Robótica 2021-2022

1st José Luis García Salas

2nd Enrique Moreno Ávila

I. Introduction

Nowadays, the research field of robotics is a must-know discipline for all the engineers who are working in technologies like Internet of Things (IoT), automation of tasks, or home automation for people with reduced mobility. The use of robots for making the daily issues easier and natural to make life more simple is becoming so common, that in actual days is strange the house that does not have electronics integrated to work with assistant like Amazon's Alexxa or have the refrigerator or cleaner bot connected to the phones with the purpose of track its activity and route.

In this subject, following the line of work of previous years, we are going to learn how the fundamentals of robotics functions, as well as making our own simple programs to make the simulated robot obeys our commands and instructions with the objective of transfer our own software to a real robot facilitated by the department of RoboLab from the EPCC in the UNEX.

To make this software, we will use a library written in C maintained by RoboLab in EPCC, Aston University and many other collaborators called Robocomp, an open-source framework based in the idea of communicate components through public interfaces.

In this letter, we are going to explain step by step the procedure to make a good software for robot control, as well as the other technologies and tools used in the making.

II. First Class

Here we are going to introduce all the new software tools we use in the design of the robot software as well as we make for the first lesson of revise C/C++ knowledge.

A. Software Tools

We run all the software in a Ubuntu 20.04.3 Focal LTS. Because of the Robocomp library y heavily based on C, we are going to write our code on it, using the GCC/G++ compiler 10.3.0 version because it brings the last dependencies and libraries to work with. Generally, in the Ubuntu version we are using, the GCC/G++ compiler are installed in the 9.3.0 version. We had to update it installing the 10.3.0 version executing **sudo apt-get install gcc-10 g++-10** to install it, and then switching the version of the compiler which the S.O. is using with **sudo update-alternatives –install /usr/bin/gcc gcc /usr/bin/gcc-10 2 0 –slave /usr/bin/g++ g++ /usr/bin/g++-10**.

Thanks to the department of RoboLab and the EPCC

For running and coding in C/C++, we use CLion, an IDE developed and maintained by JetBrains, it is one of the best for this situations due to its capacity of build by itself all the dependencies between the libraries and files, as well as good maintainance with other associated compiling tools like CMake. One good alternative for this is the code editor Visual Studio Code (VSC).

Once we installed all this, we install the first important tool for the interfaces and libraries needed for the robot behaviour, it is called Qt5 software. This is the main toolkit to use the QT libraries related to C/C++, and despite the fact that we are using CLion in first place, QtToolkit have its own IDE called QtCreator and most important, QtDesigner, which is an interface designer for tools like timer counters and lcd displays and the main tools we are going to use for the visual part. As we mentioned, the main library we are going to use is Qt, specifically related to signals and slots, this library fits perfectly with the methodology of components, we can use objects declared as QTWidgets as the future components of the robots, in which each slot is a function triggered by a signal declared previously and connected by the **connect()** method.

For the version control, we use Git, a software of code-version-control which allows us to maintain different workflows and previous status of the code in case we find an error or we make a critical mistake. With Git, we use the repository GitHub, in which we upload from our local repository of Git all the versions of the code for working everywhere in remote control without the need to do it only in classes.

The last tool used before we star to install all Robocomp library is Overleaf, Overleaf is a online text editor which allows us to have the documentation stored in the cloud, and modify the document simultaneously making the parallel documentation possible and we can review and improve our own mistakes or changing things with the security of being both of us at the same time.

Once all of these tools are installed, it is possible start installing all the Robocomp related software in the next classes and the new ZeroC middleware for the communication between interfaces.

B. QTimer and timerSimple

In this lesson we are given 2 different projects, both of them implementing the same thing, a simple timer with a counter of elapsed time and a slider to modify the period for the timer itself.

The first project, called ejemplo1, is the version of the timer implemented using QtTimer library. The main difficult here was to learn the functionality of signals and slots, and how they interconnect, despite this learning proccess, Qt libraries allow to make this very easy and simple to understand.

The second project, called timerSimple, is the same version of the timer, but using the std library of C/C++, the main goal of this project is to review and refresh the knowledge of C/C++ after so long time without use it. This timerSimple version has been heavier and deeper than its other version, we think that the most difficult part is understand the declaration and use of the thread and how is used and synchronized with other threads without hindering each other.

For the visual interface of the timer, there are two ways to modify and built it, we can go directly to the .ui file located in the project, it is a xml file and it can be modified adding manually the new objects like buttons or lcd displays and horizontal slayers. The other way, and it is the more simple way to do it, is open the .ui file with the tool of QtSoftware called QtDesigner, it is a graphic interface and you can drop the object in visual mode, making so much easier the interaction with the timer and interface.

As a conclusion to this practice, we think that the most hard part of it was the use and manage of the std library, probably because the long time we do not use C/C++, despite this, we adapt quickly and it has been an easy review practice.

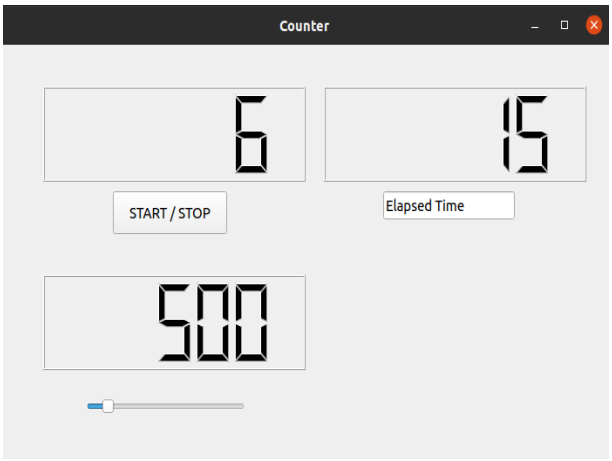

Fig. 1. Graphic representation from the interface of the timer.

III. Second Class

In this section, we are going to explain which software was necessary to install and design components, as well as describe how the behavior of the component was modified and designed with the tools and technologies mentioned in previous section .

A. Robocomp Software

As we said previously, Robocomp software is an open-source framework based in the idea of communicate components through public interfaces, all of them created by members of robotic research community as Robolab in EPCC university or Aston university.

In first stage, we realized that exist many tutorials to learn how to design and modify components such as the movement, the visualization of laser...etc. With this in mind, we followed the tutorials given by the teacher of the subject and presented in virtual campus, this tutorials taught us how to install all **robocomp** libraries and dependencies needed, as well as the functioning of **CoppeliaSim robot simulator**, the simulation environment which we are going to use by the rest of the classes.

The first thing we needed to install is robocomp library, for this purpose we had to access to the following repository on github: **https://github.com/robocomp/robocomp** and when we went to the documentation we saw a brief abstract about robocomp software explaining what it was, how was made and other type of useful information. Next, we study the requirements which were necessaries to install it and run it. The most simple way to do this was using a command existing in the github repository, the command updated all dependencies and all certificates for the S.O. After this command executed, we were be able to put install the rest of robocomp, finishing installing the last packages about Zero Ice middleware and all its libraries.

The next thing we did was install CoppeliaSim and Pyrep. CoppeliaSim is a simulator used to work with robocomp in physically complex scenes, so CoppeliaSim will be used to mimic the behavior of components designed by us in an environment designed to be like our class. Adding to this, we needed Pyrep, which it is toolkit needed to run the CoppeliaSim simluation thread through python. To install CoppeliaSim we used the download and install tutorial in the official web site and then added the following three lines necessary in the general path of our S.O, which in our case is ubuntu:

```
export COPPELIASIM_ROOT=/home/josel/software/CoppeliaSim_Edu_V4_2_0_Ubuntu20_04
#export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$COPPELIASIM_ROOT
export QT_QPA_PLATFORM_PLUGIN_PATH=/usr/lib/x86_64-linux-gnu/qt5/plugins
```

Fig. 2. Image from the path modified to use Coppelia

To have totally ready CoppeliaSim we went to the directory of robocomp and we clone the repository: **"https://github.com/pbustos/beta-robotica-class"**, and in the directories within, we started the middleware with the **rcnode &** command, after this we changed to the directory **"¬/robocomp/components/beta-robotica-class/pyrep/giraff"** and ran **"./run.sh"**, with this we had CoppeliaSim started.

B. Making components.

Once we settle up all the simulation environment, we went through component design and implementation, for this purpose, the teacher gave us 4 different starting tutorials to study and learn how to make it, the most important and in

which we focused was the fourth of them, where explains how to make a component from zero.

In the tutorial, we had been taught to use **robocompdsl**, a template generator tool integrated in robocomp, and one of the most used tools in the repository, it is heavily based in the idea of designing interfaces which can communicate between them to produce the behaviour of the component we wanted.

Once we finished the tutorial, a new component called **Controller** had been made, it was based on C code structure with headers and sources files, we focused in the **specificworker.cpp and .h**, which were the ones in charge of the behaviour.

Having the template complete, we started to follow the hyphen given in the virtual campus. For this week task we needed to make the robot works as a sweeper, formerly known as roomba, the first goal was to make the robot follows the walls of the simulated room and make an ellipse without hitting any of the walls or getting stuck. For this first stage, we modified the variable threshold to minimal, this made the robot stopped at a very close distance from an object, wall or any obstacle. Next we discovered that reducing the angular speed and narrowing down the angle to 20 degrees the robot could turn just the perfect distance to make the ellipse, despite the fact that the robot were not turning smoothly, we achieved the first objective.

When we achieved to make the ellipse path, we checked the next goal, which was make the robot be ”competitive” sweeping around 50 percent of the room in only 3 minutes of time limit. To measure this, we used a component included in the repository given to us called **aspirador**, which present to us another window with a screen showing the robot and the path travelled, and the time elapsed and percentage made.

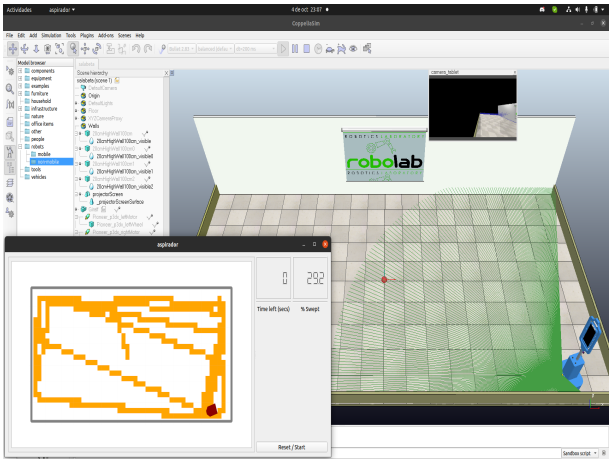

Fig. 3. Image presenting the results of the first stage from the algorithm in ellipse path

After making a lot of runs, we only achieved an average result of 25 percentage of surface swept, with this in mind, we started to experiment with the code until we arrived to this:

We realised that with this type of path, we barely achieved a half of the desired score, then we thought to make random turns in a range of degrees. To make this, we studied more carefully the code, and stated that using a flag, we could determinate in which direction the robot can turn, using the maximum rotation speed, which was 2 Rad/second and multiplying it for the value of the flag. Once we decided in which sense the robot turned, we used the **usleep** method to make the robot turn in the range we wanted (in our case, between 45 and 20 degrees). In Addiction to this, we made a very rudimentary linear velocity control, using the threshold as a limit, whenever more far away was the laser robot from an obstacle, we increased the linear speed. The resultant and final code was this:

Fig. 4. Final image from the practice 2 of the robot

Testing the code, we observed that we were close to the target of a 50 percentage of swept floor in 3 minutes, but we could not afford it, the score always was in range between 38 and 41 percentage. Seeing this, in the final class, the professor made us a final observation that helped us a lot.

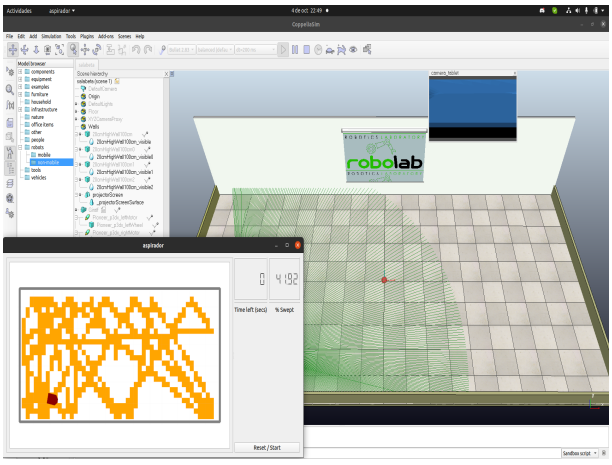

Fig. 5. Image from the random turns path in the second attempt

As we arrived to the class, the professor suggested us to

modify one variable from the component **aspirador**, which was the surface wide swept from the robot, measuring almost the double from the original one.

Testing this with our new random path model, we achieved even a better range that the one who asked to us. In 3 minutes the range of values were from 68 percentage to 71 percentage of swept surface.

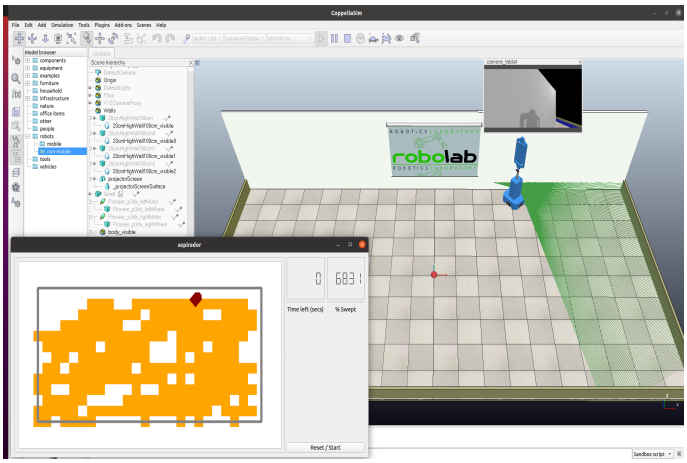

Fig. 6. Final path with component **aspirador** modified

IV. Third Class

Once we learned how to make components and how to interact with them, our professor went directly to the practice 3, which consisted on making the robot move to a point selected by us by clicking with the mouse on the grid.

For this task, we needed to organize step by step, allowing us to set 2 different phases in this practice:

A. Engaging the Grid

The first thing we needed was the graphical grid, which was the screen that represents the room simulated by **CoppeliaSim** and in which we were going to point with the mouse for the target site.

```
void SpecificWorker::new_target_slot(QPointF point) {
    target_point = point;
    target.active = true;
    qInfo() << target_point;

}
```

Fig. 7. Basic code to explain how to assignate a new target point

In this simple 3 lines, we assigned a new **QPointF** type variable to our own one, making the flag of new target true and showing in the screen using **qInfo()** method.

When we got covered and ensured that by clicking in point we send the correct coordinates to the robot, we started to make the laser of the robot. From now on, we will represent the laser as a polygon, to make this form we make the following code, which was completed by the teacher in class solving some of our problems:

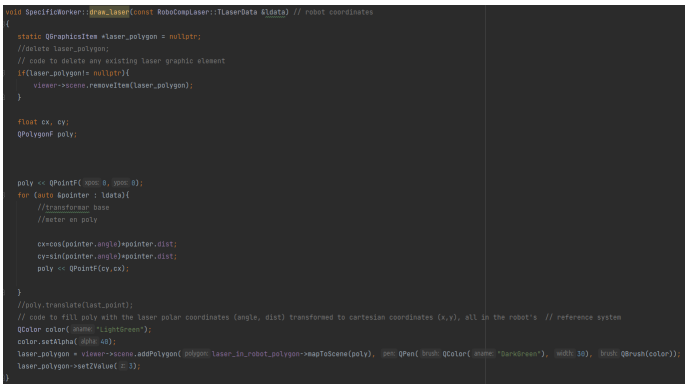

Fig. 8. Code presented in class to draw the laser

We used 2 types of objects in Qt libraries, **QPolygon**, **QPointF** and **QGraphicsItem**. To declare the laser, we created a **QGraphicsItem** variable called **laser_polygon**, in which we overturned all the information from the laser itself to the graphic environment. After we reset this variable to make sure we updated the grid, we create the **QPolygonF** object called **poly**, and we assigned to them a **QPointF** values to set the reference point from the robot. After this, we just transferred all the info from **poly** to **laser_polygon**. °

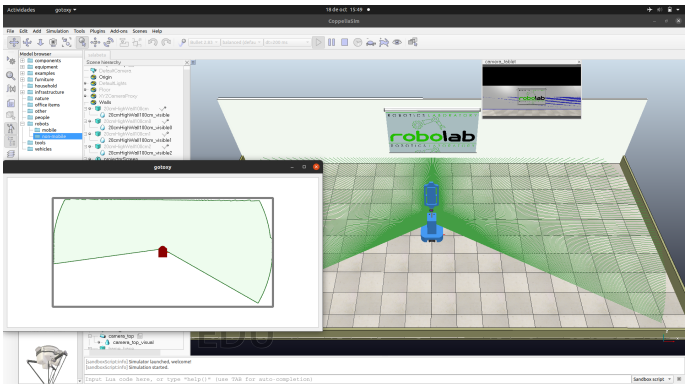

Fig. 9. Image for present how the grid was presented

B. Robot behaviour

Now that we had the environment ready to run, we focused on the robot, and how we could go to make the practice. In the shared driver given by the professor, explained that we need to separate 3 different functions: 1)Linear Speed and brake control, 2)rotation speed and 3)transfer all the information to the reference system of the robot.

For the linear speed and break control, we set a threshold to go max speed, defined by the variable in header file called **MAX_ADV_SPEED** which had a value of 1000.

When the robot cross the threshold, we made a simple calculation, reducing the max speed by a factor of the threshold

and multiplying it by the remaining distance. At the end, we returned the speed itself divided by max speed.

This would make the robot reducing the speed as long as it get closer to an obstacle.

```
float SpecificWorker::dist_to_target(float dist){
    float threshold = 500;
    float speed;
    if(dist >= threshold){
        speed = MAX_ADV_SPEED;
    }else{
        speed = dist * (MAX_ADV_SPEED/threshold);
    }
    return (speed / MAX_ADV_SPEED);
}
```

Fig. 10. Code of dist to target to calculate linear speed and brake

Once the problem with the linear speed and brake control was done, we settled up the rotation speed, in which we had to apply the next sigmoid function in order to calculate it correctly:

$$F(angle) = e^{x^2/\lambda} \tag{1}$$

Being e the constant mathematical value, x the angle calculated previously in **compute** method. For the λ, it was calculated apart, the professor explained us how to do it, but we use the data given in the shared drive, due to we realised that with this data, the gaussian distribution is more accurate and defined.

```
float SpecificWorker::rotation_speed(float beta) {
    float v= 0.0;
    float e = 2.71828;
    float lambda= 0.10857362;

    v = pow(e, y: -pow(beta, y: 2)/lambda);

return v;
}
```

Fig. 11. Code for the implementation and calculation for the rotation speed

For the third method, we had to transfer all the calculated information to the robot reference system, by doing this, we can move the robot wherever we wanted. This method, called **world_to_robot**, allowed us to change all the information calculated in the grid reference system to the robot reference system.

The main achievement here was to calculate the base switch from the grid reference system to the robot reference system, for this, we create a matrix of **Matrix2f** type from **Eigen** library and a **Vector2f** type from the same type to store the point from grid system, and apply the base switch returning from the function a type **QPointF** variable with the coordinates calculated before.

```
QPointF SpecificWorker::world_to_robot(Target target, RoboCompGenericBase::TBaseState bState)
{
    float angle = bState.alpha;
    Eigen::Vector2f T(bState.x, bState.z), point_in_world(target.destiny.x(), target.destiny.y());
    Eigen::Matrix2f R;

    R << cos(angle), -sin(angle), sin(angle), cos(angle);

    Eigen::Vector2f point_in_robot = R.transpose() * (point_in_world - T);
    return QPointF(point_in_robot.x(), point_in_robot.y());

}
```

Fig. 12. Code for the world to robot method

In the last stage, with everything calculated, we went to modify the compute method, adding the key parts to complete the behaviour.

```
if(target.active){
    Eigen::Vector2f robot_eigen(bState.x, bState.z);
    Eigen::Vector2f target_eigen( x: target.destiny.x(),  y: target.destiny.y());
    if(float distance = (target_eigen - robot_eigen).norm(); distance > 100){
        QPointF pt = world_to_robot( target: target, bState);

        float beta = atan2(pt.x(), pt.y());


        float adv = MAX_ADV_SPEED * dist_to_target(distance)* rotation_speed(beta);
        try {
            differentialrobot_proxy->setSpeedBase(adv,beta);
        }catch(const Ice::Exception &e){
            std::cout<<e.what()<<std::endl;
        }
    }
    else{
        try{
            differentialrobot_proxy->setSpeedBase( adv: 0, rot: 0);
            target.active = false;
        }
        catch(const Ice::Exception &e){
            std::cout<<e.what()<< std::endl;

        }
    }
}
```

Fig. 13. Key parts in code added to make Gotoxy works

As we present in the image, here we first save the actual coordinates of the robot and the target in type **Vector2f** from **Eigen** structures, we also calculated the resulting distance until arriving the point. After this we calculate the angle called **beta** and calculate the needed speed with the product of the results from **dist_to_target** method and **rotation_speed** method.

Concluding this practice, we think that the most difficult part of all was learning how to interact with mathematics again, despite the fact that there has been a lot of time since we did not use it in this way, once we saw how to correctly apply the formula and how implement it in C++ make the things so much easy.

Other aspect to have in mind was the fact that we got a lot of documentation in the Qt API, which was a huge source of information to our dispose.

V. Fourth Class

Once we have made Gotoxy's component and it has a correct behavior, our professor decided to go to the practice 4, where the behavior of Gotoxy will be different. Now the robot had to go to a location appointed with the mouse on the grid and had to avoid all the obstacles in it encountered.

For this purpose, we had to implement the BUG algorithm described to us by the professor in class. With this algorithm, the robot was supposed to go from its position, to the target point, surrounding any obstacle it encountered.

A. BUG Algorithm

In this algorithm the robot had to avoid an obstacle surrounding it following the next premise: When the robot encounters an obstacle, it will turn to the right and then it will go next to the outline of the obstacle. To return the path to the target, two conditions had to be fulfilled: First, we have to check if it was in the correct way to the target and second, the target must be visible.

*1) **Settling State Machine**:* Starting from the code of Gotoxy's component from **practice 3**, we stated that the best simple option here was define 4 different states for the robot: **IDLE**, **RUN OBSTACLE** and **SURROUND**. With this states, the robot had to have a range of behaviours from being waiting for a new target point, to surround the borders of the obstacles it encountered. Now we are going to explain all states can have the robot during the execution of this practice and how we implemented them in the code.

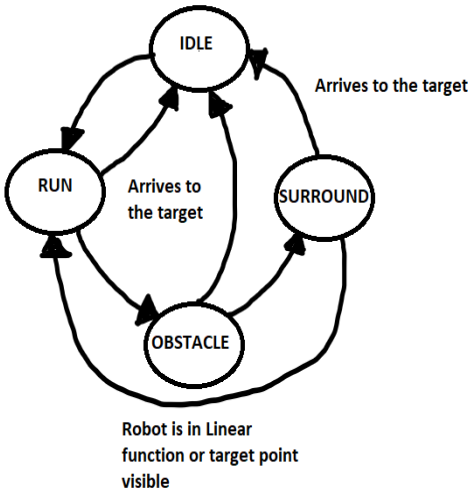

Fig. 14. Graphic representation of the state machine of the robot

For the code, we used a switch statement based on the enum class State called **estado**, in **estado** we defined the 4 states mentioned above, as well as allowed us to iterate in order of the enum type. In each of the cases of the switch, we showed in console which state we were in.

```
switch (estado) {

    //ESTADO EN ESPERA
    case State::IDLE:
        std::cout<<"estoy en estado IDLE"<<std::endl;
        try {
            differentialrobot_proxy->setSpeedBase(0,0);
        }catch (const Ice::Exception &e){
            std::cout<<e.what()<<std::endl;
        }
        if(target.active)
        {
            function.robot = QPointF(bState.x, bState.z);
            estado = State::RUN;

        }
            /////ALMACENAMOS LA POSICION ACTUAL DEL ROBOT AL ACTUALIZAR EL PUNTO////
        break;
        //EL IR PARA ADELANTE, EL CLASICO
    case State::RUN:
        std::cout<<"estoy en estado RUN"<<std::endl;
        estado = run(bState,  ldata);
        break;
        //REORIENTAR Y GIRAR EN LA PARED
    case State::OBSTACLE:
        std::cout<<"estoy en estado OBSTACLE"<<std::endl;
        estado = obstacle(bState, ldata);
        break;
        //UNA VEZ GIRADO, TIRAR ADELANTE HACIA EL PUNTO
    case State::SURROUND:
        std::cout<<"estoy en estado SURROUND"<<std::endl;
        estado = surround(bState, ldata);
        break;
}
```

Fig. 15. Figure about states can have the robot.

*2) **IDLE State**:* The first and basic state of the robot was **IDLE**, in this state, the speed of the robot was 0, and it stayed waiting until we gave it a target point by clicking into the grid. Once the component has a new target, the state changes to **RUN**. To implement this behavior we only set the speed base of the robot to zero, and when a new target point was established. We communicated to the robot the target point by **linear_function.robot = QPointF(bState.x, bState.z);**, accessing to the robot library and setting its position by a **QPointF** object used in previous classes.

```
SpecificWorker::State SpecificWorker::rvun(const RoboCompGenericBase::TBaseState &bState,  const RoboCompLaser::TLaserData  &ldata){
    float distance = 0.0;
    int semiwidth = 60;
    Eigen::Vector2f robot_eigen(bState.x, bState.z);
    Eigen::Vector2f target_eigen( x: target.destiny.x(),  y: target.destiny.y());

    ///////CONDITION TARGET EXIT///////
    if( distance = (target_eigen - robot_eigen).norm(); distance <= 100){
        target.active=false;
        return State::IDLE;
    }
    ///////CONDITION CRASH OBSTACLE///////
    if( distance_ahead(ldata,  dst: 300, semiwidth))
    {
        try
        {
            differentialrobot_proxy->setSpeedBase( adv: 0, rot: 0);
        } catch (const Ice::Exception &e)
        {
            std::cout<<e.what()<<std::endl;
        }
        return State::OBSTACLE;
    }

    ////////////////////////////////////////////////

    //OJO A ESTO, AQUÍ LOS EJES A ATAN2 SE LOS PASA CAMBIADOS, PRIMERO EJE Y, LUEGO EJE X
    float beta = atan2( y: target_to_robot.x(),  x: target_to_robot.y());
    float adv = MAX_ADV_SPEED * dist_to_target_object(distance)* dist_to_target_object(get_min_ldata_element(ldata,semiwidth).dist) *rotation_speed(beta);

    try
    {
        differentialrobot_proxy->setSpeedBase(adv,beta);
    } catch (const Ice::Exception &e)
    {
        std::cout<<e.what()<<std::endl;
    }
    return State::RUN;
```

Fig. 16. Image from the code part of RUN state

*3) **RUN State**:* Once we got the target point settled, the robot had to change to the **RUN** state from **estado**. Basically, the system drew a straight line between the robot and target point, this line or path was defined by a linear function represented in the code by **linear_function** structure. However, the robot could find an obstacle when was trying to go to the target point, this situation must be fixed in two different parts: First, when the robot encountered and obstacle, it had to

turn until its frontal part was free, this problem is represented and fixed in the state **OBSTACLE**. Once we arranged with the turning situation, the robot had to been capable of surround the obstacle to arrive to the target, this final situation is represented by state **SURROUND**. The other option for the robot was to arrive to the target point, in which case turns to the state **IDLE** again waiting for a new target.

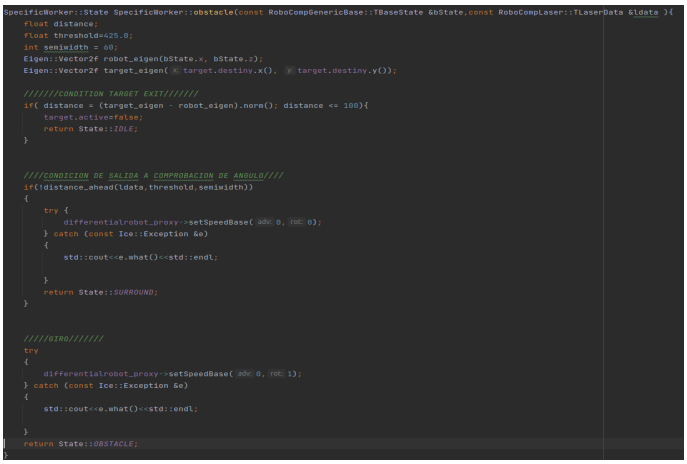

Fig. 17. Caption

*4) **OBSTACLE State**:* When the robot reached this situation, it had to turn until its front part is free and ready to proceed its pathing.

The first thing we have to implement into this state is the testing that the robot has reached the target point as we made in every other state, When the robot arrives the target point we will set the variable **target.active** to false to indicate there isn't target point and we set the robot to **IDLE** state.

To check if the robot had to pass to **SURROUND** state we have made a lambda function called **distance_ahead** that tells us if it is in the minimal distance between the robot and the object with the minimum element of **ldata** in a concrete range of a fraction of the laser represented by a **semiwidth** variable to focus in a specific angle, then we compare it with a **threshold** against of the minimum element to check if it will be higher than the **threshold**, if this happens we set the **speed base** to zero and the robot pass to **SURROUND** state.

Last, we implement the most important thing of this state, which is make the robot turn to right. To do this just have to modify the part of the angle giving a rotation speed of 1 rad/sec.

With this conditions, we defined the exit cases of the robot from this state, allowing to identify when it reached to the target point and when it had to turn and went to the **SURROUND** state.

*5) **SURROUND State**:* A simple idea we can think to the last state is basically go over the object until the robot finds the target point or the robot is in line with the line equation. Now there three options, in one of them while the robot is surrounding the obstacle, the robot finds the target point, in other option the robot changes to **RUN** state because he finds where can be the target point or he is in line with the line equation and in the last option the robot continues in the **SURROUND** state while others options aren't happening.

The first thing we are going to implement is to verify if the robot arrives to the target point, so we create two **Vector2f** to save the position of the robot and the position of the target point in addition to calculate the distance between them as a purpose of check if it is less than thousand. In the case before we will set **target.active** to false to indicate the robot has reached the target point and the robot will change to **IDLE** state.

Other thing we have to implement is to check if the robot has found the target point or he is in line with the line equation, so we check if the robot has found the target point with the function **poly.containsPoint** which requires the target point **target_to_robot** and it also requires a fill rule, which we use **OddEvenFill**. The second condition we used a function called **robotInFunction** that returns "true" if the distance between the **robot_function** and the **target_eigen** is higher than the distance between **robot_eigen** and **target_eigen**, as a aclaration all of them are variables created as **Vector2f**. If the case before is "true" the variable function **function.robot**, used as linear function of the robot, is set the point where is the robot in this moment, and the function **robotInFunction** returns "true". The function **robotInFunction** returns false when the distance calculated in the equation given by the professor gives more than ten or the case before it is not fulfilled. In conclusion, when one of the two functions, **poly.containsPoint** or **robotInFunction**, it is fulfilled the robot changes to **RUN** state.

The last thing to do in this state if the robot hasn't changed to any state is surround the obstacle. First, we calculated the speed which will be the fourth part of the maximun settled down in **MAX_ADV_SPEED** and save it in **adv**. Second, we calculated the angle we needed to rotate, so we saved in **sideConeangle** the minimun angle of the range between **ldata** and **ldata.size()/2** with the method **get_min_ldata_element**, now we have calculated the angle range we needed to rotate in function of **sideConeangle**, so if **sideConeangle** is less than zero the variable **coneBegin** is thirty and the variable textbfconeEnd is the sum of ldata.size()/2 plus thirty, if **sideConeangle** is higher than zero the assignment of the variable will be the other way around. Once we have calculated the angle range form by **coneBegin** and **coneEnd**, we have to correct the direction of the robot in the way of he doesn't award of the obstacle but he doesn't hit the obstacle, to do this we calculate the sign math we can assign to **beta**, for this we have used a similar method to **distance_ahead** called **distance_side** where we compare two hundred to the minimum element of the range angle by the way of see if the robot is going to hit the obstacle when the function returns "true", so **beta** is assigned a negative sign with the division of **-sideConeangle** with **abs(sideConeangle)**, and in the other case we compared three hundred to the angle range the way of the function return "false" the robot is going to award of the obstacle, so we assign to **beta** the same sign of **sideConeangle**

with the division of **sideConeangle** with **abs(sideConeangle)**. Last, we shown **sideConeangle** to see if it is correct and we setted the speed base with the speed **adv** and the angle form with **beta** multiplied with 0.5 to have a correct speed rotation.

```
// std::cout<<ldata.size()<<endl;
////ROBOT IS IN LINEAR FUNCTION OR IS IN POLYGON////
if (poly.containsPoint(target_to_robot, FillRule: Qt::OddEvenFill))
{
    return State::RUN;
}
```

Fig. 18. First part of the surround

```
/.../  int coneBegin;
int coneEnd;
////////LEFT////////
if(sideConeangle < 0)
{
    coneBegin = 30;
    coneEnd = ldata.size()/2 +30;
    std::cout<< "LADO IZQUIERDO"<<std::endl;
}else
    ////////RIGHT////////
{
    coneBegin = ldata.size()/2 +30;
    coneEnd = 30;
    std::cout<< "LADO DERECHO"<<std::endl;
}
```

Fig. 19. Second part of the surround

6) Sensitive Methods: As a conclusion to this practice, we wanted to highlight the use of lambda functions in this project. Lambda functions allowed us to make new type of methods which made for us great operations and clean our code in a massive way, here we used 3 diferent methods: First, **distance_ahead**, this function returns us a boolean which told us that, given the data and values of the laser represented by **ldata**, a distance as a threshold and a range of that **ldata** represented by **semiwidth**, if the robot was a minimum distance or not from an object and when to stop turning and went to **SURROUND**state. Secondly, we got **get_min_ldata_element**, heavily based on the previous one, this gave us directly the minimal element from the **ldata**, which will allowed us to calculate the distance of rotation in the corners of the object while we were surrounding an object. Last but not least, we got **distance_side**, like **distance_ahead**, this interacted with **get_min_ldata_element** to give us where we had to stop turning while we are surrounding a corner in the **SURROUND** state.

```
bool SpecificWorker::distance_ahead(const RoboCompLaser::TLaserData &ldata, float dist, int semiwidth)
{
    size_t s = ldata.size();
    auto min = std::min_element(ldata.begin() + (s/2 - semiwidth), ldata.end() - (s/2 - semiwidth), [](auto a, auto b){return a.dist < b.dist;});
    return (*min).dist <= dist;
}

bool SpecificWorker::distance_side(const RoboCompLaser::TLaserData &ldata, float dist, int iterBegin, int iterEnd)
{

    auto min = std::min_element(ldata.begin() + (iterBegin), ldata.end() - (iterEnd), [](auto a, auto b){return a.dist < b.dist;});
    return (*min).dist <= dist;
}
RoboCompLaser::TData SpecificWorker::get_min_ldata_element(const RoboCompLaser::TLaserData &ldata, int semiwidth)
{
    size_t s = ldata.size();
    auto minimal_element = std::min_element(ldata.begin() + (s/2 - semiwidth), ldata.end() - (s/2 - semiwidth), [](auto a, auto b){return a.dist < b.dist;});
    return (*minimal_element);
}
```

Fig. 20. Sensitive Methods in code

B. Final Conclusions

This practice was so much harder than the previous ones, the main problem here was that we are not used to C++ functions and libraries, as well as the algebraic use of linear functions, while we understood the theory explanation, making it to the code and implement it was another different thing. Despite all this, we enjoyed so much it and the only thing we regret is maybe the time we got to work in it.