OS report
Mr. Robot — Always ready!

Mokhles BOUZAIEN, Ruben VAN DER HAM, Youssef DOUBLI

February 2020

Contents

1 Introduction

The identify and shoot project consists of building a robot using the EV3 Brick, to play a game where the robot has to detect objects, avoid obstacles and shoot balls to the opponents' side of the field. The programming was done thanks to techniques studied during the OS course.

2 Mechanical structure

Our initial decision was to make the most minimalist and compact build possible. Although more complex builds, with sophisticated shapes (arms, moving elevators) are tempting, and might provide useful functionality, they create great mass imbalance and make it very hard for the robot to move with a high precision. So this choice was made to serve the goal of optimal mass distribution, which will eventually save us a lot of troubles. We used two tacho motors to animate the wheels, a sonar on the lower front of the robot, and a color detector mounted on the front and oriented to the floor. A gyroscope was also mounted in the middle of the robot to keep track of the orientation. Although a servo motor was placed to rotate the sonar and measure the eight of the shapes, we didn't take advantage of this for reasons of instability.

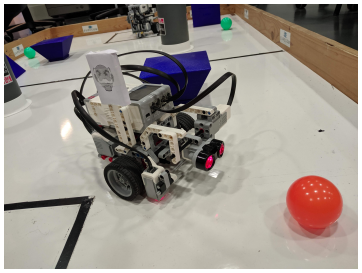

Figure 1: Mechanical Structure of The Robot

3 Motors and Sensors

Four main components were used in addition to the EV3 brick:

3.1 Tacho/Servo Motors

Tacho motors were the most documented component online, so it was the easiest component to write code for. We used different speeds but all below 100 to avoid any major perturbations for the gyroscope. For the braking mode we used "TACHO_BRAKE" (stopping without hard break) for same reason.

3.2 Sonar

The sonar is maybe the tightest bottleneck for this robot. Given the error prone and blurry feedback, and the relative complexity of the shapes, we had to make a lot of compromises to make it reach a minimal performance. The shapes didn't have a persistent signature on the sonar. The reflection of the sonar waves varies depending on the position of the shapes (rotation relative the vertical axe perpendicular to the floor) and on the presence of other sonar sensors on the field. Therefore scanning a cube for example doesn't have any persistent signature. The cylinder was relatively the easiest shape to recognize and identify due to its symmetry. The plot below shows the common problem with the sonar module we used. It is very inaccurate when reading the corners of the field, and the feedback on a empty field doesn't really reflect the reality of it. The dotted line shows the real dimensions of the field, while the blue line is the sonar reading.

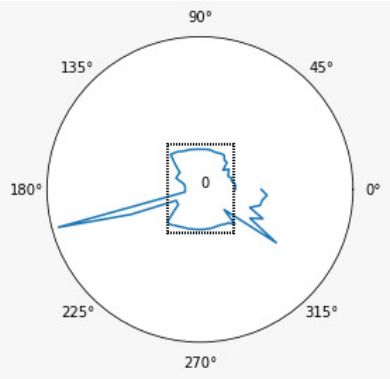

Figure 2: Sonar readings, with our play field depicted as square

3.3 Color sensor

The color sensor was used for initial testing to check for the back line separating the two fields, so that the robot does not cross to the other side.

3.4 Gyroscope

The gyroscope is used to detect the orientation of the robot in order to move in a specific direction or to detect the position of an object. It is very unstable and requires very slow movement in order to maintain a usable performance. Occasionally, it had to be reset, unplugged and re-plugged.

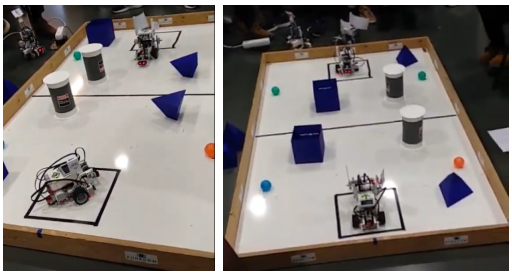

Figure 3: Scanning The Field

4 Software implementation

Our code has two main parts that are loaded in each test program: a small library that deals with all the motion related code that can be used in our main program(s), and a sensor data shared memory object, that is regularly updated by the sensor module that is compiled into a separate binary. This runs as a separate process.
The full code can be found on the Gitlab Repository of the project.

4.1 motion.c

In this file, we deal with all the motion related functions, we load the necessary libraries, we define the correct ports, and we initialize the hardware. To simplify the usage, we expose simple functions:

```
1) void pre()
2) int forward(int speed)
3) int backward(int speed)
4) int left_only(int speed)
5) int right_only(int speed)
6) int stop()
7) int stop_soft()
8) int servo_move(int position, int speed)
9) void done()
```

These functions only take the speed parameter and perform the action requested. motion.c is eventually imported into the test files, and used as fit.

4.2 sensors.c

Sensors.c is used differently than motion.c. It is actually compiled into a separate binary that runs independently from the rest of the program. Its only task is to read the input of the sensors, and put it on a shared memory thanks to System V shared memory. This memory is accessed later by the separate programs whenever they need input from the sensors. This modular approach

made developing much easier because we didn't have to worry about reading from the input or deal with any of the sensors libraries inside our main program. Sensors had to be launched separately before running any other program that requires sensors input. The structure written in the memory is the following:

```
typedef struct DATA{
  int color;
  int compass;
  int gyro;
  int sonar;
  int sensors_pid;
  int rotations_count;
  int state_left;
  int state_right;
  int angle;
} DATA;
```

When running two separate instances, the older instance exits automatically and let the newer instance take control over the shared memory to avoid any conflict when reading or updating the memory.

5 Algorithms and Terrain Exploration

The initial thought was to perform a grid scan, where the terrain is split into small regions and then explored traveling from region to another. Once over, an accurate trajectory could be generated to kick the balls while avoiding and identifying the shapes and their position.
Unfortunately this was not feasible using the sensors we have, therefore had to settle for a more basic approach. we imposed two main constraint to accomplish this:

1. Minimal movement: Once the robot is moving, we cannot track the exact position of the robot, and therefore it is very hard to go to any predefined position using only its coordinates. Hence We have to perform the minimum possible movements and avoid all extra motion.

2. Minimal speed and ramping: Every time we speed up or ramp up, the gyro suffers from great imbalance and perturbations and usually has to be reset manually. This second constraint is relatively more relaxed because we didn't necessary need high speeds anyway.

6 Initial Development and Pre-Game Tests

During the initial development phase we mainly focused on the detection of objects and balls. The first test, demonstrating the ability to find an object in our side succeeded. Also, we were able to shoot the balls at the predefined position and shoot it to the opposite side.

6.1 Test 1: Be able to find an object in your side

In order to find the object, we first tried to scan the surroundings while spinning counter-clockwise and approach the first occurrence of a sonar value above a certain threshold. Because this made it hard to identify objects further away from the closest wall, it was not feasible for use in-game.
Consequently, we tried a new technique to increase precision. Thus, we tried to establish a baseline instead. A technique which we relied heavily upon later in the game. Using this technique we deemed to stretch the potential of the sonar to its maximum with decent accuracy by comparing our fresh readings to the baseline readings. This baseline was established by performing the same motion with an empty field. We could then compare our readings during the test with the empty field instance. This should result us with the discrepancy of sonar values in certain angles, mapping to the location of the ball or potential objects.

Once we detect such an object, we approach the one with the largest discrepancy from the baseline.

6.2 Test 4: Be able to find a ball located at a predefined position

In order to find the ball, we first tried to scan the its surroundings while spinning counter-clockwise and approach the first occurrence of an increasing sonar value. Taking a small margin of inaccuracy and fluctuation into account.
This should theoretically work, because the wall approaches the robot, from its perspective. The ball would give even lower sonar values. On the contrary, when the sonar is reading its distance past the left side of the ball, it should read higher values, because the wall is further away from the ball. When that would be the case we can approach the ball.
However, our sonar sensor turned out to be insufficiently accurate in practice. It could not differentiate the small shape of the ball from the approaching wall.

Because the sonar delivered inaccurate readings from our initial starting position, and the ball position was predefined, the simplest solution is to approach the position of the ball. We do this using the gyro sensor instead of the sonar sensor. The angle can be determined using the gyro sensor which brings us closer to the ball.

6.3 Test 5: Be able to shoot a ball in the opposite side

This test is relatively simpler, because we can place the ball wherever we want. Our approach to this is to specify an angle and a distance, and then simply make the robot run in the ball direction and kick it.

First we start spinning, and check the gyro values till we reach the corresponding angle of the position of the ball. Then the robot accelerates and runs forward to the ball and kick it. When first implementing this test5, we didn't use the sonar because it was at a relatively high point, and wouldn't detect the presence of the small ball.

7 In-game Strategy

In order to minimize the movement, we had to perform the recognition task before moving from our initial position. This had two main advantages:

- We know our position exactly at the start of the game, so there was no room for error.
- We know that the gyroscope is perfectly stable and we can relatively perform 360 degrees spins with very low error margin.

Once performing a scan, it is compared to a baseline established before the actual game where we scanned the empty arena, this was done using the following procedure:

1. The arena was totally cleared from any objects.
2. "Ghost" obstacles were introduced at middle of the fields so that robot doesn't look beyond the black line and doesn't detect the objects on the other side.
3. The robot was put in a precise position that had to be remembered. This was essential because we had to use this position again for any future recognition task.
4. The sonar tilt angle was fixed to be perfectly horizontal and parallel to the floor so that we minimize the reflections on the floor. This angle had to be reset at each game to be sure that the scan is performed under the same conditions as the baseline was.
5. The robot spins 360 degrees, samples the values, averages them, and stores one value per degree.
6. The values are saved into a text file that will be reloaded for future use.

When comparing the scan during the game and the pre-established baseline, we added a tolerance coefficient to take into account the poor precision of the sonar and maybe any small variances between the two scans.
Unfortunate, our performance suffered a lot from false reads, where the sonar detects the waves sent from the robot on the other end, and eventually identifies them as objects.
Once comparing the two scans, the program generate a bitmap of 0s and 1s, where the 1s indicate the presence of an object. We scan through the bitmap and select the 3 biggest contiguous clusters of 1s.
Preliminary tests indicated that cubes usually have the widest signature, followed by the cylinders, and then the pyramids. Therefore, we report the biggest cluster as a cube, the next one as a cylinder, and the last one as a pyramid. The position is easily generated using the angle pointing to the middle of the cluster, the distance at this angle (from the scan performed at the start of the

game). These results are then reported to the server before any further action. Once the data sent, the robot performs a uses the scan results one last time to decide if there are objects on its right and/or left side. If no objects are detected, the robot takes a predefined trajectory and kicks the ball(s) into the other adversary's field. So the robot can either kick one ball, both of them, or none.

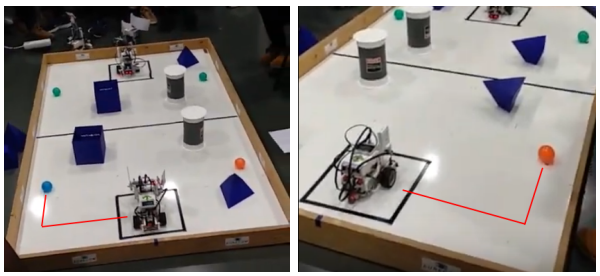

Figure 4: Predefined Trajectories to Hit The Ball

8 Results

The robot scored decent results in the final game and lost only two matches (to the same team). Our results were greatly thanks to the simple yet efficient strategy that we adopted and that was very fit to the game. The robot performed very well when it comes to shooting the balls, and was actually the only robot that succeeded in shooting the balls.
Our performance suffered mostly by the lack of decent object identification. This is mainly due to our simple approach to this, where we failed to distinguish the different configurations of pyramids, and even failed to distinguish pyramids from a cube. Our results were average on the identifying the position.

9 Conclusion

This project was very fun to work on, and provided a good opportunity to play around with the features of our Debian Linux operating system. The sonar was a tight bottleneck that limited our performance, but we managed to work around it.