ROBOTICA 2020

G1-X1
Marta García Tornero., Irene Melchor Félix.
mgarciajr@alumnos.unex.es , irmelchor@alumnos.unex.es
University of Extremadura

***Resumen*—This document explains the problems and solutions that have been obtained when we have tried to control the movement of a robot in a simulation in the Robocomp environment. Next, we will talk about the different activities proposed throughout the sessions. With these iterations, the robot has been able to have functionalities that it did not have before.**

- **Second activity: Aims to move the robot through the environment, avoiding any object that is on the simulation map.**
- **Third activity: The objective of this iteration is to make the robot go to a specific point on the simulation map, which is given by clicking with the mouse. In this case, obstacles that may exist on the map will not be taken into account.**
- **Fourth activity: This iteration is a combination of the second and third activity, that is, in this case, the robot will have to go to the point indicated by clicking with the mouse avoiding obstacles.**
- **Fifth activity: In this task, the first thing we do is give the robot a map so that it can know the environment in which it is going to move. On this map there will also be different obstacles, of which the robot does not know its position and will have to avoid them.**

I. Introduction

Throughout the course, we will use Robocomp, an open source robotics framework based on component-oriented programming and C ++ as the programming language.

Component-oriented programming is a branch of software engineering, based on the decomposition of already formed systems into functional or logical components with well-defined interfaces that are used for communication between components.

In this type of programming, components can be replaced quite easily when the user or the program requires it.

Therefore, taking into account all that has been explained, component-oriented programming is the one that best adapts to the environment to be used.

Once all this information about the environment is known, the different tasks that will be carried out throughout the course will allow us to interact with the commented tool. These tasks will consist of implementing different components to our project and various tasks that will allow the robot to dodge objects, or to choose the shortest way to calculate its route, among others.

II. SECOND TASK

II-A. Problems

The problems we have faced have been mainly two. The first was that the robot did not move by itself, but via a joystick. The second problem was that the robot did not avoid obstacles and did not have a correct delimitation of the map. When solving the first problem, we encountered another setback. This time the robot swept a very small percentage of the map due to its slow speed. When solving the second problem, we observed that the robot invaded the space of the obstacles. This was because the initial threshold was very low.

II-B. Solutions

To solve the problems mentioned above, we have followed a series of steps: In the first place, a new component called chocachoca was installed, which is responsible for driving the robot through obstacles avoiding colliding. By installing it, we solved the two main problems. Next, we increased the speed when the robot was going in a straight line. If the robot had to turn, the speed was decreased by half and a turning angle was introduced.

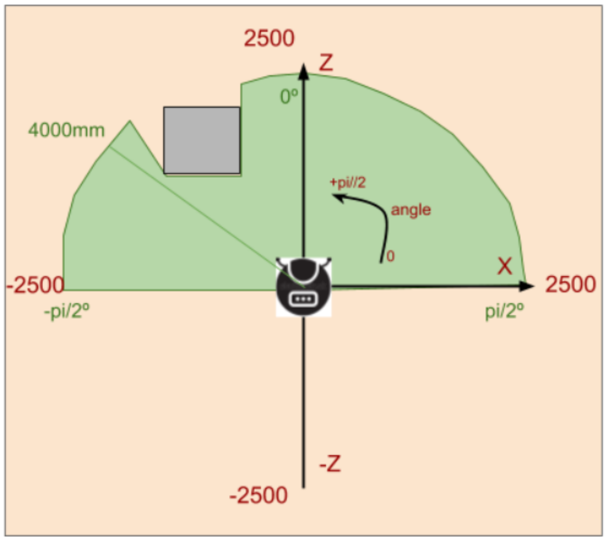

Figura 1.

Despite all these modifications, the robot was still sweeping a low percentage of the map, so we decided to implement an algorithm that checks the boxes adjacent to the robot, to verify if it had passed through them or not. In this way, it would not go through a box already visited.

To implement the adjacent boxes algorithm, a neighbor vector was created, containing the positions of the boxes adjacent to the robot. To check whether or not it had passed through any of these boxes, the vector was traversed position by position, and if it found a box it had not passed through, it would go towards it.

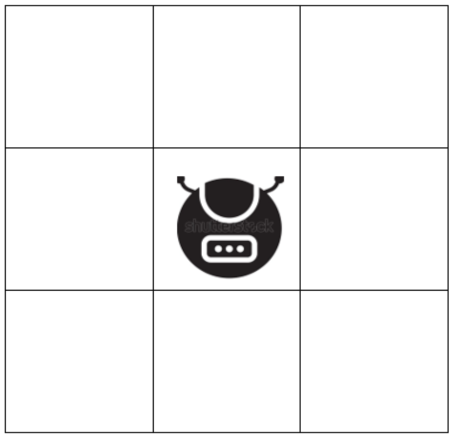

Figura 2. Adjacent matrix

II-C. *Executions*

After carrying out the entire implementation, 5 executions of 120 seconds each have been carried out to study the behavior of the robot. As can be seen in each of the captures, the times oscillate between 32.32 percent and 35.52 percent. So the average obtained between the five executions is 33.856 percent.

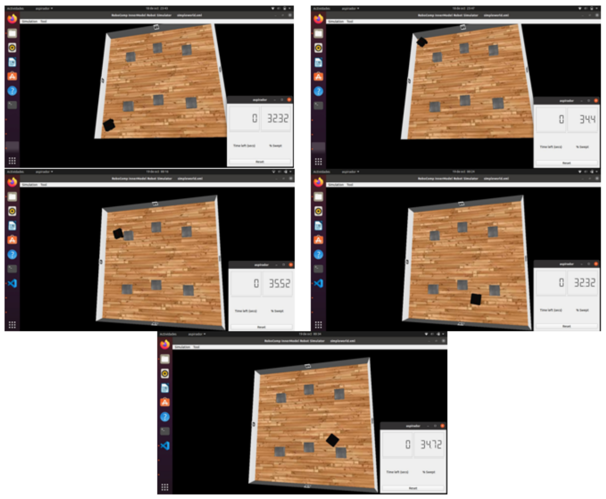

Figura 3. Execution screenshots

III. THIRD TASK

III-A. *Problems*

In this iteration, the first problem that could be found was to go to a given point on the map by clicking the mouse.

To do this, we needed to adapt the coordinates of the robot to the map, since the robot and the map do not share the same reference coordinate system, so we come across another obstacle.

Another problem we encountered was that the robot have to stop when it reached the point, it could not continue advancing, since it had already reached the target.

The last problem was to get the robot to have a continuous acceleration, since if not, it takes its highest speed and when it is close to the target it is not able to brake.

III-B. *Solutions*

To solve the problems explained in the previous section, the first thing we do is calculate the angle assuming that the robot is at the origin of coordinates, with the formula that we can see in the following figure, where beta will be our angle.

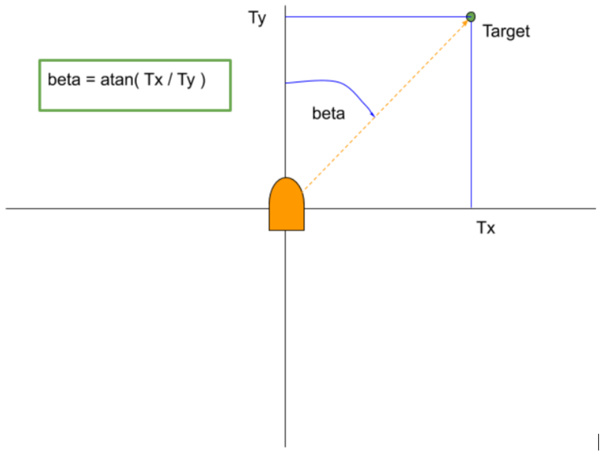

Figura 4.

Obviously, the robot will not always be at the origin of coordinates, so we are going to assume that it is at position (0,1), but without rotation, as we can see in the following image. The solution we can obtain is a geometric vector difference between the current position and the target position.

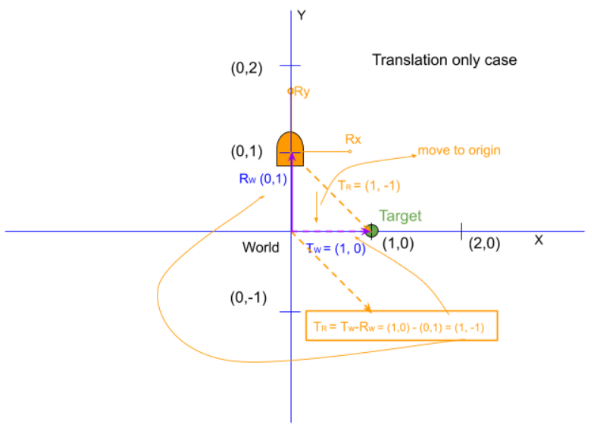

Figura 5.

Now, assuming the robot has rotated, we use the same geometric formula as above, but also adding the transpose matrix. The R is the clockwise rotating matrix built from alpha:

$$cos(a)sin(a) - sin(a)cos(a) \tag{1}$$

$$T_R = R^T(T_W - R_W) \tag{2}$$

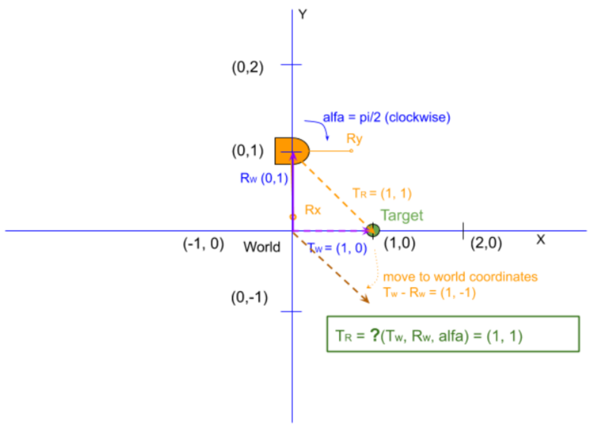

Figura 6.

To build the final controller you have two options:

1. A state machine with TURN and ADVANCE states
2. A simpler direct controller. The idea is to calculate the value of the forward speed and the rotational speed directly from beta and dist, so that they are coupled by mathematical equations.

We made the second option.

Now we need to slow down the robot, so we use the following Gaussian function:

$$y = exp(-x^2/s) \tag{3}$$

This formula corresponds to the reduceSpeedIfTurning() method.

We want the function to be 0.1 when x is 0.5.

With this formula what we achieve is that the speed is reduced when it is going to turn. The other factor to calculate

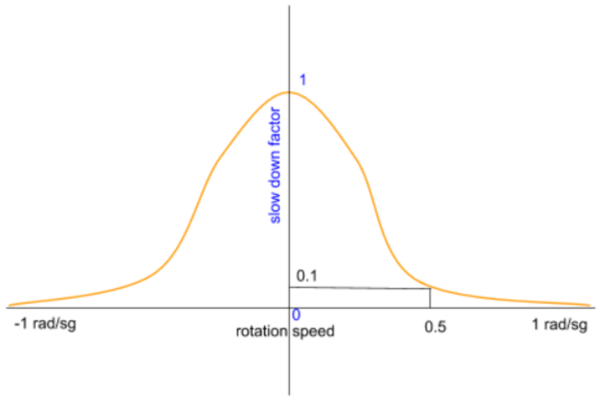

Figura 7.

is a linear function of the distance to the target.

The final speed formula to introduce it to the robot would be the following:

$$\begin{aligned} advspeed = (1000 * reduceSpeedIfTurning \\ *reducesSpeedIfCloseToTarget) \end{aligned} \tag{4}$$

For the reduceSpeedIfCloseToTarget() method, we use the min function.

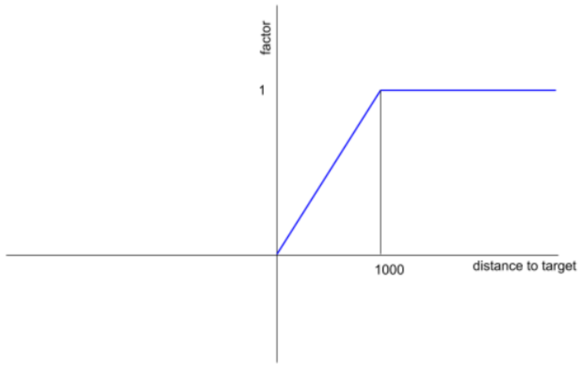

Figura 8.

III-C. Executions

To demonstrate the functionality of this implementation, we have decided to record a demo video.

https://www.youtube.com/watch?v=wU4YChe2Oio

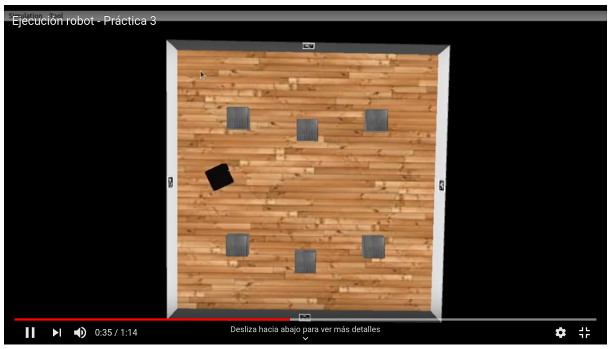
Ejecución robot - Práctica 3
0:35 / 1:14
Desliza hacia abajo para ver más detalles

Figura 9.

IV. FOURTH TASK

IV-A. *Problems*

The problem that we have encountered has been that the robot must avoid the obstacles that stand between it and the click made by the user. Due to this problem, we must find a solution that allows us to avoid these obstacles and finally, allow the robot to reach the selected target.

IV-B. *Solutions*

To solve the problems mentioned above, we have been provided with three possible algorithms:

- Bug Algoritm
- Potential Field Method
- Dynamic Window Approach

In this case we have opted for the third option, Dynamic Window Approach.

First, the laser polygon must be drawn and the possible points that the robot could access. Once all the possible advance routes have been calculated through the different formulas shown below, the most convenient route for the robot will be selected. These routes will be arches of different circumferences. Finally, once one has been chosen, the robot will go through it as shown in figure 10.

To arrive at all the possible solutions that the robot can access, the different speeds of advance and rotation must be calculated for each possible movement, applied to its current state.

The coordinates of the new position P, seen from the map's center are:

$$xc = r * cos(w * dt) \tag{5}$$

$$yc = r * sin(w * dt) \tag{6}$$

where:

$$r = V/W. \tag{7}$$

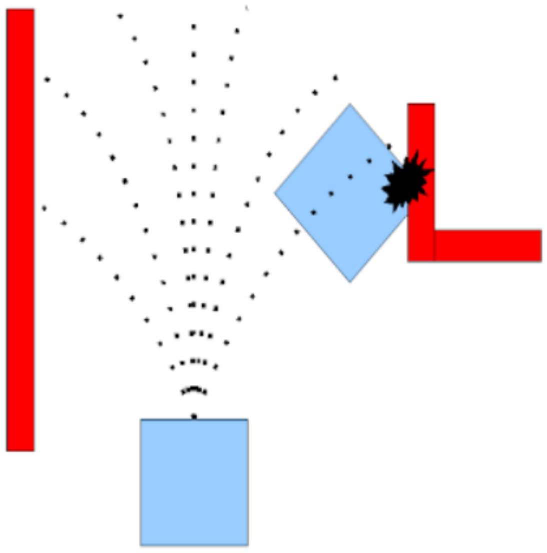

Figura 10.

Viewed from the robot's coordinate system:

$$x = r - r * cos(w * dt) \tag{8}$$

$$y = r * sin(w * dt) \tag{9}$$

and the new robot angle:

$$alpha = w * dt \tag{10}$$

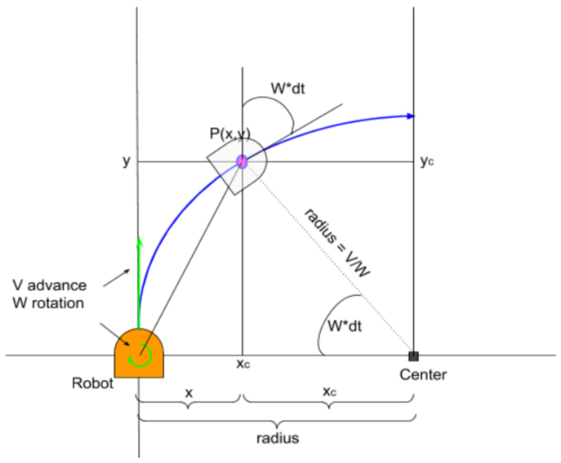

Figura 11.

Once these points have been drawn, a series of steps must be followed:

1. First the set of possible positions must be calculated, to do this the calcularPuntos() method has been implemented. Once these points have been calculated, the obstacles() method has been implemented, which will avoid any object that is in its path. After performing the previous methods, the laser polygon and the point cloud are drawn with the drawThings() method. In this way we can see if the points are being calculated correctly, and we can see if the point cloud is not drawn in the place where the objects are.

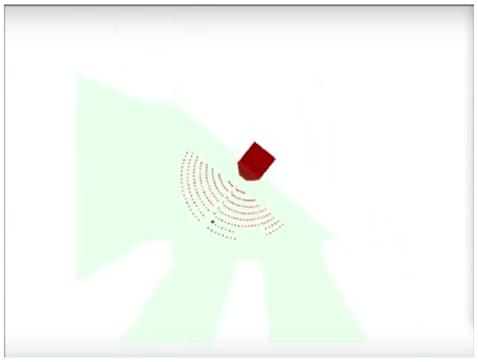

Figura 12.

2. When these positions have been calculated, the optimal point must be chosen from two criteria to reach the target. These criteria are that the point is within the laser polygon and that it is as close as possible to the target. In this case, a method called sort() has been created, which will be in charge of carrying out these tasks.
   The sort () method sorts based on the criteria mentioned above.

3. Once these steps have been carried out, the calculated values must be entered into the robot.

IV-C. *Executions*

To demonstrate the functionality of this implementation, we have decided to record a demo video.

https://www.youtube.com/watch?v=4vO2hAlgJhA

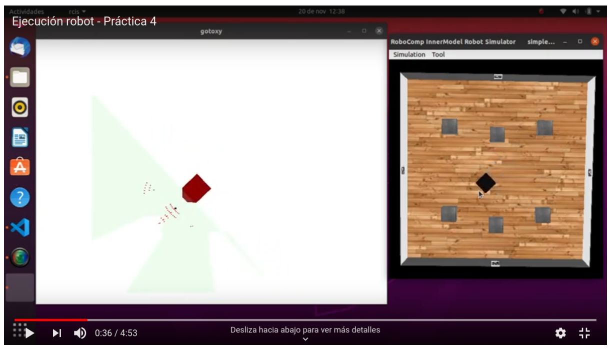
Ejecución robot - Práctica 4
gotoxy
RoboComp InnerModel Robot Simulator
Simulation Tool
0:36 / 4:53
Desliza hacia abajo para ver más detalles

Figura 13.

V. FIFTH TASK

V-A. *Problems*

The first thing we have to do is build a map of boxes, which will have two possible values: free or occupied. The next problem we have, is building a navigation function for the matrix. Finally, once the previous steps are done, we have to connect the previous practice with the current one.

V-B. *Solutions*

To solve these problems, a series of methods have been implemented. The first thing we did was add the grid.h class provided by the teacher. Next, the fillGridWithObstacles() method was implemented, which is used to obtain the coordinates of the center of the boxes (x, y), their width and depth. Afterwards, the obstacles were painted red, and to delimit the map, the border was also painted red. This causes the robot to avoid them.

The next step that has been taken is to implement the navigation function. This corresponds to the computeNavigationFunction() method. This function calls the neighbors() method which is in charge of checking the value of the boxes adjacent to the target and it changes the values of the boxes. This method has been improved by changing the value of the boxes to a big number that are next to the red boxes, to ensure that the robot does not get too close to that area.

In order to connect this practice with the previous one, some changes had to be made in the Value() structure of the grid.h. The new lines that were added were as shown below:

$$QGraphicsRectItem * paintCell = nullptr; \tag{11}$$

$$QGraphicsTextItem * textCell = nullptr; \tag{12}$$

The first change is for setBrush() to work and the box can be changed color. The second change is to be able to add text over the box.

Finally, we had to modify the order method that we had from the previous practice to adapt it to this. First, we create a vector of minima to store a tuple and a float in it. Then we create three variables A, B, C and we begin to traverse the vector given by parameters obtaining the tuples. Once the value of said tuple has been obtained, we have to introduce it into the vector of minima that had been previously created. When introduced, we use the std::minElement function which orders the elements from least to greatest and leaves the minimum as first in the list. Finally, we return it with the std::get function.

V-C. *Executions*

To demonstrate the functionality of this implementation, we have decided to record a demo video.

https://www.youtube.com/watch?v=uqEusiyL5AY

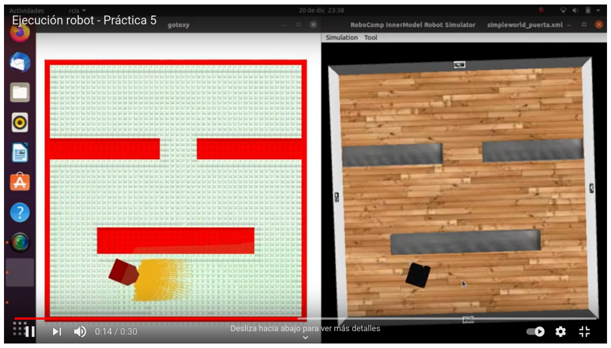
Ejecución robot - Práctica 5
gotoxy
RoboComp InnerModel Robot Simulator
simpleworld_puerta.xml
Simulation Tool
0:14 / 0:30
Desliza hacia abajo para ver más detalles

Figura 14.

VI. Conclusion

In conclusion, we can add that it seems to us a very interesting and enriching subject, but at the same time very complicated and that it requires a lot of different knowledge, such as physics and mathematics, among others. We think that being a simulation it is probably easier than if we had to do it in real life. Despite all that has been said, we are convinced that the acquired knowledge will be of great help in the future.