Práctica 1

Nestor Morales De la Fuente

1. Introducción

Un algoritmo de evitación de obstáculos necesita al menos uno más sensores de entrada. Estos sensores se procesan, en nuestro caso el tópico /scan de ROS, y se obtiene una idea generalizada de donde se encuentra el robot y los obstáculos que tiene a su alrededor.

2. Desarrollo

Para desarrollar el algoritmo ha sido creada la clase “robotClass” donde se guardaran los diferentes valores del robot. Cada vez que se recibe un valor en el tópico /scan las variables “self” del robot se actualizarán. Básicamente se pretende evitar ejecutar código innecesariamente ya que mediante la clase “rospy.Rate()” decidiremos la frecuencia con la que ejecutaremos nuestro algoritmo.

Para procesar la entrada del escáner se han sustituido los NaN por 1000. El algoritmo cogerá la mínima distancia así que los NaN no tendrán ningún efecto en este.

He dividido el topic /scan en 3 partes: “front”, “left” y “right” tal como se muestra en la Figura 1. Para ello se cuentan el número de entradas y se dividen entre 3 partes iguales. Este método hace que el algoritmo funcione sin importar el rango del escáner en cuestión, siempre que sea lo suficientemente amplio para detectar objetos a la derecha, enfrente e izquierda. En la figura se puede ver como el robot azul tiene un rango menor que el naranja, pero ambos tienen dividido el rango en 3 partes iguales.

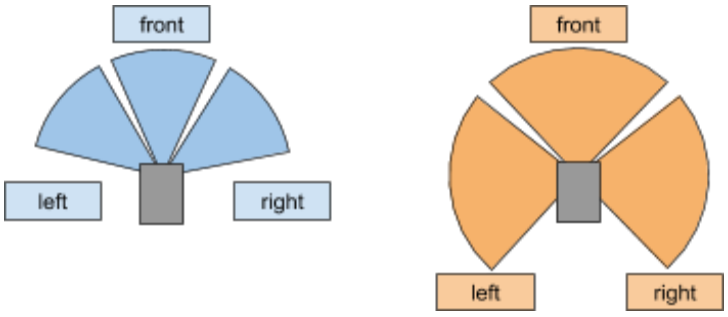


Figura 1. A la izquierda un robot con una visión de 180 grados. A la derecha un robot con una visión de 270 grados.

Una vez seccionada se coge el valor más pequeño de cada medida sin importar donde se encuentre dentro de la sección. Por ejemplo, si hay un objeto a 0.6 metros a la izquierda el robot no sabrá exactamente dónde se encuentra, solo que hay un objeto a 0.6 metros a la

izquierda. Con este método habremos reducido la entrada del radar a 3 medidas. La distancia del objeto más cercano a la izquierda, centro y derecha.
Cada vez que se produzca una ejecución en el código marcada por “rospy.Rate()”, el robot chequeará si la distancia marcada por las 3 secciones del robot (“left, front y right”) es menor que la establecida por la variable global *MIN_BUMP_DISTANCE*. En el caso de que lo sea el robot girará sobre si mismo en la dirección opuesta a donde se encuentra el robot, tal como se muestra en la Figura 2.

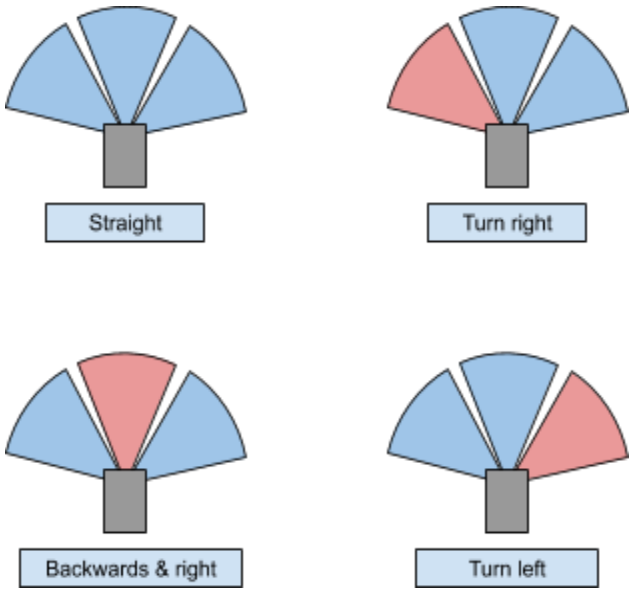


Figura 2. Acción del robot dependiendo de donde se encuentra el obstáculo más cercano

3. Pruebas

El desarrollo del algoritmo se ha hecho mayoritariamente en Gazebo. Se ha empleado Gazebo 7.4 y ROS kinetic en una máquina virtual.

3.1. Stage

Los resultados de la simulación en stage se pueden ver en la siguiente figura. El algoritmo funciona adecuadamente y la amplitud del escáner es mayor que en Gazebo. Debido a esto el robot es capaz de reaccionar mejor a obstáculos que se encuentran en los laterales.

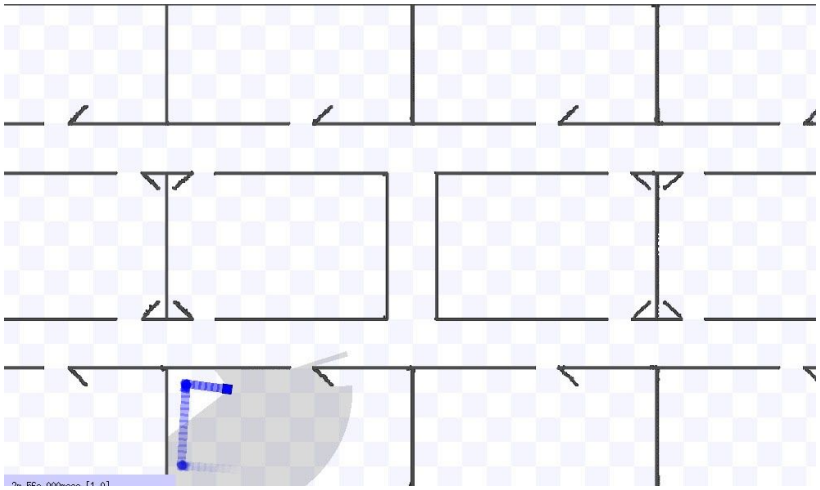


Figura 3. Ejecución del algoritmo en Stage. El Robot es capaz de orientarse y evitar todas las paredes

3.2. Gazebo

A diferencia de Stage donde todo funcionaba a la perfección, han habido varias complicaciones a la hora de ejecutar el algoritmo en Gazebo. La primera de todas es que el escáner es mucho más pequeño, lo que hace que dividir la visión en dos partes no parezca lo más útil. Por otro lado, hay muchos problemas con el “range” máximo y mínimo del láser. Los “NaN” dificultan bastante la computación, teniendo en cuenta que en Stage funcionaba muchísimo mejor. En esta simulación la velocidad si que importa y el algoritmo se ha de ejecutar con una velocidad lineal menor. Por que? Personalmente creo que se debe a la simulación en Gazebo y a no estar optimizada a la perfección, lo que en futuras versiones se consigue arreglar.

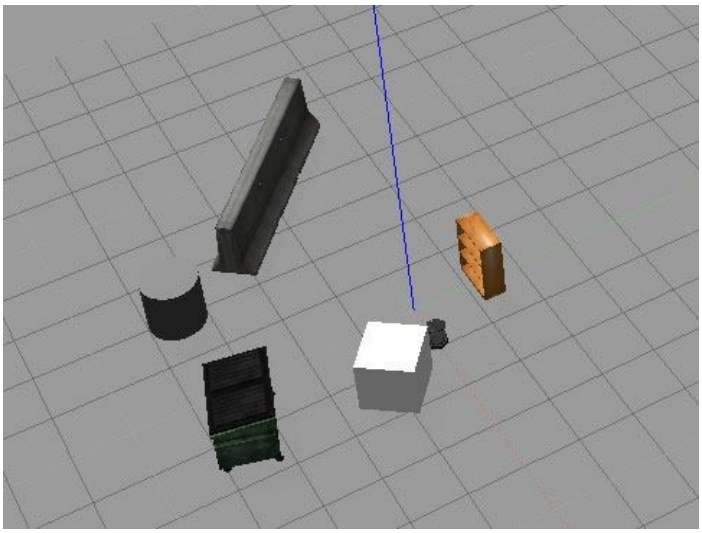


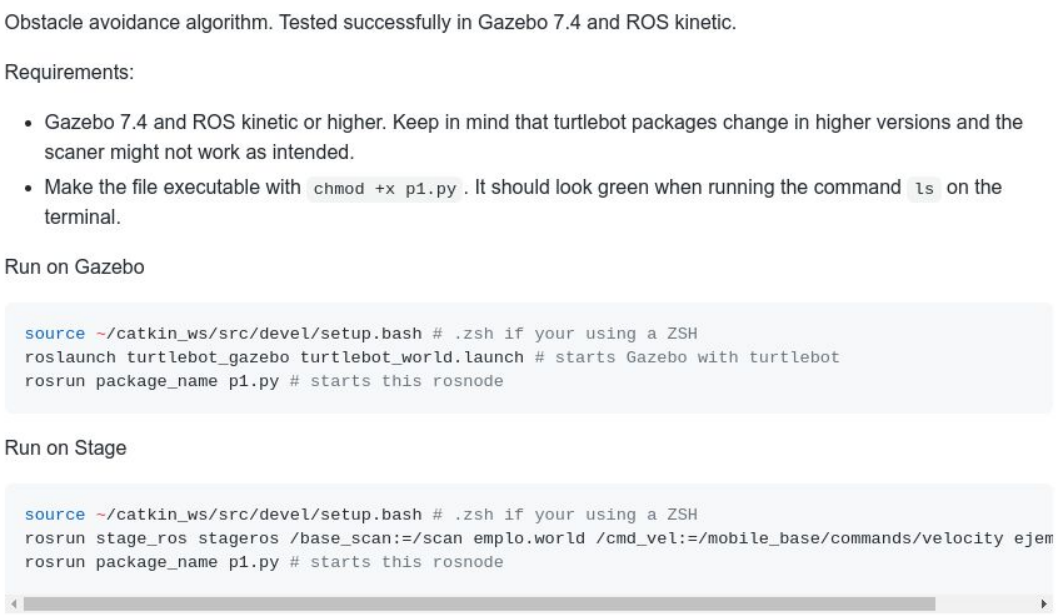
Figura 4. Ejecución de la simulación en Gazebo. El robot es capaz de orientarse y evitar los obstáculos adecuadamente siempre y cuando se mantenga una velocidad lineal de 0.2 m/s.

4. Conclusión

El desarrollo de este algoritmo en una máquina virtual ha sido muy tedioso y sin duda intentaré resolverlo en ROS Noetic de haberlo hecho otra vez. En el pasado he desarrollado codigo en ROS y lo recomendaría encarecidamente.
Por último la compatibilidad con Stage no es nada buena. Si bien nos sirve para agilizar el desarrollo del algoritmo, surgen problemas posteriores con la compatibilización de tópicos (ya que hay que hacer el remapping al lanzarlo). ROS kinetic no es una opción óptima ya que estamos hablando de Software desarrollado hace más de 4 años.
En cuanto al algoritmo estoy contento con los resultados obtenidos y su robustez. Se ha desarrollado creando una clase para que el código se ejecute de manera limpia y se puedan incluir más tópicos y acciones en el futuro. De esta manera evitamos depender del callback del /scan.

4.1 Readme

Se ha desarrollado un Readme para la ejecución del código en este repositorio de Github.



Obstacle avoidance algorithm. Tested successfully in Gazebo 7.4 and ROS kinetic.

Requirements:

- Gazebo 7.4 and ROS kinetic or higher. Keep in mind that turtlebot packages change in higher versions and the scaner might not work as intended.
- Make the file executable with `chmod +x p1.py`. It should look green when running the command `ls` on the terminal.

Run on Gazebo

```
source ~/catkin_ws/src/devel/setup.bash # .zsh if your using a ZSH
roslaunch turtlebot_gazebo turtlebot_world.launch # starts Gazebo with turtlebot
rosrun package_name p1.py # starts this rosnode
```

Run on Stage

```
source ~/catkin_ws/src/devel/setup.bash # .zsh if your using a ZSH
rosrun stage_ros stageros /base_scan:=/scan emplo.world /cmd_vel:=/mobile_base/commands/velocity ejem
rosrun package_name p1.py # starts this rosnode
```

Figura 5. Readme file del proyecto.