SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ DEGLI STUDI LA SAPIENZA

Dipartimento di Ingegneria informatica, automatica
e gestionale Antonio Ruberti

Master in Artificial Intelligence and Robotics

Collision Avoidance

Antonio Purificato
2019135

Academic year 2021/2022

Introduction

In this project are analyzed two main topics in the field of robotics: path planning and obstacle avoidance.

- Path planning: in path planning are modelled the features of the environment and, using these features, is determined a path for the navigation;
- Obstacle avoidance: given a trajectory, is possible to identify an obstacle and to avoid it, reaching with a different path the goal position. Obstacle avoidance assumes that, during motion, an obstacle is not in front of the robot; if there is an obstacle this one must be identified, using sensors and appropriate algorithms.

The architecture of this project is the SRRG Ecosystem provided by the professor. This is a complete simulation platform based on ROS containing a planner, a localizer and a map server. The environment where the robot can move is the map of DIAG (Department of Computer, Control and Management Engineering) of the University "La Sapienza" of Rome.

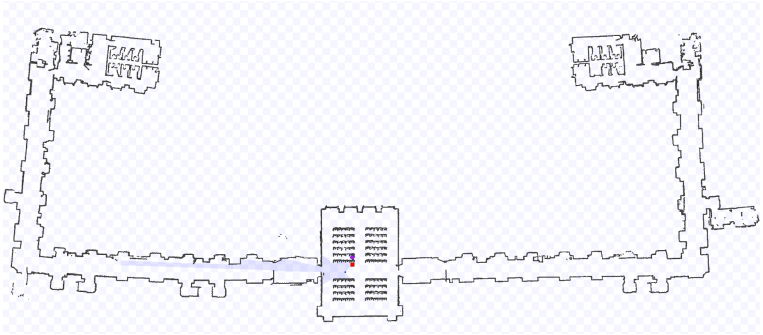

Figure 1: Map

INTRODUCTION

The goal of this project is to build a ROS node that computes a repulsive field from the local laser scans and modulates the `/cmd_vel` so that the robot does not clash to the wall. The program must receive an input, process this input and return an output that allow the agent to avoid the obstacles.
The robot acts in the environment and, during the motion, checks for obstacles in the path. To accomplish this task the environment is represented as a field of repulsive forces acting on the robot. While moving are computed the distances between all the points seen by the laser scan and the robot and, if the robot is too close to an obstacle, the forces are modified to avoid the closest obstacle. In this implementation are used 3 ROS topics for the communication:

- `/cmd_vel_call`: the user uses this topic to move the robot; in this case with the `teleop_twist_keyboard` package provided by ROS is possible to move the agent using the keyboard;
- `/cmd_vel`: is the topic used by the collision avoider to send the velocity of the robot that usually is different from the movement given by the user because of the avoidance operation;
- `/base_scan`: The avoider receives from this topic all the informations obtained by the laser scan.

Chapter 1

Deployment choices

1.1 cmd_vel_callback

```
void cmd_vel_callback(const geometry_msgs::Twist::ConstPtr& msg){

    command_received = true;
    velocity_received = *msg;
}
```

This function checks if the velocity message was received. In case of positive answer sets the boolean variable `command_received` to True and saves the message that was received in the variable `velocity_received`.

1.2 laser_callback

This function is the main component of the avoider and deals with different tasks.

```
if (!command_received ) return;
command_received = false;
```

I check if the velocity command was received and, in case of positive answer, i reset the boolean variable.

```
tf::TransformListener listener;
laser_geometry::LaserProjection laser_projection;
sensor_msgs::PointCloud cloud;

/*Transform a sensor_msgs::LaserScan into a sensor_msgs::PointCloud in
target frame.*/
laser_projection.transformLaserScanToPointCloud("base_laser_link",
 *msg, cloud, listener);
```

Transform the laser scan of the base link from a linear array into a 3D points cloud, accounting for movement of the laser over the course of the scan.

```
 tf::StampedTransform tf_obstacle;
  try{
       listener.waitForTransform("base_footprint", "base_laser_link",
       ros::Time(0), ros::Duration(1.0));
       listener.lookupTransform("base_footprint", "base_laser_link",
       ros::Time(0), tf_obstacle);
   }
   catch(tf::TransformException &e){
       ROS_ERROR("%s", e.what());
       return;
   }

   /*Coordinates in robot reference frame*/
   Eigen::Isometry2f laser_tf = convertPose2D(tf_obstacle);
```

In these lines of code i convert the points from the laser scan to the reference frame of the robot. For doing this i wait for a transform from the `\base_footprint` to the `base_laser_link` and i don't wait for longer than one second. After i received a transform i continue with the transformation and i save the result in `tf_obstacle`. In

this process is possible that we have exceptions and for this reason i use a try-catch statement. Using the function `convertPose2D` i modify my transform to an isometry using a 2x2 rotation matrix.

```
    float force_x, force_y = 0;
    float actual_distance, current_distance;
    Eigen::Vector2f actual_position, current_position;

    actual_position(0) = cloud.points[540].x;
    actual_position(1) = cloud.points[540].y;

    /*I compute the actual distance between the laser scan of my robot
    and the closest obstacle*/
    actual_position = laser_tf * actual_position;
    actual_distance = sqrt(pow(actual_position(0),2)+
    pow(actual_position(1),2));
```

For the initialization i save in a 2D vector the x and y initial coordinates of the point in front of the robot. For doing this i extract from the cloud the x and y values of the points in the middle. In fact i know that the size of the cloud is 1080 and i use 540 because in this mode i obtain the point in front of the robot. After this i compute the position of this point in the robot reference frame using the `laser_tf` and i save the norm of the distance.

```
    *I scan all the elements of the cloud and i compute the x and y
    components of the obstacle's position*/
    for(auto& point: cloud.points){
        current_position(0) = point.x;
        current_position(1) = point.y;

        /*Position of the obstacle in the robot reference frame*/
        current_position = laser_tf * current_position;

        /*Current distance between the robot and the obstacle*/
        float current_distance = sqrt(pow(current_position(0),2) +
        pow(current_position(1),2));

        /*I  compute the resultant forces*/
        force_x += current_position(0) / pow(current_distance,2);
```

```
15          force_y += current_position(1) / pow(current_distance,2);
16
17          /*If i'm closer to the obstacle wrt. the previous position
18          i update the initial position with the current position*/
19          if(current_distance < actual_distance){
20              actual_distance = current_distance;
21              actual_position = current_position;
22          }
23      }
```

I iterate throw all the points of the cloud and for all of them i save the x and y components in the vector `current_position` and i compute the position in the robot reference frame and the norm of the distance. After this i compute the resultant forces, in particular the x and y components of the module of the resultant force. At the end, if the current position is closest to an obstacle with respect to the previous position, i update the position and the distance, otherwise i continue.

```
1       if(actual_distance < 0.2){
2           force_x = -force_x / AVOIDANCE_FACTOR;
3           force_y = -force_y / AVOIDANCE_FACTOR;
```

If i am too close to an obstacle i use the resultant forces but modifying the sign, and i can stop or move the robot. I use a very important parameter that is called `AVOIDANCE_FACTOR=10000`, in this mode i am smoothing the movement of the robot during the avoidance operation. In particular the robot is able to move between the chairs of the map (this is not very easy) and deflects the trajectory to a collision-free space. In fact, out of 100 attempts, only 5 times the robot crushed the wall due to the avoidance operation and in a real scenario it would probably get damaged.

```
1           geometry_msgs::Twist msg_send;
2           msg_send.linear.x = velocity_received.linear.x + force_x;
3           msg_send.linear.y = velocity_received.linear.y + force_y;
4           msg_send.linear.z = velocity_received.linear.z;
5
6           if(actual_position(1) > 0)
7               msg_send.angular.z = - magnitude;
8           else if (actual_position(1) < 0)
9               msg_send.angular.z = magnitude;
```

```
10
11          velocity.publish(msg_send);
12      }
13      else {
14        velocity.publish(velocity_received);
15      }
16  }
```

At the end i have to publish the message. In this message i modify the x component of the linear velocity summing the x component of the resultant force and i do the same for the y component. The z component keeps unchanged. If the distance along the y axis between the robot and the obstacle is greater than zero i set the angular component of the velocity equal to the magnitude, otherwise i change the sign of the magnitude. If i am not close to the obstacle i can skip the avoidance operation and i simply send the message that was previously received.

1.3 main

```
1   int main(int argc, char **argv){
2
3   /*Starting the collision avoidance node */
4   ros::init(argc, argv, "collision_avoidance");
5   ros::NodeHandle n;
6
7   /*Subscriber for the velocity command (topic) of the robot*/
8   ros::Subscriber cmd_vel_sub = n.subscribe("cmd_vel_call", 1,
    cmd_vel_callback);
9
10  /*Subscriber for the laser scan (topic) of the robot*/
11  ros::Subscriber laser_scan_sub = n.subscribe("base_scan", 1,
    laser_callback);
12
13  velocity = n.advertise<geometry_msgs::Twist>("cmd_vel",1000);
14  ros::spin();
15  return 0;
16 }
```

- `ros::init` is used to initialize ROS and `ros::NodeHandle` for the initialization of the node. These are the main steps for the creation of a generic node in ROS.
- `cmd_vel_sub = n.subscribe("cmd_vel_call", 1, cmd_vel_callback);` subscribe to the topic `cmd_vel_call` with the master. All the messages received are sent to a callback function, in this case `cmd_vel_callback`. The same procedure can be applied to the following line:
  `laser_scan_sub = n.subscribe("base_scan", 1, laser_callback);`.
  In this case the difference is simply that we are subscribing the topic `base_scan`.
- `velocity = n.advertise<geometry_msgs::Twist>("cmd_vel",1000);`
  This call connects to the master to publicize that the node will be publishing messages on the `cmd_vel` topic. This method returns a Publisher that allows to publish a message on this topic.
- `ros::spin()` enters a loop, calling message callbacks as fast as possible.

Chapter 2

Setup and execution

2.1 Setup

0. Go in the src folder of your catkin workspace.

1. Download the C++ teleop_twist_keyboard package to move the robot in the environment using the keyboard.

```
git clone https://github.com/methylDragon/teleop_twist_keyboard_cpp.git
```

2. Download the avoidance package.

```
git clone https://github.com/antoniopurificato/collision_avoidance.git
```

3. Build the project.

```
cd .. && catkin build && source devel/setup.bash
```

2.2 Execution

0. Go in the catkin workspace.

1. Start the roscore.

```
roscore
```

2. Open a different window and run the teleop_twist_keyboard.

```
source devel/setup.bash && rosrun teleop_twist_keyboard_cpp
teleop_twist_keyboard cmd_vel:=cmd_vel_call
```

3. Open a different window and run the avoider.

```
source devel/setup.bash && rosrun collision_avoidance collision_avoider
```

4. Open a different window and run the stage.

```
source devel/setup.bash && cd src/srrg2_configs/navigation_2d/
&& rosrun stage_ros stageros
cappero_laser_odom_diag_obstacle_2020-05-06-16-26-03.world
```

Come back to the terminal window with the teleop_twist_keyboard active and use the keyboard to move the robot.