Universidad De Granada

Grado ingeniería informática (2016 – 2017)

Técnicas Sistemas Inteligentes

Memoria Práctica 1 | Robot Diambulador

Trabajo realizado por: Antonio Miguel Morillo Chica

---

1. Introducción al problema.

En la sesión 2 vimos como usar tutleblot con tele-operación además comenzamos a usar wander_bot, un paquete podía hacer que turtlebot se moviese automáticamente y pararse cuando encontrase un obstáculo. Por un lado nos suscribimos al topic scan con el que podremos acceder a la información proporcionada por LaserScan, por otro hacíamos publicaciones al topic de telecomunicación para poder controlar al robot.

Se propuso así crear un nuevo paquete, un robot que no se parase ante un obstáculo sino que lo evitara para poder circular por el mundo libremente evitando los obstáculos.

2. Solución al problema.

A primera vista el problema es simple, el robot tiene que ver un obstáculo y girarse para continuar su camino. La idea es pues, hacer que el robot rote en el mismo punto hasta que no vea ningún obstáculo y tras esto continuar recto.

Para implementar la solución nos basaremos completamente en stopper, tan solo habrá que modificar algunas líneas para que nuestra idea pueda funcionar.

3. Pasos hasta llegar a la solución.

Lo primero es crear el paquete random_walk, unicamente desplazarse al directorio de trabajo, dentro de este a src y usar la orden catking_create_package:

```
$ cd working_space/src
$ catkin_create_pkg random_walk std_msgs rospy roscpp
```

Posteriormente modifiqué los archivos de stopper para implementar la solución descrita en el apartado anterior.

La función void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan): Lee las publicaciones del sensor laser para controlar cuando hay un objeto delante.

La modificación realizada consiste en: dividir la visión en tres partes que funcionarán como izquierda, centro y derecha. Como en stopper, cuando se detecte que hay un obstáculo pasará a true, guardaremos el índice y romperemos el bucle:

```
int primerTercio = (minIndex+1+maxIndex)/3;
int segundoTercio = (primerTercio*2)-(primerTercio/2);

for (int currIndex = minIndex + 1; currIndex <= maxIndex; currIndex++) {
  if (scan->ranges[currIndex] < MIN_DISTANCE_OBSTACLE) {
    ROS_INFO("Obscalo detectado");
            posObstaculo = currIndex;
            isObstacleInFront = true;
            break;
  }
}
```

Una vez que se sabe donde está el obstáculo, es decir, isObstacleInFront es true, entonces calculamos a donde girar. Si el obstáculo si el indice está en el primer tercio de la visión giro a la derecha, si está en el tercer tercio giro a la izquierda y si está en el segundo lo que hago es girar más "fuertemente" a la izquierda ya que si es una esquina puede que tienda a meterse dentro. Demás pongo keepMoving a false para que en el startMoving, si keepMoving es false se llame a rotar y no a moveForward():

```
if (isObstacleInFront) {
  keepMoving = false;
  ROS_INFO("Calculo direccion");

  if(posObstaculo <= primerTercio){
    ROS_INFO("Obstaculo a la derecha");
    direccion = 0;
  }
  else if(posObstaculo <= segundoTercio){
    ROS_INFO("Giro fuerte");
    direccion = 2;
  }
```

```
    else{
      ROS_INFO("Obstaculo a la izquierda");
      direccion = 1;
    }
  }else{
    keepMoving = true;
  }
```

Una vez sé a donde quiero ir, gracias a que dirección es un dato miembro de la función, creo una función rotar que será llamada en startMoving():

```
  while (ros::ok()) {
    if (keepMoving) {
      moveForward();
    }else{
      rotate();
    }
    ros::spinOnce();
    rate.sleep();
  }
```

La función rotate decide como tiene que hacer el giro, hay que tener en cuenta que tuve que buscar como hacer esto y hay que publicar en msg.angular y sobre el eje z que es el perpendicular al plano:

```
void Random::rotate() {

double rotationSpeed = ANGLE_SPEED;

  if(this->direccion == 0){
    rotationSpeed = rotationSpeed;
  }else if (this->direccion = 1) {
    rotationSpeed = -1 * rotationSpeed;
  }else
    rotationSpeed = rotationSpeed + 3;

  if (rotates == 8) {
    int
  }

  geometry_msgs::Twist msg;
  msg.angular.z = rotationSpeed;
  commandPub.publish(msg);
}
```

Por último para que el launch acepte argumentos deberemos de editar el stopper.launch y dejarlo así:

```
<launch>
    <param name="/use_sim_time" value="true" />
    <param name="MIN_DISTANCE_OBSTACLE" value="0.7" />

    <!-- Launch turtle bot world -->
    <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

    <!-- Launch stopper node -->
    <node name="random_walk" pkg="random_walk" type="random_walk"
output="screen"/>
</launch>
```

Para usarlo solo necesitaremos consultar este parámetro accediendo al nodo de la forma: node.getParam("distancia_minima", MIN_DISTANCE_OBSTACLE) que lo que hará será poner en MIN_DISTANCE_OBSTACLE el valor de la cadena descrita en el launch, justo antes de usar la variable.