PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN
IIC-2685 ROBÓTICA MÓVIL

LAB 1: OPERACIONES BÁSICAS DE MOVIMIENTO Y PERCEPCIÓN EN TURTLEBOT

Fecha de entrega: lunes 26 de Abril de 2021

Conociendo a TurtleBot

En este primer laboratorio aprenderemos y pondremos en práctica las destrezas básicas de movimiento y percepción en los TurtleBots. El objetivo de esta experiencia es programar herramientas de software básicas que permitan al robot moverse en el espacio y percibir la configuración del entorno a través de la detección de obstáculos.

Como ya es sabido por todos, debido a la situación actual de pandemia que vive el mundo, no podremos interactuar con los TurtleBots físicos disponibles en el laboratorio. No obstante, esto no será un problema ya que contamos con un simulador [1] que virtualiza sus principales características, permitiéndonos trabajar a nuestro ritmo y desde la comodidad de nuestros hogares.

Antes de comenzar sus desarrollos, tengan presente que sus programas deben tener una estructura modular que haga posible identificar rápidamente posibles fuentes de errores. Para ello es importante definir la responsabilidad y alcance que tendrá cada tarea de alto nivel (ej: percepción, movimiento, etc.), y luego de esto, definir cuál es la forma más adecuada de implementarla (ej: función, clase, nodo ROS, etc.). Con respecto al lenguaje de programación a utilizar, pueden elegir el que más le acomode: Python (2.7+) o C++. Aún cuando es posible utilizar ambos lenguajes para programar distintos nodos sin tener problemas de comunicación entre ellos, se recomienda escoger solo uno y usarlo para todos sus módulos.

El día de la entrega, cada grupo tendrá un máximo de 20 minutos para presentar sus diseños, resultados y desmostraciones prácticas. Para ello, se enviará con anterioridad un formulario con los bloques disponibles, los que serán asignados por orden de llegada. Además, todos los paquetes desarrollados para resolver este laboratorio, deberán ser enviados en un archivo comprimido a través de Canvas.

Dentro de sus slides deberán incluir como mínimo: un diagrama con el diseño del software a nivel de nodos, los resultados de las mediciones solicitadas en el presente enunciado y sus conclusiones. El día de la presentación, el profesor eligirá de forma aleatoria a un miembro del grupo para presentar las slides, y a otro para hacer la demostración práctica, por lo tanto, cada integrante deberá conocer en detalle ambas secciones. Además, se solicita que en la medida de lo posible, todos los integrantes del grupo tengan prendidas sus cámaras.

1. Programando movimientos de TurtleBot

Como primera tarea, deberán ser capaces de dar instrucciones de movimiento a los TurtleBots. Los tópicos involucrados son los siguientes:

- `/yocs_cmd_vel_mux/input/navigation`: tópico para publicar mensajes de velocidad para que los ejecute el robot.
- `/odom`: tópico para leer la odometría del robot.

A lo largo de todas nuestras experiencias, usaremos el modelo de comunicación *publicar/subscribir* definido por ROS, lo cual nos ayudará a programar códigos altamente modulares. Para comenzar deberán crear

un nodo llamado `dead_reckoning_nav`, en el cual implementarán un sistema de navegación basado en una técnica conocida como *navegación por estima* (o *dead reckoning* en inglés). Esta antigua técnica tiene por objetivo realizar una estimación de la posición actual del móvil en base al punto de partida, dirección de movimiento y velocidades empleadas. En nuestro caso particular, para alcanzar un punto de destino en el espacio, estableceremos una velocidad fija y calcularemos el tiempo que ésta debe ser aplicada para recorrer la distancia que lleve al robot a la coordenada deseada.

Para su funcionamiento, el nodo `dead_reckoning_nav` deberá implementar las siguientes funciones o métodos:

`aplicar_velocidad(vel_lineal, vel_angular, tiempo)` Función que recibe como argumento una lista de velocidades lineales, una lista de velocidades angulares y una lista de tiempos de duración para cada ítem de velocidad. Esta función les servirá para especificar una trayectoria entre dos puntos.

En la figura 1, se muestra un ejemplo de como alcanzar la pose de destino (x_1, y_1, θ_1) a través de una lista de velocidades y tiempos. En este caso, se aplicará la siguiente secuencia de velocidades: I) velocidad lineal v_0 por t_0 [s], II) velocidad angular ω_1 por t_1 [s], y finalmente, III) velocidad lineal v_2 por t_2 [s].

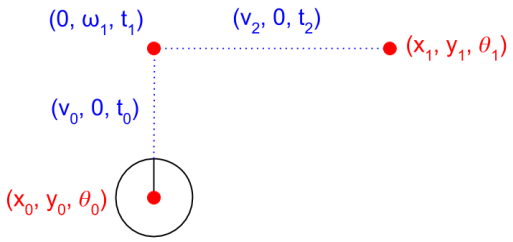

Figura 1: Lista de velocidades (lineal y angular) y tiempos de aplicación para ir de la pose (x_0, y_0, θ_0) a la pose (x_1, y_1, θ_1).

`mover_robot_a_destino(goal_pose)` Función que ordena al robot moverse a una *pose* de destino relativa a la *pose* actual. Tenga en cuenta que una *pose* queda completamente definida al definir todas las dimensiones en las que puede moverse el robot, es decir, (x, y, θ). El argumento `goal_pose` es una lista de poses (x, y, θ). Por el momento, utilizaremos valores de velocidad lineal y angular fijas en 0.2 [m/s] y 1.0 [rad/s] respectivamente. Por lo tanto, su función deberá calcular los tiempos necesarios para llegar a la(s) pose(s) especificada(s), en forma secuencial. Use la función `aplicar_velocidad` para ejecutar las listas de velocidades y tiempos que permiten llegar a cada *pose* de la lista. Esta función se actualizará en futuros laboratorios agregando control de bajo nivel y chequeo de *pose*.

`accion_mover_cb` Esta función debe actuar como *callback* de un tópico que reciba la lista de poses y mueva al robot en la secuencia especificada. En paralelo deberá programar un nodo que lea la lista de poses desde un archivo y las envíe al nodo `dead_reckoning_nav` a través de un tópico. El nombre del tópico debe ser definido como `/goal_list`.

Se recomienda que la lista de *poses* sea enviada en un mensaje de tipo `geometry_msgs/PoseArray`.

Una vez programado el nodo `dead_reckoning_nav`, deberá realizar la siguiente actividad y tomar las mediciones que se solicitan.

Actividad 1: Avanzar y Rotar

Programar un comportamiento que permita a los TurtleBots realizar tres vueltas siguiendo una trayectoria cuadrada con una arista de 1 [m]. Esto debe hacerlo enviando un arreglo con la lista de *poses* al tópico `/goal_list`, el cual debe indicar las *poses* correspondientes a los vértices del cuadrado y los ángulos para iniciar el siguiente movimiento. Mientras realiza el movimiento, registre las coordenadas (x, y) *reales* que

describen la trayectoria leyendo el tópico `/real_pose`, y las coordenadas calculadas por la odometría a través del tópico `/odom`. Realicen un gráfico con sus resultados y calculen el error promedio (considerando las tres vueltas) de la distancia entre el punto final al punto inicio. Intenten encontrar factores de corrección de rotación tal que el punto de inicio y el punto final estén cercanos (por ejemplo a 10 [cm] o menos).

Debido a que el simulador intenta replicar las condiciones físicas del robot y ambiente real, los resultados que obtendrán no serán ideales ni exactos.

Incluir en presentación: error promedio, desviación estándar, factor de corrección encontrado, explicar fenómeno observado y reflexionar acerca del por que éste se puede producir en la realidad (razones de software y hardware).

Demostración: Movimiento en cuadrado con y sin factor de corrección.

Incluir en paquete: archivo launch de nombre *avanzar_y_rotar.launch* que permita iniciar tanto los nodos que implementan su rutina, como los nodos que implementan el simulador. Deben garantizar que sea posible correr el experimento completo simplemente ejecutando:

$roslaunch < nombre\_paquete > avanzar\_y\_rotar.launch$

2. Dotando al TurtleBot de percepción básica

Hasta ahora, el nodo `dead_reckoning_nav` no tiene la capacidad de reaccionar al mundo que lo rodea. En esta parte dotaremos a los TurtleBots de un módulo de percepción básico usando el sensor *Kinect*, el cual se basará en la detección de obstáculos usando la imagen de profundidad.

El tópico a utilizar en esta sección es:

- `/camera/depth/image_raw`: tópico en el cual se publica la imagen de profundidad del sensor *Kinect*.

Para esta parte deberá programar el siguiente nodo:

`obstacle_detector` Usando la imagen de profundidad, el nodo debe publicar en forma continua (rate de 5 [Hz]) la ausencia o presencia de algún objeto que puede interferir el avance del robot, junto con la ubicación (izquierda, derecha o centro) del obstáculo. En Python, se recomienda encarecidamente usar funciones de la librería *NumPy* para el procesamiento eficiente de las imágenes. El nodo debe suscribirse al tópico de la imagen de profundidad, y publicar el estado actual mediante un mensage de tipo `std_msgs/String`, el cual podrá tomar uno de los siguientes valores: *obstacle_left*, *obstacle_right*, *obstacle_center*, *free*. Por ejemplo, si se detecta un obtáctulo en la zona izquierda de la imagen, el nodo deberá publicar el string *obstacle_left*. Puede suponer que no hay obstáculo si el robot tiene camino libre hacia adelante a una distancia de 50 [cm]. El nombre del tópico donde publicará el estado deberá ser definido como: `/occupancy_state`.

Incluir en presentación: breve explicación del criterio utilizado para la detección.

Demostración: debe agregar y eliminar paredes mostrando en el terminal la detección realizada.

Incluir en paquete: archivo launch de nombre *obstacle_detector.launch* que permita iniciar tanto los nodos que implementan la detección como los nodos que implementan el simulador. Deben garantizar que sea posible correr el experimento completo simplemente ejecutando:

$roslaunch < nombre\_paquete > obstacle\_detector.launch$

3. Uniendo acción y percepción

Para lograr el objetivo final de este laboratorio, uniremos acción y percepción para lograr evitar colisiones. Para ello deberá modificar el nodo `dead_reckoning_nav` y agregar la capacidad de recibir mensajes de detección de obstáculos desde el tópico `/occupancy_state`. Al recibir un estado del tipo *"free"*, el robot deberá seguir su trayectoria de forma normal. Por otra parte, si el nodo `dead_reckoning_nav` recibe un mensaje desde el nodo `obstacle_detector` indicando la presencia de un obstáculo (*obstacle_left*, *obstacle_right* u *obstacle_center*), deberá detener el movimiento del robot e indicar por los parlantes en que

posición se encuentra el obstáculo (*obstacle left*, *obstacle right* u *obstacle center*). Una vez que el obstáculo sea retirado, el robot deberá continuar con su movimiento hasta alcanzar su objetivo. Se recomienda modificar la función `aplicar_velocidad` para congelar su operación al detectar algún obstáculo, y luego reanudarla al detectar el camino libre.

Para darle a nuestro robot la capacidad del habla, utilizaremos el paquete de ROS *sound_play* [2]. Éste puede ser instalado por medio del gestor de paquetes de Ubuntu (apt) de la siguiente forma:

sudo apt install ros-$ROS_DISTRO-sound-play

Luego de esto podrá levantar los nodos de este paquete por medio del tag include, tal como se muestra a continuación:

< include file="$(find sound_play)/soundplay_node.launch" />

Demostración: debe agregar y eliminar paredes mostrando que el robot se detiene y avanza respectivamente. Cada vez que el robot detecte un obstáculo, se deberá escuchar la señal de audio descrita más arriba.

Incluir en paquete: archivo launch de nombre *accion_y_percepcion.launch* que permita iniciar tanto los nodos que implementan el comportamiento para evitar colisiones como los nodos que implementan el simulador. Deben garantizar que sea posible correr el experimento completo simplemente ejecutando:

$roslaunch < nombre\_paquete > accion\_y\_percepcion.launch$

Referencias

[1] https://github.com/gasevi/very_simple_robot_simulator

[2] http://wiki.ros.org/sound_play