Immortals 2020 Extended Team Description Paper

Ali Salehi[1], Omid Najafi[1], Ali Amouzandeh[1], MohammadAli Ghasemieh[2], MohammadHossein Fazeli[3], MohammadReza Niknezhad[3], Mohammad Tabasi[3], and Mustafa Talaeezadeh[1]

[1] Sharif University of Technology
[2] Pars University of Art
[3] University of Science and Technology

Abstract. This paper describes the recent work done by the Immortals Robotics Team for the upcoming competitions including the RoboCup 2023.

Keywords: RoboCup 2023 · Small Size League

1 Introduction

The Immortals Robotics Team consists of art and engineering students from different Persian universities. The team was established in 2007 with the focus of designing a robust robot while meeting all the requirements according to the Small Size League rules. The team's strategy is to adapt itself with all the rule changes in order to experience the new challenges. This makes the team to take part in the division-A which is intended for the advanced teams of the league [1].

In the previous years the team reached close enough to a SSL robot design and solved most of the issues found in the previous designs. The process of upgrading the robot can be seen in the previous years TDPs and ETDPs [3] of this team. The readers who are interested in designing or upgrading a SSL robot are encouraged to study the previous papers of this team.

Our team, Immortals, is thrilled to participate in this year's RoboCup SSL competition. With a history of innovation and success, we are constantly working to develop cutting-edge solutions in small-scale autonomous robotics. Building on our experience from last year's competition, we are currently dedicated to improving our AI software and robot electronics to address the needs we identified in our previous system. Our team is working tirelessly on designing and constructing a robot that will be smarter and more robust than ever before. Through an iterative process of development and testing, we are confident that we will create a robot team that is faster, smarter, and more agile than ever before. We look forward to showcasing our progress and competing against the best SSL teams from around the world.

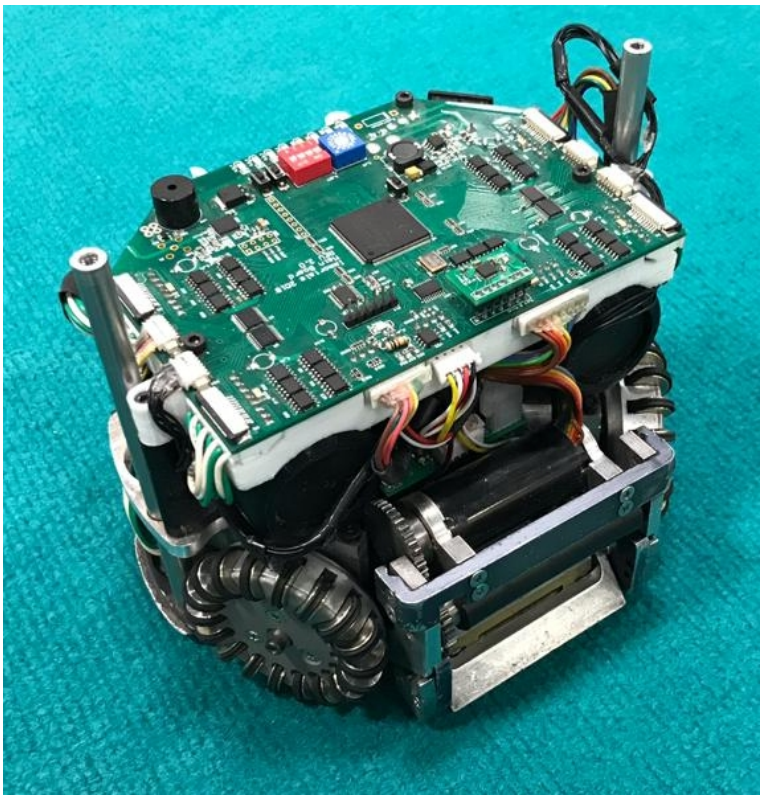

Fig. 1. Immortals current robot.

2 Software

Our team's main focus this year has been on designing a more effective AI system for our robot, with a particular emphasis on improving our debugging process. To achieve this goal, we have redesigned our AI system from the ground up, using C++20 and a modular architecture consisting of three classes: ExternalWorld, WorldState, and Player.

The ExternalWorld class has different subclasses, each of which works in one of the following ways:

- **RealWorld:** This subclass receives data from a defined SSL-vision and SSL-refbox, and sends robot commands to a defined sender (which forwards the commands to the desired robot).
- **SimulatedWorld:** This subclass receives data from a defined SSL-vision and SSL-refbox simulator, or any virtual vision or refbox software that adheres to the official protocols of the SSL Communication Protocol, and sends it to a simulator with its own defined protocol.
- **LogfileWorld:** This subclass receives data from a given logfile of a game. In this mode, the robots act according to the information contained in the logfile without any control. This mode is useful for observing how the player acts in various scenarios. For example, the debugger can show what path each robot would take if it were being controlled by our AI or what strategies to use when the gameplay enters a specific state defined in the logfile.

The WorldState class contains all the data describing the state of the game. This includes information about the locations of the robots and ball, as well as the current score and other game details. The ExternalWorld class feeds data to the WorldState module, which uses it to update its internal state.

The Player class is responsible for navigating the robot and generating commands for each robot. This is where the actual AI algorithms are implemented, including path planning, obstacle avoidance, and ball tracking. The Player class uses the data from the WorldState module to make decisions about how to move the robot and interact with other objects in the game.

We have also added both text-based and graphical loggers to our AI system to improve our debugging process. The text-based logger writes output directly to stdout, allowing us to more easily probe exceptions and debug our code. The graphical logger provides us with a visual representation of what the AI software "sees" and how it makes decisions based on the information it has. By utilizing these logging tools, we hope to more effectively debug and refine our AI system.

Throughout the development process, we focused on making our AI software as open and flexible as possible, to allow for easy implementation of new and innovative ideas. We also tested and debugged our software extensively, to ensure that it performs reliably and effectively in different game situations.

This year the main software also known as the AI software is redesigned in order to be more understandable and extendable. In previous years the software core was the same for each year but with slight changes according to rules and

new referee commands. The new members who wanted to implement a new idea in the software needed to spend a great amount of time to understand the methods to use for the robot navigation and data input.

The inputs for the software are vision data, referee commands and a configuration file which tells the initial parameter values for the network and match conditions. The outputs are simple navigation commands to the robots (e.g. moving with a given speed vector or kicking a ball with a specified amount of force).

The new software is written in C++ same as the previous one. The only reason to use this language is the performance and easy to extend capabilities of it. For example the ability to optimize the code by using GPUs are well known in this language.

2.1 Finite-State Machine

After a few years of experience in the Small Size League as a software designer the key idea that every team is looking to implement is to make the robots to take the correct action at the exact time and condition. If the actions are performed correctly, the robots will accomplish their task which is scoring a goal or preventing the opponent from scoring. Unfortunately, there are many conditions that can happen in a match and each one has its own set of solutions these solutions are the sequence of commands which are given to a set of robots. For the software designer it may be hard to implement the solutions with a group of *IF* conditions and no special structures.

In order to simplify the implementation for multiple software designers in the team, a Finite-State Machine implementation structure has been introduced. This gives a great flexibility and readability of implementation in the code. Each state is connected to other states by a condition or a set of conditions. This makes the implementation easier to probe. By tracking each state transition the faulty part of the code will be simply found.

Each state is basically a function which will be called whenever a complete picture of a field is received from the vision. The input of the function, or state, are the vision data. In each state, the set of commands which have to be given to the robots in the field are defined and if there is a condition which a state transition is required the next state will be defined to be called next time. Table 1 shows a sample group of commands which can be used in every state:

Table 1. Example commands which can be used in every state.

Function	Explanation
Navigate2Point(robot, destination, maxSpeed)	Navigate the robot to a destination.
ERRTNavigate2Point(robot, destination, maxSpeed)	Navigate the robot while avoiding obstacles.
Mark(robot, oppRobot)	Position between the goal and an opponent robot.
FetchBall(robot, point)	Navigate the robot to a position on line which the ball is moving on and most close to the *point*.
OneTouchDirect(robot, point, target)	Navigate the robot to a position on line which the ball is moving on and most close to the *point*. Kick the ball towards the *target*.
CircleBall(robot, radius, angle)	Position robot on a circle around the ball in a specific angle.
Face(robot, point)	Face the robot towards a point.
Chip(robot, power)	Robot should perform a chip kick whenever the ball was intercepted.
Direct(robot, power)	Robot should perform a direct kick whenever the ball was intercepted.
CircleKickBall(robot, target, power)	Position robot on a circle around the ball and towards the *target*, then kick the ball.

As shown in Table 1 the commands are fairly simple to be understood. This simple implementation saves a great amount of time for the programmers to extend or debug the code.

2.2 Debugging

Another experience with the previous AI software was the process of debugging the algorithms and, in general, the code itself. To overcome this problem, a logging protocol has been implemented which defines the messages that are sent from the AI software while operating. Using that protocol, whenever a computer is running the AI software while connected to the network, any other computer in the same network can run a logging software and monitor the AI software's parameters including not just the simple inputs from the vision, but also the state, predictions of the algorithms and et al. Fig 2 shows the graphical visualizer which monitors and records the parameters in the AI software. The visualizer is implement in python while the AI is written in C++ as mentioned before.

Fig 3 shows a simple plotter which visualizes the changes in the speed and commanded velocity of a single robot. This logging tool is also written in python.

2.3 Analyzing

With the tools and designs which were introduced above, it is now possible to demonstrate different features of this project. Below, we will describe the

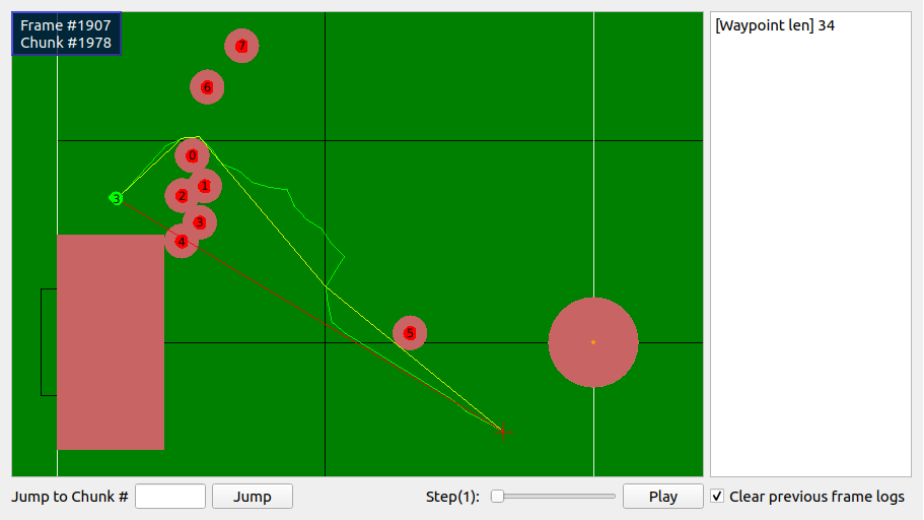

Fig. 2. A demonstration of the ERRT path plan in the graphical visualizer.

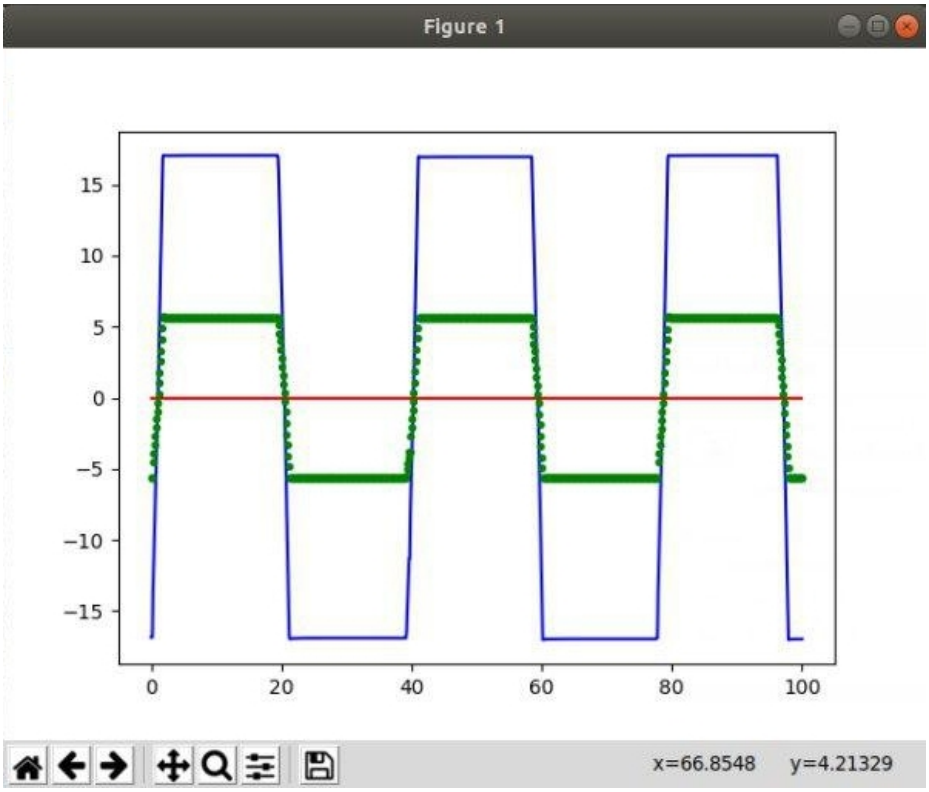

Fig. 3. An example logging program compatible with the new AI software.

analyzing process in the AI with an example. In the example the goal is to find the best spot in order for a robot to perform a one-touch kick[4].

There are many parameters to notice while performing a successful one-touch kick (e.g. Initial Ball Velocity, Robots Angle, Velocity of the robot, Target position of the kick). A simple solution to find the optimum values for the parameters is to run tests with different initializations of the parameters. Here, the tests are performed in grSim [7].

For this example at first, a ball and two robots are stationed at defined locations and a target position is randomly picked in a defined window.

Second, after waiting for a while, one of the robots moves toward the ball and kicks it towards the target position. Meanwhile, the other robot tries to reach to the target position.

Third, After the ball has been moved the second robot is commanded to perform a one touch kick and direct the ball to the center of the goal. This process continues until the ball passes the fields side line or the ball stops moving. If the ball enters the goal, the target position is tagged as a success, in other cases it is tagged as a fail. After this the *round* counter increases by one and the process is repeated from the first state.

At last, After 500 rounds the process is finished and the results are shown in the debugging tools (i.e. the graphical visualizer).

Fig 4 shows how a single round is performed.

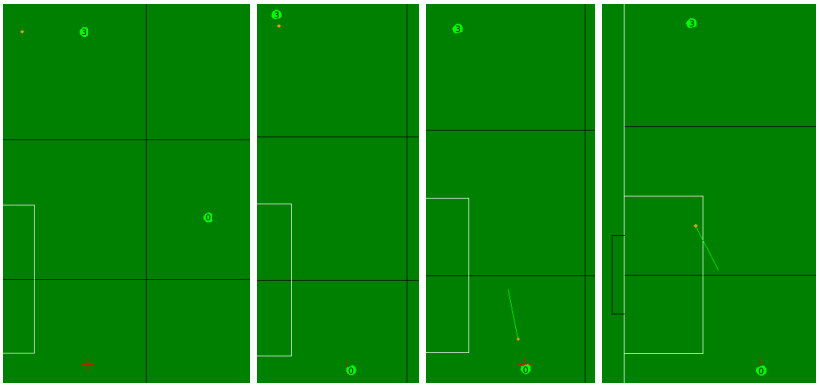

Fig. 4. The analysis process shown in the graphical visualizer. Starting from left.

To define this process the FSM chart shown in Fig 5 can be implemented in the project[5].

[4] A one-touch kick is a robots action where a ball is kicked immediately it touches the front of the robot.

[5] The chart was drawn in creately.com

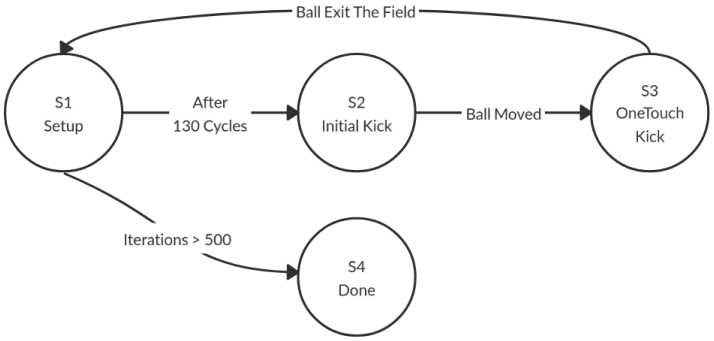

Fig. 5. The FSM chart for the analysis process.

Now that the FSM is known, each state can be implemented. The pseudocodes of the states are defined in Tables 3, 4, 5 and 6. These state functions have access to global variables which are defined in Table 2.

Table 2. Global variables of the FSM.

```
var targetPosition, initRobotPositions, initBallPosition,
  oppGoalPosition:Position;
var round, cnt:int;
var successPositions,failPositions:Position[];
var nextFunc2Run:function;
```

In Table 2, the *targetPosition* is the location where the robot will perform its one-touch kick. This variable is initialized in every round.

The *initRobotPositions* and *initBallPosition* are the initial locations of the robots and the ball in every round. These variables are defined before the start of the test. *oppGoalPosition* is the position of the center of the opponents goal line. This variable is defined before the start of the test.

successPositions and *failPositions* are two vectors which store the positions according to their tags, **fail** or **success**.

nextFunc2Run is the function which will run in the next cycle (i.e. The next time a new vision dataset is received). This variable is defined as a C++ pointer to function in our executable code.

Table 3. Implementation of the *Setup* state.

```
function S1_setup()
  placeRobots(initRobotPositions);
  placeBall(initBallPosition);
  targetPosition = pickRandomPosition();
  logData();
  if round >= 500 then
    nextFunc2Run := S4_done;
    cnt := 0;
  else if cnt >= 130 then
    nextFunc2Run := S2_initKick;
    cnt := 0;
  else
    cnt := cnt + 1;
```

In *S1_setup*(), the robots and the ball have to get placed in the specified positions. This can be done by replacing the balls by hand or by robots and at last navigating the robots towards the specified positions. If the test is being performed in a simulator (e.g. grSim) it is possible to immediately place the robot by a command. Since the current test is performed in grSim, the robots will be placed using the placement commands implemented in grSim. These commands are shown as *initRobotPositions*() and *initBallPosition*() in the pseudocode. In every cycle (i.e. every time the function is called) A variable called *cnt* is incremented by one. It is checked in an if statement to check whether it is time to transit to the next state or to stay in the current state. Another if statement checks if the round number has reached to 500, if so, the FSM will transit to the *done* state which brings the process to an end.

Table 4. Implementation of the *Initial Kick* state.

```
function S2_initKick()
  circleKickBall(Robot3, targetPosition, 100);
  ERRTNavigate2Point(Robot0, targetPosition);
  logData();
  if cnt >= 5 then
    nextFunc2Run := S3_oneTouchKick;
    cnt := 0;
  else if ballIsMoving() then
    cnt := cnt + 1;
```

In *S2_initKick*(), Robot #3 tries to aim the *targetPosition* and kick the ball towards it. Meanwhile, Robot #0 will try to reach the *targetPosition*. After a few moments when the ball is moving, the FSM will transit to the next state.

Table 5. Implementation of the *OneTouchKick* state.

```
function S3_oneTouchKick()
  halt(Robot3);
  oneTouchDirect(Robot0, targetPosition, oppGoalPosition);
  logData();
  if ballIsOut() then
    if ballInGoal() then
      successPositions.add(targetPosition);
    else
      failPositions.add(targetPosition);
    nextFunc2Run := S1_setup;
    round := round + 1;
  else if ballIsNotMoving() then
    failPositions.add(targetPosition);
    nextFunc2Run := S1_setup;
    round := round + 1;
```

In *S3_oneTouchKick*(), Robot #3 gets into a halt mode and Robot #0 will wait for the ball to reach it. Once the ball gets fetched by the robot. it will get kicked towards the *oppGoalPosition*. The state transition will not happen until the ball exits the field or stops moving. When one of the transition conditions happen it will be judged whether the test was a success or a fail according to the position of the ball in relation with the goal.

Table 6. Implementation of the *Done* state.

```
function S4_done()
  halt(Robot3);
  halt(Robot0);
  logData();
  logData(successPositions, GREEN);
  logData(failPositions, RED);
```

In *S4_done*(), the robots are halted and the results are sent to the visualizer as red and green points.

After running the code with 500 rounds in 1 hour and 20 minutes, the results were shown on the visualizer with 142 successful and 358 failed attempts. Fig 6 shows the visualizer at the end of the test. It is now clearly seen which areas have a high possibility in scoring a goal by a one-touch kick under the tested conditions.

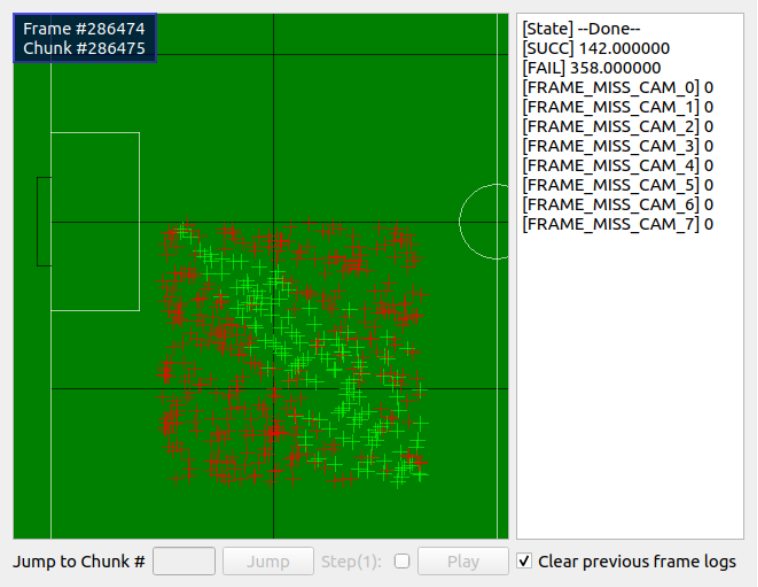

Fig. 6. The final results of the analysis.

Finally, it is worth to notice that the test has been performed in a simulation to give a better result in a short amount of time. It is clear that this test has to be made on robots in the real world. In that case, the number of rounds will obviously need to decrease to a few tens. The focus of this section was to show how an analysis procedure is taken in the Immortals AI project.

References

1. Immortals Robotics Website, http://www.immortals-robotics.com.
2. Immortals 2020 Extended Team Description Paper, https://ssl.robocup.org/wp-content/uploads/2020/03/2020_ETDP_Immortals.pdf.
3. Immortals 2019 Team Description Paper, https://ssl.robocup.org/wp-content/uploads/2019/03/2019_ETDP_Immortals.pdf.
4. Immortals 2018 Team Description Paper, https://ssl.robocup.org/wp-content/uploads/2019/01/2018_TDP_Immortals.pdf.
5. Immortals Open Source Project. https://github.com/MaGhasemieh/Immortals_ssl_opensource_mech.
6. Immortals Open Source Publish in RoboCup 2016. https://github.com/lordhippo/immortalsSSL.
7. Monajjemi, Valiallah (Mani), Ali Koochakzadeh, and Saeed Shiry Ghidary. "grSim – RoboCup Small Size Robot Soccer Simulator." In Robot Soccer World Cup, pp. 450-460. Springer Berlin Heidelberg, 2011.