ROBOTICA 2021-2022

1st Diego Gallardo Zancada
Cáceres, Spain
dgallards@alumnos.unex.es

2nd Alberto López Hoyas
Cáceres, España
alopezwy@alumnos.unex.es

I. INTRODUCTION

We have done a project about the autonomous robot Giraff-X using C++/Qt5 where we have learnt how to develop the robot code to perceive the environment surrounding it, avoiding obstacles, mapping rooms, etc... as proposed in the book *Corke´s Robotics, Vision and Control.*

First of all we have installed our tools (Qt5, C++, GCC, git, cmake and our IDE vscode). The installation step was a bit complicated and hard. We had some problems with the coding environment and we were not successful to run in our computer because of the graphical limitation so we stick with the ones of laboratories.

As the books says, the robot it is equipped with a camera and a laser sensor which senses the environment. Then we took the second approach, which it is to create a map of the environment and use it to create paths, rooms, doors and identify obstacles.

The robot will be able automatically without human control to reach a point, scan rooms, detect and avoid obstacles and move into other rooms.

This robot was planned to interact with people and to be teleoperated but now it is able to interact and move around without human presence.

Teleoperation presents several problems like having to hire a specialized operator which can commit errors because of the cognitive fatigue or others factors.

A. Materials.

We have been given a repository on github that contains the folders of robocomp, robocomp-robolab and beta-robotica-class. This folders are the ones that set up the virtual environment to practice with a virtual robot on it. The software used to set up this environment its CoppeliaSim. This folders also contain the code necessary to interact with the robot with a joystick.

We also have been given a document in google docs that contains the steps to realize the project and the steps to follow.

Our github repository where this robot project it is developed is **this.**

II. FIRST ROBOT STEPS.

The objective of this first part was to make the robot move automatically without colliding with the walls of a room or obstacles that we can put in it. To check that this task has been carried out correctly, we can run the component of the vacuum cleaner that will show us in a window the percentage of floor that we have visited of the total room in which the robot is located and the time that has passed since it has been running the program.

We started knowing that when the robot approached a wall or obstacle it should progressively slow down. So we took a value to make the robot decreasing as it exceeds that value so that the robot when it gets closer and closer to the wall or obstacle would give us time to stop it, since it cannot stop immediately, it has to be done progressively.

So we thought that in order to know which way to turn we should know which way there is a larger area so we know that in that direction there is more space to move. In order to do this we must take all the distances that the sensors provide us, dividing them into two options: left or right. After having added them separately, we divide them by the number of samples we have taken from each one so that we obtain the average distance of each side and depending on which is greater, we will turn to that side.

We also check the distance remaining to reach the target, in this way as we get closer we will decrease the speed from 500 to 300.

We have made a state machine which will choose to go straight, turn left or turn right.

Fig. 1. **State Machine.**

```
switch(state){
    case FORWARD: move_robot(advance_speed,0);
    break;
    case TURN_LEFT: move_robot(0,rotation_speed);
    break;
    case TURN_RIGHT: move_robot(0,-rotation_speed);
    break;
}
```

States: (FORWARD, TURN-LEFT, TURN-RIGHT)

Condition to FORWARD:

$$\text{Laser distance} > 800$$

1. To continue advancing, the condition must be that the lasers do not have to stop at less than 800 of distance, we only take into account the most central lasers of the robot, i.e. we only look at the distances of the lasers of the robot that go from angle -1 to angle 0.8.

Condition to jump into TURN-LEFT or TURN-RIGHT:

$$\text{Laser distance} < 800$$

To chose where to turn the condition will be:

if((right lasers distances / right number of lasers)
$>$ (left lasers distances / left number lasers))
TURN RIGHT
else
TURN LEFT

2. To turn left, the condition must be that the average distance of the negative angle lasers is greater than the distance of the positive angle lasers, otherwise the robot will turn right.

To see the movements of the robot over a room and to check the efficiency of the robot in the room in case it has to clean a room, we have installed a component called vacuum cleaner that will give us the percentage of the whole room visited. We performed three tests for 180 seconds and the best result are as follows:

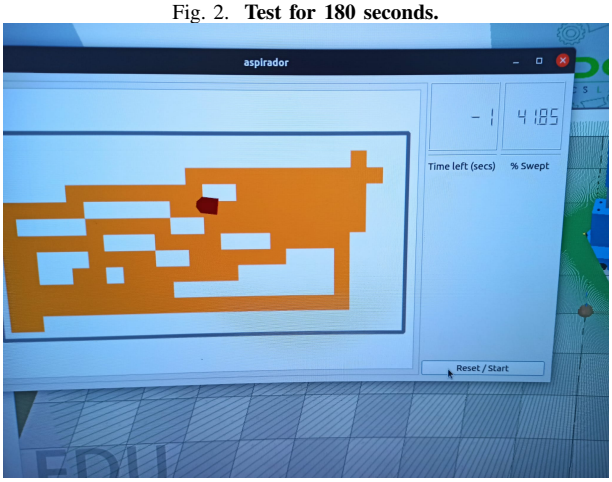
III. REACHING A POINT .

For this third part the objective was to take the robot to a location given by the vacuum component.

This location where the robot will go is chosen by clicking on the vacuum cleaner component.

The first step is to introduce some modules in our new component to draw a laser as a field of view of the robot and collect the mouse clicks.

Fig. 2. **Test for 180 seconds.**



The module to visualize the polygon of the lasers is realized by computing all the lasers of the robot and introducing to the polygon the x and y coordinates of each of the lasers, these x and y coordinates of each laser are calculated by multiplying the distance until the laser has vision (the maximum of the laser or the distance until it collides with something) by the sine in the case of x and by the cosine in the case of y.

In this third part we introduce different variables to be able to visualize the robot as well as the polygon that forms the lasers of the robot.

We also add a struct type called target that contains a QPointF and a boolean. The QPointF its a type of variable that contains two coordinates (x and y) and the boolean will tell us if the target it is active or not, if it is not active we could click again in the component and set another target.

First we must clarify that for this practice that the coordinate axis that we have in the vacuum cleaner component, is different from the coordinate axis that has the robot implemented. Because of this it is necessary to make a module to pass a point from the world coordinate axis to the robot coordinate axis. To perform this change of coordinate axis, we create a module called ***world2robot*** to which we pass as parameter the robot's base and return a tuple with two points, one x and one y.

Within this module, we extract the angle of the robot base and then create two Eigen vectors of two positions, in one vector we will save the x and y coordinates of the robot base, and in the other the x and y coordinates of the target position. The formula for transferring coordinates from one axis to another is as follows:

Formula for transferring coordinates.

$$T_r = R^T(T_w - R_w)$$

The first parameter(R^T) will be the transpose of the clockwise rotating matrix constructed by the sines and cosines of the angle of the base.

Clockwise Rotating Matrix.
$$\begin{matrix} \cos(a) & \sin(a) \\ -\sin(a) & \cos(a) \end{matrix}$$

The second parameter (T_w) will be the x and y coordinates of the target. And finally, the third parameter (R_w), will be the x and y coordinates of the robot base.

We have created a function called ***gaussian***, to which we pass as parameter the arctangent of the target points. This function returns a float that will be the value that multiplies (slow down) the speed of the robot, this is achieved by a gauss function.

Gaussian Function for the second coefficient.

$$\mathrm{f(x)} = \frac{1}{\varrho\sqrt{2\varpi}}\varepsilon^{-\frac{(x-\varkappa)^2}{2\varrho^2}}$$

We also have created a function called ***calculateSpeed*** which returns the result of multiplying: the maximum speed of the robot (1000), a function with respect to distance and the result of the ***gaussian*** function. The function with respect to distance will be 0 if the target is at less than 200 and 1 when it is over 200.

calculateSpeed = 1000 * result of gaussian function * function respect to distance

The second step will be to move the robot to the indicated coordinates. For this second step we have to pick up the coordinates where we have clicked with the mouse and pass them to the robot converting them from the coordinate axis given by the world to the robot's coordinate axis with the ***world2robot*** function. With the coordinates where we have to go and the angle that we form with the point we will obtain the distance to the point by the rule of the hypotenuse and the legs, and the speed of rotation that will be the angle. Our forward speed will be the result of the ***calculateSpeed*** function.

The last part will be to introduce to the robot this calculated advance speed and the rotational speed that has been previously discussed.

Video of the robot reaching a point

IV. Reaching A Point Avoiding Obstacles.

For this fourth part we start from the code generated in the step of arriving at a point and in the step of avoiding obstacles. The goal of this part is to get to a point even if there are obstacles in the way.

We started by creating a state machine, in which we will decide in which mode our robot will operate depending on calculations that are performed before entering it. In the state machine, the robot will be able to choose between three states:

Fig. 3. **State Machine.**

```
switch(state){
    case IDLE: move_robot(0,0);
    //WAITS FOR AN INPUT OR HAS REACHED A POINT.
        break;
    case FORWARD: move_robot(adv_speed,0);
    //IT HASNT ENCOUNTERED AN OBSTACLE IN SIGHT, GOES STRAIGHT.
        break;
    case TURN: move_robot(0,rotation_speed);
    //IT HAS ENCOUNTERED AN OBSTACLE; STARTS TO TURN. GOES TO PREVIOUS STATE.
        break;
}
```

States: (IDLE, FORWARD, TURN)

Condition to **IDLE**: distance between target and robot is less than 200.

1. The first state will be the one in which the robot stays still, that is to say when it reaches the target. In this state we set the advance speed and the rotation speed to 0.

Condition to **FORWARD**: There are no obstacles in sight and its more than 200 away from the target.

2. The second state will be advancing one, towards the point because it has it in sight. In this case we use the function ***calculateSpeed*** that we used in the last step with the distance to the target and the result of the function ***gaussian***. Then he robot will start moving towards the target in a straight line.

Gaussian Function.
$$\mathrm{f(x)} = \frac{1}{\varrho\sqrt{2\varpi}}\varepsilon^{-\frac{(x-\varkappa)^2}{2\varrho^2}}$$

Condition to **TURN**: There is an obstacle within the range of the robot, less than 900 from the robot.

3. The third state will be the skirting one, this will happen when the robot encounters an obstacle and has to go around it or look for a path where it can go.

To turn, we use a function called ***girar*** that has two loops,in the first one, the rotation speed is set to one, and in the next one it is advanced.

Video of the robot avoiding obstacles

V. MAPPING.

For this last part the goal task is to learn a map of the environment using a GRID and use it to navigate among the rooms.

GRID it is a 2D array of floor tiles that can be flipped between free and occupied. *robotGrid* will be the variable that our grid has.

First we will have to load a new world in Coppelia, our simulation platform of the environment in which the robot is located.

After this we have to replace the block where we read the robot´s position because we are no longer using *bState*, now we will be using *rState*.

For this fifth part we will have to change the method of ***world2robot*** that we created in the previous steps since now it will have as parameters the *rState* and a coordinate (x and y).

We will also have to create the method that makes the change of the axis of coordinates in reverse, that is to say from the robot to the world, this method has the name robot2world and its input parameters are the same as world2robot.

The output of these methods will be the same, an x and a y but each one in a different coordinate axis.

To update the *robotGrid* while the robot is moving, we will have to change the cells that are being covered by the lasers. All the cells through which the laser passes mean that they are not occupied except the last one, where the laser tip will be. If the laser distance is 4000 it means that none of the cells are covered. The idea is to add hit and misses for each cell, to avoid excessive cell state changes, a Bayesian filter is used.

The method that performs these updates is called ***update-map*** which has as input parameters the laser data and the *rState* that we discussed at the beginning. This method compute all lasers and build a parameterized line between (0,0) and the tip of the laser beam. This method will go from the origin to the tip giving passes of half the size of the grid tile. For this we will have to pass the coordinates of these steps to the world reference system with the method ***robot2world*** and we will call the grid method add-hit or add-miss depending of:

if(Laser distance is less than 4000) then ADD-HIT

if(Laser distance is greater than or equal to 4000) then ADD-MISS

After having performed the update-map we will have to create a state machine to explore the rooms.

States: (IDLE, INIT-TURN, EXPLORING, TO-NEXT-DOOR, SEARCHING-DOOR)

Fig. 4. **State Machine.**

```
switch(state){
    case IDLE: move_robot(0,0);
    //WAITS FOR THE ROBOT TO INITIALIZE, GOES TO NEXT STATE
        break;
    case INIT_TURN: calculate_angle(rz);
    //CALCULATES ANGLE. GOES TO NEXT STATE
        break;
    case EXPLORING:
        move_robot(0,rotation_speed);
        searching_doors();
    //STARTS TURNING, SEARCHING FOR DOORS. STOPS TURNING WHEN HAS REACHED 180 DEGREES.
    //GOES TO NEXT STATE
        break;
    case SEARCHING_DOOR:
        nextDoor = selectDoor(DoorVector);
    //SELECTS THE DOOR TO GO. MAKES A NEW NODE PATH. GOES TO NEXT STATE
        break;
    case TO_NEXT_DOOR:
        go_to(NextDoor);
    //SELECTS THE NEXT NODE PATH, GOES TO THE DESIRED POINT. GOES BACK TO EXPLORING STATE.
        break;
}
```

First state **IDLE**: It is the first state. It will enter automatically.

1. The first state will be the one in which the robot starts moving. It leads to the second state. Second state **INIT-TURN**:

The seconds state calculates the angle of the robot and stores it as a heading. Then leads to the third state.

2. Calculates the angle measuring the component rz in radians.

Third state **EXPLORE**: The third state, starts moving the robot clockwise, until its facing 180 degrees, searching for a door.

3. Calculates the angle measuring the component rz in radians again, and compares it with the previous calculated. Then, goes to the next state.

Fourth state **SEARCHING-DOOR**: The fourth state, selects which door to go to.

4. Selects the last door encountered. Then push the external midpoint of the door to a path vector. This vector selects the way to go of the robot.

Fifth state **TO-NEXT-DOOR**: The fifth state, moves the robot to the external midpoint of the given door.

5. The robot is ordered to go to the external midpoint of last encountered door, which is 1000 away from it. Then the robot goes to the **EXPLORING** state again. The external midpoint is reached using an implementation of the Dynamic Window Approach algorithm, cited in the bibliography.

To explore the rooms we will have to create a method that detects the doors.

To develop this method we will have to create a struct type Doors containing two points, of type Eigen::Vector2f.

We create a vector of gates to be able to store them.

We will also have to add a == operator to compare two gates so that we do not insert the same gate in the gate vector.

While rotating, we must compute the partial derivative of the measurement function (laser array) with respect to the distance. Then copy to a new vector the peaks in the derivative that are greater than 600 in absolute value.

Now we have to check that the pairs of peaks are gates, the conditions to be a gate are the following ones:

1. The distance between a pair of peaks is between 700 and 1200.

2. Now we create a gate with the pair of peaks and we have to check that this gate it is not among the gates of the gates vector with the operator == that we have created.

Finally the gates of the gates vector are painted into the area with the color blue and the map is updated.

This video shows the robot exploring the environment.