BRENO CUNHA QUEIROZ

Soccer Robot

SOCCER ROBOT

Introduction:

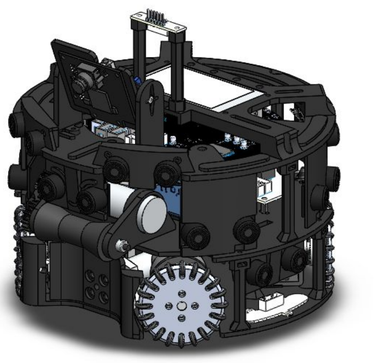
This project is based on the rules of RoboCupJunior Soccer Open. I began conducting my first research on soccer robots in the late 2016 at the CIC Robotics laboratory. In 2016, I was in a group with four students[1] to do two robots to compete in the RoboCup, but in 2017 the other students left the project and I started to conduct the research alone.

The development of the current version of the soccer robot began in December 2017 and extends to the moment. I had just a basic understanding of C, 3D modeling, circuit, and vector design when I started developing the current version.

By doing this project I discovered how exciting it is to be a researcher. I had fun at every step, especially when solving each problem using math and programming.

Structure:

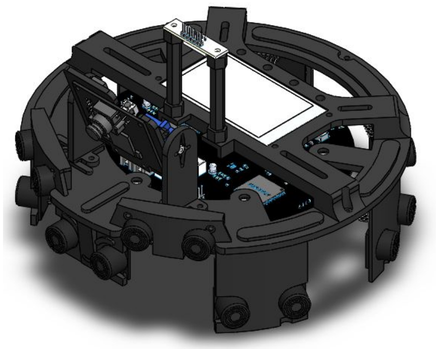
The structure of the physical robot is composed of 3 layers of polycarbonate printed in 3D. Heavier components, such as motors and battery, are designed for the lower portion of the robot, distributed over the surface to ensure stability. The twelve ultrasonic (URM37 V4.0) are distributed among the three layers.



Robot modeled in Solidworks.

Top Layer:

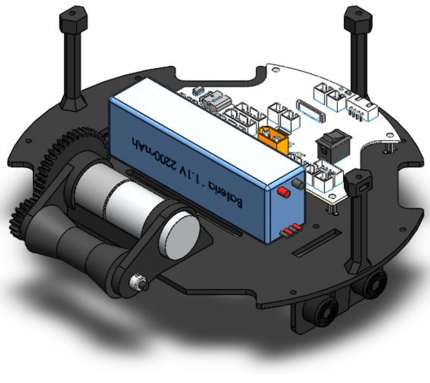
In this layer, the 9DoF sensor (CMPS11) is positioned due to the sensitivity to the electromagnetic distortions caused by the motors. In addition, in this layer it is also positioned the Pixy camera on a tiltable base with 9g servo, a Nextion display of 3.2" and the main circuit.



Top layer.

Middle layer:

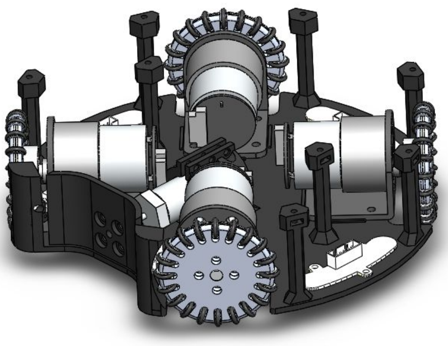
A dribbler system, a Lipo battery of 11 1 V and the secondary circuit are positioned.



Middle layer.

Bottom layer:

In this layer, there are three field line detection modules, two ball possession detection circuits, one 12V solenoid for kicking the ball and four 12V 320 RPM motors for movement.



Bottom layer.

[1]Students: Yuri Reis, Heitor Rivera, Pedro Quadros, and Leonardo Passos

Devices:

Dribbler:

The dribbler consists of a 1500 RPM 12V motor connected by gears on a silicone roller. When the robot detects that it is with the ball, the motor is activated, which makes difficult the loss of the ball during movement.

Camera support:

The omnidirectional support of the camera has the function of locating the ball in the field. I used a Pixy camera because of its precision and its pre-processing of data. The mount has extended the degree of freedom of the camera vertically, allowing the robot to always hold the image on the ball positioned in the center of the camera.

Omnidirectional wheels:

The omnidirectional wheels used allow a more dynamic movement for the robot. I used four 50mm wheels of the GTF robots.

Kick device:

The ball is kicked by a 12V solenoid. A TIP120 was used to allow the solenoid to be driven by a 5V signal from a digital port.

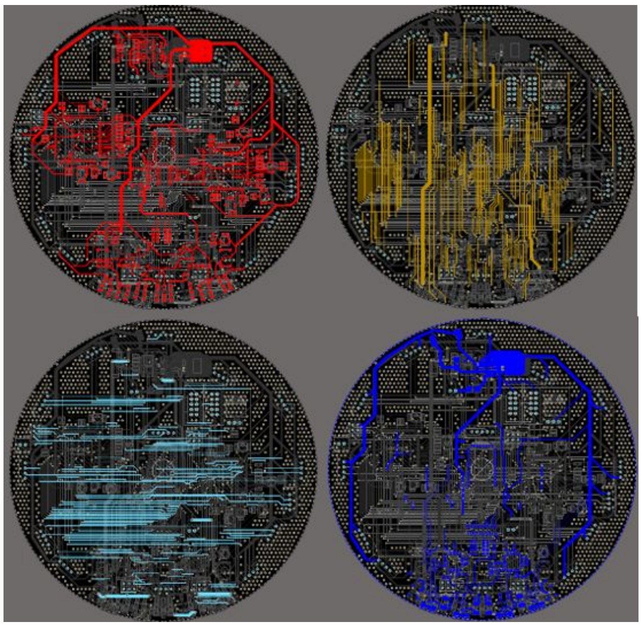
Circuits:

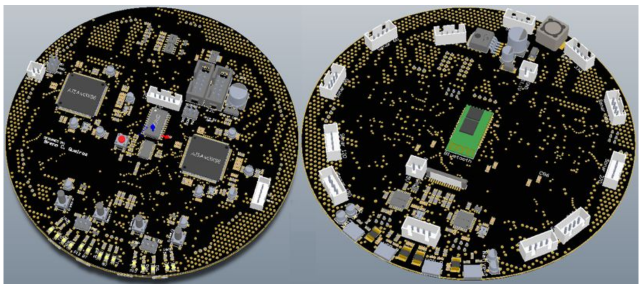
All circuits were made through Altium Designer. The library with all components used in the circuit (from resistors to microcontrollers) were created entirely by me from the datasheet of each component.

It is important to note that, although the circuits were bought in June, they only arrived on November 30 due to problems in the Customs of Brazil, which delayed much the construction and testing of the circuits.

Main Circuit:

The main circuit has four layers of signal with two microcontrollers Atmel SAM3X8E for data processing and control, chosen for being the most powerful used in the family of embedded circuits developed by Arduino. Since Arduino line microcontrollers are used, the programming of this board is compatible with Arduino IDE. The microcontrollers are individually connected to the "microchips" atmega16u2 for USB communication with the computer. Each microcontroller is programmed individually. I want to use one microcontroller only to process data and the other to make decisions during the game. In addition, the integrated circuits CD74HC4067 and PCA9685 were used for the expansion of the analogue and digital (PWM) ports available for connection of the sensors. For powering most electronic devices, the LM2576-5.0 was used, which converts 11.1V to 5V. However, the microcontrollers only work at 3.3V, so the NCP1117ST33T3G was used to convert 5V to 3.3V. At the bottom of the circuit, there is a bluetooth HC-05 to allow communication with other devices.

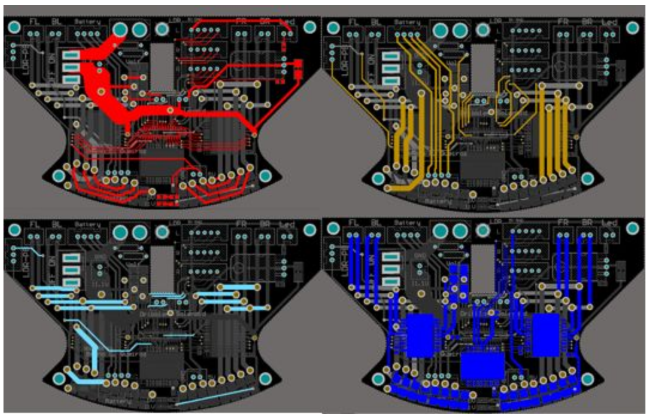


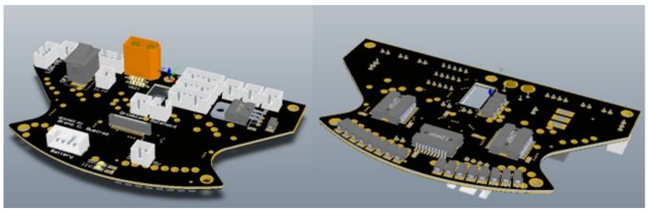


Main circuit generated in 3D.

Secondary circuit:

The secondary circuit, in turn, performs no information processing. Instead, this board acts at the interface for controlling the motors, from three L298P drivers, and in transmitting the signal from the light sensors to the microcontrollers in the main circuit. In addition, the 11.1V 2200mA battery is connected directly to this circuit, which distributes some of the energy between the motors and the other part for conversion into other voltages in the main circuit.

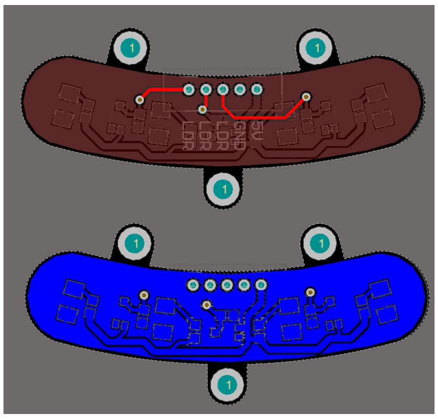


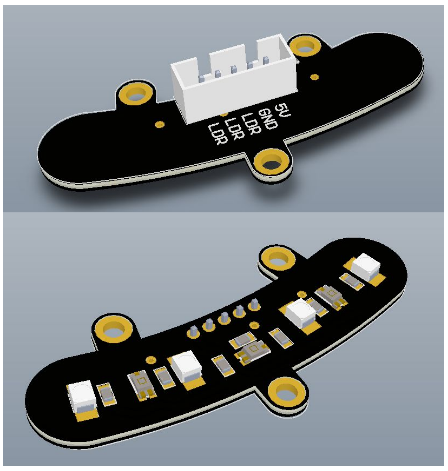


Secondary circuit generated in 3D.

Light sensor module:

There are two types of light reading circuit in this robot. The first has only one TEMT9000 photoresistor and was designed to detect ball possession. The second has three TEMT9000 photoresistors and four purple LEDs for detecting field lines (images of the second type below).

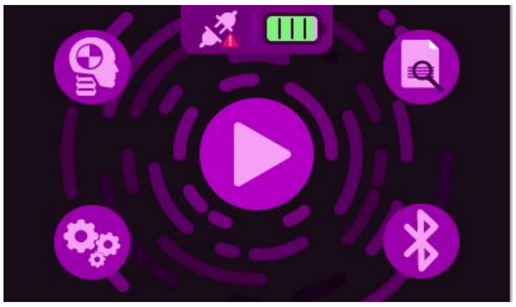




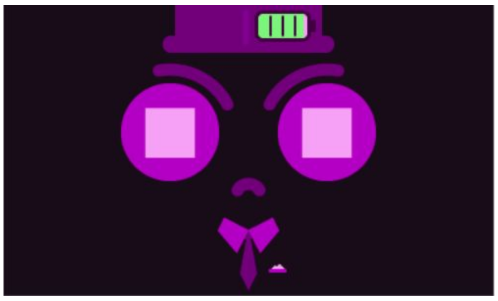
Display:

The human-robot interface is performed through a 3.2" Nextion Display. The screens were developed through the manufacturer's program, while the images used on the screens were created by me through the Adobe Illustrator program.

From the main screen it is possible to select icons to navigate to another five screens. The central button starts and stops the game program (switching between "play" and "pause.") The images below are prints taken from the Nextion program for better viewing.

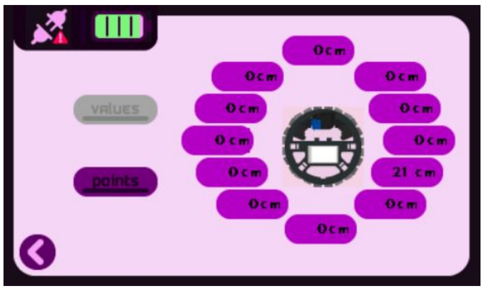


Main screen.

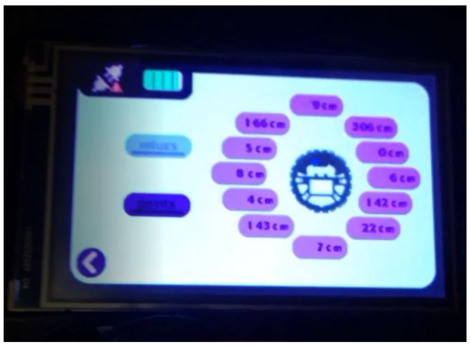


Game screen (two buttons to interrupt the game code).

By clicking the upper-right icon on the touch screen, the user is redirected to the sensor monitoring screens (camera, ultrasonic, light, 9DoF sensor). These screens helped me figure out if any sensors are badly connected or defective. In addition, it made it possible to analyze the readings of the sensors quickly even away from a computer.

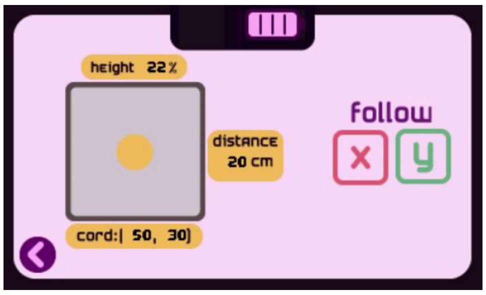


Screen to analyze the values of the 12 ultrasonic sensors.

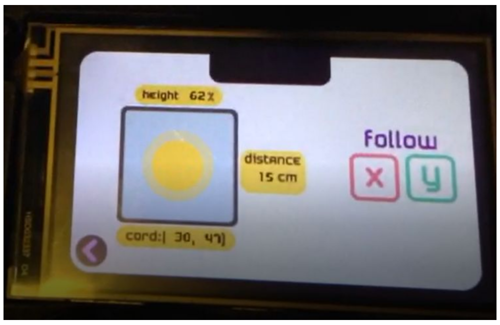


Test of the reading screen of the ultrasonic sensors.

The two images below refer to the screen for monitoring camera readings and ball detection. In this screen, the height of the identified ball object and its Cartesian coordinate in the image captured by the camera are made available. In addition, the approximate distance from the camera to the ball is shown. The "X" and "Y" buttons are used to test the movement of the robot to follow the ball. If you press the "X" button, the robot moves to the right and left to try to keep the ball in the horizontal center of the camera. If you press the "Y" button, the servo that controls the degree of tilt of the camera rotates the camera to try to keep the ball in the vertical center of the camera.

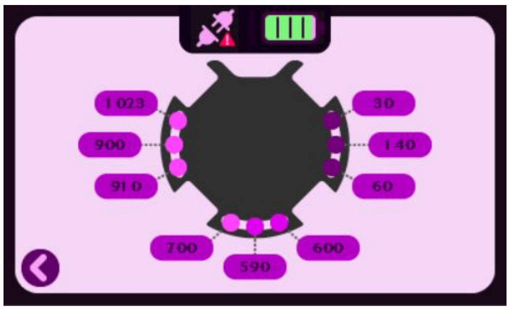


Screen for monitoring the values returned by the camera.

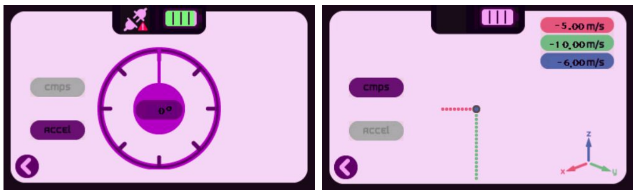


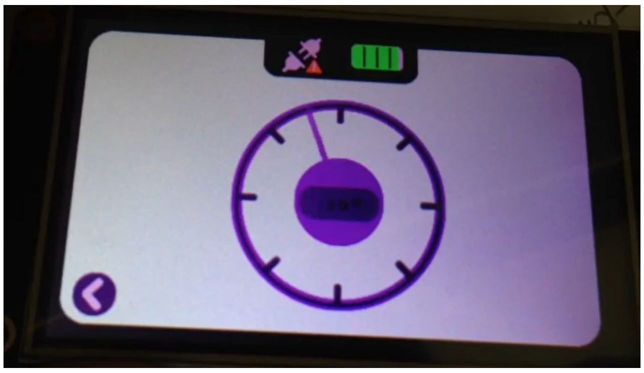
Test for recognition of the ball.

The image below refers to the screen to analyze the reading of the light sensors located in the lower layer. (No tests were performed).

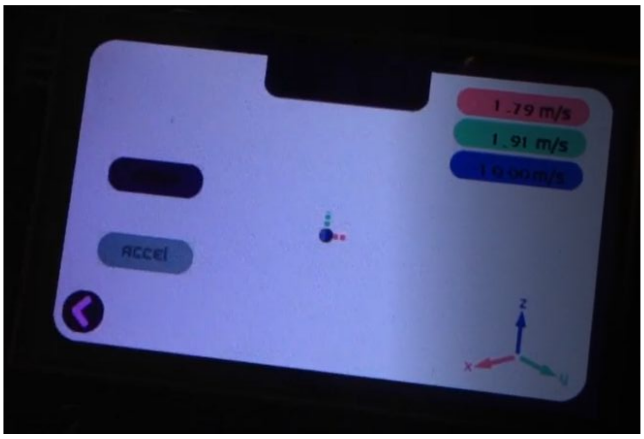


Screens to analyze the readings from the 9DoF sensor below. The left screen was developed to see the degree of inclination of the robot and the right screen was created to analyze the acceleration on the robot.





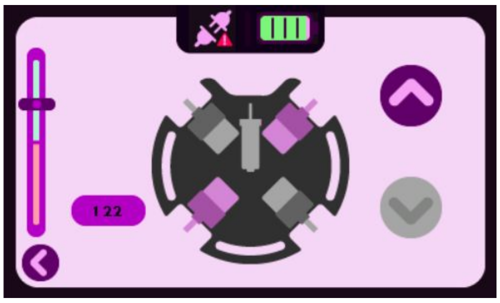
Tests on the compass monitoring screen.



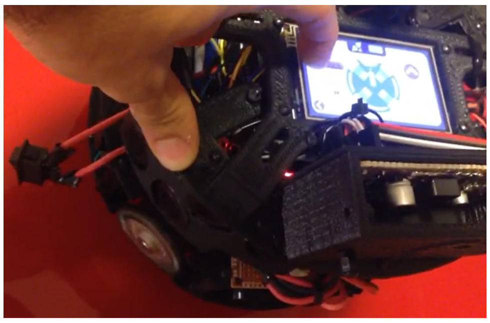
Tests on the acceleration monitoring screen.

In addition to the screens to analyze the reading of the sensors, screens were created to test each of the actuators.

In the screen responsible for testing the lower layer actuators, it is possible to select which motors to test and choose the test power (motors selected in this screen are in pink to indicate that they are selected).

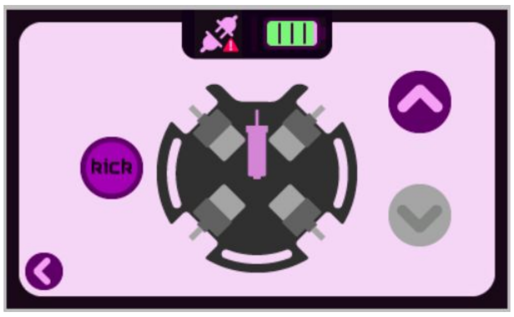


Screen for control of motors power.

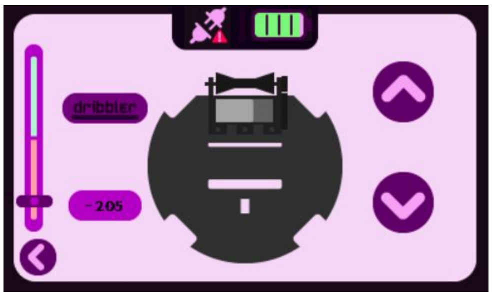


Motor screen test (shown in video too).

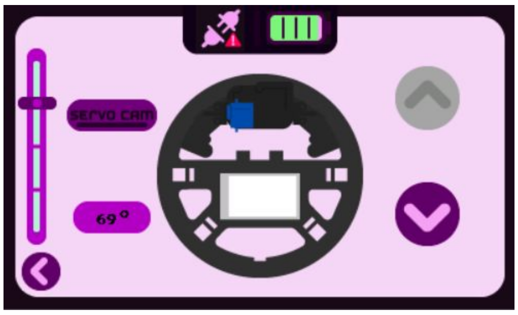
Besides, it is possible to trigger the solenoid to test the robot's kick (no tests have been performed).



In the screen responsible for testing the intermediate layer actuators, it is possible to control the motor speed in the dribbler device.



In the screen responsible for testing the upper layer actuators, you can change the degree of inclination of the camera by setting the servo speed.

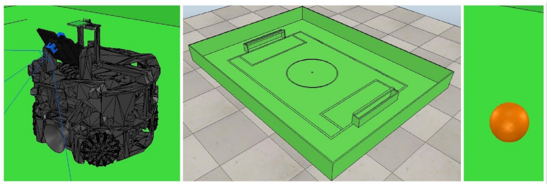


Screen to control the degree of tilt of the camera.

Test to change the degree of tilt of the camera.

Simulation:

The V-Rep program was used to simulate the football game environment because of its compatibility with the C++ language, which allows the same programming logic to be used in both the simulation and the physical robot. The physical robot works through the Arduino IDE. In V-REP, the physical engine ODE was used during all the tests to simulate the behavior of the real physics. However, it was necessary to make form adaptations to the sensors and controllers available in the simulated environment to represent more precisely the same ones of the physical robot. Moreover, it was necessary to decrease the resolution of the robot structure to avoid slowness due to the high amount of polygons.



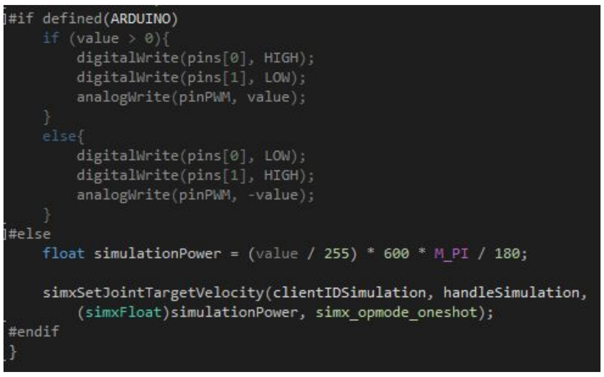
(a) (b) (c)

3D objects inserted in the simulation environment.
(a) Robot imported into simulated environment, (b) Field modeled for simulated environment, and (c) Ball shaped for simulated environment.

Programming:

When imported into the virtual environment, the mass properties are defined, the sensors and virtual controllers are created, and added to the robot. These are used later for the exchange of information with the programming environment used; in this case, Visual Studio.

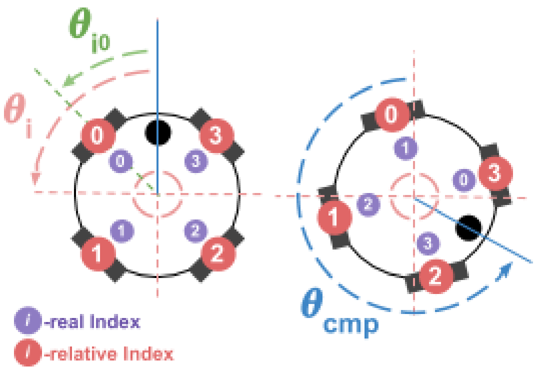
The code is compatible for both the simulation environment and the Arduino IDE. This is possible due to the use of the Object-Oriented Programming (OOP) paradigm. When creating each object, it is detected whether the Arduino IDE is being used for programming. If the Arduino IDE is detected, the called functions will be those of the physical robot. On the other hand, if it is not detected, only the code related to the simulation is activated. Since all sensor classes have this dynamic, the same code can be compiled for both the virtual environment and the Arduino board. Below is an excerpt of the code used to control the power of the motors using this dynamic.



```
#if defined(ARDUINO)
    if (value > 0){
        digitalWrite(pins[0], HIGH);
        digitalWrite(pins[1], LOW);
        analogWrite(pinPWM, value);
    }
    else{
        digitalWrite(pins[0], LOW);
        digitalWrite(pins[1], HIGH);
        analogWrite(pinPWM, -value);
    }
#else
    float simulationPower = (value / 255) * 600 * M_PI / 180;

    simxSetJointTargetVelocity(clientIDSimulation, handleSimulation,
        (simxFloat)simulationPower, simx_opmode_oneshot);
#endif
}
```

Relativization of sensors and motors:

One of the strategies to facilitate the programming of the sensors and motors, while the robot rotates, was the elaboration of a relativization of sensors and motors. This relativization consists in altering which device is used depending on the degree of inclination of the robot.



The image above shows how the relativization of the motors is performed (the black circle represents the front of the robot). As there are four motors, the robot was divided into four zones of 90 degrees. Each motor has a real index and a relative index. The real index is fixed for each motor, and depends on its location in the robot's structure. The relative index of each motor varies according to the degree of inclination of the robot, and the relative index within each zone is fixed. In this way, when programming for the front-right motor to move forward, the motor that will move is always the one with the relative index 3 (regardless of the degree of inclination of the robot).

Being i_r the relative index of a certain motor; i the motor's real index; n the number of components to be relativized; θ_{cmp} the degree of inclination of the robot in a counter-clockwise direction. θ_i the arc size of each zone ($\frac{360}{n}$). θ_{i0} the arc between the front of the robot and the first component.

I wrote the following equation in the code to calculate the relative index of both the motors and the ultrasonic sensors:

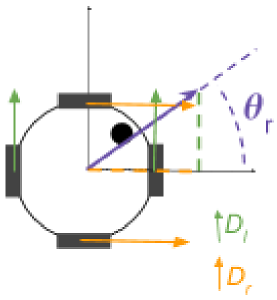
$$i_r = i - \left\lfloor \frac{\theta_f + \frac{\theta_i}{n} - \theta_{i0}}{\theta_i} \right\rfloor$$

The development of this technique made it easier not only to program the movement of the robot, but also to read the values of the twelve ultrasonic sensors.

Robot movement

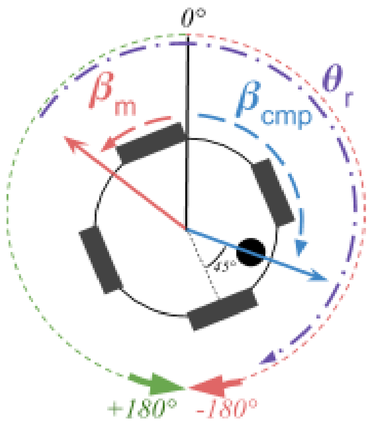
Initially, I had no idea how to move a four-wheel holonomic robot. I was able to come up with the solution after watching a Ted Talks video about drones. Vijay Kumar talked about combining different movements to accomplish the desired movement. Perfect!

When trying to apply this concept in the soccer robot, I realized that there were two main movements: diagonal-right and diagonal-left.



In the picture above, D_r is the power of the diagonal-right and D_l is the power of the diagonal-left motors.

To calculate the power in each motor to make the robot move in a certain direction, I considered each diagonal as one of the legs of a triangle between the desired direction for the movement and a straight 45 degrees from the front of the robot (resulting angle θ_r in the picture above).



In the next equation, angles symbolized by β have their intervals in]-180,180] according to the figure above.

The calculation of the resulting angle (θ_r) can be achieved through β_m (desired drive angle) together with the angle value β_{cmp} for compensation.

β_{cmp} is the reading value of the compass converted from [0,360[to]-180,180].

Thus:

$$\theta_r = 45 - \beta_{cmp} + \beta_m$$

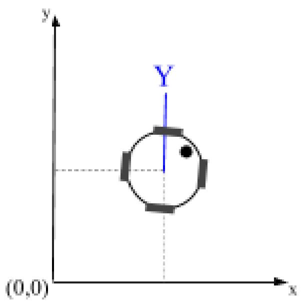
After calculating the resulting angle and applying sine and cosine to discover the power in each motor, a power compensation is performed on each motor to cause the robot to rotate to a desired angle while moving. For this, I used PID: If the robot needs to rotate clockwise while moving, the motors to the left of the robot front are accelerated and the motors to the right of the robot are decelerated, the PID control defines the rate of acceleration that two motors must undergo depending on the error between the angle of inclination of the robot and the angle of inclination desired for the movement.

For example:

If the left motors are accelerated 1.2 times, the right motors will be decelerated $\frac{1}{1.2}$ times. In this way, when applying this control in the simulation, the adjustment of the power of the motors, using PID, allowed the robot to rotate in its own axis to the desired angle at the same time that it moves in a certain direction.

Determine location

Determining the coordinate of the robot in relation to the field was an important step in this robot to perform field recognition.



I treated the field as a Cartesian plane. Thus, it was necessary for the robot to determine its X and Y coordinates regardless of their degree of rotation, using the twelve ultrasonic sensors distributed equally around the robot.

Initially, each relative ultrasonic calculates the distance on the X axis between the sensor and the wall and then stores the possible value of X in a vector. If the ultrasonic is to the left of the robot's Y-line, the value of the ultrasonic to the wall is added directly into the vector. If the ultrasonic is to the right of the robot's Y-line, the distance from the ultrasonic to the wall is subtracted from 182 (distance between the left and right wall.) Thus, all values stored in the vector correspond to the possible values for the robot's X-coordinate. However, some of the ultrasonics may be reading obstacles instead of the wall, so the vector is treated by taking out values that are very different from the majority.

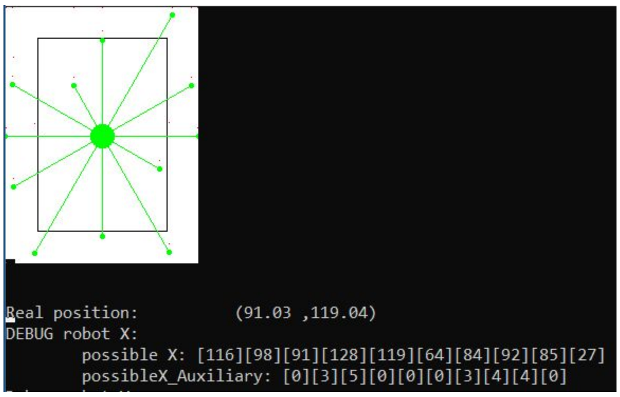
I did it this way:

Considering the vector pX = [92,45,50,87,90,120,93]. First, a nX vector of the same size is generated, each index in pX is compared to the others index in pX, if the difference between the two numbers is less than 7 (arbitrary value), +1 is added to correspondent index in nX.

Thus, the final result of nX is [4,1,1,4,4,0,4]. The values in index 1, 2 and 5 of the vector are eliminated because they are not reinforced by the other sensors. The approximate calculation of the X coordinate is done by the mean of the remaining values in the nX vector.

Additionally, values very distant from the last estimated value for the X coordinate of the robot are also discarded. In this way, the possibility of an erroneous coordinate being calculated as the real one is reduced. This is important because the robot always expects the next coordinate to be near from the actual one, so it discards coordinates too far from the previous one. If all data is discarded because of the high discrepancy between them and the last calculated value for X, the acceptable value for the discrepancy increases over time (a 100 millisecond thread has been created).

The same logic was applied to calculate the Y of the robot.



```
Real position:          (91.03 ,119.04)
DEBUG robot X:
        possible X: [116][98][91][128][119][64][84][92][85][27]
        possibleX_Auxiliary: [0][3][5][0][0][0][3][4][4][0]
```

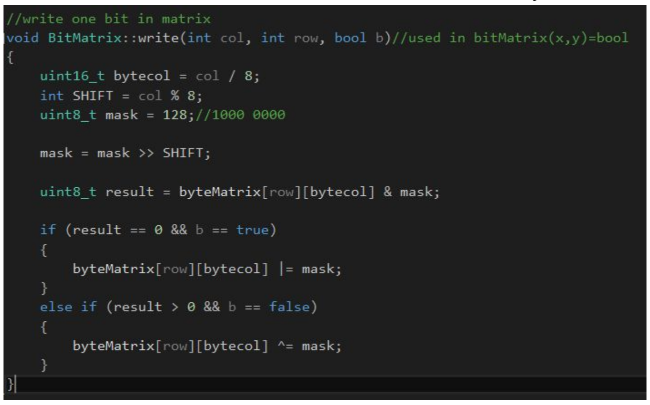
Calculation of the robot's X coordinate.

Matrix of obstacles

In order to allow the robot to create an image of the field in the database, an array of 182x243 was created. Each coordinate in this matrix represents a square centimeter in the real field and indicates whether or not one obstacle was detected at that point. However, if this array were created as the bool type, a total of 44,226 bytes would be required to perform the storage, and the total storage of an ATSAM3X8E processor (used in the main circuit) is 512kB.

In this way, I have developed a BitMatrix class, which can be manipulated as a common bool array. In this class, instead of storing a value of true/false per byte, it stores 8 true/false values per byte. From this array, it is possible to manipulate the same amount of coordinates using only an int 23x31 array, that is, 713 bytes. A reduction of 62 times in the space needed for storage.

To create this class I used bit manipulation. The image below shows the code to write on a bit of this array.

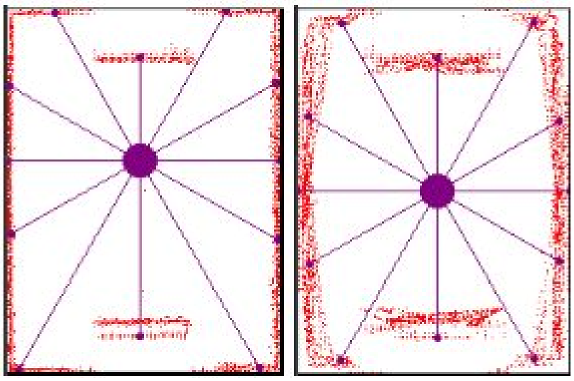


```
//write one bit in matrix
void BitMatrix::write(int col, int row, bool b)//used in bitMatrix(x,y)=bool
{
    uint16_t bytecol = col / 8;
    int SHIFT = col % 8;
    uint8_t mask = 128;//1000 0000

    mask = mask >> SHIFT;

    uint8_t result = byteMatrix[row][bytecol] & mask;

    if (result == 0 && b == true)
    {
        byteMatrix[row][bytecol] |= mask;
    }
    else if (result > 0 && b == false)
    {
        byteMatrix[row][bytecol] ^= mask;
    }
}
```

During the game, from the angle of inclination and coordinate of the robot, the coordinates are calculated for each of the twelve points in which the ultrasonic sensors are reading obstacles. These obstacles can be a wall, a goal, or another robot. All these coordinates are added in the obstacle matrix.

In addition, at the same time the field is mapped by adding obstacle points in the matrix, a matrix cleaning is performed. This cleaning consists of removing previously added obstacles whenever an ultrasonic detects that there is no more obstacle in that position. In this way, the matrix will always be as faithful as possible to the field situation.

Initially, I used the sonar reading opening degree as 2 degrees. However, when performing a test with the actual ultrasonic sensor (URM37 V4.0), I calculated that the best approximate degree of aperture is 15 degrees.



In the images above, each red dot is an added obstacle point in the field matrix. In the image to the left, the degree of aperture in the ultrasonic is of 2 degrees, while to the right, the degree of aperture of each ultrasonic is of 15 degrees.

Coordinate-based movement

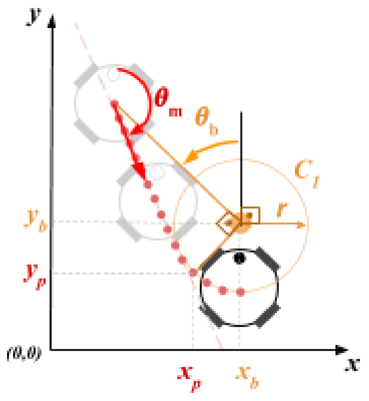
From the moment that the movement of the robot was based on the direction of the movement and the angle of rotation, I began to develop a movement based on the displacement between coordinates of the field. During the game, the robot's movement is based on the displacement between a series of points in sequence.

Considering the current position of the robot and the desired position, a coordinate vector is created. An even index represents X coordinates, and odd index represents Y coordinates. In this vector, the next coordinate that the robot will reach is always in the last two indexes. The robot always moves from its current coordinate to the coordinate defined by the last two indexes on the vectors.

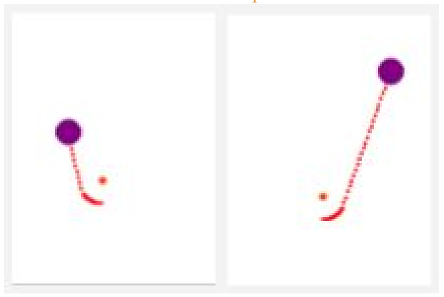
For the robot to move from its current position to the next position, it calculates the tangent of the angle between the X and Y coordinates of the points. It recognizes that it reached the coordinate with a margin of error of 7 centimeters (arbitrary value).

In this way, for the robot to move sequentially between all points, whenever the robot approaches the next coordinate, the last two values of the vector are deleted.

This method of movement allows the previous calculation of the trajectory of the robot to allow it to catch the ball during the game.



The coordinate-based locomotion allowed me a more precise displacement during the game, for example, the path to the ball is generated as follows: a sequence of points (represented in red in the figure above) is generated along a straight line from the robot to the point tangent to the circumference C_1 of radius r - minimum distance between the robot and the ball. Upon reaching this point (x_p,y_p), the coordinate sequence is generated along the radius r until it reaches the position in front of the ball (x_b,y_b). θ_m is the angle of movement of the robot to reach the point of tangency to the circumference C_1.

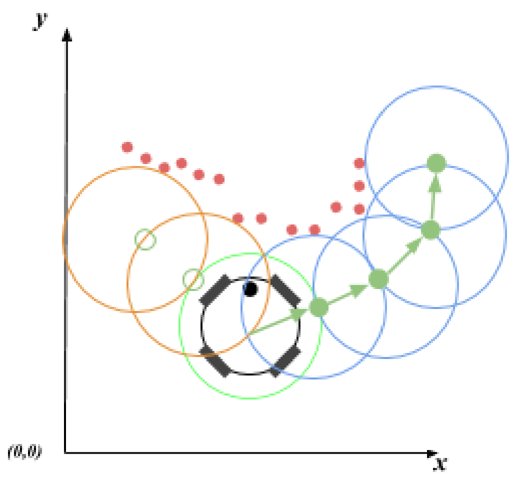


The figure above represents the points generated in the motion coordinate vectors to generate the path described above. The purple circle is in the coordinate referring to the robot in the simulation and the orange circle is the coordinate referring to the ball in the simulation.

Obstacle avoidance

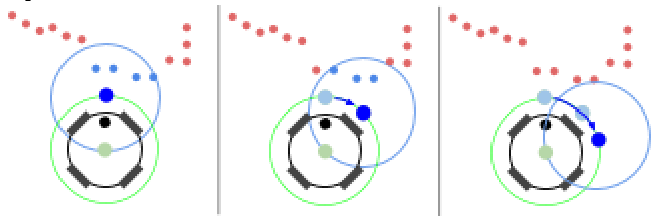
The obstacle avoidance is based on creating a trajectory in order to maintain a minimum distance from the other objects detected by the sensors. During the simulations of this deviation method, to obtain a better result, I used the real coordinate of the robot instead of using the approximate coordinate calculated through the ultrasonic sensors.

Initially, two circle sequences are generated. The first runs a path to the left of the robot, and the second runs along a path to the right of the robot. The trajectory of the robot to the final coordinate is represented by the series of coordinates of the centers of each circle of the series of generated circles.



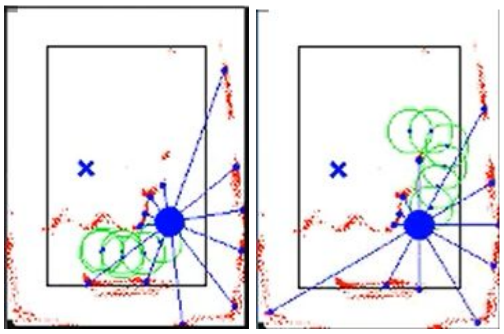
In the image above, you can see the two trajectories that are generated at all times. A trajectory spreads to the left (created by orange circles) and the other spreads to the right (created by blue circles). The red dots indicate the points at which obstacles were detected by the ultrasonic sensors. The propagation of circles occurs in such a way as to circumvent these points.

The propagation of circles on the right happens clockwise as shown in the image below. To define the first coordinate of the trajectory, a circle is created in front of the robot. After this, the blue circle continues in an hourly trajectory over the green circle until no obstacle point is within the perimeter of the blue circle. The image below represents the creation of this first circle.



The function that generates this first circle is recursive, which allows to generate all the other circles of the same trajectory following the same actions applied in the first circle. If the function that generates the first circle is hourly, all the next circles generated for this trajectory will follow clockwise.

After generating both trajectories, the robot calculates the distance between the last coordinate of both trajectories to the destination point and performs the trajectory that will end in the closest coordinate of the target.



The above images were taken during a simulation. It is possible to observe the two paths generated to reach the point "X".

Interesting fact: The idea for this deviation technique occurred in the shower.

Ball Location

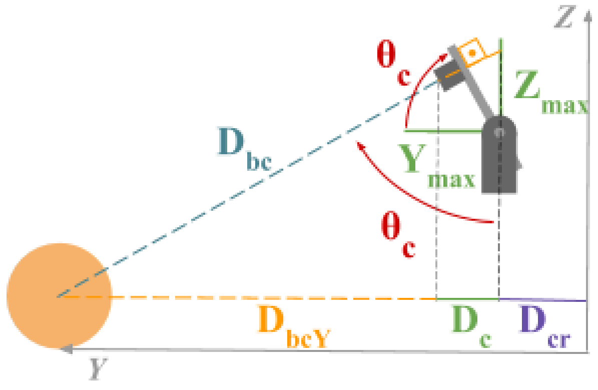
The Vision Sensor used in the simulated environment returns the objects captured on the scene in a similar way to the Pixy camera, as shown in the table below, which shows the comparison between the actual camera settings and the Vision Sensor of the simulation.

Comparison between Real Camera and Vision Simulation Sensor

Comparative Items	Pixy CAM	Vision Sensor
fps	50	20
fov vertical	47	75
fov horizontal	75	75
Resolution	1280x800	256x256

fps: frames per second; fov: field of view; Resolution: pixels.

Although the use of higher fps and better image resolution was allowed during the simulation, their use resulted in slow processing of information during the tests.



The location of the ball in the field occurs in two steps. The first occurs from the angle of inclination of the camera θ_c to calculate the distance of the ball to the center of the robot. In this calculation, D_{bc} is the distance from the camera to the ball, D_{bcY} is the projection of D_{bc} on the Y-axis (1), D_c is the distance from the camera lens to the stand base of the omnidirectional camera (2), and D_{cr} is the distance from the omnidirectional support base to the center of the robot, which is constant.

$$D_{bcY} = \sin \theta_c \cdot D_{cb} \qquad (1)$$

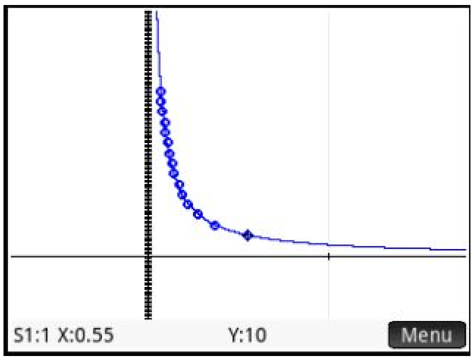
$$D_c = Y_{max} \cdot \cos \theta_c \qquad (2)$$

In this way, the total distance of the ball to the center of the robot is given by the sum of D_{bcY}, D_c e D_{cr}.

The second part for calculating the location of the ball consists of transforming the distance of the ball relative to the robot in an absolute distance to the origin of the field through (3). Where X_b and Y_b are the absolute coordinates of the ball, D_t is the total distance of the ball to the center of the robot and θ_{cmp} is the degree of inclination of the robot.

$$\begin{bmatrix} X_b \\ Y_b \end{bmatrix} = D_t \cdot \begin{bmatrix} \sin(-\theta_{cmp}) \\ \cos(-\theta_{cmp}) \end{bmatrix} \qquad (3)$$

To figure out the distance from the camera to the ball D_{bc}, I created a graph using the same graphing calculator I used in the SAT math test. In this graph, in the X-axis are the values of the percentage height of the ball in the camera and in the Y axis the actual distance from the camera to the ball.



The graph generated to calculate the distance from the camera to the ball was:

$$D(x) = 5.558 \cdot X^{-0.008}$$

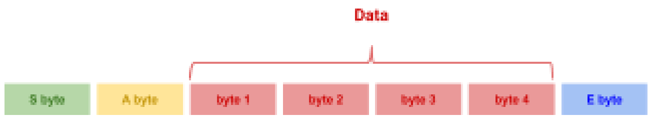
Communication

Since the main circuit has two microcontrollers, they need to communicate at all times and have methods to detect if bytes have been altered in the process. For that, I used a technique I had previously seen on YouTube. It consists of initially sending one start byte and one addressing byte for the microcontrollers to distinguish values referring to actuators, sensors and communication in general. Furthermore, at the end of sending each series of information, a byte is sent with the "sum" of the values sent to identify if the data has not been corrupted.

The first byte to be sent is the start byte. This byte has a fixed value of zero and its function is to allow the other microcontroller to identify the beginning of each sending data series. To avoid confusion between byte values and the start byte, all bytes of values that should be sent as zero are sent as one.

The addressing byte, which is sent shortly after the start byte, allows the microcontroller to distinguish what each byte of sent value means.

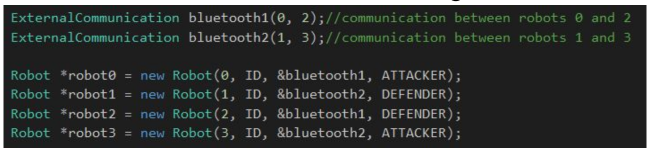
After sending the series of bytes of values, the end byte is sent, which corresponds to the sum of the bytes of values. Because a byte can only handle values from 0 to 255, the result of the sum is placed in this range. That is, if the sum is 300, 255 is subtracted and the value sent for confirmation is 45.



Different types of bytes during communication.

This communication system was used both for UART communication between microcontrollers and for communication between two robots in the simulation.

During the simulation, I created the ExternalCommunication Class. The address of each object of this class is sent as a parameter to objects from Class Robot. This definition is shown in the image below.



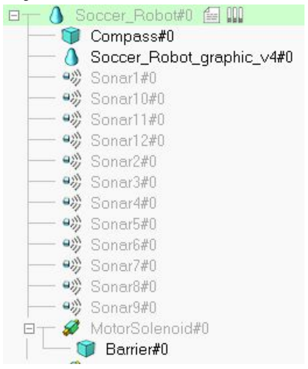
```
ExternalCommunication bluetooth1(0, 2);//communication between robots 0 and 2
ExternalCommunication bluetooth2(1, 3);//communication between robots 1 and 3

Robot *robot0 = new Robot(0, ID, &bluetooth1, ATTACKER);
Robot *robot1 = new Robot(1, ID, &bluetooth2, DEFENDER);
Robot *robot2 = new Robot(2, ID, &bluetooth1, DEFENDER);
Robot *robot3 = new Robot(3, ID, &bluetooth2, ATTACKER);
```

This class only supports performing communication between two robots. The communication happens in the same way as reading the data of the serial communication of the Arduino, the first one that is sent, is the first data to be read (FIFO - First In First Out). In this way, two vectors were created per ExternalCommunication object. The first is used to send data from robot 1 and read data from robot 2. The second is used to send data from robot 2 and read data from robot 1.

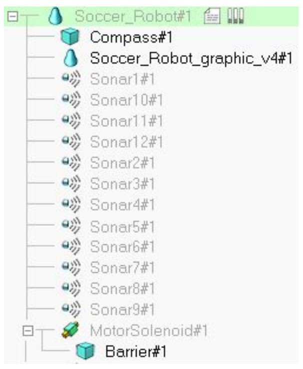
Values are added to a vector when a robot sends information from one robot to another. When the other robot reads the data in the vector, the read data is deleted from the vector.

Multiple robots

As all robot soccer programming has occurred within the Class Robot, it is possible to create multiple Robot objects to control different robots in the simulation.

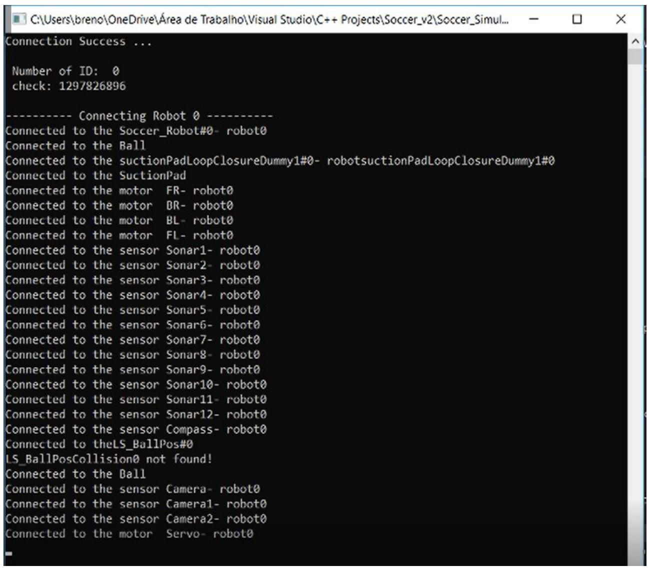


```
Soccer_Robot#0
    Compass#0
    Soccer_Robot_graphic_v4#0
    Sonar1#0
    Sonar10#0
    Sonar11#0
    Sonar12#0
    Sonar2#0
    Sonar3#0
    Sonar4#0
    Sonar5#0
    Sonar6#0
    Sonar7#0
    Sonar8#0
    Sonar9#0
    MotorSolenoid#0
        Barrier#0
```



```
Soccer_Robot#1
    Compass#1
    Soccer_Robot_graphic_v4#1
    Sonar1#1
    Sonar10#1
    Sonar11#1
    Sonar12#1
    Sonar2#1
    Sonar3#1
    Sonar4#1
    Sonar5#1
    Sonar6#1
    Sonar7#1
    Sonar8#1
    Sonar9#1
    MotorSolenoid#1
        Barrier#1
```

The above images were taken from the structure of two robots in the simulation. When the Robot object 0 is created, the robot is connected to the structures that are finalized with "# 0". In turn, when the Robot object 1 is created, the robot is connected to the structures that are finished with "# 1". Once the programming is compiled, before the screen appears with the robots, it appears which structures of the simulation are being connected with the objects of the code. Errors in connection with the structure are indicated on this screen.



C:\Users\breno\OneDrive\Área de Trabalho\Visual Studio\C++ Projects\Soccer_v2\Soccer_Simul...

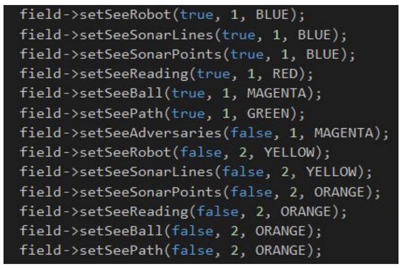
```
Connection Success ...

 Number of ID:  0
 check: 1297826896

---------- Connecting Robot 0 ----------
Connected to the Soccer_Robot#0- robot0
Connected to the Ball
Connected to the suctionPadLoopClosureDummy1#0- robotsuctionPadLoopClosureDummy1#0
Connected to the SuctionPad
Connected to the motor  FR- robot0
Connected to the motor  BR- robot0
Connected to the motor  BL- robot0
Connected to the motor  FL- robot0
Connected to the sensor Sonar1- robot0
Connected to the sensor Sonar2- robot0
Connected to the sensor Sonar3- robot0
Connected to the sensor Sonar4- robot0
Connected to the sensor Sonar5- robot0
Connected to the sensor Sonar6- robot0
Connected to the sensor Sonar7- robot0
Connected to the sensor Sonar8- robot0
Connected to the sensor Sonar9- robot0
Connected to the sensor Sonar10- robot0
Connected to the sensor Sonar11- robot0
Connected to the sensor Sonar12- robot0
Connected to the sensor Compass- robot0
Connected to theLS_BallPos#0
LS_BallPosCollision0 not found!
Connected to the Ball
Connected to the sensor Camera- robot0
Connected to the sensor Camera1- robot0
Connected to the sensor Camera2- robot0
Connected to the motor  Servo- robot0
```

One of the major benefits of programming the soccer robot through the simulation using OOP is the ability to evaluate the performance of the attack and defense strategy simultaneously when performing games with multiple robots.

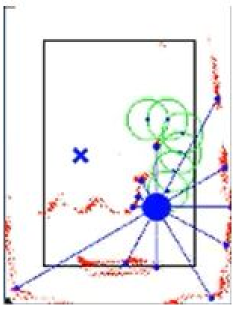
Game Monitoring

To perform the monitoring of the objects in the field, I developed a screen to show the game situation during the simulations. The creation of this screen was crucial in order to develop the recognition of the field and the diversion of obstacles.

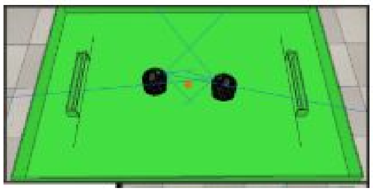
This screen is basically a 182x243 array. Each index has the color of the pixel to be displayed. The Class FieldDrawn controls and organizes everything that will be shown on this screen. Also, it is possible to enable/disable and choose the color of each thing that will be shown on the screen.

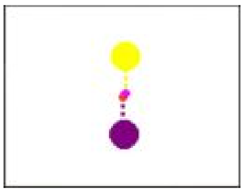


```
field->setSeeRobot(true, 1, BLUE);
field->setSeeSonarLines(true, 1, BLUE);
field->setSeeSonarPoints(true, 1, BLUE);
field->setSeeReading(true, 1, RED);
field->setSeeBall(true, 1, MAGENTA);
field->setSeePath(true, 1, GREEN);
field->setSeeAdversaries(false, 1, MAGENTA);
field->setSeeRobot(false, 2, YELLOW);
field->setSeeSonarLines(false, 2, YELLOW);
field->setSeeSonarPoints(false, 2, ORANGE);
field->setSeeReading(false, 2, ORANGE);
field->setSeeBall(false, 2, ORANGE);
field->setSeePath(false, 2, ORANGE);
```



The image on the left was the setting used to show the image to the right during the simulation. The robot and the ultrasonic readings (sonars) were defined in blue, the readings of the field were defined in red and the path to be covered by the robot was defined in green. When using more than one robot in the simulation it is possible to choose the color separately from each robot (in this setting all the color options for the second robot are disabled because only one robot was used during this simulation).





In the image above, the two robots on the left are shown in the prompt array. It is possible to see in the image to the right that each robot generated a straight path until the ball (points between the robots and the ball).

Game Class

This particular class is not related to the robot itself, but to monitoring the position of the ball and robots during the game. In this way, it is possible to create a kind of "judge" in the simulated environment and deploy rules present in RoboCupJunior Soccer during the match to better simulate the actual game. Currently, the only function of this class is the repositioning of the ball to the center of the field as it exits outside the perimeter of the field.

Current situation:

Although the simulations of the soccer robot are already well developed. The physical robot still needs improvements to work perfectly.

The solenoid used for the kick was initially very weak. To increase its strength, I used a Step Up module to increase the voltage from 12V to 50V. Even so, the final power reached not as much as it was expected. Moreover, testing using a TIP120 transistor to allow control of the 50 V solenoid from a 5 V signal resulted in a damaged TIP120. To solve the problem of power, I plan to print the solenoid structure and surround it with a copper wire to have control of the final power. In the initial tests with this solenoid I will use a TIP120. If it does not work, I will use a relay.

The dribbler device described in the modeling began to be built, but due to friction in the gears, the motor could not get enough force to turn them.

Tests of movement of the robot have already been performed. Although the robot calculated the power of each motor correctly, the final movement was imprecise because the PID was not implemented. The PID was not deployed due to interferences in the 9DoF sensor. I detected a great distortion of the electromagnetic field around the robot, I intend to be able to solve this problem when I start to use the developed circuits.

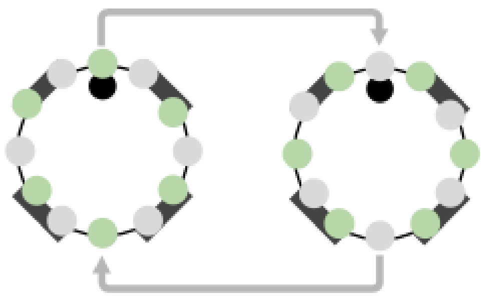
In addition, the circuits only arrived on December 5, 2018. In this way, only the peripheral circuits and the secondary circuit were tested, they worked perfectly. The main circuit is still being welded because I will only start testing with it when I am sure there are no wrong soldering points that lead to the burning of any component. I am still making sure all the connections are correct.

While the circuits did not arrive, I started to test the robot using an Arduino Mega and an Arduino DUE. The method of communication between them was described previously. It is important to note that since the Arduino Mega operates at 5V and the Arduino DUE operates at 3.3V, I used a bi-directional logic level converter that uses the BSS138 (same component used in the main circuit to allow communication with 5V sensors).

When using the two Arduino boards instead of the main circuit, I tested almost all the sensor monitoring screens (the light sensor has not yet been tested with the display). Further, all test screens for actuators also worked.

Calculating the distance from the robot to the ball using the camera worked very accurately at distances less than 1 meter.

The biggest problem encountered so far was with the ultrasonic sensors to locate the robot in the field. Since all emit the same waveform, whenever the echo returns, more than one sensor detects the wave sent. One sensor causes interference on the other sensor. Initially, I tested the alternating reading of the ultrasonic sensors, but I realized that while the sensors were on, they were emitting sound waves. The best solution was to control which sensors would be turn on/off at any time as in the image below.



🟢 -Sensor ON

⚪ -Sensor OFF

Although this solution worked, I realized that each URM37 sensor needed an average of 800 milliseconds to start returning correct values after powering them on. In this way, the reading became inefficient by the waiting time.

I am currently studying the second solution for the location of the robot in the field. This consists in using the acceleration of the robot calculated by the 9DoF sensor to implant an IMU (Inertial Measurement Unit) with an Extended Kalman Filter. Due to the accumulation of the error over time, I intend to use ultrasonic sensor readings to certify the current location of the robot. I am currently using a Kalman Filter only to handle Pixy camera values and ball possession detection.

During the development of robots for the category RoboCupJunior Open, the simulated environment is not much exploited by the competitors. In spite of this, the previous creation of the robot in the virtual environment and programming with a hybrid code presented benefits not only for a better understanding of the sensor readings, but also, creation of better strategies for the attack and defense, since it is possible to conduct tests simultaneously with attack and defense strategies.

The possibility of playing games with customized robots in simulation using RoboCupJunior Open rules also has a social application. The current cost to build competitive soccer robots is high for several high school students around the world. The possibility of creating a new category in RoboCupJunior using a simulated environment with customized robots for the Junior Open category, allows these students to create soccer robots using only a computer. In this way, it will be possible to include more students in RoboCupJunior soccer leagues.