Universidad de San Andrés, Depto. de Ingeniería

Práctica 6 - ICP y Planeamiento de Trayectorias

I-402 - Principios de la Robótica Autónoma

Prof. Ignacio Mas, Tadeo Casiraghi y Bautista Chasco

10 de noviembre de 2025

Fecha límite de entrega: 27/11/25, 23.59hs.
Modo de entrega: Enviar por el Aula Virtual del Campus **en un solo archivo comprimido** el código comentado y los gráficos (.jpg ó .pdf) y/o animaciones.

1. Implementación de algoritmo de asociación de datos

El algoritmo *Iterative Closest Point* (ICP) es una técnica ampliamente utilizada en robótica y visión para el registro de nubes de puntos. Su objetivo es encontrar la transformación (rotación y traslación) que mejor alinea dos conjuntos de puntos en el espacio. El proceso consiste en iterar entre dos pasos principales: primero, establecer correspondencias entre los puntos de ambos conjuntos (normalmente eligiendo los más cercanos), y segundo, estimar la transformación que minimiza el error cuadrático medio entre ellos. Este ciclo se repite hasta alcanzar la convergencia, es decir, hasta que la mejora entre iteraciones sea insignificante.

1.1. Implementación en Python

En este ejercicio se implementará una técnica de asociación de datos. Los archivos incluidos en esta práctica implementan los pasos básicos del algoritmo ICP (Iterative Closest Point). En este algoritmo, eligiendo el conjunto de puntos a pasar como parámetro de la función `icp` en el script `icp_framework.py` se pueden probar 4 conjuntos de datos (P1, P2, P3 y P4). El algoritmo funciona correctamente con los datos con correspondencias conocidas (i.e. P1 y P2), pero no funciona para conjuntos de datos con correspondencias desconocidas (i.e. P3 y P4). Si las correspondencias no son conocidas, deben ser estimadas antes de aplicar el algoritmo ICP.

Implementar el método de asociación de datos que considera correspondencias por puntos más cercanos en la función `closest_point_matching` y verificarlo usando los conjuntos de datos P3 y P4.

1.2. Aplicación en ROS

En esta segunda parte se implementará una versión práctica del algoritmo ICP dentro del entorno de Ros Humble, utilizando el robot TurtleBot3. El objetivo será estimar la odometría filtrada a partir de los datos del sensor láser, aplicando ICP sobre el tópico `/scan`. La información procesada permitirá una estimación alternativa de la pose del robot, reduciendo el error acumulado de la odometría interna. Para la visualización y validación del resultado, se utilizará `RViz2`, donde se analizará la convergencia del algoritmo a partir de la alineación progresiva entre los escaneos consecutivos del entorno. Este paquete se desarrolla en C++ por motivos de cómputo, un ejemplo simple de desarrollo en este lenguaje se puede encontrar en [Writing a Simple C++ Publisher and Subscriber] (https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html). Se deberá modificar el callback de Scan en el código para procesar el mensaje por ICP.

Para lanzar los paquetes se debe correr:

- Terminal 1: `ros2 launch turtlebot3_icp_localization icp_localization.launch.py`
- Terminal 2: `ros2 launch turtlebot3_custom_simulation custom_room.launch.py`
- Terminal 3: `ros2 run turtlebot3_teleop teleop_keyboard`

De forma totalmente análoga a los trabajos prácticos de filtrado anteriores.

Para tener acceso a los puntos del mensaje dentro del callback del tópico `/scan`, se puede iterar sobre la nube de puntos de la siguiente manera:

```
for (const auto& point : current_cloud->points)
{
    float x = point.x;
    float y = point.y;
    float z = point.z;

    // Aquí puedes usar los valores, imprimirlos, etc.
    std::cout << "Punto: x=" << x << " y=" << y << " z=" << z << std::endl;
}

// También se puede acceder directamente a un punto específico:
float x = current_cloud->points[i].x;
float y = current_cloud->points[i].y;
float z = current_cloud->points[i].z;
```

Nota: El algoritmo puede diverger con el tiempo, el criterio de corrección del TP es en función a la correctitud en el criterio de ICP y sus respectivos parámetros. No se permite el uso del paquete de ICP pero sí el uso de paquetes de rotación matricial por SVD.

2. Planeamiento de caminos

Los algoritmos de búsqueda en grafos como Dijkstra o A* pueden ser usados para planear caminos en grafos desde un lugar de inicio hasta un objetivo. Si las celdas de un mapa de grilla se representan como nodos conectados con sus celdas vecinas, estos algoritmos pueden aplicarse directamente para realizar planeamiento para robots. Para este ejercicio, consideramos las 8 celdas vecinas de una celda $\langle x, y \rangle$, que se definen como las celdas adyacentes a $\langle x, y \rangle$ horizontalmente, verticalmente y en diagonal.
El archivo incluido contiene una implementación de planeamiento en 2-D basado en grafos. El script `planning_framework.py` contiene el algoritmo y es el que debe ejecutarse. Los ejercicios de esta sección se realizan implementando las funciones vacías o incompletas que acompañan al script principal.

2.1. Algoritmo de Dijkstra

El algoritmo de Dijkstra se usa para calcular caminos de costo mínimo dentro de un grafo. Durante la búsqueda, siempre se busca el nodo del grafo con el menor costo desde el punto de inicio y se agregan los nodos vecinos al grafo de búsqueda.

1. Sea $M(x, y)$ un mapa de grilla de ocupación. Durante la búsqueda, las celdas se conectan con sus celdas vecinas para construir el grafo de búsqueda. Completar la función `neighbors` provista que define los vecinos de una celda. La función toma como entrada las coordenadas de una celda y el tamaño del mapa, y devuelve un vector de $n \times 2$ con las coordenadas de sus celdas vecinas, teniendo en cuenta los límites del mapa.

2. Implementar una función para los costos de un arco entre nodos que permita planear caminos de mínima longitud y libre de colisiones. Considerar la celda como un obstáculo si su probabilidad de ocupación supera cierto umbral. ¿Qué umbral se debería elegir? Implementar la función `edge_cost`.

3. Incluir información de ocupación en la función de costo que permita que el algoritmo elija celdas con baja probabilidad de ocupación sobre celdas con mayor probabilidad de ocupación.

4. Implementar el paso de actualización de costos. Para cada nodo *parent*, actualizar el costo de sus *childs* si el costo es menor a través de dicho *parent*.

2.2. Algoritmo $A^\star$

El algoritmo $A^\star$ utiliza una heurística para realizar una búsqueda informada que resulta ser más eficiente que el algoritmo de Dijkstra.

1. ¿Qué propiedades debe tener dicha heurística para asegurar que $A^\star$ es óptimo?

2. Definir una heurística para planeamiento de robots móviles en 2-D. Completar la función `heuristic` provista. La función toma como entrada las coordenadas de una celda y del objetivo, y devuelve el costo estimado hasta el objetivo.

3. ¿ Qué pasa si se aumenta la heurística usando h_2, siendo h_2 un múltiplo de la heurística h definida en el punto anterior. Analizar el comportamiento con diferentes factores: $h_2 = \{1; 2; 5; 10\} * h$