Technical Specification

SDP Group 8-D

Karolis Greblikas
Tom Macmichael
Sanyam Yadav
Valentas Dicevicius
Keith Donaldson

The University of Edinburgh

Contents

1 Overview

This document will guide you through the technical specifications of the robot designed by Group 8 for the 2016 System Design Project. This includes the hardware, software, and communications used.

2 System Architecture

The system was divided into several components that interact with each other. This modular approach helps split the different aspects of the system, allowing the system to be tested independently of other parts of the system, as well as allowing the ability for different implementations of certain code to be used. An overview of this structure can be seen in the UML diagram in Figure 2.2.

The basic data flow of the system, and the four broad components, can be seen in Figure 2.1. Each component is explained below.

Figure 2.1: Data Flow

Vision: This component is responsible for gathering information about the world. It uses various computer vision techniques to extract information from the camera feed about the objects on the pitch: ball and robots. Its only purpose is to constantly update the world model in the strategy system where the gathered information is sent.

Runner: This component receives the information about current positions of the objects on the pitch from the vision system. It updates the various objects with the data from the vision system. It takes the input from the user to decide which task to run.

Tasks: This component is called from the runner, based on the input from the user. It stores all the tasks the robot can do and interfaces with the communication (controller) class. It is the brains behind all the tasks and the logic about what should be executed, based on the objects updated by the runner.

Communication: This component is responsible for packaging the planning stage commands and sending them to the Arduino board via radio.

The sections that follow are split up into hardware and software parts. Hardware part contains the description of a desirable robot, design decisions and detailed descriptions for each hardware component. Software section specifies how vision, runner, tasks and communication components work and interact with each other.

3 Hardware

3.1 Context

The final goal of the robot design was to achieve a robot that can do reliable defending. This requires of frequent adjustments of robot orientation and frequent stops and moves again. The

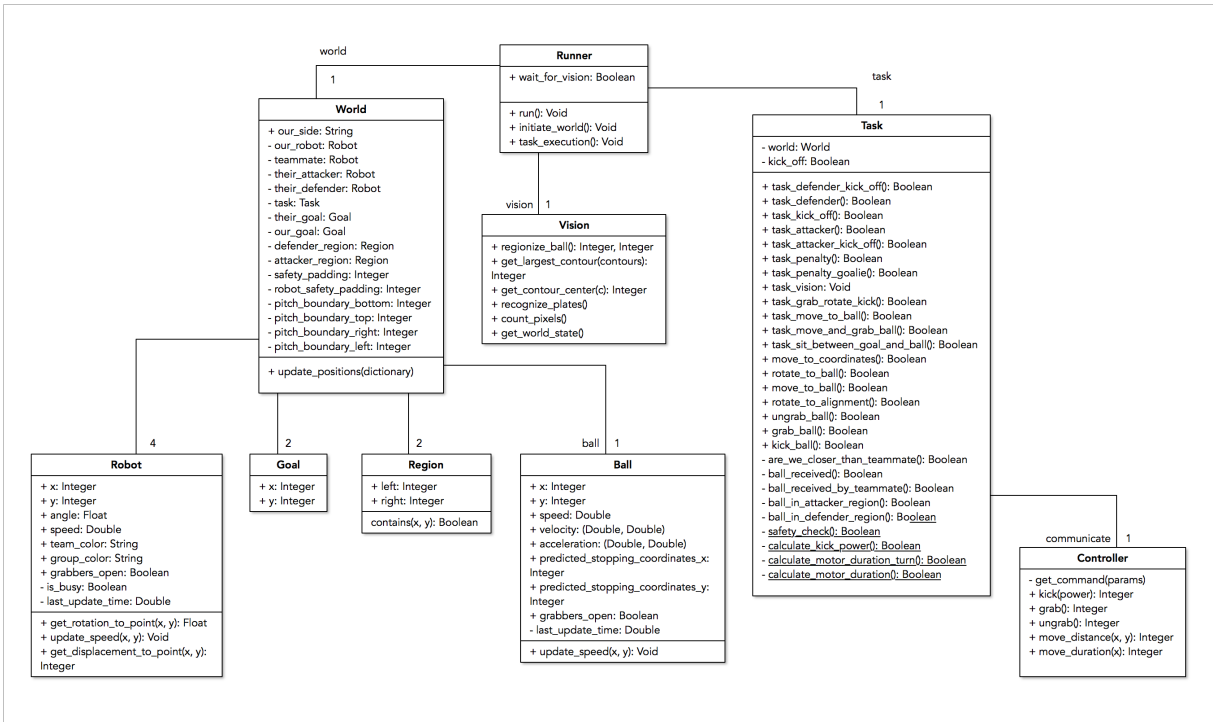

Figure 2.2: UML diagram showing an overview of the system

robot did not need to cover large distances or be fast, resulting in smaller and less powerful motors that power the movement and save battery life. However, kicking mechanism needed to be powerful because attacking teammate might be far away and the robot needed ability to pass the ball to a considerable distance. This resulted in a decision to use one more powerful NXT motor. Finally, a sensor was added that measured distances between the robot and the object to avoid collisions. Below more detailed information on each hardware component is listed.

3.2 Components

Electric Technic Mini-Motor 9v: Since 1997, this motor replaces 2838. Geared down and quite efficient, this is the motor of choice for most applications. This motor is used to power the back two wheels of the robot and give all of the driving force. Moreover, these motors power the grabbers at the front of the robot due to their small footprint.

NXT Motor: This motor is specific to the NXT set (2006). Because of the special connector of this motor (non-standard phone plug type), a cable adapter is required to drive this motor with regular 9V sources. Slow rotation speed, minimizing the need of external gear train. This makes it perfect for a kicker, giving a short, powerful burst when kicking.

The 43 x 10.7mm Wheels: Wheels affect robot's speed, power, accuracy and ability to handle variations in terrain. These wheels are relatively compact and provide a reasonable amount of grip, allowing the robot to move fairly quickly.

Ball Bearings: Ball bearings allow for quick movement and great agility as they require little force to turn, and can do so in any direction.

HC-SR04 Ultrasound Sensor: Attached to detect walls and robots to avoid collisions, this

sensor provides time in microseconds. Using the NewPing library, the sensor data can be transformed into millimetres. For more information, see Section 3.5.

There are no other 'special' parts on the robot, only standard lego pieces like gears, stop brushes, beams and pegs.

3.3 Design

Figure 3.4 takes a view from the bottom of the robot, with the colours representing the following:

Black: Wheels
Red: Ball Bearings
Orange: NXT Motor
Blue: Technic Mini-Motor
Green: 16-tooth gear
Grey: Connecting rods

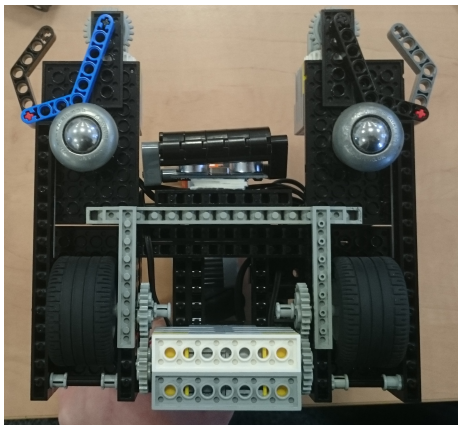

Figure 3.3: Bottom-view of the Robot

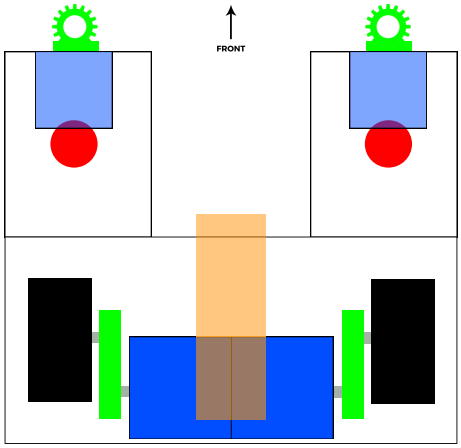

Figure 3.4: Diagram of Robot's Main Components

3.4 Decisions

The robot grabs the ball with grabbers at the two front ends of the robot. The gears at the front turn a set of "boomerang" shaped lego pieces and force the ball into the crevasse in the 'U' shape. The grabbers then release the ball and the kicker kicks the ball. Initially, the grabbers held the ball at the front of the robot, which meant the kicker had less of a backlift before striking the ball. This final design allows the kicker a good amount of backlift and, hence, is fairly powerful in comparison.

The robot turns by forcing one wheel forward and one wheel backwards concurrently. The ball bearings move freely with this allowing a very small rotation circle. The robot is fairly quick and, although it doesn't strafe, can turn sharply. Holonomic wheels were used initially, however ball bearings were used for added stability and movement. Although the holonomic wheels can also move freely, the ball bearings are sturdier and require no battery power to move.

3.5 Sensors

Whilst our vision system can reliably detect the ball's location on the pitch, there is a delay of around 500ms from the ball being at *x,y* and the world updating that the ball is at *x,y*. To overcome this, a small ultrasound sensor was attached to the front-centre of the robot (as seen in Figure 5.8) which can return the time taken (in microseconds) for a sound signal to return from hitting an object. Using the NewPing library, the returned time is converted into millimetres. With this data, the system can know almost instantaneously if the ball is within grabbing distance, hence the motivation behind using the sensor.

A sonar sensor works by emitting sound, then you 'see' your surroundings based on the sound echoing back. This is because sound takes time to travel distances. The farther the distance, the longer it takes for the sound to come back. See Figure 5.10.

The microcontroller tells the sonar signal to go. Then the sonar sensor emits a mostly inaudible sound, time passes, then detects the return echo. It then immediately sends a voltage signal to the microcontroller, which by keeping track of the time that passes, can calculate the distance of the object(s) detected.

The HC-SR04 sensor costs £6.

4 Software

The software is split into two fundamental parts: the code for the Arduino (the code for the robot) and the code for the computer, that sends remote commands to the robot. The code for the Arduino interacts directly with the motors, physically moving the robot, whereas the code on the computer is performing the logic behind the commands: determing how long to run each individual motor for.

The (remote) controlling code has three key aspects: tasks, models, and helper methods. The tasks are completable actions which the robot performs in games, where tasks can be called within tasks to create a chain effect. The models are data structures which hold key data on the important objects in a game. Finally, the helper methods are key in converting the data provided by the models into meaningful actions that the tasks can execute.

4.1 Tasks

Introduction

The structure of the system is to use lots of small subtasks that make up larger tasks that make up an overall strategy that can lead to an autonomous game of football. These are all held in the `planning/tasks.py` file.

As discussed in Section 4.2, every 21 frames a task is passed into the Task class. In previous year's a lot of groups had seemingly complicated methods of creating strategies or tasks. The tasks are written in a way that is clear to anyone reading it what it does. An example is shown in Figure 5.11.

How Tasks Are Executed

The user inputs a task that is to be executed; this is then stored within the runner and is scheduled to be executed. The most basic tasks are: grab, ungrab, kick, move, rotate. In these tasks our robot sends commands to the Communication class that handles communication with

the Arduino. In previous years strategies and communication had become tangled rather than separated: this is something that this system solves. By using a separate communication class it is possible to simply mock this class allowing testing of the task runner without actually calling the Arduino. It also allows the Arduino implementation to be switched to another: keeping the task logic separate.

In Figure 5.12, firstly the duration to run the motor for is calculated, given a particular distance; this is then sent to the communication class that sends a command to the Arduino to run the motor for that long. The communication method returns a value for how long this will take the Arduino to complete: this value is then used to effectively block any future commands being sent. This is an important step as otherwise it could end up in a situation where the Arduino is flooded with commands that either don't get executed or executed at the wrong time, leading to unexpected behaviour.

Example Task

In Figure 5.11 there is a code sample showing one of the larger tasks of going and grabbing a ball. The process is outlined below.

1. Is the robot facing the ball? If no, return `False` and go back and get vision data.
2. Has the ball stopped moving? If no, return `False` and go back and get vision data.
3. Open grabbers
4. Is the robot by the ball? If no, move to it and return `False` and get more vision data.
5. Grab the ball
6. If the ball has been received, return `True`. If no, return `False` and go back and get vision data.

After each failure, the entire process is started again from the start: so each failure (for example the robot moving to the wrong position) results in a re-calculation of everything: the rotation, the distance, the ball location. This means even if the robot has moved to the wrong location, when the process restarts it will check if it's still facing the intended destination and rotate if otherwise.

All tasks are written in a similar manner: it's clear for anyone reading what is going on and it allows quick and easy changes if needed. It also allows us to re-use a lot of code: for example the above function uses the task `task_move_to_ball`, which has a way of anticipating where a ball is moving based on its acceleration, however it can be used as an independent task outright. This approach is shown in Figure 4.5.

Calculating Motor Duration

The assumption for the system is that the robot isn't necessarily going to executed the requested task correctly on consistently. This is mainly due to the fact that so many factors could influence the robot's actual movement there is no point assuming it will work correctly. And so the motor duration functions, both for moving and turning, are arbitrarily calculated on the basis of some testing. In early iterations lots of testing was carried out to calculate the 'optimum duration $\rightarrow$ distance' function however this was found to be unreliable and so this approach was taken.

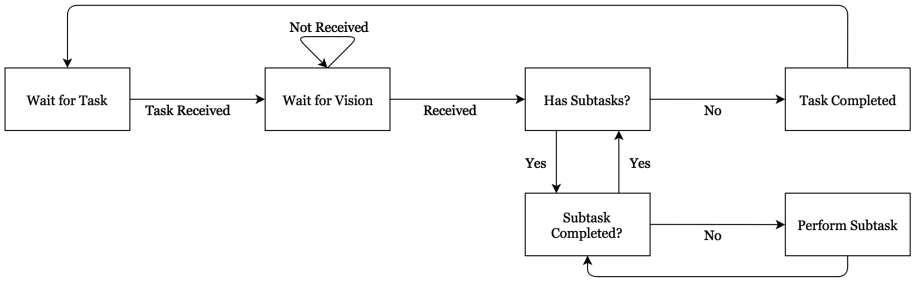

Figure 4.5: Finite state machine showing the process of the task runner

Safety Check

Before the robot is given any commands it performs a "safety check". Passed into this function are the co-ordinates that have been calculated by the task runner. Initially a boundary check is performed: are the co-ordinates within (x) units of the wall? If so there is a breach and the robot can't move there. Secondly, the safety checker calculates the path that the robot will be travelling on: from its current position to destination position. Are there any robots on this path? If so there is a breach. Each robot has a 'boundary' around it as well that is also considered a breach and it is not routed there either.

In both situations rather than simply fail it is re-routed: the furthest the robot can go without collisions is calculated and then routed there.

4.2 Runner

The task runner is the part of the system that ties together the different subcomponents: bringing together vision, communication with the Arduino and the user's requested task. After looking at previous groups' code it became clear that a lot of people had re-used the same parts of the same planning system and it had become very messy, with lots of unnecessary code and no clear structure, therefore the task runner was written from scratch. However, there were some fundamental things (such as the `World` state concept) that inspired the chosen design.

The task runner initialises a `World` object based on some parameters provided by the user: such as which side of the pitch the user is on and what colour the teams are. This also in-turn initialises `Robot` and `Ball` objects that store the location and colour information of those objects. How the different classes are linked together can be seen in Figure 2.2.

The task runner is then built on a single `while` loop that continually reads data from the vision system, updates the graphic user interface to display this vision data, and then passes this data to the world state as seen in Figure 5.13.

The vision data: (x,y) co-ordinates of the different robots and ball are passed into the `World` object, that contains the `World` and `Ball` objects.

The task to be executed is held with the runner and continually executed in this loop: every cycle, the task to be run is passed to the Task class which handles the execution of that task. To prevent outdated or incorrect vision making decisions for the robot, a task is only executed every 21 'cycles' of this while loop. This allows the vision to catch up with what is actually happening on the pitch.

Essentially this meant a 'snapshot' of the pitch status is sent to the task manager, along with a task to be executed. In the task manager itself, it determines whether this task has completed (discussed in Section 4.2) or not: if the task has successfully completed, the task manager will return `True` and the Runner class' current_task is updated to None: meaning the task runner will not be called continuously after a task has completed. This approach is like a finite state machine shown in Figure 4.5.

Decisions

The task runner was designed so that after every task requested, a check would be made to see if it had completed and if not, repeat it. This is why the system is 'corrective': after every command sent to the robot, more vision information is collected, if it hasn't met its tasks after the extra data, it re-routes and sends further commands. Gradually the robot moves toward the intended destination. This approach ensured the robot would always reach its destination. Figure 4.5 shows this procedure as a finite state machine.

4.3 Helpers

An important aspect of the system is knowing what is moving and what is not. To do this is simple: every time vision data is collected the robot and balls' positions are updated. Then, their previous position is compared with their new position, and using the time since their position was last updated the speed and velocity are calculated.

On top of that the object's acceleration is calculated. The advantage of knowing an object's acceleration is that it is possible to predict exactly where it will stop - using the object's previous velocity and the object's current velocity, as well as displacement. All of these things are calculated every vision frame and updated in the object as shown in 5.14.

These are methods which are used in performing actions:

- `calculate_speed`: Given coordinates and the time taken, it will return the speed. The function is independent on units.
- `update_speed`: Calculate the speed the ball is moving at, based on the last time the speed was updated and the distance moved.
- `get_displacement_to_point`: Uses the euclidean distance to calculate the displacement between this robot and a target co-ordinates.
- `get_rotation_to_point`: Calculates the rotation required to achieve alignment with given coordinates. This presumes the robot's angle given is the degrees from the north, +clockwise. So 10 degrees is 10 degrees clockwise from north.
- `update_positions`: This method will update the positions of the pitch objects that it gets passed by the vision system.

In the task runner the predicted stopping co-ordinates of the ball are used to anticipate where it is going to stop, rather than where it currently is, allowing the robot to move to the destination in fewer commands than otherwise would be necessary.

These helper functions are stored in `planning/helper.py`. By storing these general maths-based functions in a separate file it is possible to to use the same functions in different parts of the system, but it makes it very easy to test as no objects need to be created.

4.4 Models

The controlling code is based off of models. These models all exist within the 'World', and these 'World' elements get passed to the computational systems which interpret them.

The models in the system are:

- `Robot`
- `Ball`
- `Region`
- `Goal`
- `World`

When the runner starts it creates instances of these classes to store the data in. For example there are four robots on the pitch so four `Robot` objects are created and stored in the `World` object, held by the task runner. How these are all connected is shown in the UML diagram in Figure 2.2.

These models are updated constantly by the vision feed, or periodically by the system. This structure allows for consistent data throughout the various files which control the robot.

An illustration of the different instances of these models is shown in Figure 4.6.

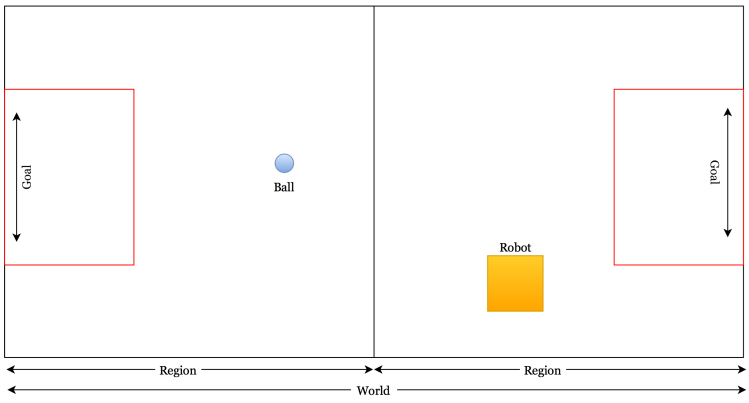

Figure 4.6: Diagram showing the different objects in the World

4.5 Communication

Subtasks move, turn, grab and ungrab interact with our Communication class in `communication/controller.py`. Here the commands are sent directly to the Arduino. The code is based on previous group's communication code that proved to be reliable and clear. It was stripped to the bare minimum, with just code for the 'fundamental' tasks.

There is an instruction interface between Python communications code and Arduino code. In Python code, instructions are packaged to a specific bit pattern for each of subtasks like grab and sent to Arduino. Arduino code, located in arduino/controller.ino, unpacks those instructions and executes corresponding low-level commands like move certain distance by running

engines for specified time period which was decoded from the instruction received.

In Figure 5.15, an excerpt of such low level code is shown. It is executed when Python communications module packages a grab instruction that is decoded in Arduino code and then executed.

Commands that can be sent to Arduino:

- `kick`
- `move_forward`
- `turn`
- `stop_engines`
- `receive_binary`
- `grab`
- `ungrab`

The decision was to keep the Arduino code as simple as possible, with all the logic in the python code. The process of programming the robot continuously was slow and tedious and testing was far trickier. By writing all of our logic in python it was possible to do much more thorough code-based testing as well as do lots more iterations as the whole process is far quicker because there is no need to reprogram the robot every time you need to make improvements or changes.

4.6 Tests

In the folder called `tests` there are numerous tests for the system. Tests ensure that any changes made to the system will not break anything else, but also for reassurance that everything in the code is working as anticipated so that the code can be ruled out of any issues. The focus is to test as much as possible in the code itself, rather than on the pitch.

There are tests for tracking the speed, velocity, acceleration of the ball. As well as tests for ensuring that the maths behind the acceleration and ball prediction system works correctly and that the safety check works as expected.

4.7 Decisions

The tasks system provides high-level commands to a converter, which then invokes methods from the task manager. This planning system is made up of simpler tasks and communicates directly with the robot's controller code. The code on the robot was designed as fairly lightweight, and most of the processing happens on the computer. This decision was taken as the processing happens much quicker on a computer than it would on the PC, regardless of the delay.

4.8 Vision

Purpose and Setup

The goal of the vision system is to obtain data about the world and supply it to the strategy system. The returned data is in the format of 2-D coordinates in pixels and angles in degrees. It is vital to setup correct camera calibrations for the vision to work accurately. At the start of the application, auto-calibration takes place. It is needed because analog cables transmit inconsistent vision data even on different computers with the same wires plugged in. It works

by adjusting camera settings iteratively. On each iteration, a frame is taken from the vision feed. Mean and standard deviation of the grayscaled frame are calculated. Camera settings like brightness and contrast are adjusted by a small value either upwards or downwards according to the actual mean, standard deviation and the goal mean, standard deviation that are constants defined before. This produces stable vision feed settings for a particular computer used automatically. This code runs in `setup_camera` method before the objects recognition code.

Detection

After camera calibration, calibrations file data is loaded for objects detection. It includes color ranges and values for various computer vision techniques like blurring, dilation etc.

Ball detection is based on realisation that in HSV color space, its color ranges from deep red to dark pink. Camera frame is converted to red and pink masks and combined, then blurred to create the final mask. Since ball is large compared to the noise in the image, it is reliably found by looking for largest contour in the mask. The coordinates of the ball center and its radius is then sent to strategy system. The logic can be found in `recognize_ball` method.

Robots detection is more complicated and involves a method developed from scratch. It is easier to isolate plates on the robot and then use the algorithm for detection only in this constrained region. At first, color ranges are used to locate deep green in the camera frame(the plate) and, in addition to that, another colour range for pixels with non green hue, but high saturation and value is used. An example mask produced by this can be seen in Figure 4.7. Masks that are of similar size to a usual robot are then fit with bounding, rotated rectangles. This isolates the areas in the frame, eliminating the noise from the other regions of camera frame. Pixels of each color are counted in those regions only and their ratios are used to determine team and group.

For instance, if there is more blue, then the team is blue. Alongside, group is determined by comparing ratios of green to pink. Finally, the orientation of the robot is determined from pixel counts and the fit bounding box. For instance, if the plate contains three green circles and one pink circle, mass center of pink circle is calculated and the closest corner of the bounding rectangle is found. After that, a vector that starts at the centre of the bounding rectangle and is parallel to the edges and corresponds to the direction the robot is facing is drawn. The angle of the vector is then calculated. A resulting example detection can be seen below with the direction vector in white and the bounding rotated rectangle in pink colours. The position of the robot is simply x, y coordinate of the bounding rectangle center. The robot coordinates, team, group and angle are all sent to the strategy system. The logic can be found in recognize_plates method.

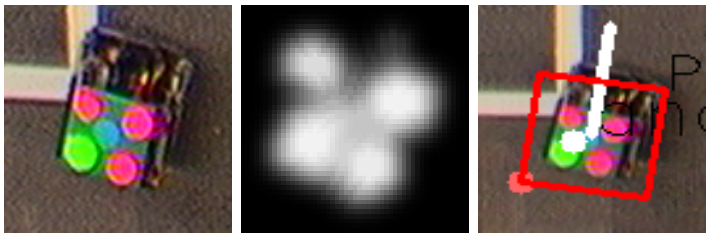

Figure 4.7: Steps of robot detection

5 Appendix

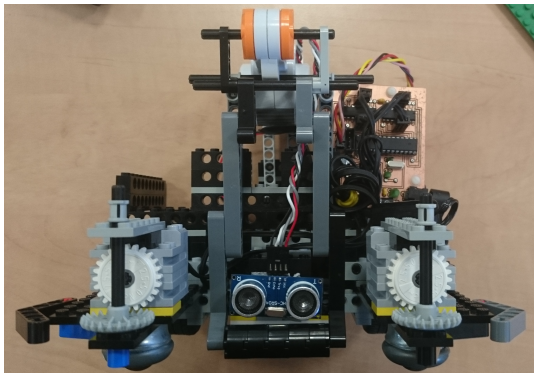

Figure 5.8: Front-view of the Robot

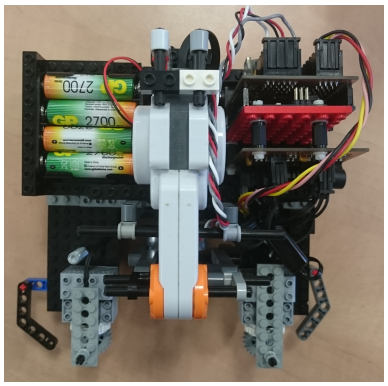

Figure 5.9: Top-view of the Robot

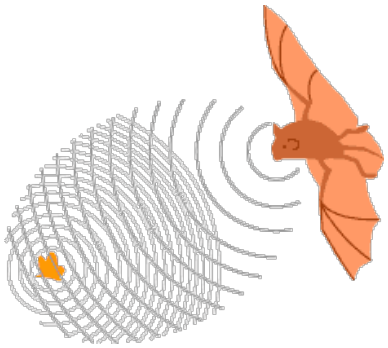

Figure 5.10: Sonar Demonstration

```
def task_rotate_and_grab(self):
    # rotate to face the ball
    if self.rotate_to_ball():
        # wait till ball has stopped
        if self._world.ball.speed < 5:
            # move to the ball with grabbers open
            if self.ungrab_ball():
                if self.task_move_to_ball():
                    if self.grab_ball():
                        return self.ball_received()
    return False
```

Figure 5.11: Code sample showing the code to execute the task 'Rotate and grab'

```
calculated_duration = self.calculate_motor_duration(distance)

# Tell arduino to move for the duration we've calculated
wait_time = self._communicate.move_duration(calculated_duration)

# Wait until this task has completed
sleep(wait_time)

# Returns false which means we'll get more data from vision first
# run this function again, to verify ok
return False
```

Figure 5.12: Code sample from tasks.py showing the communication interface

```
self.world.update_positions(data)
```

Figure 5.13: Runner.py

```
self.speed = calculate_speed(self.x, self.y, x, y, time_since_last_updated)

# predict where the ball is going to stop
self.velocity = calculate_velocity(self.x, self.y, x, y, time_since_last_updated)
self.acceleration = calculate_acceleration(initial_velocity, self.velocity, time_since_last_updated)
self.predicted_stopping_coordinates_x, self.predicted_stopping_coordinates_y
    = predicted_coordinates(x, y, initial_velocity, self.acceleration)
```

Figure 5.14: Sample showing the update_speed method of an object

```
void ungrab() {
  motors.run_motor(LEFT_GRABBER, -0.2, uint16_t(float(500)), -1);
  motors.run_motor(RIGHT_GRABBER, 0.3, uint16_t(float(500)), -1);
}
```

Figure 5.15: Code sample from controller.ino to ungrab a ball