SDP Technical Specification

Group 11

Thomas Kerber, Marek Strelec, Euan Hunter

April 19, 2016

1. Introduction

The aim of this project was to create a robot that would play a simplified version of 2-a-side football. We used the provided Arduino Xino RF board and LEGO pieces, a vision feed of the pitch, and a computer to connect Arduino and vision. This report will examine the decisions made in the design of each part and their impacts.

2. Architectural Overview

The core control of the robot was split across an external PC and the onboard Arduino Xino RF unit. An analog video feed with a top-down perspective of the pitch was supplied to the external PC. The external PC is able to communicate with the Arduino via a supplied radio frequency link. The PC uses a USB RF device, while the Arduino has an inbuilt RF transmitter and receiver.

The external PC is responsible for all decision making, while the Arduino carries out instructions passed to it quickly and accurately. To begin with, the PC listens for new frames from the video input, and processes these to extract a set of coordinates for objects of interest: The robots (which are detected by their coloured top-plates), and the ball. This is used to update in internal model of the world. A few adjustments are made to smooth over misdetections by the vision system and correct positions to account for the fact that the robot top-plates are elevated.

At set intervals, the planning module runs, taking a snapshot of the current world state. It then determines what course of action to take in the current situation, and what concrete instructions should be sent to the robot. These instructions are then encoded into the instruction ABI, and transmitted via the RF link. Finally, the Arduino receives the instructions and sets the power outputs to the motors and the kicker accordingly. Some amount of feedback from the rotary encoders on the motors allows the Arduino to adjust the motor powers on the fly, to correct its movement. The Arduino determines by itself when an instruction is over, and proceeds with the next one, or, if no instructions are left, remains idle.

2.1. Interaction of PC Components

Since the controlling PC runs the vision, planning, and communications components, their interaction is non-trivial. The entry point for the control unit is a single main file, `main.py`, which accepts various parameters, documented in appendix C. This first starts the vision subsystem in its own thread, and registers a hook for receiving new world states. These world states are generated by the vision, and then passed back to the main control unit, which uses them to update its own, more detailed, world state. Instead of immediately taking the values supplied by the vision, they are first post-processed to ensure that the values received are sensible. For example, a heuristic is used to filter out a detection of the ball several meters from its last known position as sporadic (It should be noted, however, that this will eventually switch to recognising the new position, should the ball remain there, allowing actual sudden movements to automatically correct themselves).

The planning runs on a stop-and-go mechanism. The planner is run, which sends appropriate instructions to the robot. Then, the planner waits a specific amount of time, which depends on the instructions it sent, and repeats. This loop runs in a separate thread as well. A few notable exceptions are made for this mechanism, with the planner being re-run immediately on certain conditions. These include the ball being within range of the grabbers, and the ball approaching our goal (to allow for timely blocking). Finally, the planner requires human input for information about the game. In particular, it needs to know whether the game is currently in normal play, waiting for a kickoff (and by which team), waiting for a penalty shot (and by which team), or stopped. For this purpose, a GUI is displayed to the operator, with buttons to select each of these states. Upon starting, the game state is considered to be stopped.

3. Communications

3.1. Data Transfer

Communications between the Arduino and computer are done over the supplied RF link. The frequency specified for our group is used. Further, the optional encryption was enabled, and the PAN ID set to '6810'.

Data is sent to and from the devices in packets, where the basic form of a packet is '`<target><source><data><checksum>\r\n`'. `<target>` and `<source>` are the ASCII character '1' for the our robot, and the ASCII character 'c' for the computer. For robot to computer messages, the targets 'd' and 'e' are also valid, used to send debug and error messages respectively. `<data>` is the base64 encoded binary message, and `<checksum>` a checksum of the preceding message. The base64 encoding does not use padding bytes, but is otherwise standard.

The checksum consists of 2 characters, which are calculated as the checksum of all characters at even (starting 0) and odd indices respectively, including `<target>` and `<source>`. These checksums are done by retrieving the values from 0-63 of the characters in base64 encoding, XORing these together. The resulting values from 0-63 are then encoded back into base64 characters. For example, the checksum of '`dt6bas2`' is '`La`'.

Devices ignores all packets not addressed to them, as well as malformed packets (including those with an incorrect checksum). Upon receipt of a packet, the device sends an acknowledgement. An acknowledgement packet has the form '`<target>$<checksum>\r\n`'. An acknowledgement packet is not itself acknowledged.

The computer side will resend packets which have not received an acknowledgement. The Arduino side does not, and discards packets which are longer than the internal buffer (60 bytes). To prevent flooding, only the previously sent packet is resent. This suits the application well, as previously sent packets contain instructions to the robot which are superseded by the more recent one.

3.2. Instruction ABI

For communications, the Arduino and the controlling computer share a binary interface, with the computer sending binary packets of instructions to the Arduino, and the Arduino executing them. Packets sent by the computer consist of one or more instructions for the Arduino, supplied in sequence. The first byte of the transmitted data marks the start of the first instruction. Then, the byte following the end of the current instruction marks the start of the next. Instructions are variable in length, but always begin with one byte identifying the type of the instruction. This is then followed by zero or more bytes of arguments. For example, the binary data `0x06 02 ff 00` consists of two instructions: `0x06` and `0x02 ff 00`. `0x06` is the instruction to open the grabbers, and takes no further arguments. `0x02` is the instruction to move left or right, and takes two more bytes as an argument. `0x02 ff 00 06` contains the same instructions, but in a different order. `0x06 ff 02 00` however has undefined behaviour, as there is no instruction `0xff`. The instructions of this ABI are documented in appendix A.

4. Arduino Architecture

The robot is controlled by an on-board Arduino unit. This is programmed to receive input from a native RF module, as well as the wheels' rotary encoders. Further, it controls the four movement motors, the single grabber motor, and the power flow to the solenoid kicker. The Arduino main loop does three things: It checks for new commands received via RF, it updates the rotary encoder positions with the latest reading, and it updates its internal state and, if necessary, adjusts the motor outputs. This is repeated until the Arduino is turned off.

4.1. Low-Level Commands

Internally, the Arduino operates using a set of low-level commands, which are similar but separate from the communications ABI. Examples of such commands include opening the grabbers for a set amount of time, or moving to the right a specified distance. Each command is a sequence of one or more bytes, the first of which always identifies the command. The most significant bit of the first byte indicates the interruptibility of the command, which will be discussed later, with one indicating uninterruptibility, and zero indicating interruptibility. The remaining 7 bits indicate the command itself, the different possible values and their meaning are tabulated in appendix B. As with the communications ABI, depending on the command, a set number of argument bytes will follow.

Low-level commands correspond directly to a single action the robot should currently be doing, differing in this from the communications ABI, in which a single command may specify a sequence of actions. For example, the communications '`KICK`' command describes a sequence of grabber movements and solenoid actuation, each part of which is an individual low-level command. Low-level commands typically have clearly defined Arduino outputs (spinning the grabber motors, and enabling the solenoid in the previous examples), as well as clear completion criteria, such as time elapsed or distance travelled. Once a command is completed, the Arduino commences the next low-level command immediately.

4.2. Comms Input

If the RFs internal buffer contains a valid sequence of new instructions (see section 3), this is translated into more low-level instructions which can be used by the Arduino directly, and the buffer is cleared. If the RFs internal buffer is *not* a valid sequence of new instructions, but contains a new line, it is considered to be a malformed packet, and the buffer is also cleared.

The translation to low-level commands occurs immediately upon receiving new instructions, and involves mapping each of the ABI commands to one or more low-level commands. For more detail of what low-level commands are generated for each ABI command, see appendix A. The new string of low-level commands effectively replace any in the current buffer, with a few notable exceptions. Some commands (e.g. kicking or the grabbers opening), are uninterruptible, and cannot be replaced once they have been commenced. Since these also compile down into a sequence of low-level commands, this sequence should not be interrupted. This is achieved with the 'uninterruptible' flag bit. If the first low-level command is uninterruptible, the newly to insert commands do not replace it, but instead the first interruptible command in the list. In order to allow two sequences of uninterruptible commands to be treated as separate, an interruptible `NOP` is inserted after each sequence of uninterruptible commands.

4.3. Distance

For multiple of the low-level commands, the distance the robot has travelled, or the angle it has rotated needs to be known. This is calculated from the values taken off the rotary encoders on each wheel, which indicate how far these wheels have rotated. In practice, due to the inconsistencies such as the wheels not stopping immediately and turning at differing powers, these values are not completely linear with the distance travelled or the angle rotated. Instead, a non-linear approximation function was created from measured values to translate rotary encoder positions into workable values for the distance travelled or the angle rotated. This function uses measured values for several points, behaving linearly between them.

4.4. Auto-Correction

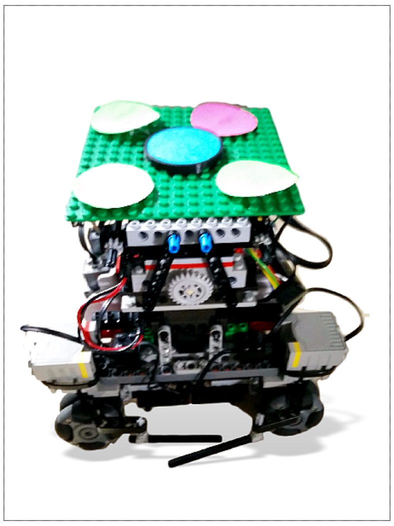
Due to the motors on different sides having different powers, and asymmetry in the design, without any corrections the robot drifts to one side. To compensate for this, the motor powers are updated every time the main loop is run, and motors whose rotary encoders show that they are ahead of the rest are throttled. In practice, this approach introduces a slight jerkiness and unpredictability to the movement, however it improved overall reliability significantly.

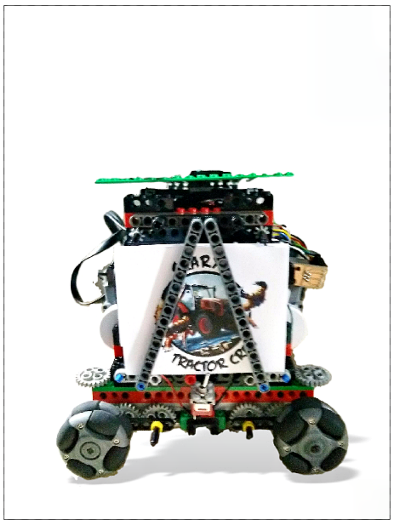
5. Hardware

5.1. Design

The robot is powered by four NXT motors at the sides. These are geared up, and each drive one of the sideways facing holonomic wheels. Holonomic wheels are used instead of ordinary wheels to allow for smooth on the spot spinning, which normal wheels are not capable of, as all four do not reside on the same turning circle. The motors are connected through a spine at the bottom of the robot, on top of which the solenoid kicker rests, facing forward. The solenoid kicker does not use a spring to reset itself, and instead relies on the ball being pushed into it to reset it. The front side of the robot has an old style LEGO motor on both the left and right sides, each to move a lightweight, slightly curved grabber arm. Both of these close at the front, and tuck away to the sides when not used. They are mounted at slightly different heights to allow them to close fully without colliding.

On the inside of the robot there is some space created by the four movement motors. This is occupied by a rotary encoder board, which receives input from the movement motors, the Arduino, along with the supplied motor and power


(a) Front side.



(b) Back side.

Figure 1: Images of the robot.

boards, and various wires and smaller components. These include the wires from each of the motors, the boards splitting the power and rotary encoder cables for the NXT motors, and a breadboard regulating the power input for the solenoid kicker. Both this breadboard, and the power board are wired together, and both draw power for a battery pack containing 4 lithium-ion batteries mounted at the rear of the robot. At the base of the battery pack a power switch is located. A frame is around the loose parts of the robot, and continues up to the top of the cabling, with beams running above the Arduino to mount the top plate on.

5.2. Implementation Notes

Grabbing the ball effectively is a fine balance of speed. Moving the grabbers too quickly kicks the ball away. On the other hand, moving too slowly may lead to missing the ball. Further, the design of the robot means that the ball must be pushed in sharply to reset the kicker. The balance between these that was found, is to close the grabbers at a moderate speed, then rapidly open them part way and rapidly close them again. This resets the kicker, does not kick the ball away in most cases, and is fast enough to be usable.

Kicking the ball is a matter of speed, the faster we can kick, the less time our opponents have to react. However, in order to kick, the ball must also be positioned precisely in front of the kicker. To achieve this, the grabbers rapidly push the ball into position again, correcting any drift from movement. They open part-way, just enough for the kicker to kick without the ball hitting the grabbers. Finally, the grabbers finish opening.

Spinning and moving tends to move the grabbers about. To avoid this, the grabbers are powered slightly during both of these actions. If the grabbers are open, they are constantly opening while the robot is moving, albeit very slowly. Likewise, the grabbers keep closing if they are already closed. For this purpose, the robot maintains a local state variable for the grabber. It is worth stating that this is not measured or sensed, so manually moving the grabbers will confuse the robot. On startup, the robot assumes the grabbers to be in their default, open position.

6. Vision

The vision system was designed to be simple and fast, both to aid in development and to minimise the delay of a movement on the pitch being registered by the system. Originally a system was being developed from the ground up, but after a number of failures and difficulties the decision was made to use Group 12's working system. Group 12's system was adapted from Group 12's vision system from last year, and was used for the rest of the project.

6.1. Theory

The task for the vision system is simple: to find and report of the position of the robots (or more accurately, their coloured top plates) and the ball from an overhead, colour image of the pitch. The design of our system addresses this as follows. First coloured areas matching the colours of the ball and top plate markers are identified in the image. These areas are then processed to determine if they are the ball, part of a top plate, or random noise. For the ball the size of the area is the deciding factor, and for top plate markers the proximity of other areas of colours that would be expected for a top plate are factored in. When an area of the image is identified as an area of interest this area is reported. When a certain object is not found the system does not report anything and this omission is handled in the main world model.

6.2. Implementation

The vision system is implemented as a Python module largely separate from the rest of the system. It utilised the OpenCV library to process the overhead camera feed, and preform the majority of processing. The vision system follows a basic flow of information: the raw camera frame is retrieved, preprocessed, and the information of interest is extracted. This information is passed to the main module, before the vision system loops back to the start. After retrieval, radial distortion is removed from the raw image. It is then blurred using a Gaussian blur to reduce noise. This undistorted noise image is then converted to the HSV (Hue Saturation Value) colour space before being passed to information retrieval. The HSV colour space is used instead of the default BGR (Blue Green Red) as the Hue component represents the colour largely independent of brightness and intensity, allowing a particular colour (or colour range) to be identified more accurate than could be using BGR.

From the HSV image, a binary image is produced for each of the colour ranges used to identify objects (a red colour range for the ball, a pink one for the pink top plate markers, etc.). This binary image is the same size as the HSV image and is white when the pixels in the HSV value are in the colour range and black elsewhere in the image. Contours are then extracted from these binary images and these contours are then preprocessed further to determine if they are a ball or top plate marker. Contours are vector representations of the edges in an image, so in this case describe the shapes of the true areas in the binary images. To determine if a red contour is the ball, the contour is first checked to ensure it is not too small according to a configurable value. The enclosing rectangle that has minimum area is then calculated. If this rectangle has close to equal length side (within a certain threshold), the contour is assumed to represent the ball. To find the top plate markers, the same process is followed. The found markers are then grouped with those within close proximity to the centre marker. The orientation of the top plate is calculate by calculating the angle between the centre marker and the 'odd one out' (the outer marker of a different colour) which is then rotated 45°to give the orientation of the front of the top plate. After this information has been extracted from the binary images, it is passed to the main module, and the vision system continues with processing the next frame.

6.3. Problems

The vision system, on average, performs well. There are, however, a number of areas that could be improved. The objects are not always detected, often due to the colour ranges being too narrow, although an increase would cause a large number of misdetections. When the objects are detected, their positions often jitter from one frame to another, due to noise in the camera images. The colour based matching also introduces other problems such as the pink top plate markers and red pieces of clothing being detected as the ball.

7. Planning

7.1. Design

The planner makes its decision based on the world state passed to it. By default, this is done once per second. However, the interval between planner decisions may be changed based on the last executed action. Some actions, such as turning, require less time than one second. Other actions, however, such as moving and closing the grabbers require significantly more time. The planner estimates this time, and sets the delay to it.

The planner uses a system of goals and actions, the hierarchy of which may be seen in Figure 2. Goals define an overall strategy in a given situation, leading to a particular aim, and are composed of a sequence of actions. In each step, the planner selects an appropriate action to achieve its aim by traversing a list of actions in the currently selected goal. This selection is done by evaluating preconditions of each action. In general, goals are high-level and do not change as often as actions.

Actions define usually one instruction given to the robot with preconditions for its execution. As already mentioned, an action can also return a specific delay time to postpone or speed up next planner execution. Actions compute any parameters to the intructions they send themselves, e.g. there may be an action to rotate to a point, but not one to rotate 30°.

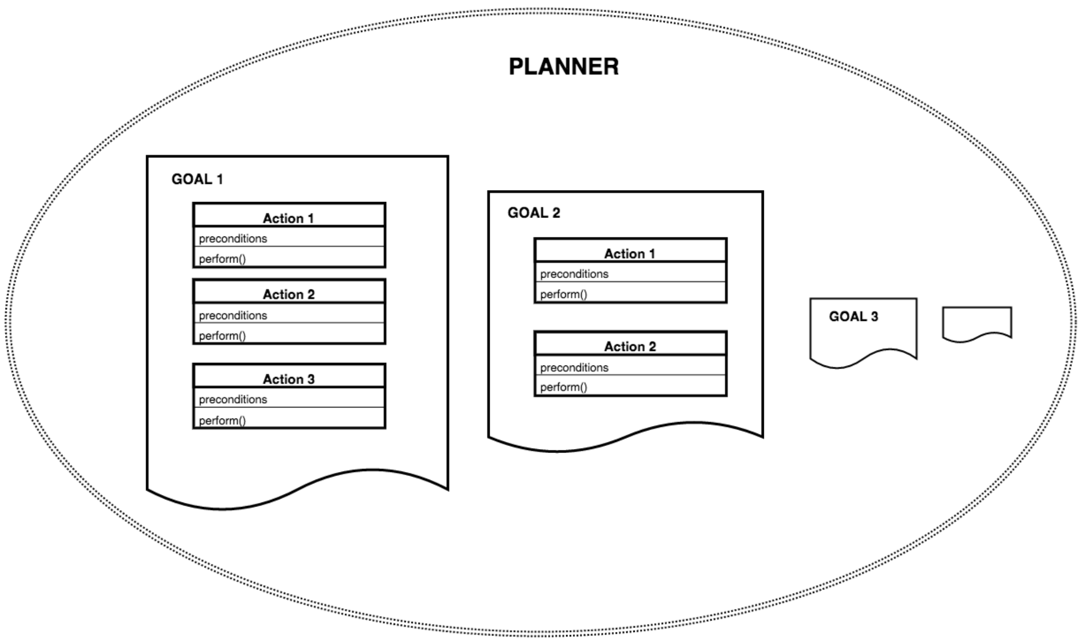


Figure 2: The overall planner structure. The planner consists of multiple goals. Each goal has a list of actions which contain two elements - preconditions and a method to execute.

7.2. Goal selection

The process of selecting a new goal follows a procedure described in Figure 3. In every step of the planner, the current world state is processed and resulting goal is generated based on this analyses. There are three main conditions that effect this decision.

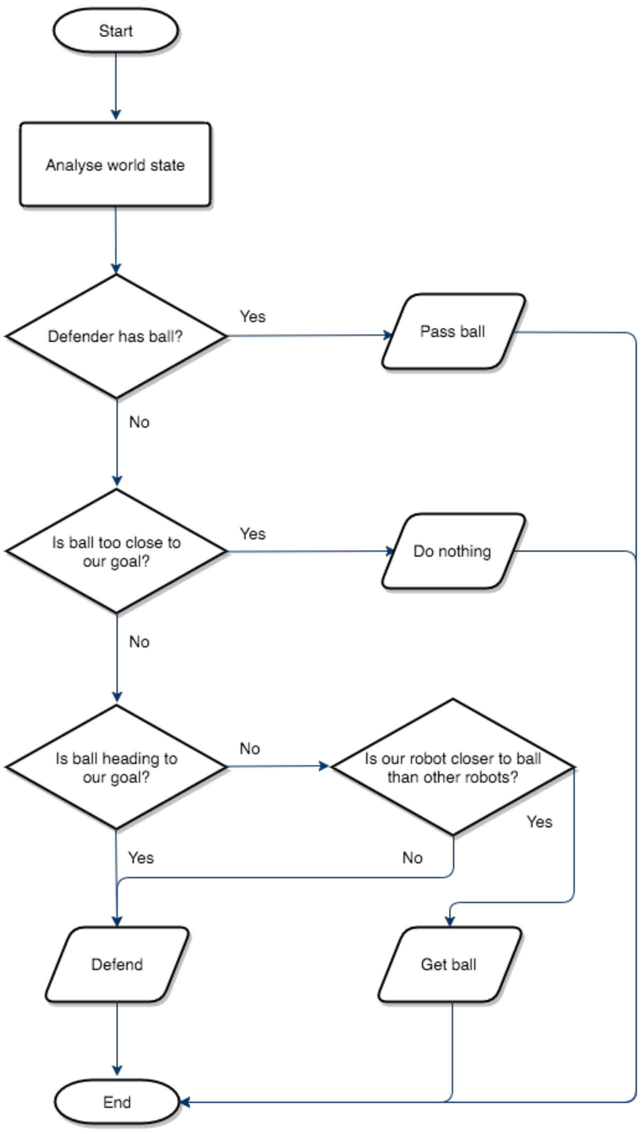

Start
Analyse world state
Defender has ball?
Yes
Pass ball
No
Is ball too close to our goal?
Yes
Do nothing
No
Is ball heading to our goal?
No
Is our robot closer to ball than other robots?
Yes
Yes
No
Defend
Get ball
End

Figure 3: A flowchart showing all main conditions effecting planner's decision to select a goal in each step.

First of all, if our robot captures the ball, it will try to pass the ball to our teammate. If the ball is not in our robot's possession, the planner has to strategically decide where to go based on the ball's position and the position of all other robots. If our robot could approach the ball faster than all other robots (our attacker included), then it would attempt to grab the ball. Otherwise, it would switch to the `Defend` goal and start defending our goal area. This involves returning to our defence zone. If our robot decides to grab the ball and the situation changes so that any other robot is closer than ours, the planner immediately changes the goal to `Defend`. Also, if the ball is heading towards our goal and there is a high chance of scoring, the planner would choose to defend over trying to catch the ball. Finally, one special condition was added. If the ball is too close to our goal, Tractor Crab's goal changes to `Idle`. The reasoning behind this is that it is very unlikely that the robot will grab the ball safely, and it is more likely for an attempt to result in an own-goal.

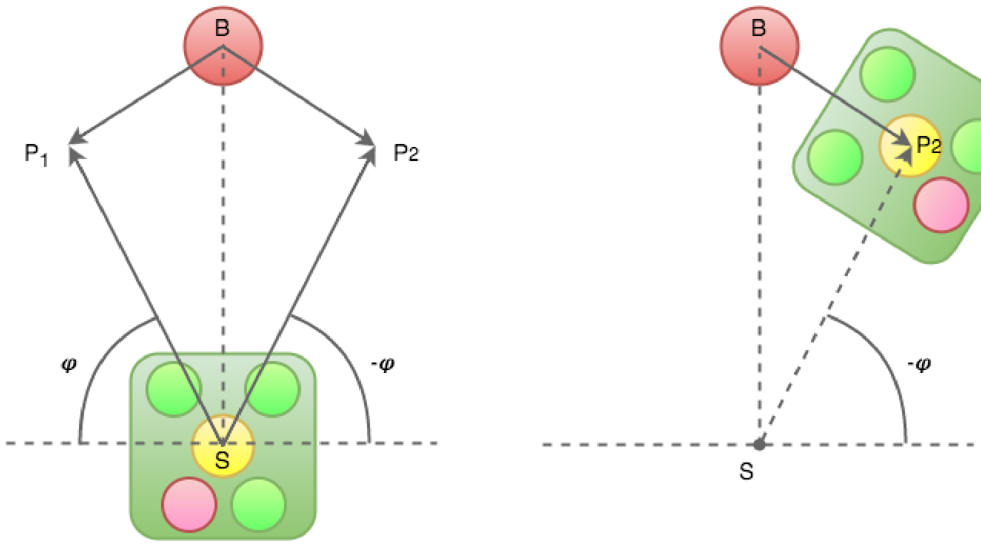
7.3. Goals

7.3.1. `GrabBall`

`GrabBall` is one of the principal goals in the planner. Its purpose is to move to the best position for grabbing the ball and eventually grab the ball. Since the robot can only move sideways, the task is not trivial, and requires a sideways approach detailed in Figure 4.

The naive way of grabbing the ball would be to approach the ball, spin it so that the robot faces the ball, and grab it. However, this approach faces a substantial shortage of hitting the ball while spinning close-by. Thus, it needs to rotate first and then approach the ball. First, the planner finds points suitable for grabbing the ball, assuming that a robot placed on a point is facing the ball. There are usually two appropriate points – catch-points.

One of the advantages of sideways movement is that neither side has priority and thus both sides can be used for any kind of task. As long as the robot faces the ball, it can move in both directions. Thus, there are 4 possible rotations - one for each catch-point and for each movement direction. Only two of these rotations lead to a situation when the robot faces the ball. Out of these, the planner picks the one that requires the smaller angle of rotation. Once the planner knows the catch-point and the corresponding rotation, it can approach the ball. This process is shown in Figure 4a. Figure 4b shows the situation when the robot is in front of the ball and is ready to grab it.



(a) The planner finds two catch-points P1 and P2. For each point, there is a corresponding angle ϕ, which represents the rotation for approaching a point.

(b) The catch-point P2 was selected, the robot rotated and moved so that it stands on P2 now. The next action would be to grab the ball.

Figure 4: Two charts representing the ball-catching algorithm.

Two obstacles had to be overcame in development. Those were mostly caused by vision inaccuracies and a dynamic environment. Firstly, our robot can overshoot in movement and miss the catch point. If this happens, it does not recompute a new catch-point. Instead, it tries to move back towards the current catch-point. Secondly, our robot can overshot in rotation. Since the rotation angle is crucial for successful movement towards a catch-point, the only possible solution is to rotation again. However, this can lead to situations when the robot keeps oscillating between two angles. Since such behaviour is undesirable, a dynamic error threshold is used. As long as the robot is within an accepted error, the rotation is accepted, with the threshold value increases after every successive unsuccessful rotation. This approach represents a trade-off between accuracy and speed. It is better to make a decision fast, even if it is not optimal, because the environment constantly changes. A found solution can be outdated if the decision/adjusting process takes too long.

7.3.2. `Defend`

The primary role of `Defend` goal is to prevent the opposing team from successfully moving the ball over the defended goal-line. This is accomplished by the robot's moving into the path of the ball and either catching it or directing it away from the goal line. To improve chances of catching the ball during the `Defend` goal, our robot is always trying to predict the ball's trajectory and stand at the most suitable defending spot. This spot, also called a defending point, is calculated as an intersection of a predicted ball's trajectory and a semi-circle surrounding the goal area. Such point is shown in Figure 5.

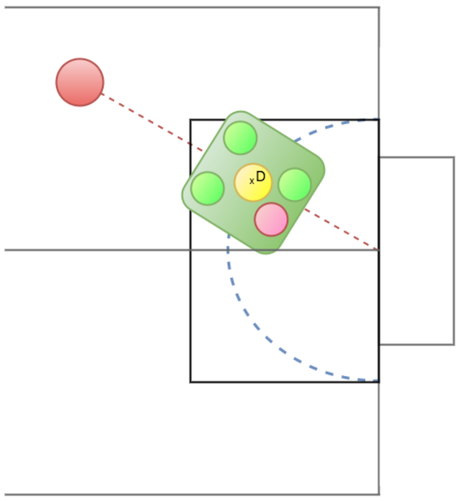


Figure 5: A football pitch with a robot using the `Defend` goal. The blue line represents the defending semi-circle. The red line shows an expected ball trajectory. Point D represents the defence point: the intersection of these lines.

Because of vision inconsistency on different parts of a pitch, the defending point is sometimes slightly off. This problem is dealt with by oscillating movement around the defending point. Since the robot is always moving from side to side, it covers more area and thus its chances to catch the moving ball are higher. A minor improvement was implemented for situations when the robot is not close to the defending point and the ball is moving towards our goal. In this case, the robot immediately moves to the side of the ball and tries to direct it away.

A. List of ABI Commands

This section will outline the available instructions for the communications ABI, the arguments they take, and the effect they will produce. A note on the notation used for arguments: The arguments are standard C types, and are read from the ABI byte buffer as this type directly. Since the Arduino is little-endian, this for example means that `0x03 29`, read as a `uint16_t`, is 10499.

A.1. `WAIT`

Instruction byte:	`0x00`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Waits for the given time in milliseconds.
Relation to low-level commands:	Maps one-to-one to the low-level WAIT command.

A.2. `BRAKE`

Instruction byte:	`0x01`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Brakes all motors for the given time in milliseconds.
Relation to low-level commands:	Relation to low-level commands: Maps one-to-one to the low-level `BRAKE` command.

A.3. `STRAIT`

Instruction byte:	`0x02`
Argument bytes:	2
Argument type(s):	`int16_t`
Action:	Moves right the given distance in millimetres. This distance may be negative, in which case the robot moves left instead.
Relation to low-level commands:	Relation to low-level commands: Maps to an equivalent low-level `STRAIT` command, followed by a low-level `BRAKE` command.

A.4. SPIN

Instruction byte:	`0x03`
Argument bytes:	2
Argument type(s):	`int16_t`
Action:	Rotates clockwise on the spot for the given number of minutes (60 minutes corresponds to one degree). This may be negative, in which case the robot rotates counter-clockwise instead..
Relation to low-level commands:	Maps to an equivalent low-level `SPIN` command, followed by a low-level `BRAKE` command.

A.5. KICK

Instruction byte:	`0x04`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Opens the grabbers and actuates the kicker for the given time in milliseconds.
Relation to low-level commands:	First, a `GRABBER_FORCE` is issued to ensure the ball is in front of a kicker. Then, a part-way `GRABBER_OPEN` is done, followed by the equivalent low-level `KICK`. Finally, a `GRABBER_OPEN` is issued to finish opening the grabbers. All of these are uninterruptible.

A.6. GRABBER_OPEN

Instruction byte:	`0x06`
Argument bytes:	0
Argument type(s):	*none*
Action:	Opens the grabbers.
Relation to low-level commands:	Maps one-to-one to an uninterruptible low-level `GRABBER_OPEN`.

A.7. `GRABBER_CLOSE`

Instruction byte:	`0x07`
Argument bytes:	0
Argument type(s):	*none*
Action:	Closes the grabbers in multiple stages to ensure the ball is caught well.
Relation to low-level commands:	First, a `GRABBER_CLOSE` is issued to get the ball close to the solenoid. Then, a `GRABBER_OPEN`, followed by a `GRABBER_FORCE` is issued, to ensure that the ball presses the solenoid in. Finally, since the `GRABBER_FORCE`'s power leads to the grabbers bouncing back to being partially open, a `GRABBER_CLOSE` is issued.

B. List of Low-Level Commands

B.1. `WAIT`

Instruction byte:	`0x00`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Waits for the given time in milliseconds.

B.2. `BRAKE`

Instruction byte:	`0x01`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Brakes for the given time in milliseconds.

B.3. `GRABBER_OPEN`

Instruction byte:	`0x02`
Argument bytes:	2
Argument type(s):	`unit16_t`
Action:	Spins the grabbers to open for the given time in milliseconds.

B.4. GRABBER_CLOSE

Instruction byte:	`0x03`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Spins the grabbers to close for the given time in milliseconds.

B.5. GRABBER_FORCE

Instruction byte:	`0x04`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Spins the grabbers to close powerfully for the given time in milliseconds.

B.6. KICK

Instruction byte:	`0x05`
Argument bytes:	2
Argument type(s):	`uint16_t`
Action:	Activates the kicker for the given time in milliseconds.

B.7. STRAIT

Instruction byte:	`0x08`
Argument bytes:	2
Argument type(s):	`int16_t`
Action:	Moves right by the given (potentially negative) distance in millimetres.

B.8. SPIN

Instruction byte:	`0x09`
Argument bytes:	2
Argument type(s):	`int16_t`
Action:	Spins clockwise by the given (potentially negative) number of minutes (60 minutes correspond to 1 degree).

B.9. `NOP`

Instruction byte:	`0x7f`
Argument bytes:	0
Argument type(s):	*none*
Action:	Does nothing. Primarily used to separate two sequences of uninterruptible commands with an interruptible `NOP`.

C. Command-Line Parameters

The following command-line parameters are supported, and while none are required the `--defender`, `--attacker`, `--color`, `--plan`, `--goal`, and `--pitch` options should all be set for normal operation.

Short	Long	Description
`-h`	`--help`	Shows a brief help text.
`-z`	`--visible`	Displays logging in purple.
`-1`	`--defender`	Takes one argument, for the path of the defender robots USB RF device.
`-2`	`--attacker`	Takes one argument, for the path of the attacker robots USB RF device.
`-l`	`--logging`	Takes a comma separated list of logging levels to use as an argument. Supported logging levels are 'debug', 'info', 'warning', and 'error'.
`-c`	`--color`	Takes the color of our team as the argument, either 'blue', or 'yellow' (or 'b' and 'y' for short) should be used.
`-p`	`--plan`	Takes the plan to use as an argument. Can be used to run subroutines other than playing the game, however for normal operation 'game' is the correct value to use.
`-g`	`--goal`	Takes the side our goal is on as an argument. Use either 'left' or 'right'.
	`--pitch`	Takes the pitch number being used as an argument, in order to enable pitch specific calibrations. Use '0' for room 3.D03, and '1' for room 3.D04.

D. Goal structure examples

The following tables show examples of Defender's goals. Each table has a list of actions, which are in a queue starting from the top. If an action's preconditions are met, the action is executed and the process finishes. If any precondition is false, the preconditions for next action in the queue are examined.

D.1. **Goal** - `Defend`

Action	**Preconditions**
`GrabBall`	• The defender can catch the ball. • The grabbers are open.
`FollowBall`	• The defender is close to the kicked ball.
`GoToDefendPoint`	• The defender is facing the defending point. • The defender is far away from the defending point. • The grabbers are open.
`RotateToDefendPoint`	• The defender is far away from the defending point. • The grabbers are open.
`FaceBall`	• The defender is not facing the ball. • The grabbers are open.
`Wiggle`	• The grabbers are open.
`OpenGrabbers`	• The grabbers are closed.

D.2. **Goal** - `Pass`

Action	**Preconditions**
`PassAction`	• None

D.3. Goal - `GetBall`

Action	Preconditions
`GrabBall`	• The defender can catch the ball. • The grabbers are open.
`GoToStaticBall`	• The defender is facing the ball. • The grabbers are open. • The ball is static.
`TurnToCatchPoint`	• The grabbers are open.
`OpenGrabbers`	• The grabbers are closed.