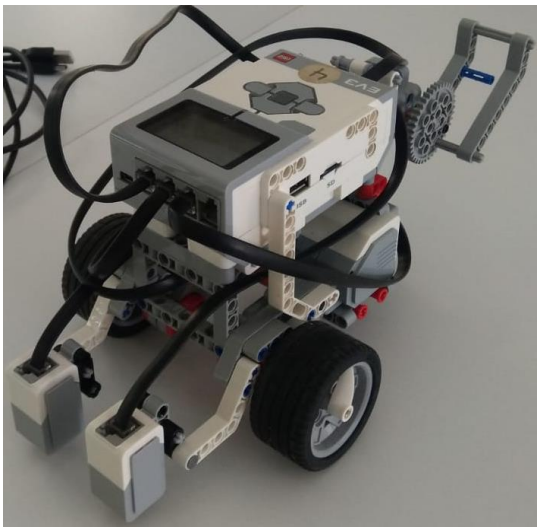
Autonomous University of San Luis Potosí

Ant simulator robot



Full name: Mendoza Rodríguez Fernando

Student´s ID: 285476

Course: Intelligent Robotics

Profesor: Dr. Núñez Varela José Ignacio

Abstract

In this project you will find an implementation of the collecting food ant's task simulation, which consist of a LEGO Mindstorms EV3 robot build up with a LEGO Kit from a behavior-based approach. When ants look for food carry out some behaviors to complete the task. Ant collecting food behavior is a good example of how Bioinspired systems are a very interesting area of study for intelligent systems, specifically for reactive (behavior-based) agents due to the simplicity of the concept behind.

1 Introduction

1.1 Problem description

Ants look for food in a very particular way, seen from a top-down approach they walk randomly until they find it, then they go back to their anthill by remembering the number of steps that they performed as well as the sun position and some visual landmarks, at the same time they leave a pheromone trace, which is used by other ants to know the best path to reach food location ("Ant," 2021).

Figure 1. Ants following a trace.

1.2 Project's Goal

The objective of this project is to build a bioinspired robot with a LEGO Mindstorms EV3 kit according to a reactive (behavior based) approach from intelligent robotics, capable of simulate the task of searching for food from ants.

To reach the objective is necessary to perform some tests over certain elements (sensors and actuators) of the robot, these tests will allow us to get information about such elements' behavior, like the accuracy of the data retrieved by the sensors from the environment and the movement precision of the motors.

2 Project Characterization

2.1 Task description

The robot is going to simulate the behaviors performed by ants when they are searching for food, it will move randomly starting from a specific point in the environment (the "anthill"), until it finds food, once reached that objective it will return to the anthill.

2.2 Environment description

The robot will be placed over a smooth light-colored surface, this is because the robot navigates and carries out its behaviors according to certain landmarks, which are detected by a color sensor, so colors must be easily distinguishable from the surface, these are the different elements on the environment (Figure 2).

- Landmarks: I use color sheets for pointing the "places" the ant must know, a blue sheet is used to represent the anthill, 2 red sheets are used to represent the places with food.
- Borders: The environment has logical borders, this is achieved by using black insulating tape, so the ant can detect the color and turn an amount of degrees to avoid it.
- Trace: A first approximation of the project was to build up the environment over a whiteboard, so the robot could leave a trace by placing down a marker. Due the robot slipped when it tried to move, I changed the whiteboard for a wood plank and I set a scenario with two sources of food, one of them already has a trace towards the anthill, it will allow the robot to show both behaviors, when it arrives at food by searching it and by following an existing trace.

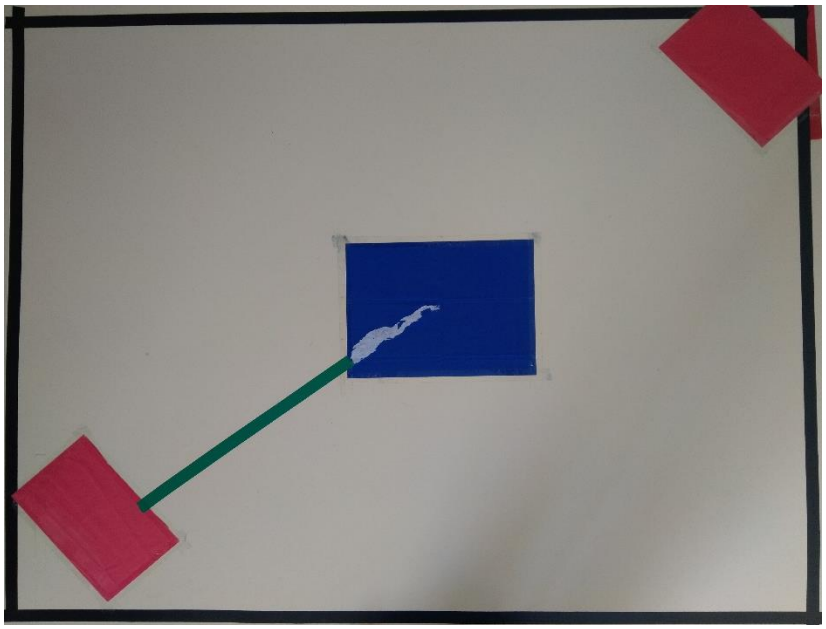


Figure 2. Robot's environment.

The environment has 114 cm x 90.5 cm and 6 cm border with the purpose of prevent robot to go out of the table while avoiding border (Figure 3).

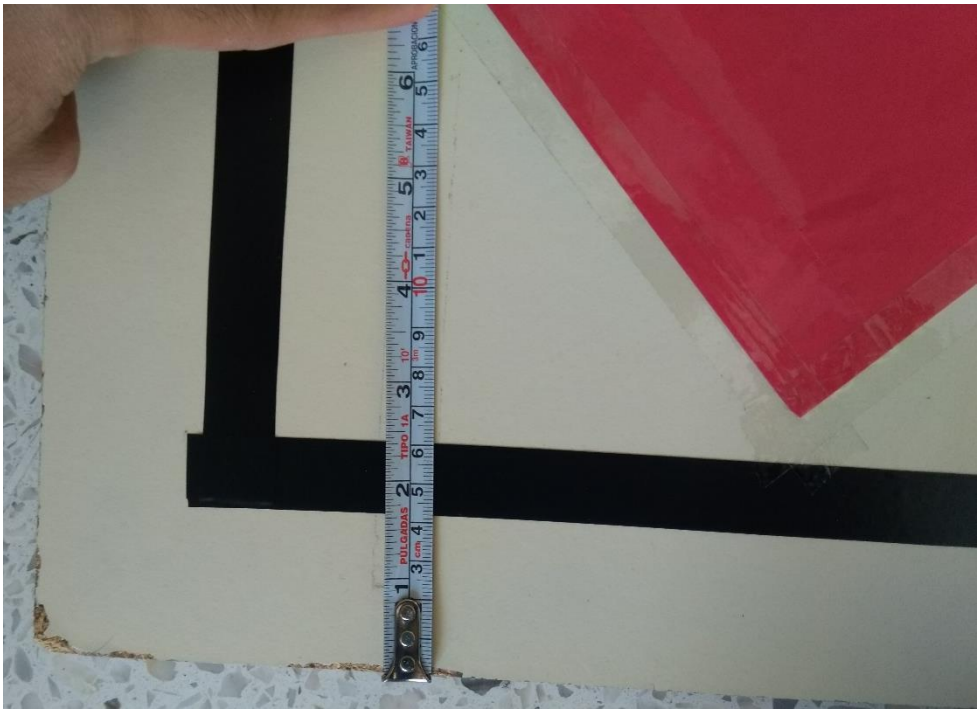


Figure 3. Border measurement.

2.3 Robot description

The LEGO Mindstorms EV3 kit (pieces, sensors, and actuators) can be used for building many kinds of robots, LEGO has a template model for mobile robots with differential traction (Figure 4), which is used in several examples in LEGO documentation (LEGO® MINDSTORMS® EV3 Hardware Developer Kit.). We are going to take it and add two color sensors that will be used to get information of the world and we will also add another motor, it will be used to place down the marker on the surface and raise it. On this project the robot is programed using Python language, it has been made possible by MicroPython, which is a lite implementation of Python 3 Language designed for being used on microcontrollers, it includes a small subset of python standard library.

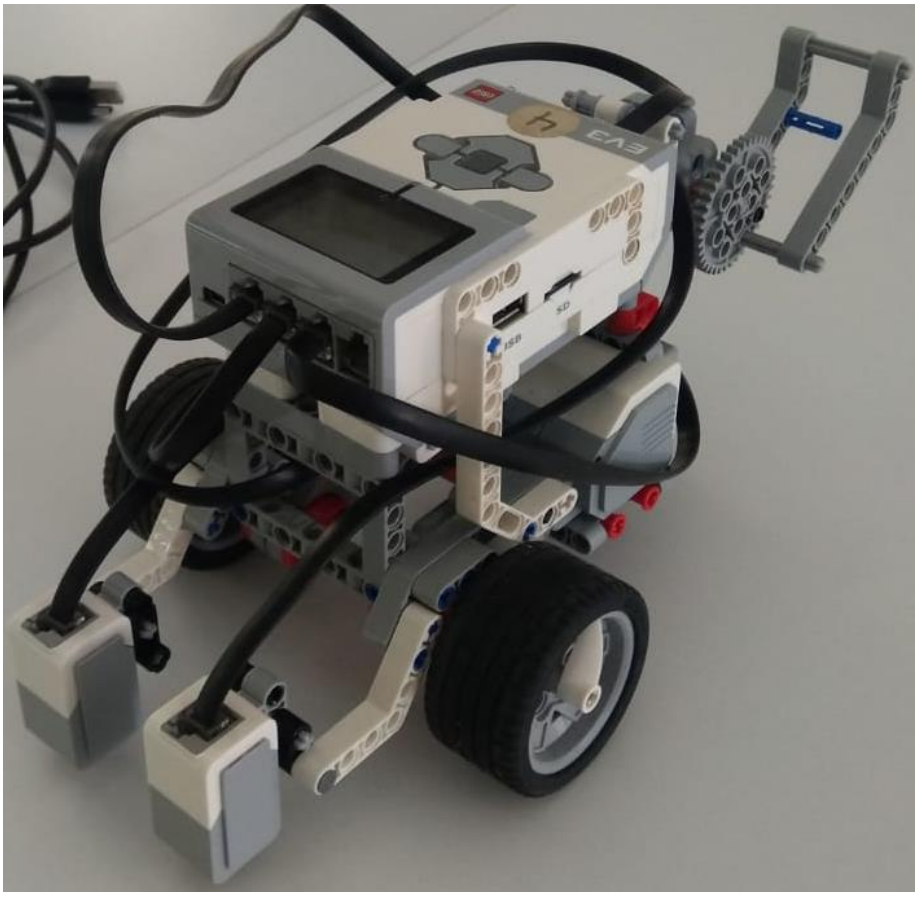


Figure 4. Ant robot.

2.3.1 Sensors

Color sensor (Figure 5): The robot needs color sensors to know where it is by detecting landmarks (black and green insulating tape, blue and red sheets). As can be seen in Figure 6 we use two sensor on the robot. Since we need to know how good is the information retrieved by the sensor (because that's the way we can say if it is accurate or not and if we must manage the error or can simply trust on the sensor), it is necessary to carry out some test over it.

Figure 5. Color sensor (LEGO®).

As we can find in LEGO documentation (LEGO® MINDSTORMS® EV3 Hardware Developer Kit.) it can deliver until 1000 samples per second (1khz) depending on the light wavelength, by default it can distinguish between 8 different colors. We can also use *EV3 Devices* Module from MicroPython library and get RGB values, which consist of 3 values (red, green, and blue), these values are 3 percentage integers which range is from 0 to 100.

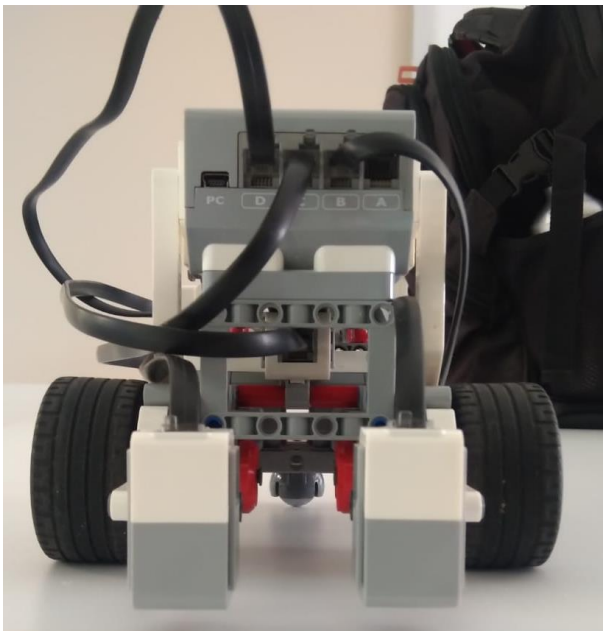


Figure 6. Color sensors in the robot.

In this case I performed measurements on certain red, green, blue, and black tonalities I'm interested in (the ones from the paper sheets and from the insulating tapes that represent the anthill, food sources, the trace, and the border), all measurements were taken under warm and cold light, for each set of samples I took 2016 observations (because this is the smaller amount among all the datasets.

For red color I obtained this distribution of values (Tables 1 and 2, Figures, 7 and 8), which means for example that for red color under warm light (Figure 7) at the RGB's red channel there were 1481 observations with a value of 57 and 535 with a value of 58, at the RGB's green channel there were 1279 observations with a value of 10 and 737 observations with a value of 11 and for RGB's blue channel there were 21 observations with a value of 7, 1992 observations with a value of 8 and 3 observations with a value of 9.

Color	Red Sheet (Warm Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	57	1481	10	1279	7	21
	58	535	11	737	8	1992
					9	3

Table 1. RGB values counting, red color under warm light.

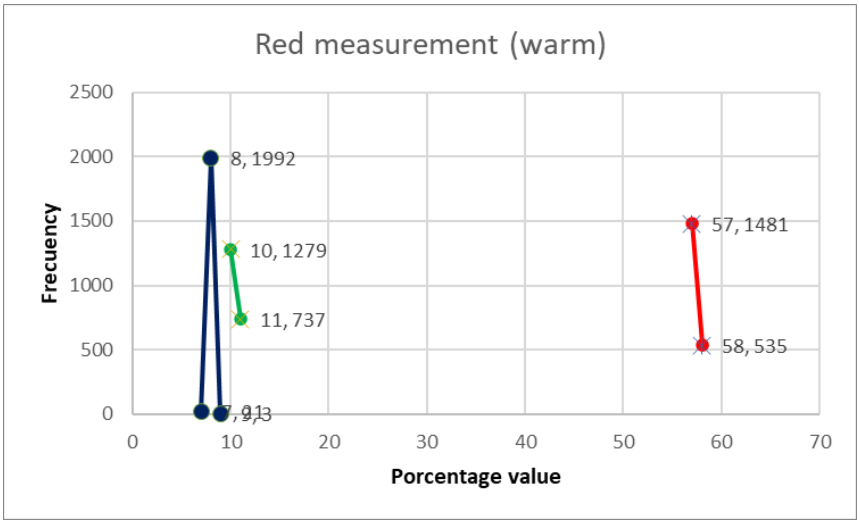


Figure 7. RGB channels dispersion, red color under warm light.

Color	Red Sheet (Cold Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	58	42	10	2015	7	110
	59	1194	11	1	8	1906
	60	780				

Table 2. RGB values counting, red color under cold light.

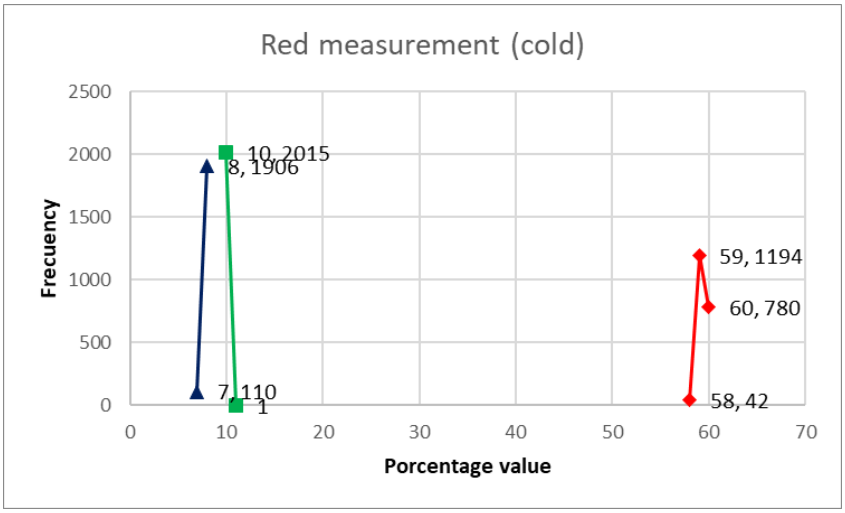


Figure 8. RGB channels dispersion, red color under cold light.

For green color in the same conditions, we obtain the information from Tables 3 and 4, we also have the graphics from Figures 9 and 10 showing graphically the data. The main idea of this test is to get a central value and a tolerance which will be the biggest difference from the central value against the highest value and the central value against the lowest value, for example in Green color on green channel, taking in account both counting tables (Figure 11 and 13) the most representative value is 37 with a frequency of 1334, the highest difference with the extreme values is 37 – 35 = 2, so our tolerance will be +/-2 on the green channel for green color.

Color	Green tape (Warm Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	10	1870	35	58	17	498
	11	146	36	624	18	1518
			37	1334		

Table 3. RGB values counting, green color under warm light.

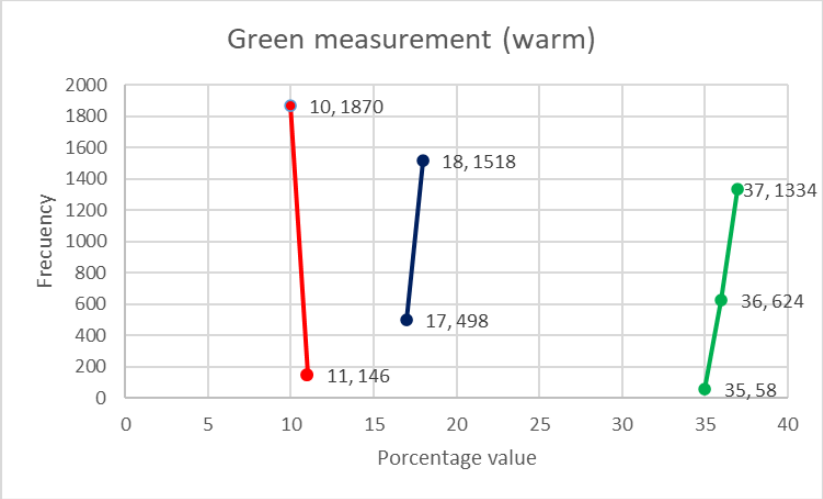


Figure 9. RGB channels dispersion, green color under warm light.

Color	Green tape (Cold Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	10	84	37	855	18	951
	11	1932	38	1161	19	1065

Table 4. RGB values counting, green color under cold light

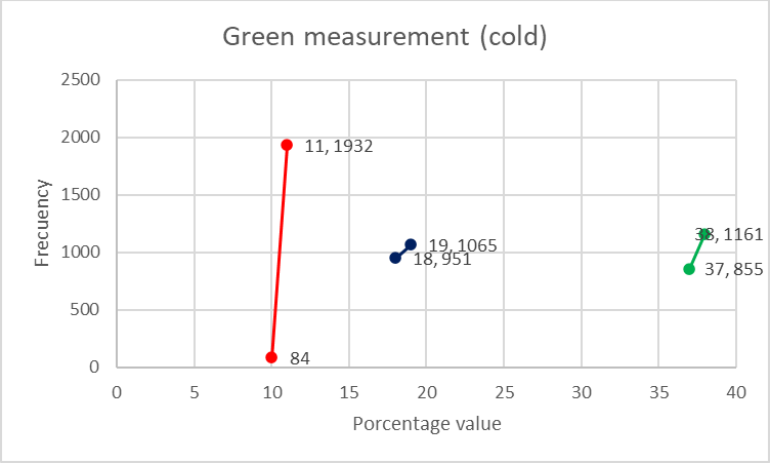


Figure 10. RGB channels dispersion, green color under cold light.

Next in the Tables 5 and 6 we can see the counting tables for the obtained data for blue color. In figures 11 and 12 we can quickly see the analysis. For this color we have the central values red = 5 with tolerance = +/- 1, green = 11 with tolerance = +/- 1 and blue = 28 with tolerance = +/-2.

Color	Blue Sheet (Warm Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	5	2016	10	4	27	156
			11	2012	28	1849
					29	11

Table 5. RGB values counting, blue color under warm light.

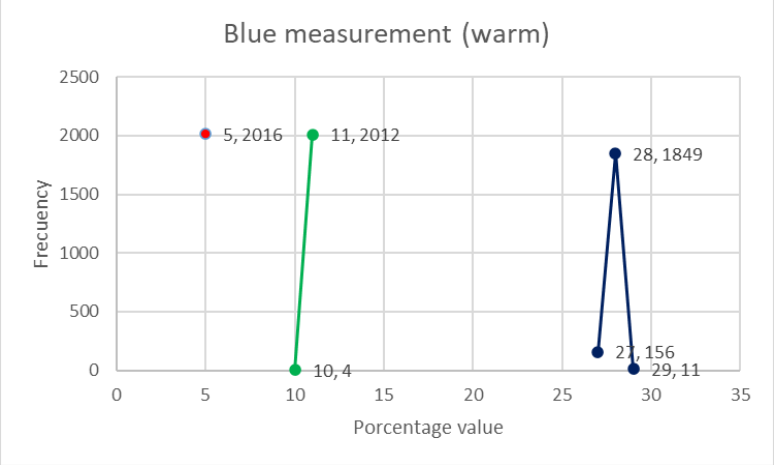


Figure 11. RGB channels dispersion, blue color under warm light.

Color	Blue Sheet (Cold Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	4	4	10	1190	26	18
	5	2012	11	809	27	1819
			12	17	28	14
					29	165

Table 6. RGB values counting, blue color under cold light.

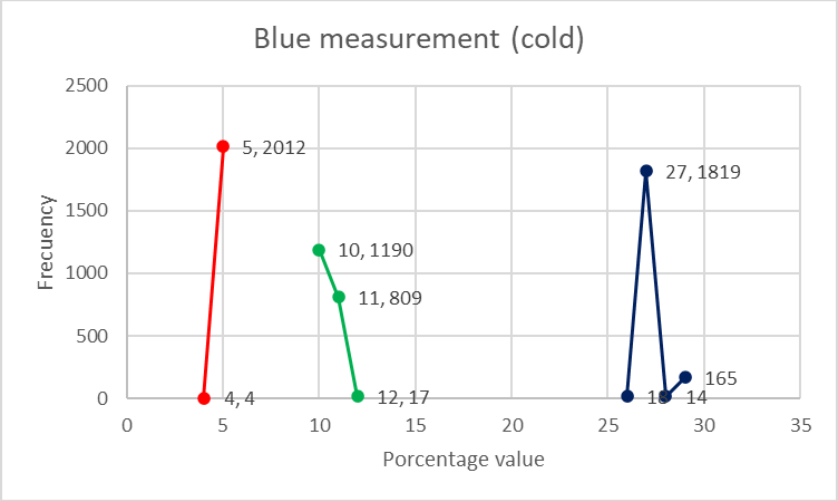


Figure 12. RGB channels dispersion, blue color under cold light.

Finally for black insulating tape we have the values counting tables in Tables 7 and 8, the graphics for the tables are the figures 20 and 22.

Color	Black tape (Warm Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	6	1910	7	2016	3	49
	7	106			4	1967

Table 7. RGB values counting, black color under warm light.

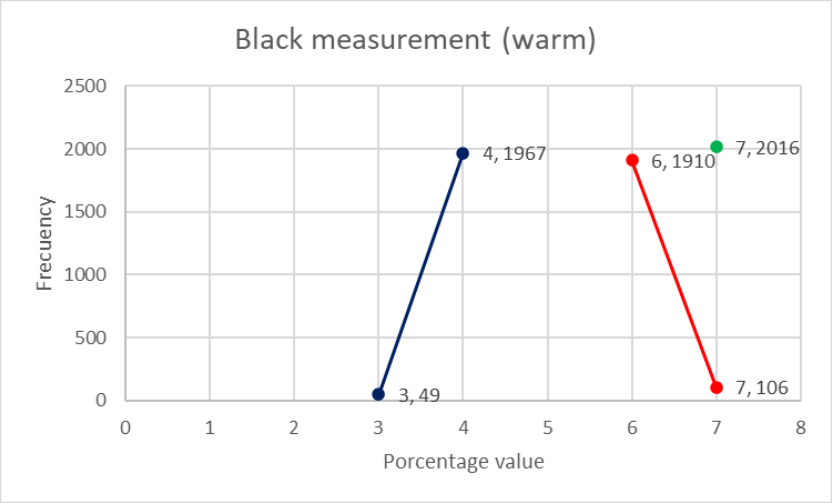


Figure 13. RGB channels dispersion, black color under warm light

Color	Black tape (Cold Light)					
RGB Chanel	Red		Green		Blue	
Observation's values/count	6	1863	7	2015	3	51
	7	153	8	1	4	1965

Table 8. RGB values counting, black color under warm light.

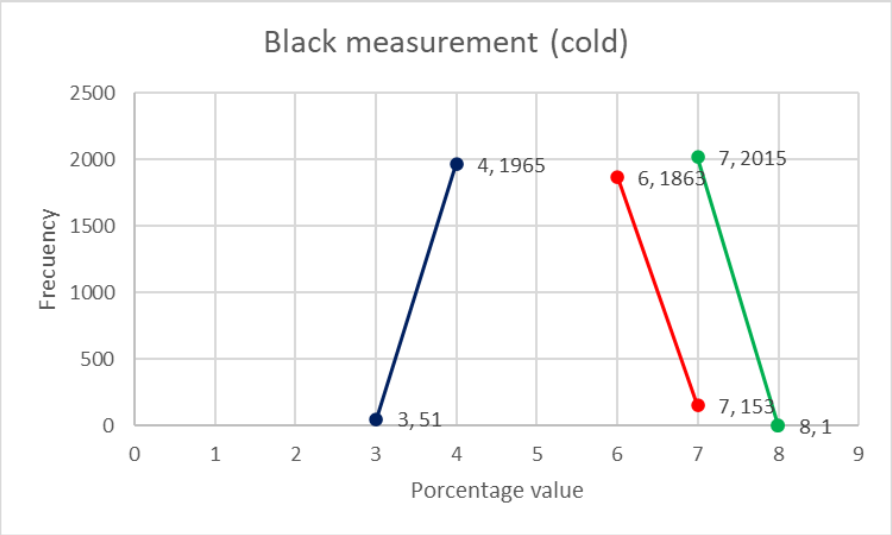


Figure 14. RGB channels dispersion, black color under cold light.

Results

Color	Central values (Red,Green,Blue)	Tolerance values (Red,Green,Blue)
Red	(57, 10, 8)	(+/-3, +/-1, +/-1)
Green	(11, 37, 18)	(+/-1, +/-2, +/-1)
Blue	(5, 11, 28)	(+/-1, +/-1, +/-2)
Black	(6, 7, 4)	(+/-1, +/-1, +/-1)

Table 9. Color sensor test results.

After the analysis of the data now we have these central values with tolerances for each RGB channel (Table 9), which will be used to classify all the inputs that the color sensor provides.

Gyro sensor (Figure 15):

The behavior of the robot includes to turn an amount of degrees in certain moments and next move straight in the new direction, due the movement of wheels is not 100% precise it would be impossible to make it turn the exact degrees just with calculated movement time, because of that it is needed to add a gyroscope to the robot (Figure 16).

Figure 15. LEGO Gyro sensor (LEGO®)

As I found on documentation (LEGO® MINDSTORMS® EV3 Hardware Developer Kit.) it has an accuracy of plus/minus 3 degrees within 90 degrees and supports maximum 440 deg/s angular speed, (which means any faster turning will not be correctly measured).

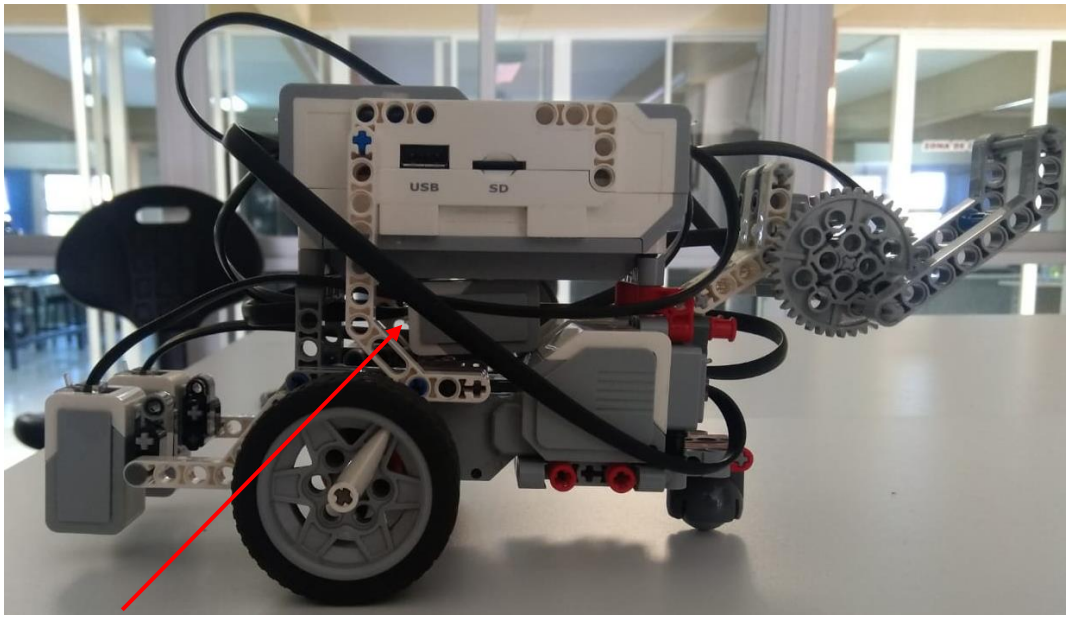


Figure 16. Gyro sensor implemented on the robot

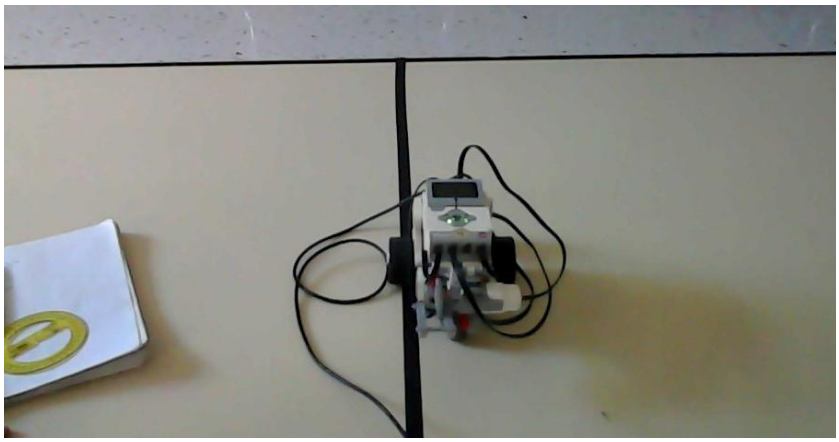


Figure 17. Angle rotation test scenario

Even so I decided to perform some measurements to know how good the sensor is (Figure 17), I programed the robot to turn with a 100 deg/s speed using 180° and 360° as the target angle, these were the results (Table 10), as we can see everything is within the manufacturer tolerance margin:

Target Angle (°)	Measured Angle (°)	Margin (°)
-180	-179	+/- 6
	-179	
	-178	
	-181	
	-179	
180	184	
	185	
	183	
	186	
	182	
-360	-359	+/- 12
	-356	
	-358	
	-358	
	-355	
360	364	
	363	
	368	
	367	
	366	

Table 7: Rotation test results

2.3.2 Actuators

Large LEGO servo motors (Figure 18):

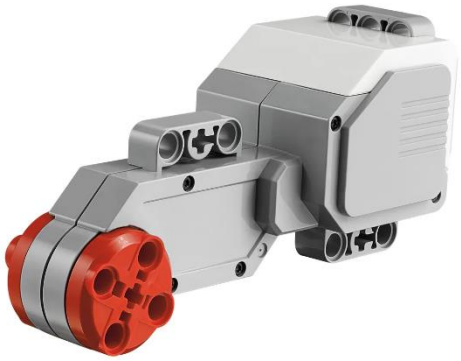


Figure 18. Lego servo motors (LEGO®)

The robot has 2 of these motors used for the traction of the wheels, they can be controlled with LEGO's Python library, we can choose between use the *Robotics* module and the *EV3 Devices* module. The first one can be used with differential traction robots, it is necessary to specify certain robot´s features, the space between both wheels, and wheel's diameter, after that we can use some high-level functions provided by LEGO, like driving with a specific direction and speed. The second module give us more specific and low-level instructions, therefore is possible to control each motor with different specifications.

Test 1 (robotics module)

I did several tests with regarding robot's navigation, the first one was to make the robot to move straight with the *Robotics* module from MicroPython library:

```
ant = DriveBase(rightMo, leftMo, wheel_diameter=56, axle_track=93)
ant.straight(3000)
ev3.speaker.beep(1000, 2000)
ant.straight(3000)
```

Basically, the robot moves straight for 3 seconds, then stop and emit a beep sound for 2 seconds (time we use to set a landmark on current robot position) and then moves again for other 3 seconds, the point where it ends is the final point.

We get the Middle point, and the end point coordinates (taking as a fact the first position is located at origin). My hypothesis is that if the robot performs a curvy trajectory, it will draw an arc due that it is caused by a constant difference of both motors' power along the time and it is possible to calculate the size for the circle which it belongs to, the more pronounced the arc is the smaller is the circle and

therefore the higher is the difference between both motors' power. The following coordinates belong to the measured points, they are given in cm and can be seen on the graph (Figure 19):

$$MidPoint = (-7,295.4)$$

$$EndPoint = (-3.5,590.4)$$

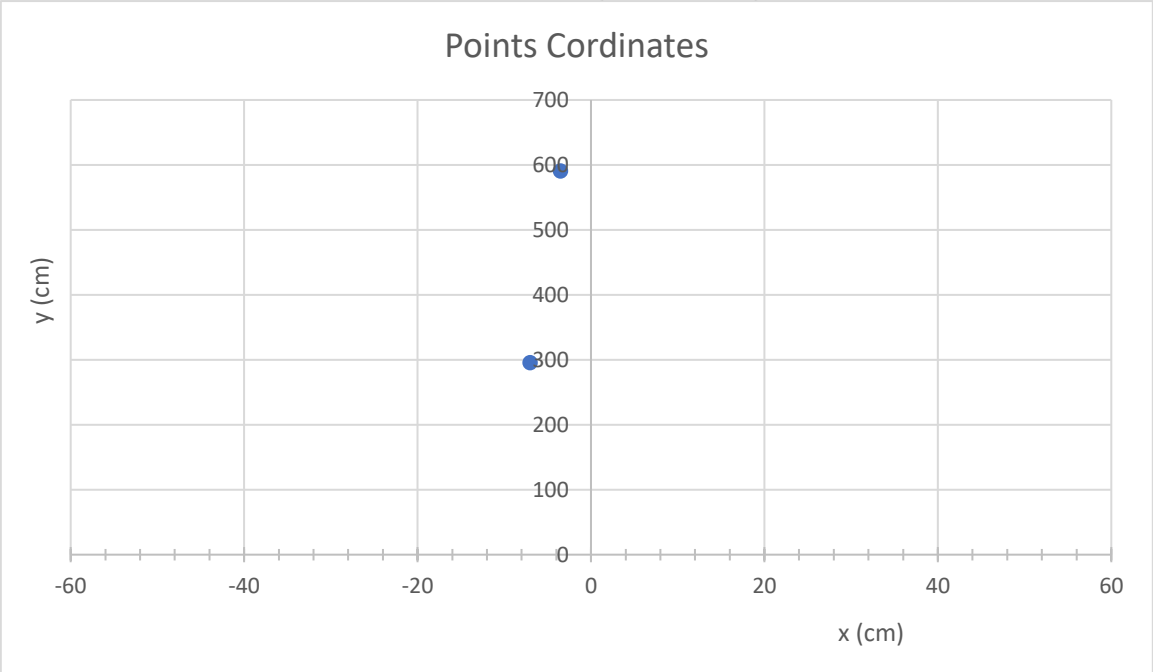


Figure 19. Curve robot trajectory

The method I used to obtain the circle measurements is the intersecting chords theorem (Figure 20).

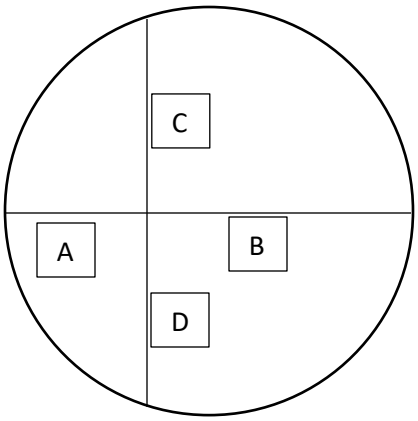


Figure 10. Intersecting chords theorem illustration.

According to the image above we can resume the theorem to the following equation,

$$A * B = C * D,$$

we can see the distance between the origin and the endpoint as a chord and the distance from middle point to the chord as sagitta (Figure 21).

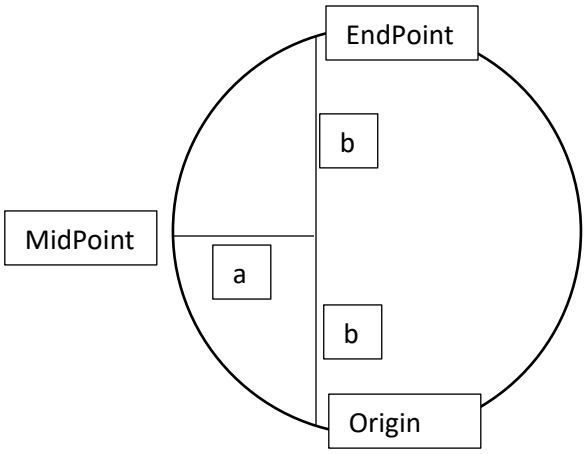


Figure 11. Intersecting chords aplication illustration

We know that sagitta is just a segment of the diameter, which at the same time can be seen as 2 times the radio, so we can say that,

$$b * b = a * (2r - a),$$

and solving for r,

$$r = \frac{a^2+b^2}{2a}.$$

As we can see in the graphic (Figure 19), the chord is not located over an axis, to make all calculous easier to understand I applied a matrix transformation for rotation. Remembering that chord starts from origin we can get the correct angle by obtaining the deviation angle between the desired axis and the endpoint it is given by the inverse tangent of the chord line tangent,

$$\theta = \tan^{-1} \frac{endPoint_x}{endPoint_y}.$$

The post-rotation points are given by the following multiplication (each point with the transformation matrix)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix},$$

in this case Θ = 0.34° and these are the rotated values:

$$MidPoint' = (-5.2487, 295.4363)$$

$$EndPoint' = (0, 590.41)$$

And the radius from the circle is:

$$r = 8304.298\ cm.$$

By having the radius now, we can calculate the difference between both wheels speed. We know that the distance between wheels is 9.3 cm, both draw a circular trajectory (Figure 22), the fastest one is the external one (because if it's faster than the other one then it covers more distance at the same time).

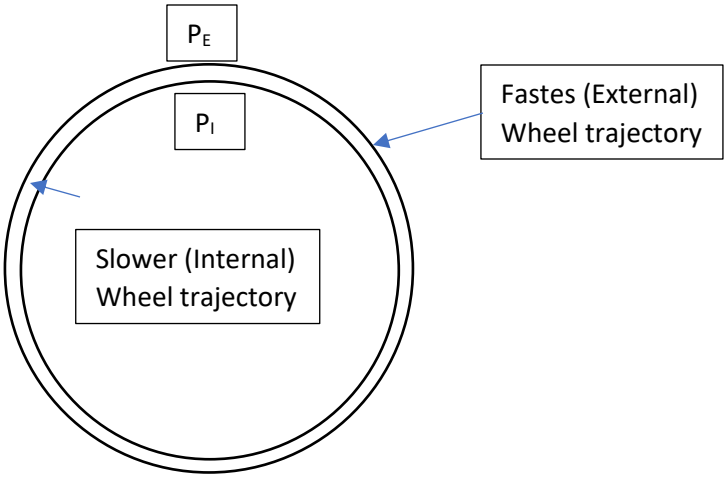


Figure 12. Robot Wheels analysis illustration

Let´s name each perimeter as P_E and P_I, we know speed formula is,

$$V = \frac{d}{t} = \frac{P}{t},$$

then, if the time for movement of both wheels is the same, we have,

$$t = \frac{P_E}{V_E} = \frac{P_I}{V_I}.$$

therefore, both velocities and both perimeters are proportional,

$$P_E = kP_I, \qquad V_E = kV_I,$$

and

$$k = \frac{P_E}{P_I}.$$

Taking in account,

$$P = 2 * r * \pi,$$

then,

$$\boldsymbol{k = \frac{2*\pi*r_E}{2*\pi*r_I} = \frac{r_E}{r_I}},$$

in the case of this robot the difference between wheels is 9.3 cm, therefore the external radius is 9.3/2 cm bigger than the one measured on the test, and the internal is 9.3/2 cm smaller.

$$k = \frac{r+4.65\ cm}{r-4.65\ cm}.$$

The result is that external motor (which in this experiment is the left motor) is k=1.00112 times faster than internal one (the right motor).

Test 2 (EV3 **devices module)**

Now that we know that the *Robotics* module is not useful for this robot, the next step is to use the *EV3 Devices* module and compensate the error. The experiment follows the same idea than the previous test, but in this case, speed value is set for each wheel, we must compensate with a factor for the right motor (which is the slower one):

```
velL=320
factor=1.00112
velR=velL*factor
time=15000
leftMo.run_time(velL, time, then=Stop.BRAKE, wait=False)
rightMo.run_time(velR, time, then=Stop.BRAKE, wait=True)
ev3.speaker.beep(1000, 1000)
rightMo.run_time(velR, time, then=Stop.BRAKE, wait=False)
leftMo.run_time(velL, time, then=Stop.BRAKE, wait=True)
```

I also wrote a program which performs all the needed operations and gives me a result (Figure 23), which consist on the middle point and the end point before and after rotation transformation, as well as the resulting circle radius and the k factor:

```
midP = [22,227]
endP = [69,448]

#All operations are performed assuming that start point is located at origin

def getAngle(eP):
    opLeg=eP[0]
    adLeg=eP[1]
    #I will leave it in radians due most of the py functions ask for it in
this way
```

```
    return math.atan(opLeg/adLeg)

def pointRot(point, angle):
    return ( [point[0]*math.cos(angle) - point[1]*math.sin(angle),
        point[0]*math.sin(angle) + point[1]*math.cos(angle) ])


def getDifInfo(mP,eP):
    trueMP=pointRot(mP,getAngle(eP))
    trueEP=pointRot(eP,getAngle(eP))
    #Obtaining radius
    a=trueEP[1]/2
    b=abs(trueMP[0])
    r=(pow(a,2)+pow(b,2))/(2*b)
    rLeg=4.65
    factor=(r+rLeg)/(r-rLeg)
    print("Middle point before rotation: " + str(mP))
    print("End point before rotation: " + str(eP))
    print("Middle point after rotation: " + str(trueMP))
    print("End point after rotation: " + str(trueEP) + "\n")
    print("Circle radius is: " + str(r) + "\n")
    print("Exterior wheel " + str(factor) + " times faster than internal
    wheel" + "\n")

getDifInfo(midP,endP)
```

```
Middle point before rotation: [-7, 295.4]
End point before rotation: [-3.4, 590.4]

Middle point after rotation: [-5.298760375524401, 295.4354128036833]
End point after rotation: [0.0, 590.4097898917327]

Circle radius is: 8225.88687574779
Exterior wheel 1.0011312166454458 times faster than internal wheel
```

Figure 13. Program results with calculated data for the test 1 result.

The First attempt was done with the k value obtained in the result from Test 1 (k=1.00112), the left speed was set to 300 deg/second, which means the right speed was 320 deg/second *1.00112 = 320.3584 deg/second. The result at the first iteration was not as good as expected, the next points were obtained,

$$MidPoint = (22{,}227),$$

$$EndPoint = (69{,}448).$$

The radius was 2011.1774 cm and the left wheel was 1.0046 faster than the right wheel, so the difference between both motors power was even higher than the one obtained in Test 1 with the *Robotics* module, which probably means that the factor needed to compensate depends on the speed of the motors (deg/ second) and is not constant, then I decided to carry out several iterations with different settings changing the speed, the compensation factor and if round or not the final speed result (Figure 26, 28, 30).

The goal is to find an adjustment factor that maximizes the circle resulting radius, therefore minimizing the resulting difference factor *k*.

1.- Speed = 320 deg/s, Round = no

For this iteration, as can be seen in Figure 24 for the settings from Table 11 the optimal adjustment factor can be found near of 1.005.

Speed (deg/s)	Round()	Adjustment factor	Resulting radius (cm)	Resulting difference factor	Turning sense
300	No	1.00112	2011.1774	1.0046	Clockwise
		1.002	3898.815	1.0024	Clockwise
		1.005	23518.79	1.00039	Clockwise
		1.006	913.609	1.01023	Conter clockwise
		1.007	5178.125	1.0018	Conter clockwise
		1.01	1162.927	1.008	Conter clockwise

Table 11. First iteration results

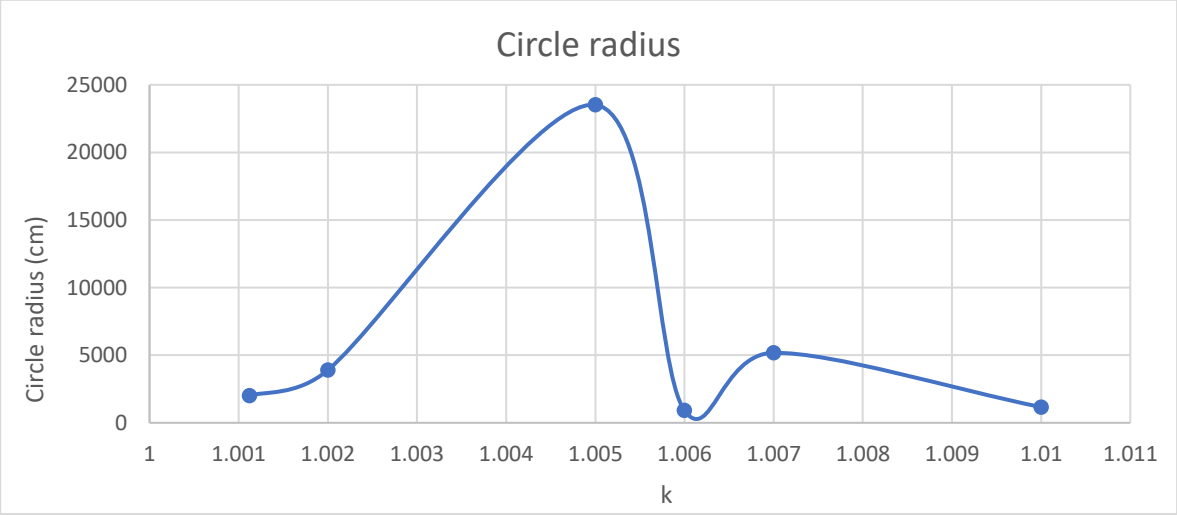


Figure 24. graphic of the first iteration results

2.- Speed = 320 deg/s, Round = yes

In this case just 3 values were tested (Table 12 and Figure 25) because if we round the speed after adjusting it with the factor there would be a lot of non-integers possibilities which wouldn't be tried.

Speed (deg/s)	Round()	Adjustment factor	Resulting radius (cm)	Resulting difference factor	Turning sense
320	Yes	1.005	2053.708	1.00454	Conter clockwise
		1.0054	2055.021	1.004536	Conter clockwise
		1.0057	2455.71	1.0038	Conter clockwise

Table 12. Second iteration results

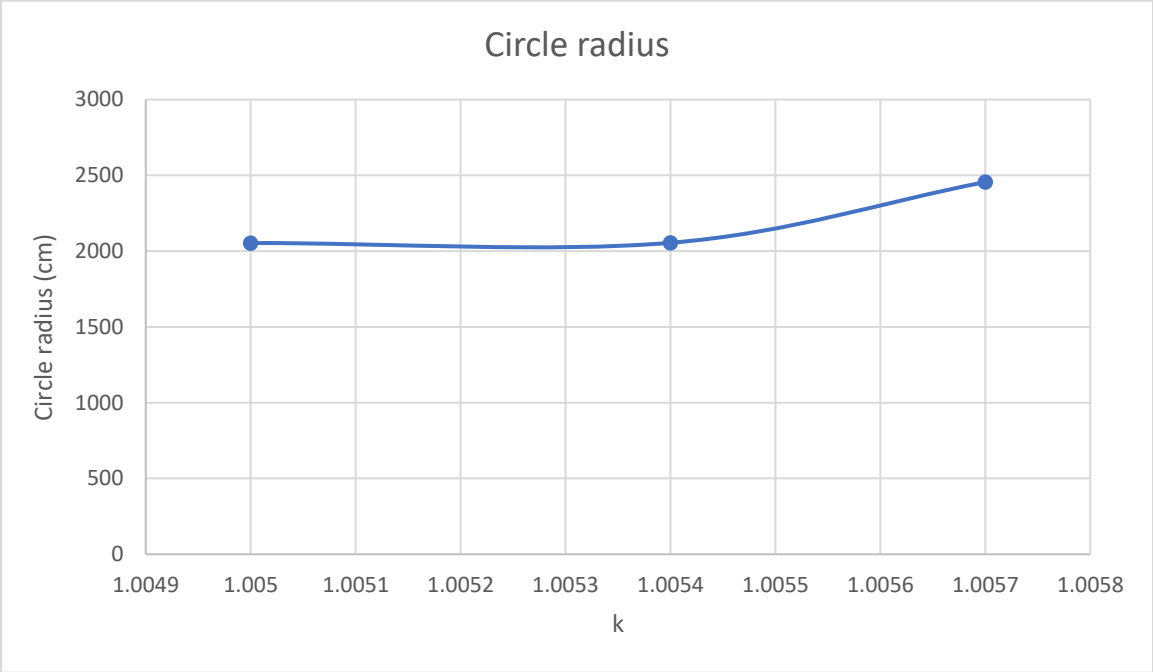


Figure 25. Graphic of the second iteration results

3.- Speed = 320 deg/s, Round = no

With 400 deg/s the maximum radius can be found near of 1.007 (Figure 26). The best value we could obtain is 1.0072 (Table 13), with a resulting radius of 40087.3613 cm and equivalent to 400.8736 m, which is enough bigger four the purpose of this project, the ant will move inside a small area so the error will not be enormous.

Speed (deg/s)	Round()	Adjustment factor	Resulting radius (cm)	Resulting difference factor	Turning sense
400	No	1.006	6149.46	1.0015	Conter clockwise
		1.0065	3374.2275	1.00276	Conter clockwise
		1.007	2636.3834	1.003534	Conter clockwise
		1.0072	40087.3613	1.000232	Clockwise
		1.00725	7883.9416	1.00118	Conter clockwise
		1.0075	1787.46	1.0052	Conter clockwise

Table 13. Third iteration results

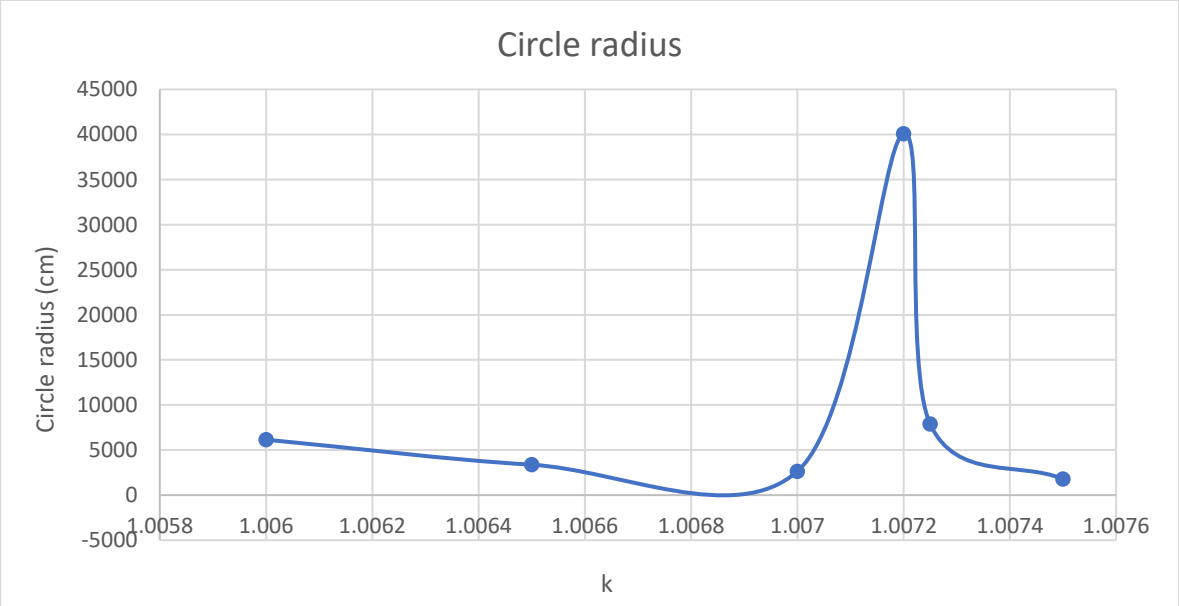


Figure 14. Graphic of the third iteration results

Whiteboard marker motor (Medium LEGO servo motor):

Figure 15. Medium large servo motor (LEGO®)

This servo motor (Figure 27) is used to rotate the marker joint (Figure 28), no matter if it actually has the marker on, it will always perform the subtask, due the limits on this project scope I did not performed tests on this actuator, I just programed it to turn 90° for descending the joint and -90° for raising it.

Figure 16. Marker joint motor implemented

3 Robot Architecture

3.1 Architecture

In the Figure 29 we can see the robot architecture, it contains all the modules that make possible the robot can simulate the task of searching for food. Below is a brief description of each one of them.

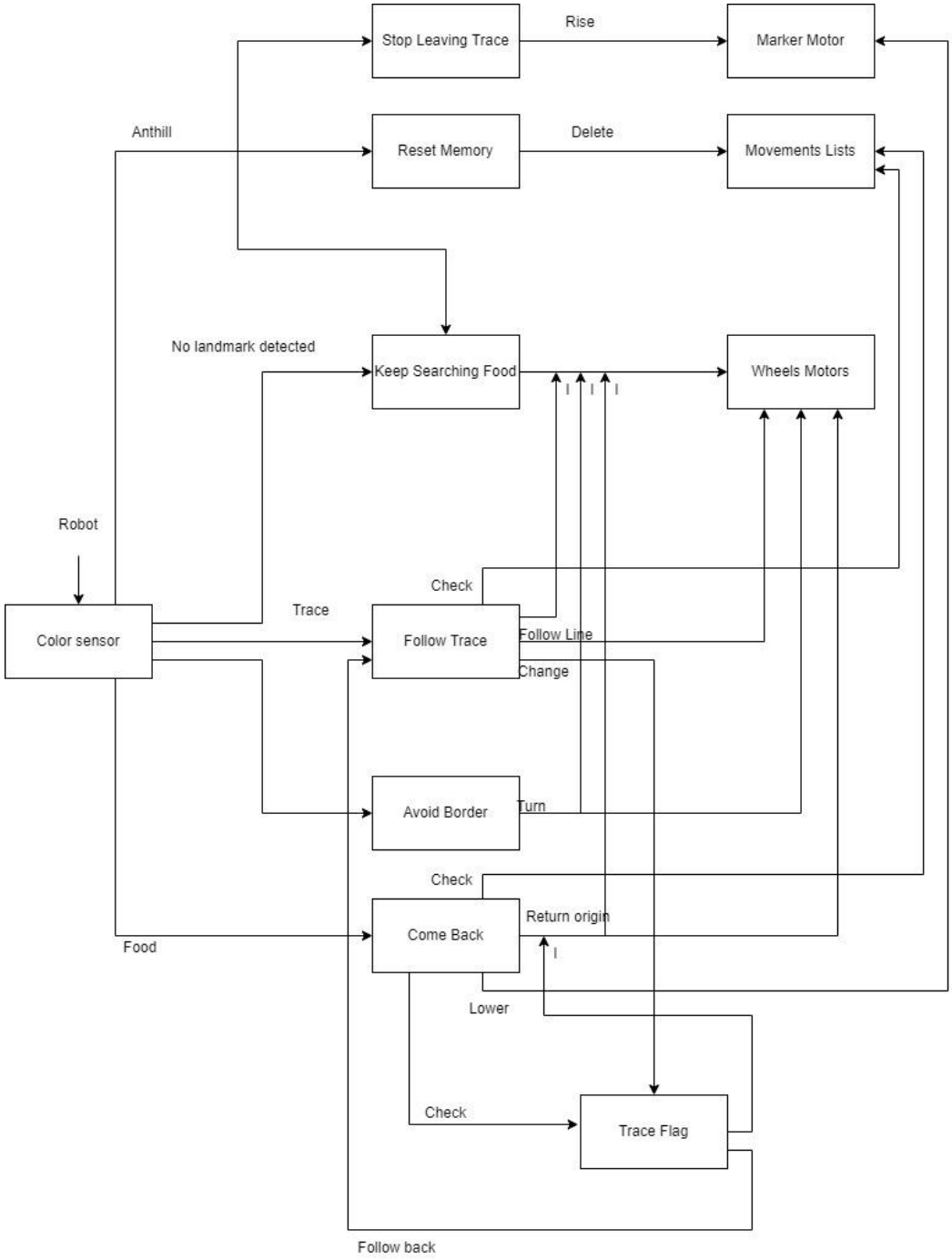


Figure 17. Architecture diagram

- Color sensor: The color sensors implemented on robot, it retrieves data from the environment and if that data corresponds to any landmark, then the corresponding behavior is performed.
- Keep Searching Food: If Color sensor detects the robot is located on the anthill or detects no landmark then it keeps wheels moving while the robot stills searching for food.
- Follow Trace: If the color sensor detects the trace color, then it checks the movements lists to know which direction of the line to choose and moves the wheels motors to follow it
- Avoid Border: If the color sensor detects the border color, then makes the wheels to move to turn and avoid it.
- Come Back: If the color sensor detects the color used to represent food, then it comes back, it checks if food was reached by following a trace or by searching it, in the second case checks the movements lists to compute the origin position, it will always lower the marker joint.
- Trace Flag: If the robot has reached food by following a trace, then it follows it back
- Reset Memory: If the color sensor detects the anthill color, then the robot forgets all the previous stored movements.
- Movements Lists: A lists with stored information about distances and angles from all the previous robot movements stored after the last reset.
- Stop Leaving Trace: If the color sensor detects the anthill color, then the robot rises the marker joint.
- Marker Motor: Servo motor that moves marker joint.
- Wheels Motors: Servo motors that move wheels of the robot.

3.2 Behaviors

Keep Searching:

The robot chooses a random degree amount between -90° and 90° and a distance between 20 cm and 40 cm, then it first turns the angle and finally moves the distance chosen towards the new direction.

Follow Trace

If the robot finds a trace, then it computes its current location and direction (angle) according to all the previous movements, then using that information it determines the trace angle, if the difference between the robot direction and the trace's angle is less than 90° it follows the line in the current direction, otherwise the robot turns to the other side of the line (Figure 30). After choosing a side to follow then while the sensors don't detect the goal (anthill or food) it moves straight, if the trace is detected again by one of sensors robot turns 10° to that sensor side and so on until the goal is reached.

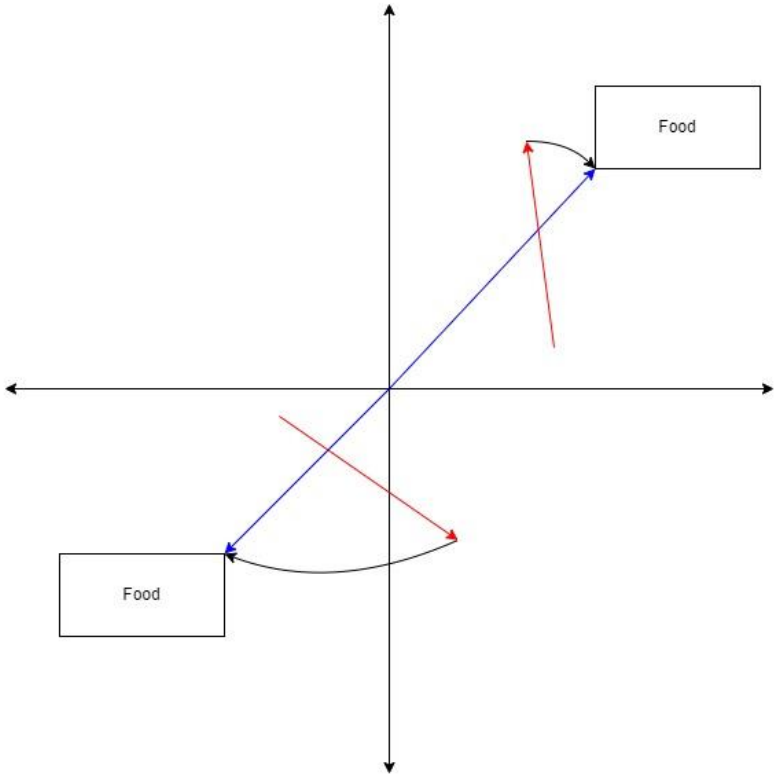


Figure 18. Following line illustration

Avoid Border

If the robot detects a border, then it chooses an angle between 90° and 180° then it turns to the opposite side in which border was detected, next it chooses a distance between 10 cm and 30 cm to move straight. Basically, another *keep searching* movement, but making us sure the border is avoided.

Come Back

If the robot detects food, then it checks how it was reached, if it was by following a trace it just turns until it detects the trace again and follows it back. If it was by searching for food, then the robot computes its current position and direction and use it to calculate the distance and the angle needed to turn and after that move straight to reach the anthill.

4 Experiments and results

Color detection

As it was mentioned before we carried out data analysis on color sensors' retrieved information about the colors needed to detect landmarks, the first attempt to detect

colors was to use the dispersion of each RGB channel to classify every sensor input, but it failed, during the testing of the robot's complete task the robot failed to detect the colors, so the data retrieved by sensors checked again. The dispersion margin just moved to higher values which means the sensor gets highly affected by Ambiental light, therefore the central values and tolerances we got are now useless.

```
bBound=(6,7,4)
bBoundTol=(1,1,1)

red=(57,10,8)
redTol=(3,1,1)

green=(11,37,18)
greenTol=(1,2,1)

blue=(5,11,27)
blueTol=(1,1,2)
```

The code above are the structures containing the values obtained during the color sensors' test, the code below is the classification for all inputs. It checks for each input if the value of each channel is between the central value plus the tolerance and the central value minus the tolerance for the current if´s color. For example, the firs if checks the input from the right sensor comparing with the defined values for black.

```
rSens = rCSensor.rgb()
lSens = lCSensor.rgb()
senSide = label = None
#If the robot senses black (environment bounds)
#First two ifs check if its black color

if(rSens[0]>bBound[0]-bBoundTol[0] and rSens[0]<bBound[0]+bBoundTol[0]
and rSens[1]>bBound[1]-bBoundTol[1] and rSens[1]<bBound[1]+bBoundTol[1]
and rSens[2]>bBound[2]-bBoundTol[2] and rSens[2]<bBound[2]+bBoundTol[2]):
    senSide=1
    label=bBoundLabel
else:
    if(lSens[0]>bBound[0]-bBoundTol[0] and lSens[0]<bBound[0]+bBoundTol[0]
    and lSens[1]>bBound[1]-bBoundTol[1] and lSens[1]<bBound[1]+bBoundTol[1]
    and lSens[2]>bBound[2]-bBoundTol[2] and lSens[2]<bBound[2]+bBoundTol[2]
    ):
        senSide=0
        label=bBoundLabel
    else:#Check if its red color
        if(rSens[0]>red[0]-redTol[0] and rSens[0]<red[0]+redTol[0]
        and rSens[1]>red[1]-redTol[1] and rSens[1]<red[1]+redTol[1]
        and rSens[2]>red[2]-redTol[2] and rSens[2]<red[2]+redTol[2]):
            senSide=1
```

```
            label=rLabel
        else:
            if(lSens[0]>red[0]-redTol[0] and lSens[0]<red[0]+redTol[0]
            and lSens[1]>red[1]-redTol[1] and lSens[1]<red[1]+redTol[1]
            and lSens[2]>red[2]-redTol[2] and lSens[2]<red[2]+redTol[2]):
                senSide=0
                label=rLabel
            else:#Check if its green color
                if(rSens[0]>green[0]-greenTol[0] and rSens[0]<green[0]+greenTol[0]
                and rSens[1]>green[1]-greenTol[1] and rSens[1]<green[1]+greenTol[1]
                and rSens[2]>green[2]-greenTol[2] and rSens[2]<green[2]+greenTol[2]):
                    senSide=1
                    label=gLabel
                else:
                    if(lSens[0]>green[0]-greenTol[0] and lSens[0]<green[0]+greenTol[0]
                    and lSens[1]>green[1]-greenTol[1] and lSens[1]<green[1]+greenTol[1]
                    and lSens[2]>green[2]-greenTol[2] and lSens[2]<green[2]+greenTol[2]):
                        senSide=0
                        label=gLabel
                    else:#Check if its blue color
                        if(rSens[0]>blue[0]-blueTol[0] and rSens[0]<blue[0]+blueTol[0]
                        and rSens[1]>blue[1]-blueTol[1] and rSens[1]<blue[1]+blueTol[1]
                        and rSens[2]>blue[2]-blueTol[2] and rSens[2]<blue[2]+blueTol[2]):
                            senSide=1
                            label=bLabel
                        else:
                            if(lSens[0]>blue[0]-blueTol[0] and lSens[0]<blue[0]+blueTol[0]
                            and lSens[1]>blue[1]-blueTol[1] and lSens[1]<blue[1]+blueTol[1]
                            and lSens[2]>blue[2]-blueTol[2] and lSens[2]<blue[2]+blueTol[2]):
                                senSide=0
                                label=bLabel
                            else:
                                label="none"
                                sense=-1
```

Due the lack of time I just implemented some general rules that all colors I used in this project can fulfill, for example if the input color is black we can expect it to have small values for all the RGB channels, if the input color is red we can expect it to have higher values on the red channel, if the input color is green we can expect it to have higher values on the green channel, finally, if the input color is blue we can expect it to have higher values on the blue channel,.

```
if(rSens[0]<20 and rSens[1]<20 and rSens[2]<20):
        senSide=right
        label=bBorderLabel
    else:
        if(lSens[0]<20 and lSens[1]<20 and lSens[2]<20):
            senSide=left
            label=bBorderLabel
        else: #Check if its red color
            if(rSens[0]>rSens[1] and rSens[0]>rSens[2] and rSens[0]>50 and rSens[1]<20 and
rSens[2]<20):
                senSide=right
                label=foodLabel
            else:
                if(lSens[0]>lSens[1] and lSens[0]>lSens[2] and lSens[0]>50 and lSens[1]<20 and
lSens[2]<20):
                    senSide=left
                    label=foodLabel
                else:#Check if its blue color
                    if(rSens[2]>rSens[0] and rSens[2]>rSens[1] and rSens[2]>25 and rSens[0]<20 and
rSens[1]<20):
                        senSide=right
                        label=anthillLabel
                    else:
                        if(lSens[2]>lSens[0] and lSens[2]>lSens[1] and lSens[2]>25 and lSens[0]<20
and lSens[1]<20):
                            senSide=left
                            label=anthillLabel

                        else:#Check if its green color
                            if(rSens[1]>rSens[0] and rSens[1]>rSens[2] and rSens[1]>35 and
rSens[0]<20 and rSens[2]<31):
                                senSide=right
                                label=traceLabel
                            else:
                                if(lSens[1]>lSens[0] and lSens[1]>lSens[2] and lSens[1]>35 and
lSens[0]<20 and lSens[2]<31):
                                    senSide=left
                                    label=traceLabel

                                else:
                                    label="noone"
```

Movement accuracy (returning to anthill and following trace)

When searching for food as it has been mentioned the robot saves all movements information (angles and distances), but due to turnings and straight movements are not 100% accurate then the position and the angle towards the center that the robot can compute is not accurate. It sometimes makes the robot go to a wrong direction or to move a wrong distance until anthill, it also affects when the robot wants to follow a trace because if the computed position of the center is wrong then the angle calculated for the traces are wrong too.

5 Conclusion

As can be seen along all the document, it is possible to build a behavior-based approach robot with LEGO Mindstorms EV3 kit, although the robot has some failures due the lack of accuracy of the LEGO sensors and actuators it works and perform its task properly.

This project can be improved in several ways, may be would not worth it to attempt the robot learn the complete task by reinforcement because it has a big space for variables (the amount of degrees and the amount of distances along all the movements needed to reach food) and a really big and continuous searching space, in other world it is located into the real world instead of world specifically created for the robot (like a robot from a simulator program), although the environment has landmarks it actually stills being located into a non-discrete environment and would be very complicated to do it.

Maybe it could implement probabilistic localization and movement as well which would make the robot to take in account all the movements and turnings probability. This means that if the robot has a landmark as an objective of the current movement (like when the robot is going to the anthill after finding food) and it fails (if the landmark is not detected at the end of the movement) then the robot could search the target inside a complete area which would belong to the deviation generated by all the previous movement instead of just continue searching, at the same time every time that the robot detects a landmark it would reinforce the belief that it has about its location, it would reduce the deviation area for the belief keeping it small.

Another aspect that can be improved for this project is to implement a better method to classify the color sensor inputs, in this subtask it is a good idea to try with reinforcement learning, every time the robot classifies the input in the correct color it would receive a reward. We can also try with a similar idea to the followed in this document, that is using the differences of the values of the RGB channels instead of just the central values, for example on Table 9 we have the RGB central values (6, 7, 4), may be because of the light the values could move to (10, 11, 8), which would make impossible to classify the color, but we know that the differences among them

still being the same, for example the difference for the red channel compared with the blue channel 6 – 4 = 2 for the first values and 10 – 8 = 2 for the last values.

References

Ant. (2021, November 28). In *Wikipedia*. https://en.wikipedia.org/wiki/Ant

Ant colony optimization algorithms. (2021, December 28). In *Wikipedia.* *https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms*

LEGO® MINDSTORMS® EV3 Hardware Developer Kit. (2013). LEGO Group. https://www.mikrocontroller.net/attachment/338591/hardware_developer_kit.pdf

LEGO®. LEGO Group. https://www.lego.com/en-us

MicroPython. George Robotics Limited. https://micropython.org/

D. E. Hernández Sánchez, J. R. Eguibar Cuenca, C. Cortés Sánchez and J. F. Reyes Cortés. Diseño, construcción y modelo dinámico de un robot móvil de tracción diferencial aplicado al seguimiento de trayectorias. CONGRESO INTERNACIONAL ANUAL DE LA SOMIM. (2017, September 20). http://somim.org.mx/memorias/memorias2017/articulos/A3_189.pdf

LEGO® MINDSTORMS Education EV3 MicroPython. (2019). LEGO Group. https://pybricks.com/ev3-micropython/

Angles of Intersecting Chords Theorem. https://www.varsitytutors.com/hotmath/hotmath_help/topics/angle-of-intersecting-chords-theorem

Annex A

In this section we have the source code for the robot, it is written according the robot architecture so you can go back and look if you need.

```
#Initialize sensors
rCSensor = ColorSensor(Port.S4)
lCSensor = ColorSensor(Port.S1)
gyroSensor = GyroSensor(Port.S2, Direction.CLOCKWISE)


#Initialize motors
rightMo = Motor(Port.C)
leftMo = Motor(Port.B)
markerMo = Motor(Port.D)

listDist = []
listTurn = []

bBorderLabel="black"
foodLabel ="red"
traceLabel="green"
anthillLabel="blue"

#Labels for side of the sensor
right=1
left=-1


#Constants for wheels in terms of degrees
speed=400 #Speed for each wheel in terms of degres/second
turnSpeed=100 #Turning speed
factor=1.0072
#Turning speed for each wheel
velLTurn=turnSpeed
velRTurn=velLTurn*factor
#Moving speed for each wheel
velL=speed
velR=factor*velL
#Constants for wheels in terms of distance
####Actuali this is the
radBet = 47 #The distance between both wheels (axis lenght)
radWh = 28 #The radius of wheels
#speed for each wheel in terms of Distance/second
wheelSpeed=(speed*radWh*math.pi/180)/1000 #units are given in mm/s
```

```
#Timer for movement
movingTimer = StopWatch()
#Variables used to control movement
currMovementTime = 0
currTurningAngle = 0
#logical flags for processes
movingFlag = False
turningFlag = False
arrivedByLine = False
markerDown = False



def turn(angle):
    global turningFlag
    #If the angle is 0 it is not necesary to turn
    if(angle!=0):
        #Side it will turn
        turnSen = angle/abs(angle)
        #Setting the velocities
        velLT = turnSen*velLTurn
        velRT = -turnSen*velRTurn
        #Turning
        leftMo.run(velLT)
        rightMo.run(velRT)
        a=0
        gyroSensor.reset_angle(0)
        #Loop used to control the turning
        while abs(gyroSensor.angle())<abs(angle):
            a=a+1
        stop()
    #Storing the turning angle
    listTurn.append(angle)

def move(dist):
    global currMovementTime
    global movingFlag
    #Calculating the time
    time = ((dist*10/wheelSpeed))
    #Global var used to control the movement time
    currMovementTime = time
    movingFlag=True
    movingTimer.reset()
    #Moving
```

```
    leftMo.run_time(velL, time, then=Stop.BRAKE, wait=False)
    rightMo.run_time(velR, time, then=Stop.BRAKE, wait=False)
    #Storing the distance
    listDist.append(dist)


#Function for stopping the motors
def stop():
    leftMo.hold()
    rightMo.hold()

#Function that senses the color, it returns a flag for the color detected
#And the side, if it has not detected any landmark color returns none
def sense():
    rSens = rCSensor.rgb()
    lSens = lCSensor.rgb()
    senSide = label = None

    if(rSens[0]<20 and rSens[1]<20 and rSens[2]<20):
        senSide=right
        label=bBorderLabel
    else:
        if(lSens[0]<20 and lSens[1]<20 and lSens[2]<20):
            senSide=left
            label=bBorderLabel
        else: #Check if its red color
            if(rSens[0]>rSens[1] and rSens[0]>rSens[2] and rSens[0]>50 and
rSens[1]<20 and rSens[2]<20):
                senSide=right
                label=foodLabel
            else:
                if(lSens[0]>lSens[1] and lSens[0]>lSens[2] and lSens[0]>50
and lSens[1]<20 and lSens[2]<20):
                    senSide=left
                    label=foodLabel
                else:#Check if its blue color
                    if(rSens[2]>rSens[0] and rSens[2]>rSens[1] and
rSens[2]>25 and rSens[0]<20 and rSens[1]<20):
                        senSide=right
                        label=anthillLabel
                    else:
                        if(lSens[2]>lSens[0] and lSens[2]>lSens[1] and
lSens[2]>25 and lSens[0]<20 and lSens[1]<20):
                            senSide=left
                            label=anthillLabel
```

```
                        else:#Check if its green color
                            if(rSens[1]>rSens[0] and rSens[1]>rSens[2] and
rSens[1]>35 and rSens[0]<20 and rSens[2]<31):
                                senSide=right
                                label=traceLabel
                            else:
                                if(lSens[1]>lSens[0] and lSens[1]>lSens[2]
and lSens[1]>35 and lSens[0]<20 and lSens[2]<31):
                                    senSide=left
                                    label=traceLabel
                                else:
                                    label="noone"
    return label,senSide



def avoidBorder(side):
    #First of all wwe have to save the current distance (if the rovot is
moving straight)
    interruptMoving()
    #if it was detected by right sensor then side is 1 and the turn must by
counter clockwise
    deg = 0
    while deg==0:
        deg = urandom.randint(90,180)*(-1*side)
    turn(deg)
    #Distance will be chosen in cm
    dist=urandom.randint(10,30)
    move(dist)




#Returns the resulting angle and distances since the last reset of memory
def sum ():
    #We have to invert the sign of each angle in order to know the correct
values
    #We need to obtain the sum of vector with all distances acording to
their angles
    currAngle = 0
    sumXDis = 0
    sumYDis = 0
    for i in range(len(listDist)):
        #Sum of the angles
```

```
        currAngle = currAngle + (-listTurn[i])
        currDist = listDist[i]
        #Obtainign cosdinates for each vector
        currXDis = currDist * math.cos(math.radians(currAngle))
        currYDis = currDist * math.sin(math.radians(currAngle))
        #Sum of the distances
        sumXDis = sumXDis + currXDis
        sumYDis = sumYDis + currYDis
    #We don´t need to turn more than 360°, this line aplies module
    currAngle=(currAngle/abs(currAngle))*(abs(currAngle)%360)
    #Get a positive equivalent
    if(currAngle<0):
        currAngle=360+currAngle
    return sumXDis, sumYDis,  currAngle

#Obtains the distance and direction needed to go back
def getBackParam():
    sumX, sumY, angle=sum()
    #Obtaining the direction from the origin to the robot
    finalAngle = (math.atan(sumY/sumX)*180)/math.pi
    if(sumX<0):
        finalAngle+=180
    #Obtaining the distance from the origin to the robot
    backDistance = math.sqrt(sumX*sumX + sumY*sumY)
    #Obtaining the angle due the current direction and position of the robot
    angle=(angle/angle)*(abs(angle)%360)
    backAngle = - (round(finalAngle+180 - angle))%360
    return backDistance, backAngle

#if the robot was moving then the last distance has not been completed
#And it is replaced
def replaceLastDistance(time):
    #Obtaining the distance traveled with the current time
    newDist = (time*wheelSpeed)/10
    listDist[-1] = newDist

#If a movement has not been completed and the robot needs to stop then it is
interrupted
def interruptMoving():
    global movingFlag
    if(movingFlag):
        stop()
        replaceLastDistance(movingTimer.time())
        movingFlag=False
```

```
def downMarker():
    global markerDown
    markerMo.run_target(200, 90, then=Stop.BRAKE, wait=True)
    markerDown = True

def upMarker():
    global markerDown
    markerMo.run_target(200, -90, then=Stop.BRAKE, wait=True)
    markerDown=False


def followLine(side):
    global arrivedByLine
    interruptMoving()

    varWhile=True
    #First of all we need to calculate the angle of the line found,
    #We know tath the line comes from origin so, we just need the point we
found
    #to calculate the tangent
    sumX, sumY, currRobAngle=sum()
    #currRobAngle=225
    lineAngle = (math.atan(sumY/sumX)*180)/math.pi

    if(sumX<0):
        lineAngle+=180
    #lineAngle=360

    #If angle between robot angle and line angle is less than 90
    #It is aligned in the correct direction of the line, otherwhise
    #We must turn to the other side of the line
    if(not arrivedByLine and abs(lineAngle-currRobAngle)>90):
        turn(90*side)
        side=-side

    #runFlag tells us if robot motors are runing
    runFlag=False
    color=None
    currSide=side

    senSide=0
    #It will follow the line until if finds food or until it finds anthill
    if(arrivedByLine):
        goal = anthillLabel
    else:
```

```
        goal = foodLabel
    while(color!=goal):
        color,senSide = sense()
        if(color!=traceLabel and not runFlag):
            leftMo.run(velL)
            rightMo.run(velR)
            runFlag = True
        if(color==traceLabel):
            stop()
            turn(10*senSide)
            runFlag=False
        #If the robot choosed the wrong side we must correct it
        if(goal==anthillLabel and color==foodLabel):
            goal=foodLabel
        if(color==anthillLabel and goal==foodLabel):
            goal=anthillLabel
    #this Flag is true only when the robot finds food
    #is used to know if the robot must follow back
    if(goal==anthillLabel):
        arrivedByLine=False
    else:
        arrivedByLine=True


def comeBack():
    interruptMoving()
    #If the robot arrived by line we just have to follow it again
    if(arrivedByLine):
        color=None
        while(color!=traceLabel):
            color,side=sense()
            turn(1)
        downMarker()
        followLine(side)
    else:
    #If the robot arrived by searching then computes the back parameters
    #Next it turns and moves the specified distance
        dis, angle = getBackParam()
        turn(angle)
        downMarker()
        move(dis)

#Deletes the current memory
def resetList():
    global listDist
```

```
    global listTurn
    listDist = []
    listTurn = []

#Function for moving control
def keepSearching():
    global movingFlag
    global currMovementTime
    #If its moving straight we must look out
    if(movingFlag):
        if(movingTimer.time()>currMovementTime):
            currMovementTime=0
            movingFlag=False
    else:
    #If not we must continue searching
        deg=urandom.randint(-90,90)
        turn(deg)
        #Distance will be chosen in cm
        dist=urandom.randint(20,40)
        move(dist)

while True:
    color,side = sense()
    #If the robot finds a trace
    if(color==traceLabel):
        followLine(side)
        color,side = sense()
    #It will find food when it detects green
    if(color==foodLabel):
        comeBack()
    #If the robot finds a border
    if(color==bBorderLabel):
        avoidBorder(side)
    else:
        #If it has arrived to the anthill
        if(color==anthillLabel):
            if(markerDown):
                upMarker()
        #If it has not detected anything
        keepSearching()
```