Software Survey 2026

Team name

RobôCIn

Which division(s) are you applying for? If your used software differs between divisions, please fill out the survey once per division.

Large Size (height < 190 cm, weight < 80 kg)

Is your software fully or partially OpenSource? If so, where can it be found?

We are using the code provided by Booster itself (https://github.com/BoosterRobotics/robocup_demo) as our base software, and we are making changes based on it. In its current state, we haven't made any very significant changes yet, but we plan to release a more robust version as soon as we have one.

Are you using any software developed by other teams? If so, list every component that you are reusing and the team that originally developed it.

In October 2025, we went to Sao Paulo where we met with other teams from the Brazilian Humanoid League. These teams helped us correct some errors we had in our code. We had some basic problems that didn't seem to have a clear cause, but after the teams' support, we were able to run them correctly. All of them were using the RoboCup_Demo codebase mentioned earlier, with some adjustments, and after this event, we used the RoboFEI codebase to fix our errors.

Are you using any datasets in your research? If you are using your own datasets, are they public?

No, we are not using it.

Please list the scientific publications your team has made since the last application to RoboCup (or if not applicable in the last 2 years).

Here is a list of some of the undergraduate theses, conference and journal publications, and preprints published between 2024 and 2025 related to RobôCIn as a whole, since this is our first year with the humanoid category:

Efficient Lines Detection for Robot Soccer, https://arxiv.org/abs/2507.19469
SPO: Single Pass Optimization for Soccer Simulation 2D, https://ieeexplore.ieee.org/abstract/document/10786402
Improving Inertial Odometry Through Particle Swarm Optimization in the Robocup Small Size League, https://link.springer.com/chapter/10.1007/978-3-031-55015-7_8
Reinforcement Learning-driven automatic calibration for color segmentation-based robot detection, https://ieeexplore.ieee.org/document/10786410
PilotStation: Estação de Controle de Múltiplos Drones Baseada em Web, https://repositorio.ufpe.br/bitstream/123456789/65552/3/TCC%20Tiago%20Henrique%20Rodrigues%20Pedrosa%20Gon%c3%a7alves.pdf
RobôCIn SSL-Unification: A Modular Software Architecture for Dynamic Multi-robots systems, https://link.springer.com/chapter/10.1007/978-3-031-55015-7_26
Architecture, Implementation and Performance Analysis of a Microservices-Based System for the RoboCup Small Size League, https://repositorio.ufpe.br/handle/123456789/58253
Analysis of the Introduction Trajectory Planning in Control Applied in Small Size League Robots, https://link.springer.com/chapter/10.1007/978-3-031-85859-8_12

Are there any other contributions you would like to share with the RoboCup community?

All the contributions we have as a team are detailed here and in other deliverables, such as on the website and in the TDP. We hope to have much more to contribute in the future, especially to the Humanoid League.

Which approach are you using to generate the robot walking motion?

To generate the robot's walking motion, we utilize the high-level locomotion control API provided by the manufacturer, Booster Robotics. Our approach consists of calculating, within the decision node (brain), the desired velocity vector composed of linear velocities in the X and Y axes and the angular velocity Theta, subsequently sending it to the motion control board (booster_motion). In the implementation, this is achieved through the RobotClient::setVelocity method, which publishes a booster_msgs::CreateMoveMsg message to the /LocoApiTopicReq topic, allowing the Booster firmware to abstract the inverse kinematics and the omnidirectional walking gait generation.

Which approach are you using to generate other motions of the robot (e.g. kicking, standing up)?

For discrete and complex movements, we also utilize the proprietary API from Booster Robotics via Remote Procedure Calls (RPC) encapsulated in ROS 2 messages. For example, the robot performs the stand-up action by sending a specific command through the RobotClient::standUp method, which triggers the booster_msgs::CreateGetUpMsg() message, leaving trajectory generation and balance control to be managed internally by the manufacturer's software. Other movements, such as waving or head positioning, follow the same logic (e.g., RobotClient::waveHand and RobotClient::moveHead), where the decision node (brain) simply requests the action and the motion control board (booster_motion) executes it.

Do you have a kinematic or dynamic model of your robot? If so, how did you create it (e.g. measure physical robot, export from CAD model)?

Yes, we utilize a kinematic model of the robot. This model was created based on the official specifications and the CAD model of the Booster T1 provided by the manufacturer. It is primarily used within our vision system to perform Eye-to-Base coordinate transformations, allowing us to accurately map detected objects from the camera frame to the robot's Cartesian frame using extrinsic parameters defined in our configuration files. Regarding dynamics, we rely on the model embedded within the manufacturer's low-level control firmware to handle stability and locomotion.

What approaches are you using in your robot's visual perception?

The team employs YOLOv8, utilizing both detection and segmentation variants. This enables the robot to identify the ball, goalposts, field markings and other agents. Furthermore, to ensure real-time inference we utilize the NVIDIA TensorRT to optimize the model for our specific GPU making it fast enough to track moving objects during matches. This process involves converting YOLOv8 weights into TensorRT .engine files, which are optimized binary representations that maximize performance on the target hardware.

Are you planning with objects in Cartesian or image space? If you are using Cartesian space, how do you transform between the image space and cartesian space?

Our software performs planning in 3D Cartesian Space by transforming camera data into

spatial coordinates. This pipeline utilizes a ray-plane intersection method, based on the objects located on the ground plane (Z = 0). The process begins with back-projection, where we apply the intrinsic matrix to map pixel coordinates to a normalized 3D ray in the camera's system. Subsequently, we apply the extrinsic matrix to transform this ray into the robot's base frame. Finally, we calculate the intersection of the transformed ray with the ground plane to determine the object's precise coordinate in meters.

Do you have some form of active vision (i.e. moving the robots camera based on information known about the world)?

Yes, the robot features active vision. The ball tracking is managed by the Behavior Tree, which implements a closed-loop control adjusting the head's pitch and yaw to keep the ball centered or locate it when it is not visible. The execution of this task on high level decisions are dispatched to RobotClient::moveHead, which interfaces with the Booster SDK API to command neck joints to specific target angles.

What approach are you using to localize your robot?

To localize the robot, we utilize a visual localization approach based on field features, which integrates odometry with visual observations. The system combines odometry data, provided by the motion control board via the odometry (/odometer_state) topic, with the detection of field landmarks, such as lines and goal posts, to estimate the global position. In the implementation, the Locator class continuously updates the relative position using odometry and applies a pose correction mechanism whenever reliable visual features are detected, aligning the robot's pose with the field map to mitigate accumulated drift.

Is your team performing team communication? Which communication protocol are you using?

Yes, our team actively performs communication between robots using the UDP protocol. For internal strategic coordination, we rely on a custom structure named TeamCommunicationMsg rather than the standard SPL message. This structure encapsulates vital information including the robot's ID, its assigned role (such as striker or goalkeeper), the ball's visibility and relative coordinates, and the robot's tactical intention. The data exchange process is handled in the brain_communication.cpp file, where the function unicastCommunication serializes this custom message for transmission to the network, while spinCommunicationReceive listens for

incoming datagrams to update the collective World Model. While the SPLStandardMessage definition exists in our codebase, it is reserved exclusively for communication with the GameController and is not used for intra-team data exchange.

What approach are you using for navigation? Are you avoiding obstacles?

For navigation, we utilize an omnidirectional motion controller that executes velocity commands to drive the robot. The system is implemented in robot_client.cpp, specifically through the moveToPoseOnField method, which calculates the error vector between the robot's current position and the target pose to generate proportional velocities that minimize this error. Regarding obstacle avoidance, the current implementation does not employ complex path planning or active avoidance strategies like A* or potential fields. Instead, the robot follows a "greedy" approach, navigating in a straight line toward the goal. To maintain stability during navigation, if the target is located beyond a specific longRangeThreshold, the logic prioritizes rotating the robot to face the target before it begins to move forward.

How is the behavior of your robots structured? (e.g. Behaviour Trees)

The behavior of our robots is structured using Behavior Trees, implemented via the BehaviorTree.CPP library. The system's architecture is defined in the game.xml file, which serves as the main tree and manages transitions between high-level game states (such as Ready, Set, and Playing) based on input from the GameController. Within the Playing state, the logic branches into specific subtrees depending on the robot's assigned role, utilizing dedicated XML files like subtree_striker_play.xml for strikers and subtree_goal_keeper_play.xml for goalkeepers. At the code level, brain_tree.cpp is responsible for loading these trees and registering custom action nodes, such as Walk, Kick, and FindBall, allowing the system to cyclically "tick" the tree and determine the robot's next action in real-time.

Are you simulating your robot? If so, which simulator are you using and for what purpose do you use simulations?

Yes, our team utilizes the NVIDIA Isaac Sim to validate the development. Simulation allows us to test software iterations without the constraints of hardware. This includes verifying Behavior Tree logics, testing the vision functioning and simulating match scenarios integrated with the Game Controller within a controlled virtual environment.

What operating system is running on your robot and which middleware are you using (for example Ubuntu 22.04 and ROS2 Galactic)?

Both the Motion Control and the Perception boards run on Ubuntu 22.04. The team utilizes ROS2 Humble as our main middleware, adopting standard client libraries such as std_msgs and sensor_msgs. Additionally, Booster_msgs file defines custom message types to facilitate asynchronous topic-based communication between modules and the booster SDK. Beyond that, the team utilizes the Fast DDS, which is the transport layer, coordinating communication via a publish-subscribe system.

Is there anything else you would like to share that did not fit any previous question?

The system incorporates a dedicated Game Controller bridge that translates UDP packets from the RoboCup GameController into ROS2 topics. This enables the robot to respond with low latency to game state transitions and referee commands, such as penalties or kickoff signals.