Technical Specification

SDP Group 12

April 20, 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. The robot should work in collaboration with a robot from another group to form a team. To build our system, we used the provided Arduino Uno (Xino RF) board and LEGO pieces, a vision feed of the match area, and computers to connect the two and implement further logic.

This report will examine the decisions made in the design of each component of the system and the impact that these decisions made. Furthermore, it will explain the principles our work relies on.

1.1 Work Distribution and Management

To efficiently manage the workload of the project, the group was split into subgroups, working on specific components of the system, and the tasks for each component were then split between the members of each subgroup.

Additionally, to further implement effective time management, many tasks were shared with Group 11 and large contributions were made by both groups to the communications and planning components of our system.

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 is passed to the predictor, which smooths 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 to 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 the detected centre is indeed 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 robot 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. Predefined hue ranges for the colours distinguished are shown in Figure 1. 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 formed using the according minimum values of relevant recorded pixels as the lower bound, and the maximum absolute value (255) as the upper bound. This worked more reliably than extracting the maximum from the clicked areas as well.

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

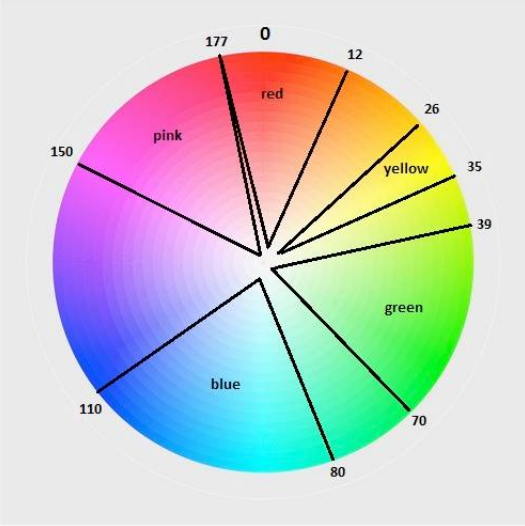


Figure 1: Predefined Hue Ranges

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:

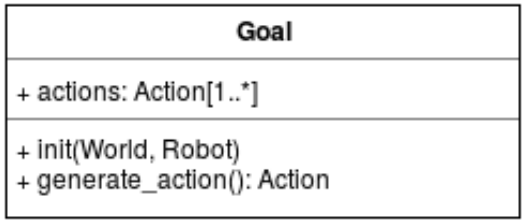


Figure 2: Definition of super class for goals

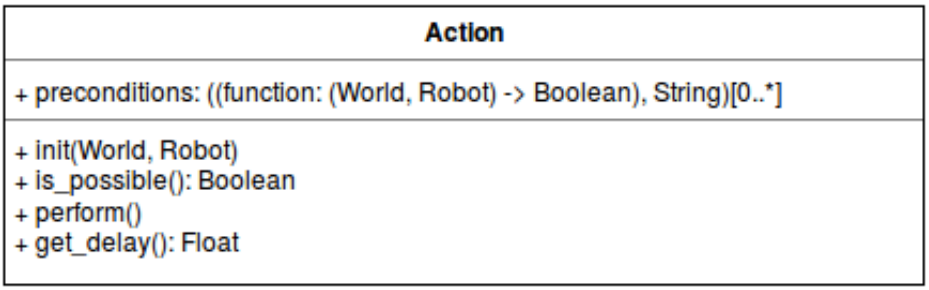
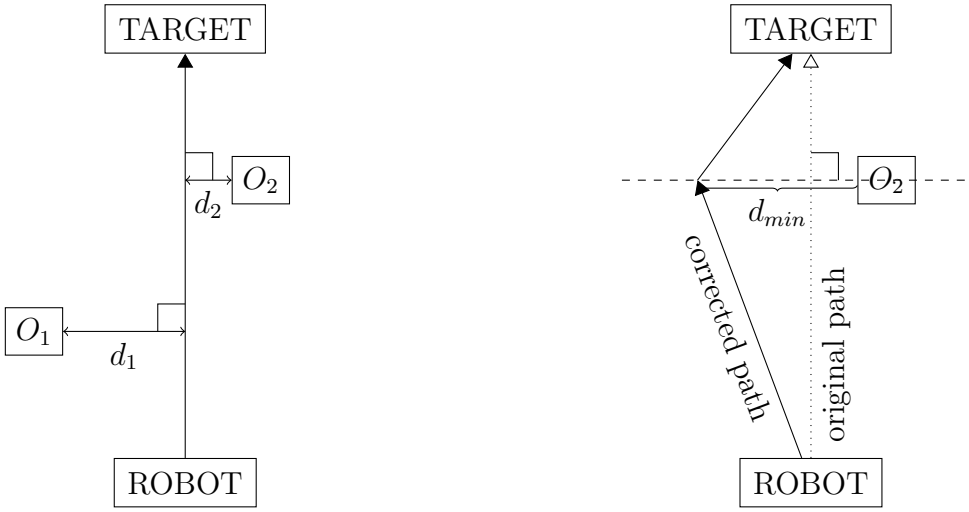


Figure 3: Definition of super class for actions

The planner makes its decision based 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 implement 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.

In order to succeed, the planner must be able to detect obstacles on the path of a robot's movement and on the path of a pass or shot on goal. Having established a location and target, the planner checks for obstacles as seen in Figure 4a. O_i is an obstacle if ($d_i >$ `THRESHOLD`).



(a) Detection (b) Avoidance

Figure 4: Handling Obstacles

By finding the distance of each object from the path and using a constant threshold, the planner establishes whether or not, and if so which, objects will be obstacles to a given path. If an obstacle is found, the planner must, if possible, generate a new path avoiding the obstacle. This is done using the line perpendicular to the path passing through the obstacle. The planner iterates over points at fixed distances along the line, working outwards from the original path, testing for obstacles on this corrected path. This produces a minimal path avoiding the obstacle. This can be seen in Figure 4b.

The planner also incorporates logic outwith this high level design for obeying specific rules, closing the robot's grabbers after a fixed time open and releasing the ball after a fixed time holding it. These work through communication with the robot and timers running alongside the main planner thread.

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 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 List of Low-Level Commands

4.2.1 MOVE

Instruction byte:	0x00
Argument bytes:	2
Argument type(s):	uint16_t
Action:	Moves the given distance in mm

4.2.2 TURN

Instruction byte:	0x01
Argument bytes:	2
Argument type(s):	uint16_t
Action:	Turns the given angle in degrees +ve is clockwise

4.2.3 GRABBER_OPEN

Instruction byte:	0x02
Argument bytes:	2
Argument type(s):	unit16_t
Action:	Opens the grabbers to the open position, responds with 'grabbersOpen'

4.2.4 GRABBER_CLOSE

Instruction byte:	0x03
Argument bytes:	2
Argument type(s):	uint16_t
Action:	Closes the grabbers, responds with 'BC' if a ball was caught 'NC' if no ball was caught

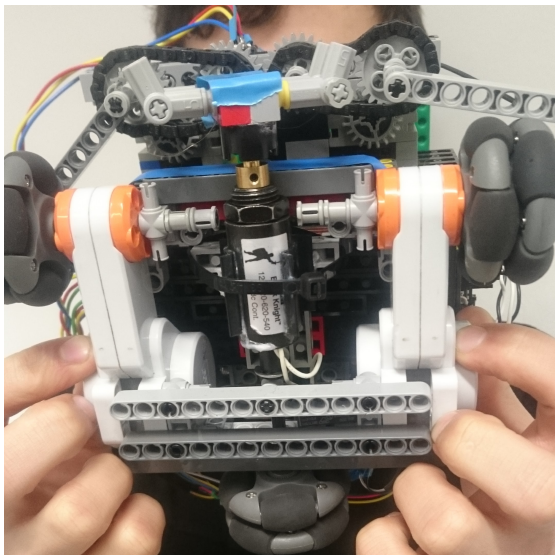
4.2.5 KICK

Instruction byte:	0x04
Argument bytes:	2
Argument type(s):	uint16_t
Action:	Close the grabbers to align the ball, open the grabbers, kick for a time proportional to distance required (in cm), close grabbers

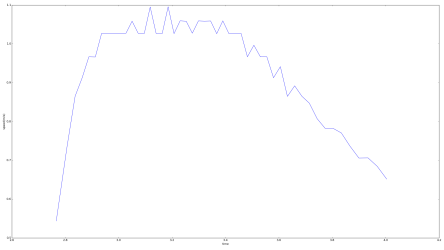
4.2.6 PING

Instruction byte:	0x05
Argument bytes:	2
Argument type(s):	uint16_t
Action:	Respond with positions of motors as a formatted string, for debugging purposes

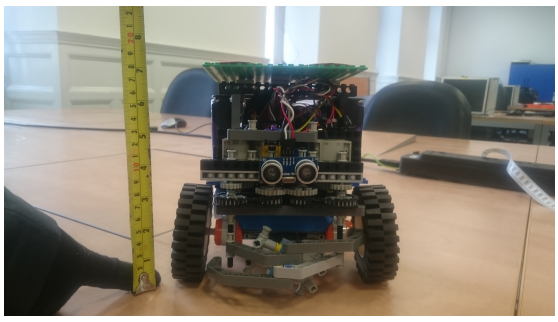
5 Hardware



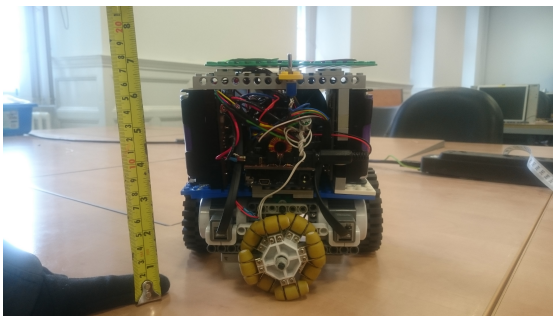
(a) Wheelbase (with old wheels)



(b) Speed Time profile



(a) Front Assembly



(b) Wheels as viewed from rear

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 2). 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.

5.1.5 Wheels

The wheels at the left and right are large 80mm in diameter ribbed rubber wheels. The wheel at the rear is a large holonomic wheel 64mm in diameter

5.1.6 Performance

The performance was measured through the rotary encoders to be:
max acceleration(stable):2.5ms^{-2}
max speed(stable):1ms^{-1}
max deceleration (hard brake):20ms^{-2}
see Figure 1 for profiling

5.1.7 Design Decisions

The 3 wheeled design was chosen for simplicity and accuracy. The robot does not need to strafe as it is an offensive robot and does not defend the goal, it approaches the ball head on and it shoots head on. The 3 wheeled design requires only 2 motors, which is power efficient, and makes the turning and moving software simple. The 3rd wheel was initially powered but it caused the robot to turn inaccurately as it was difficult to precisely match the speeds of the 3 motors. The 'T' shape made forward movement and mounting the grabbers trivial and gave plenty of room to mount a kicker underneath. Having both powered wheels facing directly in the direction of primary movement gave excellent acceleration and speed.

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. This allows the grabbers to be opened to a point where they do not hit the wheels, maintain a closed position without constant current and detect when the grabbers will not fully close, implying there is a ball in the way. There is an ultrasound module for automatically avoiding collisions mounted between the motors.

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

5.2.2 Design Decisions

The motors were chosen as the shape of them provided good structure for strong mounting. It was decided to use 2 for symmetrical weight distribution. The chain was added after it was found the grabber touching the ball was usually slower and could cause the mechanism to jam when one grabber closed faster.

The rotary encoder was added when it was found to be exceptionally difficult to determine if the ball was in the grabbers, as well as exceptionally difficult to prevent the grabbers hitting the wheels. An added benefit was that the grabbers could keep the ball without a constant current, by powering on if they opened beyond a threshold during movement.

5.3 Kicker

5.3.1 Position

The kicker is a solenoid mounted on the underside of the robot (see Figure 4) connected through a relay to the battery packs, with the relay signal going to the power regulator board and being controlled directly by the arduino. On the end of the solenoid is a curved piece of lego designed to keep the ball straight as it pushes it.

5.3.2 Performance

Max distance: 3.2m Max error (radius of target):15%

5.3.3 Design Decisions

The solenoid kicker was chosen as it took up little space, was reliable and was powerful. With the saving in space longer grabbers could be fitted, making grabbing more reliable. Due to unreliable performance and a lack of space a previous design using motors to drive a kicker was abandoned.

5.4 Board and Battery Assembly

5.4.1 Structure

The boards are mounted on a platform in the middle of 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. All wires are kept on the inside of the robot.

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):2.65V(min), 3.7(nominal) 4.2V(max) Voltage (total) :10.6V(min), 14.8V(nominal), 16.8V(max) 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 power the regulator board, which regulates the voltage to the motors and arduino. The kicker solenoid is powered directly from the power pack. The relay powering the kicker is connected to the arduino pin 6 and ground. The motor and encoder boards are connected to the power shield by I2C wires.

5.4.4 Switch

A yellow switch turns power from the power packs to all parts of the robot on and off.

5.5 Design Decisions

The design was subject to much revision, and was changed after encountering specific problems. It was decided to add a switch after trouble stopping the robot when it was out of control, this also helps preserve battery life when not in use. The rotary encoder board and motor board were initially facing out to give easy access to plug things in, however it was found this often caught on other robots and was to vulnerable, so they were made to face inwards. The connections were also originally all removable but this proved to be unreliable and removal was rarely needed to actually remove them so it was decided to solder and heat shrink all the connections permanently.

The batteries in particular were a key revision. The kicker was initially powered from the regulator board but this was found to be not powerful enough and caused the arduino to restart as it drew too much current. The robot also performed inconsistently at first as the batteries discharged (when using the old AA batteries). Testing with the probes found the cause to be the regulator board outputting to low a voltage because the input voltage had dropped to low. This was fixed by using the higher voltage batteries which also have a higher maximum current, allowing them to fully utilise the motors and the solenoid without restarting the arduino. The batteries had the added benefit of cutting off completely for low voltage protection, this allows us to know exactly when to change the batteries. The resulting robot was faster, more consistent and lasted longer.

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 aligns the ball with the grabbers then releases them until they are at position 0 (fully extended) then kicks a time dependent on the distance required, then closes the grabbers to position 13 (fully closed) for continued movement. The formula for kick time in ms used by the arduino is $1.3 * (D - 46)$ where D is distance required in cm.

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. On success: reply "BC" On failure: reply "NC"

6.6 Release

The grabbers are released to position 0. The reply "grabbersOpen" is always sent.

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 the $radiusofthewheels * pi * rotation$. `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	£10.90
Canwelum 18650 protected cells	8	£5	£40
769-JQ1A-5V-F Relay	1	£2.80	£2.80
Nitecore charger	0.5 [2]	£14.99	£7.50
Battery holders [??]	2	£1.94	£3.88
TOTAL			£87.08

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] https://www.amazon.co.uk/gp/product/B015IW0XC4/ref=oh_aui_detailpage_o02_s00?ie=UTF8&psc=1

[3] https://www.amazon.co.uk/gp/product/B00CQKBECA/ref=oh_aui_detailpage_o02_s00?ie=UTF8&psc=1

[4] http://www.mouser.co.uk/ProductDetail/Panasonic-Industrial-Devices/JQ1A-5V-F/?qs=sEN%2fkO1EG6ZZ0oeJzHipHQ%3d%3d