UNIVERSIDAD DE SEVILLA

Memoria de Laboratorio de Robótica

Álvaro Pérez-Borbujo Mohedo
Miguel Tejado García
Ángela Ríder Jiménez

Laboratorio de robótica

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Enero de 2025

Índice

1. Introduction

Este proyecto consiste en la implementación un algoritmo de seguimiento de trayectoria y evitación de obstaculos en ROS y simularlo en un robot utilizando el simulador gazebo.

El robot utilizado es el modelo Burger de Turtlebot 3.

El algoritmo elegido ha sido el algoritmo VFF, Virtual Force Field, que se explicará más adelante en esta memoria.

2. Estructura de ROS

Se ha utilizado el middleware de ROS como intermediario en la comunicación con el robot y el simulador. La distribución de ROS utilizada ha sido 'noetic'.

Se ha implementado todo dentro del docker proporcionado por los profesores de la asignatura llamado 'lab_rob_container'.

En la siguiente imagen se detallan los nodos utilizados y la comunicación mediante topics entre ellos. Los nodos que han sido creados en este proyecto son /Turtlebot-Controller y /Obstacle_detection.

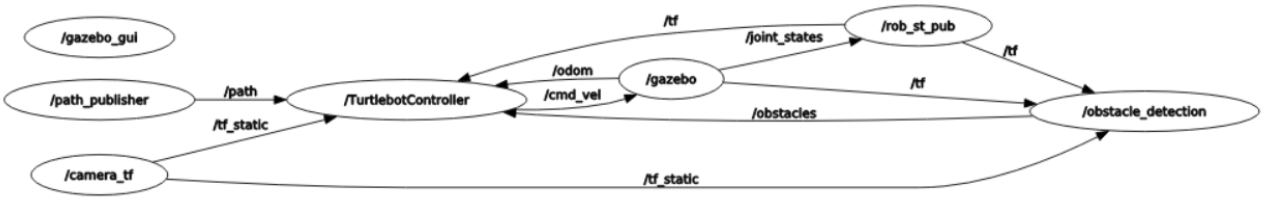

Figura 1: Estructura de nodos de ROS

El nodo del controlador, recibe un camino y la posición del robot respecto al eje de referencia fijo 'odom', además de las transformaciones entre ejes de referencia necesarias para la localización del robot, manejadas por el paquete tf en ROS. Y envia al robot, dentro del nodo del simulador Gazebo, los comandos de movimiento, que para el turtlebot son velocidad lineal y velocidad angular.

El nodo de detección de obstaculos recibe los datos de un LiDar, publicados en el topic /scan y envia al controlador mediante el topic /obstacles, un array de vectores de la posición de los obstaculos detectados.

3. Algoritmo VFF

3.1. Descripción conceptual

El algoritmo VFF, por sus siglas en ingles Virtual Force Field, es un algoritmo de navegación local que ha tenido gran importancia desde su creación en 1989 [2]. Este algoritmo permite la planificación de movimiento hacia el destino a la vez que evita de obstáculos en tiempo real.

Se basa en la creación de un conjunto de fuerzas virtuales o imaginarias, que dan el nombre al algoritmo, principalmente de dos tipos: una fuerza de atracción hacia el objetivo y fuerzas repulsivas de los obstáculos.

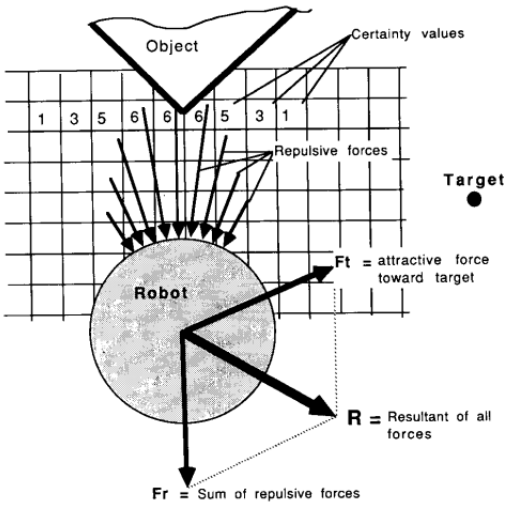

Figura 2: Concepto de Virtual force field

La fuerza de atracción, inicia en la posición del robot, tiene dirección hacia un punto objetivo y magnitud contante(en vff tradicional). En el caso de que el robot siguiera una trayectoria, esta deberá ser dividida en puntos o subobjetivos, y el subobjetivo actual será el punto destino al que haga referencia el algoritmo.

Las fuerzas repulsivas tienen dirección contraria al vector posición de cada obstáculo respecto al robot y una magnitud inversamente proporcional a la distancia

entre el robot y el obstáculo. De tal manera, que los obstáculos cercanos crean una fuerza repulsiva mayor que los que están más alejados. En el algoritmo original se consideraba una rejilla de certeza (certainty grid) que recogía la certeza del robot, determinada por los sensores de detección elegidos, de que haya un obstáculo en cada punto de la rejilla. De tal forma que para cada punto de la rejilla se calculaba su fuerza repulsiva correspondiente. La suma de todas las fuerzas repulsivas se llama la fuerza repulsiva total.

La fuerza resultante es la suma de la fuerza de atracción y la fuerza repulsiva total y proporciona al robot la dirección y la magnitud de la velocidad que debe tomar en cada momento. Por lo tanto, la fuerza atractiva termina la dirección ideal del robot para acercarse al objetivo de manera más directa y las fuerzas repulsivas modifican la dirección para evitar y rodear obstáculos de forma que se llegue al objetivo de forma segura.

A lo largo de los años se han desarrollado distintas versiones y mejoras respecto al algoritmo original y tradicional. Un que destaca es el algoritmo VFH, Vector Field Histogram [1], por los mismo autores un par de años después. Esto indica que los conceptos desarrollados en este algoritmo tuvieron repercusión en el ámbito de la planificación local en robótica.

Se ha elegido este algoritmo entre otros considerados por su completud, ya que englobaba los dos objetivos impuestos, y la robustez de su implementación. Además, consideramos que los resultados obtenidos cumplen con los deseado.

3.2. Implementación

3.2.1. Control de trayectorias

En primer lugar se va a describir como se manejan la información que recibe el controlador, en especial la trayectoria recibida.

Las trayectorias se definen en mensajes de tipo 'nav_msgs/Path', que son publicados en el topic /path cada 1 segundo por un publicador de trayectorias. El publicador de trayectorias usado es el proporcionado por los profesores de la asignatura.

Cada vez que se recibe una trayectoria, se guarda la trayectoria y el numero de

puntos de esta en una variable interna del controlador y se da como 'goal', punto objetivo actual, el primer punto de la trayectoria.

Por otro lado cada vez que se ejecuta el codigo del controlador, con una frecuencia de 20 Hz, se comprueba si se ha recibido algun 'goal' y además de si se ha alcanzado ya el objetivo. Esto ultimo lo realiza la función 'goalReached', que calcula la distancia entre el robot y el objetivo y comprueba si es menor a una tolerancia impuesta. La tolerancia del error en distancia elegida es de 0.25 m para los puntos intermedios o subobjetivos y 0.15 m para el objetivo final.

Cuando se alcanza un subobjetivo se actualiza la variable que guarda cual es el subobjetivo actual para pasarle al algoritmo el siguiente punto. Si el objetivo alcanzado es el final, se para el robot y se prepara para recibir otra trayectoria.

3.2.2. Creación de fuerzas virtuales

La **fuerza de atracción** se calcula con un parámetro de fuera constante y la dirección del vector posición del punto objetivo respecto al eje de coordenadas del robot.

Las **fuerza de repulsión** son calculadas utilizando como dirección la posición de los obstáculos, respecto al eje de coordenados del robot, negada y la magnitud un parámetro proporcional dividido por la distancia al objetivo. De tal forma que la fuerza indica la dirección opuesta del obstáculo y la magnitud es mucho mayor cuanto más cerca esta el obstáculo. La fuerza tiene un limite superior para evitar errores si la distancia es cero. Además, los obstáculos detectados están a una distancia máxima de 0.65 metros. Como se han detectados los obstáculos de describirá mas adelante. Estas fuerzas (el numero de fuerzas depende del numero de obstáculos detectado) se suman en una fuerza repulsiva total.

La **fuerza resultante** es la suma de la fuerza de atracción y la fuerza repulsiva total. Todas las fuerzas son vectores descritos respecto al eje de coordenadas 'base_footprint'.

3.2.3. Transformación de fuerza virtual resultante a velocidades

Como se ha mencionado antes, las señales de control que recibe el robot son la velocidad lineal, hacia el frente del robot (Eje x del eje 'base_footprint') y la velocidad

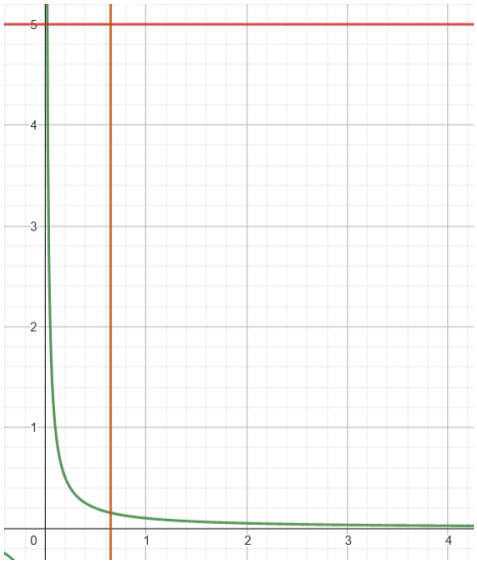

Figura 3: Magnitud de las fuerzas repulsivas (y) respecto a la distancia (x)

angular (respecto al eje z de los mismo ejes). El mapa considerado es en dos dimensiones por lo que el robot se mueve respecto al plano xy.

Como el vector de fuerza resultante esta respecto al eje de coordenadas 'base_footprint', todo lo descrito a continuación se hace en referencia a este eje.

La velocidad lineal es la proyección del vector fuerza resultante en el eje x, dirección al frente del robot.

La velocidad angular es calculada por un controlador PD (Proportional Derivative) discreto considerando el ángulo del vector fuerza. Este ángulo representa el error en ángulo de la orientación actual del robot y la orientación deseada para dirigirse de forma frontal a la dirección del vector fuerza. La parte derivativa se ha añadido para disminuir oscilaciones y suavizar la velocidad.

Ambas velocidades están saturadas por una velocidad máxima permitida.

Además,se ha considerado la situación donde el objetivo esta un poco por delante y a un lado del robot, donde en primer momento si la velocidad angular calculada no es suficiente el robot podría quedarse haciendo círculos o espirales alrededor del objetivo. Para evitar que ocurra esta situacción, se ha incluido un caso donde cuando la velocidad lineal es muy pequeña y la velocidad angular algo mayor, se anula la velocidad lineal para que el robot se posición de forma correcta.

3.2.4. Problemas y comentarios

Este algoritmo aunque la teoría de la implementación es relativamente sencilla, es muy sensible a los parámetros elegidos. Hay muchas consideraciones que tener en cuenta para determinarlos, entre otros: que la fuerza de atracción sea mayor que la repulsiva de normal pero que la repulsiva sea mayor cuando vaya a chocarse, que velocidad angular sea lo suficientemente rápida de reaccionar pero no ocurran oscilaciones, etc.

Además, es un algoritmo de planificación local sensible a mínimos locales por lo si el objetivo es muy lejano y se encuentra un obstáculo justo entre el robot y el objetivo, mínimo local, es probable que se queda un poco atascado. Hay maneras de lidiar con los mímicos locales, como se propone en el algoritmo original [2] cambiando a otro algoritmo distinto, pero no ha dado tiempo de implementar en ente proyecto.

4. Detección de obstáculos

A la hora de realizar la detección de obstáculos se tuvo en cuenta desde el principio cómo se debían publicar los obstáculos detectados, es decir, el tipo de mensaje. Para, posteriormente, en el nodo principal que usa el **algoritmo VFF**, poder usar las coordenadas de los obstáculos detectados (representando a la vez en **RVIZ** para tener *feedback* visual (véase Figura 3), los vectores que van del *frame "base_footprint"* a los obstáculos), se publican las coordenadas de estos en un mensaje personalizado llamado **Vector3Array**, que como su nombre indica, es una lista de mensajes llamada *"vectors"* tipo *Vector3* los cuales solo guardan coordenadas tipo (x,y,z).

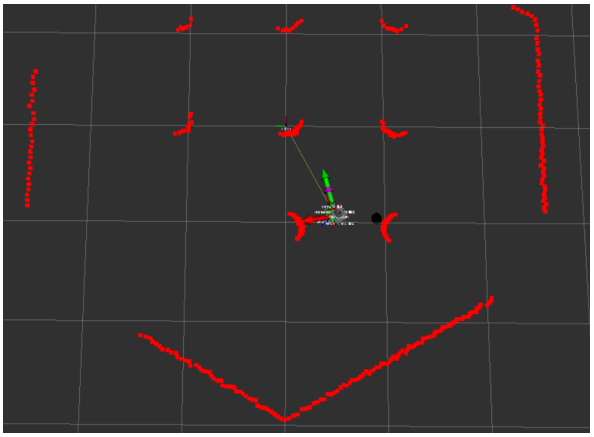

Figura 4: *Feedback* visual de la detección

Teniendo esto en cuenta, el procedimiento para detectar los obstáculos funciona de la siguiente manera. En primer lugar, los datos del *LiDar*, que son las distancias desde *"base_scan"* hasta los supuestos obstáculos, se guardan en una variable intermedia, auxiliar, según la región del espacio. Esto último es porque una vez se procesen los datos del sensor, se guardarán en un diccionario propio de la clase **Detector**, que es en la que ocurre todo. Este diccionario llamado **self.Regions_Report** tiene 12 elementos que representan 12 regiones del espacio alrededor del sensor si dividimos $360^o/30^o$, y cada *"value"* representaría las coordenadas (x,y,z) de los obstáculos y el valor mínimo de las distancias de cada región: (x,y,z,min_value). Continuando con esta primera parte, una vez se guardan los datos en esa variable intermedia para cada región, se filtra y se guardan solo los valores menores o iguales a una distancia máxima a los obstáculos y que no sean infinito. Una vez filtrado, se extrae el mínimo de cada región, sin embargo, en caso de que la lista esté vacía porque no se cumplieran las condiciones del filtro, se queda el *min_value* como 0, igual que el ángulo. Si sí se había encontrado un mínimo, obtenemos el ángulo para esa distancia con:

angle = next(i for i, v in enumerate(scan.ranges) if v == min_value);

y se calculan las coordenadas de los obstáculos con la proyección de estas distancias en los ejes X e Y (en Z sería 0); guardando ya en el diccionario mencionado estos valores junto con el mínimo.

En segundo y último lugar, se transforman todos los puntos guardados en el diccionario (ósea, se recorre un bucle y se comprueba primero que las listas tengan información en cada región para que no haya fallos en los cálculos, y que el min_value sea mayor que 0 por lo explicado antes) de *"base_scan"* a *"base_footprint"*, que se toma como la base del robot, usando un objeto tipo *PoseStamped* para ello. A continuación se extraen las coordenadas de los puntos y se añaden a la lista del tipo de mensaje personalizado (**Vector3Array**), que se ha llamado en la clase: **self.obs**. Una vez añadidos todos los obstáculos detectados válidos a la lista del mensaje, se publica en el *topic* **obstacles**, y se vacía la lista para que la siguiente vez que se reciben datos del *LiDar* se rellene con lo nuevo, que en caso de que el robot no se haya movido, podría ser lo mismo, pero se hace así para no tener que comprobar si lo que estaba guardado es igual que lo nuevo.

4.1. A tener en cuenta

En el código presentado y entregado, los puntos detectados en primer lugar y guardados en el diccionario según el *frame* del *LiDar*, en vez de pasarse directamente a *"base_footprint"*, se pasan a *"odom"* y ya posteriormente al *frame* del robot. Esto a priori es un gasto computacional innecesario, pero se ha puesto así porque antes el algoritmo comprobaba la distancia entre los obstáculos nuevos y los antiguos para saber si eran o no los mismos, y para ello era necesario que estuvieran en *"odom"*. Sim embargo, por si se quisieran hacer ampliaciones y añadir funcionalidades de este tipo y dado que al final son muy pocos los puntos que se transforman, se ha dejado al ver que simple su función de forma correcta.

Por otro lado, por simplicidad, se han reducido las regiones de interés para detectar obstáculos las de delante del **TurtleBot3**, que irían de [-75,75]º. Esto además hace que la complejidad computacional de las transformaciones de *frame* se reduzca más si cabe.

5. Conclusión

Se ha implementado un algoritmo de navegación reactivo que es capaz de seguir una trayectoria o dirigirse a un punto mientras esquiva obstáculos, además de la detección de obstáculos como tal.

Los resultados conseguidos son satisfactorios. Sin obstáculos, se dirige al objetivo de forma relativamente rápida y sigue la trayectoria de forma más o menos fiable y de forma suave. Con los obstáculos, los resultados son mixtos, a veces se consigue un movimiento suave y otras se frena demasiado al detectar un obstáculo cerca.

Se han añadido algunos videos con los resultados obtenidos.

De nuevo se menciona que el algoritmo es sensible a mínimos locales y puede haber situaciones en las que le cueste decidir donde moverse.

5.1. A tener en cuenta para probarlo

Ambos nodos se ejecutan al lanzar el launch 'VFF.launch' incluido en el paquete.

Además, se puede visualizar las fuerzas en tiempo real si se añade el topic 'visualization_marker' a la configuración de rviz. La fuerza de atración se pinta en verde, las repulsivas en rojo y la fuerza resultante en morado.

Referencias

[1] Johann Borenstein, Yoram Koren et al. “The vector field histogram-fast obstacle avoidance for mobile robots”. En: *IEEE transactions on robotics and automation* 7.3 (1991), págs. 278-288.

[2] Johann Borenstein y Yorem Koren. “Real-Time Obstacle Avoidance for Fast Mobile Robots”. En: *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS* 19.5 (1989).