FC Portugal Codebase

1. Setup

1.1. Rcssserver3d

https://software.opensuse.org//download.html?project=science:SimSpark&package=rcssserver3d
or
https://gitlab.com/robocup-sim/SimSpark/-/wikis/home

1.2. RoboViz (visualizer tool)

https://github.com/magmaOffenburg/RoboViz/releases

1.3. Agent

Install dependencies and clone project

```
sudo apt install libgsl-dev
pip3 install numpy pybind11 psutil

cd ~ # or choose another directory for the codebase
git clone https://github.com/m-abr/FCPCodebase
```

Test if it starts (it may take some time on the first run to compile C++ modules)

```
(cd FCPCodebase; python3 Run_Utils.py)
```

Optional step to prevent numpy from using multiple threads inefficiently (recommended)

```
export OMP_NUM_THREADS=1
echo "export OMP_NUM_THREADS=1">>~/.bashrc
```

Optional dependencies to interact with integrated reinforcement learning gyms:

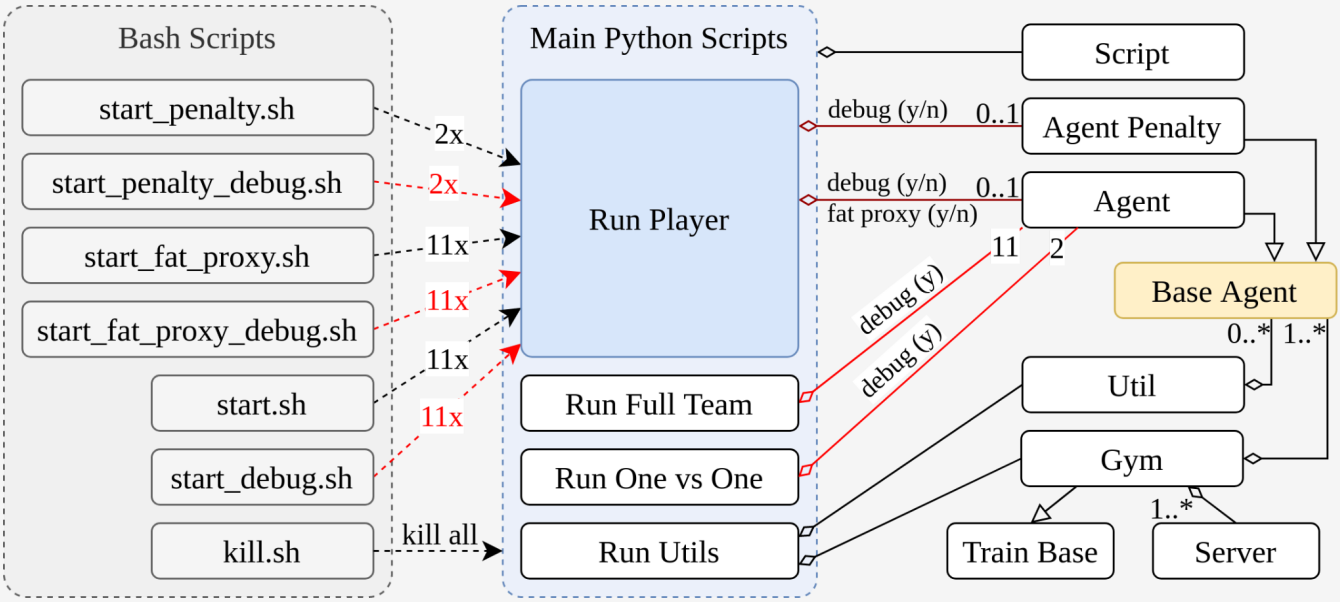
```
# Install Stable Baselines3 (SB3) (easy)
pip3 install stable-baselines3 gym shimmy

# Starting from stable-baselines3 version 2.0.0, users will receive a warning
recommending transitioning to Gymnasium environments, since we use an OpenAI Gym
environment. This should be ok. However, if you encounter any compatibility
issues, consider installing an earlier version >= 1.6.
```

This step is optional and **not recommended** for most users (for more details, refer to the GitHub issue https://github.com/m-abr/FCPCodebase/issues/3). An alternative to the previous step is to install a modified version of stable-baselines3 with symmetry extensions. Using this version only makes sense if the user is familiarized with symmetry mappings and defines them for the NAO robot.

```
# it is based on stable-baselines3 (SB3) v1.6
# it should be cloned to the parent directory of FC Portugal's codebase
# e.g. "/home/user/FCPCodebase" and "/home/user/stable-baselines3"
git clone https://github.com/m-abr/Adaptive-Symmetry-Learning stable-baselines3
```

2. Structure

2.1. Codebase Structure



Above is an overview of the codebase structure. The main components are described as follows:

Bash Scripts: these scripts are shortcuts to launch multiple agents in independent threads

- **start.sh** - launches 11 independent instances of **Run Player**, each initiating 1 **Agent**
- **start_debug.sh** - same as previous but each **Run Player** initiates 1 **Agent** in *debug mode*
- **start_fat_proxy.sh** - launches 11x **Run Player**, each initiating 1 **Agent** in *fat proxy mode*
- **start_fat_proxy_debug.sh** - same but each **Run Player** initiates also in *debug mode*
- **start_penalty.sh** - launches 11x **Run Player**, each initiating 1 **Agent Penalty**
- **start_penalty_debug.sh** - same but each **Run Player** initiates 1 **Agent Penalty** in *debug mode*
- **kill.sh** - kills any instance of the Main Python Scripts

Main Python Scripts: these are the main entry point files, Python execution starts here

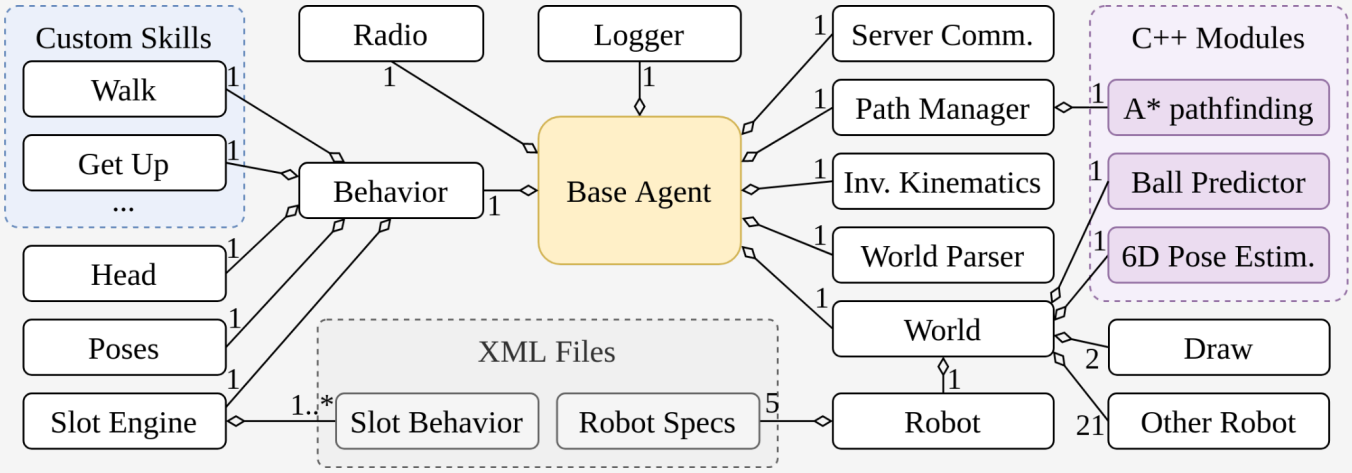
- **Run_Utils.py** - Interactive menu to run:
  - **Util** - a set of interactive demonstrations, tests and utilities to execute key features of the team/agents, as well as read and change server parameters easily;
  - **Gym** - OpenAI custom gym implementations to train new skills through reinforcement learning. Gyms also support interactive model loading, testing, and retraining.
- **Run_Player.py** - creates 1 **Agent** or **Agent Penalty** in *normal mode* or *debug mode*, based on input arguments, and then runs the selected agent in an infinite loop.

- **Run_Full_Team.py** - creates 11 agents in the same thread, and runs each of them sequentially, in an infinite loop. Since they all run on 1 thread, it can be slow, but useful for debugging.
- **Run_One_vs_One.py** - creates 2 agents, 1 per team, and runs both in an infinite loop.

Important Classes:

- **Script** - used by Main Python Scripts. Handles command line arguments, implements batch operations to manage multiple agents on the same thread, compiles C++ modules if the binaries are not found or are older than the source files;
- **Base_Agent** - implements an agent that automatically processes information from the server to update its internal model of the world. It has integrated tools that automate many procedures, creating a high-level interface to interact with the robot and environment. The *debug mode* displays extra information during initialization, enables logs, drawings, and special commands sent via the server's monitor port (e.g. beam player to 3D position, kill server, move ball, set game time, set play mode, kill any agent)
- **Agent** (Agent.py) - implements the main agent strategy, used in actual soccer matches. The strategy is currently reused for the Fat Proxy Challenge created by magmaOffenburg, by translating local commands into fat proxy commands. However, if the user wants distinct agents for the main competition and the challenge, we suggest creating a copy of **Agent.py** named **Agent_Fat_Proxy.py** and modifying both files to remove unnecessary components. Then, adjust **Run_Player.py** to invoke the appropriate agent class based on the "**a.F**" argument, in the same way that "**a.P**" is used to invoke **Agent Penalty**.
- **Agent** (Agent_Penalty.py) - implements the main strategy employed during a penalty shootout. It controls the goalkeeper and one kicker.
- **Server** - creates/deletes instances of servers, and checks for TCP port collisions
- **Train_Base** - parent class of all gyms. It mainly implements 4 functionalities:
    - **User interface** - implements an auxiliary method to guide the user in searching for trained models within the project. Another feature allows the user to interactively control the frame rate during model testing through simple input commands;
    - **Train model** - trains a model using Stable Baselines3 (SB3), while automating the creation of callbacks to evaluate and save models (at user defined intervals or just the best model). It also creates backups of the gym environment for future reference. Additionally, it displays an evaluation graph in the command line at user-defined intervals, facilitating a quick assessment of the learning progress;
    - **Test model** - tests a model, prints statistics and saves them to a log file;
    - **Export model** - exports a model into a binary file that can be integrated in the team as a new skill.

2.2. Base Agent Structure



The **Base Agent** has 1 instance of each of the following classes:

- **Radio** - automated internal communication with teammates to share the position and state of every visible agent and the ball (for a detailed algorithm explanation see 2022 Champions paper);
- **Logger** - a logger object is linked to a specific topic, taking on the responsibility of timestamping and saving messages in log files (errors, warnings or any kind of useful information);
- **Server_Comm** - handles all communication with the server through the official *agent port* (initialization, speed of robot joints, messages to teammates, pass commands and beams) and the unofficial *monitor port* (beam player to 3D position, kill server, move ball, set game time, set play mode, kill any agent);
- **Path_Manager** - gathers obstacles based on game conditions and determines the best path to a user-defined target (using A*). Among various path methods, the `get_path_to_ball` is the most comprehensive, avoiding obstacles and intersecting a moving ball, while positioning and rotating the robot to perform precise actions such as kicking or starting to dribble.
    - **a_star.so** - custom A* pathfinding implementation library in C++, optimized for the soccer environment. Defines 2 types of obstacles:
        - static obstacles like goalposts and outer field lines (which can be crossed by the player when walking normally but not when dribbling);
        - dynamic obstacles like players, the ball, and exclusion areas defined in certain play modes. Every obstacle can be associated with a hard exclusion radius, defining strictly forbidden areas, and/or a soft exclusion radius, specifying areas to avoid. The intensity of avoidance in the soft exclusion zone is configurable.
- **Inverse_Kinematics** - computes inverse kinematics for each leg, given the relative position of each ankle and the 3D orientation of each foot. Also provides auxiliary methods for coordinate transformations and trajectory computation;
- **World_Parser** - custom S-expression parser for server messages, standardizing frames of reference by remapping certain axes. Additionally, some joints are inverted, such that actions

performed with the left limbs mirror those executed by the right limbs (allowing for easier specification of symmetric behaviors);

- **World** - internal world model, updated through vision and communication with teammates. It monitors various time metrics such as game time, server time, and client time, along with capturing the game state and dynamic objects including self, teammates, opponents, and the ball. Has methods to calculate the relative and absolute ball velocity, given a time window, and manipulate position predictions. Its main classes are:
    - **ball_predictor.so** - custom C++ library to predict the position and velocity of a rolling ball (on the ground). It also predicts when a robot, with a specific walking speed, will intersect the rolling ball.
    - **localization.so** - 6D pose estimation algorithm based on a custom probabilistic model (for a detailed algorithm explanation see 6D Localization paper). Video demo.
    - **Draw** - class to draw on RoboViz. It draws arrows in addition to all the native shapes and text annotations. This class is accessible through the world object but it can also be created independently from an agent, as seen in the *Draw util*.
    - **Other_Robot** - information about the state of teammates or opponents
    - **Robot** - internal robot state, including information about localization, orientation, joints, joint targets, body parts, sensors. Provides methods to handle joint control, IMU computation, forward kinematics, pose transformations (3D translation and orientation);
        - Robot Specs - XML files with the specification of 5 types of robot models;
- **Behavior** - manages low-level behaviors, automatically resetting them when it detects a transition
    - **Head** - algorithm that controls the robot's head orientation
    - **Poses** - a pose is defined by a single keyframe (1 angular position per joint)
    - **Slot_Engine** - manages the execution of slot behaviors
        - Slot Behavior - XML files that define behaviors composed of multiple keyframes
    - **Custom skills** (e.g. Walk) - skills that require a custom preparation and/or execution

3. Environment Basics

3.1. Robot types

Type 0: the standard NAO robot (22 joints)
Type 1: longer legs and arms (22 joints)
Type 2: faster ankle-pitch and slower ankle-roll speed (22 joints)
Type 3: longest legs and arms and a wider hip (22 joints)
Type 4: the standard NAO with toes: 1 additional joint in each foot (24 joints)

3.2. Joints

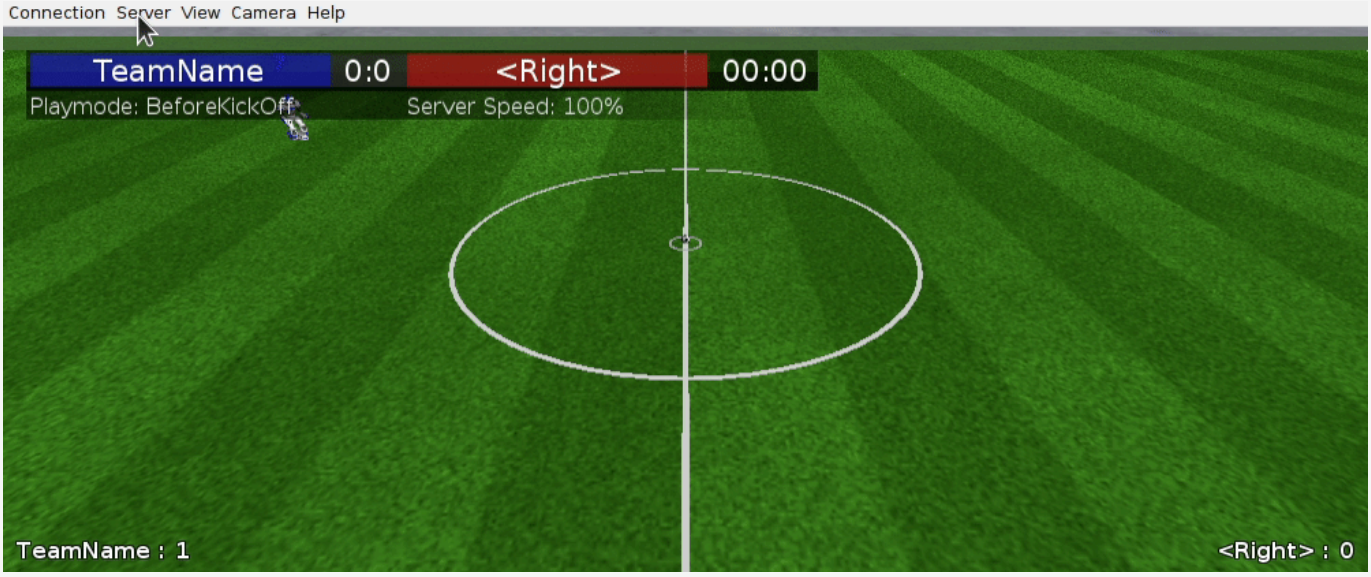
Joint Index	Description	Min Direction	Min	Max	Max Direction
0	Head yaw	right	-120	+120	left
1	Head pitch	down	-45	+45	up
2/3	Leg yaw/pitch	forward / out	-90	+1	backward / in
4/5	Leg roll	in	-25	+45	out
6/7	Leg pitch	backward	-25	+100	forward
8/9	Knee	flexion	-130	+1	extension
10/11	Foot pitch	down	-45	+75	up
12/13	Foot roll	in	-45	+25	out
14/15	Arm pitch	down	-120	+120	up
16/17	Arm roll	in	-1	+95	out
18/19	Elbow yaw	in	-120	+120	out
20/21	Elbow roll	extension	-1	+90	flexion
22/23	Toe pitch (type 4 only)	down	-1	+70	up

3.2.1. Remarks

- The index order is always left/right, except for the head;
- The **World_Parser** inverts the sign of joints **5**, **13**, **17**, **18**, **20** (identified by the server as **rlj2**, **rlj6**, **raj2**, **laj3**, **laj4**), so that all joint pairs behave symmetrically
    - This inversion is performed on the received joint perceptor values, as well as on the sent joint actuator values
- The last 2 joints (toes) exist only in NAO robot type 4
- The names assigned by the server to each joint perceptor, corresponding to the order in the table above, are **hj1**, **hj2**, **llj1**, **rlj1**, **llj2**, **rlj2**, **llj3**, **rlj3**, **llj4**, **rlj4**, **llj5**, **rlj5**, **llj6**, **rlj6**, **laj1**, **raj1**, **laj2**, **raj2**, **laj3**, **raj3**, **laj4**, **raj4**, **llj7**, **rlj7**

4. Getting Started

4.1. Minimal Example



Create a file example.py in the root folder, with the following content:

```
from agent.Base_Agent import Base_Agent as Agent

# Args: Server IP, Agent Port, Monitor Port, Uniform No., Robot Type, Team Name
player = Agent("localhost", 3100, 3200, 7, 1, "TeamName")

w = player.world

while True:
    player.behavior.execute("Walk", w.ball_abs_pos[:2], True, None, True, None)
    player.scom.commit_and_send( w.robot.get_command() )
    player.scom.receive()
```

After creating this file, run it:

```
python3 example.py
```

This minimal example will create a single player (by creating an object from the class `Base_Agent`).

- The robot will constantly try to follow the ball until it falls or the simulation is closed.
- The robot slows down near the ball since it is approaching the target.
- In RoboViz, change the play mode to **PlayOn** to allow the ball to be moved.
- In RoboViz, try to manually relocate the ball with your cursor (select the ball with the cursor, move the cursor to the desired position, press CTRL + LEFT MOUSE BUTTON to move the ball).

4.2. Improve the Minimal Example

4.2.1. How to beam the player to another location?

The `unofficial_beam` method uses the server's monitor port to instantly relocate the player under any game condition (in practice it takes a few time steps for the server to beam the player). Its inputs are the desired **3D position** of the robot (use `w.robot.beam_height` so that the robot is placed on its feet) and the **rotation** (set it to `0` degrees to rotate the robot towards the opponent's side). This command can be used at any point in the code, not just in the beginning. Try other locations and rotations [-180,180]:

```
w = player.world
player.scom.unofficial_beam((-3,0,w.robot.beam_height), 0)

while True:
    …
```

The position and velocity of the ball can also be changed using `unofficial_move_ball`. For more commands like these see /communication/Server_Comm.py. During official matches, commands that start with "unofficial" cannot be used. The official version of the player beam is `commit_beam`, which only works before the initial kickoff or after a team scores.

4.2.2. How to get up after falling?

Each behavior includes a `is_ready` method, which checks if the behavior is prepared for execution given the current agent/game conditions. Concerning the `Get_Up` behavior, `is_ready` returns True if the robot has fallen and needs to get up.

```
while True:
    if player.behavior.is_ready("Get_Up"):
        player.behavior.execute_to_completion("Get_Up")
```

```
    player.behavior.execute("Walk", w.ball_abs_pos[:2], True, None, True, None)
    player.scom.commit_and_send( w.robot.get_command() )
    player.scom.receive()
```

The `execute_to_completion` method is suitable only for testing purposes, as it blindly executes the behavior until the end, without considering its surroundings. For instance, if the robot is getting up and it is beamed by the referee (after some team scores or due to fouls), the get-up behavior should be aborted. The fail-safe approach is to use the normal `execute` method, although it requires the creation of a variable to indicate if the robot is currently getting up:

```
getting_up = False

while True:
    if player.behavior.is_ready("Get_Up") or getting_up:
        getting_up = not player.behavior.execute("Get_Up") # True on completion
    else:
        player.behavior.execute("Walk", w.ball_abs_pos[:2], True, None,True,None)
    player.scom.commit_and_send( w.robot.get_command() )
    player.scom.receive()
```

4.2.3. How to modify the walking parameters?

The **Walk** behavior has the following arguments:

- **target_2d**: 2D target in absolute or relative coordinates (use is_target_absolute to specify)
- **is_target_absolute**: True if target_2d is in absolute coordinates, False if relative to robot's torso
- **orientation**: absolute or relative orientation of torso, in degrees (or None to go towards target)
- **is_orientation_absolute**: True if orientation is relative to field, False if relative to robot's torso
- **distance**: distance to final target [0,0.5] (influences walk speed when approaching the final target)
    - set to None to consider target_2d the final target
    - set to 0.5 to remain at full speed near the target
    - this variable is not the same as setting the walking speed since the relation between distance and speed is not linear in this behavior, but, with that in mind, you can use it to control the speed.

Prevent the robot from slowing down near the ball. Change the distance argument to 0.5:

```
player.behavior.execute("Walk", w.ball_abs_pos[:2], True, None, True, 0.5)
```

Make the robot align its torso with the opponent's goal at all times. To use internal math operations, include the statement `from math_ops.Math_Ops import Math_Ops as M`. The `vector_angle` method calculates the angle of the vector originating from the robot's head 2D position and terminating at the center of the opponent's goal (15,0).

```
ori = M.vector_angle( (15,0)-w.robot.loc_head_position[:2] )
player.behavior.execute("Walk", w.ball_abs_pos[:2], True, ori, True, 0.5)
```

4.2.4. How to approach the ball already aligned with the opponent's goal?

A simple approach is to detect when the player is not aligned with the goal, and then position itself behind the ball. After the robot is aligned with the opponent's goal, it can walk forward and push the ball.

```
while True:
    player_2d = w.robot.loc_head_position[:2]
    ball_2d = w.ball_abs_pos[:2]

    if player.behavior.is_ready("Get_Up") or getting_up:
        getting_up = not player.behavior.execute("Get_Up") # True on completion
    else:
        ori = M.vector_angle( (15,0)-player_2d ) # Goal direction
        if M.distance_point_to_segment(player_2d,ball_2d, ball_2d
                            + M.normalize_vec( ball_2d-(15,0) ) ) > 0.1: # not aligned
            player.behavior.execute("Walk", ball_2d + M.normalize_vec(
                            ball_2d-(15,0) )*0.3, True, ori, True, None)
        else: # Robot is aligned
            player.behavior.execute("Walk", (15,0), True, ori, True, 0.5)
    player.scom.commit_and_send( w.robot.get_command() )
    player.scom.receive()
```

However, an issue arises when the player stands between the ball and the goal. The player's attempt to position itself behind the ball can inadvertently push the ball in the opposite direction. To address this, the **Path Manager** offers a solution by creating a path that navigates around obstacles, including the ball. For our specific objective of positioning the robot relative to the ball, we use the `get_path_to_ball` method (refer to the function docstring for details).

```
if M.distance_point_to_segment(player_2d,ball_2d, ball_2d
                    + M.normalize_vec( ball_2d-(15,0) ) ) > 0.1: # not aligned
    next_pos, next_ori, dist = player.path_manager.get_path_to_ball(
                    x_ori=ori, x_dev=-0.3, torso_ori=ori)
    player.behavior.execute("Walk", next_pos, True, next_ori, True, dist)
```

```
else: # Robot is aligned
    player.behavior.execute("Walk", (15,0), True, ori, True, 0.5)
```

4.2.5. How to kick the ball if it is on the opponent's side of the field?

Check if the first coordinate of the ball is greater than zero, and then execute the `Basic_Kick`, which is custom wrapper around a slot behavior defined with 2 keyframes. It internally prepares the robot for the kick using the **Path Manager**, as we did in the previous example. So, executing the `Basic_Kick` continuously is enough for the robot to autonomously chase the ball and kick it in the desired direction. Keep in mind that the `Basic_Kick` has a fixed distance and does not produce a very reliable kick.

```
while True:
    player_2d = w.robot.loc_head_position[:2]
    ball_2d = w.ball_abs_pos[:2]
    goal_dir = M.vector_angle( (15,0)-player_2d ) # Goal direction

    if player.behavior.is_ready("Get_Up") or getting_up:
        getting_up = not player.behavior.execute("Get_Up") # True on completion
    else:
        if ball_2d[0] > 0: # kick if ball is on opponent's side (x>0)
            player.behavior.execute("Basic_Kick", goal_dir)
        elif M.distance_point_to_segment(player_2d,ball_2d, ball_2d
                            + M.normalize_vec( ball_2d-(15,0) ) ) > 0.1: # not aligned
            next_pos, next_ori, dist = player.path_manager.get_path_to_ball(
                            x_ori=goal_dir, x_dev=-0.3, torso_ori=goal_dir)
            player.behavior.execute("Walk", next_pos, True, next_ori, True, dist)
        else: # Robot is aligned
            player.behavior.execute("Walk", (15,0), True, goal_dir, True, 0.5)
    player.scom.commit_and_send( w.robot.get_command() )
    player.scom.receive()
```

4.2.6. How to use command line arguments to initialize the agent?

To use the internal arguments system, import the class Script and create a new instance. Through that instance it is possible to access the value of each internal argument.

```
from scripts.commons.Script import Script

script = Script()
```

```
a = script.args

# Args: Server IP, Agent Port, Monitor Port, Uniform No., Robot Type, Team Name
player = Agent(a.i, a.p, a.m, a.u, a.r, a.t)
```

To list the parameters and their default values run:

```
python3 example.py -h
```

To change the default value edit “config.json” in the project’s root folder. At this point, if you call `python3 example.py` it will create an agent with the default parameters. To customize any parameter, simply follow the standard approach:

```
python3 example.py -t NewTeamName
```

4.2.7. How to draw on RoboViz for debugging?

When creating a `Base_Agent`, the drawings are enabled by default. However, you can enable them explicitly during the agent’s initialization:

```
player = Agent(a.i, a.p, a.m, a.u, a.r, a.t, enable_draw=True)
```

Then, add these line to the while block, at the end:

```
w.draw.annotation((*player_2d,0.6), "Hello!", w.draw.Color.white, "my_info",
flush=False)
w.draw.line(player_2d, ball_2d, 3, w.draw.Color.yellow, "my_info", flush=True)
```

The first command adds a text annotation on top of the robot that says `Hello!`, and the second command adds a line connecting the robot to the ball. Drawings can consist of a single element or a group, and have a unique identifier that allows us to overwrite previous drawings with the same name. In the above example, we are creating a drawing called `my_info` with two elements: an `annotation` and a `line`. We end the drawing by setting `flush=True`. (see world/commons/Draw.py for more drawing options)

4.2.8. Code with all the previous modifications

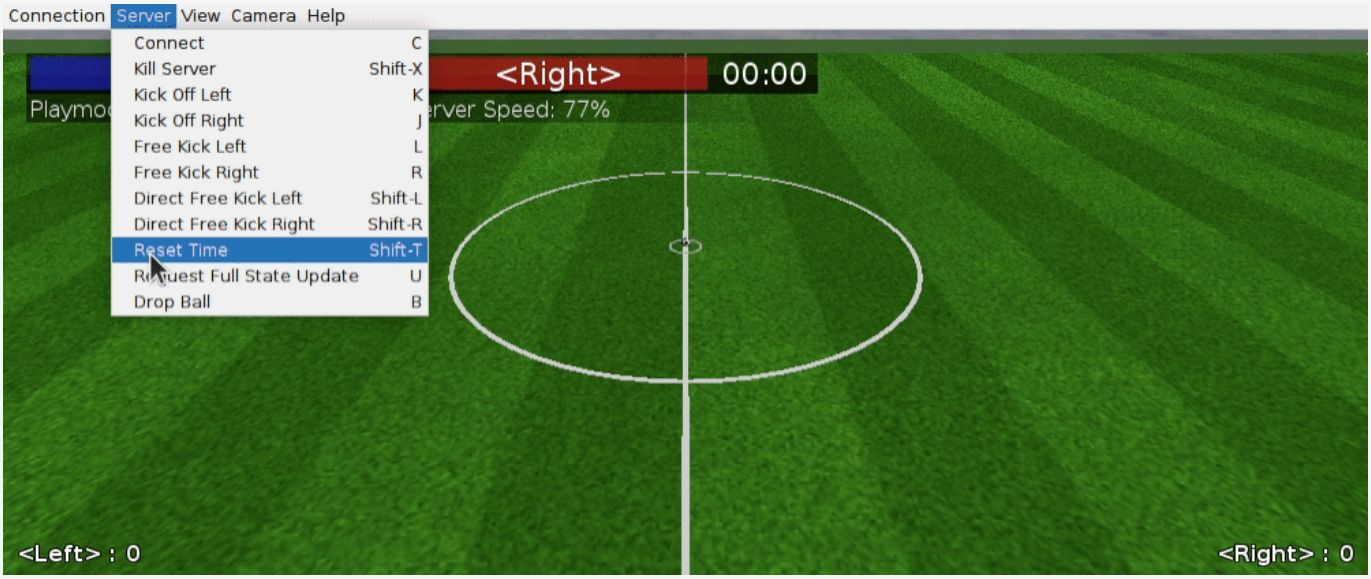

```
from agent.Base_Agent import Base_Agent as Agent
from math_ops.Math_Ops import Math_Ops as M
from scripts.commons.Script import Script

script = Script()
a = script.args

# Args: Server IP, Agent Port, Monitor Port, Uniform No., Robot Type, Team Name
player = Agent(a.i, a.p, a.m, a.u, a.r, a.t, enable_draw=True)

w = player.world
player.scom.unofficial_beam((-3,0,w.robot.beam_height), 0)

getting_up = False

while True:
    player_2d = w.robot.loc_head_position[:2]
    ball_2d = w.ball_abs_pos[:2]
    goal_dir = M.vector_angle( (15,0)-player_2d ) # Goal direction

    if player.behavior.is_ready("Get_Up") or getting_up:
        getting_up = not player.behavior.execute("Get_Up") # True on completion
    else:
        if ball_2d[0] > 0: # kick if ball is on opponent's side (x>0)
```

```
            player.behavior.execute("Basic_Kick", goal_dir)
        elif M.distance_point_to_segment(player_2d,ball_2d, ball_2d
                            + M.normalize_vec( ball_2d-(15,0) ) ) > 0.1: # not aligned
            next_pos, next_ori, dist = player.path_manager.get_path_to_ball(
                            x_ori=goal_dir, x_dev=-0.3, torso_ori=goal_dir)
            player.behavior.execute("Walk", next_pos, True, next_ori, True, dist)
        else: # Robot is aligned
            player.behavior.execute("Walk", (15,0), True, goal_dir, True, 0.5)
    player.scom.commit_and_send( w.robot.get_command() )
    player.scom.receive()


w.draw.annotation((*player_2d,0.6),"Hello!",w.draw.Color.white,"my_info",flush=False)
    w.draw.line(player_2d, ball_2d, 3, w.draw.Color.yellow, "my_info", flush=True)
```

4.3. Single-thread multi-agent examples

Agents can be instantiated by calling `example.py` multiple times with distinct uniform numbers (parameter -u). Alternatively, multiple agents can coexist within the same script and run on a single thread. This method simplifies debugging and is advantageous for learning scenarios that require a centralized control approach. Nevertheless, operating on a single thread may result in slow performance and is not suitable for official competitions, where each agent is required to function independently, communicating exclusively through the official communication channel.

4.3.1. One vs One



In this example, we create two agents representing opposing teams, each aiming to score against the other through kicks. Remember to change play mode to **PlayOn** to allow the ball to be moved.

```
from agent.Base_Agent import Base_Agent as Agent
from math_ops.Math_Ops import Math_Ops as M
from scripts.commons.Script import Script

script = Script()
a = script.args

# Args: Server IP, Agent Port, Monitor Port, Uniform No., Robot Type, Team Name
p1 = Agent(a.i, a.p, a.m, a.u, a.r, a.t)
p2 = Agent(a.i, a.p, a.m, a.u, a.r, "Opponent")
players = [p1,p2]

p1.scom.unofficial_beam((-3,0,p1.world.robot.beam_height), 0)
p2.scom.unofficial_beam((-3,0,p2.world.robot.beam_height), 0)

getting_up = [False]*2

while True:
    for i in range(len(players)):
        p = players[i]
        w = p.world
```

```
        player_2d = w.robot.loc_head_position[:2]
        ball_2d = w.ball_abs_pos[:2]
        goal_dir = M.vector_angle( (15,0)-player_2d ) # Goal direction

        if p.behavior.is_ready("Get_Up") or getting_up[i]:
            getting_up[i] = not p.behavior.execute("Get_Up") # True on completion
        else:
            p.behavior.execute("Basic_Kick", goal_dir)

        p.scom.commit_and_send( w.robot.get_command() )

    # all players must commit and send before the server updates
    p1.scom.receive()
    p2.scom.receive()
```

4.3.2. Three vs Three

The `Script` class provides batch operations to handle multiple agents. In the following example, we create 3 agents for each team, and all of them will try to kick the ball.

```
from agent.Base_Agent import Base_Agent as Agent
from math_ops.Math_Ops import Math_Ops as M
from scripts.commons.Script import Script

script = Script()
a = script.args

# Args: Server IP, Agent Port, Monitor Port, Uniform No., Robot Type, Team Name
# This could be done in 1 line, but calling batch_create twice enhances readability
script.batch_create(Agent, ((a.i,a.p,a.m,u+1,a.r,"Home") for u in range(3)) )
script.batch_create(Agent, ((a.i,a.p,a.m,u+1,a.r,"Away") for u in range(3)) )

players = script.players
p_num = len(players) # total number of players
script.batch_unofficial_beam( (-3, 4.5-abs(3*i-7.5), 0.5, 0) for i in range(p_num) )

getting_up = [False]*p_num

while True:
    for i in range(p_num):
        p = players[i]
        w = p.world
```

```
        player_2d = w.robot.loc_head_position[:2]
        ball_2d = w.ball_abs_pos[:2]
        goal_dir = M.vector_angle( (15,0)-player_2d ) # Goal direction

        if p.behavior.is_ready("Get_Up") or getting_up[i]:
            getting_up[i] = not p.behavior.execute("Get_Up") # True on completion
        else:
            p.behavior.execute("Basic_Kick", goal_dir)

    script.batch_commit_and_send()
    script.batch_receive()
```

You can change the number of players per team by modifying these two values (e.g. home team has 2 players and away team has 5):

```
script.batch_create(Agent, ((a.i,a.p,a.m,u+1,a.r,"Home") for u in range(2)) )
script.batch_create(Agent, ((a.i,a.p,a.m,u+1,a.r,"Away") for u in range(5)) )
```

5. Behaviors

There are 4 types of behaviors as introduced in the structure section: head, poses, slot behaviors, and custom. There is one head algorithm to control the head orientation, several poses composed of a single keyframe, several slot behaviors composed of multiple keyframes, and custom behaviors that generate an action every single time step. All behaviors must have a **unique name**, regardless of their type.

5.1. Usage

A behavior can be executed for a single time step, using:

```
player.behavior.execute("Walk", w.ball_abs_pos[:2], True, None, True, None)
player.scom.commit_and_send( w.robot.get_command() )
player.scom.receive()
```

The first command updates the internal array `w.robot.joints_target_speed` with target values for each joint. While some behaviors update this array directly, most convert target joint positions to target speeds using `w.robot.set_joints_target_position_direct`. The second command sends the target speeds to the server, and the third command waits for the server to send feedback. An alternative to calling these three methods is:

```
player.behavior.execute_to_completion("Get_Up")
```

which internally would call: (this code is simplified)

```
while not player.behavior.execute("Get_Up"):
    player.scom.commit_and_send( w.robot.get_command() )
    player.scom.receive()
```

This method leverages the fact that behaviors return `True` when completed. However, most behaviors never return `True`, since they do not end. Another limitation of this approach is that the behavior is blindly executed until the end, without considering the game state. So `execute_to_completion` should only be used in tests, not in the actual agent.

5.1.1. Remarks

- There is no need to manually reset a behavior. Internally, the behavior's reset function is automatically invoked if, in the previous time step, the agent was executing a different behavior.
- If you call `execute` twice before invoking `execute`, the second execution will overwrite `w.robot.joints_target_speed` for the set of joints controlled by the second behavior. As a result, the server will receive either only the target speed for the second behavior or a combination of speeds from both behaviors.
- After generating an action command string for the server using `w.robot.get_command()`, the array `w.robot.joints_target_speed` is reset to zero. In this way, if the target speed for a group of joints is not defined in a given time step, `w.robot.get_command()` assigns a target speed of zero to those joints.
    - Interestingly, if `w.robot.get_command()` were to exclude the joints that were not set from the action command string, the server would consider the last known speed targets for those specific joints, instead of assuming a speed of zero.

5.2. Custom Behavior

How to create a custom behavior called `My_Skill`:

- Create a folder for the behavior: /behaviors/custom/My_Skill
- Create the main file for the behavior with the same name: /behaviors/custom/My_Skill/My_Skill.py
- Add a class to My_Skill.py **with the same name as the file** and the following elements:

```
class My_Skill():

    def __init__(self, world) -> None:
        self.world = world
        self.description = "Shown when executing: Run_Utils.py -> Behaviors"
        self.auto_head = ##True to False##

    def execute(self,reset) -> bool: # You can add more arguments
        r = self.world.robot
        if reset:
            ##reset behavior##
        ##execute behavior##
        behavior_has_finished = ##True to False##
        return behavior_has_finished

    def is_ready(self) -> any: # You can add more arguments
```

```
        ''' Returns True if this behavior is ready to start/continue '''
        return True
```

These 3 functions are expected to exist (by the Behavior Class from /behaviors/Behavior.py):

__init__(self, world) -> None

- The initializer method must implement 2 class variables: '**self.description**' is a short description, '**self.auto_head**' should be True if the behavior allows autonomous head movement; set to False if the head should remain stationary or move as part of the behavior.

execute(self, reset [,optional_arg1,optional_arg2,...]) -> bool

- This function executes is called externally at every time step (0.02 s) while the behavior is being executed

is_ready(self [,optional_arg1,optional_arg2,...]) -> any

- This function is not used internally, it's meant for the Agent. It indicates whether the behavior is ready to start or continue under the current game conditions.

In the end, add your new class to the declaration of behaviors in /behaviors/behavior.py

5.2.1. Usage

Suppose in the beginning we are walking towards coordinates (0,0). This command will internally call `walk.execute(reset, (0,0), True, ori, True, dist)` from behaviors/custom/Walk/Walk.py, where the `reset` argument is automatically set to `True` when switching from another behavior.

```
player.behavior.execute("Walk", (0,0), True, ori, True, dist)
```

At a certain point we will call our behavior, which will internally call `my_skill.execute(True)`.

```
player.behavior.execute("My_Skill")
```

And in the following time steps, the same command will call `my_skill.execute(False)`.

If `My_Skill` defines extra arguments for the `execute` method, such as:

```
def execute(self, reset, my_arg1, my_arg2, my_arg3=10) -> bool:
```

to execute `My_Skill`, you must provide values for the arguments in the order they are defined, without explicitly naming them, yielding two options, since there is one optional argument:

```
player.behavior.execute("My_Skill", value1, value2, value3)
# OR
```

```
player.behavior.execute("My_Skill", value1, value2)
```

5.2.2. Running nested behaviors

Custom behaviors can internally invoke other behaviors. This happens in the `Basic_Kick`, where it internally executes the `Walk`. However, instead of `execute` we must use `execute_sub_behavior`. This ensures that the behavior manager, implemented in 'Behavior.py,' correctly assumes that the primary behavior (in this case, `Basic_Kick`) is still in progress. Otherwise, unexpected situations may occur. When using `execute_sub_behavior`, the reset is not handled automatically. It must be explicitly included as the second argument, as shown below:

```
behavior.execute_sub_behavior("Walk", reset, pos, True, ori, True, dist)
```

6. Logs

Logs are files that can be used to record errors, warnings or any kind of useful information. During execution, the first log entry will trigger the agent to create a single folder in the logs directory:

./logs/yyyy-mm-dd_hh-mm-ss__RANDOMSTRING/

There is a random string at the end to avoid collisions if several agent instances are running in parallel. Inside that folder are stored all logs of the current session. The name of each log file is given by the log's topic. By default, each agent has a unique log file, whose topic is given by the agent's team and uniform number:

./logs/yyyy-mm-dd_hh-mm-ss__RANDOMSTRING/TeamName_Unum.log

To write something to the agent's log, use:

```
player.world.log("Found something worth logging!")
```

The log function above is a shortcut for:

```
player.logger.write("MyFile.py found something worth logging!", True,
player.world.step)
```

The `write` function receives the message to be written, a boolean to indicate if a timestamp is required, and the number of the current simulation step, which is written to the file before the actual message. The step number is optional but it helps when reading logs, to know when something happened.

`player.logger` is the agent's data logger object, from the class `Logger` and it was created like this in **Base_Agent.py**:

```
self.logger = Logger(is_enabled=enable_log, topic=f"{team_name}_{unum}")
```

To disable the agent's log, set enable_log to False during instantiation:

```
player = Agent(..., enable_log = False)
```

Or to disable the log temporarily:

```
player.logger.enabled = False
# Do something
player.logger.enabled = True
```

To create a new log file:

```
from logs.Logger import Logger
mylogger = Logger(is_enabled=True, topic="mylogger")
mylogger.write(msg="my message 1", step=player.world.step)
mylogger.write(msg="my message 2", step=player.world.step)
```

This code generates a log file at **/logs/yyyy-mm-dd_hh-mm-ss__RANDOMSTRING/mylogger.log** and writes both messages with a timestamp prefix.

7. Team Communication

The team communication channel provided by the server is used to its full extent by the agent (through the `Radio` class defined in communication/Radio.py). It is currently used to transmit the position and state of every visible robot and the position of the ball. For a detailed algorithm explanation see 2022 Champions paper.

7.1. How does the communication channel work?

- The `commit_announcement` command allows an agent to broadcast a message to every player on the field
- The message range is 50m (the field is 36m diagonally, so ignore this limitation)
- The hear perceptor indicates 3 things:
    - the message
    - the origin team
    - the origin absolute angle (set to "self" if the message was sent by oneself)
- Messages are heard in the next step
- Messages are only sent every 2 steps (0.04s)
- Messages sent in muted steps are only heard by oneself
- In any time step, a player can only hear one other player besides itself
- If two other players say something, only the first message is heard
    - This ability exists independently for messages from both teams, meaning that teams cannot spam the communication channel to block the opponent's communication
- In theory, a player can hear its own message + the 1st teammate to speak + the 1st opponent to speak
- In practice, the opponent doesn't matter because our team's parser ignores messages from other teams

Message characteristics:

- Max. 20 characters, ascii between 0x20, 0x7E except space ' ' and left/right parenthesis '(' and ')'
- However, the server has some bugs:
    - sending ' or " cuts off the message at that point
    - '\' at the end or near another '\'
    - ';' at beginning of message

8. Pathfinding

The agent uses a custom A* pathfinding implementation library in C++, optimized for the soccer environment (the algorithm implementation is briefly explained in 2022 Champions paper).

The path can be computed by invoking `a_star.compute(param_vec)`, where `param_vec` is defined as a NumPy array (float 32) with the following elements:

Index	Description
0	start x field coordinate
1	start y field coordinate
2	allow path to go out of bounds? (useful when player does not have the ball)
3	go to opponent's goal? (path goes to the most efficient part of the goal)
4	target x field coordinate (only used if param_vec[3]==0)
5	target y field coordinate (only used if param_vec[3]==0)
6	timeout in us (maximum execution time)
(optional)	
7..11	**obstacle 1**: x, y, hard radius (max:5m), soft radius (max:5m), repulsive force for soft radius (min:0)
12..16	**obstacle 2**: x, y, hard radius (max:5m), soft radius (max:5m), repulsive force for soft radius (min:0)
…	**obstacle n**: x, y, hard radius (max:5m), soft radius (max:5m), repulsive force for soft radius (min:0)

It returns a NumPy array (float32), where 'n' represents the last index, containing the following elements:

Index	Description

0..n-2	• contains path from start to target (up to a maximum of 1024 positions)
	• each position is composed of x,y coordinates (so, up to 2048 coordinates)
	• the return vector is flat (1 dimension) (e.g. [x1,y1,x2,y2,x3,y3,...])
	• reasons why path may not end in target:
	  ○ path is longer than 1024 positions (which is at least 102 meters!)
	  ○ reaching target is impossible or timeout (in both cases, the path ends in the closest position to target that has been found)

n-1	number indicating the path status: **0** - success **1** - timeout before the target was reached (may be impossible) **2** - impossible to reach target (all options were tested) **3** - no obstacles between start and target (returned path has only 2 points: start & target)
n	A* path cost

8.1. Map of field

- The algorithm has a 32m by 22m map with a precision of 10cm (same dimension as field +1 meter border)
- The map contains information about field lines, goal posts and goal net
- The path may go outside the field (out of bounds) if the user allows it, but it may never go through goal posts or the goal net (these are considered static inaccessible obstacles)
- The user must only specify dynamic obstacles through the arguments

8.2. Repulsive force

- The repulsive force is implemented as an extra cost for the A* algorithm
- The cost for walking 10cm is 1, and the cost for walking diagonally is sqrt(2)
- The extra cost of stepping on a position with a repulsive force f=1 is 1
- For any given position on the field, the repulsive force of >=2 objects is combined with the max function, max(f1,f2), NOT f1+f2!
- If path starts on inaccessible position, it can go to a neighbor inaccessible position but there is a cost of 100 (to avoid inaccessible paths)

Example 1:

```
Map 1     Map 2     Map 3
..x..     ..o..     ..o..
..1..     ..o..     .o1..
..o..     ..o..     ..o..
```

- Consider **Map 1** where 'x' is the target, 'o' is the player, and '1' is a repulsive force of 1
- In **Map 2**, the player chooses to go forward, the total cost of this path is: 1+(extra:1)+1 = 3
- In **Map 3**, the player avoids the repulsive force, cost: sqrt(2)+sqrt(2) = 2.83 (optimal solution)

Example 2:

```
 Map 1      Map 2      Map 3      Map 4
...x...    ..oo...    ...o...    ...o...
..123..    .o123..    ..o23..    ..1o3..
...o...    ..oo...    ...o...    ...o...
```

- Consider **Map 1** with 3 positions with 3 distinct repulsive forces, going from 1 to 3.
- In **Map 2**, the player avoids all repulsive forces, cost: 1+sqrt(2)+sqrt(2)+1 = 4.83
- In **Map 3**, the player goes through the smallest repulsive force, cost: sqrt(2)+(extra:1)+sqrt(2) = 3.83 (optimal solution)
- In **Map 4**, the player chooses to go forward, cost: 1+(extra:2)+1 = 4.00

8.3. Obstacles

- hard radius: inaccessible obstacle radius (very high repulsive force)
- soft radius: accessible obstacle radius with user-defined repulsive force (fades with distance) (disabled if <= hard radius)

Examples:

obstacle(0,0,1,3,5) -> obstacle at pos(0,0), hard radius of 1m, soft radius of 3m & repulsive force 5

- the path cannot be at <=1m from this obstacle, unless the path were to start inside that radius
- the soft radius force is maximum at the center (5) and fades with distance until (0) at 3m from the obstacle
- so to sum up, at a distance of [0,1]m the force is very high, [1,3]m the force goes from 3.333 to 0

obstacle(-2.1,3,0,0,0) -> obstacle at pos(-2.1,3), hard radius of 0m, soft radius of 0m & repulsive force 0

- the path cannot go through (-2.1,3)

obstacle(-2.16,3,0,0,8) -> obstacle at pos(-2.2,3), hard radius of 0m, soft radius 0m & repulsive force 8

- the path cannot go through (-2.2,3), the map has a precision of 10cm, so the obstacle is placed at the nearest valid position
- the repulsive force is ignored because (soft radius <= hard radius)

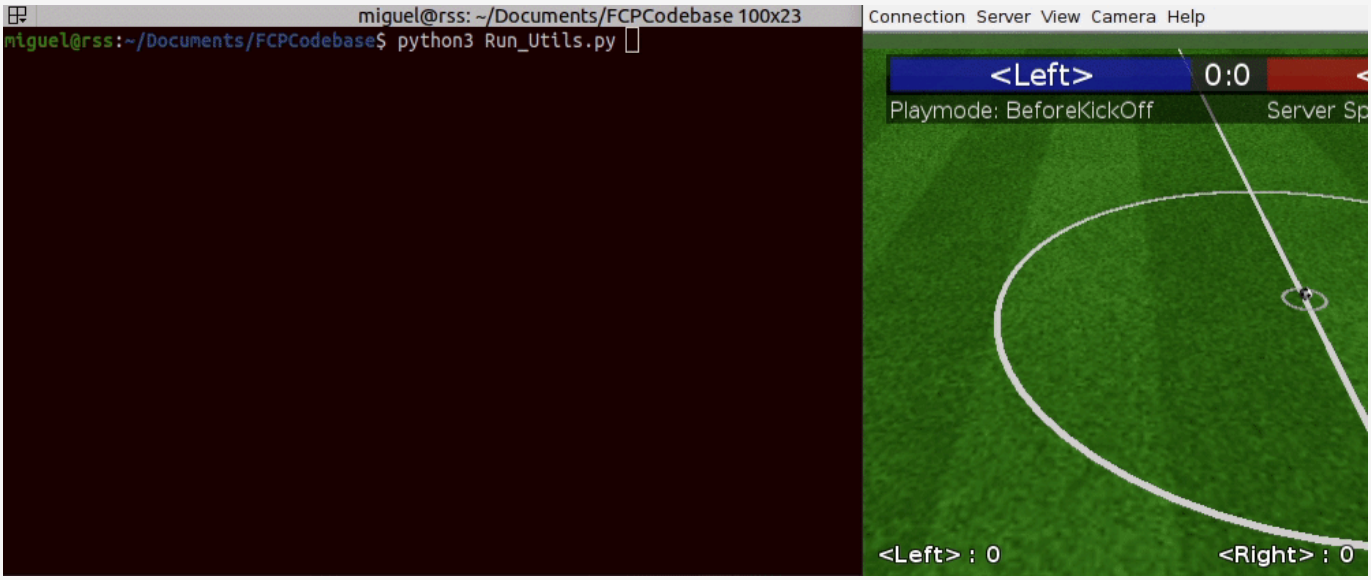
9. Utilities

9.1. Beam

Beam player to a specific coordinate, and change its orientation, between -180 and 180 degrees.

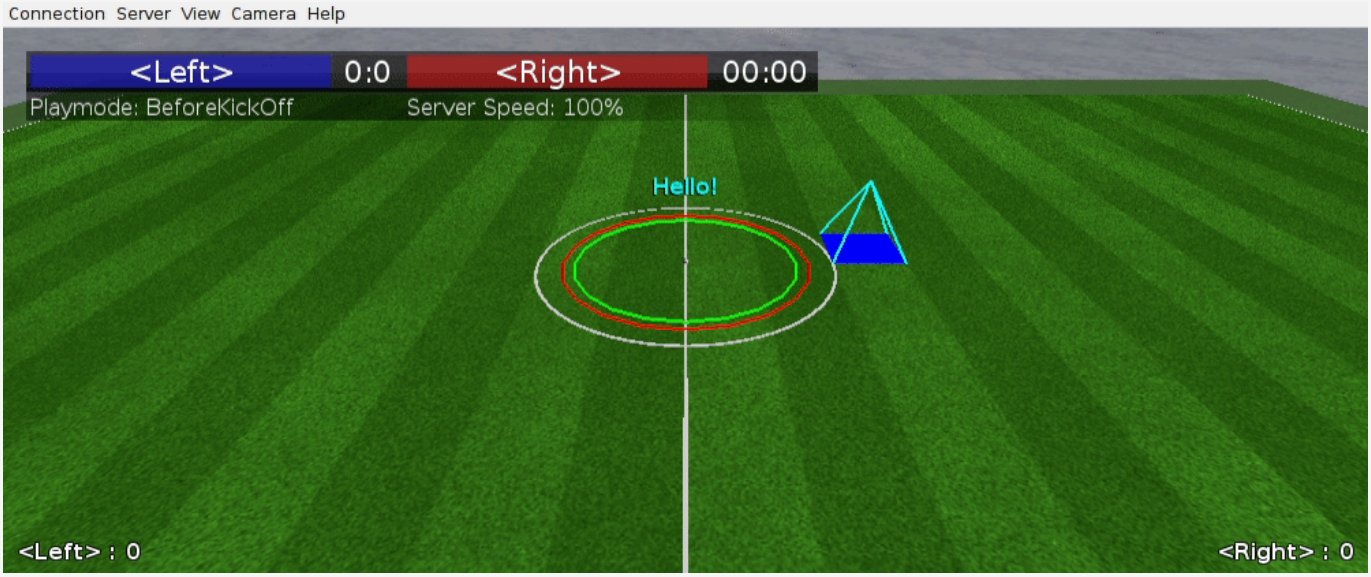
9.2. Behaviors

Interactively test all behaviors, including new XML-based skills, which are automatically loaded.



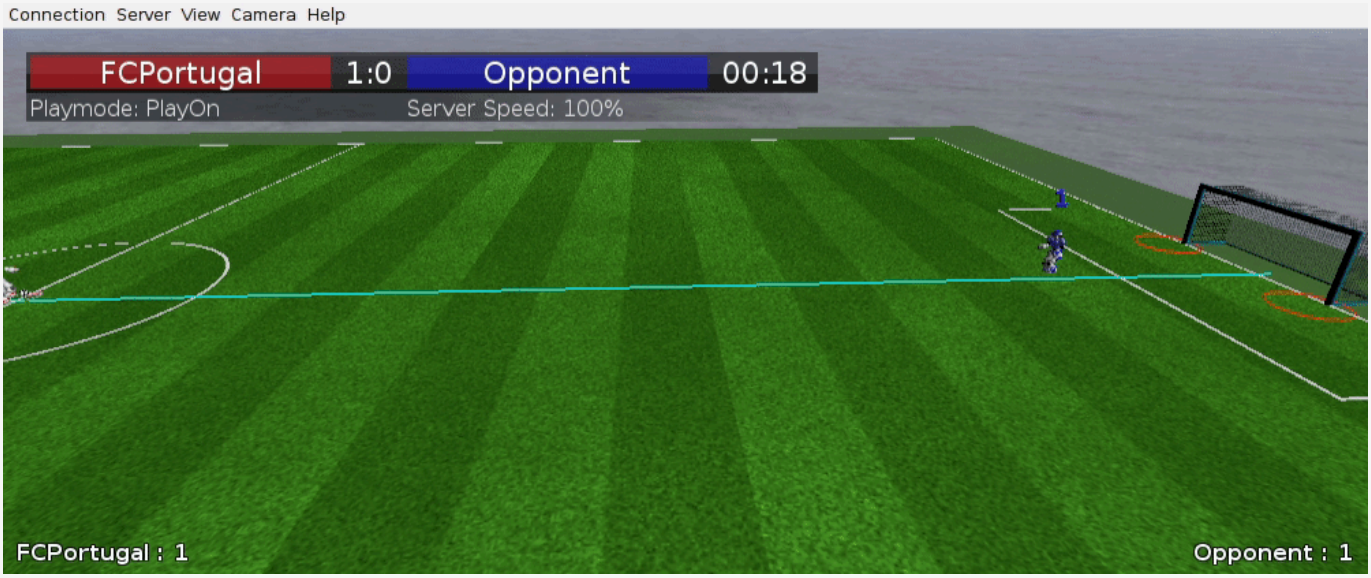
9.3. Drawings

Demonstration of drawings.



9.4. Dribble

Using the dribble skill to score.



9.5. Forward Kinematics

Absolute positions of body parts and joints, and 3D orientation of body parts, using forward kinematics and vision to self-locate.

9.6. Get-Up

Tests the get-up behavior in different conditions.

9.7. IMU

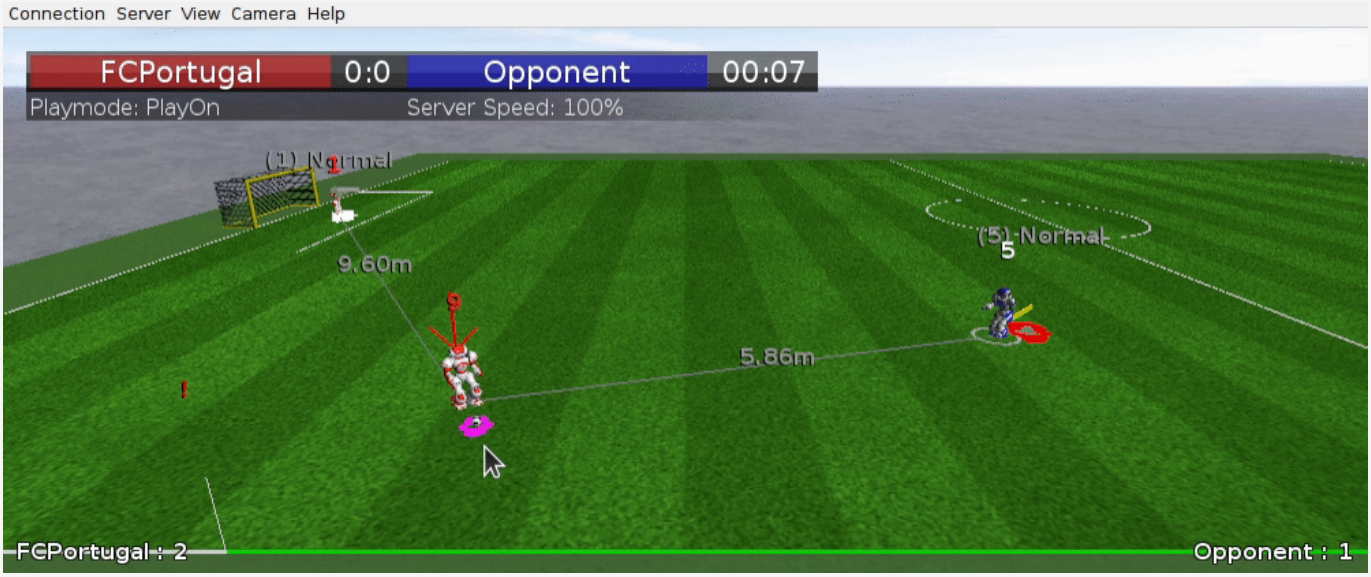
Example of using the IMU (accelerometer + magnetometer). For orientation, it is accurate. For translation it should be avoided due to accumulated errors. (see ‘loc’ and ‘imu’ variables in world/Robot.py)

9.8. Kick

The robot will kick towards the center of the field.

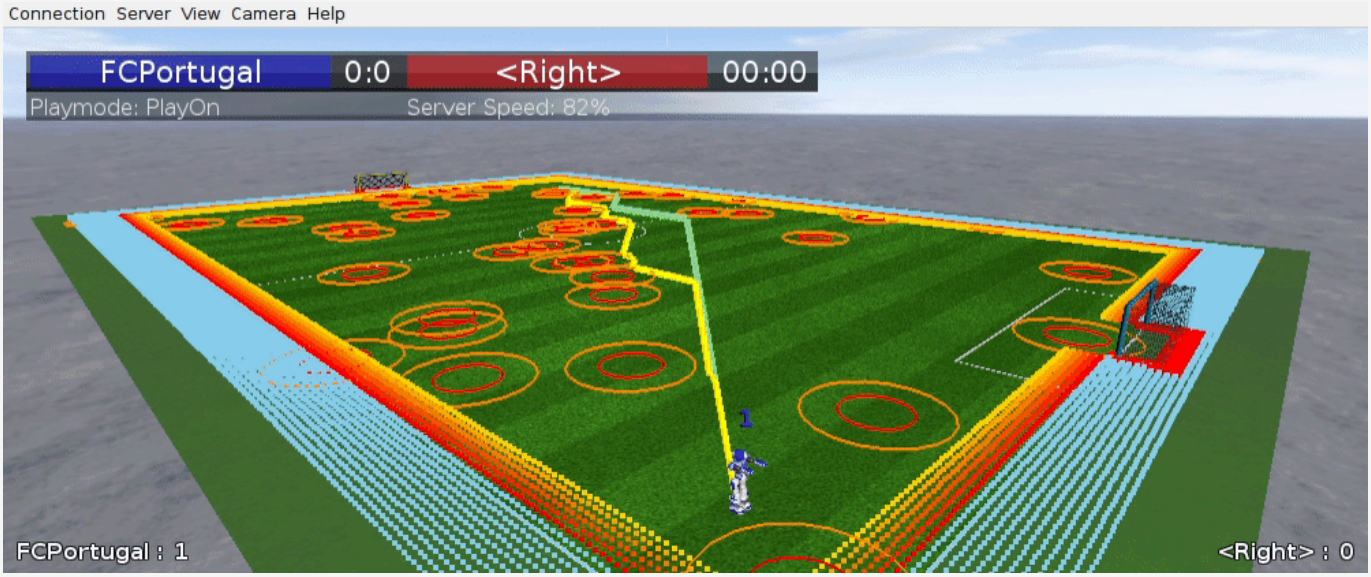
9.9. Localization

This demo draws what the agent can perceive from the environment within its limited FOV, including field lines, goal posts, corner flags, teammates, opponents, and the ball.



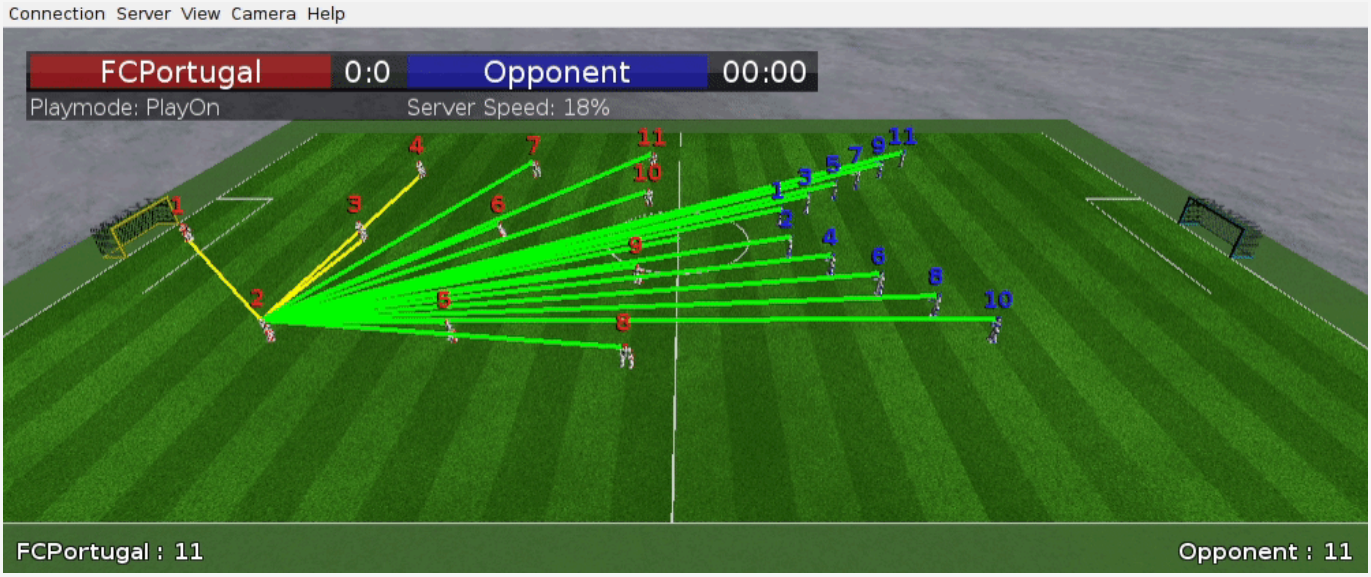
9.10. Pathfinding

Demonstration of pathfinding around virtual obstacles with two routes: one from the agent to the ball, allowing for out-of-bounds movement, and another from the ball to the agent, constrained within bounds.



9.11. Radio Localization

Green lines signal players/ball within the FOV. Yellow lines indicate player/ball locations known through teammate communication. Red lines would indicate unknown positions (i.e. failing communication).



10. Reinforcement Learning Gyms

10.1. Train Example

In this example, we will train the robot to fall as fast as possible.

Run Utils script with robot type 0 (or any other robot type)

```
python3 Run_Utils.py -r 0
```

Select: **Server** to ensure proper server configuration (soccer rules, sync mode, real time, cheats)

```
=============================================================================================
       Setting      | Value |                    Description
--------------------+-------+-------------------------------------------------------------
0-Official Config   |  Off  |  Configuration used in official matches
1-Penalty Shootout  |  Off  |  Server's Penalty Shootout mode
2-Soccer Rules      |  On   |  Play modes, automatic referee, etc.
3-Sync Mode         |  On   |  Synchronous communication between agents and server
4-Real Time         |  Off  |  Real Time (or maximum server speed)
5-Cheats            |  On   |  Agent position & orientation, ball position
6-Full Vision       |  Off  |  See 360 deg instead of 120 deg (vertically & horizontally)
7-Add Noise         |  On   |  Noise added to the position of visible objects
8-25Hz Monitor      |  On   |  25Hz Monitor (or 50Hz but RoboViz will show 2x the actual speed)
=============================================================================================
```

Return to main menu
Select: GYMS/Fall
Select: Train

If you see this message, the optimization is running:

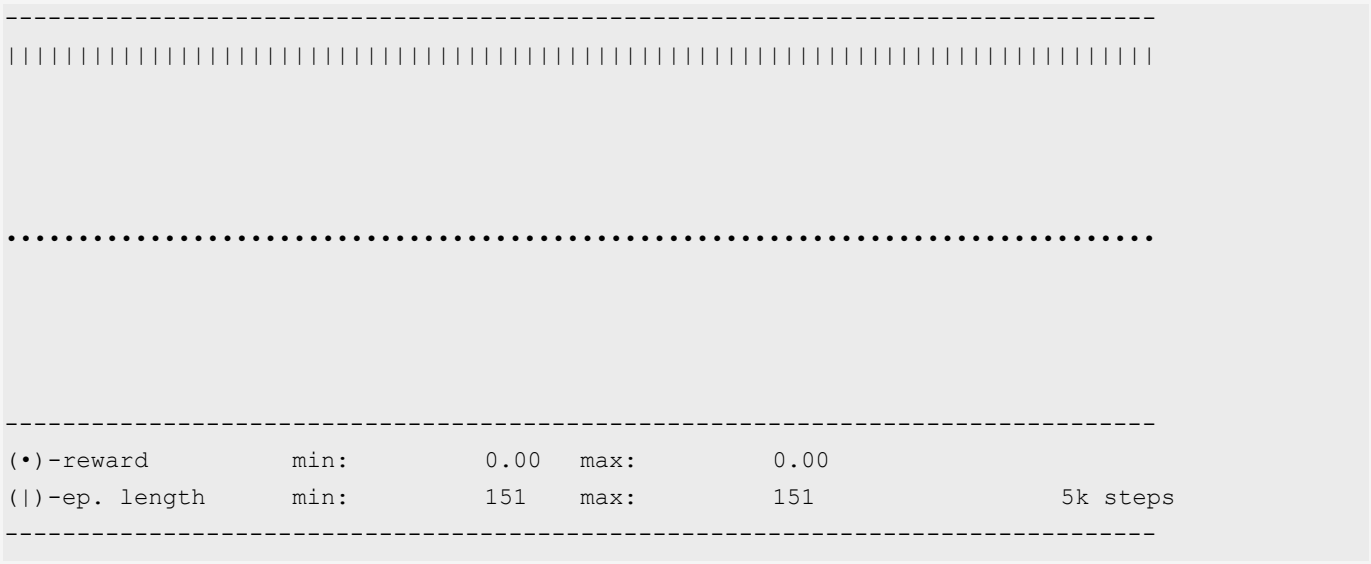
```
----------------------------
| time/              |     |
|    fps             | 294 |
|    iterations      | 1   |
|    time_elapsed    | 1   |
|    total_timesteps | 512 |
----------------------------
```

During training, 5 servers + 5 gym threads are launched:

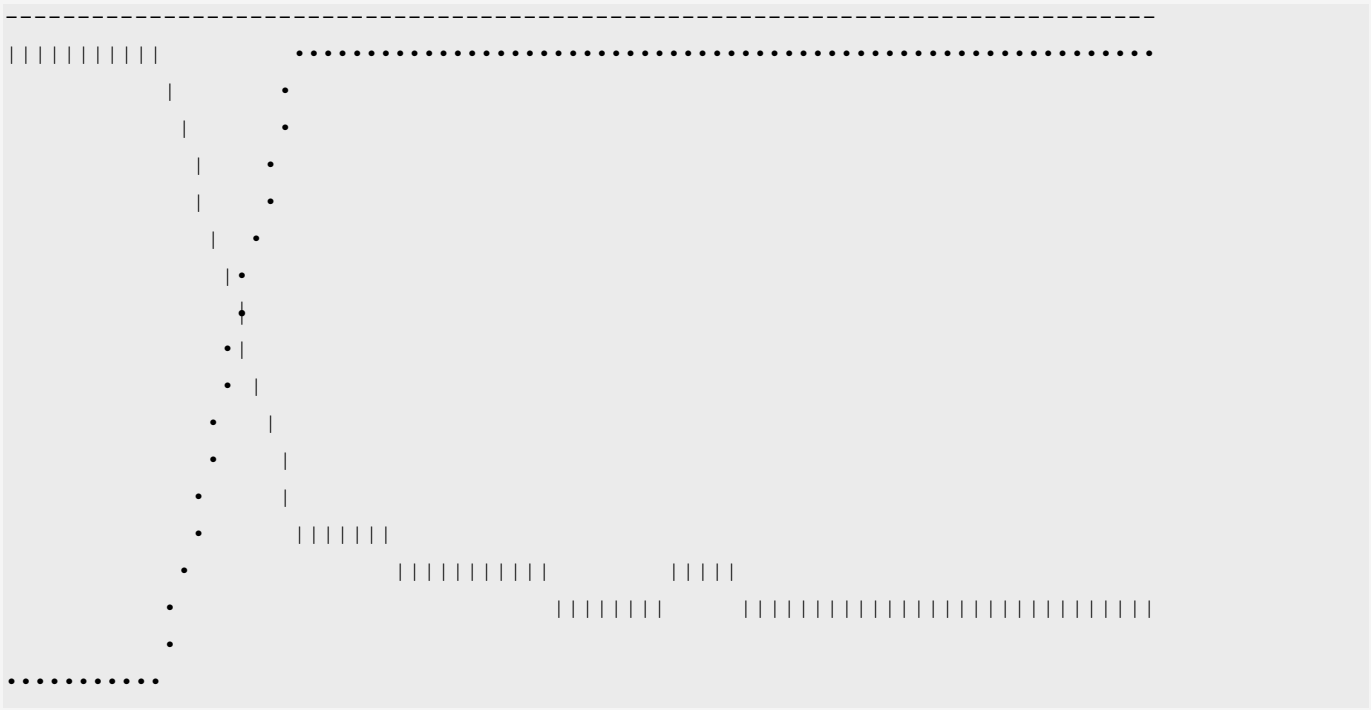
- Train Gym no. 1 can be seen through RoboViz at port 4200 (‘roboviz.sh --serverPort=4200’)
- Train Gym no. 2 can be seen through RoboViz at port 4201

- Train Gym no. 3 can be seen through RoboViz at port 4202
- Train Gym no. 4 can be seen through RoboViz at port 4203
- An Evaluation Gym can be seen at port 4204 (runs once every 10 train iterations)

A new evaluation graph is made by our codebase every 10 iterations, after evaluation:



After about 5 minutes, the train ends, and the final evaluation is shown. The agent receives a reward of 1 every time the robot falls within 3 seconds (the maximum episode duration). The algorithm will try to obtain that reward as fast as possible, which leads to the episode duration dropping from 150 time steps (3 seconds, since every time step is 0.02 s), to about 20 time steps (which is equivalent to 0.4 seconds).



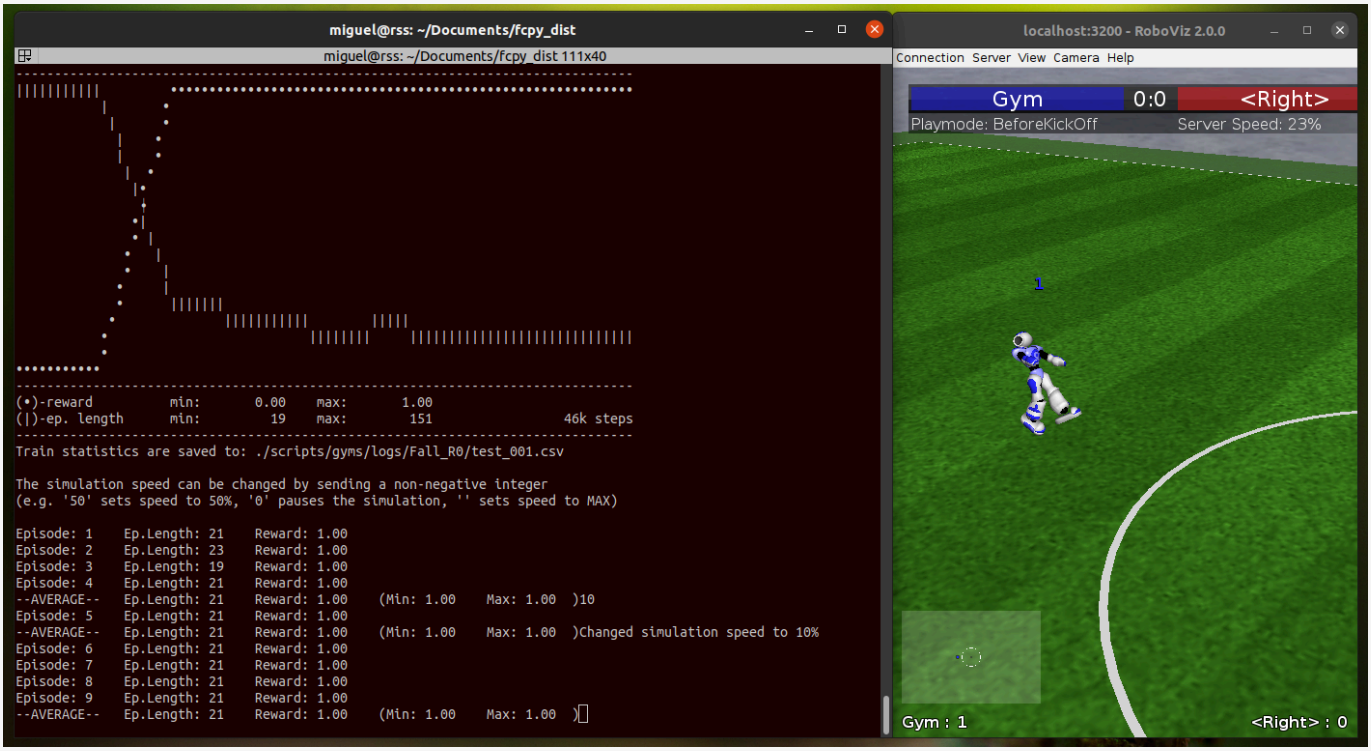
```
--------------------------------------------------------------------------------
(•)-reward           min:        0.00     max:        1.00
(|)-ep. length       min:          19     max:         151                  46k steps
--------------------------------------------------------------------------------
Train start:     13/11/2023 23:57:52
Train end:       14/11/2023 00:02:48
Train duration:  0:04:55
Model path:      ./scripts/gyms/logs/Fall_R0/
Killed 5 rcssserver3d processes starting at 3100
```

To test the trained model, open RoboViz on port 3200.
If you exited the script after the train has ended, launch it again:

- Run Utils script with the same robot type used for training
  ```
  python3 Run_Utils.py -r 0
  ```
- Select: GYMS/Fall
- Follow the instructions below

If you did not exit the script after the train has ended, start here:

- Select: Test
- Select the first model (they are sorted by date)
- Select the best model or last model
  - The model should be now running on RoboViz
- In the terminal, write **10** and press **ENTER** to reduce the simulation speed to 10% of real time, to better analyze the behavior in RoboViz.



Abort the test with CTRL+C.
If an error occurs during training or testing, the servers will run indefinitely until manually stopped:

```
pkill rcssserver3d -e -9; pkill simspark -e -9
```

10.2. Simultaneous train sessions

Multiple train sessions can run simultaneously as long as they use different ports for agent and monitor. Example:

```
python3 Run_Utils.py
```

If the default ports for agent and monitor in "config.json" are 3100 and 3200, then `Run_Utils.py` will train the existing gyms in ports:

- **Thread 0**: *launches 1 server with agent port 3100 and monitor port 4200, and connects to it*
- **Thread 1**: *launches 1 server with agent port 3101 and monitor port 4201, and connects to it*
- …
- **Thread N**: *launches 1 server with agent port 3100+N and monitor port 4200+N, and connects to it*

Note that it adds 1000 to the default monitor port, so that gyms can be trained with many simultaneous threads, while avoiding port collisions.

If the user chooses to test an existing model, instead of training a new one, only 1 thread is used. The thread launches a server with (**agent port**: default_port-1) and the default monitor port, as shown below:

- **Thread 0**: *launches 1 server with agent port 3099 and monitor port 3200, and connects to it*

So, by default, you can have 2 instances of `Run_Utils.py` running in parallel, without changing the default ports, as long as one is training and the other is testing. If you want to train 2 models in parallel, specify new ports when invoking the second `Run_Utils.py`:

```
python3 Run_Utils.py -p 5100 -m 5200
```

And if you need to train or test even more models in parallel, just specify other ports (ensure you space them apart to prevent collisions, especially when using many threads per training):

```
python3 Run_Utils.py -p 7100 -m 7200
python3 Run_Utils.py -p 9100 -m 9200
…
```

10.3. Integrate behavior in team

After running the previous test example, inside the project's root folder, the directory **root_folder/scripts/gyms/logs/Fall_R0/** will contain a file called **last_model.zip.pkl** or **best_model.zip.pkl** or both, depending on the model that was tested in the previous step.

- Note that the .pkl file is only created while testing, and not during training.
- The .pkl file can be used by custom behaviors.

See **root_folder/behaviors/custom/Fall/** for an example integration that applies the trained model to any robot type (note that robot type 4 has 2 extra joints (toes), for instance if you trained type 0, then type 4 will assume that the toes are not controlled).

You can rename the .pkl you trained and replace the fall.pkl in **root_folder/behaviors/custom/Fall/**

To test the integrated behavior, set **Real Time→On** (optional server configuration), then run the server:

```
rcssserver3d
```

Run the Utils script with the desired robot type:

```
python3 Run_Utils.py -r 0
```

Select: Behavior
Select: Fall
Select: Get_Up
Select: Fall
…

10.4. Create a custom gym

To create your own gym, start by duplicating one of the existing gym files. Maybe duplicate Basic_Run.py for locomotion related tasks, or Fall.py for anything else. This is just a suggestion to reduce the initial configuration effort. If you decide to duplicate Fall.py, please be aware that the PPO hyperparameters are intentionally set to be simplistic and may not effectively train more complex behaviors (especially n_steps_per_env, total_steps and learning_rate).

You can create an empty file as long as:

- It contains a class with the same name as the file, that inherits from `gym.Env`, where `gym` is the OpenAI Gym class
    - The class must define some methods as defined by the OpenAI Gym:

        - reset(self) -> observation
        - step(self, action) -> observation, reward, done, info
        - render(self, mode) -> None
        - close(self) -> None
    - The class must define: `self.observation_space` and `self.action_space`
- It contains a class `Train` that must define two methods to be called by `Run_Utils.py`
    - __init__(self, script) -> None
    - train(self, args) -> None
        - when training a new mode, args is an empty dictionary
        - when retraining a model, args contains a dictionary with 3 items
          `{"folder_dir":fd, "folder_name":fn, "model_file":mf}`
    - test(self, args) -> None
        - args contains a dictionary with 3 items
          `{"folder_dir":fd, "folder_name":fn, "model_file":mf}`

'args' example:

```
{'folder_dir': './scripts/gyms/logs/Fall_R1', 'folder_name': 'Fall_R1',
'model_file': './scripts/gyms/logs/Fall_R1/best_model.zip'}
```

Train your custom gym like the train example and then integrate it into the agent by creating a new behavior.

10.4.1. Referee interference

By default, when the gym initiates a server instance, it begins in the "BeforeKickOff" play mode. In a training context, this default setting can be problematic, as the referee may disrupt the training process by adjusting the ball's position or beaming players for entering the opponent's field. There are at least three solutions:

- turning off the referee through the server's configuration parameters (this option has not been working lately due to a server bug);
- set play mode to GameOver (self.player.scom.unofficial_set_play_mode("GameOver")) at the end of the gym's __init__ method;
- set play mode to PlayOn (self.player.scom.unofficial_set_play_mode("PlayOn")) at the end of the gym's __init__ method and then reset the game time frequently (or at every step) (player.scom.unofficial_set_game_time(0)). This approach has the benefit of invoking self collisions, which might be beneficial when learning some behavior. However, the referee will still intervene in situations such as the ball going out of bounds or entering any goal.

11. Generate Binary for Competitions

Within the project's root directory, specifically in the "bundle" folder, you'll find a bash script named "bundle.sh." This script is responsible for creating a binary named "fcp," which, when executed, internally runs the Python script "Run_Player.py". To merge the application and all its dependencies into a single executable file, PyInstaller is invoked with the "--onefile" flag.

The files that are included in the executable are defined in this section of “bundle.sh”:

```
pyinstaller \
--add-data './world/commons/robots:world/commons/robots' \
--add-data './behaviors/slot/common:behaviors/slot/common' \
--add-data './behaviors/slot/r0:behaviors/slot/r0' \
--add-data './behaviors/slot/r1:behaviors/slot/r1' \
--add-data './behaviors/slot/r2:behaviors/slot/r2' \
--add-data './behaviors/slot/r3:behaviors/slot/r3' \
--add-data './behaviors/slot/r4:behaviors/slot/r4' \
--add-data './behaviors/custom/Dribble/*.pkl:behaviors/custom/Dribble' \
--add-data './behaviors/custom/Walk/*.pkl:behaviors/custom/Walk' \
--add-data './behaviors/custom/Fall/*.pkl:behaviors/custom/Fall' \
${onefile} --distpath ./bundle/dist/ --workpath ./bundle/build/ --noconfirm
--name fcp Run_Player.py
```

As an example,

```
--add-data './world/commons/robots:world/commons/robots'
```

adds all the files contained in `'./world/commons/robots'` to a directory, nested in the executable, with the same structure `'world/commons/robots'`.

Upon running the "bundle.sh" script, two folders are created:

- “bundle/**build**/” with temporary files that can be ignored or deleted after the compilation;
- “bundle/**dist**/” with the “fcp” binary, along with useful scripts to launch the team using the binary, and a kill script that is required for RoboCup competitions.

The binary can be invoked with the same arguments as the original “Run_Player.py” script. However, the debug mode (e.g. ‘fcp -D 1’) is ignored when executing the binary to prevent its accidental usage during competitions. This feature is implemented in “/scripts/commons/Script.py” through:

```
if getattr(sys, 'frozen', False): # disable debug mode when running from binary
   self.args.D = 0
```

12. Additional resources

This codebase release forms an integral part of the paper:

- Abreu, M., Reis, L. P., & Lau, N. (2023). Designing a Skilled Soccer Team for RoboCup: Exploring Skill-Set-Primitives through Reinforcement Learning. *arXiv preprint arXiv:2312.14360*.

Other relevant papers:

- Abreu, M., Kasaei, M., Reis, L. P., & Lau, N. (2022). FC Portugal: RoboCup 2022 3D Simulation League and Technical Challenge Champions. In *Robot World Cup* (pp. 313-324). Cham: Springer International Publishing.

Youtube videos: https://www.youtube.com/@m_abreu/videos