Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Quantifying Player Performance in Simulated Humanoid Robot Soccer Games

vorgelegt von

Jan Gutsche

MIN-Fakultät

Fachbereich Informatik

Studiengang: Bachelor Informatik

Matrikelnummer: 7061491

Abgabedatum: 19.05.2023

Erstgutachter: Prof. Dr. Jianwei Zhang

Zweitgutachter: M. Sc. Niklas Fiedler

Betreuer: M.Sc. Jasper Güldenstein

Acknowledgments

I would like to thank my advisors for helping me if I had any questions and for providing valuable feedback. Similarly, I want to thank the Hamburg Bit-Bots members, in particular Timon Engelke, Florian Vahl, and Jörn Griepenburg for the help debugging hard problems and providing a lot of infrastructure that I used to simulate matches. Special thanks to the RoboCup humanoid league technical committee for the trust and commitment to run the data collection software alongside the official Humanoid League Virtual Season 2023 games and for providing me with the resulting data.

Abstract

This thesis proposes innovative methods for performance analysis of virtual humanoid soccer robots, specifically in the virtual RoboCup humanoid soccer domain. The objective is to extract meaningful and statistically significant metrics about team and player performance from simulated games, enabling improved comparability. A data structure is designed to collect data from official Humanoid League Virtual Season (HLVS) games. Data analysis is based on the works of Pereira et al. [1] with their modular SoccerAnalyzer framework. Recorded game data can be processed to calculate various metrics. Some proposed metrics are player speed, ball possession, kick distance, and self-localization accuracy. Informative visualizations can be created from those. The obtained metrics could also be utilized for automated testing of robot software.

Zusammenfassung

Diese Arbeit schlägt innovative Methoden zur Leistungsanalyse von virtuellen, humanoiden Fußballrobotern vor, insbesondere im Bereich des virtuellen, humanoiden Fußballs im RoboCup. Das Ziel ist es, aussagekräftige und statistisch signifikante Metriken über die Leistung von Teams und Spielern aus simulierten Spielen zu extrahieren, um eine verbesserte Vergleichbarkeit zu ermöglichen. Es wird eine Datenstruktur entworfen, um Daten von offiziellen Spielen der virtuellen, humanoiden RoboCup Liga zu sammeln. Die Datenanalyse basiert auf den Arbeiten von Pereira et al. [1] mit ihrer modularen SoccerAnalyzer-Software. Aufgezeichnete Spieldaten können verarbeitet werden, um verschiedene Metriken zu berechnen. Vorgeschlagen werden z.B. Spielergeschwindigkeit, Ballbesitz, Schussweite und Genauigkeit der Selbstlokalisierung. Hiervon können informative Graphen erstellt werden. Die ermittelten Metriken könnten auch für automatisierte Tests der Roboter-Software genutzt werden.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AWS Amazon Web Services. 9, 25

DOF degrees of freedom. 22, 24

GPU graphics processing units. 9

GUI graphical user interface. ix, 7, 8

HL RoboCup Humanoid League. x, 3, 4, 6, 7, 9, 15, 25, 26, 31

HLVS Humanoid League Virtual Season. v, ix, 1, 3–9, 13, 20, 21, 24–26, 33, 54, 55

ODE Open Dynamics Engine. 4, 5

Protobuf Protocol Buffers. 8, 9, 25

SIM2D RoboCup 2D Soccer Simulation League. 26

SSL RoboCup Small Size League. 26

1. Introduction

In this thesis, novel methods for analyzing the performance of virtual humanoid soccer robots will be developed and evaluated for usage in the virtual RoboCup humanoid soccer domain. The goal of this development is, to enable quantifiable analytics of virtual robot soccer games with statistical significance or, in other words, to be able to extract meaningful team- and player-performance data from these simulated games to achieve better comparability. Recent advancements in the realm of robot soccer simulation environments are enabling these new possibilities (see section 2.2).

Soccer analytics, as an application of data science and data mining, that has been utilized for several years in the field of professional human soccer games. Products from various commercial vendors and community-driven tools are being used to improve a team's decision-making and to better the insight as well as understanding for the public (see section 3.1). Another application of soccer analytics is to provide predictions for the sports betting industry. To the best knowledge of the author, only recently, some methods of soccer analytics were introduced to the humanoid robot soccer domain. Nevertheless, some approaches from the human soccer domain could be transferred to our domain.

The approach of this work includes the development of a data structure designed to collect data from official Humanoid League Virtual Season (HLVS) matches. Therefore, the setup for generating those games has been extended. The recorded files will be processed, to calculate various game metrics, which later can be used to create meaningful visualizations. Such a set of metrics could include average speed of players, ball possession, maximum kick distance and accuracy of self-localization. Assuming, reliable metrics could be extracted from simulated game recordings, this data could further be used for manual or automated tests, by running simulated games and quantitatively comparing new player strategies and algorithms to previous versions. This could even be included in the workflow of a *Continuous Integration* (CI) pipeline. Another conceivable application of the resulting data is in the development of machine learning and, especially, reinforcement learning models.

1.1. Research Questions

This thesis tries to answer the following research questions:

- Which metrics are meaningful in humanoid robot soccer?
- Do these metrics have the same predictive capability as in human soccer?

1.2. Structural Outline

Knowledge that is fundamental for reading this thesis will be given in chapter 2. Afterwards, related work in chapter 3 gives an overview of sports analysis and data analysis in the real of RoboCup. The approach of this thesis will be presented in chapter 4. Results of this approach are then evaluated in chapter 5. A discussion about the results and achievements of this thesis follows in chapter 6. Finally, this thesis will be concluded in chapter 7 and future work can be found in chapter 8.

2. Fundamentals

This chapter presents and describes topics and software that is fundamental for the understanding of this thesis.

2.1. Data analytics

Data analytics is a process with the goal of gaining insights or knowledge from data. This process often involves multiple steps which can include defining requirements for data, data collection, processing, cleaning, analyzing, and visualizing.

2.1.1. Pandas

Pandas is a widely used open-source library for the Python programming language for data analysis. [2, 3]. The library provides two flexible data structures, a `Series` for 1-dimensional and a `DataFrame` for 2-dimensional data. Additionally, Pandas provides many functions for filtering, cleaning, reshaping, and aggregating diverse datasets. Integrations with other libraries for handling data in Python such as NumPy, Matplotlib, and Seaborn simplifies workflows. Finally, Pandas can import data from various formats and export to many as well, including CSV, HDF5, Microsoft Excel, and SQL.

2.2. Humanoid League Virtual Season

The HLVS is an annual remote competition organized and carried out by the RoboCup Humanoid League (HL) since 2021. This event originates from the remote RoboCup 2021 (worldwide) competition that took place during the SARS-CoV-2 pandemic. Due to travel restrictions, in-person robot matches (supported by human team members) were not possible. This triggered the development of a common virtual simulation environment for humanoid robot soccer games.

The HLVS as a virtual competition is very much less expensive and less effort to both participate and conduct, since no travel expenses and no robot-hardware problems occur. Due to these advantages, it and the standalone simulation environment are particularly suited for testing the robot's high-level software.

Virtual matches get simulated biweekly, giving participating teams time to prepare for the next match and to improve their robot software. This competition format begins with a round-robin phase, where each team plays against each other once and collects points, similar to soccer tournaments. Afterward, in a series of final games, the winner, and subsequent ranking gets selected.

Figure 2.1.: Screenshot taken from recording of the second match on game day 3 (K-GD3-2) from the HLVS 2022/23. UTRA Robosoccer plays in blue, Hamburg Bit-Bots in red.

Similarly to the real RoboCup Humanoid League (HL), a set of rules exists, that describes criteria for the game of play, constraints on the player's robot design and other game objects, and allowed and disallowed player behavior [4].

2.2.1. HLVS 2022/23

The latest season is the HLVS 2022/23 originally planned to begin in December 2022, but was delayed to start on the 19th of March 2023 with its final game day on the 30th of April 2023. The results, live-stream commentary and recordings can be found on the official website[1]. Figure 2.1 taken from this season shows the simulated environment including the field and multiple robots.

2.2.2. Webots

The simulation of the HLVS virtual matches gets performed using the open-source robot simulator Webots. Webots is developed and maintained by the Swiss company Cyberbotics Ltd. [5]. It uses a fork of the physics engine Open Dynamics Engine (ODE) [6] providing rigid-body dynamics and collision detection, is compatible with multiple platforms, and provides

[1] https://humanoid.robocup.org/hlvs2023/schedule-for-the-humanoid-league-virtual-season-2022-23/

interfaces to interact with the simulation in many programming languages such as C++, Java and Python.

Running a simulation in Webots is not continuous, instead the simulation is performed in discrete time steps, typically on the millisecond scale. The simulator always progresses the environment by this basic time step. HLVS games are simulated using a basic time step of 8 ms, meaning every change of the simulated environment spans a multiple of 8 ms of in-simulation time.

The real-time factor describes how fast a running simulation progresses compared to the real-time. A real-time factor of 1.0 means that the time inside the simulation progresses synchronously with the real time. Factors larger than 1.0 mean that the simulation time progresses faster than real time, for example, a real-time factor of 2.0 means two seconds of simulation happen during one real second. Analogous, a real-time factor of less than 1.0 means the time inside the simulation runs slower than real time. The real-time factor for calculating a simulation using Webots depends on the complexity of the environment. Collisions of objects and robots with each other have a large impact on the runtime performance. Other factors are the robot control software, and the available hardware.

The simulation environment consists out of a world, usually defined by a world-file. This world defines basic simulation properties as the basic time step, gravity, and friction between different materials. The world can then be augmented by `Nodes`, which are defined in `.proto`-files. Syntactically, those files are based on the VRML97 standard, as defined in [7, 8], but since then, the PROTO definition has been enhanced. PROTO definitions can also contain scripting languages such as Lua or JavaScript, drastically increasing the possibilities. A `Node` object can represent light sources, geometry such as solid objects and devices such as actuators and sensors [9].

This 3D physics-based simulation, however, can only be considered an approximation of the real world. This is due to multiple reasons: First, typically, objects from the real world cannot be modeled perfectly, as there are always some imperfections and inaccuracies in the transfer to 3D geometry, for example considering mass distributions and surface properties like friction. Secondly, there is a trade-off in the integration process (calculating new rigid body states after a given time step) between accuracy and stability. In this context, accuracy means, how well does the simulated behavior matches the real world. Stability is concerned with non-physical behavior, such as sudden explosions of the environment for no particular and reality-based reason. ODE has chosen to be very stable, in a trade for less accuracy. Still, the accuracy can be improved by simulating or integrating in smaller time steps, but this, on the other hand, increases computing times [10].

Simulation inaccuracies and simplifications limit the possibilities of knowledge- and model-transfer from the simulation environment to the real world. This reality gap effect also appears in the official simulation environment of the HLVS, for example, algorithms for stand-up motions for humanoid robots that have been tuned using this environment do not perform perfectly similarly in reality [11].

The HLVS competition makes use of Webots' web animation feature. During simulation, 3D recordings can be created, which can be interactively replayed using HTML and other web technologies. This enables simple and user-friendly presentations to spectators in real-time

after the long-running simulation has completed.

2.2.3. Automatic Referee

Relying on the judgment of human referees for HLVS and unofficial test games is not feasible because of the time expense, scheduling issues, and reduced reproducibility. Therefore, an automatic refereeing software has been developed for the RoboCup 2021 Worldwide as part of the HL simulation setup. This automatic referee is written in the programming language Python and acts as a supervisor controller for the Webots simulator. As a supervisor, it can query the states of all simulated objects and control and manipulate the simulation. This happens in an alternating way in a loop, where the supervisor controller runs and then gives control to the Webots simulator to progresses the environment by an amount of time set by the supervisor. Thereby, the supervisor controller is capable of limiting the real-time factor of the simulation. A required feature of the automatic referee is to guarantee a maximum real-time factor of 1.0, in order to allow the player's robot controllers enough time to process and act. This is done by delaying the next stop of the simulator by a respective amount of time, in case the automatic referee finished its control loop early.

The automatic referee has a diverse set of configurable options, to influence its behavior. These options are split into two configuration files. The first one is called the `referee_config.yaml`, which contains constants and threshold values which mostly depend on the rule set; this file does not need to get changed between different games. The second file is the `game.json` which is used to define game setups. It includes parameters that define the type of match (e.g., normal or knockout), which teams participate in which lineup, host IP addresses and others, that are susceptible to change. This file also contains options to enable or disable other components of the HLVS setup. For official games, however, the following components are enabled: the UPD-bouncer (see section 2.2.7), web-animation recording (see the previous section 2.2.2), and data collection methods from this thesis (see section 4.2).

The runtime of the referee software can be thought of as three distinct phases of the HLVS simulation process. These phases can be summarized as follows: The automatic referee software starts with the initial setup phase, which loads the configuration, starts necessary sub processes for the game-controller (see next section 2.2.5) and UDP-bouncer, randomizes and loads the simulation world and loads the robots, starts a recording, and starts the game simulation with the initial state; all according to the configuration. The second phase is running the main loop, which gets iterated for each simulation step until the game is over. This loop progresses the Webots simulation, measures the latest players states, calculates decisions according to the rules, and then enforces them by, e.g., placing the players to the sides and communicating penalties to the game controller. The third and final phase runs after the game has finished or was somehow interrupted before the end. This determines the winning team of the game, and afterwards finalizes all the network sockets and sub processes, ends the recording and closes the Webots simulator.

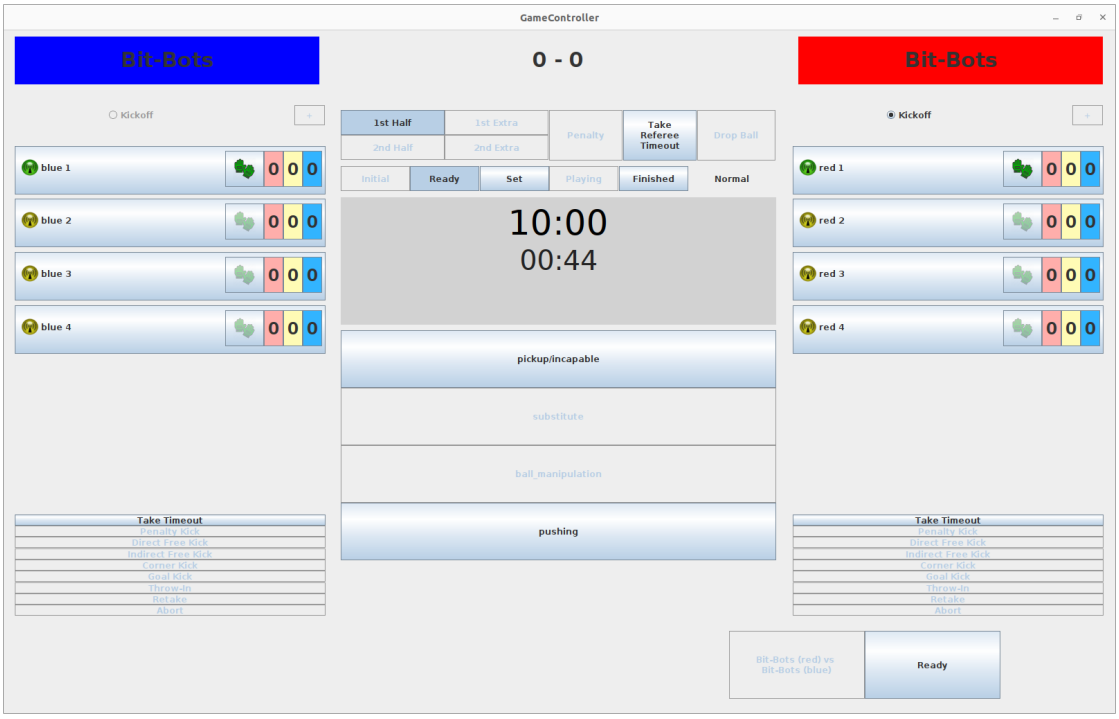

Figure 2.2.: Screenshot taken from the game controller's GUI at the beginning of a test game. The left and right columns display the current state of and controls over the blue and red team, respectively (in this test game, both teams are called `Bit-Bots`). The middle column shows the current score, game state, clock and controls.

2.2.4. Test Suite

In conjunction with the automatic referee, a comprehensive test suite has been implemented to verify and track its correctness. Many tests exist to check the behavior of the referee in standard situations. Each test starts a separate game setup, including a Webots environment and referee. Then simulation objects are manipulated by the test to generate a desired game situation, afterwards checks verify that the referee acted as expected.

2.2.5. Game Controller

The game controller is a component of the official HLVS simulation setup. However, originally, it was developed to be operated and monitored by human referees during HL games. Figure 2.2 exemplarily shows the game controller's graphical user interface (GUI), as it would be used by a human referee.

This software is intended to track the current state of a match and communicate this to the players. This state involves the main clock, timers for penalties, and standard situations, it tracks the goal score, and player's warnings, and cards. Communicating this state to the players usually happens over UDP broadcast messages over the local wireless network. Similarly,

players frequently send hearth-beat messages back to the game controller. The protocol that defines this communication is not based on Protocol Buffers (Protobuf) or similar tools, but instead uses raw bytes[2].

For HLVS games, some changes have been made to reuse this software. The most significant change is, that the game controller is not operated through its GUI, but rather remotely through an extension protocol. This protocol is text-based and enables communication between the automatic referee and game controller software. The automatic referee sends simulation time updates and decisions like penalties, cards and standard situations. The game controller acknowledges this and returns the current tracked state, this includes updates, like passing of player's timeouts. Another change is, that the communication with the players does not happen over UPD-broadcasts directly, rather both parties exchange UPD packets via a proxy, which is the UPD-bouncer (see section 2.2.7).

2.2.6. Robot Controllers

Robot controllers are another component of the HLVS. During the setup phase, the automatic referee spawns one robot controller process per player to interface between Webots and a team's robot software on a different host machine. To bridge these components, a Protobuf based protocol specified by the technical committee is used[3]. The robot controller software queries the latest sensor measurements from Webots after simulation steps with a frequency, that is defined by the teams. Allowed sensors are accelerometers, touch sensors, cameras, force sensors, gyroscopes, and joint position sensors. These measurements together with the current simulation time are then converted to Protobuf messages to be forwarded directly to the team's software using a TCP connection. The robot controller also listens for new actuator commands coming from the team players to set them in the Webots simulator.

2.2.7. UDP-Bouncer

The UPD-bouncer is also part of the HLVS setup. It is a proxy for UDP network packets for the game controller and team-communication, distinguished by different ports. This software consumes the `game.json` configuration file similarly to the automatic referee, and thus knows the IP-addresses and network ports of the simulator host machine and the hosts for both team's player's software. In the case of packets coming from the game controller, those get forwarded to all players. When receiving team-communication packets, the bouncer uses the sender IP-address to determine the sender's team and then forwards these packets to all remaining team players. To verify that the software is working correctly, it writes a log of its configuration and all team-communication packets to a file, which can be inspected by a human.

[2] `https://github.com/RoboCup-Humanoid-TC/GameController/wiki`
[3] `https://cdn.robocup.org/hl/wp/2021/06/v-hsc_simulator_api_v1.01.pdf`

2.2.8. Infrastructure

The official HLVS infrastructure is hosted on remotely accessed Ubuntu virtual machines by Amazon Web Services (AWS). Those machines, also called instances, are all running simultaneously in the same location. The hardware specifications for the servers are defined by the technical committee[4]. The instance running the Webots simulator, automatic referee, UDP-bouncer, and robot controllers is of type `g4dn.4xlarge`. The up to eight machines hosting team player software are specified as `4dn.xlarge`. Both types have similarly powerful graphics processing units (GPU) hardware, but the instance running the simulation has more general processing power accompanied by more system memory. Teams provide their player software in the form of Docker images, which are then pulled and started on the player instances.

2.2.9. Team-Communication

Team-communication as specified in the HL ruleset is a method that allows the team's robot software to communicate internally during a game using UDP broadcasts over the local wireless network [4]. For virtual games in the HLVS, the direct communication is not possible, this is why the UDP-bouncer (see section 2.2.7) is used to forward this. This communication should enable coordinated behavior, but is bandwidth-limited in order to not overwhelm the network. Standard protocol definitions (mitecom and Protobuf-based) are recommended by the technical committee to increase compatibility between teams. However, team-specific extensions are possible and using the standard is completely optional.

[4] `https://humanoid.robocup.org/wp-content/uploads/v-hsc_server_specification_v1.03.pdf`

3. Related Work

During the last decades, much research and development has gone into data mining and data visualization methods for human sports. In section 3.1, we will give an overview of sport analytics and specifically soccer analytics. Afterward, we will continue this overview with recent research from the RoboCup domain in section 3.2. Moreover, the SoccerAnalyzer framework by Pereira et al. [1] will be presented in detail (see section 3.2, as our approach will be based on their work.

3.1. Data Analysis in Sports and Soccer

The origins of sports analytics began in the 1950s, when Reep et al. in *Skill and Chance in Association Football* [12] analyzed passes in soccer, that resulted in goals. In the 1960, notes and recordings of basketball, American football and later on baseball sports were investigated manually. Following technological developments in the 1980s, computers were involved for data-gathering as well as its analysis [13].

Nowadays, sports analytics is a well established process and impacts sports in many aspects. For example, individual performance metrics help sport scouts select new talents; during games, real-time information gets evaluated, which a coach can use to influence a team's game play [13].

Sports analytics and therefore soccer analytics relies on the availability of data to extract insight from. Some datasets for soccer games are publicly available [14], are open for academic research [15, 16], or are provided by businesses as sample data[1,2]. Additionally, real-time data and analysis from professional sports games is offered to various news outlets and betting providers.

Data contained in those datasets often consists of a video (and sometimes also an audio) recording, that can be augmented with data from wearable sensors and, recently, sensors integrated into the soccer ball itself. FIFA, together with research institutions, introduced an *Inertial Measurement Unit* (IMU) sensor running at 500Hz inside the soccer ball and a camera system (at 50Hz) tracking all limbs of the players (in total 29 positions per player) to detect offside situations and support the referees in their decision. Additionally, this system provides detailed 3D reconstructions of the scene for spectators and viewers [17]. In [15], Pettersen et al. used a lightweight, wearable sensor unit with radio transmission position tracking, an accelerometer, a gyroscope, a heart-rate sensor and a compass. With a rate of 20Hz, the system returns, among other data points, the player's heading, direction and speed.

[1] `https://github.com/statsbomb/open-data`
[2] `https://github.com/metrica-sports/sample-data`

Kern et al. utilized a wearable sensor to gather data for a soccer headers detection neural network [16].

3.2. Data Analysis in RoboCup

Recently, multiple RoboCup soccer leagues did make efforts gathering data which can be used for analyzing and extracting interesting metrics. 2019, Mellmann et al. from the *Standard Platform League* (SPL) presented tools for collection and synchronization of video, communication, and robot's log data in [18]. Their work also included a web platform[3] for viewing the game recordings augmented with additional time synchronized events of the individual robots (e.g., the falling of a robot).

In the SPL rule set for 2022 in [19], an open research challenge regarding video analysis and statistics was formulated. The subsequent dataset of robot and ball annotations of 35000 images from the RoboCup 2019 is publicly available.[4] Currently, only the B-Human team has released its source code in [20], which uses a YOLOv5[5] neural network to detect robots and balls in the images. Using intrinsic and extrinsic camera calibration, the detections are then mapped to field coordinates, enabling metrics such as distance walked by players, and ball possession. Additionally, visualizations of the area controlled by players on the field or a heat map of ball positions on the field can be created.

Other RoboCup soccer leagues already provide a long-lasting archive of game logs and recordings: Small Size League (SSL) [21], Simulation League 2D (SIM2D) [22], and Simulation League 3D (SIM3D) [23]. Those game logs include among other information positions of each player and other entities, game states, player states and scores. A recent approach of analyzing data from two of those leagues will be presented in the following section 3.2.

SoccerAnalyzer

The SoccerAnalyzer is an open-source Python library for data analysis for game logs from the SIM2D and SSL, written by Pereira et al. [1]. The authors propose a modular software architecture which serves as a common framework to be extended for analysis in all soccer leagues in the RoboCup domain.

A simple interface for Jupyter Notebooks was provided by the authors, whereas comprehensive plans for building a user interface based on web technologies are planned. Parts of those plans have already been implemented in the related project WebSoccerMonitor[6].

[3] https://www2.informatik.hu-berlin.de/~naoth/videolabeling/index.php
[4] https://github.com/RoboCup-SPL/Datasets
[5] https://github.com/ultralytics/yolov5
[6] https://github.com/robocin/WebSoccerMonitor

4. Approach

The approach of this thesis concerns several parts that will be described in the next sections. To analyze data from humanoid robot soccer matches, the data needs to be collected and stored in a concise and structural manner. As described in section 2.2, running and simulating a HLVS game includes many interrelated software systems. Each of those components processes and holds different important information, of which a subset is of interest for this thesis purposes. In section 4.1, a data structure to hold and manage this data will be proposed. The utilization of this structure for data collection during the simulation of the matches is described in section 4.2. Finally, data analysis is the topic of section 4.3.

4.1. Data Structure

A concise and efficient data structure is needed to enable correct, fast, and maintainable data collection and analysis. To achieve this, a Python software module, called `data_collection` has been developed, which makes heavy use of the `dataclass` feature provided by the standard library. Although this software module could be used independently of the HLVS setup, it is currently contained in a fork of the technical committee's `hlvs_webots` software repository[1]. Additional internal information of a player's robot-control software, such as cognition of the robot's environment or actions intended by the behavior are of interest for analyzing some metrics. Due to time constraints during this thesis, this data has not yet been represented in the data structure.

4.1.1. Components

The data structure consists out of three main parts: firstly, high-level classes `DataCollector` and `Match` that manage and store the following dataclasses, secondly the `StaticMatchInfo` dataclass, and lastly the dataclass `Step` for dynamic match information. Those components will be described in the following paragraphs.

High-level Data

The high-level classes present a simple and clean interface for later data collection usage, they hold references to the dataclasses during the simulation phase of the game, and are responsible for safely storing the information to the file system. The top-level class is called `DataCollector`, which gets initialized by the user with some minimal configuration and a `Match` object. Once the `DataCollector` is created, it will automatically save the given match

[1] `https://github.com/jaagut/hlvs_webots/tree/main/controllers/referee/data_collection`

to the file system in repeated intervals as defined by the `autosave_interval`. By calling `finalize()` on the object, it will save the match data and close gracefully. In case of fatal errors during the simulation process, when the `DataCollector` object gets deleted, it will also try to save the latest match information. All of this storage logic has been implemented to reduce the damage by data loss of the match information. The `Match` object, that has been given to the `DataCollector`, implements a generalized `save(...)` method, which is utilized by the storage logic from above. Although, the main purpose of the `Match` class is to hold references to a `StaticMatchInfo` object and a history of `Step` objects. Those will be described next. For a more detailed overview of these high-level classes, see the UML class diagram in figure 4.1.

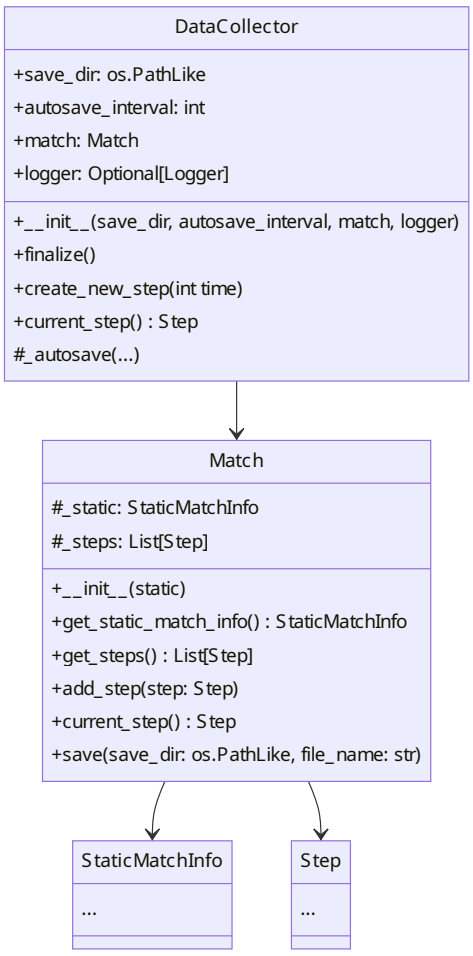

Figure 4.1.: UML class diagram of the high-level classes `DataCollector` and `Match`. `StaticMatchInfo` and `Step` have been simplified, as the details will be shown later.

Static Data

The `StaticMatchInfo` class is designed to hold and organize metadata information, which does not change during a match. This includes information such as the match type (normal, knock-out, round-robin, or drop-in), the league of the match (HL kid- or adult-size), simulation settings, and participating teams. This metadata is held in the form of an object tree, where each object is defined by its own dataclass. For example, `StaticMatchInfo` has a reference to a `StaticTeams` objects, which in turn holds two references to `StaticTeam` objects; and so forth. An overview of this data structure can be seen in figure 4.2.

Variable Data

Finally, the `Step` class has been designed to organize dynamic information during the simulation of a match. This information includes the poses of all moving objects, e.g., the ball and team-players, and additionally decisions made by the automatic referee and game controller software. Similar to the `StaticMatchInfo` class, `Step` organizes information on a higher level by keeping references to more specific dataclasses in a tree hierarchy. As previously described in section 2.2, the simulation runs in discrete time steps. This way, all relevant match information can be measured and collected after each simulation step. In practice, the user of this `data_collection` module creates and fills out a new `Step` object after each simulation step, to append it to the `Match` object's list of previous `Step` objects. Thereby, the list grows during simulation-time and contains dynamic information about the match for each measured time step. In figure 4.3, an UML class diagram detailing the `Step` class can be found.

The complete UML class diagram showing all three parts of the `data_collection` module can be found in figure 1 of the appendix.

Features

The usage of the `dataclass` feature from the Python standard library [24] for most components was very deliberate, as this simplifies the class definition. For example, defining the class `StaticMatchInfo` only involves adding a decorator (`@dataclass`) and listing the class field names (optionally with types). This can be observed in the source code listing 1. Using this decorator, an initializer method, getter- and setter-functions for all fields and string-representation are generated. The decorator also takes optional arguments. In the example, the optional `frozen` argument has been set. This guarantees, that fields of the respective class cannot be overwritten after being initialized once. Wherever feasible, this feature is used in the `data_collection` module to prevent accidental modifications. Additionally, the `dataclasses-json` library[2], which introduces a `DataClassJsonMixin`, has been used. All previously mentioned dataclasses inherit from this mixin, which automatically adds `to_json()` and `to_dict()` functions. With every dataclass in the hierarchy tree having this feature, it is possible to easily convert the whole tree recursively to both Python dictionary objects and

[2] `https://github.com/lidatong/dataclasses-json`

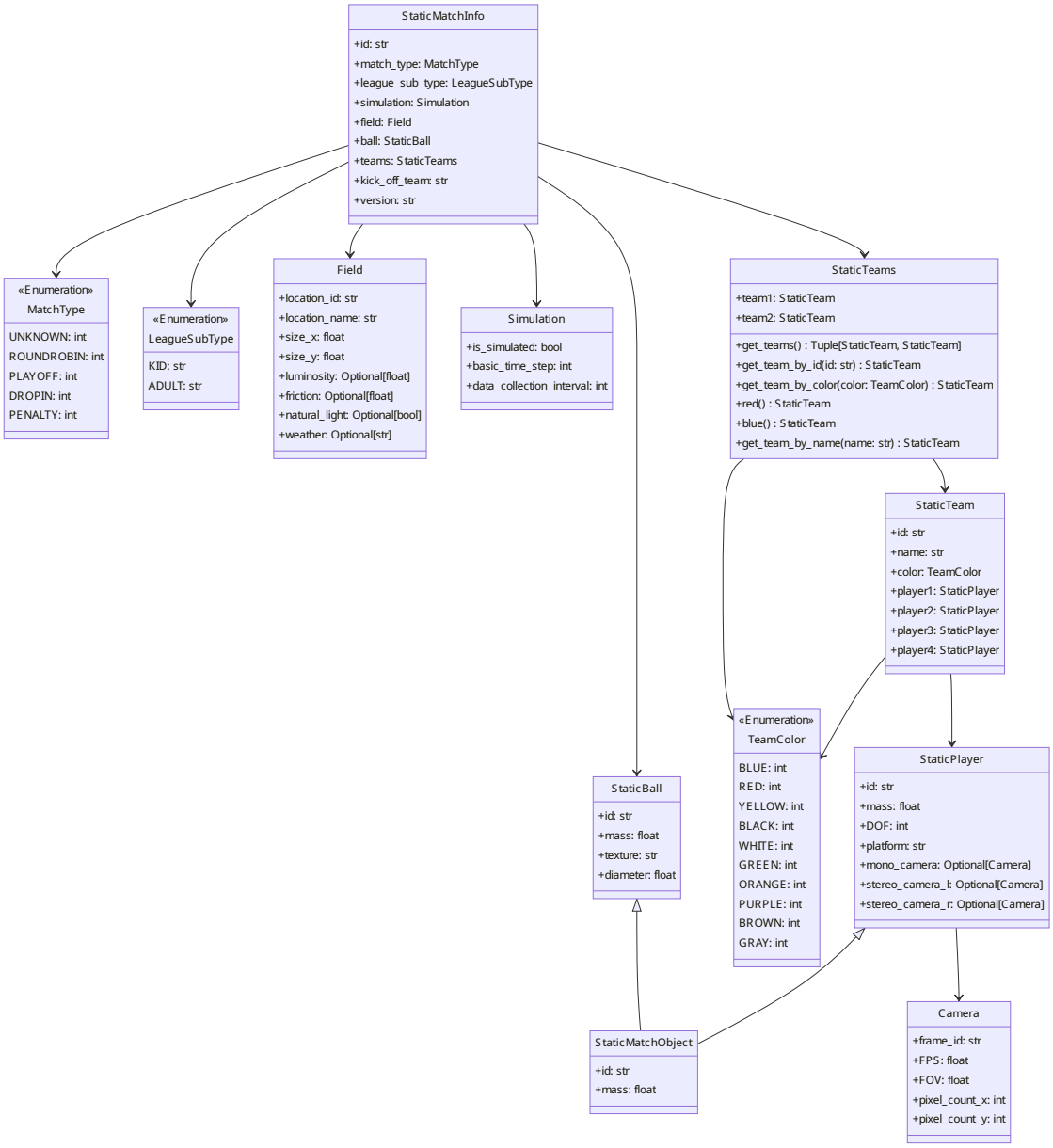

Figure 4.2.: UML class diagram `StaticMatchInfo` related classes.

JSON strings. This is used to save the `StaticMatchInfo` metadata as a JSON file and to more easily convert the list of `Step` objects to a Pandas `DataFrame`.

Section 2.1.1 highlighted some advantages of using the Pandas [2, 3] software library for data analysis purposes. To be able to use the features of Pandas, the dynamic match data has to be converted to a `DataFrame`. This data format can only handle data in a flat, rectangular shape with labeled columns of similar data. However, until this point, the dynamic match data

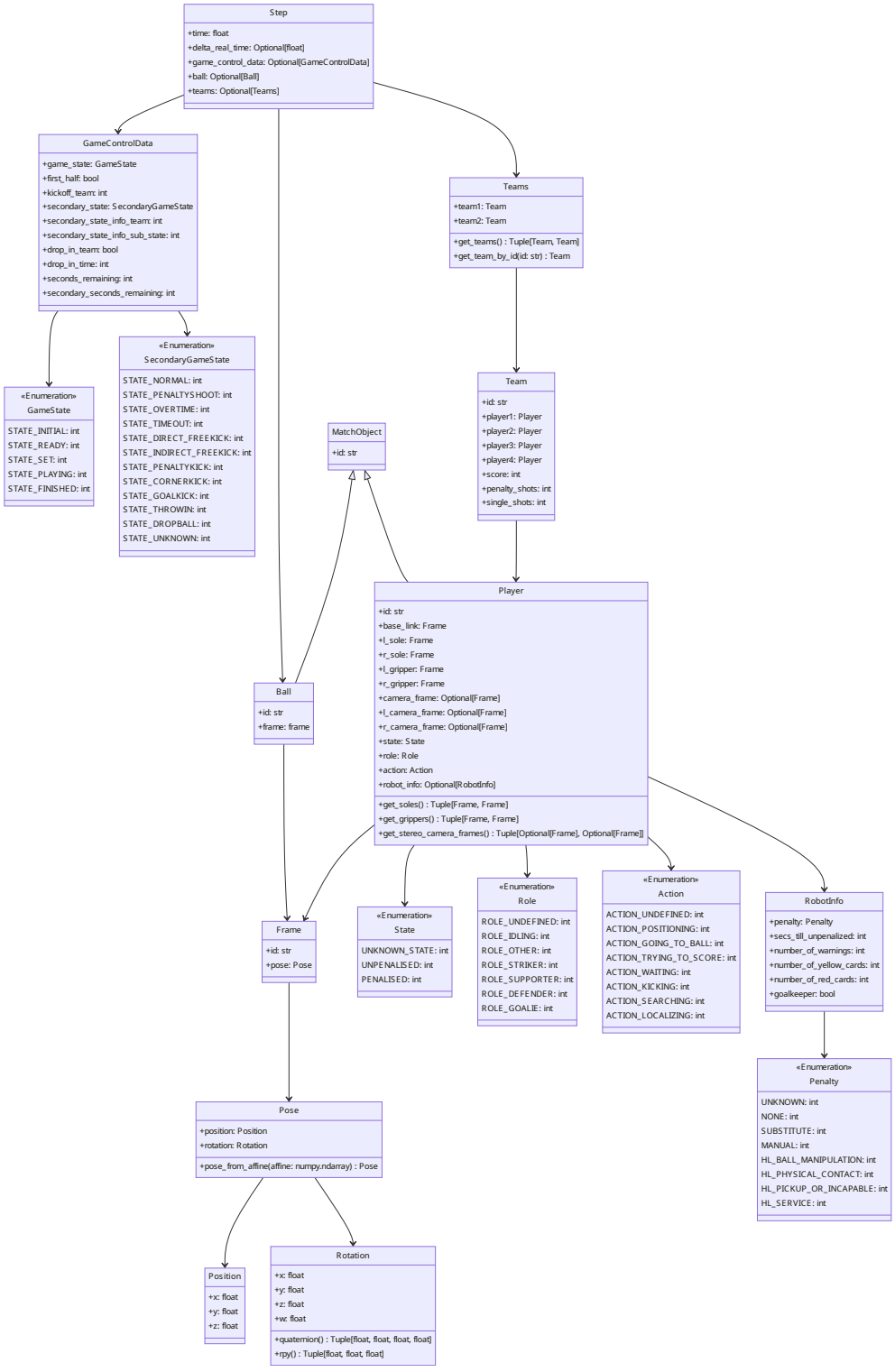

Figure 4.3.: UML class diagram of the `Step` and related classes.

```python
@dataclass(frozen=True)
class StaticMatchInfo(DataClassJsonMixin):
    """Static information about a match.

    :param id: Match id
    :param match_type: Type of this match (Normal, KnockOut, RoundRobin, DropIn)
    :param league_sub_type: Sub type of this match (Kid, Adult)
    :param simulation: Simulation data
    :param field: Field data
    :param ball: Ball data
    :param teams: Team data
    :param kick_off_team: Id of the team that kicks off
    :param version: Version of the match_info package
    """

    id: str
    match_type: MatchType
    league_sub_type: LeagueSubType
    simulation: Simulation
    field: Field
    ball: StaticBall
    teams: StaticTeams
    kick_off_team: str

    version: str = "0.0.1"
```

Listing 1: Source code of the `StaticMatchInfo` Python class

is stored in a list of hierarchical `Step` objects. This data conversion can be achieved by first utilizing the `to_dict()` method provided by the `DataClassJsonMixin` to get proper Python dictionaries from the `Step` objects. Finally, using the `json_normalize(...)` function from Pandas with the list of dictionaries, results in a `DataFrame` object. This method flattens the hierarchical dictionary structure, by creating a column in the new `DataFrame` for each item and labeling it with a concatenation of the item's keys, including a separator character. Resulting column labels may look like the following example, using a dot character as a separator: `teams.team1.player1.base_link.position.x`. The column with this exemplary label contains the values for every measured simulation step for the X-component of the position of the base_link frame from the first player from the first team. A source code snippet implementing the conversion process just described can be found in the listing 2.

```
steps: List[Step] = self.get_steps()
if steps:  # Check if the steps list is not empty
    df: pandas.DataFrame = pandas.json_normalize(
        [step.to_dict() for step in steps]
    )
```

Listing 2: Snippet from the `save(...)` method from the `Match` class source code

4.1.2. File Format

The Pandas `DataFrames` containing the variable match data need to be written to a file system to make them persistent and sharable. The Pandas software library supports 13 formats for exporting and importing `DataFrames`. Three basic format types exist for writing and reading data: text, binary and SQL [2]. One of those various formats has to be selected. The SQL format with a corresponding database is unnecessarily complex, as just a single exported file is easier to manage and distribute. Thus, only text and binary formats are suitable. For further selection, the `HTML` format has been disregarded, as it is rather intended to present data from a `DataFrame` instead of storing it. Still, 10 file formats remain, with 3 text formats and 7 binary formats. The remaining available file formats are:

- text formats:
  - `CSV` [25]
  - `JSON` [26, 27]
  - `XML` [28]
- binary formats:
  - `MS Excel` [29]
  - `Python Pickle` [30]
  - `HDF5` [31]
  - `Feather` [32]
  - `Parquet` [33]
  - `ORC` [34]
  - `Stata` [35]

To narrow those down further, benchmarks have been run to measure the read and write durations, as well as the exported file sizes. Short export and imports durations should be preferred over longer ones, as this occupies less CPU resources or enables more frequent automatic saves. Similarly, smaller file sizes are considered a benefit, as less valuable storage space gets taken. In order to create those benchmarks, a software called `Pandas File Format Benchmarking`[3] has been developed. The tests included in this software use a Pandas

[3] `https://github.com/jaagut/PandasFileFormatBenchmarking/`

DataFrame that is held in memory. This data is then exported in one of the above-mentioned file formats and re-imported afterwards. The time it took to export and import the DataFrame as well as the file size are measured. For each file format, the tests are repeated several times, at default 10 times.

The results, that will be presented shortly, were achieved using original data from the HLVS 2023 match from game day 3 the second game (K-GD3-2). All 10 above-mentioned file formats were tested 10 times, except for XML and MS Excel which were only tested once, as they took significantly longer than the other formats. This behavior also showed in preliminary test runs, which suggests, this behavior is not uncommon for these. Thus, repeating runs for XML and MS Excel unfeasible and not necessary, as fast imports and exports are of interest. Compared to the average of all other formats (3.799 s), write durations for XML and MS Excel are 15.568 and 73.475 times higher, respectively. Read durations are much higher, with 7257.5055 times the average (2.224 s) for XML and 66.964 times for MS Excel. The exported file sizes do not such extreme variations with 12.299 and 1.004 times the average of 152.121 MB for XML and MS Excel respectively.

The following charts disregard both file formats XML and MS Excel, as detail would get lost when including them. Figure 4.4 shows that the format Feather is the fastest, i.e., it has the shortest write and second-shortest read durations in this benchmark scenario. As can be seen in figure 4.5 this format produces also the smallest exported file sizes. Measurements from all tests can be found in the appendix in table 2.

Due to these results, Feather has been selected as the main file format for reading and writing Pandas DataFrames in this approach. Although, as a backup solution, the close second format Python Pickle was chosen.

4.1.3. Testing

To ensure the correctness of the management and store functionality of the DataCollector, a few software tests using the pytest framework [36] have been implemented. These tests ensure, that the creation of StaticMatchData, Step, and *DataCollector* objects with dummy data is possible. Afterward, tests check if the automatic save and finalize methods write the correct files to the file system.

4.2. Data Collection

This section is concerned with the actual process of gathering data from the simulation environment and filling in the data structure as described in section 4.1. In order to reduce manual effort from people simulating games, this process needs to be as automatic as possible. As seen in section 2.2, the HLVS simulation environment consists out of many individual parts, each having a specific purpose and handling different information. Therefore, it needs to be decided which information should be gathered from what component. The main component providing data will be the HLVS automatic referee software. However, it cannot provide everything needed for the DataCollector data structure, which is why some post-processing is needed.

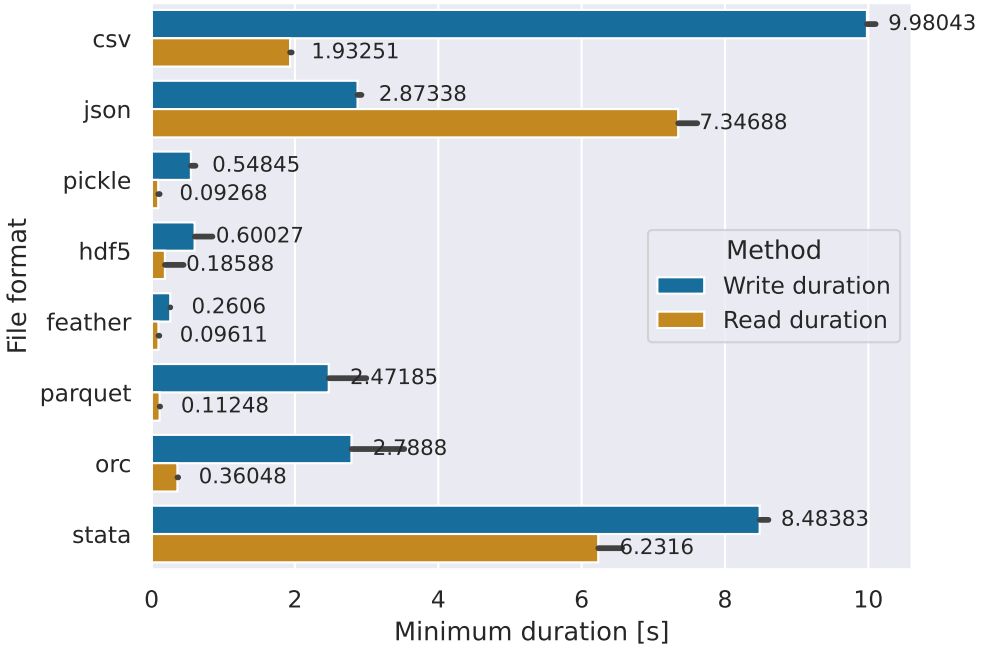

Figure 4.4.: Bar plot showing the minimum write and read durations (in blue and yellow respectively) including confidence intervals for the 8 remaining file formats in Pandas measured in 10 trials. The format `Feather` has the shortest write duration of 0.261 s and has the second-shortest read duration of 0.096 s close behind `Python Pickle` with 0.093 s. The confidence intervals also show that there is no large variation for this format.

4.2.1. Integration into the automatic referee software

Regarding all software components comprising the HLVS setup, the automatic referee – hereafter just called referee – has access to most of the desired information. This is, since it is a Webots supervisor-controller and therefore can access all data relevant to the simulation itself, more than any robot-controller. Additionally, the referee software communicates with the game controller software, thus always has the most recent state. Lastly, the referee implements the game's rule's logic, and as such already uses information, wanted for collection purposes. For those reasons, the referee was chosen as the primary component to extend with the data collection process.

Section 2.2.3 established three distinct phases of the referee's runtime. To tie into these three phases, the data collection has been split into matching parts:

Initialization

The configuration that gets read in the initial setup phase also includes configuration regarding the data collection. The data collection can be configured to be turned off, in which case,

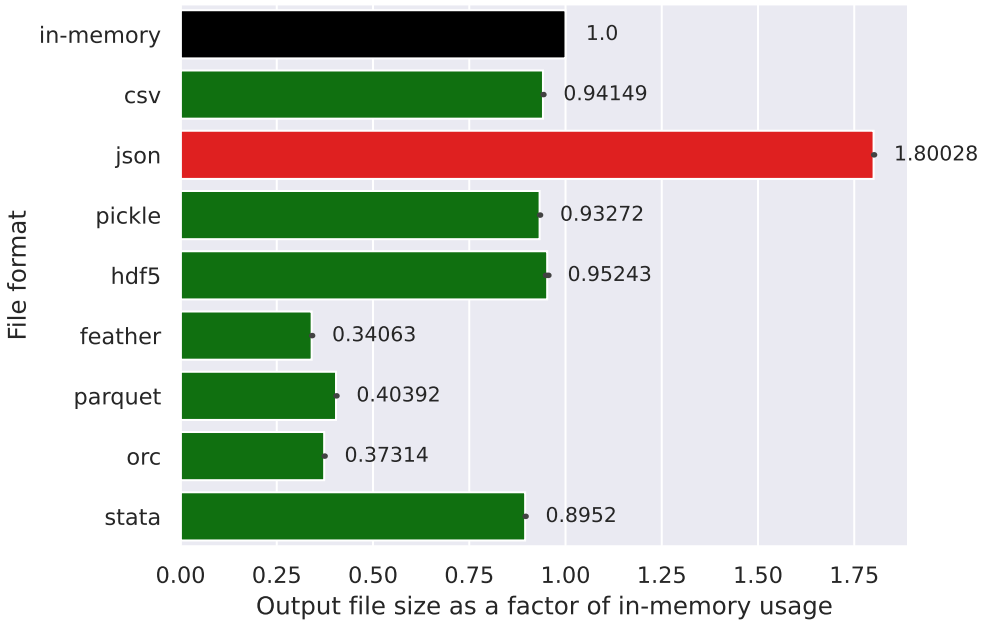

Figure 4.5.: Bar plot showing the size of exported files for each format compared as a factor to the in-memory representation of the Pandas `DataFrame` which utilized 183.285 megabytes in RAM (shown in black). Formats represented by green bars have a file size that is smaller compared to the in-memory size. Analogous, bars is red are larger. The format `Feather` has the smallest file size with $0.341 \cdot 183.285$ megabytes $= 62.500$ megabytes. All file formats are smaller than the in-memory representation, except for the `JSON` text format.

everything regarding the gathering process is skipped, and the referee runs without it as normal. In case, the data collection is enabled, first, references to each desired frame or `Node` object from the supervisor controller gets collected. Having a reference to the `Node` objects simplifies and speeds up the pose gathering later on. The references (one for the ball and up to eight per player) are stored in a dictionary for later use during the main loop. Secondly, the `DataCollector` data structure (see section 4.1) gets initialized. This involves setting the configured output directory and interval for automatic saves, and setting the `StaticMatchInfo`. As previously described in section 4.1.1, the `StaticMatchInfo` holds hierarchical metadata about the match. To fill in this information, the game's configuration and previous referee setup are queried. However, not every necessary information can reasonably be gathered at this time, due to high complexity or missing interfaces. Thus, the values for a player's mass, degrees of freedom (DOF), and camera settings get filled with dummy information. Later on, a post-processing step fills in the correct information (see section 4.2.2).

Main loop

During the main loop of the referee, all time variable information is gathered that comprises the `Step` class from section 4.1.1. Each iteration of the main loop progresses the simulation and time. Shortly after that, and before any decisions and actions are taken by the referee, the data collection runs if conditions apply. Two conditions determine whether the data collection runs. The first condition checks whether the data collection process is enabled, this is the case, if it was configured as such in the first place and no errors occurred during the data collection process until now. In case, something goes wrong, an exception handler turns off the data collection, so it does not further impact the game simulation. The second condition checks whether a configurable interval of steps has passed. This subsampling mechanism of the data collection process has been introduced to reduce computational resource overhead. At default, the data collection is configured to run only every 8th step. With a basic simulation time step of 8 ms and running every 8th step, this means 64 ms of simulation time passes between each data collection run, i.e., it runs at 15.625 Hz.

In case both conditions apply, and thus the collection of variable data should run, four steps happen. The first step is to create and set a new `Step` object to the `DataCollector`. This only requires the current simulation time. The second step is to gather and set data regarding the ball to the newest `Step` object. For this, the reference to the ball's `Node` object is used to query the current pose. The next step gathers the latest game controller message, if available, and sets complete `GameControlData` to the `Step` object. As seen previously in figure 4.3, the `GameControlData` object holds information such as the (main and secondary) game state, remaining time and kickoff teams. Lastly, the fourth step happens, which sets the team data to the `Step` object. This includes the current score of the team and information about each player. For each player, current game controller information is used, such as the number of penalties and cards. Additionally, a pose is queried from each of the previously collected up to eight player's `Nodes`:

- `base_link`
- `l_sole`
- `r_sole`
- `l_gripper`
- `r_gripper`
- `camera_frame` (optional)
- `l_camera_frame` (optional)
- `r_camera_frame` (optional)

Those nodes are based on REP 120 [37] and necessary, to compute the `base_footprint` frame afterwards. After those four steps, and all decisions and action by the referee happened, the final value gets set to the latest `Step` object, which is the duration it took from the

beginning of this simulation step, i.e., the current main loop iteration until this last action in real time.

Finalization

The third and final phase of the data collection runs together with the finalization of other components of the referee. This stops and finalizes the *DataCollector* object, which will save the now complete data to the file system.

4.2.2. Post-Processing

After the integration of data gathering processes in the HLVS automatic referee software, some information is still missing. As mentioned above, some metadata fields have been filled with dummy data, which needs to be replaced. Additionally, some player-internal information about cognition of the robot's environment or actions intended by the behavior is of interest, but could not be provided by the referee.

Additional robot data

The referee filled in dummy values for the player's mass, DOF, and camera settings. In order to replace this with the correct information, a small tool has been developed. It reads in a JSON file with the static match information exported by the data collection process and a second JSON file with manually collected correct data. The tool matches the name of the robot platform and inserts the correct data for each player, thereby replacing the dummy data. To manually determine the mass of each robot platform, a Webots simulation has been started including that robot, then the mass can be read from the Webots GUI. Regarding the DOF, the number of joints in the robot's Proto model has been counted. Information about the camera settings has also been read from the Proto model. This method is only feasible for a small number of teams and robot platforms. The tool also assumes that the robot models do not change during the virtual season.

Merging team-communication data

In need of the player-internal information about cognition and behavior, another HLVS setup component has to be utilized. The most cohesive, flexible and accurate information could be gathered by defining a standard interface that describes, how team software can expose those internals at every time step, such that it can be collected and analyzed afterwards. However, this approach was seemingly not feasible, as this would require considerable development effort of each team.

As a compromise, the team-communication system could be utilized to gather the missing information. Compared to the new standard interface, this method does not necessarily provide the internal information for every simulation step, as the team-communication's frequency is controlled by the individual teams. Team-communication is used by team players to exchange internal information between team members via a computer network. Such

communication could include perceptual data as the self-localization pose, or a pose of a recently detected game objects, and strategical information enabling players to agree on tactics. The HL technical committee published a common and standardized communication protocol, `RobocupProtocol`[4] based on Protobuf [38] messages. This protocol already gets used by some teams, and thus no additional effort is needed from those teams. It is also extensible, for example, the team Hamburg Bit-Bots has extended the `RobocupProtocol`[5]. The team-communication works by sending UDP packets to a network host provided by the HLVS infrastructure during the simulation time. This host runs the so called `UDP-bouncer` software. The main purpose of the `UDP-bouncer`, as described in section 2.2.7, is to distribute the game-controller and team-communication packages over the local network. However, this tool also logs every received package, including the team-communication data, to a file called `bouncing_log.txt`.

The post-processing step consists of reading this log file, parsing the team-communication messages, and finally extending the `DataFrames` exported by the data collection framework. This is done in multiple steps using another tool. The `bouncing_log.txt` file is a simple text file and contains logs apart from team-communication messages as well. The file also contains a list of IP addresses that are associated with one of two teams. Each logged message is written to a new line prefixed with a header containing the receiving time, sender IP, and sender port address. Thus, each line can be evaluated individually. Using the sender IP address, a message can be correlated with a team. This tool tries to parse the raw binary message contents as a `RobocupProtocol` message using a library generated by the Protobuf compiler. If this succeeds, the tool has access to the original data sent by a team player. Using a similar column label naming scheme as in section 4.1.1, the tool adds many new columns to the previously exported `DataFrame`. Before running this tool, an exemplary `DataFrame`[6] had 461 columns, afterwards 2593 columns. To fill in data from the team-communication messages to the new columns, the tool iterates through the parsed messages. Synchronizing the simulation times is necessary, as the data collection and team-communication can run with different frequencies. Synchronization is done by selecting the first row in the `DataFrame` with a simulation time greater than the message's simulation time. Once all team-communication messages have been inserted into the data collection, the `DataFrame` is re-exported to the file system.

4.2.3. Running Test Games

Some experiments require simulating test games locally without using the AWS as described in section 2.2.8. To simplify this process and reduce mistakes, a bash script has been developed. This tool reads a `game.json` file given as a single argument to gather the IP-address and port configuration for all hosts. Using the command `ssh`, it cleans up possible residues from previous test games, pulls the latest team's docker image and starts it on the machines configured as team players. Afterwards, it starts the Webots simulation world, including the

[4] `https://github.com/RoboCup-Humanoid-TC/RobocupProtocol`
[5] `https://github.com/bit-bots/RobocupProtocol/`
[6] `https://data.bit-bots.de/17gutsche/BA/test_repeat10/ba_repeat_01/data/`

automatic referee and the other components. Once the simulation is finished, every process and docker container gets stopped. One important note is, that it seems to be the game controller software can handle only one team player connection per IP address. Running multiple docker containers on the same machine but with different ports leads to connection issues.

4.3. Data Analysis

This section will concentrate on analyzing the structured data that has been gathered using the methods previously mentioned. By analyzing the data, new knowledge and insight should appear. The main contribution of this approach is the implementation and visualization of additional metrics based on the `SoccerAnalyzer` architecture.

4.3.1. Extension of the SoccerAnalyzer

After structured data has been collected using the methods previously presented in the sections 4.1 and 4.2, it needs to be analyzed in order to gain knowledge from it. This approach will build on top of the `SoccerAnalyzer` by Pereira et al. in [1] as introduced in section 3.2. The `SoccerAnalyzer` has a modular architecture, that is designed to be extended further. A schematic overview of the `SoccerAnalyzer` framework's architecture can be found in figure 4.6.

Instead of only allowing CSV files for input, the `Match` object takes any `DataFrame` and optionally additional metadata. Thereby, `Match` holds both, the static and the time-variable data from HLVS games. On its own, the `SoccerAnalyzer` can handle games from the RoboCup 2D Soccer Simulation League (SIM2D) and RoboCup Small Size League (SSL). This has been extended by creating an additional `CategoryMapping` for the RoboCup Humanoid League (HL) kid size. This class maps between league-agnostic terms used in the `SoccerAnalyzer` and terms used in the HLVS data, which are mostly column labels from the `DataFrame`. Additionally, the `CategoryMapping` contains landmarks from the HL, which represent the field-, line-, and goal dimensions as specified in the rules [4]. A new function has been introduced to draw the field-lines in plots created with Matplotlib [39]. An example can be seen in figure 4.7.

The `MatchAnalyzer` module did not require any adjustments besides enabling new analysis modules. Some metrics were already implemented by Pereira et al. as Analysis modules, including a ball history, expected goals and ball possession. However, those modules needed to be modified to handle three-dimensional information in addition to the two-dimensional information as before.

4.3.2. Metrics

The following table 4.1 proposes the metrics to implement.

Metric	Input	Description
Sim/real-time quotient	Time to calculate simulation, Start-, End time	also called Real-time-factor
Correlation between object localizations and simulation environment parameters	Object localizations of players (team communication), field- and ball characteristics	
Self-localization	Player pose, player localization (team communication)	Accuracy of player's own localization on the field
Ball estimation	Player's localization of the ball (team communication), Ball pose	Accuracy/Recall
Robot estimation	Player's localization of other players (team communication), Player poses	Accuracy/Recall
Looking direction	Player's camera frame	Where does a player look
Objects in view	Player's camera frame, camera matrix, object poses	What objects should be visible to a player
Ball touches	Collision matrix	Ball collided with another object
Ball kicks	Collision matrix, player's sole frame	Ball collided with a player's foot with considerable force
Ball kick force	Collision matrix, player's sole frame	
Ball kick distance	Ball kicks, ball poses	Distance moved by ball after being kicked
Ball kick speed	Ball kicks, ball poses	Speed of ball movement after being kicked
Successful kicks (excl. dribbles)	t.b.d.	
High kick	Ball kicks, ball poses	Ball gained height during kick
Ball dribble	Collision matrix, ball poses, player poses	Ball moves together with a player after repeated collisions
Ball dribble distance	Ball dribbles	
Ball dribble speed	Ball dribbles	
Kicks vs. dribble	Ball kicks, ball dribbles	

Metric	Input	Description
Time to kick	Ball kicks, player poses	Time between a player stops in ball's vicinity and kicks
Passing accuracy	Ball kicks, player poses	Kicks towards a teammate
Ball distance to own goal	Ball pose, goal pose	
Ball possession	Ball pose, player poses	Ball is in possession of nearest player
Time under control	Ball possession	Duration of ball possession by player or team
Ball movement distance while in possession	Ball possession, ball pose	
(Own-)Goals	Goal event	
Goal kick	Ball kicks, goal pose, goal event	Kicks in the vicinity and direction of a goal, resulted in a goal?
Defended goals	Goal kick, collision matrix	Unsuccessful goal kick, defended via collision with opponent's player
Meantime between goals and standard situations	Game controller events	
Goalie switches, cards, warnings	Game controller events	Count and frequency of...
Active player	Player state and role	Player is active, if state unpenalized and role is not supporter or idling
Base-footprint frame	Base-link pose, l/r-sole pose	As in REP 120
Walking distance	Base-footprint pose	
Walking speed	Base-footprint pose	
Walking acceleration	Base-footprint pose	
Walking step	Sole pose	Count, frequency, and distance of a step
Support polygon	Sole frames	
Upright?	Player base link	If a player is upright
Player is falling?	Player base link	Player falls, if base-link is outside of support polygon or tilted and below median height?
Player has fallen?	Player base link	If a player has fallen down

Metric	Input	Description
Time to stand up	Falling, upright	Time it takes for a player to get from Falling $\rightarrow$ (Fallen $\rightarrow$) Upright

Table 4.1.: Listing and description of proposed metrics.

However, only a subset of these metrics has been fully implemented:

- Waling speed
- Absolute errors of self-localization
- Absolute errors of ball estimation

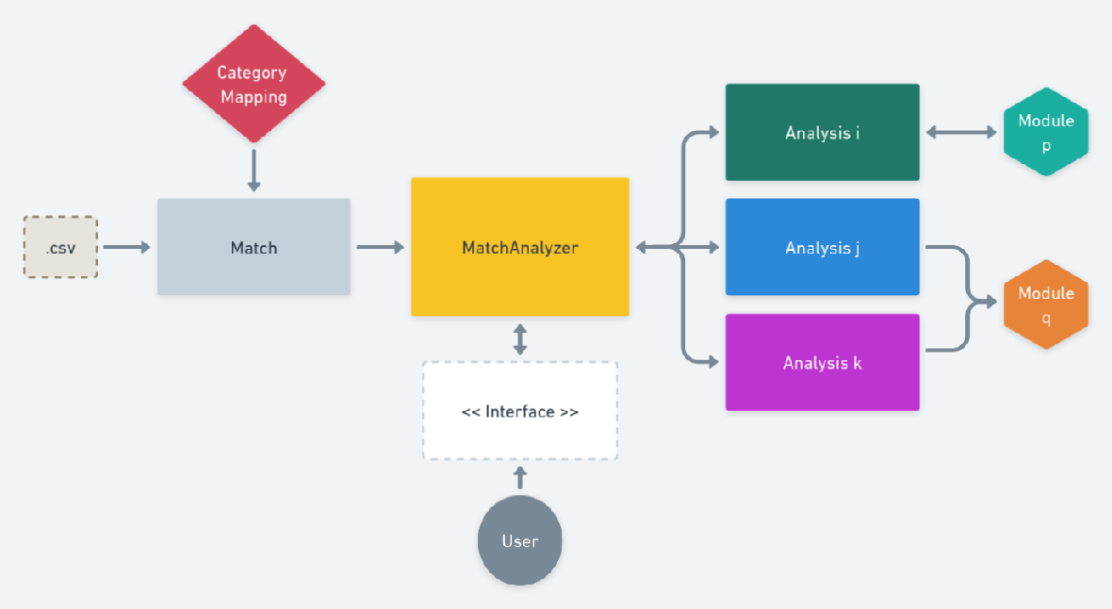

Figure 4.6.: Schematic overview of the modular architecture of the `SoccerAnalyzer` framework [1]. The main component of this architecture is the `MatchAnalyzer` in the center. Every other component interfaces with this piece. The `MatchAnalyzer` needs a `Match` (on the left) as input to analyze data from. The `Match` module represents a single soccer game. The `.csv` block signifies the import of a Pandas `DataFrame` from a CSV file. Different RoboCup leagues have vastly different rules and definitions, this is where the `CategoryMapping` helps to unify the data such that the `Match` object can be analyzed regardless of the current league. `Category` refers to some RoboCup league. `Analysis` modules are used by the MatchAnalyzer to calculate requested metrics from the game. Those can make use of auxiliary `Modules` that abstract common steps of analyzing a match. Finally, the `MatchAnalyzer` provides the analysis' results through a programming interface. This interface can be used by a user, e.g., to visualize plots in `JupyterNotebooks`.

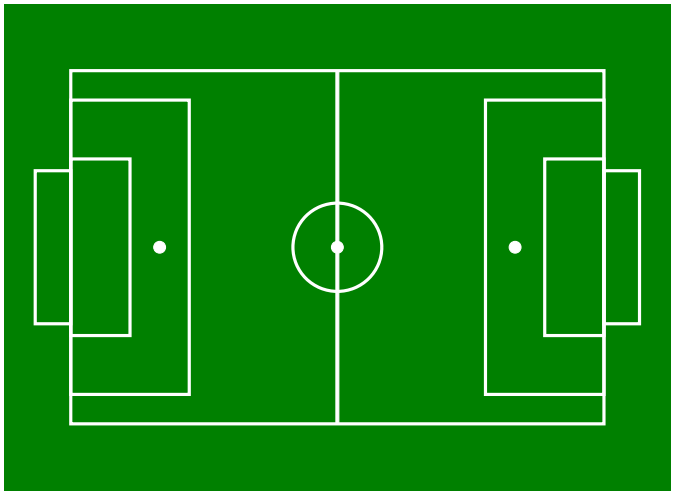

Figure 4.7.: Exemplary drawing of the HL pitch.

5. Evaluation

To answer the research questions from the beginning of this thesis (see section 1.1), the approach needs to be evaluated. Metrics that are calculated in the approach need to be statistically significant in order to be meaningful. For this, several experimental games with the same player software version have been simulated in the section 5.1.

However, this is not enough to answer questions about the predictive capability of the metrics. For this, multiple simulated matches with different team's software versions would be needed to compare those and to reason about. Unfortunately, this has not been done in the realm of this thesis. Therefore, no conclusions can be made about this.

5.1. Significance of Metrics

To evaluate whether the metrics introduced in section 4.3.2 are statistically significant, multiple matches have been simulated using the exact same team's robot software version. The simulation with identical configuration has been repeated ten times. The outcome of those matches varied widely, as the simulation and often times the robot's code is not deterministic.

The specific configuration that was chosen is a four players against four players setting with the usual domain randomization and the default data collection interval of 8. The robot software for both teams was an identical version from the Hamburg Bit-Bots team. Software from other teams participating in the HLVS 2022/23 did not work properly, as stable or as intended, which can be observed in the recordings. Selected was a Docker image, that previously has been used on the second and third game day's match during the season. Qualitatively compared to other versions, this image showed the least breakdowns and failures during the official games. The recordings and collected data of the ten trials can be found online [1].

5.1.1. Results

All ten repeated games start similarly, with, in total, eight robots walking onto the field and placing; the red team has kick-off in the first half of the game. Normally, a match spans 20 minutes of simulated game time, with 10 minutes each per half-time. However, four out of ten games, with over 25 minutes, ran considerably longer caused by many accidental collisions and ball-holdings which were penalized by the automatic referee with direct free kicks and penalty kicks. The remaining six games were stopped preliminary, due to an error in the communication between the automatic referee and the game controller. The error was

[1] `https://data.bit-bots.de/17gutsche/BA/test_repeat10/`

seemingly caused by non-normal player behavior (e.g., fouls) during direct free kick-timers. For example, the shortest game 08 ran just over one minute of simulation time.

Another anomaly appeared in most games: This is that some players, after seemingly random time, stopped moving and reacting to any situation. When this happened, the player usually was in the transition between different actions (from walking to falling, from falling to standing up or from standing up to walking again), or the player was just teleported to the field' side by the automatic referee.

The following table 5.1 shows a summary of key events during the ten repeated games. All time stamps are rough estimates of the exact time an event happened. Moreover, the time stamps have been read from the game's main clock; thus it shows the remaining minutes and seconds per half-time and decreases.

Match	Half-Time	Time Stamp	Description
01	1st	3:26	Player blue4 is unresponsive on the side
	1st	1:48	Game ends 0-0 preliminary with 15:01 minutes total simulation time
02	1st	2:32	0-1 score for blue after penalty kick
	1st	1:10	A blue player in unresponsive on the ground
	1st	0:25	1-1 scored by red
	2nd	10:00	Three players per team are unresponsive
	2nd	5:40	1-2 score for blue
	2nd	5:40	2-2 scored by red
	2nd		Remaining red player is unresponsive after collision
	2nd	0:50	2-3 score for blue
	2nd	0:00	Regular ending with 26:54 minutes total simulation time
03	1st	9:15	Player blue2 unresponsive on the side
	1st	8:50	1-0 score for red after penalty kick
	1st	1:39	Game ends 0-0 preliminary with 13:14 minutes total simulation time
04	1st	5:33	Game ends 0-0 preliminary with 8:41 minutes total simulation time
05	1st	6:54	Game ends 0-0 preliminary with 6:15 minutes total simulation time
06	1st	8:20	0-1 score for blue
	1st	4:40	0-2 score for blue after penalty kick
	1st	2:30	1-2 scored by red
	2nd	10:00	All players but blue2 are unresponsive
	2nd	7:00	1-3 score for blue
	2nd	4:30	1-4 score for blue
	2nd	2:00	1-5 score for blue
	2nd	0:00	Regular ending with 27:52 minutes total simulation time
07	1st	8:30	1-0 score for red after penalty kick

Match	Half-Time	Time Stamp	Description
	1st	5:00	Player blue3 unresponsive after getting up
	1st	2:10	One red player receives a red card, seemingly for hand play
	2nd	10:00	Only two players per team remain responsive
	2nd	7:38	Game ends 1-0 for red preliminary with 22:20 minutes total simulation time
08	1st	9:48	Game ends 0-0 preliminary with 1:12 minute total simulation time
09	1st	7:19	Player blue2 unresponsive after falling
	1st	4:50	Player blue3 unresponsive after falling
	1st	3:25	1-0 score for red after penalty kick
	2nd	7:50	1-1 scored by blue
	2nd	5:30	2-1 score for red after own goal by blue
	2nd	2:40	2-2 score by blue
	2nd	0:30	2-3 score for blue
	2nd	0:00	Regular ending with 27:56 minutes total simulation time
10	1st	9:25	Player blue2 unresponsive after getting up
	1st	8:30	1-0 score for red
	1st	5:40	Player red3 is unresponsive after falling
	1st	3:30	1-1 score by blue after penalty kick
	1st	0:25	Player red4 unresponsive after getting up
	2nd	5:50	2-1 score for red
	2nd	4:40	Player blue4 unresponsive after falling
	2nd	0:20	Player blue3 unresponsive after falling
	2nd	0:00	Regular ending with 27:39 minutes total simulation time

Table 5.1.: Summary of the ten repeated games.

The following sections present plots from the evaluated metrics. A more detailed view can be found in the appendix.

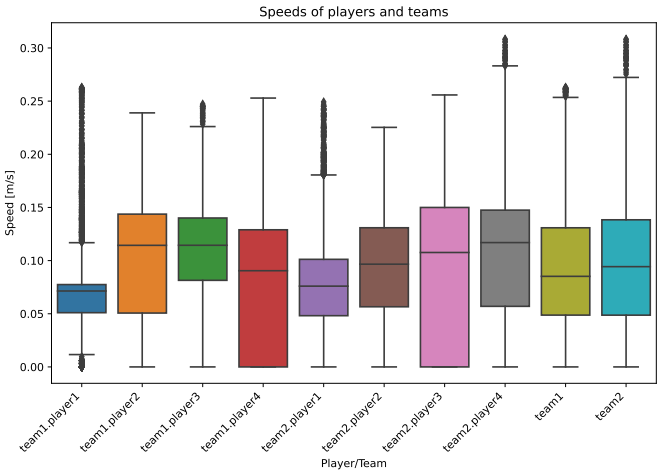

Figure 5.1.: Repeated match 01: The speed of all players and teams

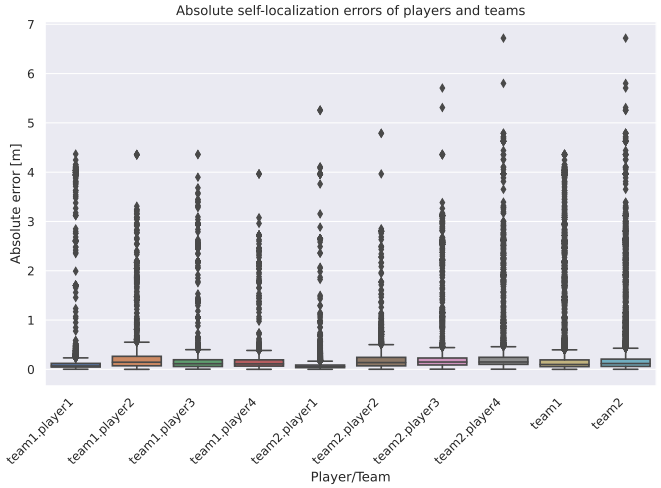

Figure 5.2.: Repeated match 01: The absolute error of self-localization of all players and teams

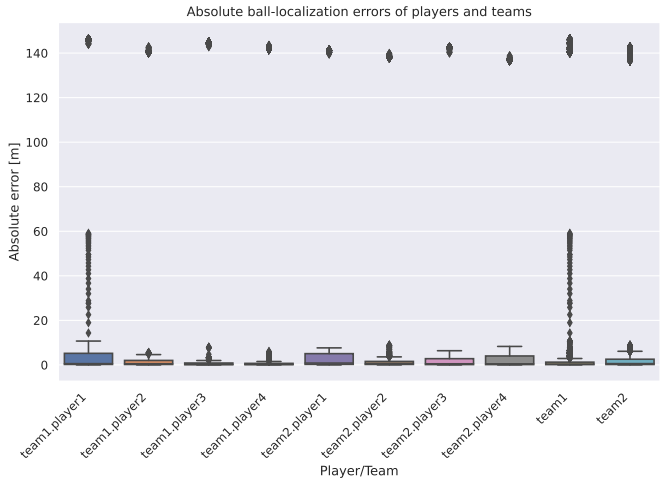

Figure 5.3.: Repeated match 01: The ball localization of all players

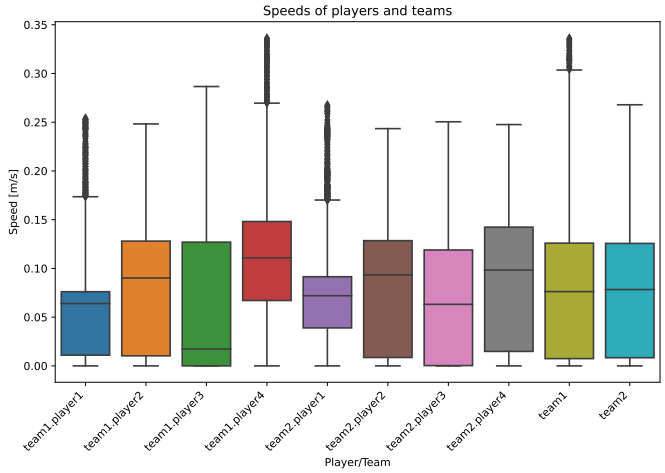

Figure 5.4.: Repeated match 02: The speed of all players and teams

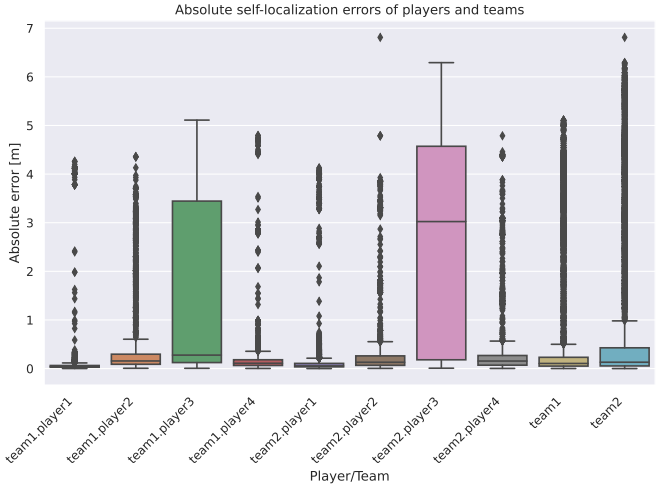

Figure 5.5.: Repeated match 02: The absolute error of self-localization of all players and teams

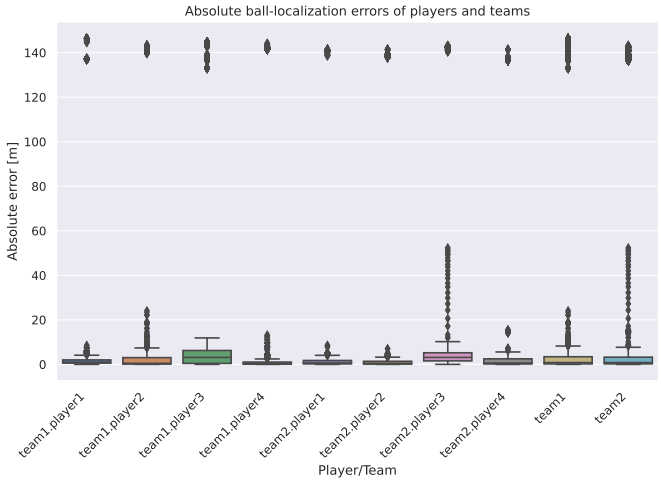

Figure 5.6.: Repeated match 02: The ball localization of all players

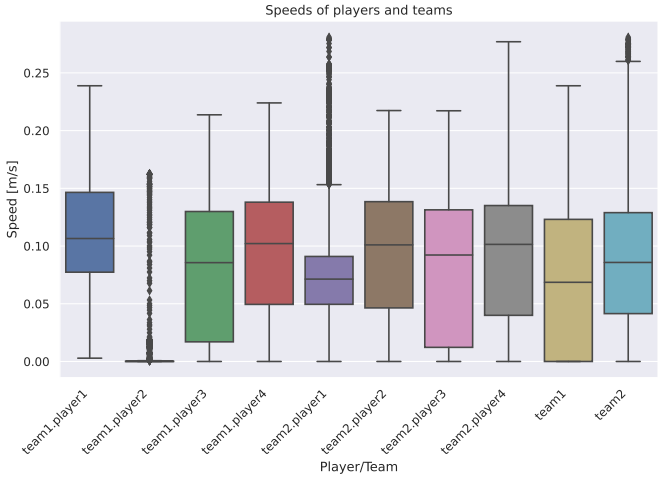

Figure 5.7.: Repeated match 03: The speed of all players and teams

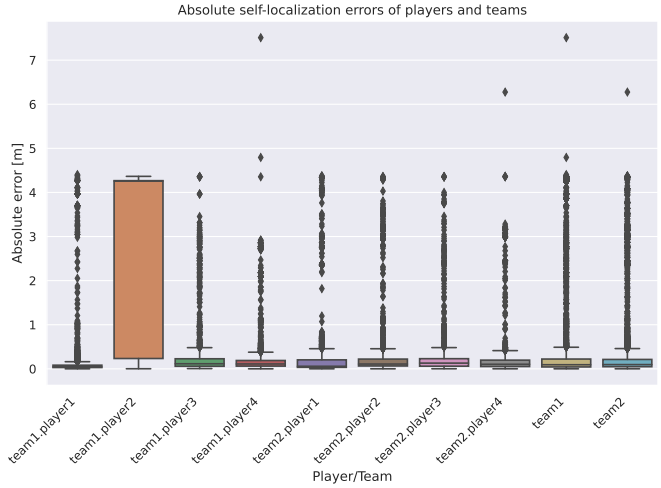

Figure 5.8.: Repeated match 03: The absolute error of self-localization of all players and teams

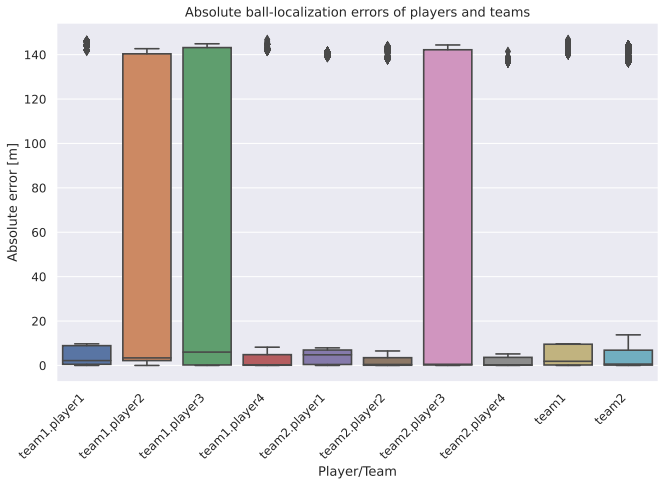

Figure 5.9.: Repeated match 03: The ball localization of all players

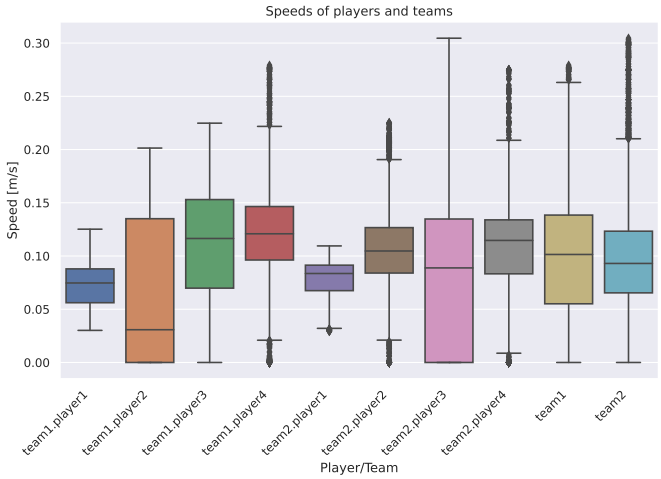

Figure 5.10.: Repeated match 04: The speed of all players and teams

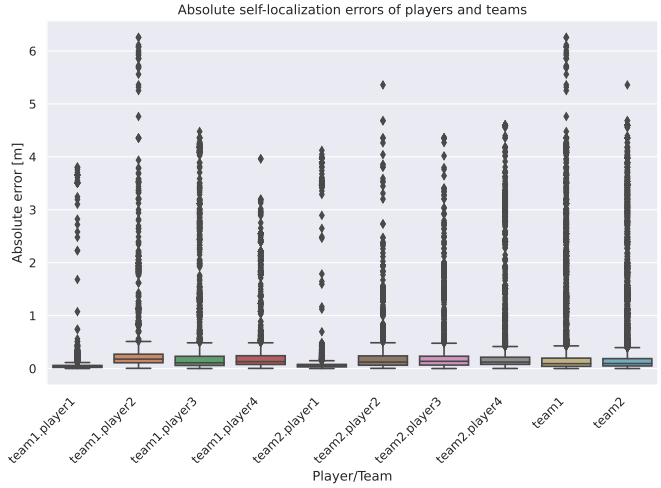

Figure 5.11.: Repeated match 04: The absolute error of self-localization of all players and teams

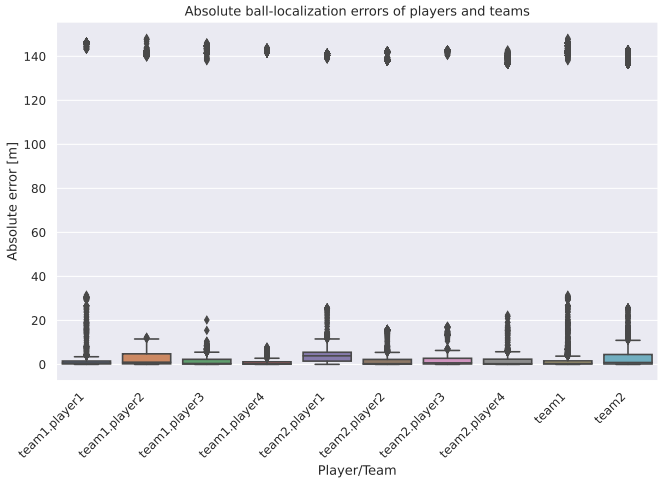

Figure 5.12.: Repeated match 04: The ball localization of all players

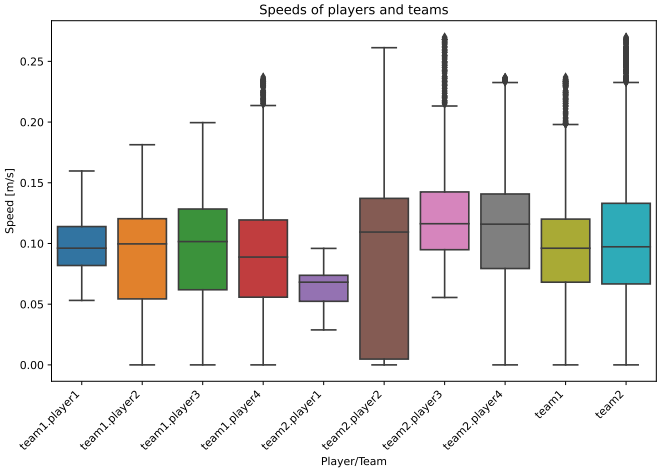

Figure 5.13.: Repeated match 05: The speed of all players and teams

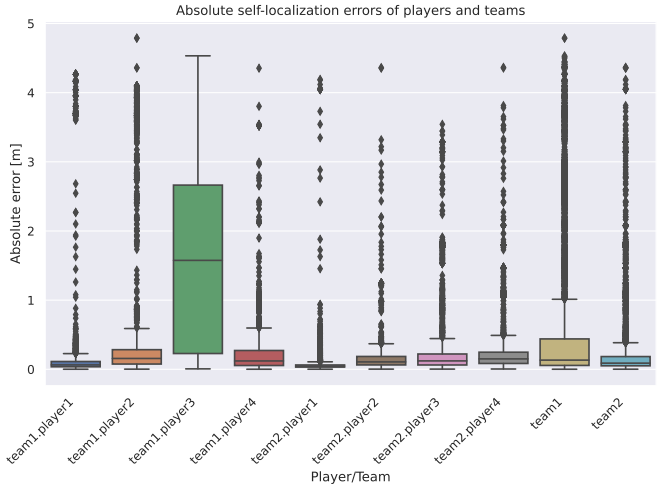

Figure 5.14.: Repeated match 05: The absolute error of self-localization of all players and teams

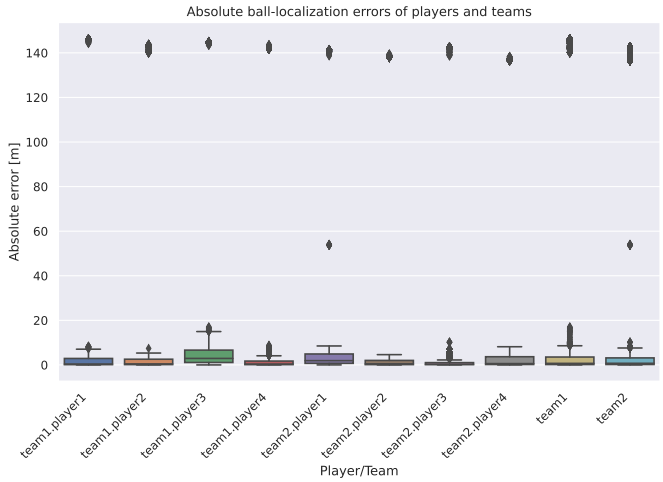

Figure 5.15.: Repeated match 05: The ball localization of all players

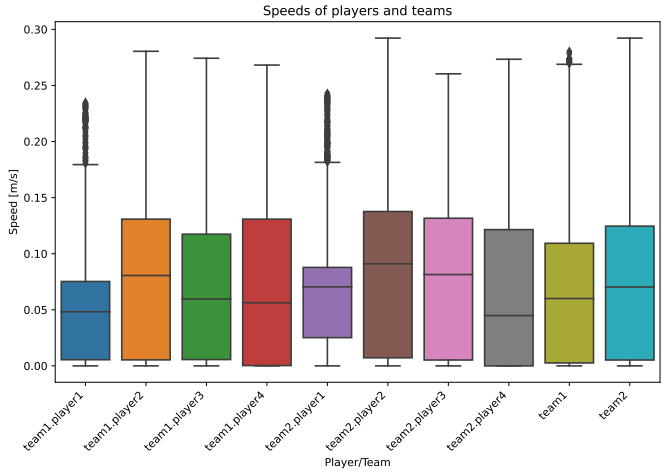

Figure 5.16.: Repeated match 06: The speed of all players and teams

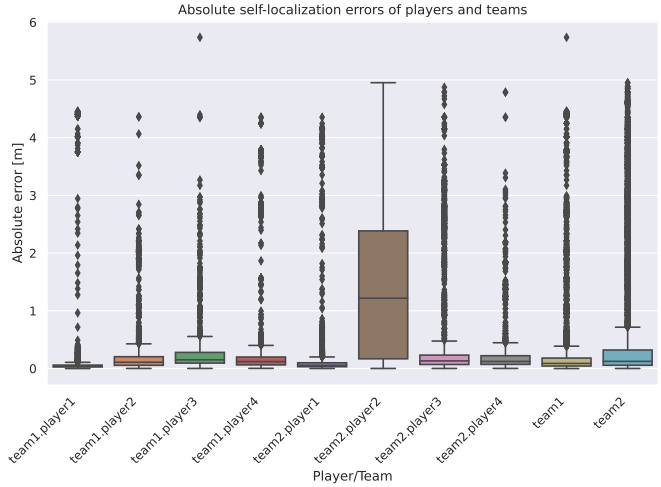

Figure 5.17.: Repeated match 06: The absolute error of self-localization of all players and teams

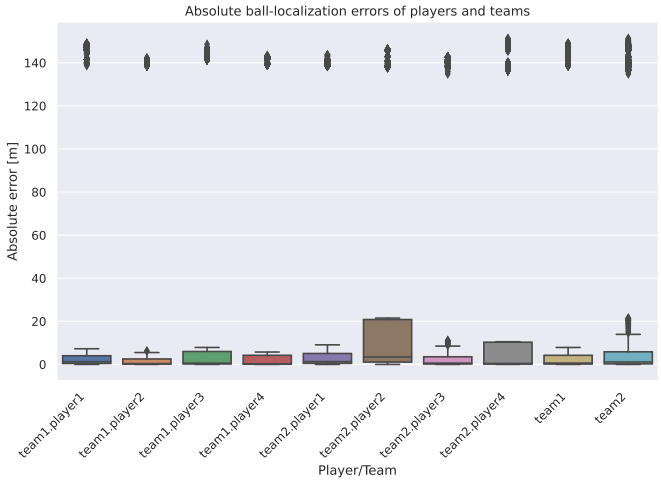

Figure 5.18.: Repeated match 06: The ball localization of all players

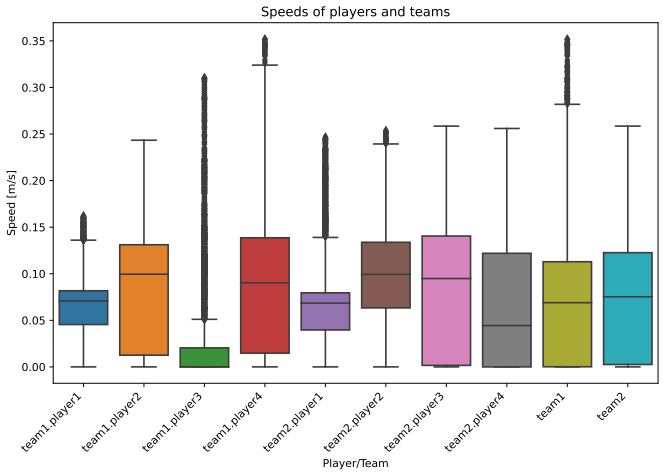

Figure 5.19.: Repeated match 07: The speed of all players and teams

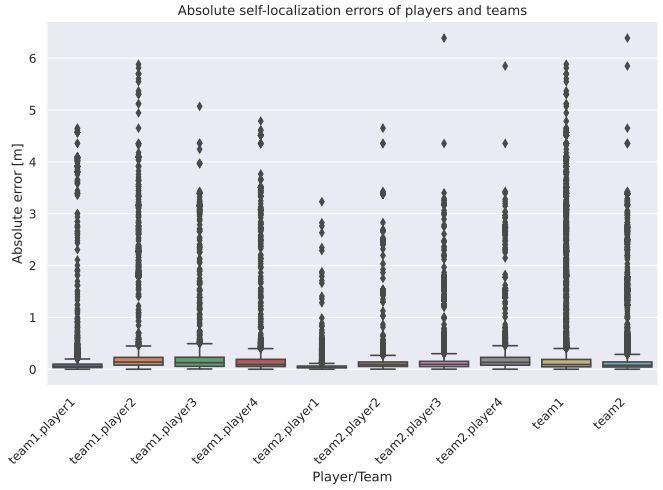

Figure 5.20.: Repeated match 07: The absolute error of self-localization of all players and teams

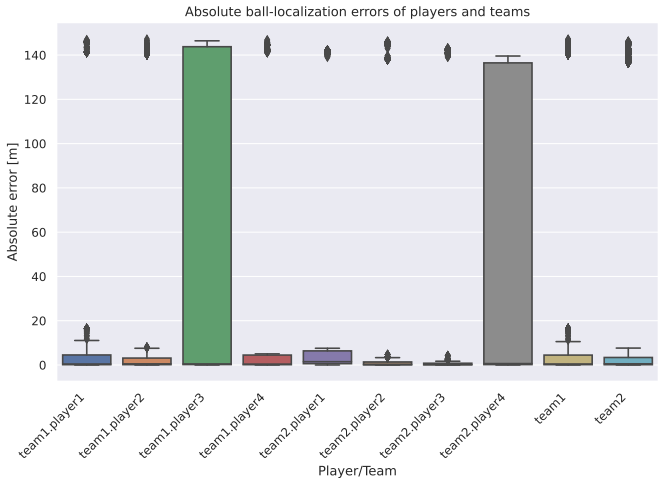

Figure 5.21.: Repeated match 07: The ball localization of all players

Repeated match 01

Repeated match 02

Repeated match 03

Repeated match 04

Repeated match 05

Repeated match 06

Repeated match 07

Repeated match 08

Repeated match 09

Repeated match 10

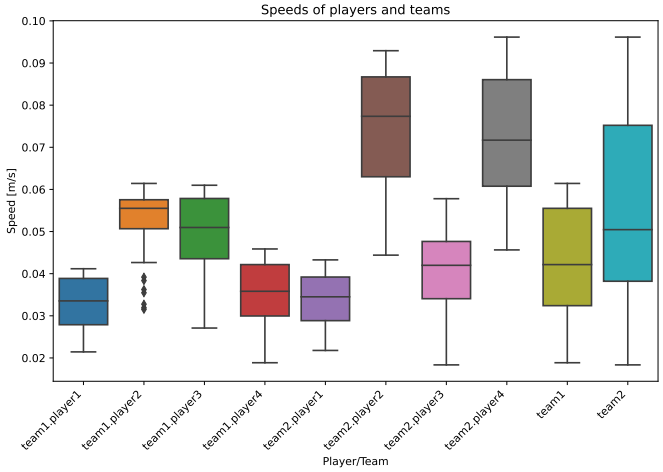

Figure 5.22.: Repeated match 08: The speed of all players and teams

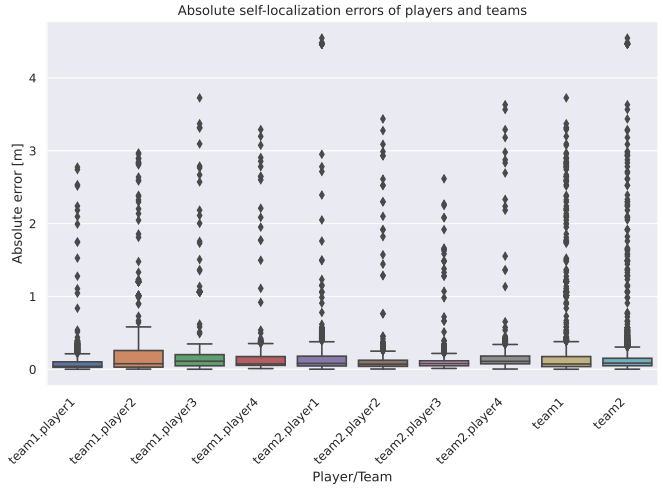

Figure 5.23.: Repeated match 08: The absolute error of self-localization of all players and teams

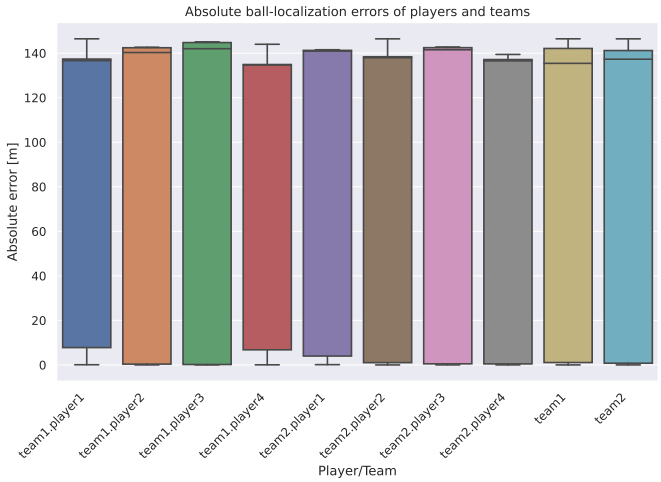

Figure 5.24.: Repeated match 08: The ball localization of all players

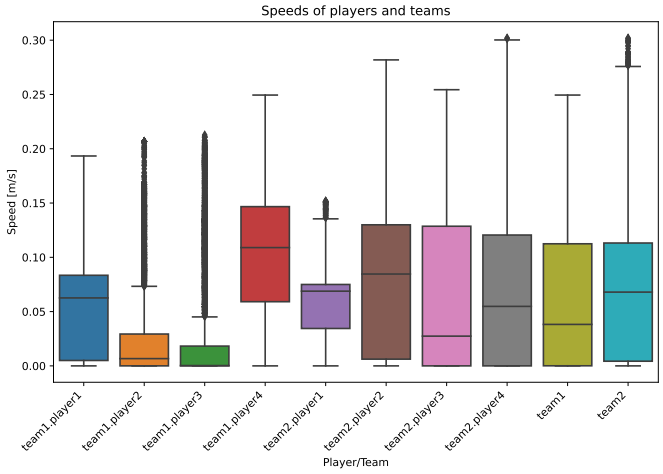

Figure 5.25.: Repeated match 09: The speed of all players and teams

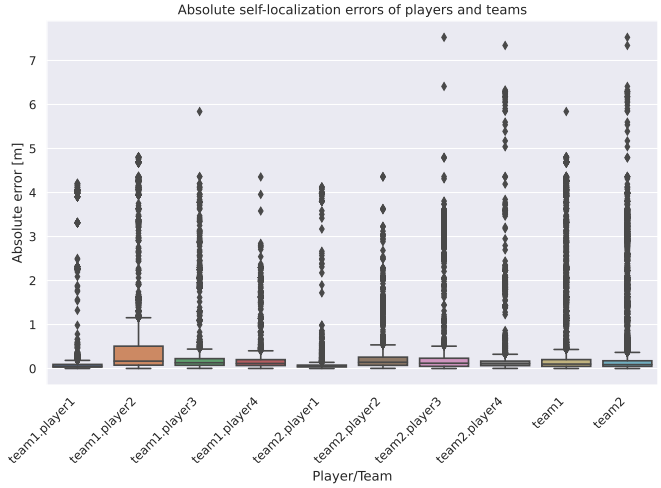

Figure 5.26.: Repeated match 09: The absolute error of self-localization of all players and teams

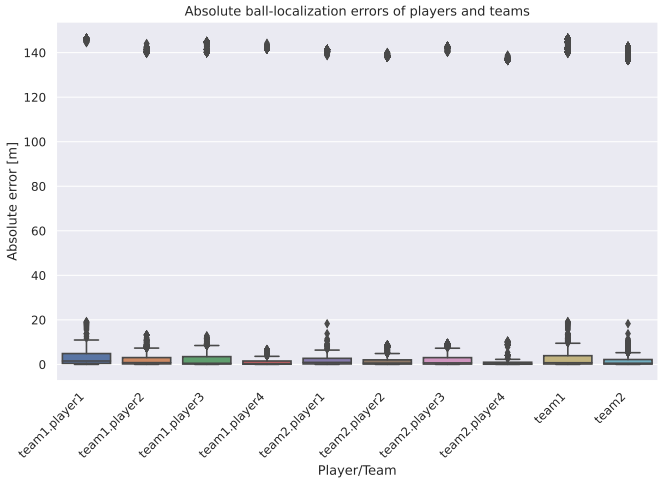

Figure 5.27.: Repeated match 09: The ball localization of all players

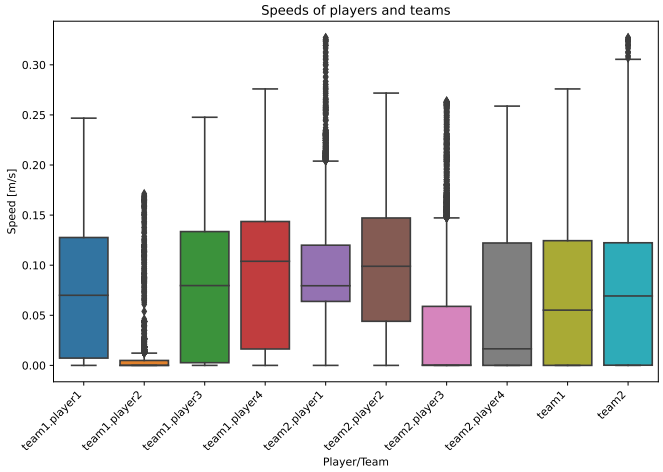

Figure 5.28.: Repeated match 10: The speed of all players and teams

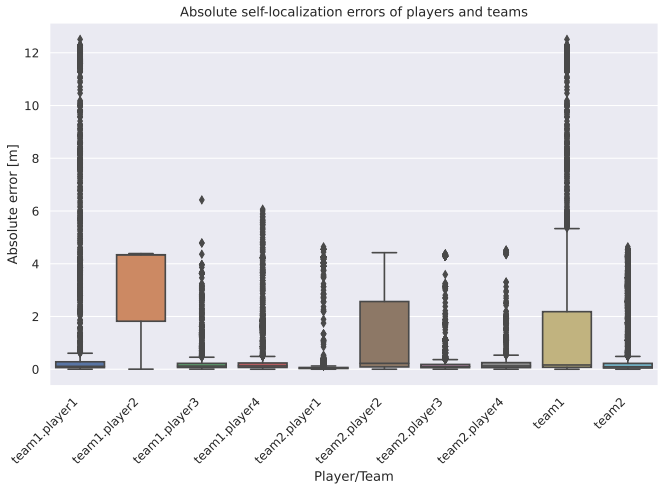

Figure 5.29.: Repeated match 10: The absolute error of self-localization of all players and teams

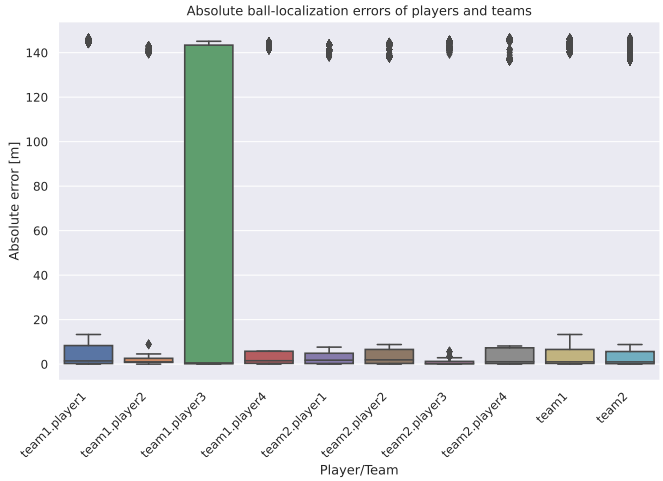

Figure 5.30.: Repeated match 10: The ball localization of all players

6. Discussion

In this chapter, the degree of answering the research questions will be discussed, especially regarding the evaluation (see section 6.1). Another discussion in section 6.2 surrounds possible impacts and values of this thesis and its development.

6.1. Research Questions

To recapitulate, this thesis tried to answer two research questions:

- Which metrics are meaningful in humanoid robot soccer?
- Do these metrics have the same predictive capability as in human soccer?

For measuring the meaningfulness of metrics, the evaluation in section 5.1 tried to determine statistical significance for each of the implemented metrics. This could not be achieved for multiple reasons. Firstly, not many of the proposed metrics have been fully implemented, and therefore they could not be evaluated. Secondly, no quantitative evaluation showed the significance of single metrics. The evaluation experiment did run 10 repeated simulations and collected the data of those, but only qualitative comparisons were given, though it should be possible to conduct quantitative evaluation from that. Moreover, the simulated games showed many anomalies, which reduces the validity of the results even more. Assuming, several functioning Docker images of more multiple teams would exist, the evaluation of the statistical significance of metrics could be vastly improved by running multiple experiment-sets of repeated games.

Answering the second question about the predictive capability would have required data from many experimental matches with progressing versions of team software. This has not been done, and thus no validation has happened to resolve this question.

6.2. Impact and Value

Besides not being able to answer the research questions, the approach provides value to RoboCup teams that plan to participate in upcoming virtual seasons. The value lays in the development of the data structure and data collection methods in the previous approach. Those components appear to be working very reliable. The resulting data can be analyzed using many other common tools. The data is not specifically tailored towards the (extended)

`SoccerAnalyzer`. Additionally, all data collected from official HLVS 2023 games[1] and simulated game for the purpose of evaluating this thesis[2] are fully available online.

[1] `https://data.bit-bots.de/HLVS/2023/`
[2] `https://data.bit-bots.de/17gutsche/BA/test_repeat10/`

7. Conclusion

The goal of this thesis was to apply common methodologies from data analysis to simulated the RoboCup Humanoid League Virtual Season (HLVS) games. For this, a data structure written in Python has been proposed and the official setup to run HLVS matches has been extended to collect necessary data from the season's games. The data that resulted from that has been made available online. Analyzing the data was done by extending the `SoccerAnalyzer` Python framework from Pereira et al. [1]. This tool already provided some metrics; however, others have been proposed in this work. A small subset of the proposed metrics have been implemented to be evaluated. The evaluation tried to show the meaningfulness of the metrics by measuring the statistical significance. A quantitative validation of this could not be provided. Moreover, a possible predictive capability of the metrics has not been measured. Therefore, the impact of this thesis is reduced, still the approach can be used by RoboCup teams to develop and extend tools and gain insight themselves.

8. Future Work

Many questions are still unanswered and need more work to be resolved. In other cases, simplification have been made, but components need to be completed. This chapter lists some ongoing work.

Regarding the data structure, the team-communication, and player collision have not been modeled. Collecting collisions from the supervisor controller may be computationally expensive, this needs to be investigated. The Webots supervisor API also provides a feature called pose tracking to reduce querying overhead. Unfortunately, this did not work in versions used by this thesis, but in newer Webots versions, this should be fixed. The data collection interval for subsampling has not been empirically tested. All of this may impact the runtime performance of the automatic referee, this needs to be investigated. The impact of the data collection extension in the automatic referee on correctness has not been investigated.

8.0.1. Dimensions and Aggregation

Almost all metrics can be filtered down or aggregated over a range of dimensions. However, this has not been implemented. In a match, we can filter metrics by spacial or temporal relations. For example, a user wants to know the distance walked by a specific robot in the second halftime in the own side; therefore as a time frame, we select the second half, we select the own field half and the player we want. All dimensions and their subtypes can be seen in table 8.1.

Additionally, for many data fields and metrics, it is useful, to gather them in an aggregated way instead of raw values. Raw values can be a boolean, string, integer, floating-point value or a vector, matrix, or list. Whereas an aggregated value can be a minimum, maximum, average, median, count, or trajectory of a collection of values or a frequency of or meantime between events.

Progress in Team's Performance

Once the significance of metrics has been determined, another set of test games will be simulated, this time using different versions of the same team's software. The goal of this experiment is to find out, whether the analysis of the recordings can show progress between the software versions. - correlation: metric <-> shot goals then predictive capability - applicability of sim in real world

8.0.2. Usability

web platform

Dimension	Subtype	Description
Matches	Season	Matches belonging to one season
	Set of matches	Custom selection of matches
Time frame	Match	Time frame of a whole match (incl. extension and penalty shootout)
	Halftime	Time frame of one halftime
	Time slice	Custom time frame (optionally depends on game event)
Time step		Discrete step in time of soccer simulation or recording
Area	Field half	Area of a soccer field half
	Area	Custom area on the field (optionally moving with an object)
Team		Team consisting out of multiple players
Player		Robot playing on the soccer field
Object	Ball	Ball object from the game
	Goalpost	A goalpost object from the game

Table 8.1.: Description of possible dimensions

Assuming, reliable metrics could be extracted from simulated game recordings, this data could further be used for manual or automated tests, by running simulated games and quantitatively comparing new player strategies and algorithms to previous versions. This could even be included in the workflow of a *Continuous Integration* (CI) pipeline. Another conceivable application of the resulting data is in the development of machine learning and, especially, reinforcement learning models.

Bibliography

[1] F. N. A. Pereira, M. F. B. Soares, O. R. C. ao, T. T. Alves, T. H. R. P. Gonçalves, J. R. da Silva, T. I. Ren, P. S. G. de Mattos Neto, and E. N. S. Barros, "A Library and Web Platform for RoboCup Soccer Matches Data Analysis," in *RoboCup 2022 Symposium*. Springer, 2023, pp. 177–189.

[2] The pandas development team, "pandas-dev/pandas: Pandas," 2023. [Online]. Available: https://zenodo.org/record/7857418

[3] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56–61.

[4] RoboCup Humanoid League TC, "RoboCup Soccer Humanoid League Laws of the Game 2021/2022," accessed: 2023-05-17. [Online]. Available: https://humanoid.robocup.org/wp-content/uploads/RC-HL-2022-Rules-3.pdf

[5] O. Michel, " Cyberbotics Ltd. Webots™: professional mobile robot simulation ," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, p. 5, 2004.

[6] R. Smith *et al.*, "Open dynamics engine," 2007.

[7] A. L. Ames, D. R. Nadeau, and J. L. Moreland, *The VRML 2.0 sourcebook*. John Wiley & Sons, Inc., 1997.

[8] R. Carey and G. Bell, *The annotated VRML 2.0 reference manual*. Addison-Wesley Longman Ltd., 1997.

[9] Cyberbotics Ltd., "Webots Reference Manual," accessed: 2023-05-17. [Online]. Available: https://cyberbotics.com/doc/reference

[10] R. Smith, "Open Dynamics Engine Manual," accessed: 2023-05-17. [Online]. Available: https://ode.org/wiki/index.php/Manual

[11] S. Stelter, M. Bestmann, N. Hendrich, and J. Zhang, " Fast and Reliable Stand-Up Motions for Humanoid Robots Using Spline Interpolation and Parameter Optimization ," in *IEEE ICAR*, 12 2021.

[12] C. Reep and B. Benjamin, "Skill and Chance in Association Football," *Journal of the Royal Statistical Society. Series A (General)*, vol. 131, no. 4, pp. 581–585, 1968.

[13] E. Morgulev, O. H. Azar, and R. Lidor, "Sports Analytics and the Big-Data Era," *International Journal of Data Science and Analytics*, vol. 5, no. 4, pp. 213–222, 2018.

[14] L. Pappalardo, P. Cintia, A. Rossi, E. Massucco, P. Ferragina, D. Pedreschi, and F. Giannotti, "A Public Data Set of Spatio-Temporal Match Events in Soccer Competitions," *Scientific data*, vol. 6, no. 1, pp. 1–15, 2019.

[15] S. A. Pettersen, D. Johansen, H. Johansen, V. Berg-Johansen, V. R. Gaddam, A. Mortensen, R. Langseth, C. Griwodz, H. K. Stensland, and P. Halvorsen, "Soccer Video and Player Position Dataset," in *Proceedings of the 5th ACM Multimedia Systems Conference*, 2014, pp. 18–23.

[16] J. Kern, T. Lober, J. Hermsdörfer, and S. Endo, " A Neural Network for the Detection of Soccer Headers From Wearable Sensor Data ," *Scientific Reports*, vol. 12, no. 1, pp. 1–12, 2022.

[17] Fédération internationale de Football Association (FIFA), "Semi-automated Offside Technology to Be Used at FIFA World Cup 2022 (TM)," accessed: 2023-05-19. [Online]. Available: https://www.fifa.com/technical/media-releases/semi-automated-offside-technology-to-be-used-at-fifa-world-cup-2022-tm

[18] H. Mellmann, B. Schlotter, and P. Strobel, "Toward Data Driven Development in RoboCup," in *Robot World Cup*. Springer, 2019, pp. 176–188.

[19] RoboCup Standard Platform League Technical Committee, "RoboCup Standard Platform League (NAO) Rule Book," accessed: 2023-05-19. [Online]. Available: https://spl.robocup.org/wp-content/uploads/SPL-Rules-2022.pdf#subsection.B.4

[20] B-Human, "B-Human's Video Analysis App," accessed: 2023-05-19. [Online]. Available: https://github.com/bhuman/VideoAnalysis

[21] RoboCup Small Size League, "RoboCup Small Size League Game Log Archive," accessed: 2023-05-19. [Online]. Available: https://ssl.robocup.org/game-logs/

[22] RoboCup Soccer Simulation League, "RoboCup Soccer Simulation League 2D Game Log Archive," accessed: 2023-05-19. [Online]. Available: https://archive.robocup.info/Soccer/Simulation/2D/

[23] RoboCup Soccer Simulation League, "RoboCup Soccer Simulation League 3D Game Log Archive," accessed: 2023-05-19. [Online]. Available: https://archive.robocup.info/Soccer/Simulation/3D/

[24] Python Software Foundation, "Python 3.10.11 documentation – data classes," accessed: 2023-05-07. [Online]. Available: https://docs.python.org/3.10/library/dataclasses.html

[25] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," RFC 4180, Oct. 2005. [Online]. Available: https://www.rfc-editor.org/info/rfc4180

[26] T. Bray, "RFC 8259: The JavaScript object notation (JSON) data interchange format," 2017.

[27] Ecma International, "ECMA-404 – The JSON Data Interchange Syntax 2nd Edition," December 2017, accessed: 2023-05-12. [Online]. Available: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf

[28] W3C - World Wide Web Consortium, "Extensible Markup Language (XML) 1.1 (Second Edition)," September 2006, accessed: 2023-05-12. [Online]. Available: https://www.w3.org/TR/2006/REC-xml11-20060816/

[29] Ecma International, "ECMA-376 – Office Open XML file formats 5th edition," December 2021, accessed: 2023-05-12. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-376/

[30] Python Software Foundation, "Python 3.10.11 documentation – pickle – Python object serialization," accessed: 2023-05-12. [Online]. Available: https://docs.python.org/3.10/library/pickle.html

[31] HDF Group, "HDF5 Documentation," accessed: 2023-05-12. [Online]. Available: https://portal.hdfgroup.org/display/HDF5/HDF5

[32] W. McKinney, "Feather GitHub Repository," accessed: 2023-05-12. [Online]. Available: https://github.com/wesm/feather

[33] Apache Software Foundation, "Apache Parquet Documentation," accessed: 2023-05-12. [Online]. Available: https://parquet.apache.org/docs/

[34] Apache Software Foundation, "Apache ORC," accessed: 2023-05-12. [Online]. Available: https://orc.apache.org/

[35] StataCorp LLC, "File formats .dta – Description of .dta file format 115," accessed: 2023-05-12. [Online]. Available: https://www.stata.com/help.cgi?dta_115

[36] H. Krekel, B. Oliveira, R. Pfannschmidt, F. Bruynooghe, B. Laugher, and F. Bruhin, "pytest 6.2.5," 2004, accessed: 2023-05-07. [Online]. Available: https://github.com/pytest-dev/pytest

[37] T. Moulard, "Coordinate Frames for Humanoid Robots," accessed: 2023-05-19. [Online]. Available: https://www.ros.org/reps/rep-0120.html

[38] Google LLC, "Protocol Buffers Documentation," accessed: 2023-05-16. [Online]. Available: https://protobuf.dev/

[39] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

Appendices

A. Data Structure Complete UML Class Diagram

B. Pandas File Format Benchmarking Results

File Format	Write Duration [s]	Read Duration [s]	File Size [B]
csv	10.286947048036382	2.1948208790272474	180942721
csv	10.023249280173331	1.9335246719419956	180942721
csv	9.98883936717175	1.9953098089899868	180942721
csv	10.192911381833255	1.9325142491143197	180942721
csv	10.102485560113564	2.1530611768830568	180942721
csv	10.125943074934185	1.9474970770534128	180942721
csv	10.110906084999442	1.9553595760371536	180942721
csv	10.215702585177496	2.0022814080584794	180942721
csv	9.980430486844853	2.006097888108343	180942721
csv	10.14780302811414	1.992749661207199	180942721
json	3.193962930003181	8.32795168994926	345991506
json	3.2353430609218776	8.328224291093647	345991506
json	3.1948386910371482	8.449956130003557	345991506
json	3.241574323968962	7.849857060937211	345991506
json	3.0512268820311874	7.380913545144722	345991506
json	2.9032476018182933	7.860276029910892	345991506
json	2.902596662985161	7.613485580077395	345991506
json	3.475083524128422	7.3509946160484105	345991506
json	2.928332105046138	7.346875760005787	345991506
json	2.873381247976795	7.712733270134777	345991506
xml	59.14449173491448	16139.828588937875	1961486431
excel	279.1365820460487	148.92179826204665	160116324
pickle	0.5484468378126621	0.3314504351001233	179257732
pickle	0.568036031909287	0.3469477309845388	179257732
pickle	0.5616264417767525	0.3337416781578213	179257732
pickle	0.66393374488689	0.10860279784537852	179257732
pickle	0.8907915260642767	0.11045060004107654	179257732
pickle	0.8373289639130235	0.35091720009222627	179257732
pickle	1.0111449100077152	0.09267587587237358	179257732

File Format	Write Duration [s]	Read Duration [s]	File Size [B]
pickle	0.6130795690696687	0.1128802530001849	179257732
pickle	0.670917300041765	0.11561686987988651	179257732
pickle	0.6322005169931799	0.1221329199615866	179257732
hdf5	0.600272913929075	0.18587912898510695	180583292
hdf5	1.0059687909670174	0.4776888801716268	181274748
hdf5	0.9605527459643781	0.4562627640552819	181964156
hdf5	1.0132272210903466	0.5145521408412606	182424004
hdf5	0.9457813289482147	0.48141981987282634	182883852
hdf5	0.6929234419949353	0.44616533489897847	183343700
hdf5	1.2064036149531603	0.44097292609512806	183803548
hdf5	1.0340712869074196	0.4890656170900911	184263396
hdf5	0.8493852431420237	0.45685938210226595	184723244
hdf5	0.6657042820006609	0.45922697708010674	185183092
feather	0.26414139894768596	0.202806465793401	65465578
feather	0.27630632114596665	0.099273368017748	65465578
feather	0.2611083360388875	0.09611283615231514	65465578
feather	0.27651436091400683	0.18111494183540344	65465578
feather	0.28596399701200426	0.10481450916267931	65465578
feather	0.28555155522190034	0.10845902794972062	65465578
feather	0.26125139417126775	0.11720397416502237	65465578
feather	0.26060459204018116	0.1920069910120219	65465578
feather	0.3067712539341301	0.19267511297948658	65465578
feather	0.41317550209350884	0.1776385388802737	65465578
parquet	3.0819843269418925	0.17009179899469018	77628286
parquet	2.9701130150351673	0.1557669979520142	77628286
parquet	2.988494185032323	0.13251241808757186	77628286
parquet	3.2292723490390927	0.11247510300017893	77628286
parquet	3.0991539279930294	0.11964415619149804	77628286
parquet	3.0169663338456303	0.13040212308987975	77628286
parquet	3.0957914078608155	0.13498451886698604	77628286
parquet	3.106660140911117	0.12388993101194501	77628286
parquet	2.471850768197328	0.23204648192040622	77628286
parquet	2.99633996700868	0.11567287403158844	77628286
orc	3.808645844226703	0.3774917018599808	71713603
orc	3.653143160045147	0.3790003480389714	71713603
orc	3.650477315997705	0.3713955990970135	71713603
orc	3.356333422008902	0.3952555258292705	71713603
orc	4.119027923094109	0.3604830331169069	71713603
orc	2.7888031071051955	0.5332794091664255	71713603
orc	3.249960631830618	0.4005263508297503	71713603
orc	3.6549363310914487	0.3639301718212664	71713603

File Format	Write Duration [s]	Read Duration [s]	File Size [B]
orc	3.7039073570631444	0.5615057570394129	71713603
orc	3.5258315990213305	0.37877844204194844	71713603
stata	9.12018607603386	6.56543051591143	172046339
stata	8.792505032150075	6.648476668866351	172046339
stata	8.715335379820317	6.688728197943419	172046339
stata	8.809489796170965	6.231604540953413	172046339
stata	8.483828759053722	6.528916190145537	172046339
stata	8.606506797950715	6.384347824146971	172046339
stata	8.760864146985114	6.643513730028644	172046339
stata	8.79665143089369	7.027401988161728	172046339
stata	8.529906493844464	6.712406593840569	172046339
stata	8.70403677993454	6.690408627036959	172046339

Table 2.: All Measurements of the Pandas file format benchmarking tests. Each row represents a single test run, producing measurements for the write and read durations (in seconds) and the size of exported files (in Bytes).

C. Additional plots from repeated experiments

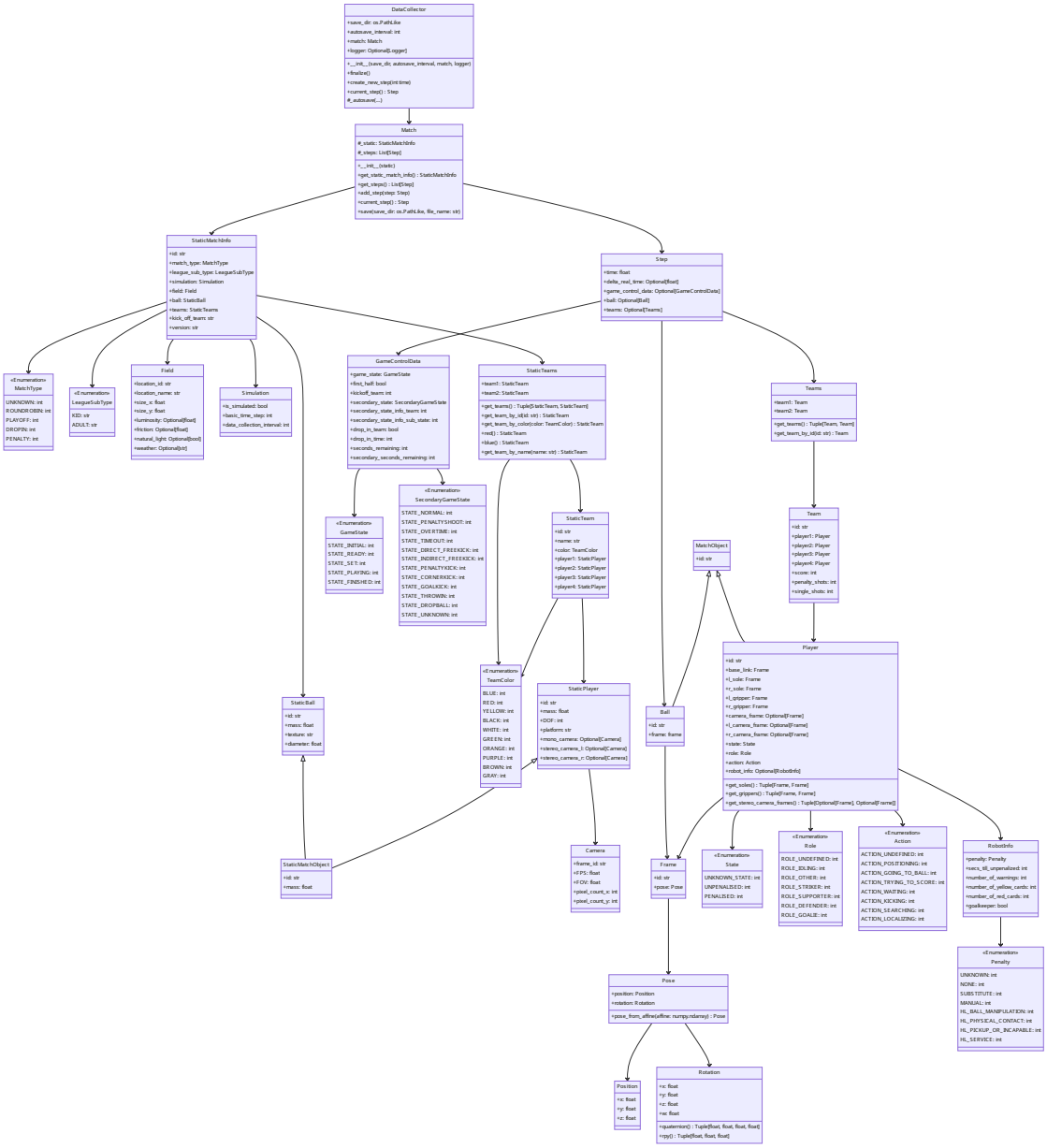

Figure 1.: Complete UML class diagram of the `data_collection` software module

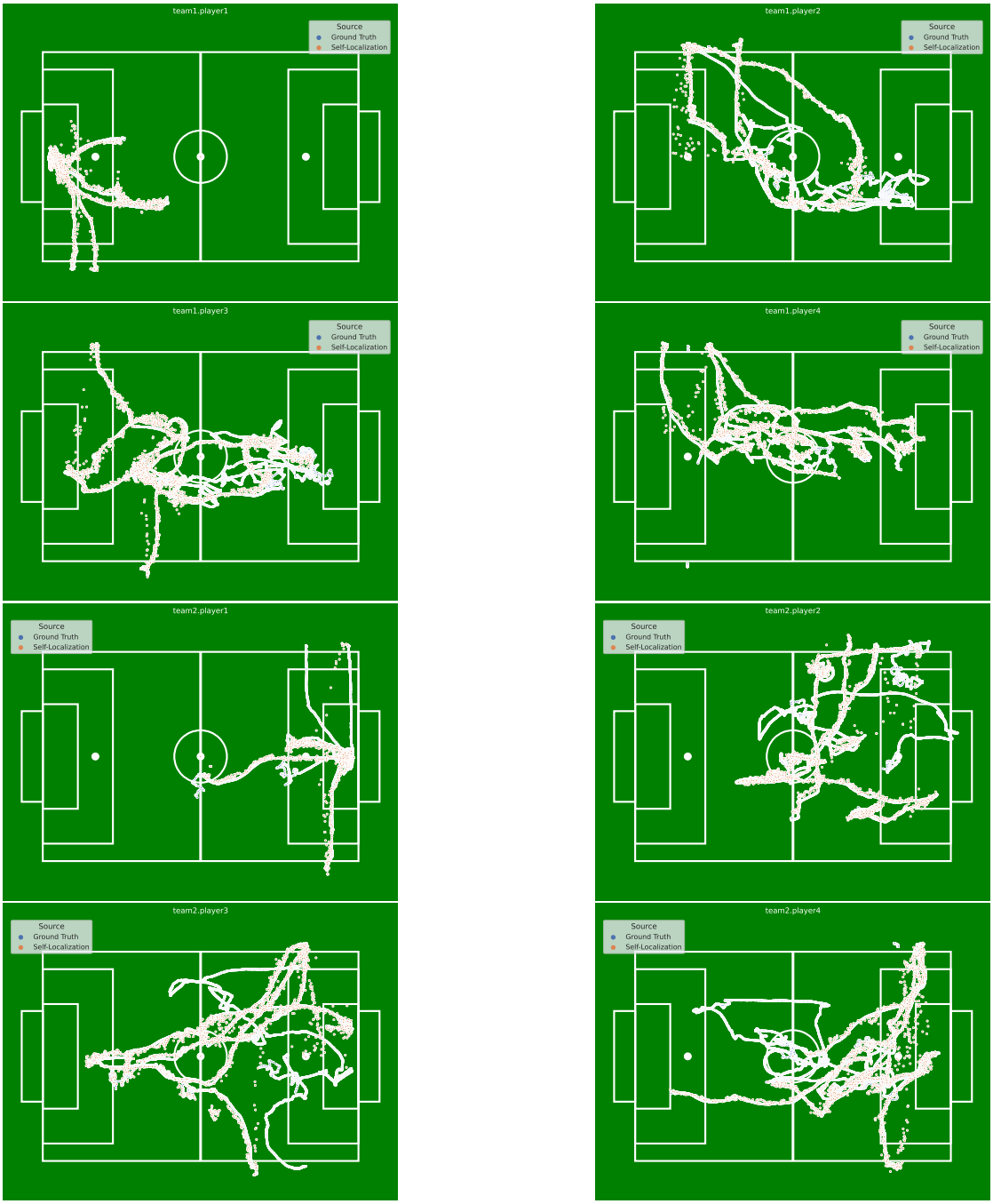

Figure 2.: Experiment Repeated 01 The self-localization of all players

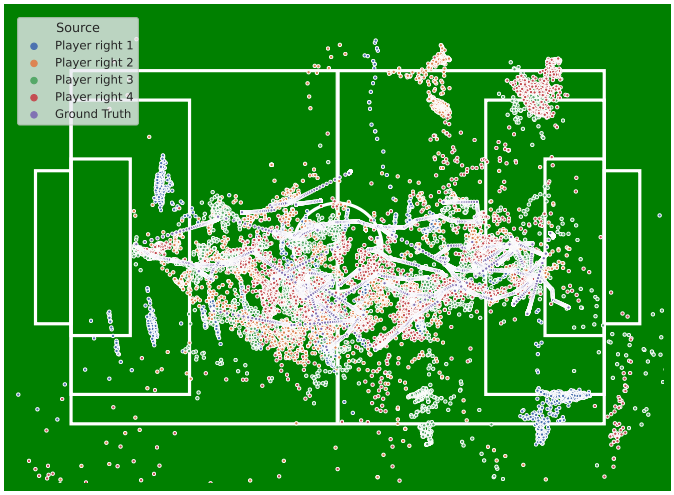

Figure 3.: Experiment Repeat 01 Absolute error of the ball localization on the field

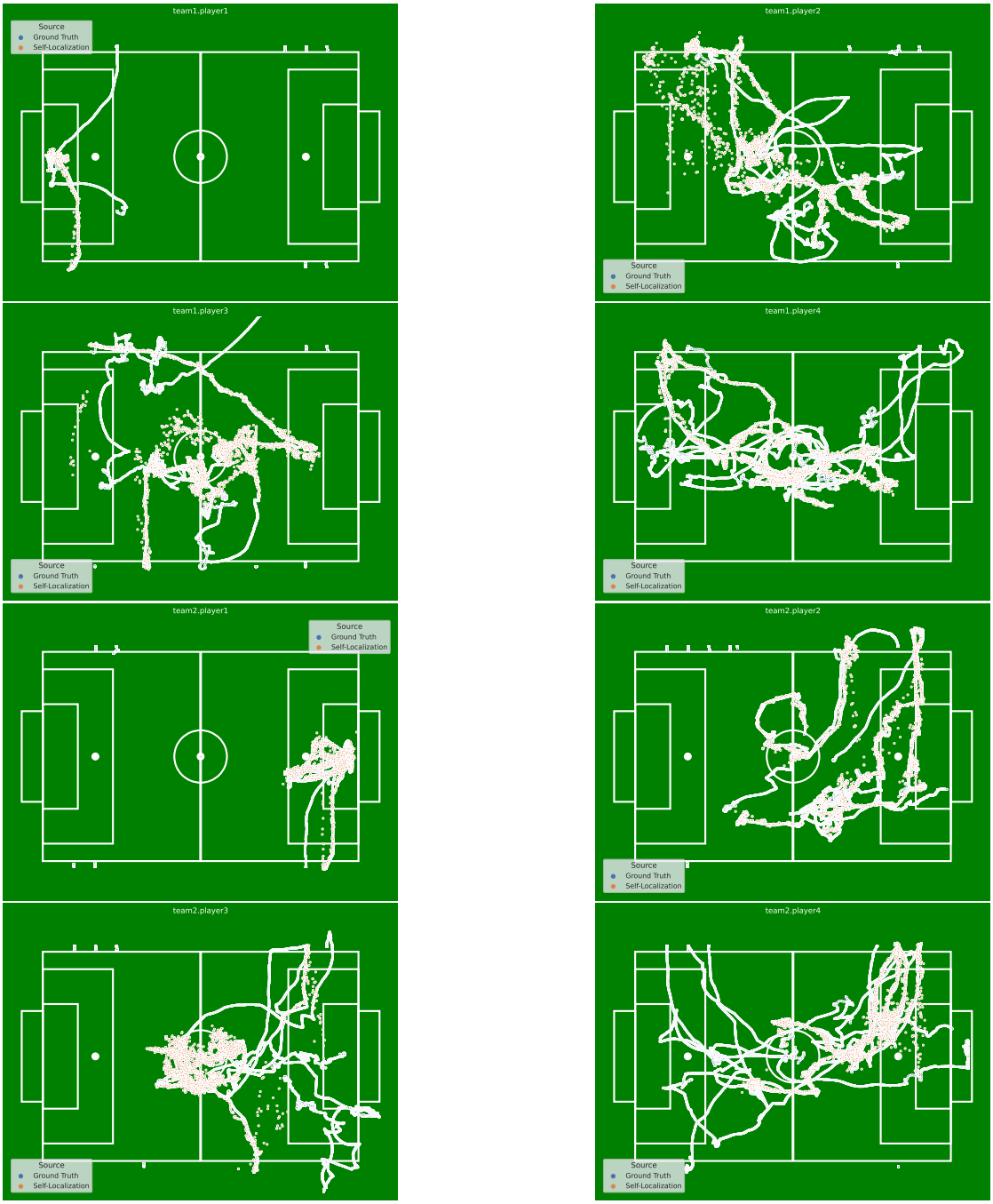

Figure 4.: Experiment Repeated 02 The self-localization of all players

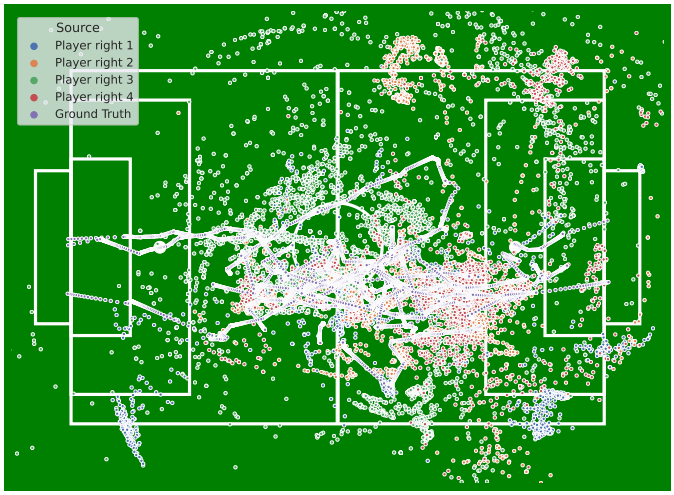

Figure 5.: Experiment Repeat 02 Absolute error of the ball localization on the field

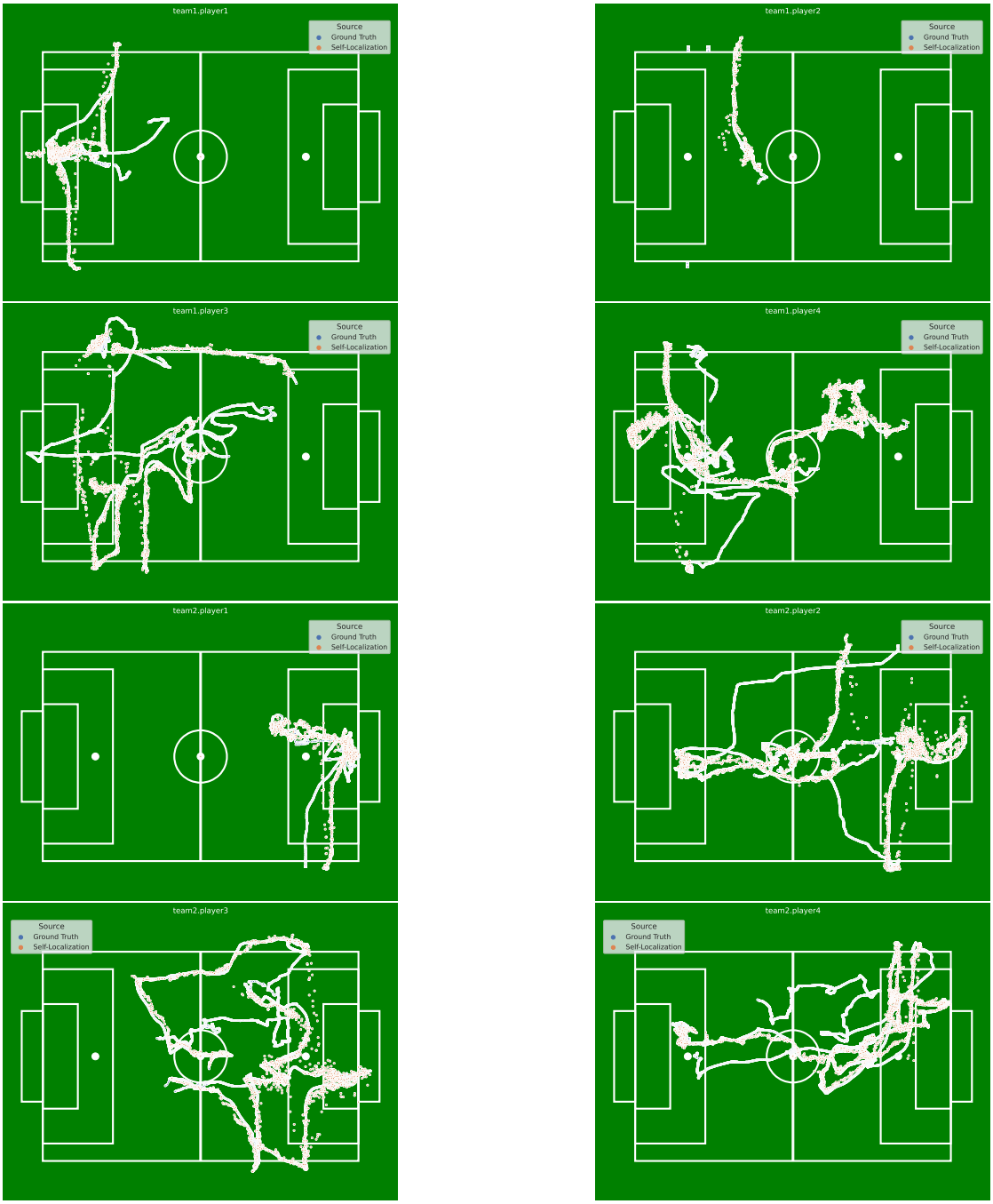

Figure 6.: Experiment Repeated 03 The self-localization of all players

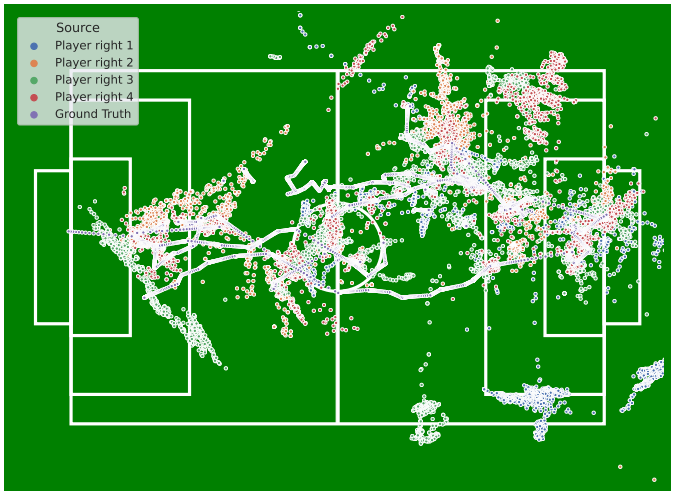

Figure 7.: Experiment Repeat 03 Absolute error of the ball localization on the field

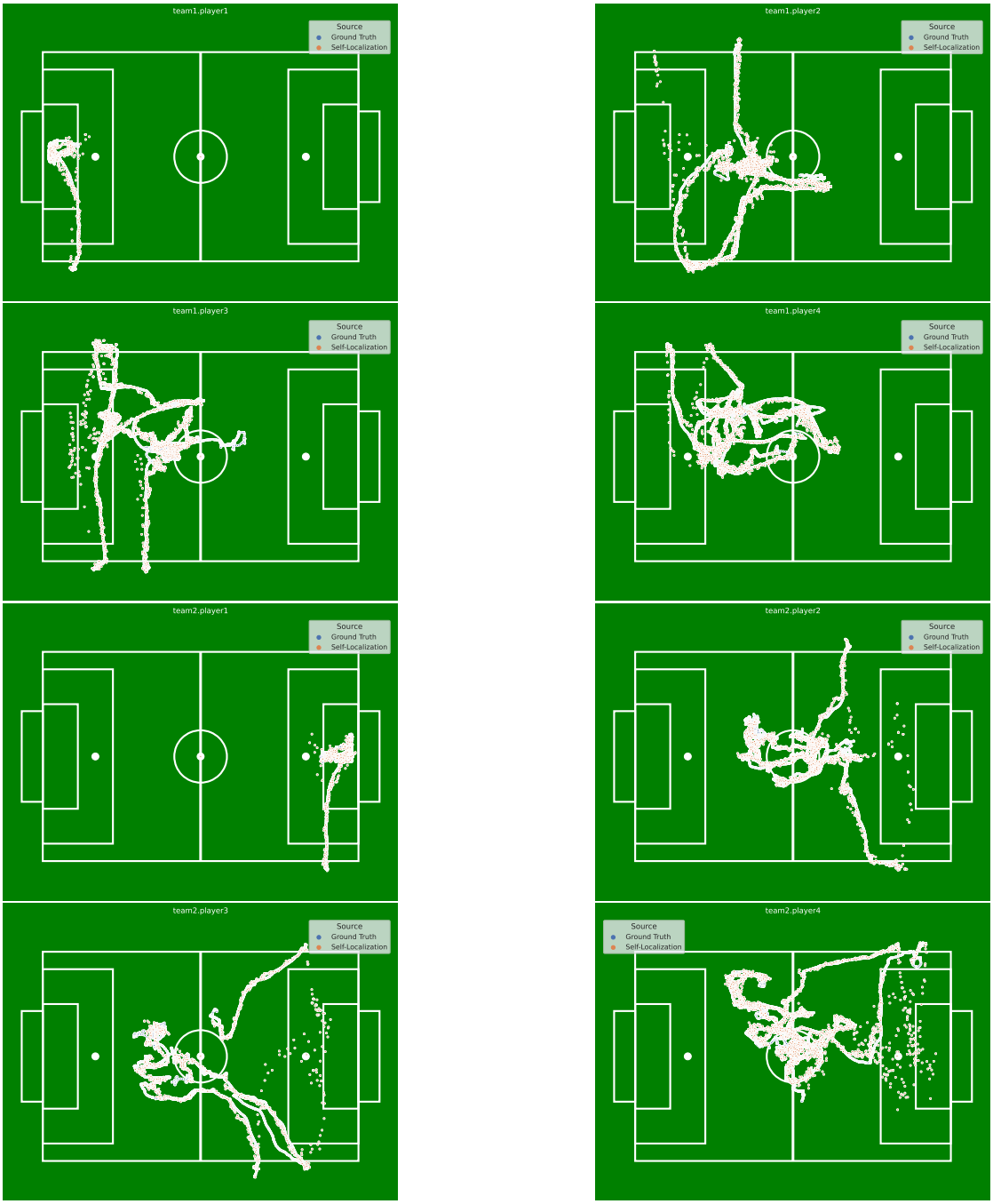

Figure 8.: Experiment Repeated 04 The self-localization of all players

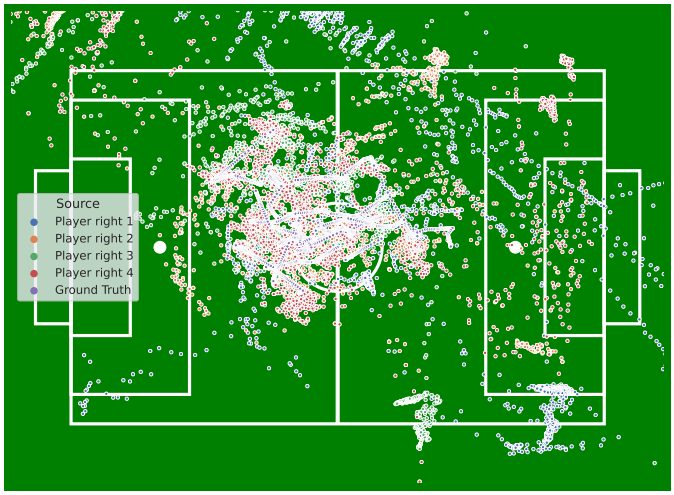

Figure 9.: Experiment Repeat 04 Absolute error of the ball localization on the field

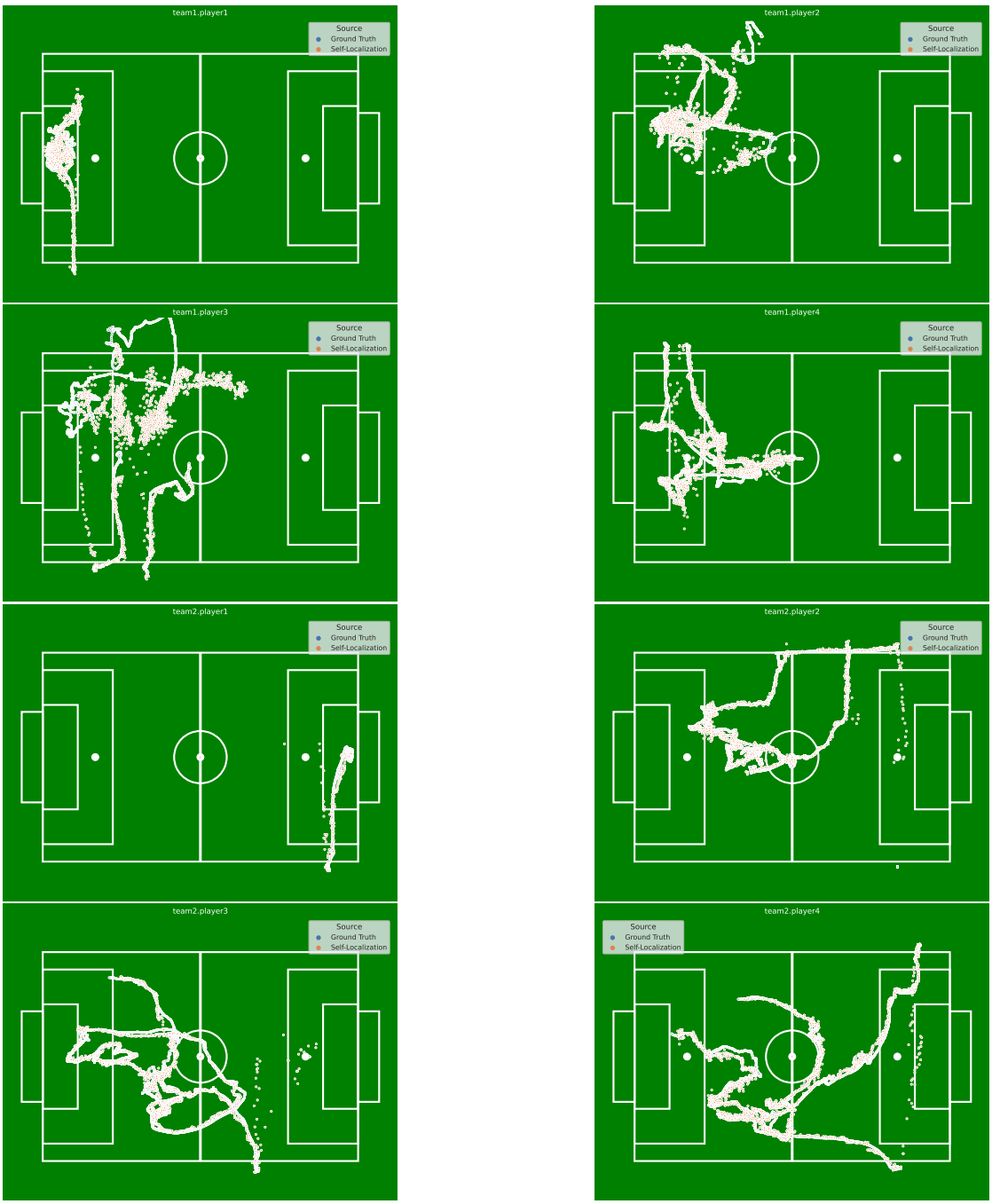

Figure 10.: Experiment Repeated 05 The self-localization of all players

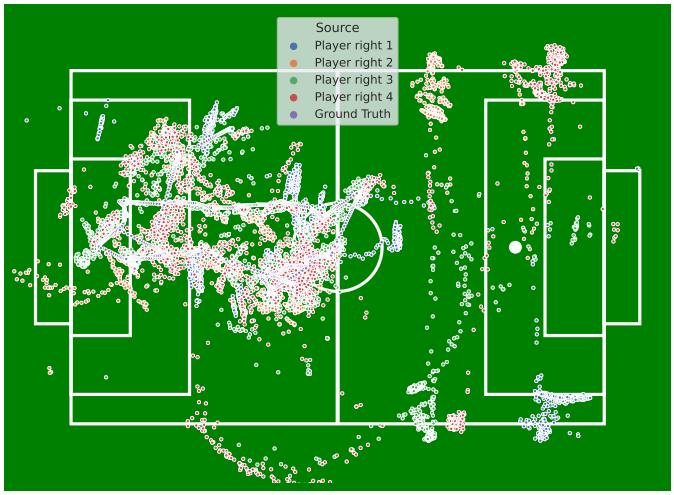

Figure 11.: Experiment Repeat 05 Absolute error of the ball localization on the field

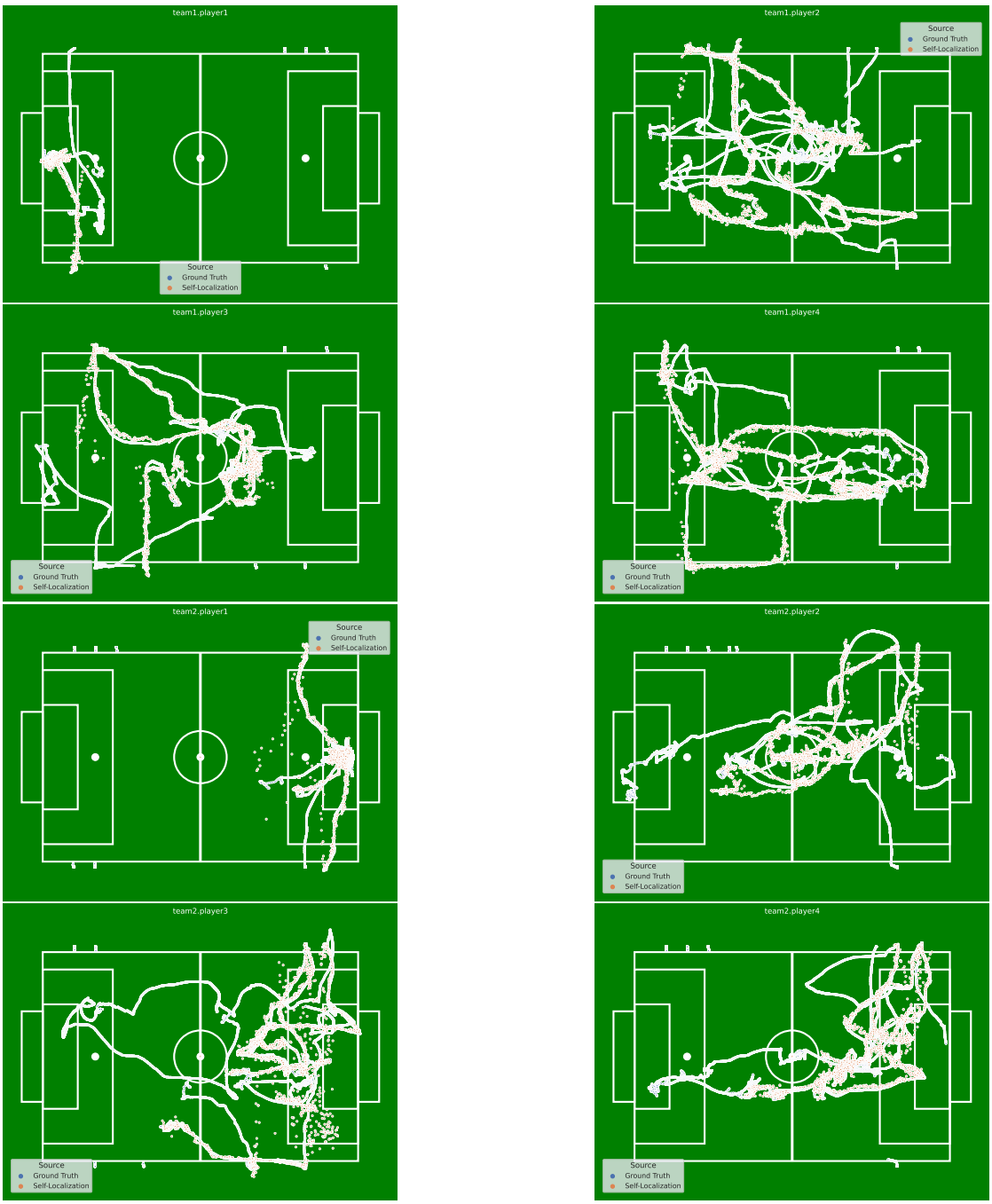

Figure 12.: Experiment Repeated 06 The self-localization of all players

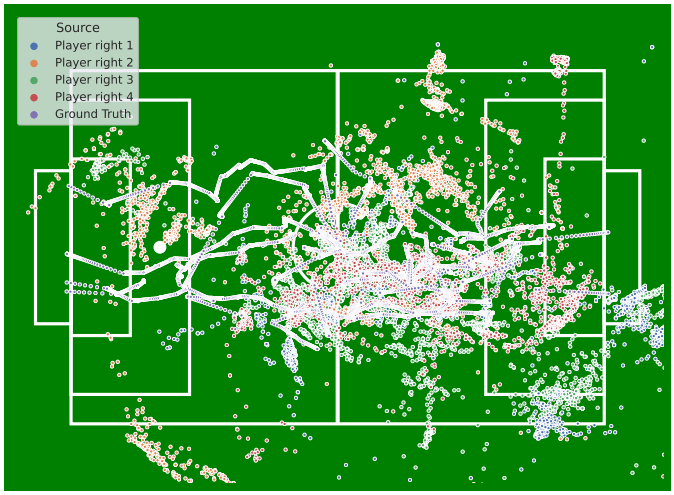

Figure 13.: Experiment Repeat 06 Absolute error of the ball localization on the field

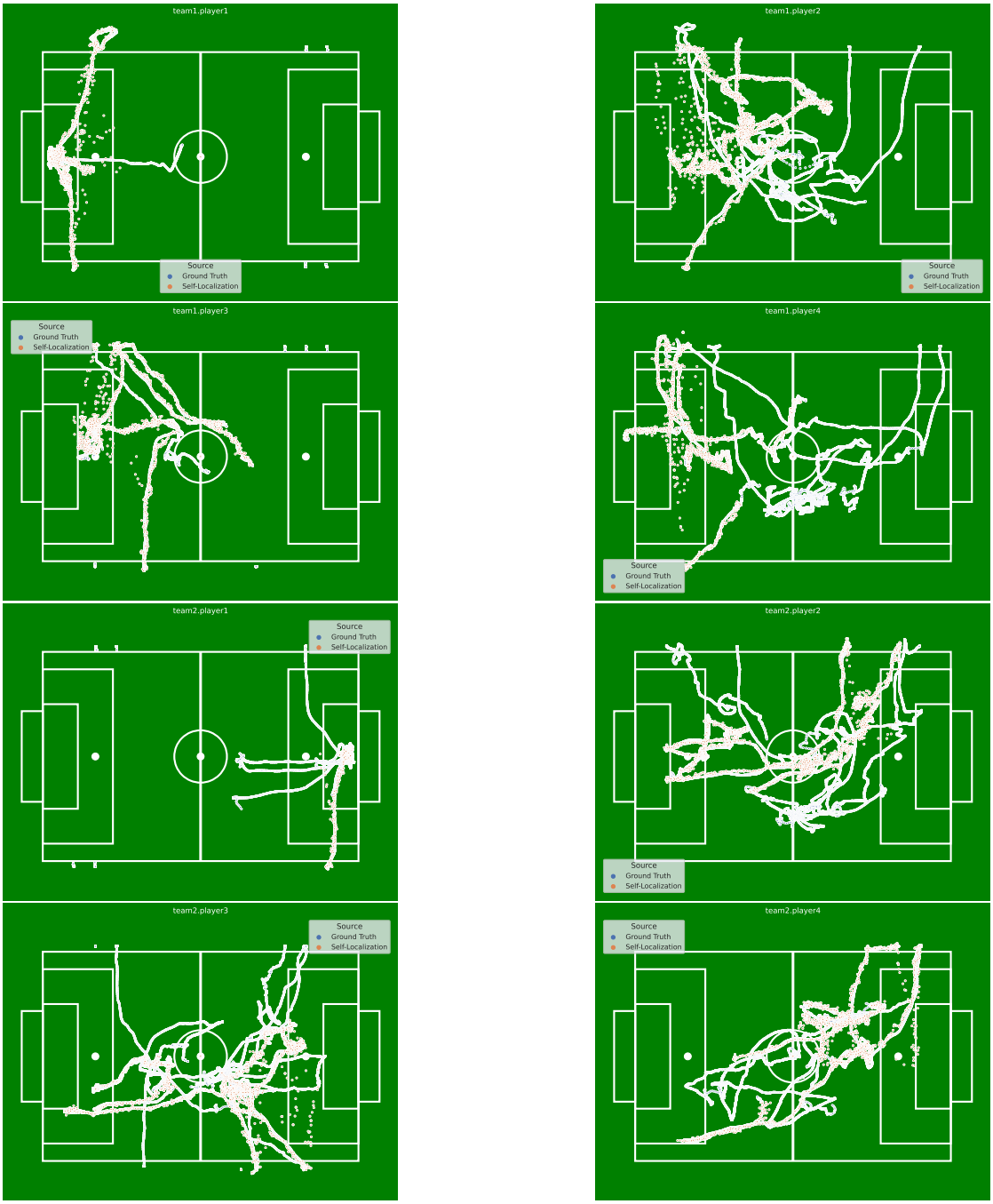

Figure 14.: Experiment Repeated 07 The self-localization of all players

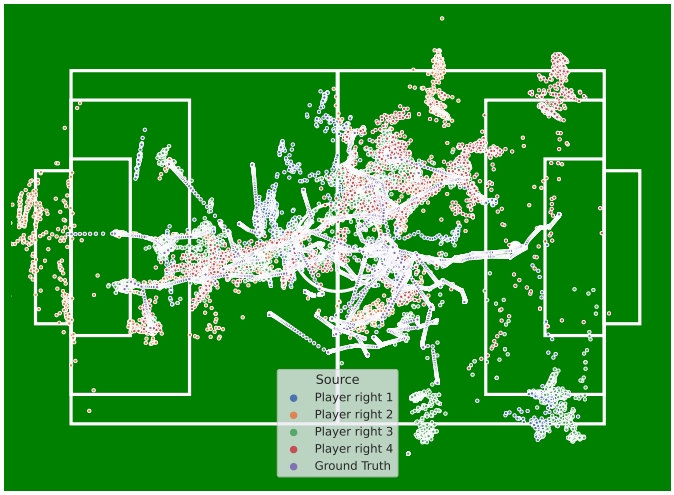

Figure 15.: Experiment Repeat 07 Absolute error of the ball localization on the field

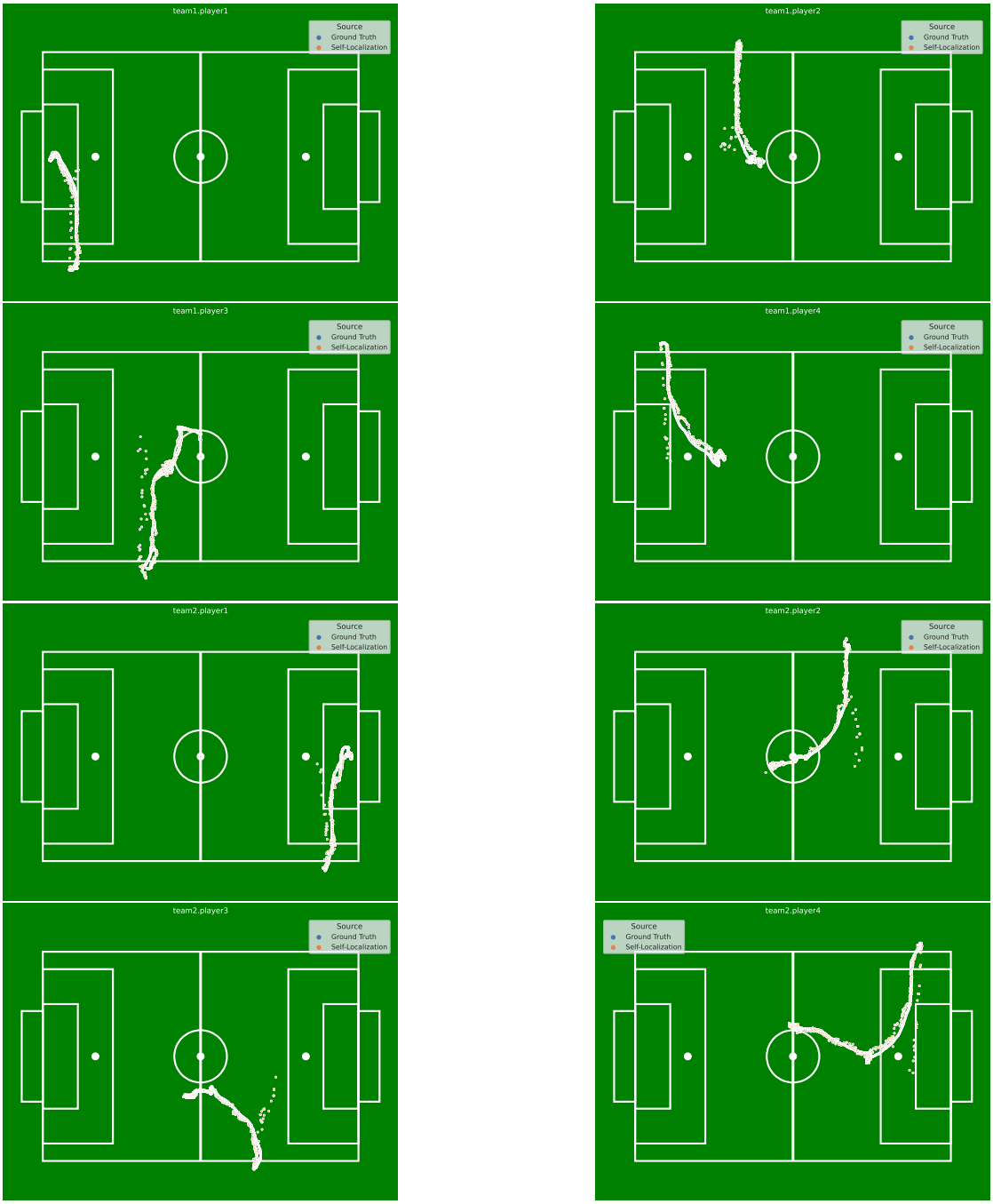

Figure 16.: Experiment Repeated 08 The self-localization of all players

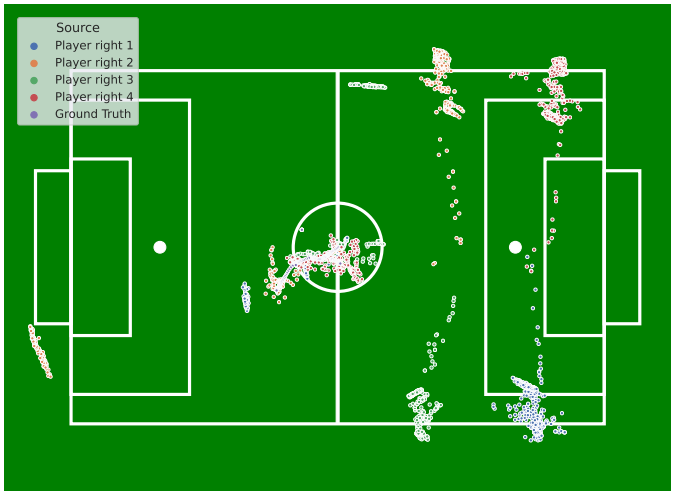

Figure 17.: Experiment Repeat 08 Absolute error of the ball localization on the field

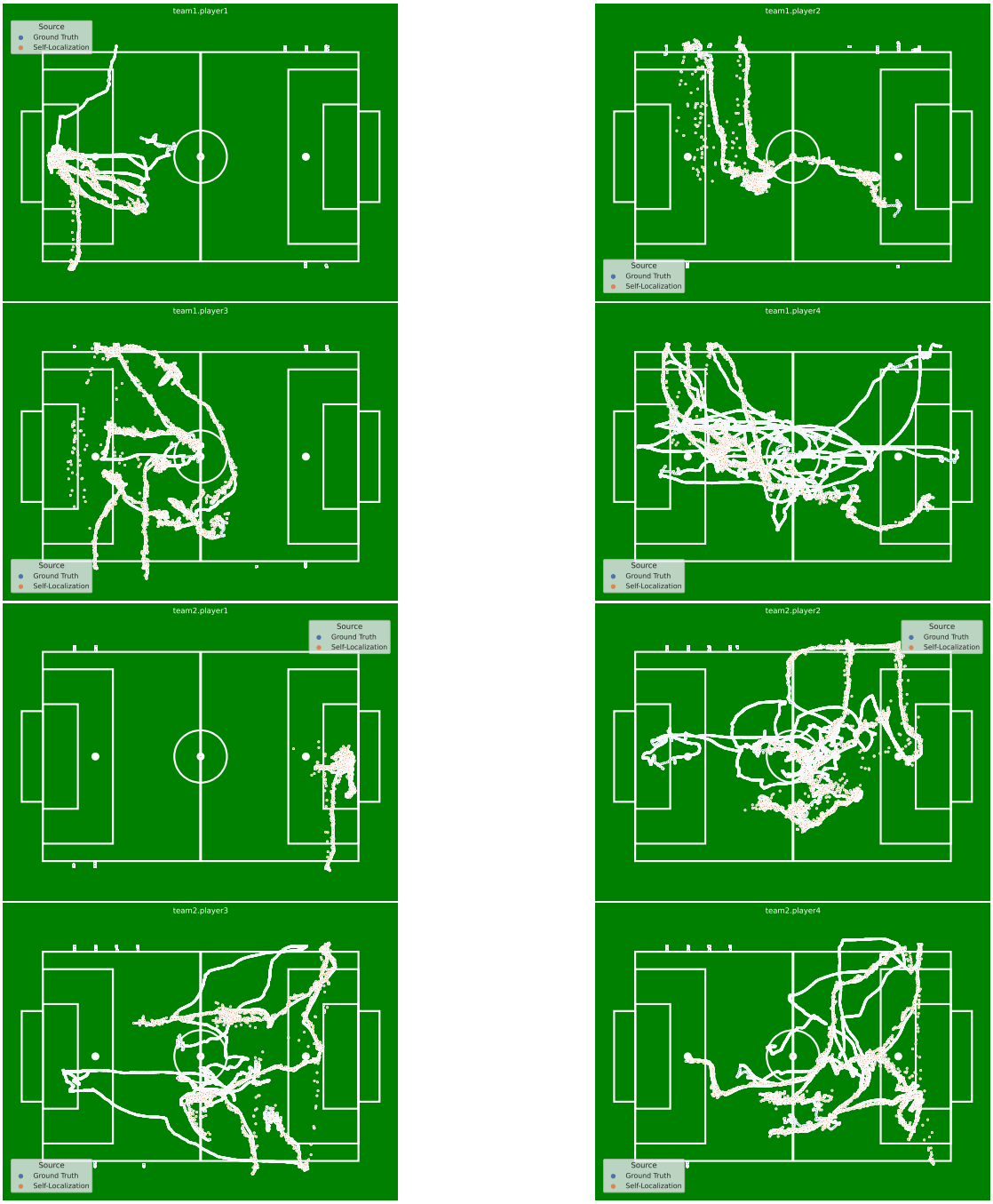

Figure 18.: Experiment Repeated 09 The self-localization of all players

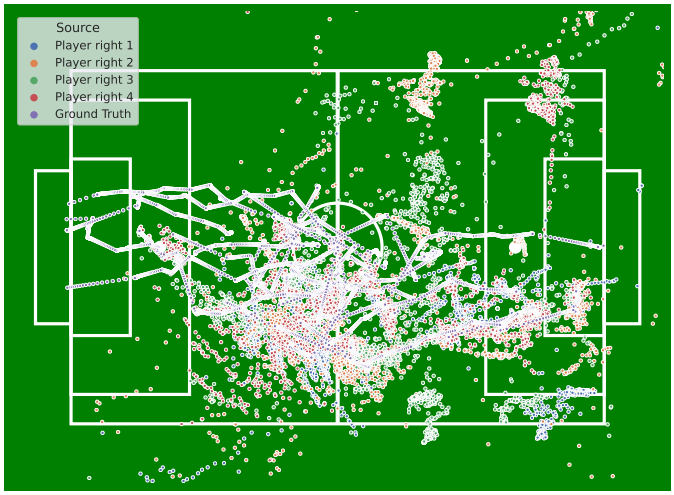

Figure 19.: Experiment Repeat 09 Absolute error of the ball localization on the field

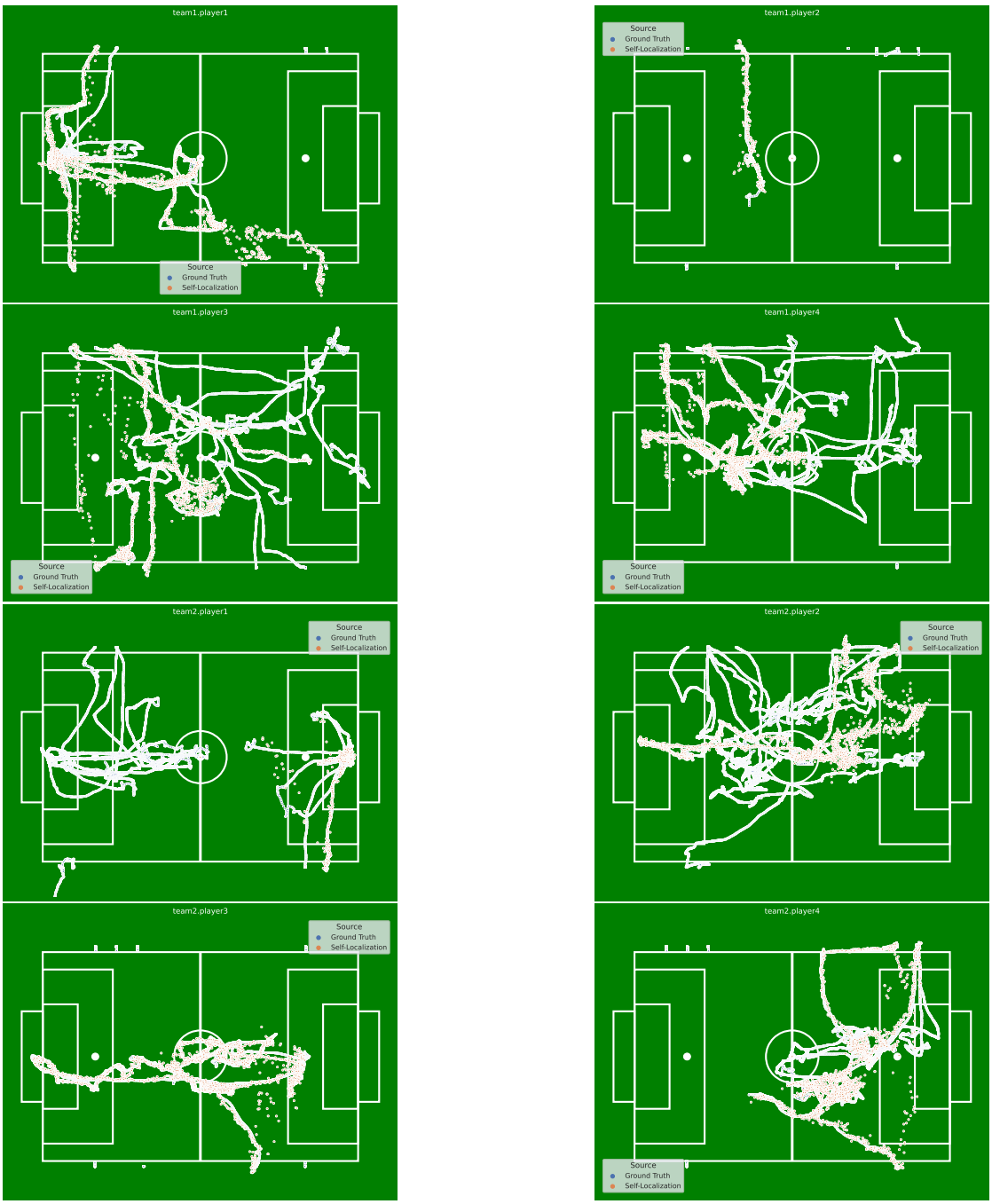

Figure 20.: Experiment Repeated 10 The self-localization of all players

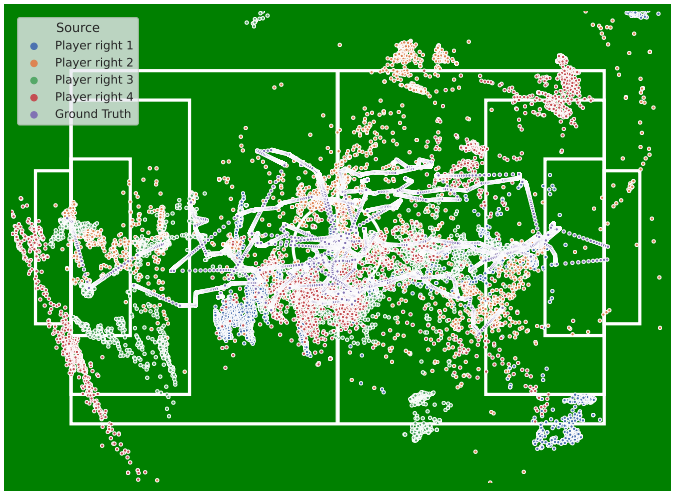

Figure 21.: Experiment Repeat 10 Absolute error of the ball localization on the field

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der elektronischen Abgabe entspricht.

Hamburg, den 19. Mai 2023

J. Gutsche
Jan Gutsche

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 19. Mai 2023

J. Gutsche
Jan Gutsche