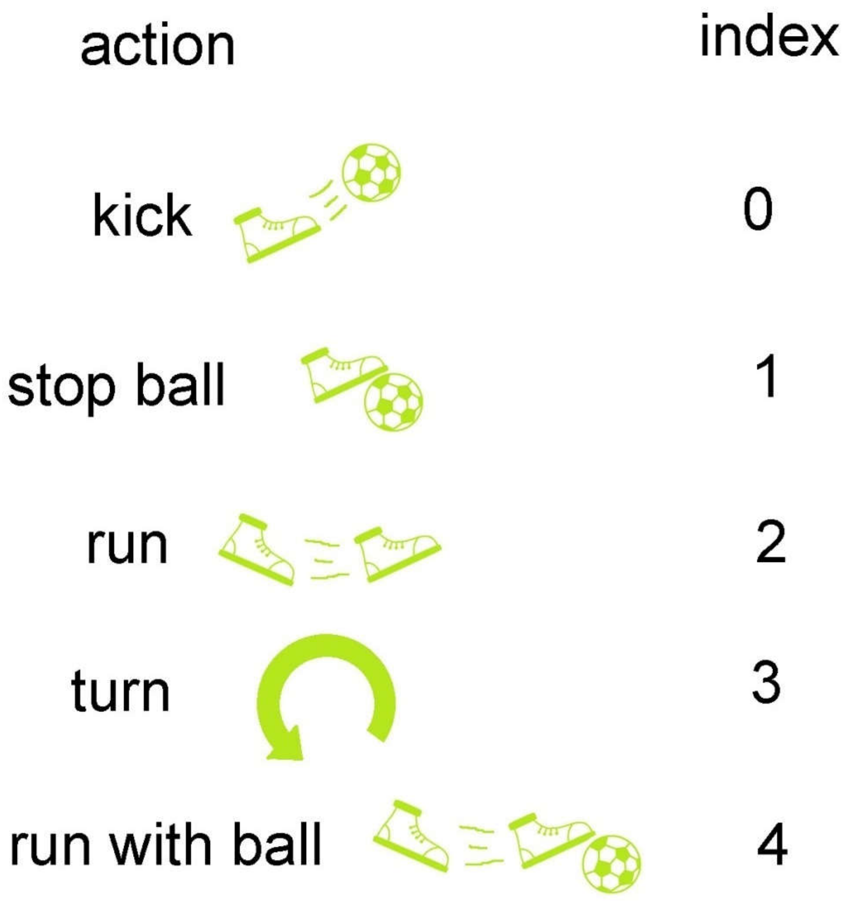
Figure. Orientation of player

The angular velocity is defined in degree. And anti-clockwise rotation stands for positive angular velocity and clockwise as negative.

Each player has two properties, "radius" and "kick radius", which are set 25 and 50 respectively. Considering player as a rigid body, so it has some volume that cannot be penetrated by other rigid bodies. The radius parameter is used to denote the radius of player's body. Kick radius is the range that player can interact with the ball. If the distance to the ball is larger than this radius, some actions may not function, as stop the ball or something.

Action Space

There are 5 actions for every player, "kick", "stop ball", "run", "turn" and "run with ball". The actions will be indexed from 0-4, as they are specified in function.



action	index
kick	0
stop ball	1
run	2
turn	3
run with ball	4

Figure. Action space

For each action, there will be two parameters with it, one is a vector of dimension of 2, as [1, 0], another is scalar. For some of the actions, the parameters determine the direction and strength of the action.

0. kick
for "kick" action, the vector determines the direction of kicking, and the scalar determines the

magnitude of the kicking vector. The vector will then be added to the velocity of the ball. The player's will be penalized 0.8 times for the kicking action (as can imagine that you cannot kick the ball while keeping running at a high speed). There will be another attenuation of velocity of ball. If you are kicking a ball straight to direction you are facing, you can use all your strength that the ball will inherit the full velocity given by the kicking. However if you are kicking the ball to the direction backward, it is hard for you to use all your strength, so in this case, only maybe 30% strength of kicking will be casted on ball. So there will be a linear attenuation determined by the kicking vector and the vector of the player's orientation. So the attenuation relation is shown in figure below.

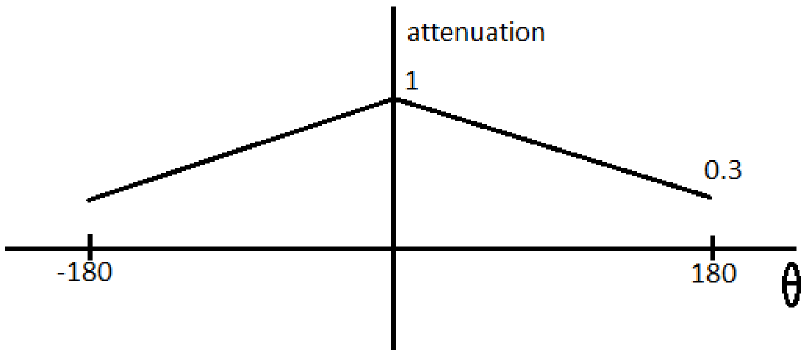


Figure. Attenuation

1. stop the ball
If the ball is in the kick radius of player, this action will cause both the velocity of player and velocity of the ball to be [0, 0]. This action will not use the vector and scalar parameter even though they are fed.

2. run
This action cause player to run to a direction by given vector, and scaled by the scalar. If the given speed is larger than the speed limit (defined in the player class), the speed will be regulated to maximum speed. Running is also attenuated by the function defined in action "kick", which means running backward will be much slower.

3. turn
This action cause the player to turn its body. This action will only use the scalar. Positive value means player turns its body anti-clockwise. The value will be regulated in [-20, 20].

4. run with ball
This action cause player to run to a direction by given vector, and scaled by the scalar. If the given speed is larger than the speed limit (defined in the player class), the speed will be regulated to maximum speed. Running is also attenuated by the function defined in action "kick", which means running backward will be much slower. If the ball is in the kick radius of player, the ball will get the same velocity as the player.

Observation

What can players observe is crucial in implementing learning algorithms. In this simulator, you can use full observation or partial observation.

function `get_global_obs()` will return a numpy array in RGB image form of size (500,700,3). It is a fully observable setting.

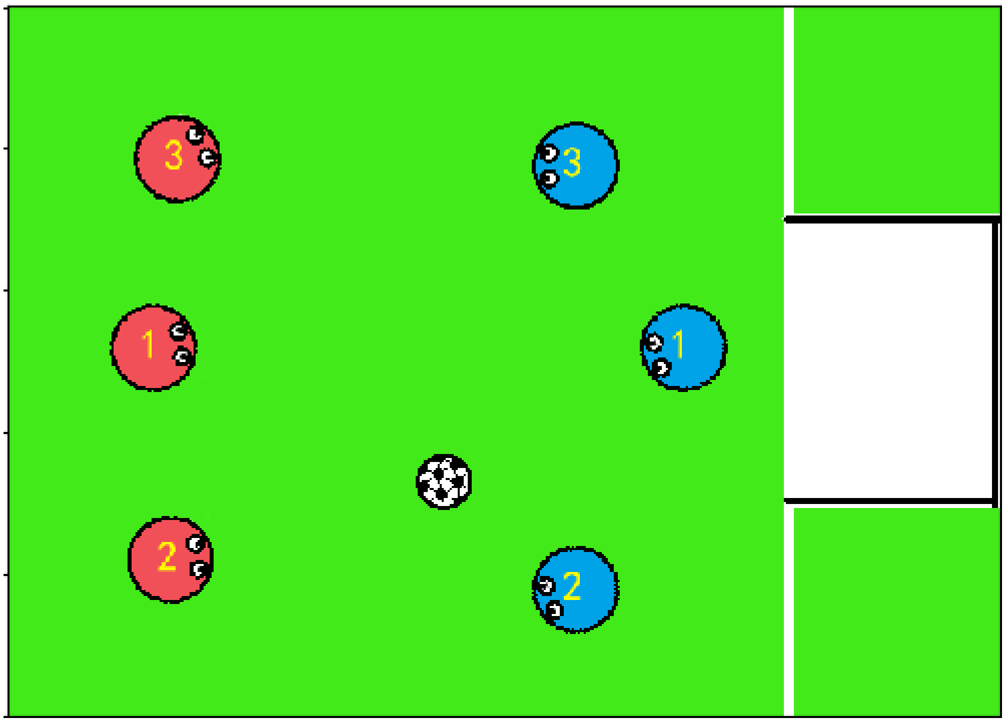


Figure. Full Observation

Also the simulator has a partial observation return. Function `get_agt_obs(index)` will return a view of a player by giving the index of a player. The returned observation is a numpy array in RGB image form of size (500,700,3). However the player's observation only contains useful contents in a narrow angle on the direction of looking, other regions are covered by fogs, as:

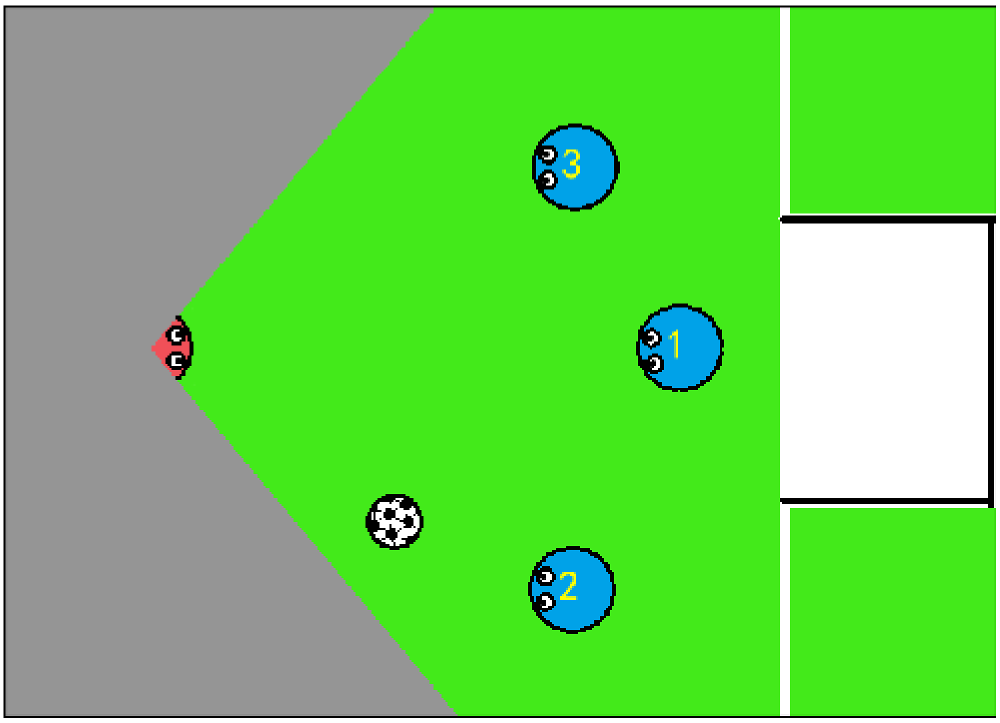


Figure. Partial Observation

The angle of view is defined in class player, the default angle in the image is 100 degrees.

Class Organization

In the simulator, 3 classes are defined, "Ball", "Player" and " EnvSoccer".

There are bunch of functions can be called, but only some of them can be useful to users. I will pick some for introduction.

Class Ball

Member variables

```
self.pos = [277., 244.]         # the position of ball
self.last_pos                   # position of last time step
self.vel = [0., 0.]             # velocity of ball
self.friction  = 0.99           # friction from 0-1 causes ball's velocity
                                to attenuate. 1 means no attenuation
self.radius = 20.               # The radius of the ball, may be considered
                                for collision.
```

Member functions

users do not have to call any member functions in this class.

Class Player

Member variables

```
self.pos = [103., 241.]          # position of player
self.vel = [0., 0.]             # velocity of player
self.theta = 0                  # rotation of player, based on [1, 0],
                                bounded in -180-180
self.omega = 0.                 # angular velocity of player in degree,
                                positive stands for anti-clockwise
self.radius = 25.               # the radius of the player, may be
                                considered for collision.
self.kick_radius = 50.          # the radius that player can interact with
                                the ball
self.view_angle = 50.           # the angle of view for partial observation,
                                the angle of view is two times of this value.
                                It means the real angle is 50*2 =100 degrees
self.max_speed = 30             # max running speed of player
self.team = team                # team of player, 0 stands for red team,
                                1 stands for blue team
self.role = role                # the position of each player in team, from
                                1-3, 1 stands for player in mid lane, 2 is
                                bottom lane, 3 is top lane
```

Member functions

users do not have to call any member functions in this class.

Class EnvSoccer

Member variables

```
self.map_size = [500, 700]
self.ball = Ball(0.99)          # initialize ball object, with friction
                                coefficient. (velocity attenuation)
self.red_score = 0              # score of red team
self.blue_score = 0             # score of blue team
self.player_list = []           # list of players on the ground.
```

Member functions

`get_global_obs()`

# return an image of full observation of size (500,700,1)

`get_agt_obs(index)`

# return the local observation of the index-th player in the self.player_list. Image of size (500,700,1).

`step(action_index_list, action_vector_list, action_const_list)`

# to be called at each time step. Update the system. The first parameter "action_index_list" is a list of action index integer 0-4. The second parameter is a list of 2D vector, and the third is a list of float scalar. The vector and scalar is as the auxiliary parameter of action, as introduced in action space.

`reset_game()`

# relocate the ball and players to the initial position, reset scores of both teams to 0.

`relocate()`

# relocate the ball and players to the initial position, but keeps the scores

`count_red_player_num()`

# return the current number of players in red team.

`count_blue_player_num()`

# return the current number of players in blue team.

`add_player(team)`

# add a new player to the player_list. "team" is a parameter that if it is 0, add a new red team player, if it is 1, add a new blue team player. If the number of players is at maximum (3), this function will do nothing. The "role" of the added player will be automatically assigned. From 1 to 3.

Example

Here is an example using test function, and it is in "test_Soccer.py". The action is random chosen for each agent, click and run.

```
from env_Soccer import EnvSoccer
import matplotlib.pyplot as plt
```

```python
import random

env = EnvSoccer()
fig = plt.figure()
max_MC_iter = 2000
env.ball.vel[0] = env.ball.vel[0] + 40 * random.random()
env.ball.vel[1] = env.ball.vel[1] + 60 * random.random()

for MC_iter in range(max_MC_iter):
    print(MC_iter)
    plt.imshow(env.get_global_obs())

    rand_action_list = []
    rand_vec_list = []
    rand_const_list = []
    for i in range(6):
        temp = [20 * (random.random()-0.5), 20 * (random.random()-0.5)]
        rand_action_list.append(random.randint(0, 4))
        rand_vec_list.append(temp)
        rand_const_list.append(20 * random.random())
    # add random velocity
    env.step(rand_action_list, rand_vec_list, rand_const_list)
    plt.pause(.04)
    plt.draw()
```