System Design Project - Report 3

Group 12 - Robot Unicorn Defenders
Marc Howarth

I. Introduction

For the third milestone I found myself interweaving within all the sub-teams. Vision, construction and movement. Also finding the time to give our robot a victory song such that when it score a goal it sings the Mexican Hat Dance.

II. GUI

A critical part of being able to test our robot has been the ability to select different options easily on the GUI. Figure VII-A shows the GUI and all different choices that we can choose from.

Most of the GUI was implemented by Joe however, I realised, after testing, that we would need the ability to choose which goal is ours, and which is our opponents. After implementing this, some of the strategies were updated as they had assumed a single goal position.

III. Sensors

One of the construction team's iterations included a light sensor placed behind the kicker. I decided to utilise the sensor within the code on the brick. This required a lot of calibration because the lighting levels in different rooms changed the values of high and low level of light needed. To get around this, I create a program that calibrated the light levels and outputted the required parameters.

We used the light sensor to determine when the ball was in front of it, however it could have also been used to know whether the robot was about to run into a wall or any obstacle. The light sensor emits a red light which may effect other teams vision, so we decided to not use a light sensor and include two touch sensors instead.

IV. GoToBall

After failing to reach Milestone 2 in time, I set to work on a strategy that could approach a ball. Being able to use the simulator whilst building this strategy was helpful. It was hard to know where to start, as the strategy had to handle 5 blocks of data (see Figure 2) every second and make decisions based on this. I have included a brief overview of my code in Appendix VII-A.

V. Moments

I briefly joined the vision team to help with orientation calculation. In order to get a more stable prediction we developed a method using image moments. An image moment is a function of certain particular weighted moments of the pixel intensities of the image. The function we used for calculating this orientation is:

$$\Theta = \frac{1}{2}\arctan\left(\frac{2\mu'_{11}}{\mu'_{20} - \mu'_{02}}\right) \tag{1}$$

We found to give a very steady orientation, but only in the right half plane. This is because *Theta* only gets results in the right half plane and when the robot is facing anywhere in the left half plane, the orientation given was always facing the opposite direction.

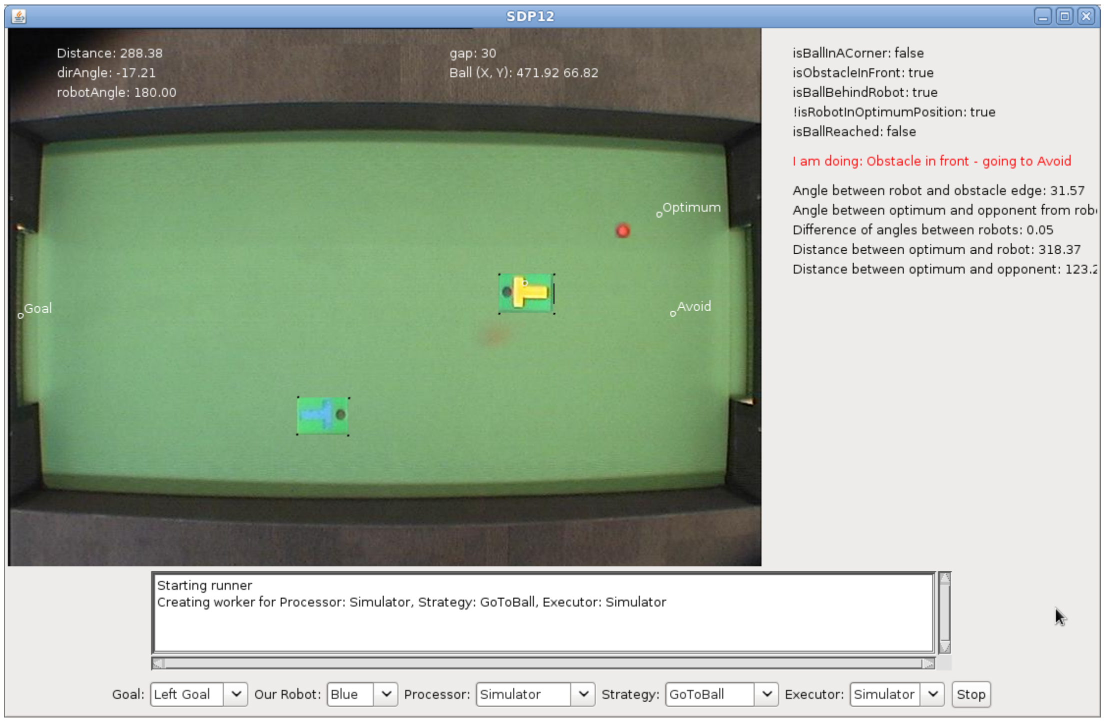
To get around this we used a helper function - in this instance the center of mass method. This helper function would detect if the robot was facing the left half plane and adjust the results given by the moments by Pi. The results were perfect except for when the robot was facing directly upwards or downwards. An idea to fix this, is to vary the moments formula to show results in the lower and upper half plane and include this alongside the helper function.

VI. Testing

Whilst the strategy had been tested in the simulator, no work was done on the robot itself until the day before the milestone. The first results with the robot were good, with the robot was trying to get to it's destination point but weaving left and right to do so. The differences between real life and the simulator soon came clear, and so we increased the thresholds used to determine when the robot should go straight. Also, to decrease the weaving, we made sure the robot couldn't go from turning left to turning right without going straight first (and vice versa). This did lead to some inefficient turning on the pitch but helped improve the overall results of the strategy.

VII. Appendix

A. GUI



Distance: 288.38
dirAngle: -17.21
robotAngle: 180.00

gap: 30
Ball (X, Y): 471.92 66.82

isBallInACorner: false
isObstacleInFront: true
isBallBehindRobot: true
!isRobotInOptimumPosition: true
isBallReached: false

I am doing: Obstacle in front - going to Avoid

Angle between robot and obstacle edge: 31.57
Angle between optimum and opponent from rob
Difference of angles between robots: 0.05
Distance between optimum and robot: 318.37
Distance between optimum and opponent: 123.2

```
Starting runner
Creating worker for Processor: Simulator, Strategy: GoToBall, Executor: Simulator
```

Goal: Left Goal Our Robot: Blue Processor: Simulator Strategy: GoToBall Executor: Simulator

Fig. 1. Screenshot of GUI running the simulator

- *Goal*: Choose the goal that we are aiming at.
- *Our Robot*: Determines if our robot is mounted with the blue or yellow plate.
- *Processor*: Get the location information from either a file, a process (i.e. our vision system) or a simulator.
- *Strategy*: A list of strategies including goToBall and takePenalty.
- *Executor*: Execute the commands on the simulator or through Bluetooth.

RobotA_x	RobotA_y	RobotA_t	Ball_x	Ball_y	RobotB_x	RobotB_y	RobotB_t

Fig. 2. Data received from Vision

```
while UpdateVision() do
    $left \leftarrow 1$
    $right \leftarrow 1$
    $threshold \leftarrow 30$
    $requiredAngle \leftarrow getAngleBetweenBallAndRobot()$
    $diffAngle \leftarrow AngleOfRobot() - requiredAngle$
    if diffAngle $<$ 0 AND abs(diffAngle) $<$ 180 OR diffAngle $>$ 180 AND abs(diffAngle) $>$ 180 then
        $right \leftarrow -1$
        $left \leftarrow 1$
    else
        $right \leftarrow 1$
        $left \leftarrow -1$
    end if
    if abs(diffAngle) $<$ threshold then
        $right \leftarrow distance$
        $left \leftarrow distance$
    end if
    rotateWheels(right, left)
end while
```

Algorithm 1: GoToBall Strategy