Autonomous Robots
University of Zaragoza

Laboratory 3: Simulation and control of a quadrotor using Gazebo

Alberto Zafra Navarro & Daniel Sanz Valtueña

This laboratory session is composed of several tasks centered around the control, modeling and simulation of a quadrotor (Hector quadrotor) within the ROS framework, utilizing Gazebo, a 3D dynamic simulator. This report will cover the different proposed exercises related to the previously mentioned tasks.

Exercise 1

In exercise 1 it was required to identify the mass of the quadrotor, where the sonar sensor is described, the topic where ROS publishes the measurements of this sonar sensor; and, finally, the type of data published by this topic and the values published by the sensor when the quadrotor is standing on the ground.

In order to obtain the previously mentioned information we went through the different URDF files contained in the `hector_quadrotor` package, more specifically under the `hector_quadrotor_description/urdf` directory.

From the URDF model it was possible to obtain the following information:

- **Quadrotor mass** = 1.477 kg. (found in line 12 of quadrotor_base.urdf.xacro).
- **The sonar sensor information** (also described in quadrotor_base.urdf.xacro file, in lines 32-35):

```
32    <!-- Sonar height sensor -->
33    <xacro:sonar_sensor name="sonar" parent="base_link" ros_topic="sonar_height"
     update_rate="10" min_range="0.03" max_range="3.0"  field_of_view="\${40*pi/180}"
     ray_count="3">
34      <origin xyz="-0.16 0.0 -0.012" rpy="0 \${90*pi/180} 0"/>
35    </xacro:sonar_sensor>
```

  Within the previous lines of code, it is described that the package publishes the measurements of the sensor in the **sonar_height** topic.

Once the simulation is started, it is possible to obtain the type of data published by executing the command "`rostopic type /sonar_height`", which returns that the type of the data is "**`sensor_msgs/Range`**". On the other hand, it is possible to evaluate the values published in the topic by executing the command "`rostopic echo /sonar_height`". An example of the values obtained when the quadrotor is standing on the ground is shown below:

```
header:
  seq: 365
  stamp:
    secs: 36
    nsecs: 800000000
  frame_id: "sonar_link"
radiation_type: 0
field_of_view: 0.6981319785118103
min_range: 0.029999999329447746
max_range: 3.0
range: 0.16396480798721313
```

In the previous example it is possible to distinguish five different sections of the published message.

- **Header**: this section contains the timestamp of the published message.
- **Radiation_type**: this section describes the type of radiation used by the sensor, determining if it is a sound sonar, an IR sensor,...
- **Field_of_view**: this section describes the size of the arc that the sensor is reading, measured in radians.
- **Min_range** and **max_range**: these sections describe the minimum and maximum distance where the sensor can detect anything, measured in meters.
- **Range**: finally, this section determines the distance to the detected object, in meters. It is worth noting, that the values of the range data should be within the min_range and the max_range. A remarkable fact is that the sonar has a defined offset, in the URDF file, of 0.16 m above the floor (-0.16 m in the reference system placed in the center of the quadrotor). Therefore, when the drone is placed in the floor, the sensor is detecting an obstacle/object (the floor) at that distance.

Exercise 2

In exercise 2 it was required to write the default values of the following PID controllers: “twist controller linear.x”, “twist controller, angular.z” and “pose controller yaw”. These default values were found in the ”controller.yaml” file, which is situated in the directory `hector_quadrotor_controller/params` and are the following:

- **twist controller linear.x** = `k_p:5.0; k_i:1.0; k_d:0.0; limit_output:10.0; time_constant:0.05`
- **twist controller angular.z** = `k_p:10.0; k_i:5.0; k_d:5.0; time_constant:0.01`
- **pose controller yaw** = `k_p:2.0; k_i:0.0; k_d:0.0; limit_output:1.0`

Exercise 3

In exercise 3 it was required to find, in the `twist_controller.cpp` file, the name of the variables that store the desired and current linear.y velocities. After analyzing the code, the variables that contain the current and desired velocities are shown in lines 185 and 186 as follows:

```
185 Twist command = command_.twist; // Command == desired
186 Twist twist = twist_->twist();  // twist == actual
```

Therefore, for accessing to the current linear.y velocity the user should access to the `twist.linear.y` variable, whereas, for accessing to the desired linear.y velocity the user should access to the `command.linear.y` variable.

Additionally, in exercise 3 it was required to copy the lines of code that implement the linear.z controller, this implementation is shown in line 241 of the aforementioned program, and is structured as follows:

```
241     acceleration_command.z = pid_.linear.z.update(command.linear.z, twist.linear.z,
    acceleration_->acceleration().z, period) + gravity;
```

Exercise 4

In exercise 4 it was required to copy the lines of code, in the `motor_controller.cpp` file, that rotate the quadrotor in z. After searching within the program, it was determined that the following lines (137-142) are the ones which rotate the quadrotor in z. They specify the input signal to each motor (in voltage) which consist of the force and torque that each motor has to apply in order to advance (force) and rotate (torque) in the z direction. It is worth noting that both, torque and force, are divided by a scale factor which is used to obtain the proportional voltage required in order to obtain the desired force/torque.

Furthermore, in lines 144-146 the drone updates the current torque by inverting the previous scale operation for latter applying the closed loop control in the controller.

```
137     double nominal_torque_per_motor = wrench_.wrench.torque.z / 4.0;
138     motor_.voltage[0] = motor_.force[0] / parameters_.force_per_voltage +
       nominal_torque_per_motor / parameters_.torque_per_voltage;
139     motor_.voltage[1] = motor_.force[1] / parameters_.force_per_voltage -
       nominal_torque_per_motor / parameters_.torque_per_voltage;
140     motor_.voltage[2] = motor_.force[2] / parameters_.force_per_voltage +
       nominal_torque_per_motor / parameters_.torque_per_voltage;
141     motor_.voltage[3] = motor_.force[3] / parameters_.force_per_voltage -
       nominal_torque_per_motor / parameters_.torque_per_voltage;
142
143     motor_.torque[0] = motor_.voltage[0] * parameters_.torque_per_voltage;
144     motor_.torque[1] = motor_.voltage[1] * parameters_.torque_per_voltage;
145     motor_.torque[2] = motor_.voltage[2] * parameters_.torque_per_voltage;
146     motor_.torque[3] = motor_.voltage[3] * parameters_.torque_per_voltage;
```

Exercise 5

In exercise 5 it was required to write the command that publishes a linear velocity at z of 0.15 m/s in the "`cmd_vel`" topic and to check it out visually on the simulator. As it can be seen in figure 1, this publication was performed from the terminal by executing the commad "`rostopic pub /cmd_vel geometry_msg/Twist ...`". However, after publishing the message the drone started a displacement in the positive z direction (upwards) and it did not stop even after terminating the execution of the command. This is because the published message was a velocity publication, thus the drone will be applying that velocity to its movement until another velocity is received by the controller.

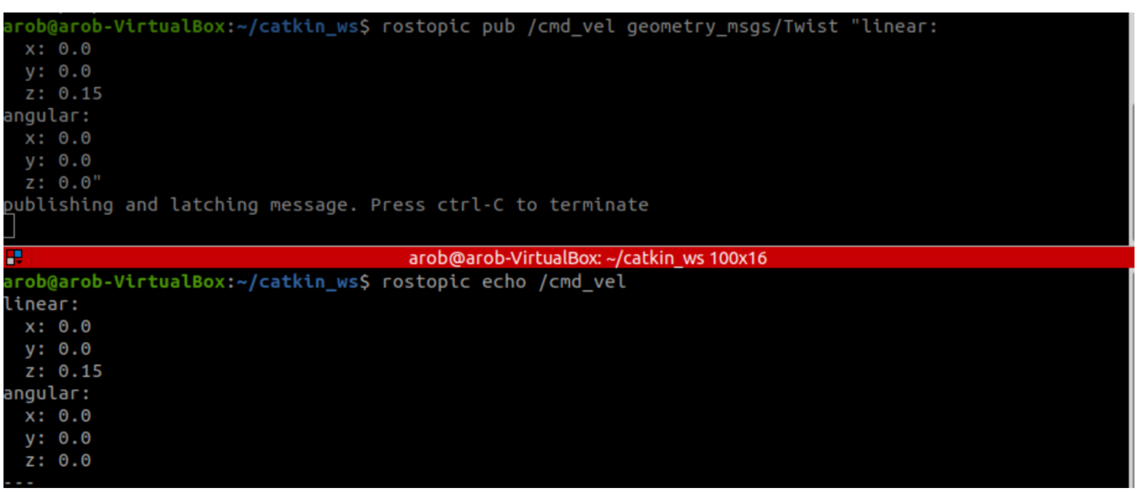


Figure 1: rostopic pub and rostopic echo on /cmd_vel

Exercise 6

In exercise 6 it was required to modify the `controller.launch` file in order to perform a pose controller, by adding the ”`controller/pose`” definition. Once this was done, it was required to test the drone behaviour by publishing on the “`command/pose`” topic a message to make the quadrotor go up 2 meters.

For this exercise, the `controller.launch` file was modified as requested by adding to line 4, more specifically to the ”`args`” section the ”`controller/pose`” definition.

```
1 <launch>
2   <rosparam file="$(find hector_quadrotor_controller)/params/controller.yaml" />
3 
4   <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
      output="screen" args="  controller/twist controller/pose --shutdown-timeout 3"/>
5 
6 </launch>
```

Therefore, after executing the command ”`rostopic pub /command/pose geometry_msg/Twist ...`” the drone moved upwards until reaching a distance of 2 m above the floor as it is shown in figure 2.

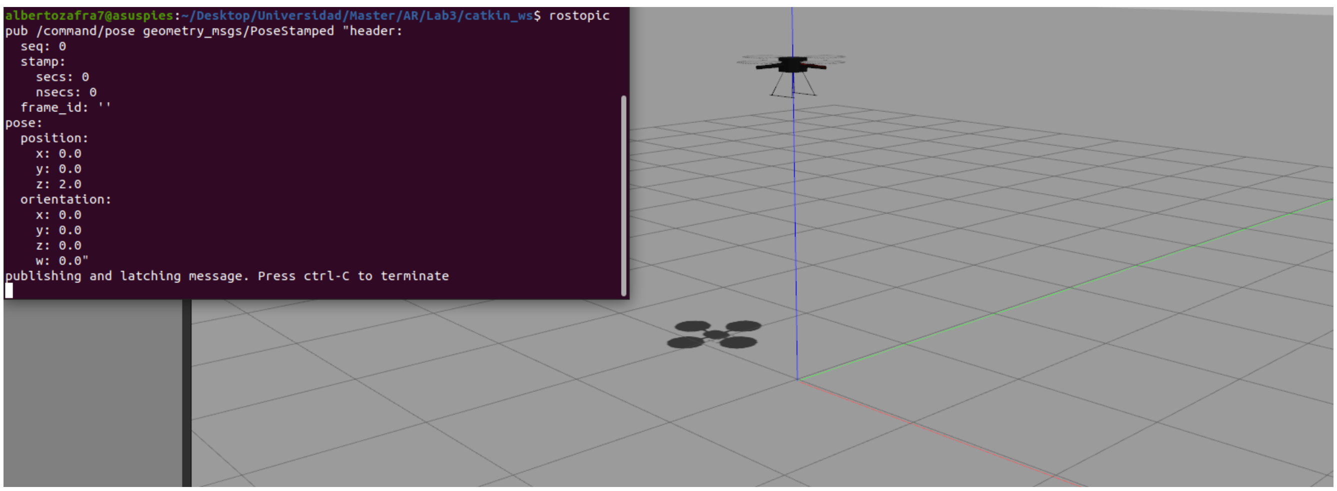


Figure 2: rostopic pub and drone behaviour on /command_pose

Exercise 7

In exercise 7 it was required to create a new package, `p12_arob_lab3`, with a node called `followTargets3D` that reads from a text file the list of 3D targets (X,Y,Z) and publish them one by one to the topic `/command/pose`. The new goal is published when the robot is “sufficiently close” to the previous target.

The previous package `p12_arob_lab2`, which had a simillar node called `followTargets`, was used as a starting point, where the only needed modifications were the addition of a new component to each of the poses, the addition of an id file evaluation, that evaluates the parameter given when launching the node and reads the desired trajectory file depending on this parameter; and the modification of the publisher and subscriber nodes, in order to publish the messages the ”`/command/pose`” topic was used, whereas, in order to receive the pose information of the quadrotor, the ”`/ground_truth/state`” topic was used.

A remarkable part of the code is the creation of the ”`isMoving()`” and ”`ReloadGoal()`” methods, that were initially implemented in order to solve a problem that caused the drone not moving to the initial target, as the controller of the quadrotor did not receive the message published within the `followTargets3D`’s constructor.

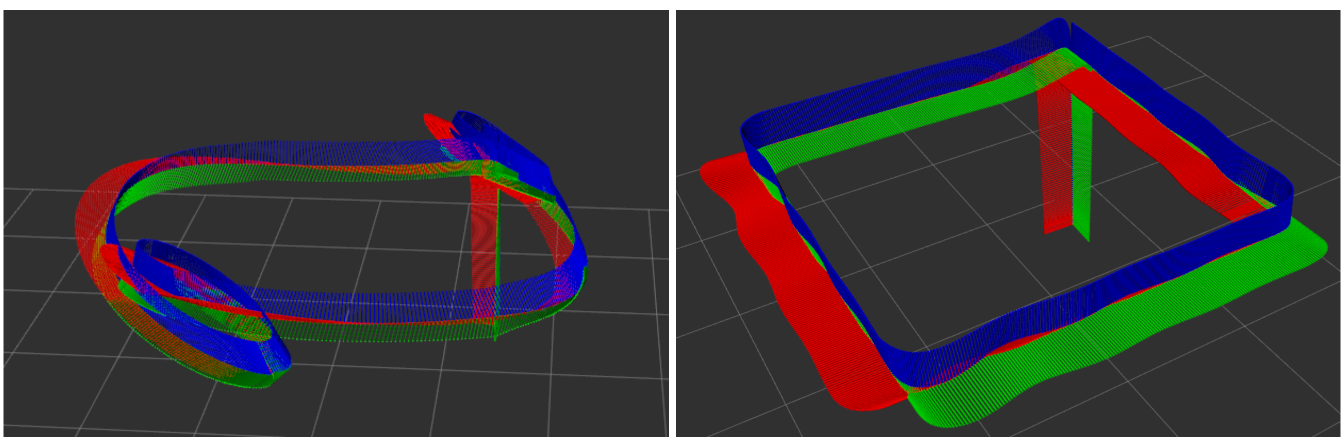
Nonetheless, its use was deprecated due to the modification of the ”`latch`” parameter, which was set as true, in the publisher’s creation. What this ”`latch`” parameter does is that when a connection is latched, the last message published is saved and automatically sent to any future subscribers that connect. Avoiding possible errors of the goal publication.

The developed code of `followTargets3D` is contained in the Appendix.

Exercise 8

In exercise 8 it was required to visualize the trajectory followed by the quadrotor for the different lists of targets provided in the lab, by using `rviz` with the configuration `arob_lab3.rviz`, also provided in the lab. Furthermore, an explanation for the varying behaviors of the quadrotor was required.

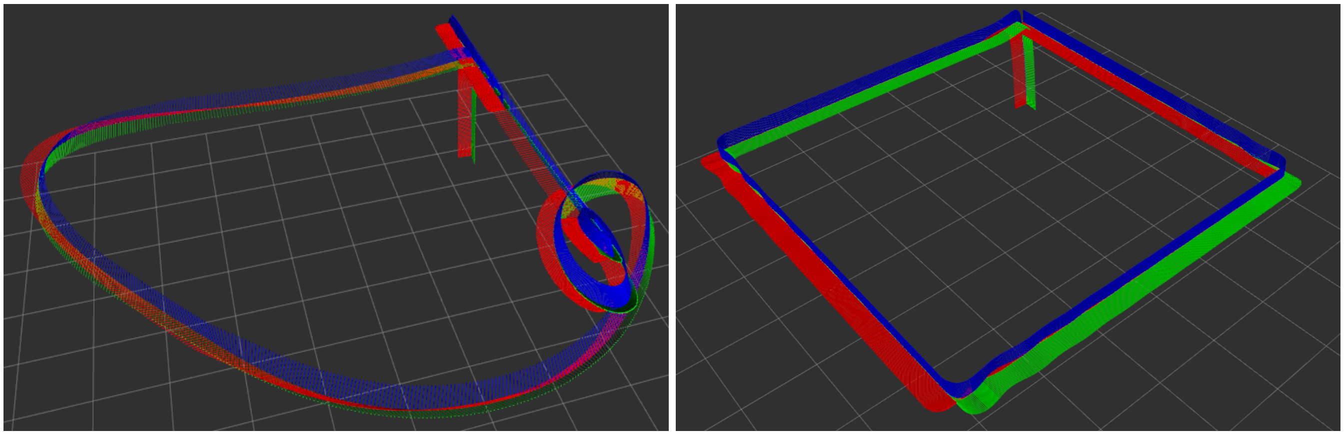
As it has been previously stated, in Exercise 7, the execution of the trajectories was performed by running the node with a parameter that indicates the desired trajectory to execute as ”`rosrun p12_arob_lab3 followTargets3D X`”. For instance, for executing the trajectory contained in the file `target4.txt` the command would be ”`rosrun p12_arob_lab3 followTargets3D 4`”. Thus, after executing each of the trajectories provided in the lab its visualization has been grouped in figures 3-5.



(a) Trajectory 1

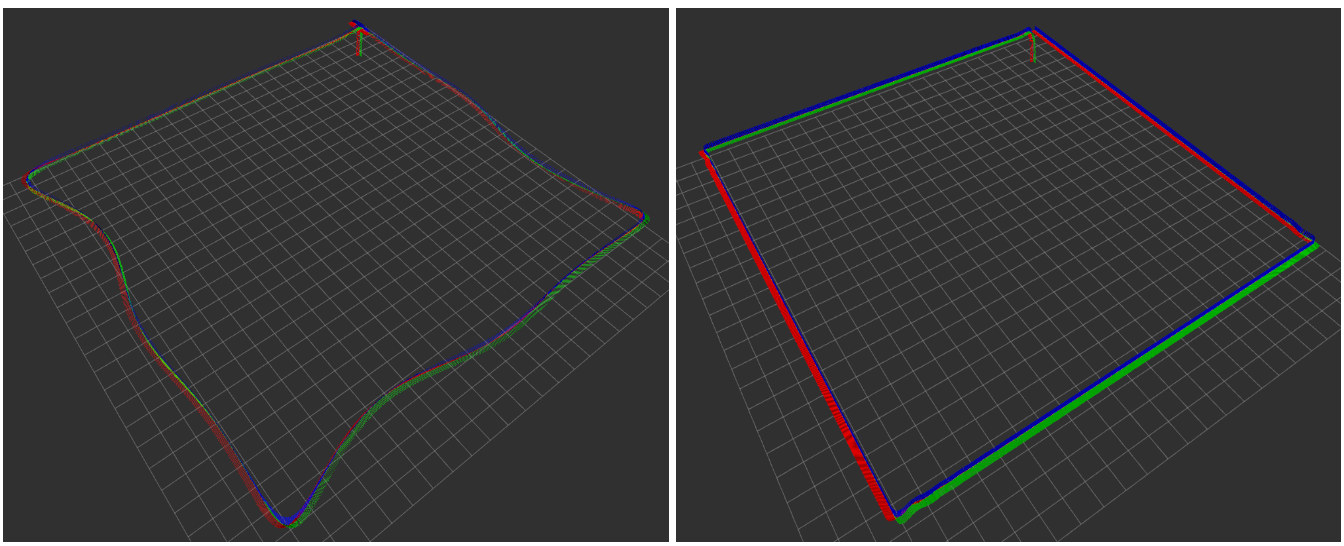
(b) Trajectory 4

Figure 3: Trajectories 1 and 4.



(a) Trajectory 2 (b) Trajectory 5

Figure 4: Trajectories 2 and 5.



(a) Trajectory 3 (b) Trajectory 6

Figure 5: Trajectories 3 and 6.

As it can be seen in the previous figures, the trajectories 1 and 4, 2 and 5, 3 and 6, represent the same path. However, the behaviour of the quadrotor in each case is different, having a more uniform movement in trajectories 4-6, where the drone performs a squared-shape path, whereas in trajectories 1-3 this squared-shape path is not easily differenciated. This is due to the amount of targets used in each trajectory, being paths 1-3 defined by 5 control points and paths 4-6 defined by almost 400 control points. A small amount of defined targets provoke a big acceleration of the quadrotor between each of the control points. This high acceleration may cause the overshooting of the next target, being some goals not reached in the first attempt and having the drone to correct its position, by generating a circular movement around the control point, until it is finally reached. On the other hand, a big amount of defined targets provoke a uniform controlled velocity in most of the cases, avoiding problems such as overshooting. Nonetheless, the distance between targets is also important, as if this distance is not correctly established, it might result in a tilting of the quadrotor, which accelerates and decelerates in a really small amount of time.

Appendix

followTargets3D.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
#include <bits/stdc++.h>
#include <ros/ros.h>
#include <ros/package.h>
#include <geometry_msgs/PoseStamped.h>
#include <nav_msgs/Odometry.h>

using namespace std;

class FollowTargets3DClass {
  ros::NodeHandle nh_;
  ros::Publisher goal_pub_;
  ros::Subscriber position_sub_;
  geometry_msgs::PoseStamped Goal;
  ifstream inFile;
  std::vector<std::vector<float> > targets;
  int currentTarget; //index with the next target to reach
  bool end_reached;
  int file_id = 0;

  std::vector<double> prev_pos = {0,0,0};


public:
  FollowTargets3DClass(int argv_file_id) { //in the contructor you can read the targets from the text file

    // Subscribe to the /base_pose_groun_truth topic
    position_sub_ = nh_.subscribe("/ground_truth/state", 1, &FollowTargets3DClass::UpdateGoal, this);
    // Assign the node goal to be a publisher in the topic "goal"
    goal_pub_ = nh_.advertise<geometry_msgs::PoseStamped>("/command/pose", 1, true);

    std::string packagePath = ros::package::getPath("p12_arob_lab3");
    //ROS_INFO("PackagePath: %s", packagePath.c_str());
    // We load the 6 files
    file_id = argv_file_id;
    evaluate_file_id(packagePath);
    end_reached = false;
    GetNewGoal();
  }

  ~FollowTargets3DClass() {
  }


  void evaluate_file_id(std::string packagePath){
      switch(file_id){
      case 1:
        loadTargetsFromFile(packagePath+"/src/targets.txt");
        break;
      case 2:
        loadTargetsFromFile(packagePath+"/src/targets2.txt");
        break;
```

```
      case 3:
        loadTargetsFromFile(packagePath+"/src/targets3.txt");
        break;
      case 4:
        loadTargetsFromFile(packagePath+"/src/targets4.txt");
        break;
      case 5:
        loadTargetsFromFile(packagePath+"/src/targets5.txt");
        break;
      case 6:
        loadTargetsFromFile(packagePath+"/src/targets6.txt");
        break;
      default:
        loadTargetsFromFile(packagePath+"/src/targets.txt");
    }
  }

  //complete the class by adding the functio that you need

  void loadTargetsFromFile(const std::string& filename) {
    inFile.open(filename);


    if (!inFile.is_open()) {
      std::cerr << "Error: Unable to open the file " << filename << std::endl;
    } else {
      std::string line;
      int i = 0;
      while (std::getline(inFile, line)) {
        // Parse the line to extract x and y values
        std::istringstream iss(line);
        char delimiter = ';';
        std::vector<float> target(3);

        if (iss >> target[0] >> delimiter >> target[1] >> delimiter >> target[2]) {
          targets.push_back(target);
        } else {
          std::cerr << "Warning: Invalid format in line: " << line << std::endl;
        }
      }
      currentTarget = 0;
      inFile.close();
    }
  }

  void GetNewGoal() {

    if(!targets.empty()){
      std::vector<float> target = targets.front();
      targets.erase(targets.begin());

      std::cout << " Goal Update: ("<< target[0] << ", " << target[1] << ", " << target[2]
    << ")" << endl;

      //  update the goal
      Goal.pose.position.x = static_cast<float>(target[0]);
      Goal.pose.position.y = static_cast<float>(target[1]);
      Goal.pose.position.z = static_cast<float>(target[2]);


      //std::cout << " Goal= ("<< Goal.pose.position.x << ", " << Goal.pose.position.y << ")
```

```
" << endl;
      // pubilsh the new goal
      goal_pub_.publish(Goal);
    } else if(!end_reached){
      std::cout << "Padron has passed by all the targets, please add more difficulty to the circuit!!!" << endl;
      end_reached = true;
    }
  }

  void UpdateGoal(const nav_msgs::Odometry& msg) {
    //float ex = Goal.pose.position.x - msg.pose.pose.position.x;
    //float ey = Goal.pose.position.y - msg.pose.pose.position.y;
    //float ez = Goal.pose.position.z - msg.pose.pose.position.z;

    std::vector<double> current_pos{msg.pose.pose.position.x,msg.pose.pose.position.y,msg.pose.pose.position.z};
    std::vector<double> goal_pos{Goal.pose.position.x,Goal.pose.position.y,Goal.pose.position.z};

    // If the robot is close to the goal move to the next target
    if(euclidean_dist(current_pos,goal_pos)  < 0.2) {
      if(!targets.empty()){
        std::cout << endl << endl << "Target " << currentTarget << " Reached!!!" << endl << "Moving to the next target..." << endl;
        currentTarget++;
      }
      GetNewGoal();
    }


    // If the robot is not moving, reload the goal, because it might be due to a bad publication in the topic
    // This has been solved by stablishing the latch parameter on the publisher to true
    //if(!isMoving(current_pos))
      //ReloadGoal();

    // Update previous position
    prev_pos = current_pos;



  }

  // Euclidean distance calculus
  float euclidean_dist(std::vector<double> origin, std::vector<double> goal){
    float ex = goal[0] - origin[0];
    float ey = goal[1] - origin[1];
    float ez = goal[2] - origin[2];

    float dist = sqrt(pow(ex,2)+pow(ey,2)+pow(ez,2));

    return dist;
  }

  // Checks if the robot is moving by comparing the previous position with the current position
  bool isMoving(std::vector<double> current_pos){

      if(euclidean_dist(current_pos,prev_pos)  < 0.2)
        return false;
```

```
      else
        return true;

  }


  // Re-Publish the goal, in case that the robot didn't receive it correctly
  // Normally called only for the first goal
  void ReloadGoal(){
    goal_pub_.publish(Goal);
  }

};


int main(int argc, char** argv) {

  int file_id = 0;
  if(argc > 1)
    file_id = stoi(argv[1]);

  ros::init(argc, argv, "followTargets3D");
  ros::NodeHandle nh("~");
  FollowTargets3DClass FT(file_id);


  ros::spin();
  return 0;
}
```