Technical Specification

SDP Group 12

April 17, 2016

Contents

1 Introduction

The aim of this project was to create a robot that would play a simplified version of 2-a-side football. We were recommended to use the provided Arduino Uno (Xino RF) board and LEGO pieces, a vision feed of the testing area (pitch), and computers to connect the two and implement any further logic.

This report will examine the decisions made in the design of each part and their impacts. Furthermore it will explain the principles our work relies on.

1.1 Work Distribution and Management

Overall the group was efficiently split into subgroups working on specific tasks. All team members have made contributions to the outcome of the project, though the weights vary.

Furthermore, many tasks were shared with Group 11, and large contributions introduced by teamwork between the two halves of the team.

2 Vision

The vision system is aimed to be a simple and robust platform, providing the latest world description. A requirement for the vision system was to avoid increasing the delay caused by the video feed. The initial world model will be passed to the predictor, which will smooth out the vision output and buffer object positions. This keeps record of object locations, should it be impossible to detect it in a few following frames, and also avoids confusing the planner with erratic object movement from false detections. The resulting world model is then transmitted to the planner in control of the robots. The vision system includes several calibration routines, and saves configurations per pitch room and computer used for accessing the vision feed.

2.1 Approach

The vision system is common for both groups in the team. It was developed by Group 12 and augmented with ideas from Group 11's initial vision platform.

The vision uses a HSV (hue, saturation, value) coded frame rather than the default BGR (blue, green, red). This separates the kind of colour from its intensity and brightness. When a frame is captured, first the radial distortion is removed, allowing object positions to be directly mapped for the planner. Then, blur is applied to the image, simplifying colour clusters, causing objects become more "connected" and thus easier to identify. The frame is then converted to HSV and all future processing will take place over the HSV frame.

The main approach of the vision system is to detect clusters of colour. Examples of this include yellow and blue clusters signifying the centres of robots and a suitably sized red cluster being the ball. For each colour, clusters are detected by creating a mask based on configured thresholds, and finding contours within that mask. In case more than two objects of the same type are detected, only the best match is taken. Potential robots are identified by a yellow or a blue cluster being the centre dot of a robot. After a potential robot centre is identified, green and pink colour clusters are detected in a 40x40 pixel area around the centre. The existence of the pink and green clusters confirm the detected centre indeed being a robot. If none are found, then this robot centre is deemed a false positive and skipped, detection continues with the next best central cluster. Following verification, the robot is identified within the yellow/blue team by the count of the aforementioned pink dots: If we find three pink clusters, then the robots is the pink team member, otherwise it is the green team member. The pink dots on a green top plate were significantly easier to detect than the green dots, and thus chosen for identification.

Once the robot is identified as the pink/green member of the yellow/blue team, the system determines the orientation of the robot. This is done by constructing a vector from the back left dot (pink circle for green robot; green circle for pink robot) to the robot centre, and corrected by around 45 degrees counter-clockwise. The correction amount is dynamically configurable to allow for differences in dot positions on the top plates.

For the ball positioning, the largest red cluster is taken. Ball movement is determined by the time difference between the frames and the last known ball position: a vector is constructed from the previous position to the current position. Velocities of all objects are calculated relative to the movement from the previous frame.

Colour thresholding can become unreliable if the thresholds are not perfectly adjusted, or the environment (e.g. lighting conditions) changes. If the vision system behaves in an unexpected manner, it needs to be recalibrated.

2.2 Semi-Automatic Calibration

The vision system has several calibration routines, most importantly a colour calibrator.

The colour calibration routine is, by default, run every time the vision system is started, but can be skipped, defaulting to the last saved configuration. The vision platform will prompt the user to click on objects of each of the relevant colours in sequence. On every click, a 3x3 square of pixels around the click is extracted, and the pixels within a predefined hue range (hard coded for each colour) are recorded. The thresholds for saturation and value are the minimum and maximum values of each amongst all recorded pixels. Predefined hue ranges for the colours distinguished are shown in Figure 1.

This makes it easy to recalibrate the whole vision system on each launch to adapt to environment changes.

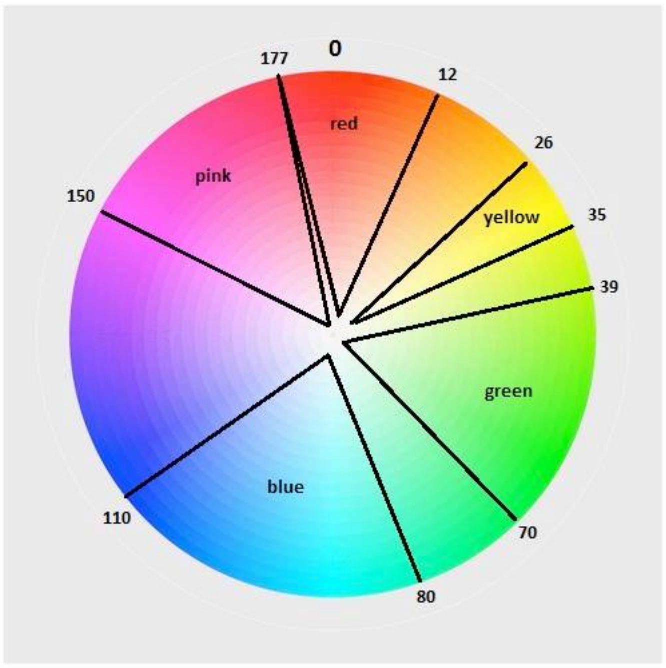


Figure 1: Predefined Hue Ranges

2.3 Manual Calibration

The platform implements a manual way of determining threshold values for colour filters, as well as a pitch cropping specification routine.

The vision system GUI offers sliders to manually determine the threshold values for the colour masks. This is useful for debugging and initial calibration of the colours.

The cropping calibration, which is by default disabled (trigger in the vision code is commented out) determines which parts of the vision feed to use. This has to be run very rarely - once for each camera after every change in the camera angle or position. This provides a quick and efficient way of correcting the pitch area and ignoring any outlying distractions.

3 Planning

3.1 Design

The planner uses a system of goals and actions. Goals define an overall strategy in a given situation leading to a particular aim. Actions define one instruction given to the robot with preconditions for its execution. Goals are composed of a sequence of actions. A goal object selects the next action required to achieve its aim by traversing its ordered list of actions and evaluating their preconditions.

In the implementation, all goals and actions are derived from their respective super classes. These are defined as follows:

```
class Goal(object):
    '''
    Base class for goals
    '''
    def __init__(self, world, robot):
        self.world = world
        self.robot = robot

    # Return the next action necesary to achieve the goal
    def generate_action(self):
        info("Generating action for goal: {0}"\
                .format(self.__class__.__name__))
        for a in self.actions:
            if a.is_possible():
                return a
        return None


class Action(object):
    '''
    Base class for actions
    '''
    preconditions = []

    def __init__(self, world, robot, additional_preconds=[]):
        self.world = world
        self.robot = robot
        self.preconditions = self.__class__\
                .preconditions + additional_preconds

    # Test the action's preconditions
    def is_possible(self):
        info("Testing action : {0}".format(self.__class__.__name__))
        for (condition, name) in self.preconditions:
            if not condition(self.world, self.robot):
                info("Precondition is false: {0}".format(name))
                return False
        info("Action possible: {0}".format(self.__class__.__name__))
        return True

    # Do comms to perform action
    def perform(self, comms):
        raise NotImplementedError

    # Get messages relating to action
    def get_messages(self):
        return []

    def get_delay(self):
        return DEFAULT_DELAY
```

The planner makes its decision based on only on the world state passed to it. This is described by an instance of the class passed to the planner. This object's state is updated by passing a new set of positions (as a dictionary of vectors) to the `update_positions` method of `World`. The `World` class describes the state of the world from the vision, including position vectors for the robots, the ball and the goals. The `World` class and its associated classes also provide methods on their data providing the planner with information about the world. Further utility functionality can be found in "utils.py"

The overall planner is used by calling `plan_and_act(world)`. This selects a goal based on the given world state using its `get_goal()` method. From this goal an action is generated using `generate_action()`. The method then runs (using `actuate(action)`) and returns a delay giving the time until the planner should be run again.

3.2 Implementation

The attacker robot's goals and their respective actions are explained in Table 1. The planner chooses a goal based on ball position, as explained in Table 2.

Goal	Action	Preconditions
AttackPosition	TurnToDefenderToReceive	Attacker in score zone
	GoToScoreZone	Attacker is facing score zone
	TurnToScoreZone	None
Score	Shoot	Attacker has ball and attacker can score
	TurnToGoal	Attacker has ball
GetBall	GrabBall	Attacker can catch ball and attacker's grabbers open
	GoToGrabStaticBall	Ball static, attacker facing ball, attacker's grabbers open
	OpenGrabbers	Ball in attacker's grab range, attacker's grabbers closed
	GoToBallOpeningDistance	Attacker is facing ball
	TurnToBall	None
AttackerBlock	TurnToBlockingAngle	Attacker in blocking position
	GoToBlockingPosition	Attacker is facing blocking position
	TurnToFaceBlockingPosition	None

Table 1: Actions and Preconditions by Goals

Robot in possession	Goal
Our attacker	Score
Our defender	AttackPosition
Their attacker	AttackPosition
Their defender	AttackerBlock
Ball free	GetBall

Table 2: Goals chosen dependent on ball position

3.3 Extensibility

Extending the planner is simply a case of adding goals and actions then adding the logic to select these in `select_goal(world)`. Any new goals or actions should subclass Goal and Action respectively and override methods were stated.

In actions, the logic for performing should be placed in `perform(comms)`. This method is passed a `CommsManager` object through which the robot can be sent instructions. No more than one call should be made to this object in any given action. A new action should also override `get_delay` giving an appropriate delay (in seconds) before the planner should run again. New actions can also have preconditions defined in a variable `preconditions`.

In goals, it may only be neccessary to write a Goal subclass with an ordered list of actions (from last to first). Otherwise the `generate_action()` method can be overriden but similar logic should be followed.

3.4 Integration

An instance of the `Planner` is kept by the "main.py" script. Based on the delays given by the planner, it calls the planner at varying intervals, passing it the latest world model provided by the vision. The planner uses a `CommsManager` object to make control the robot.

4 Communications

The communications module was in vast majority developed by Thomas Kerber of Group 11. Similarly, this section is an adapted version of his report.

Communications between the Arduino and computer are done over the supplied RF link. The frequency is set to the group frequency specified. Further, the optional encryption was enabled with a randomly generated key, and the PAN ID was set to '6810'.

The communication follows the protocol described below. Communicated instructions are mapped to low level command, which are then executed on the Arduino. This is detailed in the following sections.

4.1 Communication Protocol

The communications protocol is formed of packets of the basic form:
`<target><source><data><checksum>\r\n`.

`<target>` and `<source>` are the ASCII character '1' or '2' for the robots of group 11 and 12 accordingly, 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>` is a checksum of said 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 indicies respectively, including `<target>` and `<source>`. These checksums are calculated by retrieving the value of each the character in base64 encoding in the integer range 0-63, and XORing these together. The resulting value from 0-63 is then encoded back into a base64 character. For example, the checksum of '`dt6bas2`' is '`La`'.

All devices ignore all packets not addressed to it, as well as malformed packets. The validity of a packet is verified by the packet structure and the checksum. Upon successful receipt of a packet, the recipient sends an acknowledgement to the sender. An acknowledgement packet has the form '`<target>$<checksum>\r\n`'. Acknowledgement packets are not acknowledged again.

The computer side will resend packets which have not been acknowledged. The Arduino side does not verify packet receipt or resend packets, and discards packets which are longer than the internal buffer (60 bytes). To prevent flooding, only the last 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.

4.2 Instruction ABI

For communications, the Arduino and the controlling computer share an application binary interface (ABI). The computer sends binary packets of instructions to the Arduino, and the Arduino executes them. Packets sent by the computer consist of one or more instructions for the Arduino, supplied in sequence. The first byte of the transmitted instruction data marks the start of the first instruction. 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. The next instruction starts immediately after the previous.

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 its arguments. `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 the following section.

4.3 List of ABI Commands

This section outlines the available instructions for the communications ABI, the arguments they take, and the effects they have on the robot. A note on the notation for arguments: The arguments are standard C types, and are read from the ABI buffer as such type directly. Since the Arduino is little-endian, `0x03 29` is 10499 if read as a `uint16_t`.

4.3.1 `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

4.3.2 `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

4.3.3 `STRAIT`

Instruction byte:	`0x02`
Argument bytes:	2
Argument type(s):	`int16_t`
Action:	Moves right the given distance in millimeters. 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

4.3.4 `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

4.3.5 `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 uninterruptable.

4.3.6 `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 uninterruptable low-level `GRABBER_OPEN`

4.3.7 `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 solonoid. Then, a `GRABBER_OPEN`, followed by a `GRABBER_FORCE` is issued, to ensure that the ball presses the solonoid in. Finally, since the `GRABBER_FORCE`'s power leads to the grabbers bouncing back to being partially open, a `GRABBER_CLOSE` is issued.

4.4 List of Low-Level Commands

4.4.1 `WAIT`

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

4.4.2 `BRAKE`

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

4.4.3 `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

4.4.4 `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

4.4.5 `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

4.4.6 `KICK`

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

4.4.7 `STRAIT`

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

4.4.8 `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)

4.4.9 `NOP`

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

5 Hardware

5.1 Wheelbase

5.1.1 Dimensions

'Drive' wheelbase: 15cm
Radius of 'turn' wheel: 11cm

5.1.2 Origin

The origin is directly between the 'Drive' wheels

5.1.3 Arrangement

The robot uses a 3 wheeled 'T' design with 2 parallel 'drive' wheels along one axis of rotation (capable of rotating independently) and a third unpowered 'turn' wheel rotating perpendicular to these 2 (figure 1.1.1). The 'origin' of the robot should be directly between the 2 drive wheels which are 15cm apart. The 'turn' wheel is 11cm back from the origin, the left wheel 7.5cm to the left and the right 7.5cm to the right.

5.1.4 Motors

The 2 drive motors are lego NXT motors and have rotational encoders built in, this is used to turn and move accurately.

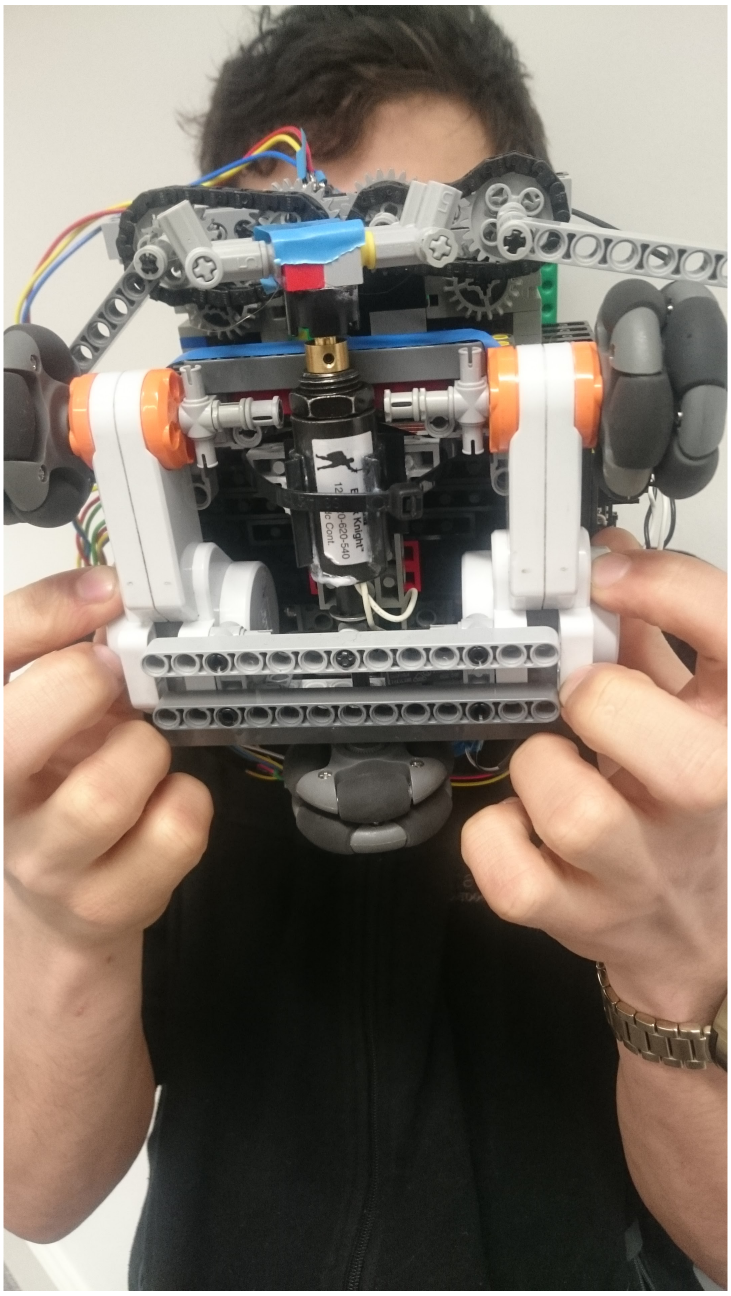
5.2 Front Assembly

5.2.1 Description

The robot Has a 2 motor assembly mounted on 4 vertical bars coming up from the struts joining the left and right NXT drive motors. These are symmetrically mounted and linked by a chain in order to keep them synchronised. The right grabber is lower than the left one so that they do not hit into each other. On one of the centre axles there is a rotary encoder to give the position of the grabbers to the arduino, through the rotary encoder board. position 0 is open, 10 is closed with ball and 13 is closed without ball.

The grabber assembly is fully detachable and can be clipped on and off.

Figure 2: Wheelbase



5.3 Kicker

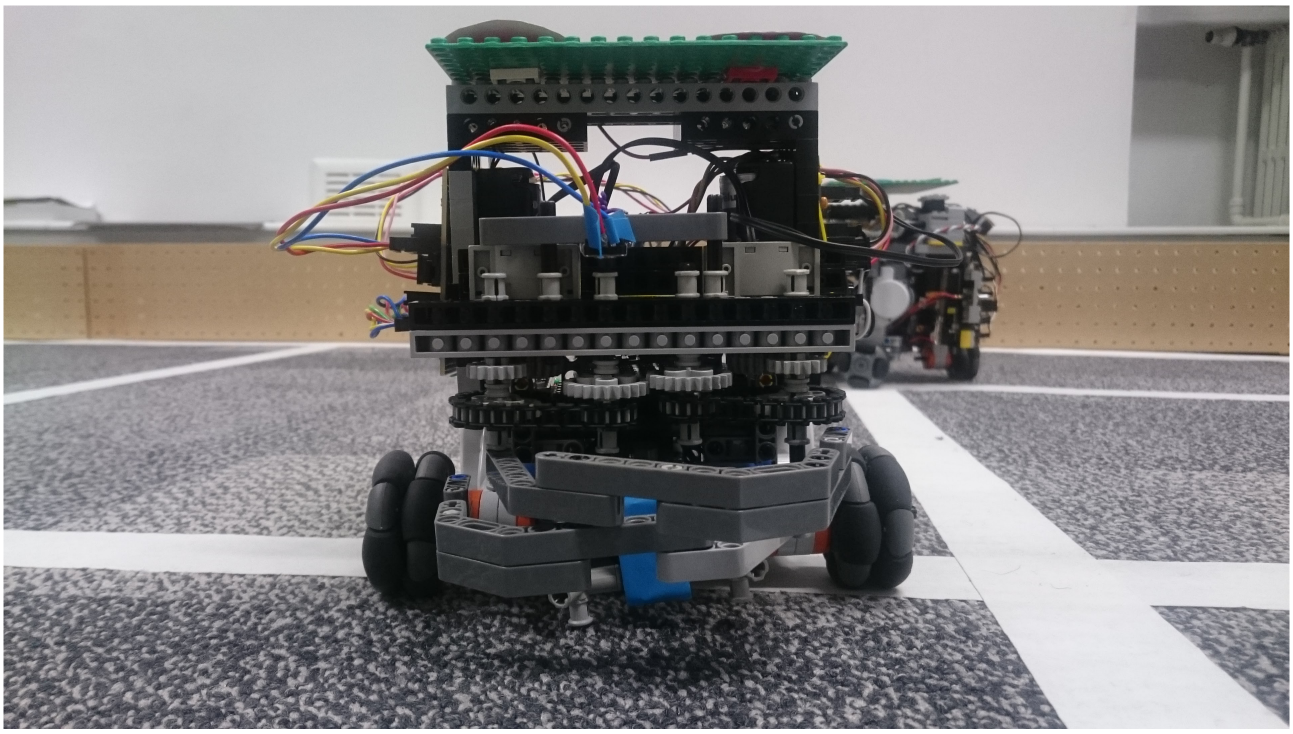
5.3.1 Position

The kicker is a solenoid mounted on the underside of the robot (see figure 1.1.1) connected through a relay to the battery packs, with the relay signal going to the power shield and being controlled directly by the arduino. on the end of the solenoid is a curved kicker designed to keep the ball straight.

5.3.2 Power

The kicker is capable of kicking a distance of 3.2m at full power with an approx error of 15 percent

Figure 3: Front Assembly



5.4 Board and Battery Assembly

5.4.1 Structure

The boards are mounted on a platform between the 3 wheels. The arduino and power shield are mounted in the centre with 2 battery packs of 2 18650 Li-Ion batteries mounted sideways on either side. The rotary encoder board is mounted on the back of the right battery pack and the motor driver board is mounted on the left battery pack. As the batteries face in the packs must be detached to replace them. Both packs clip onto the platform with lego connectors.

5.4.2 Batteries

The batteries are 4* 18650 Canwelum Li-Ion cells wired in series in 2 battery packs. They can be removed and charged individually in the Li-Ion charger.

Voltage (per cell):	2.65 - 4.2
Voltage (total):	10.6 - 16.8
Max Current:	7A
Internal Resistance (per cell):	0.150 Ohms
Internal Resistance (total):	0.6 Ohms
Capacity:	2250mAh
Protection:	overvolt and undervolt

5.4.3 Connections

The batteries are connected in series and must be connected to the power board through the yellow switch, with the splitters on the line powering the solenoid relay (orange wires)

5.4.4 Switch

A yellow switch turns the robot on and off

6 Arduino Software

6.1 Main Loop

The main loop listens on the serial port and executes any command received in a non-blocking manner.

6.2 Command Set

The commands are as follows: kick(distance) grab() release() turn(angle) move(distance)

6.3 Command Response

In response to a command the robot immediately drops what it is doing and runs that command.

6.4 Kicking

The robot releases the grabbers until they are at position 0 then kicks a time dependent on the distance required. It then closes the grabbers to position 13 (fully closed) for continued movement.

6.5 Grabbing

The robot closes the grabbers until they are fully closed or 800ms whichever is sooner, if the grabbers only close to position 10 the grab is considered a success and this is sent to the planner.

6.6 Release

The grabbers are released to position 0.

6.7 Turn

The robot accelerates up to a calibrated turning speed, then maintains that speed until it has just enough time to decelerate to the required position. The wheels are kept turning the same distance using the distance from the rotary encoders by powering down a motor if it has gone too far and powering it up if it has not gone far enough. The turn is considered finished when the averaged distance from the left and right wheels matches the calculated distance required to rotate the desired angle (by using radius of wheels:2.5cm, radius from origin: 7.5cm).

6.8 Move

The robot accelerates up to the calibrated speed and then maintains that speed until it has just enough time to decelerate to the desired distance. It keeps the wheels going the same distance using the rotary encoders by reducing power to a wheel that has gone too far. The distance is determined by `radius of the wheels` $\times \pi \times$ `rotation`.

7 Parts and Costs

Item	**Count**	**Price Each**	**Price**
Duplicate keys	2	£2.50	£5
Holonomic wheel	1	£6	£6
NXT motor	2	£5	£10
43362 Electric Technic Mini-Motor 9v	2	£0.50	£1
PED 121-420-610-540, solenoid, tubular, pull, 6V [1]	1	£10.91	£10.91
Canwelum 18650 protected cells	8	£5	£40
Nitecore charger	0.5 [2]	£14.99	£7.50
Battery holders	2	£3.88	£7.76
TOTAL			£79.17

Table 3: Parts and costs

[1]part is out of production, equivalent part: `http://uk.farnell.com/multicomp/mcsmt-1325s12std/solenoid-tubular-push-12v/dp/2008785?CMP=ADV-CRUK-LF`

[2]split between the 2 groups in Team F