Immortals 2023 Extended Team Description Paper

Ali Salehi, Mohammad Tabasi, Omid Najafi, Ali Amouzandeh, MohammadHossein Fazeli, MohammadAli Ghasemieh, MohammadReza Niknezhad, and Mustafa Talaeezadeh

http://www.immortals-robotics.com

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 Small Size League team was founded in 2008 and participated for the first time in RoboCup 2009 in Graz. The team consists of computer and electrical engineers and currently focuses on the Small Size League. There have been some changes in the mechanics and electronics of the robots in the past years. The process can be seen in the previous years TDPs and ETDPs. [3].

This year, the team focused on modernizing the electronics and software architecture. Efforts were also made to resolve issues observed during recent competitions, including RoboCup 2018, Montréal. In addition to the current robot, there is a 3D-printed prototype robot that was presented in 2018 and has been improved and tested since then. The goal is to achieve a modular, flexible, and reliable platform that would reduce the maintenance and future development costs of the robots.

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.

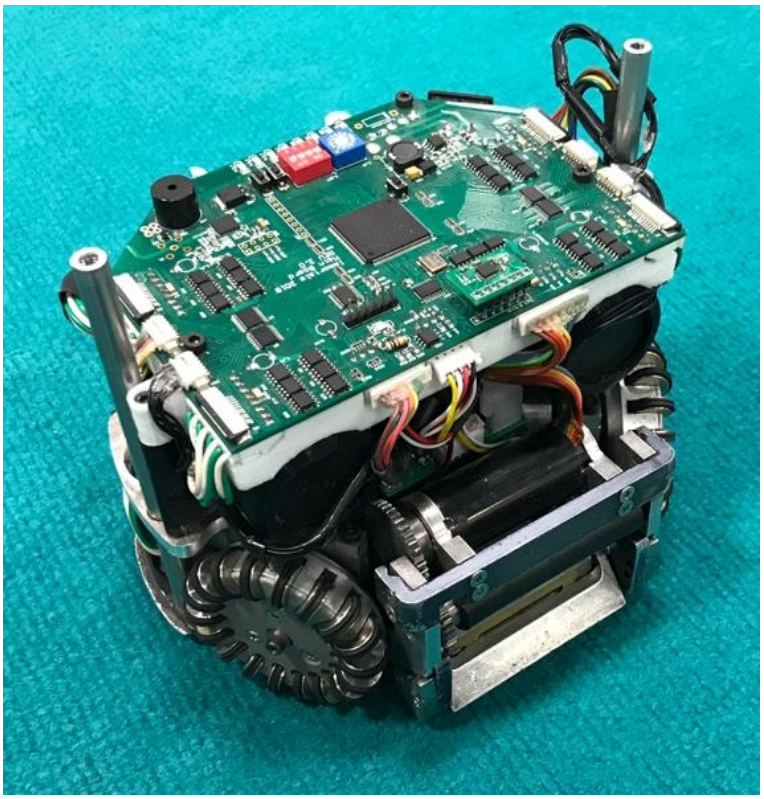

Fig. 1. Immortals current robot.

2 Kicking System

Before 2018 the limit for the ball velocity was 8m/s. In order for robots to kick a ball that much fast, a strong shooting system including capacitors and solenoids where required. Since RoboCup 2018, Montréal, Canada, the maximum speed limit has been decreased to 6.5m/s which makes teams to wonder if they want to redesign their robots kicking system and optimize the space consumption or battery power in their robot. This year the Immortals robots kicking system has been modified in order to optimize the energy consumption from the battery. This way robots have the chance to stay longer in a match without the need of their batteries to get changed.

3 Electronics

In 2018, changes were made to the electronics to modernize the designs and replace the old parts with their new counterparts. We tested them during RoboCup 2018, and the results show a solid improvement in reliability while reducing production costs. Currently, all robots use these circuits. The reader is referred to this team's previous year's TDP [1] for more details on the main circuit.

This year, we're planning to redesign all of our electronics from scratch to reflect the latest developments in the league and also in the industry. The main goals are (1) reliability, (2) expandability, and (3) being more competitive. It should be mentioned that at the time of this writing, the design work is still ongoing and we don't have the final boards manufactured. We hope to equip at least half of our robots with the new parts in RoboCup 2023. We will publish the designs on our GitHub page shortly after the competition.

3.1 Main board

The current main board used by the team was designed in 2010 and, apart from minor changes, has been used in its original form ever since. It uses a Xilinx Spartan3 FPGA as its main and only processor. A soft processor (TSK3000) is used inside the FPGA to handle more sequential logic, while the FPGA itself is used to read encoders, drive BLDC motors, drive the boost converter, etc. While this design is flexible and comparatively cheap to produce, it has shown its age in recent years. The main drawbacks are:

1. The TSK3000 runs at around 36 MHz and in its current configuration is way too limited to develop any more sophisticated local processing and motion planning. Some efforts were done in the past to move parts of performance-critical C code to the logic gates, but it would make the implementation harder to change and extend. On the other hand the debugging workflow was too limiting, and any changes to the code needed a full rebuild of the FPGA project. All these factors resulted in the team using pretty much the same framework for several years, without being able to make major changes.

2. The BLDC motor commutation is a simple 6-step trapezoidal commutation. It is easy to implement, but is inefficient and causes a high amount of torque ripple. Implementing a more sophisticated method, e.g. Field Oriented Control, requires massive changes to the PCB.
3. We use nRF24L01 chip for wireless communication, with a custom payload layout on top of its Enhanced ShockBurst (ESB) protocol. This gives us a lot of flexibility, but being a low-level protocol means we need to add any higher-level feature needed, e.g. discovery.
4. The waveform needed to drive the boost converter in the kicking board is generated by the FPGA. This meant that we were able to freely change it to match our needs, but in practice deemed too fragile.
5. There are no current protections in the board. Any malfunction in the board itself or in another part of the robot, e.g. a stuck wheel, causes damage to the parts. This hugely reduces reliability and increases maintenance costs, and caused damage to the battery.

To resolve these issues the work started on designing a new main board from scratch. The main features are:

1. Raspberry Pi Compute Module 4 as the local compute unit on robot. We intend to move parts of the skill execution, data fusion and prediction, and motion planning to it.
2. ESP32-C series as the wireless link using WiFi. We're currently using the ESP32-C2 and aim to upgrade to the recently released ESP32-C6 in the future which has WiFi 6 support. This will greatly enhance the bandwidth and the feature-set of the link and will allow us to add robots as regular links to our software stack. This way they will receive world state and the AI output necessary to execute local skills.
   It is worth mentioning that the latency characteristics of using WiFi instead of a low-level protocol is yet to be determined. The latency requirements after moving more processing to the robot's local processor will be more relaxed. But, we are considering adding a separate nRF chip to handle latency-sensitive if the new approach causes issues.
3. CAN protocol will be used to connect the main board to other boards, including the kicking board. This will give us a more robust and flexible base that we can build on top.
4. Dedicated BLDC driving IC, TMC4671. It implements Field Oriented Control (FOC) for BLDC motors, and includes various closed-loop control methods. This offloads the local motor control functionality from the main MCU to the dedicated HW, which is more reliable in terms of latency.

3.2 Kicking board

In the previous years, we used a boost converter that was drived by the FPGA from the main board. There were also two discharge and a charged state pin connected directly to FPGA's IO pins. The were major issues with this design

both in terms of reliability and charging performance.
This year we redesigned the kicking board with the following features:

1. A dedicated flyback capacitor charging IC, LT3570, is used. This simplifies the design and improves performance and reliability at the same time. We use DA2034 transformer and the BSC109N10NS3G MOSFET.
2. A STM32F103 MCU is used on the board to handle CAN protocol to the main board, and controls the charging IC, variable resistors, and discharge IGBTs.
3. A high-power resistor network composed of three 2.4K 3W resistors is added to the board to discharge the capacitors when needed, without using the kicker magnets. The FQD5N60C MOSFET drived by a ZXGD3009E6 is used to control the discharge.
4. Two IGB50N60T IGBTs drived by a single UCC27624DR are used to discharge the capacitors to the kicking magnets.

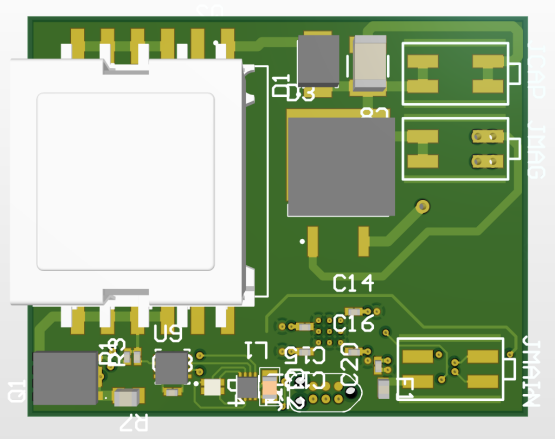

Fig. 2. The new kicking board design

4 Software

4.1 Architecture

Our current software stack is was developed in 2009 in C++ and has seen several additions and improvements throughout the years. This has resulted in a performant and robust software, and at the same time hard to maintain codebase that is too fragile to the new changes.

It is also a single application that handles world state estimation, AI, and motion planning of the robots. This has the added benifit of being able to change

the data flow between various parts quite easily. But forces us to implement everything in C++ in to produce one single application that runs on one machine.

The goal this year is to refactor the codebase into separate parts that are connected via nanomsg. This allows us to move the lower-level motion planning and skill execution to the robots' local processor, and develop GUIs using other technologies.

At the time of writing this paper, this effort is still ongoing, but we are confident that we can make the transition to the new stack in time for RoboCup 2023.

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.

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.

4.2 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.

4.3 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 3 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 4 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.

4.4 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

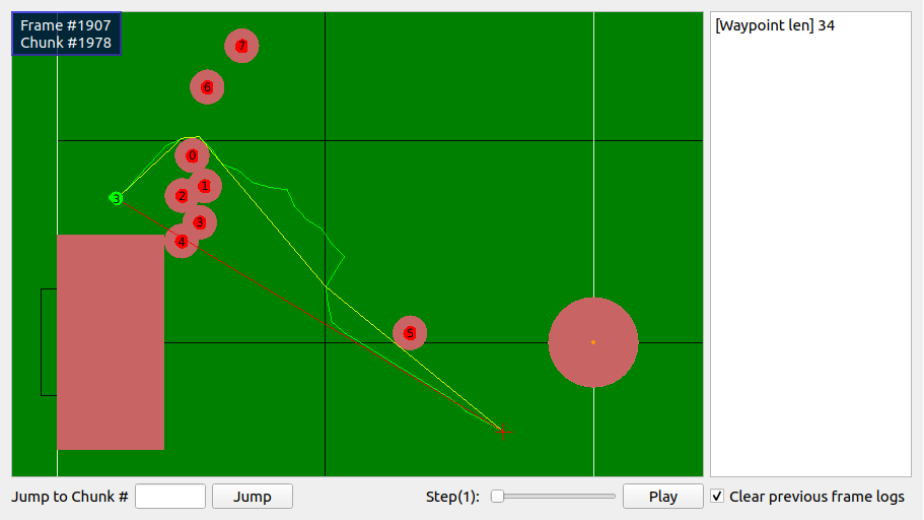

Fig. 3. A demonstration of the ERRT path plan in the graphical visualizer.

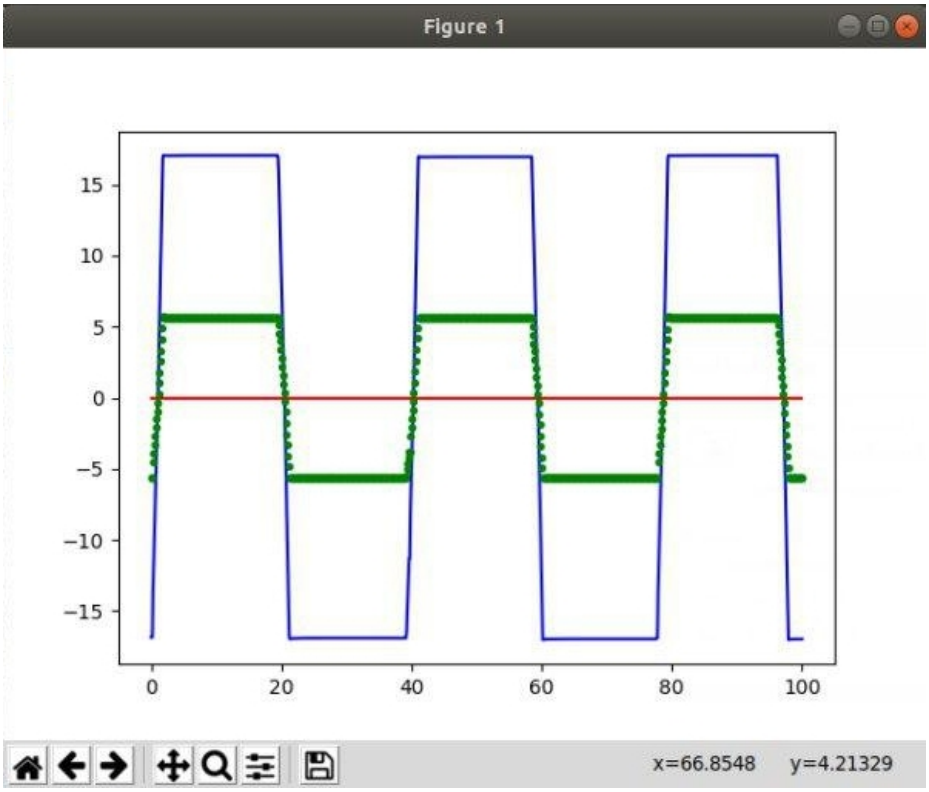

Fig. 4. An example logging program compatible with the new AI software.

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[1].

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 5 shows how a single round is performed.

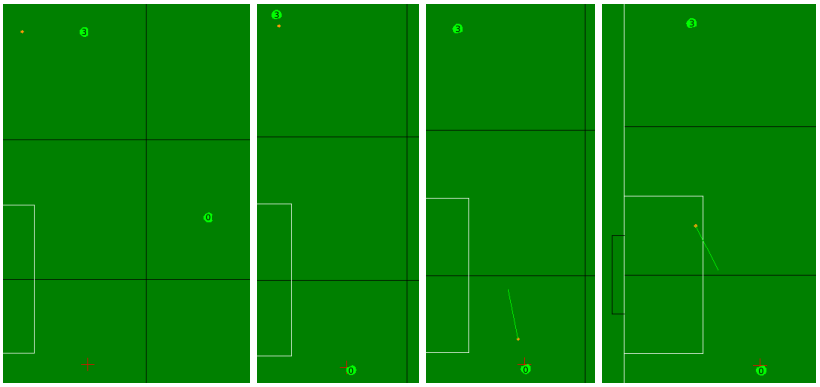

Fig. 5. The analysis process shown in the graphical visualizer. Starting from left.

To define this process the FSM chart shown in Fig 6 can be implemented in the project[2].

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

[2] The chart was drawn in creately.com

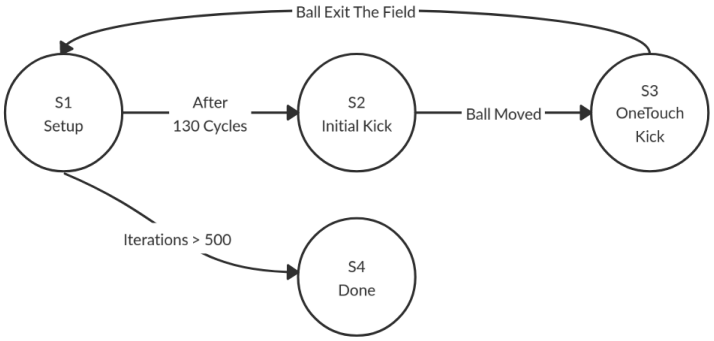

Fig. 6. The FSM chart for the analysis process.

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 7 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.

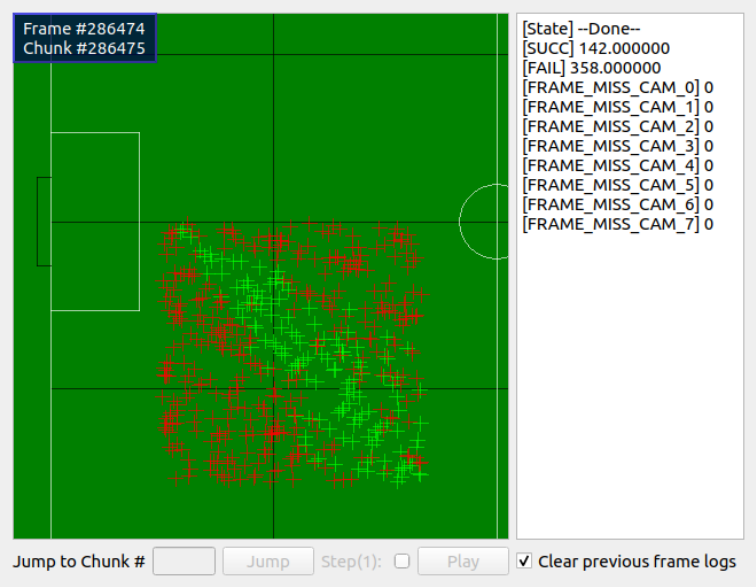

Fig. 7. The final results of the analysis.

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.