RMI report

Pedro Gonçalves[88859] and Pedro Silva[89228]

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro, 3800 Aveiro, Portugal

Abstract. In this report we will describe the different approaches and logic implemented to complete each of the 3 different challenges proposed on the first assignment of RMI. This implementation is based on the given objectives and the different types of data (obstacle sensors' values, compass and gps coordinates). To move the robot, we send a speed value for each of the 2 robot motors. On the first challenge, that value is sent according to the obstacles that the sensors read. On the challenge 2 we approach the navigation based on a noiseless compass and gps, to discover the entire map, with the help of an a-star algorithm to send our robot to the positions he hasn't explored yet. Finally, we print the entire map to a text file. For the last challenge, we have to navigate through the map once more, this time to find all the beacons in it, and, with the help of the same a-star algorithm, discover the best closed path that allows to visit all the beacons, starting and ending in the same starting spot and print those coordinates to a text file.

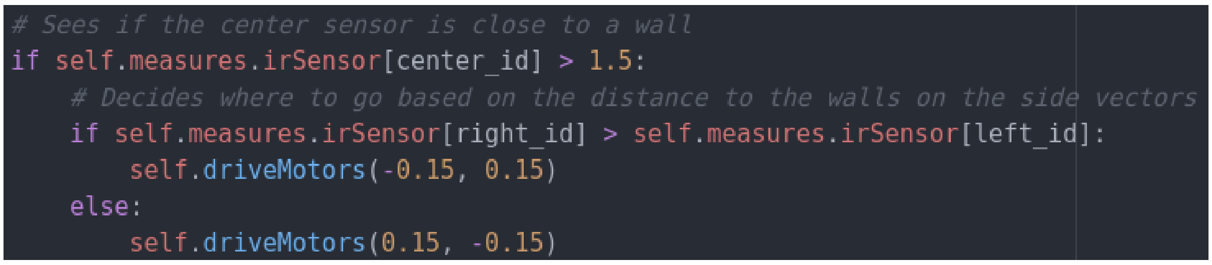
1 C1 challenge

1.1 Implementation

In the first challenge the goal was to make the robot go as fast as possible without colliding. To do that, all the decisions had to be done considering the values given by the obstacle sensors.

1.2 Movement

The code is based on an if-else block, with the first condition having higher priority than the next ones and so on. The first condition, and the most important one (see Fig. 1), checks if the value from the front obstacle sensor is high (the robot is close to the wall) and then, if true, proceeds to rotate to the side that has the lowest value on the obstacle sensor, the one that is further from the walls. The other conditions are there to adjust his position on a straight line, because sometimes the robot can get close to the side walls.

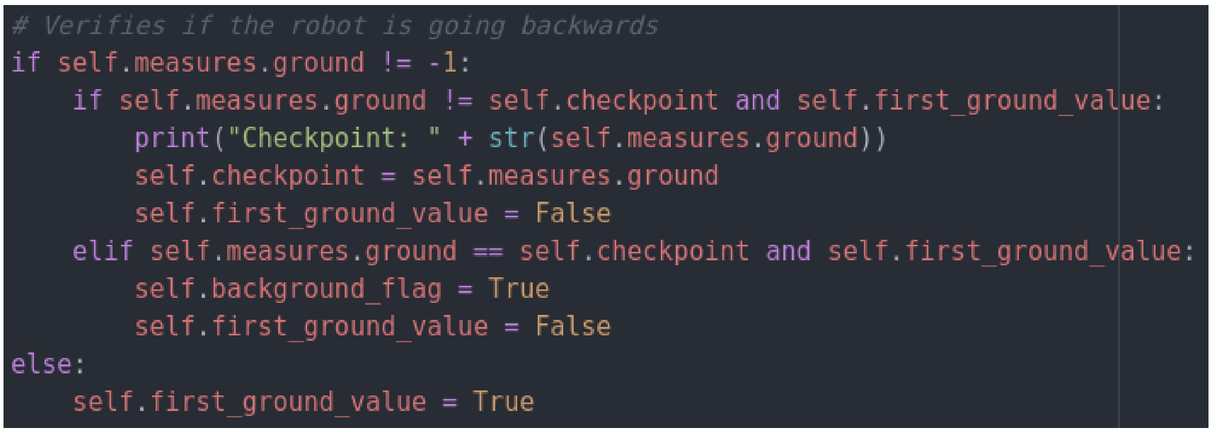


```python
# Sees if the center sensor is close to a wall
if self.measures.irSensor[center_id] > 1.5:
    # Decides where to go based on the distance to the walls on the side vectors
    if self.measures.irSensor[right_id] > self.measures.irSensor[left_id]:
        self.driveMotors(-0.15, 0.15)
    else:
        self.driveMotors(0.15, -0.15)
```

Fig. 1. Most important condition for the robot movement

1.3 Wrong way detection

Sometimes the robot can approach a front wall and make a turn that leaves him facing the wrong way on the map. To avoid going backwards, we check the ground sensor. If the ground sensor detects an unexpected value, then we know that the robot is not going in the right direction (see Fig. 2). After that, the robot enters a turnaround phase to go back to the pretended direction.



```python
# Verifies if the robot is going backwards
if self.measures.ground != -1:
    if self.measures.ground != self.checkpoint and self.first_ground_value:
        print("Checkpoint: " + str(self.measures.ground))
        self.checkpoint = self.measures.ground
        self.first_ground_value = False
    elif self.measures.ground == self.checkpoint and self.first_ground_value:
        self.background_flag = True
        self.first_ground_value = False
else:
    self.first_ground_value = True
```

Fig. 2. Code that detects if the robot is going backwards

1.4 Turnaround

In this phase, we store the values from the front and back sensors of the robot when we detect that it is backwards for the first time (when he passes the wrong checkpoint). After that, we start rotating the robot until his front sensor value equals the value from the back sensor stored and vice versa.

2 C2 challenge

2.1 Implementation

On the second challenge the first thing we did was changing the right and left sensor's location to 90º and -90º, so that they would all have a 90º angle. Then, we started reading the given gps and compass to navigate steadily on the center of each cell. Our priorities were rather simple: on each cell the robot would stop and read all the sensors, if he could, he would first go to the left, then go in front, then go to the right and lastly he would turn back if no other option was available.

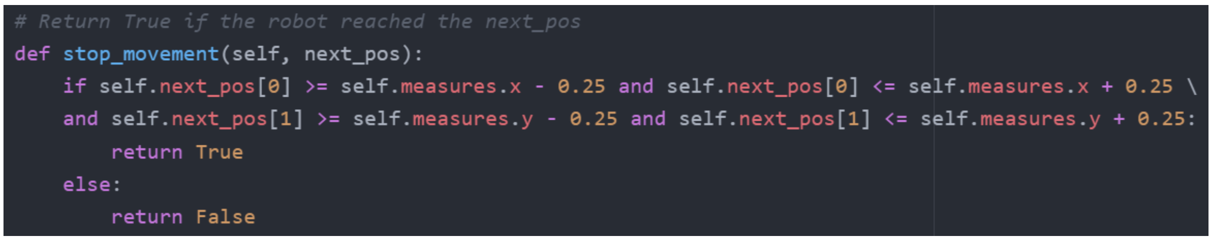
2.2 Movement

To move the robot to where we wanted, we have to read the compass and the sensors in every cell, because if we want him to go to the left and he is turned to the left, we simply have to give the motors an equal speed value and he will go to that position, but if he is turned to the right or backwards we have to rotate him.

Therefore, we created a function *turn()* that receives 2 parameters, *degrees* and *direction*, the first one is an integer with the value the robot has to rotate, and the second is a string with the direction (left or right) of the rotation, so that he always rotates as fast as possible. This function is very precise so the compass will always be 0, 90, -90 or -180 on return.

After the *turn()* function, the robot is facing the correct direction, so all we have to do is send him in front "*self.driveMotors(0.13, 0.13)*" until he is in the center of the next cell.

Due to the robot motors noise, we had to create a function *stop_movement()* that checks if the robot has reached the position passed as an argument (see Fig. 3). This position has an associated error, since, even though the gps and compass have no noise, the robot motors have some, so the robot won't stop exactly in the center of the cells.



```
# Return True if the robot reached the next_pos
def stop_movement(self, next_pos):
    if self.next_pos[0] >= self.measures.x - 0.25 and self.next_pos[0] <= self.measures.x + 0.25 \
    and self.next_pos[1] >= self.measures.y - 0.25 and self.next_pos[1] <= self.measures.y + 0.25:
        return True
    else:
        return False
```

Fig. 3. Condition to stop the movement

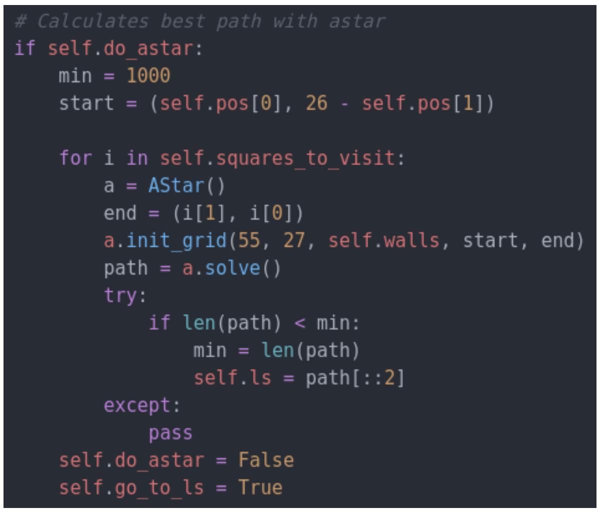
2.3 Intersections

We can already tell that the solution above has some flaws, the robot will always go through the same path. So we decided to store all the positions the robot hasn't been in on a list (*self.squares_to_visit*). In every cell, the robot reads all the positions and stores the ones he can walk on, and when he visits any cell on the list, we remove the cell from it, that way the list is always updated with only positions the robot hasn't visited.

Then, we created a logic so that, if the robot has already been in the next position he is going to, he calculates the closest path to the positions on the list *self.squares_to_visit*, with a-star, and goes to the position with the shortest path.

2.4 A-star

The A-star algorithm we implemented [1] takes as arguments the *width*, *height*, *walls*, *start* position and *end* position of the maze. We only had a small problem with this algorithm, since we don't know the walls on the beginning of the map we have to consider that all the cells on the map arewalls, and as the robot is discovering the map, we are removing the visited position from the *walls* list. Now it's rather simple, we just need to pass the start position, which is the position the robot is in and the end position, which is a position in the *self.squares_to_visit* list (see Fig. 4).



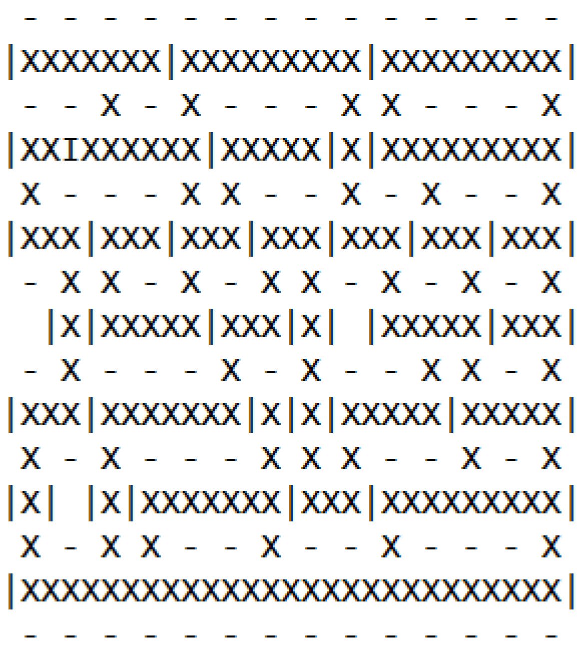
```python
# Calculates best path with astar
if self.do_astar:
    min = 1000
    start = (self.pos[0], 26 - self.pos[1])

    for i in self.squares_to_visit:
        a = AStar()
        end = (i[1], i[0])
        a.init_grid(55, 27, self.walls, start, end)
        path = a.solve()
        try:
            if len(path) < min:
                min = len(path)
                self.ls = path[::2]
        except:
            pass
    self.do_astar = False
    self.go_to_ls = True
```

Fig. 4. A-star call to calculate the shortest path to destination

To make the robot go to the a-star calculated position, is once more a combination of the *turn* and *stop_movement* functions.

2.5 Drawing the map

To draw the map as intended, we started by reading the starting position and mark the walkable stops with an ‘X’ and the walls with an ‘-’ if they’re horizontal and ‘|’ if they're vertical. Then we just had to do the same procedure for every cell and print the final result to a text file, ending up with the following maze(see Fig. 5).



```
 - - - - - - - - - - - - - -
|XXXXXXX|XXXXXXXXX|XXXXXXXXX|
 - - X - X - - - X X - - - X
|XXIXXXXXX|XXXXX|X|XXXXXXXXX|
 X - - - X X - - X - X - - X
|XXX|XXX|XXX|XXX|XXX|XXX|XXX|
 - X X - X - X X - X - X - X
  |X|XXXXX|XXX|X| |XXXXX|XXX|
 - X - - - X - X - - X X - X
|XXX|XXXXXXX|X|X|XXXXX|XXXXX|
 X - X - - - X X X - - X - X
|X| |X|XXXXXXX|XXX|XXXXXXXXX|
 X - X X - - X - - X - - - X
|XXXXXXXXXXXXXXXXXXXXXXXXXXX|
 - - - - - - - - - - - - - -
```

Fig. 5. Maze printed on text file

3 C3 challenge

3.1 Implementation

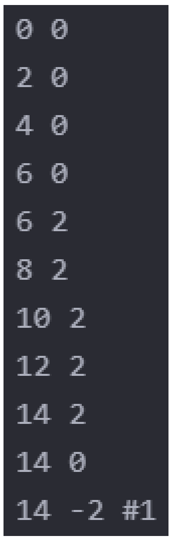
For the last challenge, we used the same logic as in challenge 2. The only difference being that in this one we had to discover the best closed path that allowed the robot to visit all the beacons, starting and ending in the same starting spot.

3.2 Best path

Every time the robot finds a beacon, we store the position in a list (*self.beacons_ls*), and when all beacons are found, and the map is entirely discovered, we use the a-star algorithm to find the best closed path.

3.3 Print to file

Finally, when we have the coordinates, we just print them to a text file as asked, with an identifier for the beacon coordinates (*#beacon_id*) (see Fig. 6).



```
0 0
2 0
4 0
6 0
6 2
8 2
10 2
12 2
14 2
14 0
14 -2 #1
```

Fig. 6. Coordinates to the first beacon.

References

1. https://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/, last accessed 2021/11/15.