Práctica 2: Navegación local sin y con mapa

Lunes SGrupo 1.4:
Francisco Carrillo Perez.
Jose Antonio España Valdivia.
Rafael Leyva Ruiz.
Cristina Zuheros Montes.

30 Mayo 2016

Índice

1. Tareas para mejorar el comportamiento reactivo del robot respecto al uso de campos de potencial (navegación local sin mapa).

1.1. Conseguir aumentar el campo de visión. El campo de visión del robot simulado es de 270º, ¿cómo hacer para que el escaneo láser manejado sea mayor que el actualmente implementado?

Solución

Para mejorar el campo de visión del robot se han modificado las variables MIN_SCAN_ANGLE_RAD y MAX_SCAN_ANGLE_RAD, que están declaradas en el archivo "MyLocalPlanner.h" al valor 135.0 cuando originalmente estaban establecidas a 30.0.

```
double MIN_SCAN_ANGLE_RAD = -135.0/180*M_PI;
double MAX_SCAN_ANGLE_RAD = +135.0/180*M_PI;
```

De esta forma y tras hacer la conveniente conversión a radianes con los que trabaja el robot, el resultado al compilar es que el campo de visión del robot es de 270 grados.

1.2. Conseguir que no se demore tanto tiempo en detenerse cuando esté lo suficientemente cerca del objetivo. ¿Por qué tarda tanto en detenerse cuando está próximo al objetivo?. ¿Cómo solucionarlo?.

Solución Para contestar a la pregunta sobre porque tarda tanto en detenerse es tan sencillo como fijarse en como está implementada la asignación de la velocidad de nuestro robot en el código. Cuando nosotros calculamos la componente atractiva del objetivo y entramos en dicho campo de atracción, el módulo de la velocidad es inversamente proporcional a la distancia del objetivo existente, es decir, cuando nosotros nos acercamos al objetivo más pequeña es la velocidad y por ende más despacio se va a mover nuestro robot.

Solucionamos este problema realizando las siguientes modificaciones en el método setDeltaAtractivo().

```
double d = distancia(posGoal, pos);
double theta = atan2(posGoal.y - pos.y, posGoal.x - pos.x);
if (d-CAMPOATT.radius < TOLERANCIA) {
 deltaGoal.x = deltaGoal.y = 0;
 return;
}
if (d <= CAMPOATT.spread/2 + CAMPOATT.radius ) {
 deltaGoal.x = CAMPOATT.intens * (CAMPOATT.spread/2 + CAMPOATT.radius) * cos(theta);
 deltaGoal.y = CAMPOATT.intens * (CAMPOATT.spread/2 + CAMPOATT.radius) * sin(theta);
 return;
}
if (d <= CAMPOATT.spread + CAMPOATT.radius ) {
 deltaGoal.x = CAMPOATT.intens * (d - CAMPOATT.radius) * cos(theta);
 deltaGoal.y = CAMPOATT.intens * (d - CAMPOATT.radius) * sin(theta);
 return;
}
if (d > (CAMPOATT.spread + CAMPOATT.radius)) {
 deltaGoal.x = CAMPOATT.intens*CAMPOATT.spread*cos(theta);
 deltaGoal.y = CAMPOATT.intens*CAMPOATT.spread*sin(theta);
 return;
}
```

La modificación que hemos realizado en este método para solucionar el problema es la de actualizar la velocidad a una velocidad constante cuando estamos a la mitad de distancia del objetivo, más concretamente usando el campo de atracción del objetivo cuando el valor de este es la mitad y así podemos solucionar el problema de que cuando nos acercamos al objetivo la velocidad de nuestro robot se va reduciendo considerablemente.

1.3. Conseguir que el robot no tenga “comportamiento suicida”, e.d., cuando está cerca de un obstáculo acelera y choca con él. ¿Cuál es la causa del comportamiento suicida?. ¿Cómo evitarlo?.

Solución

El problema que se plantea resolver en este punto primero se intento resolver ajustando correctamente los campos de potencial pero como mas adelante el lector podrá apreciar esto no dio ningún resultado.

Este problema viene dado directamente por la forma de calcular las velocidades tanto angular como lineal del robot, ya que ‘v_lineal’ es calculada como el modulo del vector ‘DeltaTotal’con la siguiente formula:

$$\sqrt{DeltaTotal.x^2 + DeltaTotal.y^2}$$

Por lo tanto cuando la Delta Total crece mucho debido al factor repulsivo calculado a partir del escaneo de obstáculos crece mucho dicho modulo se dispara hasta cantidades de orden 10^{10} al menos en nuestro caso, por lo que el robot avanza muy rapidamente en la dirección del obstáculo por lo que choca.

Para ello se ha cambiado la función “LocalPlanner::setV_lineal” para que cuando el calculo de esa velocidad sea tan grande como el comentado esta velocidad sea establecida a mano a 0, para que de ese modo el robot no choque, si no que sólo aplique la el giro dado por la velocidad angular, obteniendo así un giro sin avance lineal y manteniendo el robot a salvo.

Veamos cómo ha quedado:

```
v_lineal =  sqrt(delta.x*delta.x + delta.y*delta.y);
ROS_INFO("velocidad linear %f", v_lineal);

if (v_lineal > 100000000) {
 v_lineal = 0.0; //aqui va 0.1
 ROS_INFO("v_lineal pequeña");
}
```

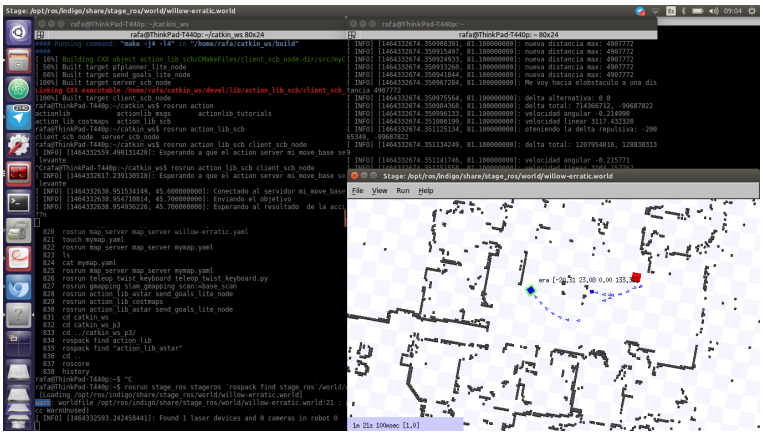

Figura 1: Mapa 1.

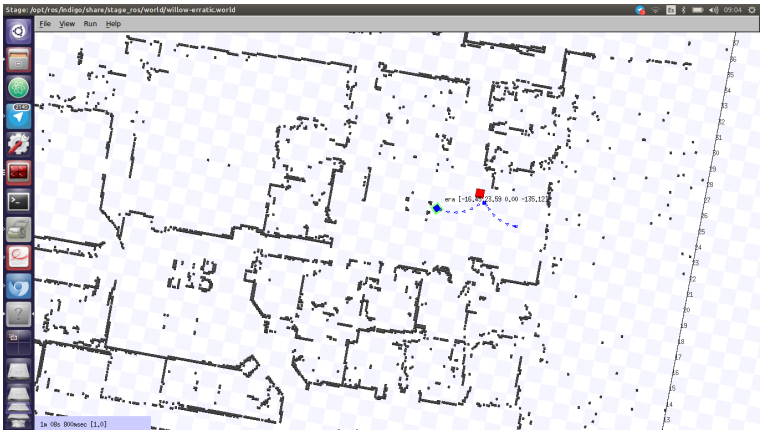

Figura 2: Mapa 2.

1.4. Conseguir que, una vez que el robot se queda atrapado en una esquina (en un mínimo local), trate de salir de esa situación. ¿Cuál es la causa del mínimo local?. ¿Cómo conseguirlo sin utilizar memoria (e.d., información de un mapa)?.

Solución

Que el robot se quede atrapado en una esquina nos indica que el vector atractivo y repulsivo son muy similares, por lo que el calculo de las velocidades es nulo y el robot no consigue avanzar.

Para solventar este problema se ha usado la técnica de Fuerzas artificiales, del ingles *virtual force field*, que consiste en establecer una fuerza virtual con el objetivo de intentar que el robot salga de el punto de atasco.

Nuestra implementación en concreto ha sido mediante una función en el objeto LocalPlanner, identificada como "LocalPlanner::EscapaMinimoLocal()", mediante la cual calculamos un vector conocido como "deltaAlt" que apunta hacia el objeto mas lejano en el campo de visión del robot, que con suerte sera un espacio en blanco, que en el escaner se representa como un objeto a distancia el máximo de la sensibilidad del láser.

Esta delta se obtiene con los mismos cálculos que el componente atractivo del campo, pero con unos valores diferentes declarados en "CAMPOALT".

La función descrita con anterioridad es usada en el "Server node" tras comprobar que el robot esta en un mínimo local (Los vectores atractivo y repulsivo son practicamente iguales) .

Ademas se ha modificado el método del "LocalPlanner" que calcula el componente "DeltaTotal" para tener en cuenta este caso y cambiar los cálculos oportunos.

Figura 3: Mapa 2.

Figura 4: Mapa 2.

1.5. Conseguir suprimir las oscilaciones del robot. ¿Cuál es la causa de las oscilaciones?. ¿Cómo eliminarlas en la mayor medida posible?.

Solución

Las oscilaciones se producen cuando el calculo de las velocidades da como resultado un giro pequeño en un sentido en una iteración de ROS, y un giro contrario en la siguiente. Esto se produce cuando la velocidad lineal es pequeña. Al se considerablemente grande, no hemos apreciado oscilaciones.

De modo que añadimos la siguiente restricción en el método setv_Angular():

```
if (v_lineal < 0.06) {
 v_angular = 0;
}
```

Con esto lo que conseguimos es que cuando el robot vaya con una velocidad lineal muy pequeña, no se produzcan dichas oscilaciones ya que ponemos la velocidad angular a cero.

2. Tareas para mejorar el comportamiento del robot mediante técnicas de navegación local con mapa.

2.1. Contemplar el uso de distintos mapas para la experimentación. Los mapas pueden generarse a partir de una imagen mediante el paquete mapserver. Ver explicación sobre manejo de mapas y mundos simulados en la documentación adjunta

Solución

Para ello vamos a utilizar los archivos dados como ejemplo que se encuentran en la plataforma PRADO. Los mapas utilizados se muestran en la Figura 2.1 y la Figura 2.1, han sido obtenidos de Google Images.

Para cogerlos con el rosrun, creamos dos archivos distintos .world, que serán los que determinarán qué imagen se va a utilizar, y las coordenadas y la orientación con la que aparecerá el robot.

Las imágenes deben estar bien determinadas. En blanco es donde el robot podrá moverse y no hay obstáculos, en negro se representarán los obstáculos y en gris las zonas que no se utilizan.

La posición del robot la podemos modificar en la siguiente línea del archivo .world:
origin [1.5000 0.000 0.200 -50.000]
Siendo el primer parámetro la x, el segundo la y, el tercero la z y por último la orientación.

El mapa que va a cargar se puede determinar en la siguiente línea:
bitmap "../maps/mapa1.jpg"

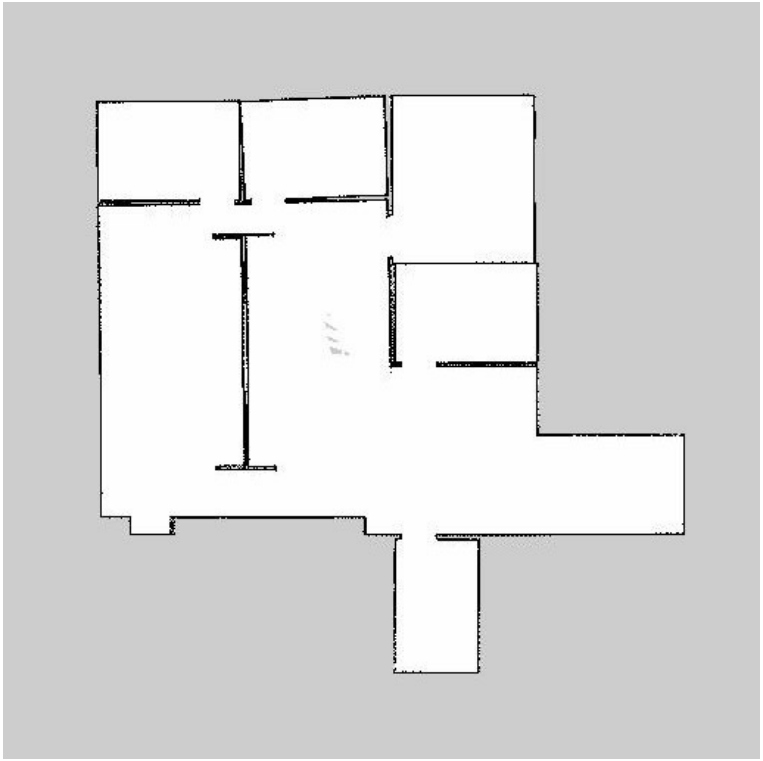

Figura 5: Mapa 1

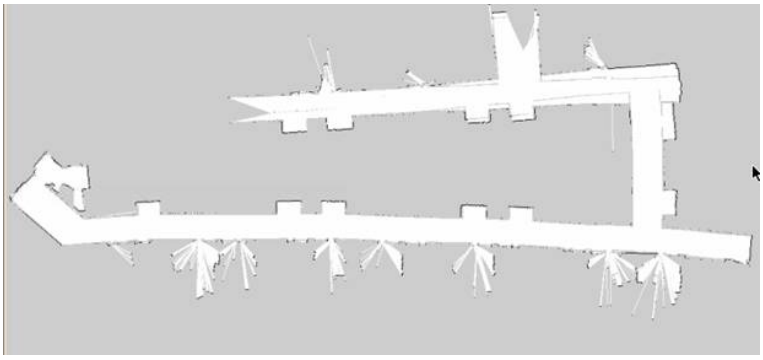

Figura 6: Mapa 2