UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"SEGUIMIENTO DE TRAYECTORIAS GPS CON UN ROBOT MÓVIL USANDO NUBES DE PUNTOS LIDAR"

TRABAJO FIN DE GRADO

Junio -2024

AUTOR: Víctor Márquez Fernández

DIRECTOR/ES: Arturo Gil Aparicio

ÍNDICE

1. INTRODUCCIÓN

1.1 Objetivos

El objetivo principal de este Trabajo de Fin de Grado (TFG) es desarrollar un sistema de seguimiento de trayectorias a partir de coordenadas GPS para un robot móvil utilizando nubes de puntos LiDAR. Este sistema permitirá al robot navegar de manera autónoma en entornos complejos, garantizando una alta precisión en la detección de obstáculos y la planificación de rutas. Para lograr esto, se plantean los siguientes objetivos específicos:

- Integración de sensores LiDAR y GPS
- Desarrollo de algoritmos de planificación de trayectorias
- Implementación del robot Husky al software
- Simulación y validación en entornos virtuales
- Evaluación y análisis de los resultados

1.2 La robótica móvil

La robótica móvil es una rama de la robótica que se centra en el diseño, desarrollo y aplicación de robots que son capaces de moverse por su entorno. Estos robots están equipados con varios tipos de mecanismos de movilidad, como ruedas, orugas o patas, habilitándolos a navegar a través de diferentes terrenos y realizar tareas automáticamente o controlados por humanos.

Figura 1 De izquierda a derecha: Robot móvil con ruedas, robot móvil oruga, robot cuadrúpedo.

A parte de estar dotados con mecanismos de movilidad, los robots móviles se caracterizan por tener sensores para poder detectar su entorno, dispositivos de localización y *mapping,* algoritmos de navegación y control de movimiento y, con inteligencia artificial y *machine learning* lo que nos lleva al concepto de robot móvil inteligente.

Un robot móvil inteligente está equipado con avanzadas capacidades sensoriales, toma de decisiones e interacción con el entorno lo que le permite realizar tareas complejas autónomamente o con poca intervención humana. Estos robots utilizan inteligencia artificial y *machine learning* para adaptarse a entornos dinámicos, aprender de experiencias y mejorar su rendimiento con el tiempo.

Las aplicaciones de los robots móviles son varias, pero se pueden destacar las siguientes:

- **Automatización industrial:** Los robots móviles se utilizan para el transporte de materiales y la asistencia en líneas de producción.

Figura 2 Robot móvil CHEKOV de Magna Power.

- **Robots sanitarios:** Robots que envían medicación y suministros en los hospitales.

Figura 3 Robot móvil TUG.

- **Robots agrícolas:** Robots utilizados para tareas como la supervisión de cultivos y la cosecha.

Figura 4 Robot móvil Agrobot E-series.

- **Robots serviciales:** Utilizados para la atención al cliente en hoteles y restaurantes.

Figura 5 Robot móvil Bellabot.

- **Robots exploradores:** Utilizados para explorar entornos peligrosos o inaccesibles, recopilando datos y realizando tareas críticas.

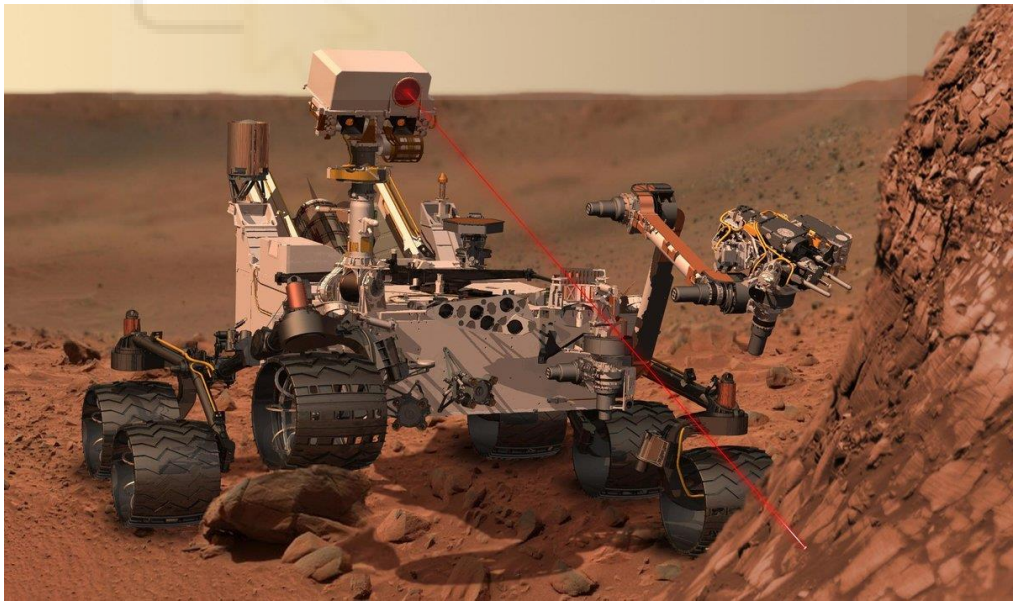


Figura 6 Robot móvil Cursiosity.

- **Robots de seguridad:** patrullan zonas, detectan intrusiones y vigilan posibles amenazas mejorando la seguridad en diversos entornos.

Figura 7 Robot móvil Nightscope K5.

1.3 Navegación

La navegación en robótica se refiere al proceso por el cual un robot determina y sigue una trayectoria de un lugar a otro dentro de su entorno. Esto implica una combinación de varias tecnologías y algoritmos para lograr tareas como la localización, *mapping*, la planificación de trayectorias y la evitación de obstáculos. La navegación es un aspecto crítico de la robótica, que permite a los robots realizar operaciones autónomas en diversos entornos, desde espacios interiores estructurados a terrenos exteriores no estructurados.

Las componentes más importantes de la navegación son las siguientes:

- **Localización:** Es el proceso de determinar la posición de un robot en un entorno determinado. Puede ser absoluta, proporcionando las coordenadas exactas del robot, o relativa, indicando la posición del robot respecto a su punto de partida u otros puntos de referencia. Los métodos más comúnmente utilizados son los siguientes:
    - **Odometría:** Utiliza datos de sensores de movimiento para estimar los cambios de posición a lo largo del tiempo.
    - ***Inertial Measurement Units* (IMU):** Realizan un seguimiento de la aceleración y los índices de rotación del robot.
    - ***Global Positioning System* (GPS):** Proporciona datos de posición precisos en entornos exteriores.

- ***Mapping*:** Consiste en crear una representación del entorno en el que opera el robot. Este mapa puede utilizarse para la navegación y la evitación de obstáculos. Algunas de las técnicas utilizadas son:
  - **Mapas basados en cuadrículas:** Dividen el entorno en una cuadrícula en la que cada celda representa un espacio libre, un obstáculo o un espacio desconocido.
  - **Mapas topológicos:** Representan el entorno como un grafo de nodos (puntos de referencia) y aristas (caminos entre puntos de referencia).
  - **Mapas 3D:** Utilizan datos de sensores como LIDAR o cámaras de profundidad para crear representaciones tridimensionales del entorno.

Junto con la localización y el *mapping* otros aspectos importantes son el *path planning* y la evitación de obstáculos, en ambos se entrará en detalle más adelante dada su importancia en el proyecto.

1.4 Sensores utilizados en robótica móvil

Los sensores desempeñan un papel crucial en la robótica móvil, ya que permiten a los robots percibir su entorno e interactuar con él. La elección de los sensores depende de la aplicación específica, el entorno y los requisitos del robot. Se indagará en los diferentes tipos de sensores, profundizando en los más utilizados durante el proyecto como el GPS y el LiDAR.

Sensores de proximidad:

- **Sensores de infrarrojos (IR):** Utilizan luz infrarroja para detectar objetos. Son baratos y útiles para la detección de corto alcance, pero son sensibles a la luz ambiente y a las superficies reflectantes.
- **Sensores ultrasónicos:** Emiten ondas ultrasónicas y miden el tiempo que tarda el eco en volver. Son buenos para detectar objetos a varios alcances y funcionan bien en diferentes condiciones de iluminación, pero pueden verse afectados por superficies blandas que absorben el sonido.
- **LIDAR (Light Detection and Ranging):** Es una tecnología de teledetección que utiliza luz en forma de un láser pulsado para medir distancias variables hasta la superficie. Esta tecnología genera datos tridimensionales precisos de la superficie

de la escena y sus características. LiDAR es ampliamente utilizado en topografía, cartografía, silvicultura, geología, arqueología, y muchas otras áreas.

Figura 8 Sensores LiDAR.

El funcionamiento de LiDAR se basa en medir el tiempo que tarda un pulso de luz en viajar desde el emisor, reflejarse en un objeto y regresar al receptor. Este tiempo de vuelo se usa para calcular la distancia entre el sensor y el objeto. El proceso básico es el siguiente:

1. **Emisión del Pulso**: El emisor de láser genera un pulso de luz, que es dirigido hacia el objeto o superficie que se va a medir.
2. **Reflexión del Pulso**: Cuando el pulso de luz alcanza la superficie del objeto, parte de la luz es reflejada de vuelta hacia el sensor.
3. **Detección del Pulso**: El receptor del LiDAR detecta el pulso de luz reflejado y mide el tiempo que tardó en regresar al sensor.
4. **Cálculo de la Distancia**: Usando la fórmula $Distancia = \frac{c \cdot t}{2}$ donde c es la velocidad de la luz y t es el tiempo de vuelo medido, el sistema calcula la distancia entre el sensor y el objeto.
5. **Georreferenciación**: Combinando los datos de distancia con la información de posición proporcionada por el GPS y la orientación proporcionada por la IMU, el sistema puede determinar la ubicación precisa en el espacio de cada punto medido.

Sensores de visión:

- **Cámaras:** Captan información visual del entorno. Pueden ser monoculares (una sola cámara) o estereoscópicas (dos cámaras para la percepción de la profundidad). Las cámaras son versátiles y proporcionan muchos datos, pero

requieren una gran capacidad de procesamiento y algoritmos sofisticados para su interpretación.

- **Cámaras de profundidad:** Combinan la imagen tradicional con la detección de profundidad. Estos sensores proporcionan información 3D sobre el entorno y son útiles para aplicaciones como el reconocimiento de objetos y la navegación.

Telémetros:

- **Radar:** Utiliza ondas de radio para detectar objetos y medir su distancia. Es menos habitual en robótica móvil pero útil en entornos con poca visibilidad.

Inertial Measurement Units (IMU):

- **Acelerómetros:** Miden la aceleración lineal a lo largo de uno o varios ejes. Ayudan a determinar el movimiento y la orientación del robot.
- **Giroscopios:** Miden la velocidad de rotación. Son esenciales para mantener la orientación y la estabilidad.
- **Magnetómetros:** Miden los campos magnéticos y pueden utilizarse como brújulas digitales para la orientación.

Sensores de posicionamiento:

- ***Encoder*s:** Miden la rotación de ruedas o articulaciones, proporcionando información sobre la posición y el movimiento del robot. Son fundamentales para la odometría, que estima el cambio de posición del robot a lo largo del tiempo.
- **GPS (Global Positioning System):**

El Sistema de Posicionamiento Global es una red de satélites que proporcionan datos de ubicación, navegación y cronometraje a receptores en la Tierra. Originalmente desarrollado por el Departamento de Defensa de los Estados Unidos para aplicaciones militares, el GPS se ha convertido en una herramienta esencial para diversas aplicaciones civiles y comerciales, desde la navegación en automóviles hasta la sincronización de redes eléctricas y telecomunicaciones.

Estructura del sistema GPS

El sistema GPS consta de tres segmentos principales:

1. **Segmento espacial**: Consiste en una constelación de al menos 24 satélites (con un total operativo cercano a 31 para redundancia) que orbitan la Tierra a una

altitud de aproximadamente 20,200 kilómetros. Estos satélites están distribuidos en seis planos orbitales inclinados en aproximadamente 55 grados respecto al ecuador, lo que asegura que al menos cuatro satélites sean visibles desde cualquier punto de la Tierra en cualquier momento.

2. **Segmento de control**: Este segmento está compuesto por estaciones de monitoreo y control ubicadas en diversas partes del mundo. Las estaciones de monitoreo rastrean los satélites GPS, verifican la integridad de las señales y actualizan los datos orbitales y temporales que los satélites transmiten a los usuarios.
3. **Segmento de usuario**: Incluye receptores GPS utilizados por civiles, militares y aplicaciones comerciales. Estos receptores pueden variar desde dispositivos pequeños como relojes inteligentes y teléfonos móviles hasta equipos más sofisticados en aeronaves, barcos y vehículos militares.

Funcionamiento del GPS

El funcionamiento del GPS se basa en la trilateración, un proceso que determina la posición del receptor midiendo su distancia a varios satélites. A continuación, se detalla cómo se realiza este proceso:

1. **Transmisión de señales**: Cada satélite GPS transmite continuamente una señal de radio que incluye:
    - Una marca temporal precisa.
    - Datos sobre la posición orbital del satélite (efemérides).
    - Información sobre el estado del satélite (almanac).
2. **Recepción de señales**: Un receptor GPS en la Tierra capta las señales de al menos cuatro satélites. Dado que las señales viajan a la velocidad de la luz, hay un retraso entre el momento en que la señal es enviada por el satélite y el momento en que es recibida por el receptor. Este retraso se utiliza para calcular la distancia al satélite.
3. **Cálculo de distancias**: El receptor calcula la distancia a cada satélite multiplicando el tiempo que tarda la señal en llegar por la velocidad de la luz. Con la distancia a cuatro satélites conocidos, el receptor puede determinar su posición en tres dimensiones (latitud, longitud y altitud) mediante el proceso de trilateración.
4. **Corrección de errores**: Existen varios factores que pueden afectar a la precisión del GPS, incluyendo:

- Errores en el reloj del receptor.
- Retrasos en la señal causados por la ionosfera y la troposfera.
- Errores orbitales (desviaciones de los satélites de sus trayectorias previstas).
- Señales reflejadas que pueden causar errores en la recepción.

Para minimizar estos errores, el sistema GPS emplea técnicas de corrección como:

- **Corrección Diferencial GPS (DGPS)**: Utiliza estaciones terrestres de referencia cuya posición es conocida para corregir los errores en las señales GPS.

Aunque el GPS es una tecnología ampliamente utilizada y generalmente confiable, existen varias situaciones en las que su funcionamiento puede verse comprometido, llevando a errores o a la imposibilidad de determinar una ubicación precisa. A continuación, se describen algunas de las principales situaciones problemáticas para los GPS:

1. Interferencia de señal

- **Interferencia ionosférica y troposférica**: Las capas superiores de la atmósfera pueden afectar a la señal GPS. La ionosfera, cargada de partículas ionizadas, puede causar retrasos en la señal, especialmente durante las tormentas solares. La troposfera, con sus variaciones de temperatura y humedad, también puede afectar la precisión.
- **Interferencia Electromagnética (EMI)**: Dispositivos electrónicos y transmisores de alta potencia pueden generar interferencias electromagnéticas que degradan la señal GPS. Equipos como radios, radares, y ciertos tipos de equipos industriales pueden causar este tipo de interferencia.

2. Obstrucción de Señal

- **Entornos urbanos (Efecto Caño)**: En ciudades con edificios altos y estructuras densas, las señales GPS pueden ser bloqueadas o reflejadas, creando múltiples trayectorias que confunden al receptor. Esto puede causar grandes errores de posicionamiento.
- **Entornos subterráneos y bajo techo**: En túneles, estacionamientos subterráneos, edificios y otros entornos cubiertos, las señales GPS no pueden penetrar adecuadamente, lo que impide que los receptores obtengan una señal suficiente para calcular la posición.
- **Áreas densamente boscosas**: Las hojas y ramas de los árboles pueden bloquear y atenuar las señales GPS, especialmente en bosques densos o selvas.

3. Condiciones ambientales extremas

- **Tormentas solares**: Las explosiones de partículas cargadas provenientes del sol pueden interferir significativamente con las señales GPS, causando errores temporales en la navegación y en los sistemas de sincronización.
- **Condiciones meteorológicas severas**: Tormentas severas, nevadas intensas y otras condiciones meteorológicas adversas pueden afectar la capacidad de los receptores GPS para captar señales satelitales, aunque esto generalmente tiene un impacto menor en comparación con otras fuentes de error.

¿Qué es el GPS/RTK?

El GPS/RTK es una técnica de navegación y posicionamiento que utiliza las señales del GPS estándar, pero con correcciones en tiempo real que permiten una precisión mucho mayor. El GPS/RTK logra una precisión centimétrica mediante la utilización de correcciones diferenciales en tiempo real proporcionadas por una estación base fija, a diferencia del GPS base, que se basa únicamente en las señales de los satélites y tiene una precisión de varios metros. Esta diferencia en precisión hace que el GPS/RTK sea adecuado para aplicaciones que requieren alta exactitud, mientras que el GPS base es más adecuado para usos generales donde la precisión extrema no es crítica.

A pesar de sus ventajas, el GPS/RTK tiene algunas limitaciones y desafíos:

1. **Requiere visibilidad directa**: Para obtener la máxima precisión, tanto la estación base como el receptor móvil deben tener una buena visibilidad de los satélites GPS, lo que puede ser un problema en áreas densamente arboladas, urbanas o montañosas.
2. **Dependencia de comunicación**: La precisión del GPS/RTK depende de la transmisión en tiempo real de las correcciones, lo que requiere un canal de comunicación fiable entre la estación base y el receptor móvil.
3. **Alcance de la estación base**: La precisión de las correcciones disminuye con la distancia entre la estación base y el receptor móvil, generalmente a medida que esta distancia excede los 10 a 20 kilómetros.
4. **Costos**: Implementar un sistema GPS/RTK puede ser costoso debido a la necesidad de equipos especializados y mantenimiento de la infraestructura.

2. ALGORITMOS DE PLANIFICACIÓN

2.1 Introducción

Los algoritmos de planificación en robótica móvil son esenciales para permitir la navegación autónoma y la ejecución de tareas. Estos algoritmos se encargan de generar rutas o trayectorias viables que un robot puede seguir para alcanzar un objetivo concreto evitando obstáculos y optimizando determinados criterios, como la distancia, el tiempo o el consumo de energía. Cabe destacar la diferencia entre *path planning* y *motion planning*.

- ***Path planning*:** Consiste en encontrar una trayectoria sin colisiones desde un punto de partida hasta una posición de destino. Se centra en los aspectos geométricos de la trayectoria.
- ***Motion planning*:** Amplía la planificación de trayectorias teniendo en cuenta la cinemática y la dinámica del robot, garantizando que la trayectoria generada pueda seguirse dadas las restricciones físicas del robot.

2.2 Tipos de algoritmos de planificación

A continuación, veremos algunos de los algoritmos de planificación más utilizados en robótica móvil:

- **Algoritmo de Dijkstra:**

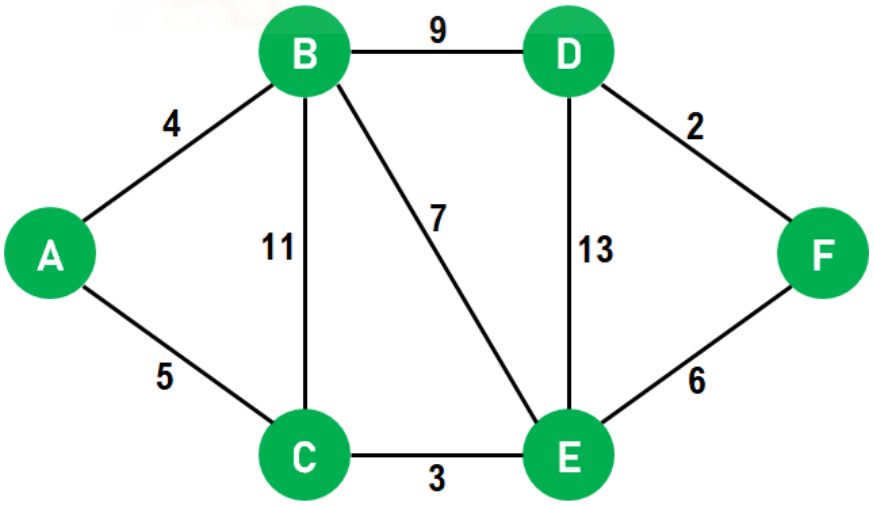


Figura 9 Algoritmo Dijkstra.

El algoritmo de Dijkstra es un conocido algoritmo para encontrar el camino más corto entre nodos de un gráfico, que puede representar, por ejemplo, redes de carreteras o cuadrículas de navegación robótica. Se trata fundamentalmente de un algoritmo de búsqueda que funciona explorando todos los caminos posibles de

forma exhaustiva para determinar el más corto. Los pasos del algoritmo son los siguientes:

1. Se marcan todos los nodos como no visitados. Se crean un conjunto de todos los nodos no visitados al que llamaremos *Q*.
2. Se asigna a cada nodo un valor de distancia desde el inicio: para el nodo inicial, es cero, y para todos los demás nodos, es infinito, ya que inicialmente no se conoce ningún camino hacia estos nodos. Durante la ejecución del algoritmo, la distancia de un nodo *N* es la longitud del camino más corto descubierto hasta el momento entre el nodo de partida y *N*.
3. Del conjunto no visitado, seleccionamos el nodo actual que tenga la menor distancia finita; inicialmente, éste será el nodo de partida, que tiene distancia cero. Si el conjunto no visitado está vacío, o contiene sólo nodos con distancia infinita (que son inalcanzables), entonces el algoritmo termina yendo al paso 6.
4. Para el nodo actual, se calcula la distancia tentativa desde dicho nodo hasta sus vecinos utilizando la siguiente fórmula:

$$dt(v_i) = Da + d(a, v_i)$$

Donde:

- $\boldsymbol{dt(v_i)}$ es la distancia tentativa al nodo vecino $\boldsymbol{v_i}$.
- $\boldsymbol{Da}$ es la distancia acumulada del nodo actual $\boldsymbol{a}$,
- $\boldsymbol{d(a, v_i)}$ es la distancia entre el nodo actual $\boldsymbol{a}$ y el nodo vecino $\boldsymbol{v_i}$.

Si la distancia tentativa $\boldsymbol{dt(v_i)}$ es menor que la distancia actualmente almacenada en el vector D para el nodo $\boldsymbol{v_i}$, entonces se actualiza el vector con esta distancia tentativa. Formalmente, si:

$$dt(v_i) < Dv_i$$

entonces,

$$Dv_i = dt(v_i)$$

5. Cuando se haya terminado de considerar todos los vecinos no visitados del nodo actual, se marcará el nodo actual como visitado y se eliminará del conjunto de no visitados.

6. Una vez que se sale del bucle (pasos 3-5), cada nodo visitado contendrá su distancia más corta desde el nodo de partida. Para obtener la ruta a un nodo visitado, se comenzará con este nodo, y repetidamente se elegirá su vecino entrante con la distancia más corta, hasta llegar al nodo de partida.

El pseudocódigo es el siguiente:

```
1  function Dijkstra(Graph, source):
2
3      for each vertex v in Graph.Vertices:
4          dist[v] ← INFINITY
5          prev[v] ← UNDEFINED
6          add v to Q
7      dist[source] ← 0
8
9      while Q is not empty:
10         u ← vertex in Q with minimum dist[u]
11         remove u from Q
12
13         for each neighbor v of u still in Q:
14             alt ← dist[u] + Graph.Edges(u, v)
15             if alt < dist[v]:
16                 dist[v] ← alt
17                 prev[v] ← u
18
19     return dist[], prev[]
```

- **Algoritmo A*:**

Un popular algoritmo de búsqueda basado en heurística que utiliza una función de coste para encontrar el camino más corto. La función de coste es la siguiente:

$$f(n) = g(n) + h(n)$$

Donde:

- $\boldsymbol{f(n)}$ es el coste total estimado de la solución más "barata" a través del nodo $\boldsymbol{n}$.
- $\boldsymbol{g(n)}$ es el coste real desde el nodo inicial hasta el nodo actual.
- $\boldsymbol{h(n)}$ es la función heurística que estima el coste desde el nodo actual $\boldsymbol{n}$ hasta el nodo objetivo. Es una heurística admisible, lo que significa que nunca

sobreestima el coste real. Algunas comúnmente utilizadas son la distancia euclídea y de Manhattan.

Los pasos del algoritmo son:

1. Se inicializan dos listas, la lista abierta que inicialmente contiene el nodo de inicio y la lista cerrada, inicialmente vacía.
2. Para cada iteración se selecciona el nodo de la lista abierta con el valor más pequeño de $\boldsymbol{f(n)}$.
3. Para el nodo seleccionado se calcula el coste real de alcanzar cada vecino desde el nodo actual y el coste heurístico desde cada vecino al objetivo.
4. Para cada vecino se calcula el coste total estimado $\boldsymbol{f(n)}$, si un vecino no está en la lista abierta lo añadimos. Si un vecino ya está en el conjunto abierto y su coste $\boldsymbol{f(n)}$ es inferior al coste $\boldsymbol{f(n)}$ registrado anteriormente, se actualiza el coste $\boldsymbol{f(n)}$ del vecino y establece su "padre" (nodo del que proviene) en el nodo actual.
5. Después de evaluar los vecinos del nodo actual, se mueve el nodo actual a la lista cerrada, indicando que este ha sido completamente evaluado.
6. Si el nodo objetivo es alcanzado o la lista abierta está vacía (no existe ningún camino factible) el bucle se detiene, si no, se vuelve al paso 2.
7. Una vez alcanzado el nodo objetivo, la ruta puede reconstruirse siguiendo los nodos "padre" desde el nodo objetivo hasta el nodo de inicio.

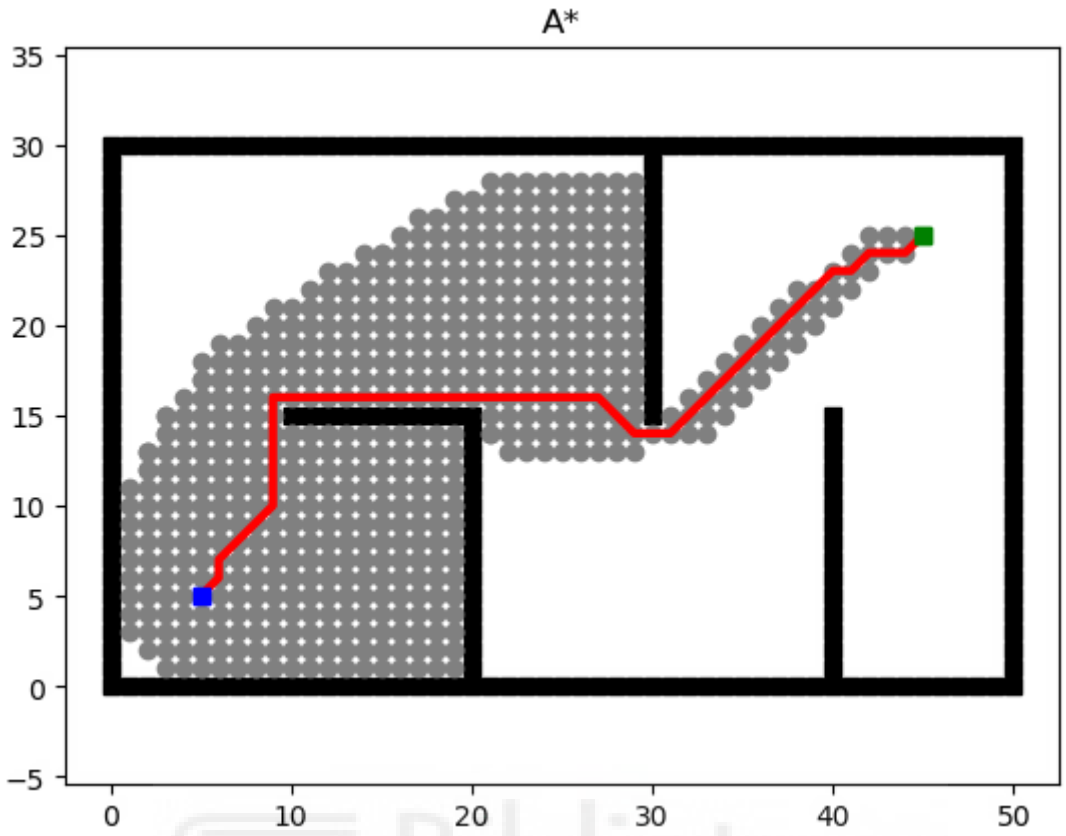


Figura 10 Ejemplo solución algoritmo A.*

- **Algoritmo D*:**

Amplía las capacidades del algoritmo clásico A* gestionando eficazmente los cambios en el entorno, como nuevos obstáculos.

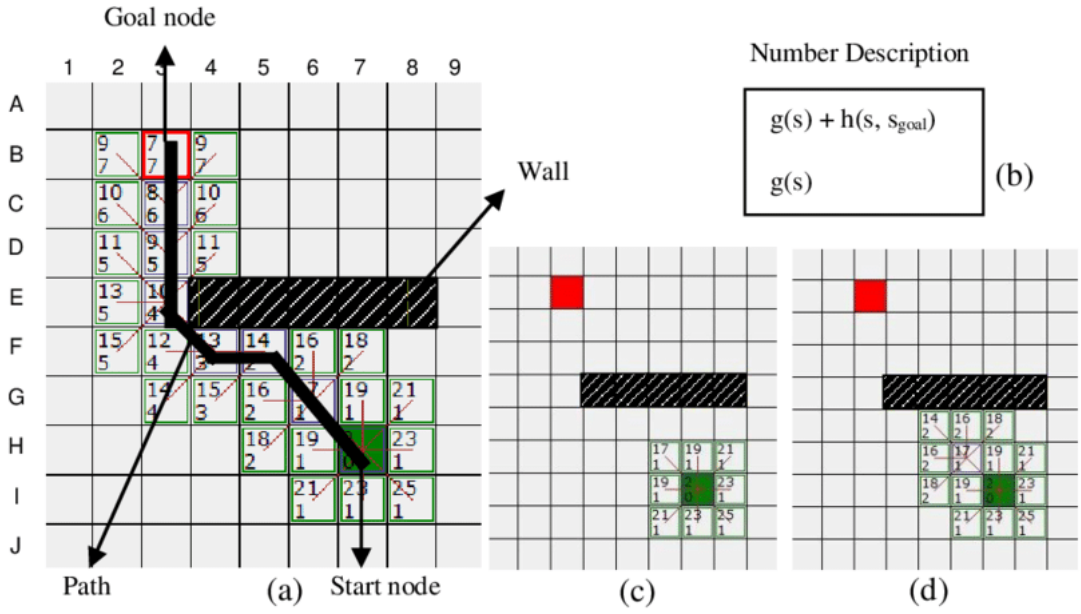


Figura 11 Ejemplo solución algoritmo D.*

- **RRT (Rapidly-exploring random trees):**

Construye un árbol muestreando aleatoriamente el espacio y conectando los nodos al nodo del árbol existente más cercano. En el siguiente apartado indagaremos más sobre cómo funciona y sus variaciones.

- ***Probabilistic roadmaps* (PRM):**

El algoritmo primero toma muestras aleatorias del espacio de configuración del robot, comprueba si estas muestras están en un espacio factible y luego se usa un planificador local para interconectar las muestras.

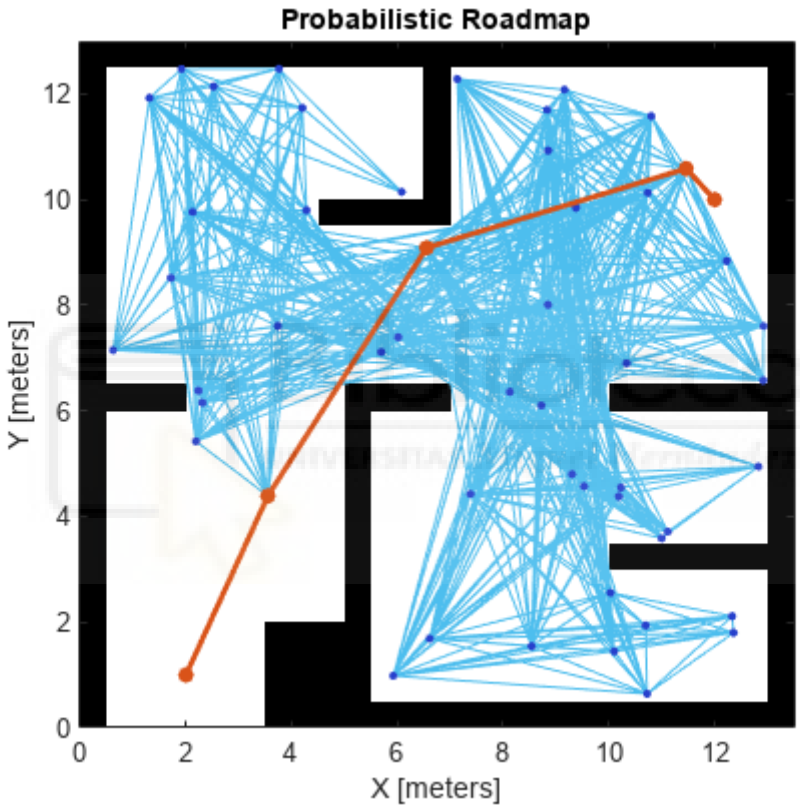


Figura 12 Ejemplo solución algoritmo PRM

2.3 RRT (Rapidly-exploring random trees)

El algoritmo RRT fue introducido en 1998 por Steven M. LaValle. Se trata de un simple algoritmo iterativo que busca rápidamente caminos factibles en entornos complejos. La idea es hacer crecer gradualmente un árbol que llene el espacio tomando muestras aleatorias del espacio y conectando el punto más cercano del árbol con la nueva muestra aleatoria. Esto hace que eventualmente los vértices estén distribuidos de forma uniforme.

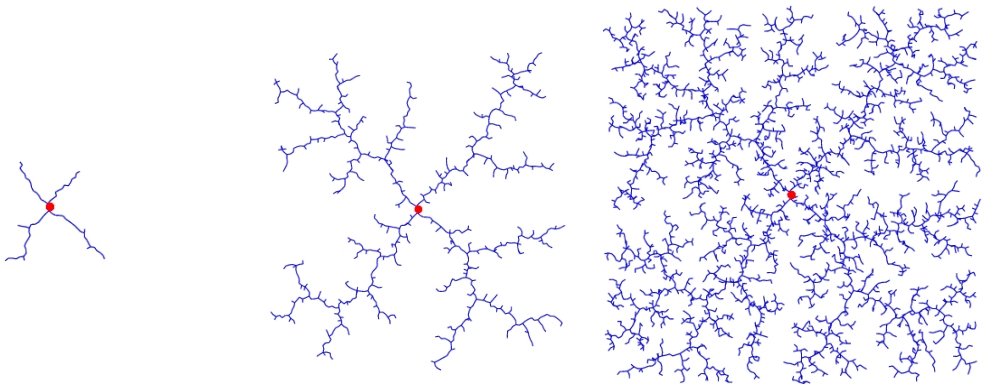


Figura 13 Expansión del algoritmo RRT.

Algunas de las principales características que hacen que el algoritmo sea utilizado en robótica móvil son las siguientes:

- **Adaptación a entornos complejos:** El RRT puede navegar a través de espacios con variedad de objetos, pasadizos estrechos y entornos que cambian dinámicamente.
- **Fácil implementación:** El algoritmo es relativamente sencillo de implementar en comparación con otros algoritmos más sofisticados de planificación de trayectorias. Esta simplicidad permite un rápido desarrollo y puesta a punto en aplicaciones robóticas móviles.
- **Bajo coste computacional:** El algoritmo puede generar rápidamente trayectorias factibles, lo que lo hace adecuado para aplicaciones en tiempo real en las que es crucial tomar decisiones rápidas.
- **Variedad de robots móviles:** El RRT puede adaptarse a distintos tipos de robots móviles, como robots con ruedas, drones y coches. Su flexibilidad para manejar diversas cinemáticas y dinámicas de robots es una ventaja significativa.

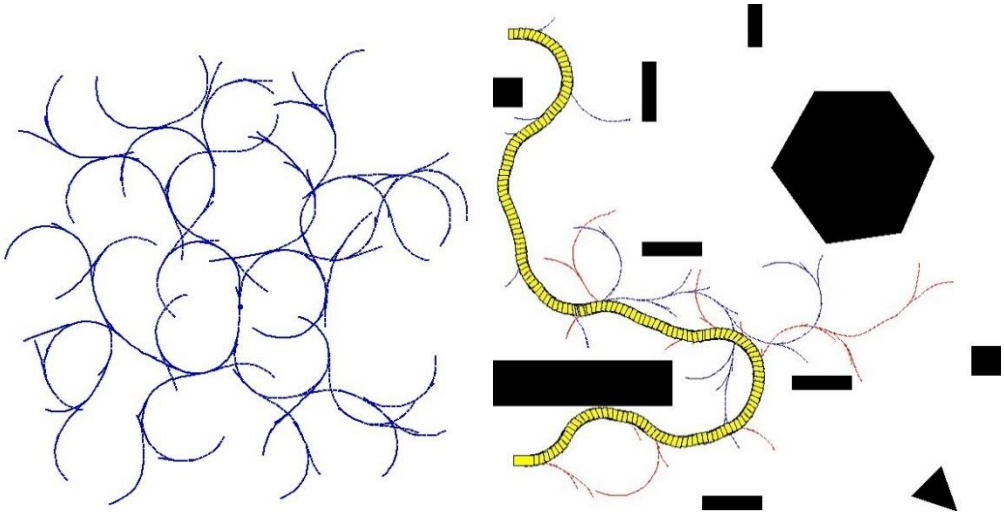


Figura 14 Ejemplo de RRT adaptado a un robot con la cinemática de un coche.

Los pasos del algoritmo RRT son los siguientes:

1. Se define el espacio C dentro del cual debe planificarse la trayectoria, llamaremos C_{free} al espacio perteneciente a C en el que los cuerpos no colisionan con ningún obstáculo.
2. Se inicializa el árbol $\mathcal{T}$, este árbol contiene un solo vértice q_{init} ,que es la posición inicial del robot.
3. Se intenta extender el RRT añadiendo un nuevo vértice q_{rand} aleatoriamente muestreado.
4. Identificamos el nodo $q_{nearest}$ del árbol $\mathcal{T}$, que es el vértice más cercano al vértice muestreado q_{rand}.
5. Nos movemos de $q_{nearest}$ hacia q_{rand} una distancia ε para generar un vértice nuevo q_{new}.
6. Se verifica que tanto q_{new} como el camino de $q_{nearest}$ a q_{new} pertenecen a C_{free} y por lo tanto no se intersecan con ningún obstáculo. Si pertenecen a C_{free} se añade q_{new} al árbol $\mathcal{T}$.
7. Se comprueba que q_{new} se encuentra a una distancia menor a ε del punto objetivo q, si es así el camino ha sido encontrado y se detiene el proceso. Si no, se vuelve al paso 3 y se repite el proceso.

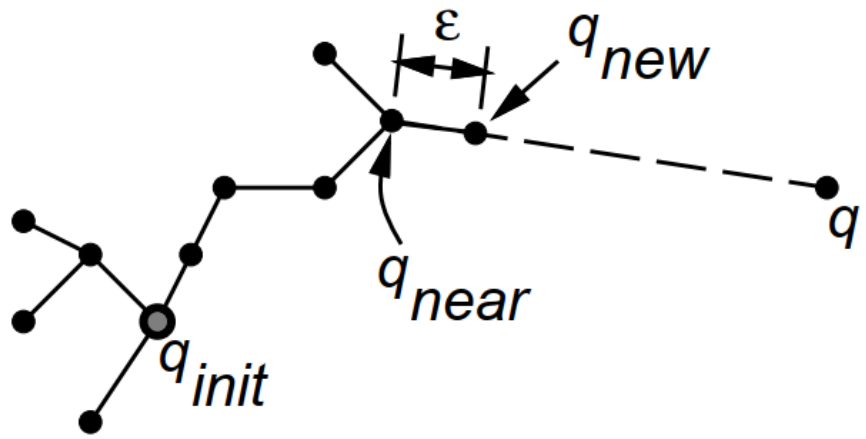


Figura 15 Operación EXTEND del algoritmo.

El código base del algoritmo RRT es el siguiente:

```
BUILD_RRT(q_init)
 1  T.init(q_init);
 2  for k = 1 to K do
 3      q_rand ← RANDOM_CONFIG();
 4      EXTEND(T, q_rand);
 5  Return T
```

```
EXTEND(T, q)
 1  q_near ← NEAREST_NEIGHBOR(q, T);
 2  if NEW_CONFIG(q, q_near, q_new) then
 3      T.add_vertex(q_new);
 4      T.add_edge(q_near, q_new);
 5      if q_new = q then
 6          Return Reached;
 7      else
 8          Return Advanced;
 9  Return Trapped;
```

Figura 16 Algoritmo básico del RRT.

Donde *Reached* supone el caso en el que q_{new} está a una distancia de q menor que ε; *Advanced* es cuando $q_{new} \neq q$ y *Trapped* cuando el nuevo vértice propuesto es rechazado porque no pertenece a C_{free}.

Para hacer que el número de iteraciones sea menor se han desarrollado varias modificaciones.

- **Bi-directional RRT:** En esta variación en vez de generar un solo árbol se generan dos, $\mathcal{T}_{init}$ y $\mathcal{T}_{goal}$, se repiten los mismos pasos que en el algoritmo RRT básico solo que los árboles se intercambian los roles cada iteración, por lo tanto, ambos crecen simultáneamente. Cuando la distancia entre los vértices q_{new} y q_{new_goal} es menor a ε se considera que los árboles están conectados y que el camino ha sido encontrado.

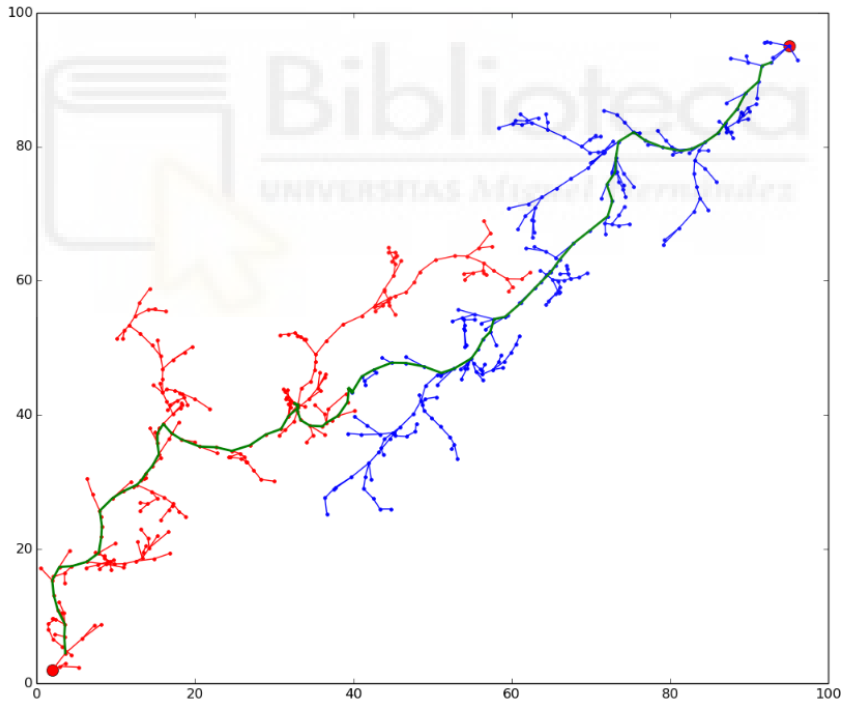


Figura 17 Ejemplo solución para Bi-directional RRT.

- **RRT-Connect:** Como en *Bi-directional RRT* se generan dos árboles $\mathcal{T}_{init}$ y $\mathcal{T}_{goal}$. Sin embargo, en el paso 3 en vez de que el nuevo vértice sea muestreado aleatoriamente este se genera repetidamente en dirección q_{new_goal} con una distancia ε hasta que se alcanza un obstáculo. Después, los roles son

intercambiados entre los dos árboles y el algoritmo termina cuando la distancia entre los vértices q_{new} y q_{new_goal} es menor a ε.

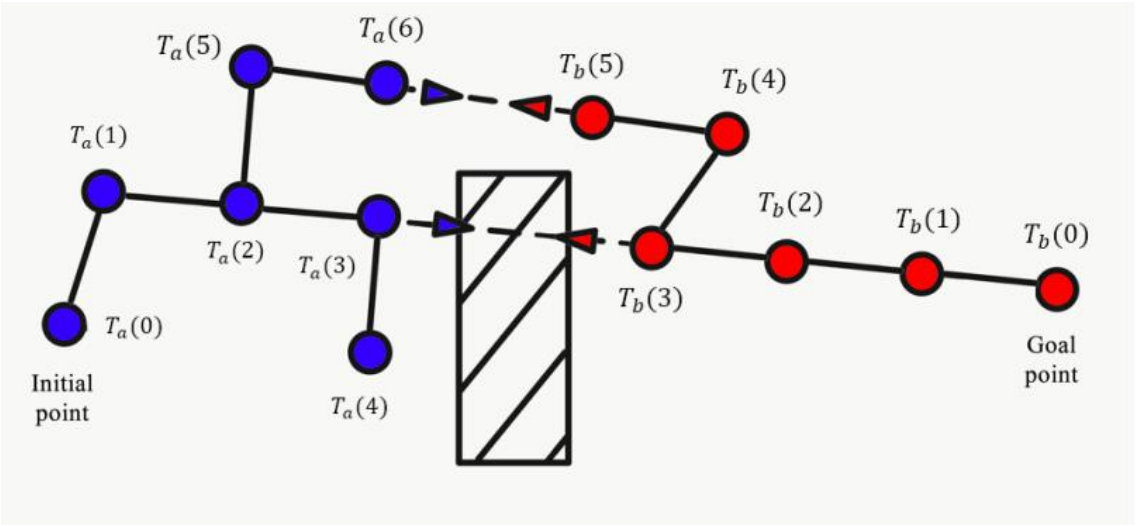


Figura 18 Algoritmo RRT-Connect.

Aquí está el pseudocódigo para su implementación:

```
CONNECT(T, q)
1  repeat
2      S ← EXTEND(T, q);
3  until not (S = Advanced)
4  Return S;
```

```
RRT_CONNECT_PLANNER(q_init, q_goal)
1  T_a.init(q_init); T_b.init(q_goal);
2  for k = 1 to K do
3      q_rand ← RANDOM_CONFIG();
4      if not (EXTEND(T_a, q_rand) = Trapped) then
5          if (CONNECT(T_b, q_new) = Reached) then
6              Return PATH(T_a, T_b);
7      SWAP(T_a, T_b);
8  Return Failure
```

Figura 19 Código base algoritmo RRT-Connect.

- **RRT-Connect To Goal:** Es una mezcla entre el RRT básico y el RRT-Connect, se genera un árbol $\mathcal{T}$ y éste crece en repetidamente en dirección a q_{goal} hasta que se

alcance un obstáculo. El camino es encontrado cuando q_{new} está a una distancia de q_{goal} menor que ε. Esto hace que la solución converja rápidamente.

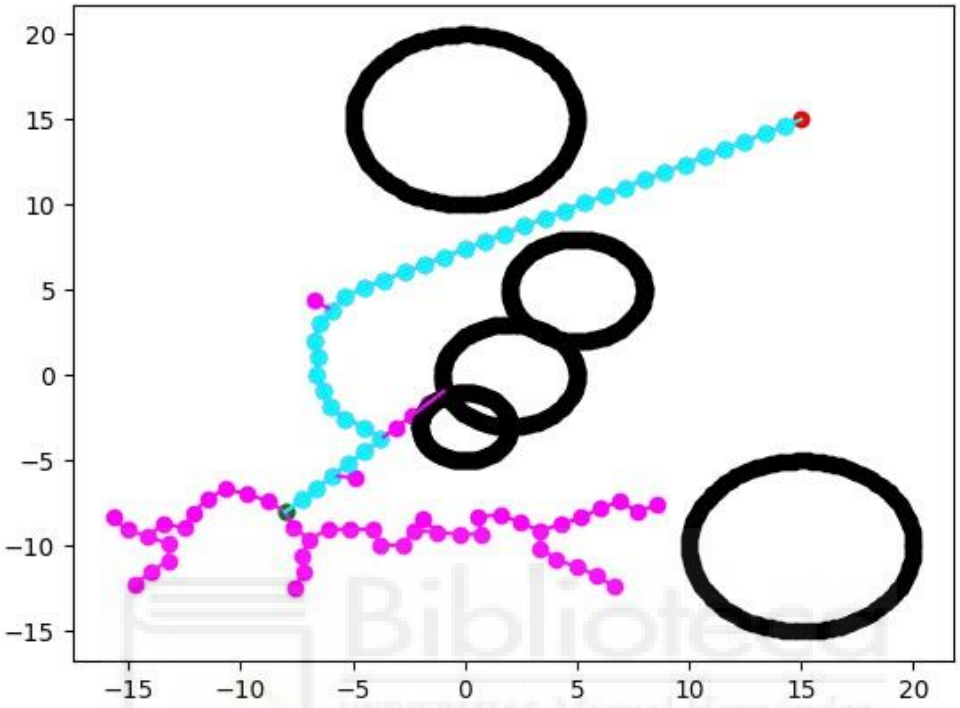


Figura 20 Ejemplo solución encontrada para algoritmo RRT-Connect To Goal.

Todos estos algoritmos han sido aplicados en entornos bidimensionales, sin embargo, todos ellos se pueden aplicar a entornos tridimensionales con nubes de puntos.

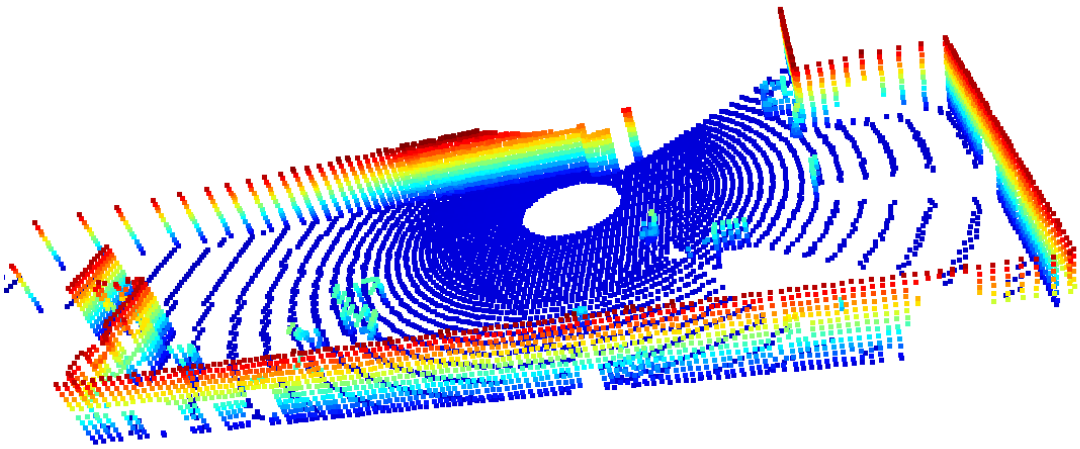


Figura 21 Espacio tridimensional formado por nubes de puntos.

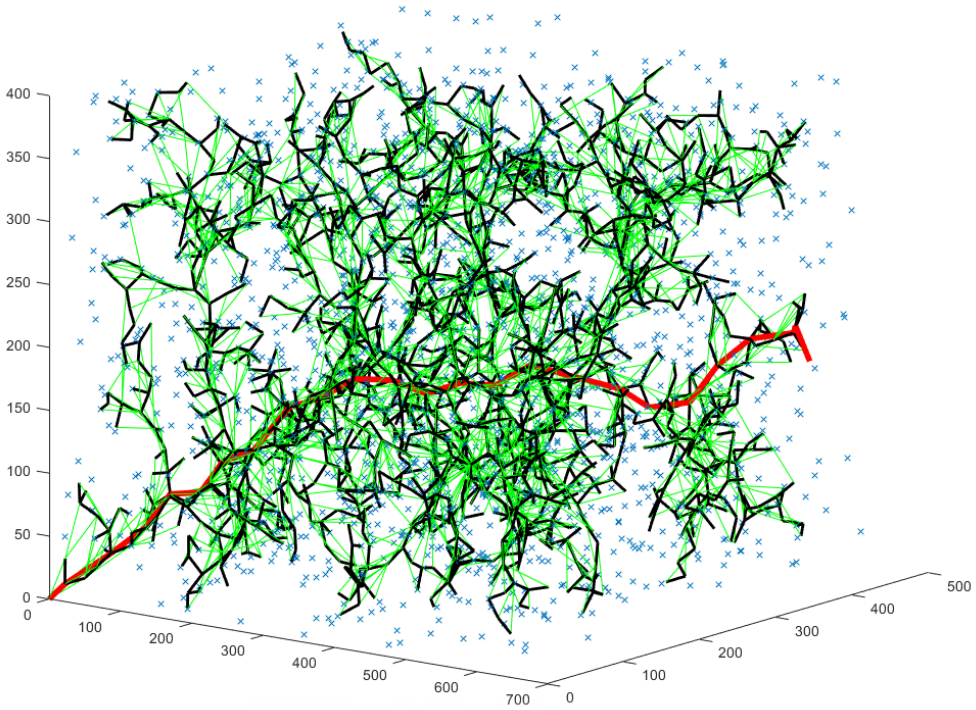


Figura 22 Ejemplo de solución del algoritmo RRT sobre nubes de puntos.

La trayectoria en bruto obtenida por el RRT puede ser subóptima, con giros y vueltas innecesarios. Pueden aplicarse técnicas de posprocesamiento para suavizar la trayectoria, normalmente se suele hacer mediante la construcción de una spline a partir de los nodos del camino.

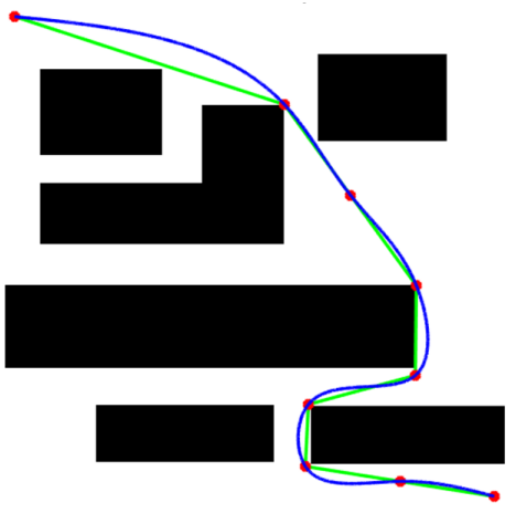


Figura 23 trayectoria suavizada.

3. IMPLEMENTACIÓN

3.1 CoppeliaSim

CoppeliaSim, anteriormente conocido como V-REP, es un simulador de robots utilizado en la industria, la educación y la investigación. Fue desarrollado originalmente por Toshiba R&D y actualmente está siendo desarrollado y mantenido activamente por Coppelia Robotics AG, una pequeña empresa ubicada en Zurich, Suiza.

Figura 24 Logo CoppeliaSim.

Se basa en una arquitectura de control distribuido con scripts Python y Lua, o plug-ins C/C++ que actúan como controladores síncronos individuales. Los controladores asíncronos adicionales pueden ejecutarse en otro proceso o máquina a través de varias soluciones de middleware (ROS, remote API, ZeroMQ) con lenguajes de programación como C/C++, Python, Java y Matlab.

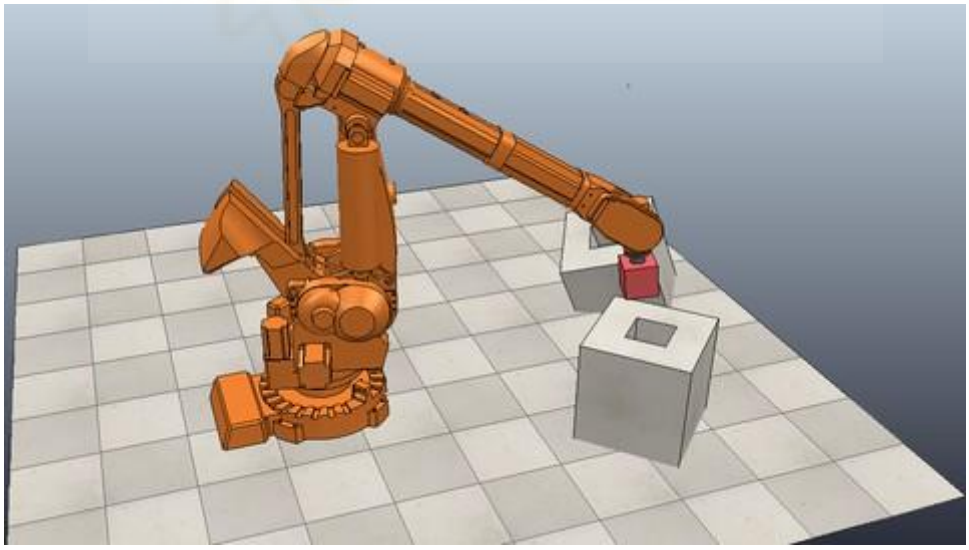


Figura 25 Escena del simulador CoppeliaSim.

CoppeliaSim utiliza un motor cinemático para los cálculos de cinemática directa e inversa y varias bibliotecas de simulación física (MuJoCo, Bullet, ODE, Vortex, Newton Game Dynamics) para realizar simulaciones de cuerpos rígidos. Los modelos y las escenas se construyen ensamblando varios objetos (mallas, articulaciones, varios sensores, nubes de

puntos, árboles, etc.) en una estructura jerárquica. Las funciones adicionales, proporcionadas por plug-ins, incluyen: planificación de movimiento, visión sintética y procesamiento de imágenes, detección de colisiones, cálculo de distancia mínima, interfaces gráficas de usuario personalizadas y visualización de datos.

3.2 Robot Husky

El robot Husky fue desarrollado por *"Clearpath Robotics"* que abarca la fabricación y venta de diversos vehículos terrestres y marítimos utilizados para la investigación robótica, así como la venta de componentes individuales para la creación de prototipos robóticos.

Figura 26 Logo de Clearpath Robotics.

El HUSKY es un robusto vehículo todoterreno UGV (*unmanned ground vehicle*) conocido por su robustez y su fiabilidad en diversas condiciones ambientales. Se utiliza ampliamente en la investigación académica e industrial para tareas como la navegación autónoma, la integración de sensores y el desarrollo de software robótico.

Figura 27 Robot Husky navegando sobre un terreno fangoso.

El robot tiene la capacidad de montar una gran cantidad de sensores y módulos para la percepción y navegación como sensores LiDAR, cámaras, IMU's y *encoders*. Además, el Husky A200 puede montar pequeños robots manipuladores industriales para ser manejados telemáticamente o de forma autónoma.

Figura 28 Robot Husky montado con dos robots manipuladores.

Pasando a las características técnicas, el robot está compuesto por dos motores con una potencia máxima de 1000 W, lo que le permite alcanzar velocidades de hasta 1m/s, subir pendientes de 45º y soportar una carga máxima de 75kg.

El peso base del robot es de 50kg, con una batería aproximada de 3 horas en funcionamiento, cuenta con bus de comunicación RS232 a 115200 baudios, las dimensiones son las siguientes:

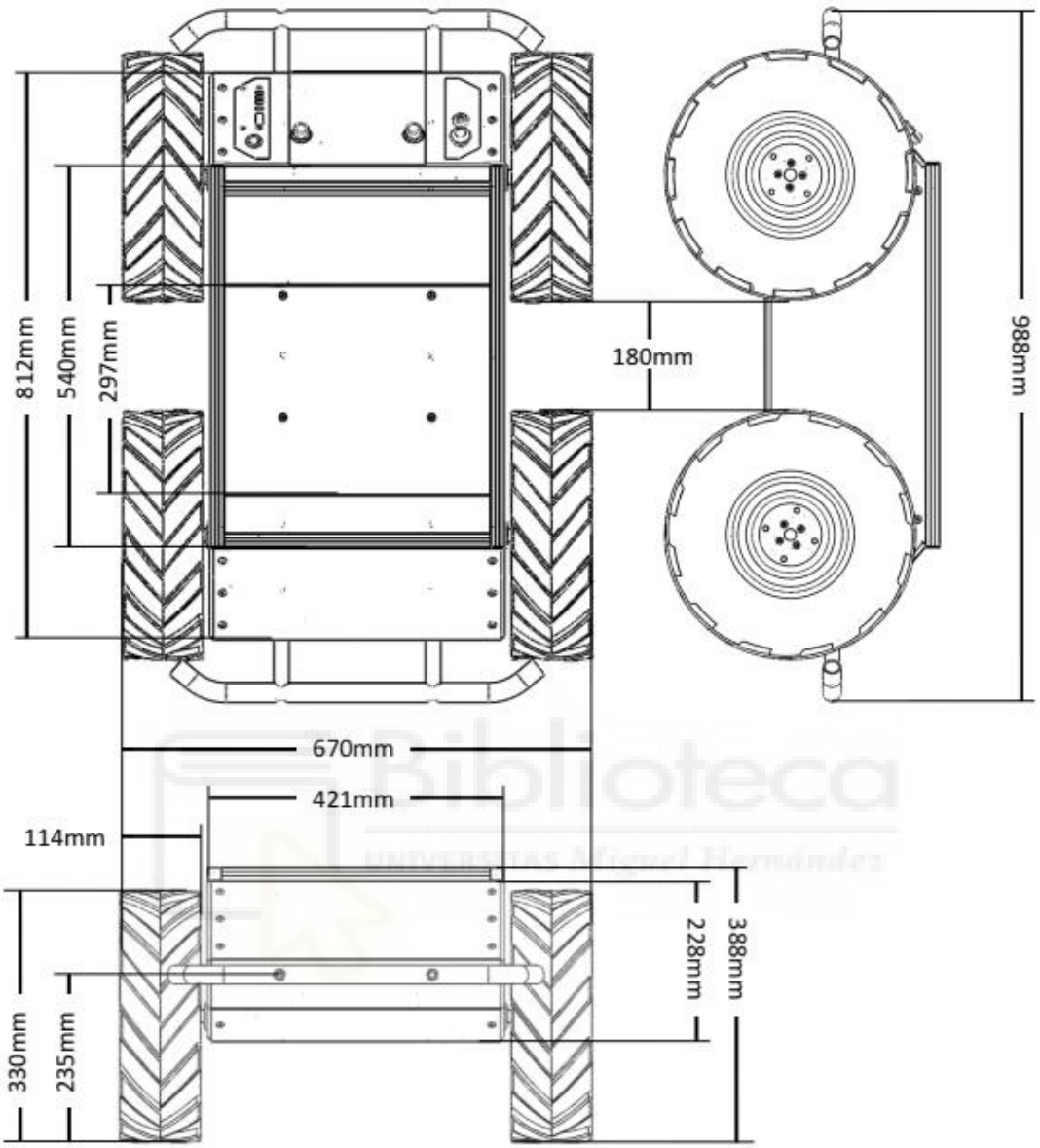


Figura 29 Dimensiones del robot Husky.

El Husky se puede programar en varios lenguajes y entornos ya que contiene drivers y APIs para: ROS Melodic, ROS Kinetic, C++ Library, Mathworks, ROS MOOS-IvP, LabVIEW y Python.

El modelo cinemático del Husky es de tipo diferencial, el modelo diferencial se basa en dos ruedas que se accionan de manera independiente y están ubicadas a ambos lados de su cuerpo. Al ajustar la velocidad de rotación de cada rueda por separado, el robot puede cambiar de dirección, lo que elimina la necesidad de un mecanismo adicional para la

dirección. Si las dos ruedas giran en la misma dirección y a la misma velocidad, el robot irá en línea recta. Si ambas ruedas giran con la misma velocidad en sentidos opuestos, el robot girará alrededor de sí mismo.

Este modelo cinemático tiene algunas restricciones como la restricción no-holonómica:

$$\dot{y}\cos(\theta) - \dot{x}\sin(\theta) = 0$$

Ecuación 1

Esta ecuación establece que el componente lateral de la velocidad (perpendicular a la dirección de avance del robot) debe ser cero, lo que significa que el robot no puede moverse lateralmente.

Las ecuaciones que nos proporciona el fabricante son las siguientes:

$$v = \frac{v_r + v_l}{2}, \omega = \frac{v_r - v_l}{b}$$

Ecuación 2

Donde *v* representa la velocidad lineal de la plataforma y *w* la velocidad angular. v_r y v_l son las velocidades lineales de las ruedas derecha e izquierda, *b* es la distancia entre ruedas (L en la *Figura* 30), que es igual a 0.555 *m* para el robot Husky.

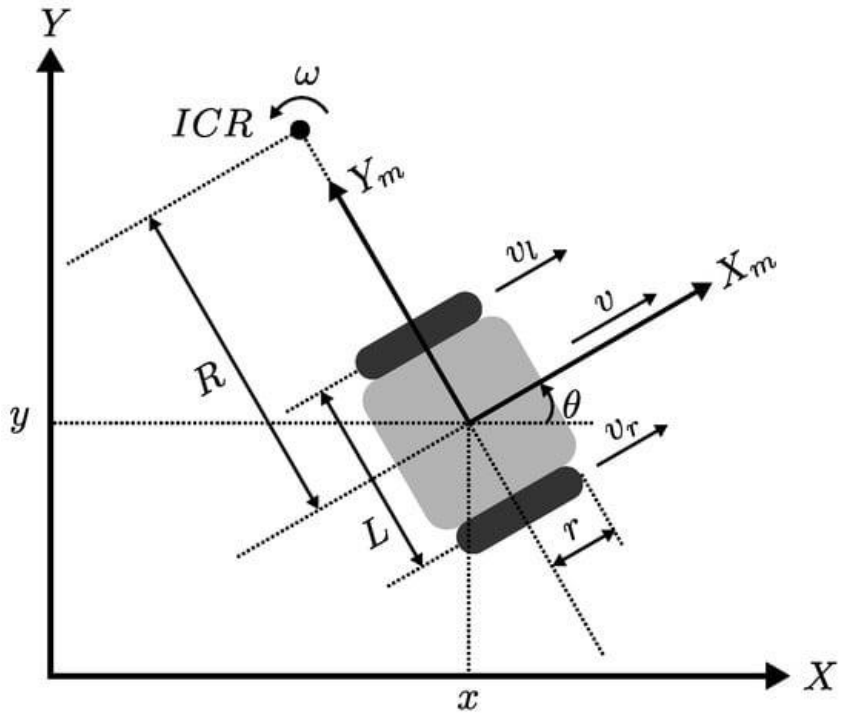


Figura 30 Esquema modelo robot diferencial.

Usando las siguientes ecuaciones e integrándolas se podrá obtener la posición del robot (x, y) y la orientación θ en un plano 2D.

$$\dot{x} = v\cos\theta$$

$$\dot{y} = v\sin\theta$$

$$\dot{\theta} = w$$

Ecuación 3

Para el control del robot se despeja y sustituye *wr* y *wl* en Ecuación 2 para dejar las ecuaciones en función de *v* y *w*, además de *r* (radio de la rueda) y *b* (distancia entre ruedas) que son constantes.

$$w_r = \frac{v + w \cdot \frac{b}{2}}{r}$$

$$w_l = \frac{v - w \cdot \frac{b}{2}}{r}$$

Ecuación 4

3.3 Escena de CoppeliaSim

Para la escena de CoppeliaSim se decidió insertar un mapa tridimensional a escala 1:1 de parte de la universidad, para ello primeramente se exporta de www.openstreetmap.org el mapa en formato *.map*. A continuación, se convierte el archivo *.map* a un modelo *.OBJ* con "OSM2World" y este se introduce en el simulador.

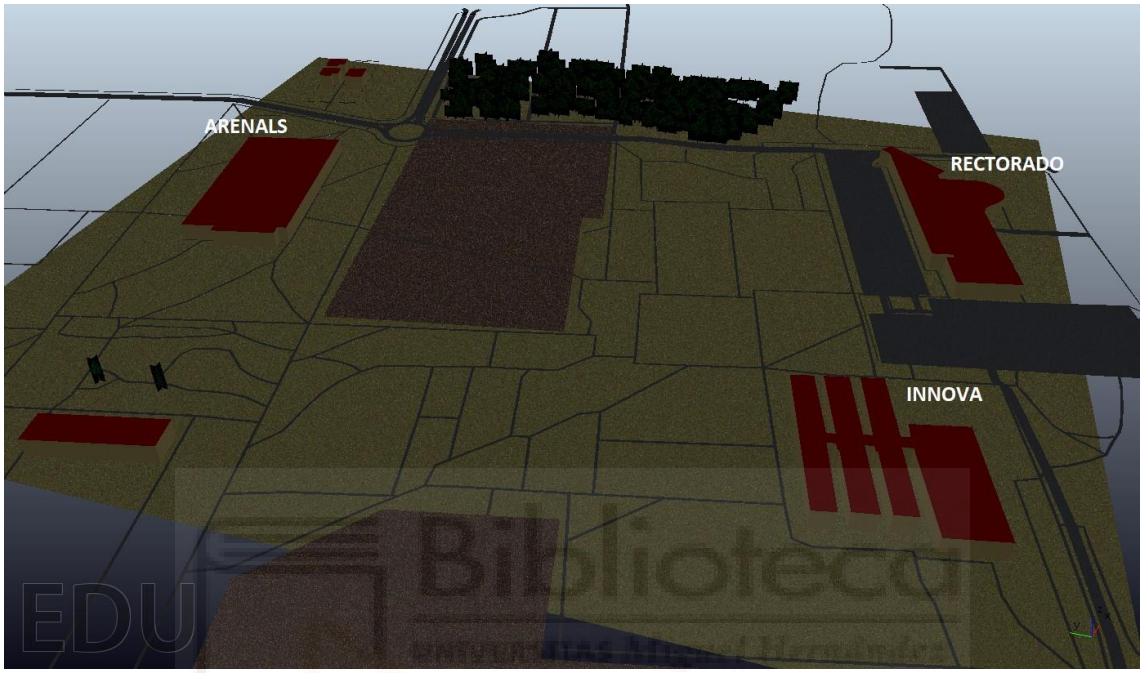


Figura 31 Modelo de parte de la universidad importado a CoppeliaSim.

El modelo dinámico del Husky está constituido por 4 *joints* controlados por velocidad (posteriormente en el código estableceremos que $w_{r1} = w_{r2}$ y $w_{l1} = w_{l2}$ ya que el Husky tiene solo dos motores). Este modelo dinámico está construido con objetos primitivos para mejorar el rendimiento de la simulación.

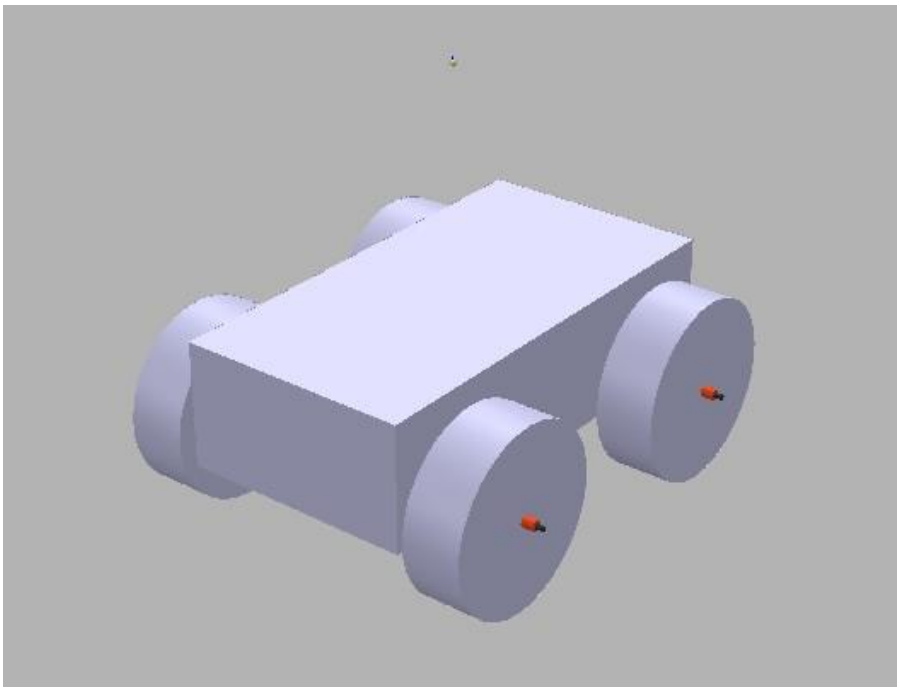


Figura 32 Modelo dinámico del Husky en CoppeliaSim.

El modelo completo con el LiDAR incluido es el siguiente:

Figura 33 Modelo completo del Husky en CoppeliaSim.

Hay que tener en cuenta también la orientación de la base para que, cuando se le comande que se mueva hacia adelante las ruedas se desplacen en la dirección x positiva.

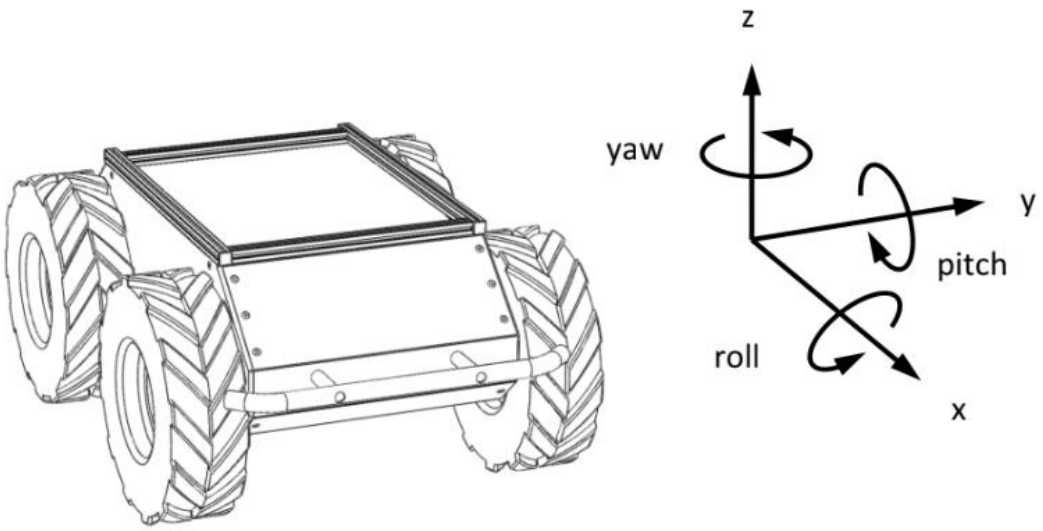


Figura 34 Orientación establecida por el fabricante.

Además, se añadirán distintos obstáculos y objetos como palmeras, arbustos o personas para realizar distintas pruebas.

3.4 Descripción del software

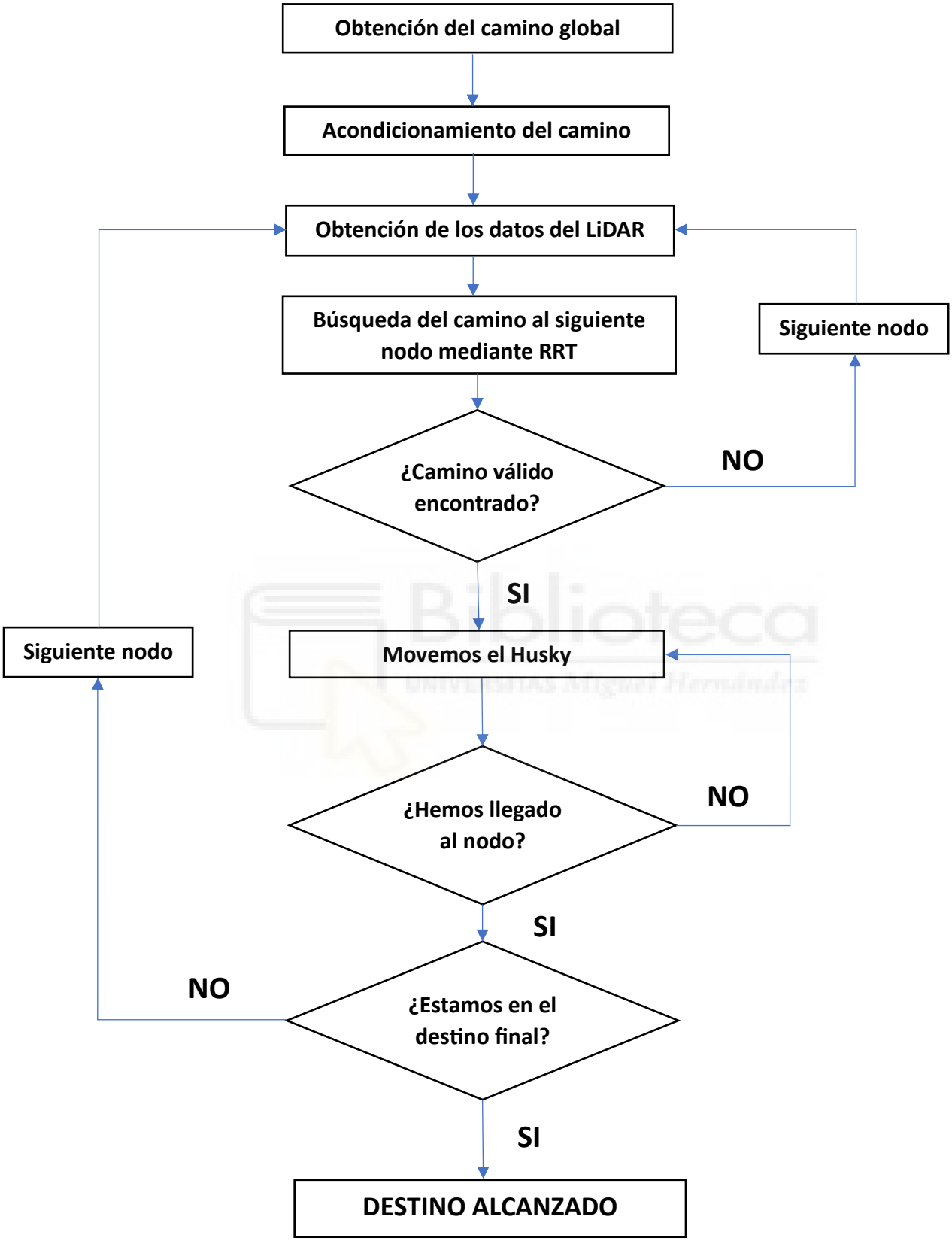
Para el desarrollo del software se ha utilizado Python, es un lenguaje de programación interpretado de alto nivel muy utilizado para diversas aplicaciones, desde el desarrollo web hasta la informática científica y la inteligencia artificial. Creado por Guido van Rossum y publicado por primera vez en 1991, Python hace hincapié en la legibilidad y simplicidad del código, lo que permite a los programadores expresar conceptos en menos líneas de código en comparación con lenguajes como C++ o Java.

Python se utiliza habitualmente en robótica debido a su facilidad de aprendizaje, legibilidad y amplias librerías. Entre las principales razones se incluyen:

- **Facilidad de uso:** La sencilla sintaxis de Python permite una rápida creación de prototipos y desarrollo, por lo que es accesible para los principiantes y eficiente para los desarrolladores experimentados.
- **Librerías robustas:** Python cuenta con potentes librerías como ROS para robótica, NumPy y SciPy para procesamiento de datos, OpenCV para visión por computador y TensorFlow para *machine learning*.
- **Integración de hardware:** Python interactúa bien con el hardware a través de librerías para microcontroladores y comunicación serie.
- **Multiplataforma:** Python funciona en varios sistemas operativos, lo que garantiza una amplia compatibilidad.
- **Comunidad y documentación:** La activa comunidad de Python y su completa documentación proporcionan un sólido soporte y recursos.

La estructura general del software implementado se compone por las siguientes librerías/módulos:

- **Librería pyARTE**: Librería de robótica creada por Arturo Gil, se ha utilizado para manejar matrices homogéneas de forma eficiente y para controlar la simulación de CoppeliaSim.
- **API CoppeliaSim**: Se ha utilizado para poder comunicar los scripts de Python con el simulador CoppeliaSim.
- **Librería RRTPlanner**: Se ha utilizado y modificado esta librería que contiene los algoritmos base de los diferentes tipos de RRT. Estos se han modificado para satisfacer los requerimientos del problema. Se utilizará principalmente la función *find_path_connect_to_goalPC().*
- **Overpass_routes.py**: Este fichero contiene las funciones utilizadas para la obtención del camino global a partir de las coordenadas GPS.
- **Husky.py**: Este fichero contiene la clase HuskyRobot() en la cual se define el tanto el comportamiento del robot Husky como sus controladores de movimiento y aceleración.
- **Traj_mod.py**: Este fichero contiene las funciones utilizadas para el acondicionamiento del camino encontrado previamente.
- **Ejemplo.py**: Este fichero contiene un ejemplo de la implementación total del programa.



Obtención del camino global

Para la obtención del camino se ha utilizado *overpass-turbo.eu*. Overpass Turbo proporciona un entorno interactivo para realizar consultas en vivo de la base de datos de OpenStreetMap y visualizar los resultados en un mapa. Los resultados se pueden exportar para análisis y visualizaciones adicionales.

En el presente caso se van a utilizar los nodos del mapa para poder formar el camino, para ello se introducirán las siguientes líneas de código en la página web.

```
node({{bbox}});
// print results
out body;
```

La salida será una serie de datos en XML de la cual se podrán extraer distintos datos según la última línea del código, las opciones son las siguientes:

	Own object type	Own object ID	For nodes: Own coordinates	For ways: IDs of member nodes	For relations: ID, type, role of members	Own tags	Timestamp, Version, Changeset, User, User ID
out ids	yes	yes					
out skel	yes	yes	yes	yes	yes		
out body	yes	yes	yes	yes	yes	yes	
out tags	yes	yes				yes	
out meta	yes	yes	yes	yes	yes	yes	yes

Figura 35 Opciones de código para extraer la información.

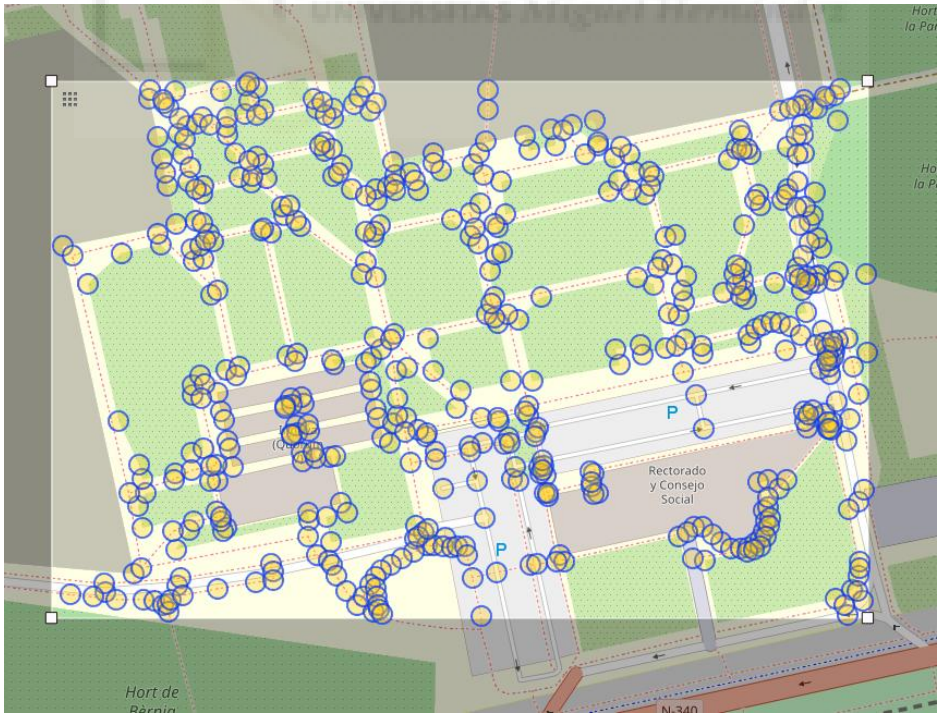


Figura 36 Nodos obtenidos de OverpassTurbo.

Estos datos se exportan en un .OSM, que se manejarán posteriormente en Python con la librería *osmnx*. El objetivo es que, dadas las coordenadas de inicio y destino, se encuentren los nodos que forman el camino más corto.

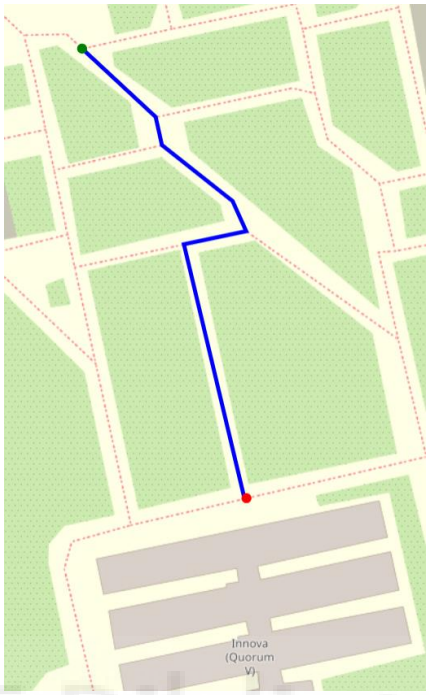


Figura 37 Trayectoria generada a partir de los nodos de OverpassTurbo.

Para ello, primero se establece nuestra coordenada de inicio y destino y se buscan los nodos más cercanos a estos. A continuación, mediante el algoritmo Dijkstra se busca el camino más corto que conecta estos nodos y finalmente se obtienen las coordenadas de estos para su posterior utilización. El código es el siguiente:

```
def getTrajNodes (origin_point, destination_point):
    # Load the .OSM file (nodes and edges)
    filepath = 'C:/Users/User/Desktop/tfg/path_planning/osms/export.osm'
    G = ox.graph.graph_from_xml(filepath, bidirectional=True,
    simplify=False,retain_all=True)

    # Get the nearest nodes to the locations
    origin_node = ox.distance.nearest_nodes(G, X=origin_point[1],
                  Y=origin_point[0], return_dist=False)
    destination_node = ox.distance.nearest_nodes(G, X=destination_point[1],
                       Y=destination_point[0], return_dist=False)

    print('Origin node: ', origin_node)

    print('Destination node: ', destination_node)

    # Finding the optimal path
    route = nx.shortest_path(G, origin_node, destination_node, weight='length')
    # by default, it uses dijkstra
    lines = node_list_to_path(G, route)
    print(lines)
    dataout = []
    for i in range(len(lines)):
        dataout.append(lines[i][0])
        dataout.append(lines[len(lines) - 1][1])
    print('Dataut',dataout)

    return dataout
```

Acondicionamiento del camino

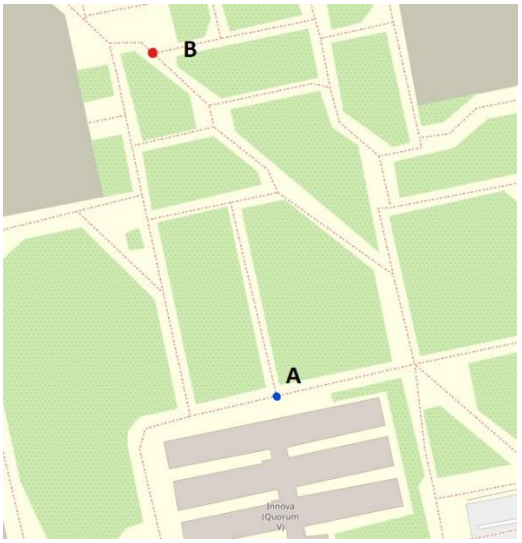
En esta parte del código se trata de modificar el camino previamente hallado para facilitar posteriormente la tarea de búsqueda del planificador RRT.

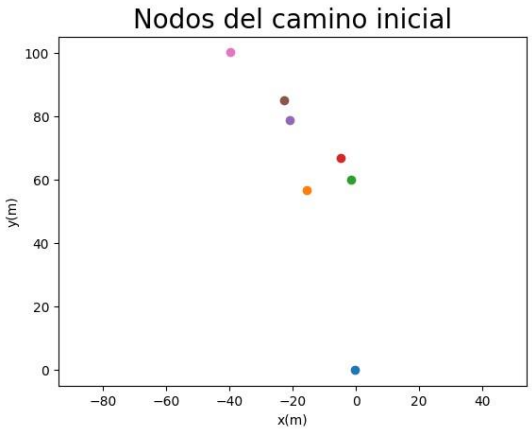
Primero se tiene que convertir las coordenadas ya que de Overpass Turbo se reciben en coordenadas geográficas. Para ello se convierten las coordenadas geográficas a UTM mediante la librería *utm.* Después, se establecen las coordenadas de nuestro sistema de referencia y se devuelven los nodos en coordenadas *x*, *y*.

```
def convertCoordinates (coord0, coord2):
    """

    :param coord0: Lat/long coordinates
    :param coord2: Lat/long coordinates
    :return: cartesian coordinates
    """
    base = utm.from_latlon(coord0[0], coord0[1])
    dest = utm.from_latlon(coord2[1], coord2[0])
    x = dest[0] - base[0]
    y = dest[1] - base[1]
    print(x, '   ', y)
    return [x, y]
```

Las distancias entre los nodos que se obtienen de Overpass Turbo son, en muchas ocasiones demasiado grandes y el rango del LiDAR es limitado, por eso es necesario obtener nuevos nodos, esto se conseguirá interpolando linealmente cada “x” metros. En el presente caso se interpolará cada 3 metros ya que es una buena distancia para el rango de trabajo del LiDAR. Además de interpolar linealmente, se realizará un suavizado en las esquinas mediante *splines* para evitar curvas cerradas durante la navegación.





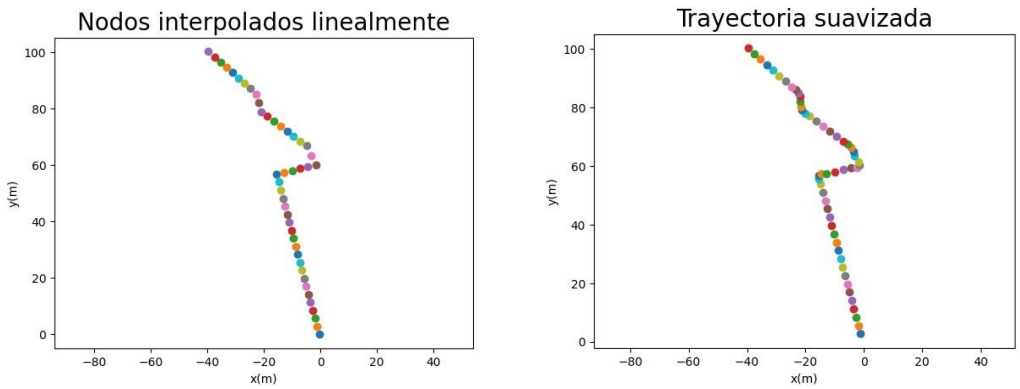


Figura 38 Pasos para el acondicionamiento del camino. De arriba a la izquierda a abajo a la derecha: Coordenadas de inicio y destino, nodos obtenidos de OverpassTurbo, nodos interpolados linealmente, trayectoria suavizada.

Búsqueda del camino mediante RRT

De todas las modificaciones del algoritmo RRT vistas anteriormente se ha empleado el *RRT-Connect To Goal* sobre nubes de puntos tridimensionales ya que, el *RRT-Connect To Goal* nos proporciona una rápida convergencia de la solución, cosa que es muy importante en la robótica móvil.

Dado que el Husky es un robot de exteriores el cual puede navegar por terrenos irregulares con baches y obstáculos, trabajar en un entorno tridimensional con nubes de puntos resulta lo más adecuado.

Para el desarrollo de esta parte el tutor Arturo Gil me proporcionó el código base del algoritmo al que yo posteriormente le hice las modificaciones necesarias para la implementación de nuestro caso en particular. El código proporcionado se puede encontrar en https://github.com/4rtur1t0/RRTPlanner.

Primeramente, se crea una nube de puntos a partir de los datos obtenidos del LiDAR, se establece el tamaño del voxel (en el apartado de experimentación se comentará más) y se calcula la transitabilidad de todos los puntos. Los puntos que sobrepasan los 15 cm de altitud o los baches que son más profundos que 7 cm son considerados obstáculos, el resto transitables.

Otros parámetros ajustables son ε, que es la distancia a la que se encontrará el nuevo nodo del árbol (véase en *Figura* 15). Dadas las dimensiones del Husky, para comprobar si este colisiona con obstáculos lo representaremos como una esfera de 70 cm de radio. También es conveniente ajustar el número máximo de iteraciones ya que un número muy grande puede provocar altos tiempos de computación.

Movimiento del Husky

Se controlará el robot en velocidad, tanto lineal como angular mediante la *Ecuación 4*.

Además, para proteger al robot de esfuerzos no deseados se le ha incorporado un control de velocidades y aceleraciones máximas admitidas.

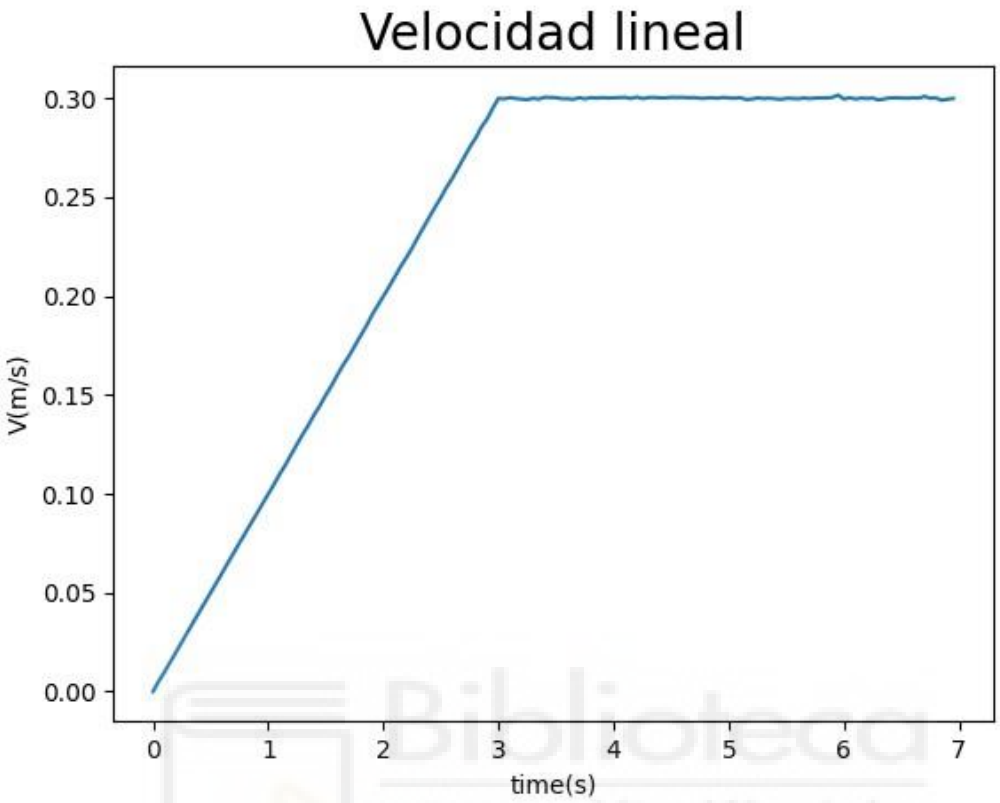


Figura 39 Velocidad del robot a lo largo del tiempo tras ser comandado a 0.3 m/s.

Para comandar que el robot vaya a un punto en específico se controlará la trayectoria ajustando la velocidad angular en función del error en orientación del robot frente al punto de destino. Cuando la distancia al punto de destino es mínima, se pasa al siguiente objetivo.

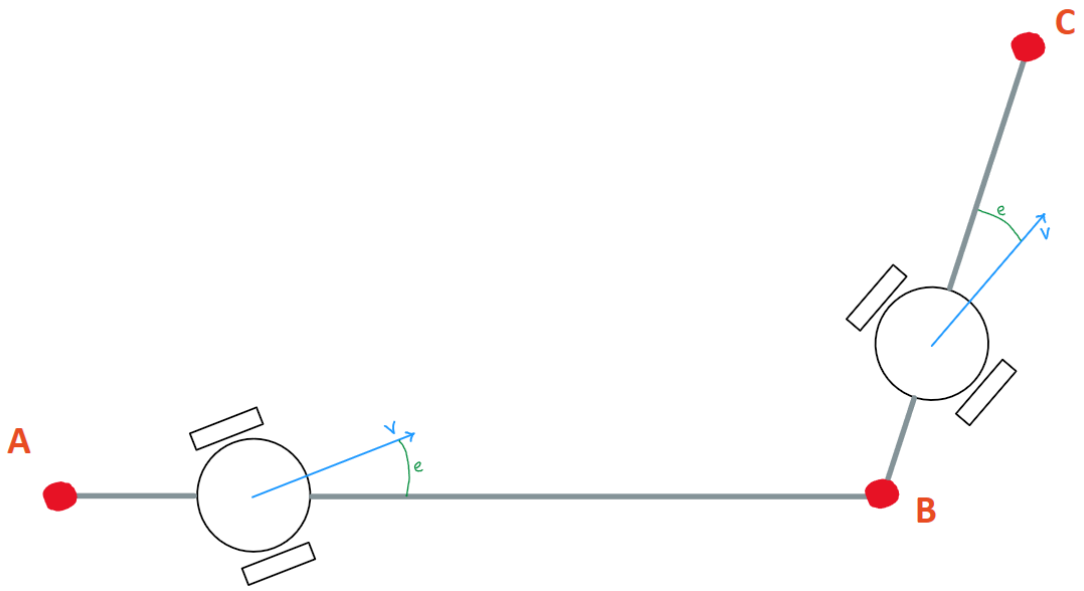


Figura 40 Control de la trayectoria del robot.

```
w = Kw * eom
w = np.clip(w, -self.Wmax, self.Wmax)
wcont = abs(w / self.Wmax)
V = Vdes * (1 - Kv * wcont ** 2)
V, w = self.checkmax(V, w)
```

También ha sido añadido un algoritmo que hace que si el robot se encuentra más cerca del objetivo siguiente que del actual se omite el actual y pasamos directamente al más cercano.

4. EXPERIMENTACIÓN

4.1 Seguimiento de trayectorias en ausencia de obstáculos

Antes de pasar a las pruebas de las diferentes trayectorias a seguir por el Husky, hay que comentar un par de consideraciones/problemas.

El primero de los problemas es que como el modelo del Husky en CoppeliaSim tiene las ruedas lisas esto ocasionaba deslizamientos no deseados, para ello se ha modificado el coeficiente de fricción de las ruedas en CoppeliaSim logrando así minimizar los deslizamientos. Sin embrago, a la hora de la puesta a punto con el Husky real sería conveniente ajustar los parámetros *Kw* y *Kv* para un correcto funcionamiento.

Otro problema es que, al aplicar el controlador en velocidad, si se utilizaba un controlador lineal esto provocaba pequeñas variaciones constantes en la velocidad que podrían llegar a dañar la reductora del motor. Para solucionar este problema se sustituyó el controlador lineal por uno exponencial, consiguiendo así eliminar estas pequeñas variaciones.

Velocidad con el controlador lineal vs con el controlador exponencial:

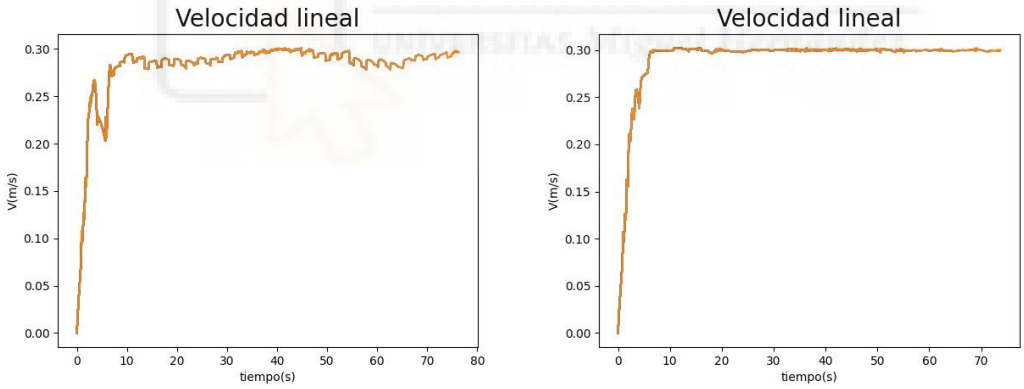


Figura 41 -A la izquierda: Velocidad con controlador lineal. -A la derecha: Velocidad con controlador exponencial.

A continuación, se verá el comportamiento del Husky al realizar el seguimiento de distintas trayectorias libres de obstáculos.

Trayectoria lineal:

Trayectoria lineal generada imponiendo una velocidad media de 0.3 *m/s*. (Comenzando con una orientación del robot diferente a la de la trayectoria)

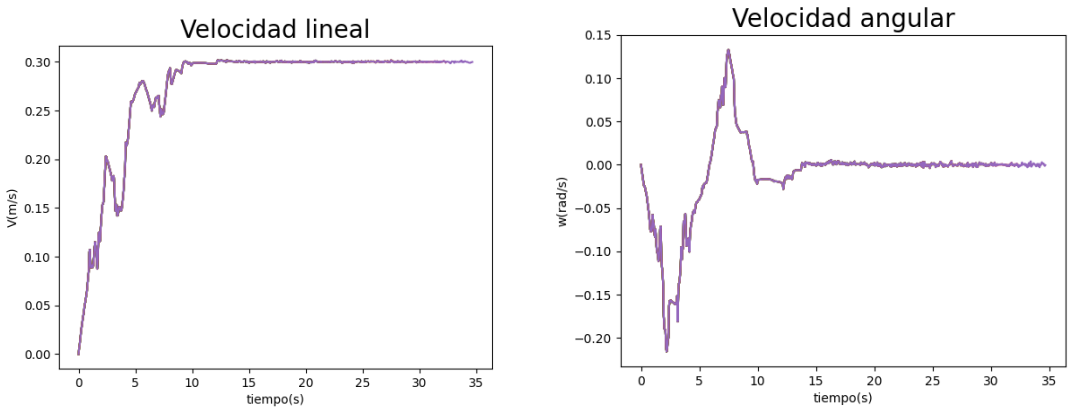


Figura 42 Velocidad lineal y angular del robot a lo largo del tiempo.

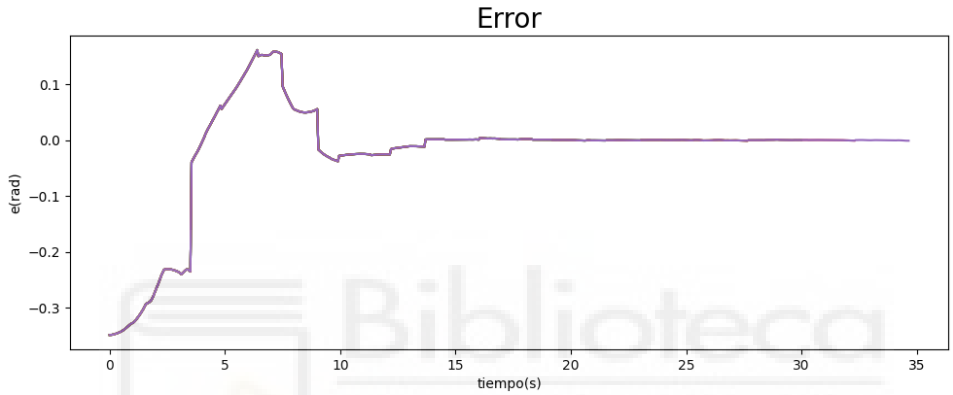


Figura 43 Error en la orientación del robot a lo largo del tiempo.

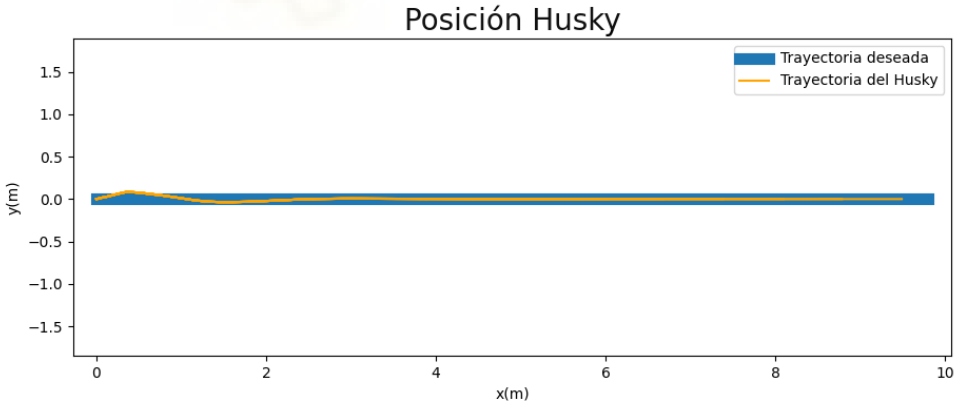


Figura 44 Posición del robot en comparación a la trayectoria deseada del robot.

Como podemos observar en la *Figura 43* Error en la orientación del robot a lo largo del tiempo.el sistema se establece en un tiempo razonable dadas las restricciones de velocidad y aceleración tanto lineales como angulares. Además, como se puede ver en las *Figura 42* Velocidad lineal y angular del robot a lo largo del tiempo.la velocidad lineal se mantiene constante a 0.3 *m/s* cuando el sistema se ha establecido.

Trayectoria sinusoidal:

Trayectoria sinusoidal de 3 m de amplitud imponiendo una velocidad media de 0.3 *m/s*.

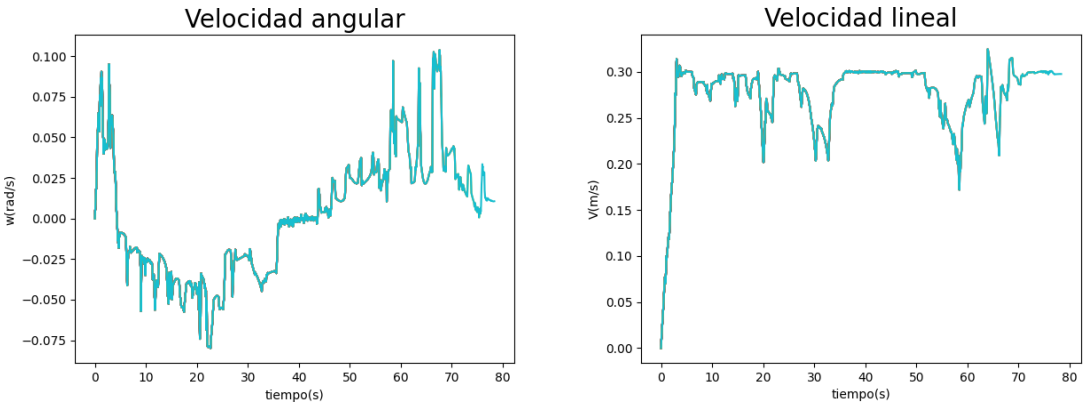


Figura 45 Velocidad lineal y angular del robot a lo largo del tiempo.

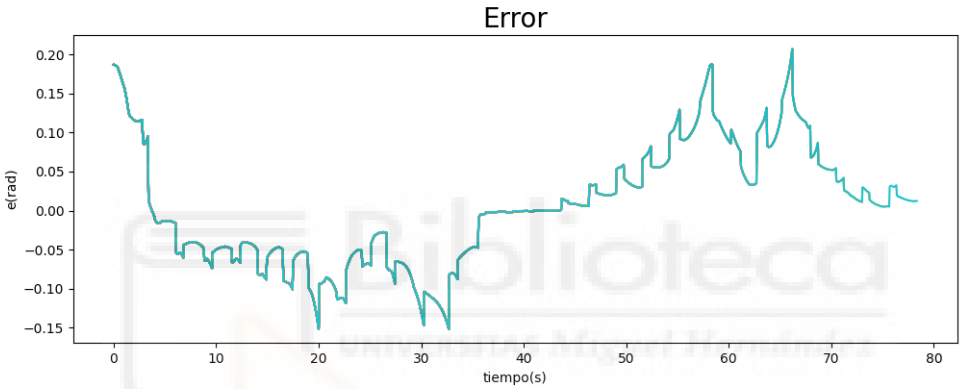


Figura 46 Error en la orientación del robot a lo largo del tiempo.

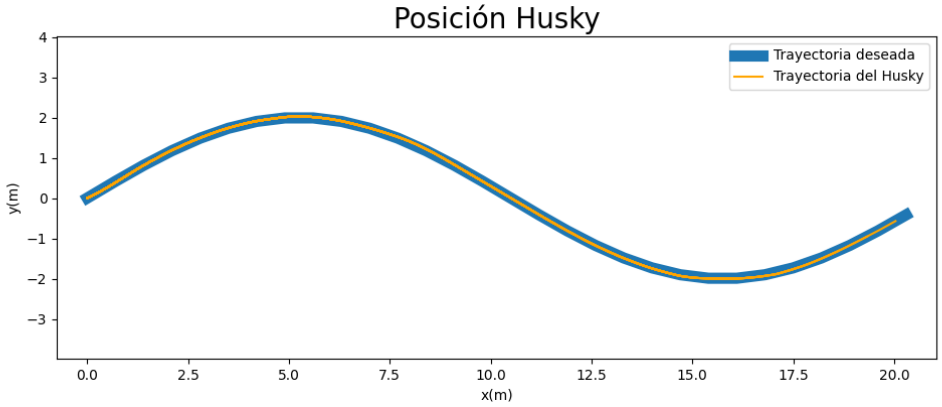


Figura 47 Posición del robot en comparación a la trayectoria deseada del robot.

Trayectoria circular:

Trayectoria circular de 5 *m* de radio imponiendo una velocidad media de 0.3 *m/s*.

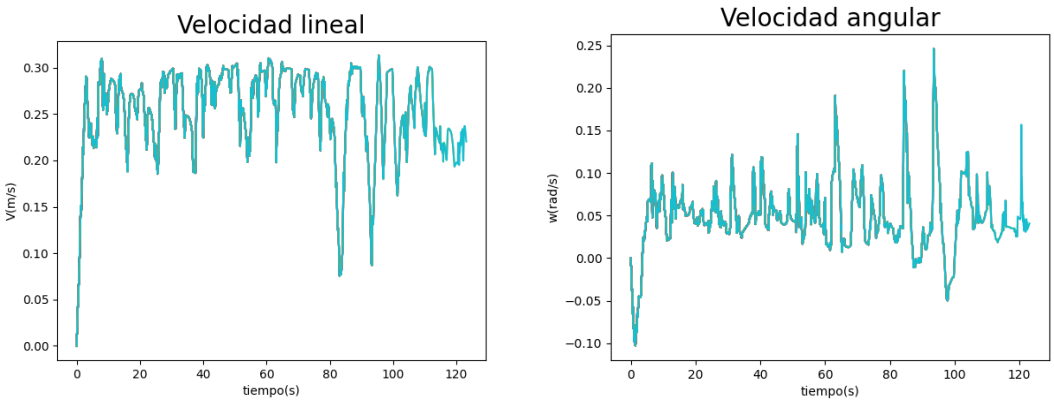


Figura 48 Velocidad lineal y angular del robot a lo largo del tiempo.

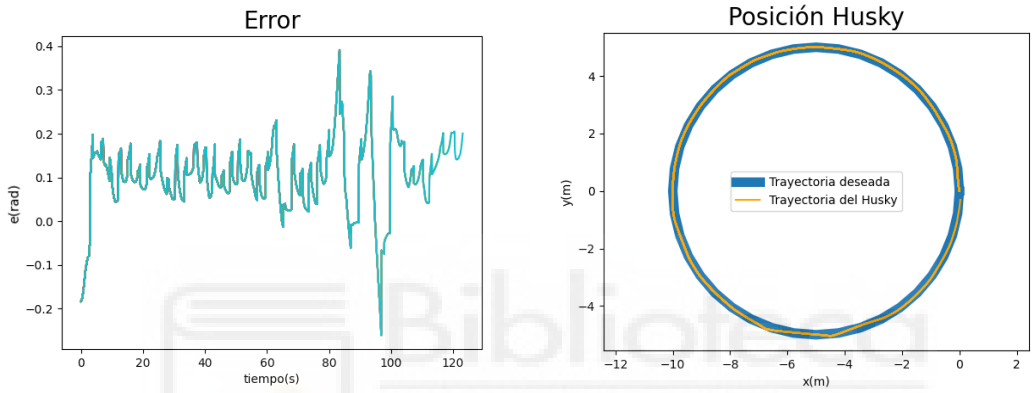


Figura 49 Izquierda: Error en la orientación del robot a lo largo del tiempo. Derecha: Posición del robot en comparación a la trayectoria deseada del robot.

Tanto en la trayectoria sinusoidal como en la circular podemos observar que sigue correctamente la trayectoria, pero nunca se establece la velocidad en 0.3 m/s, esto se debe a que en estas trayectorias el robot está continuamente girando y por el controlador impuesto nunca puede alcanzar esa velocidad.

Trayectoria genérica:

Trayectoria por los caminos de la UMH, más concretamente la descrita en la *Figura* 37, imponiendo una velocidad de 0.3 *m/s*.

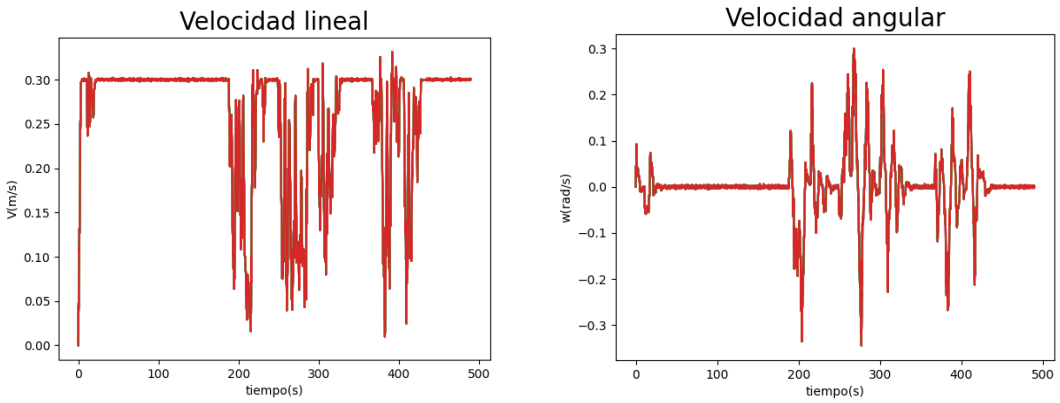


Figura 50 Velocidad lineal y angular del robot a lo largo del tiempo.

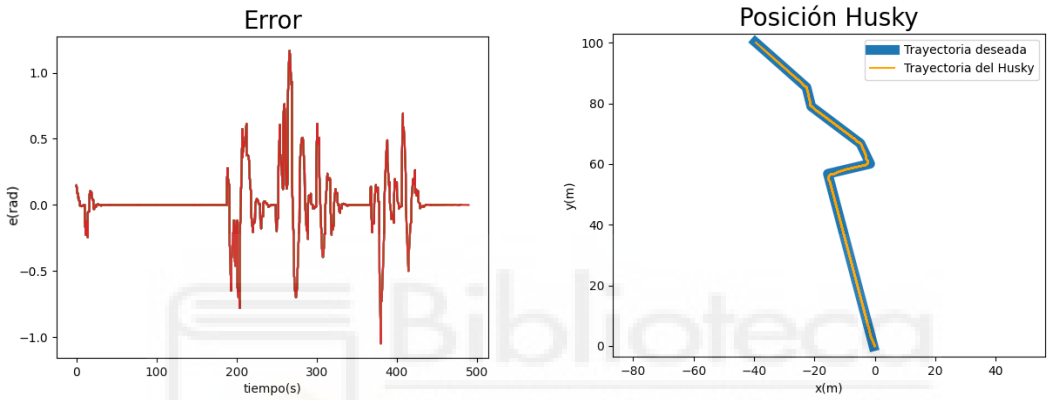


Figura 51 Izquierda: Error en la orientación del robot a lo largo del tiempo. Derecha: Posición del robot en comparación a la trayectoria deseada del robot.

En la trayectoria genérica se puede observar como se ha alcanzado el objetivo perfectamente manteniendo el error nulo y velocidad constante en las rectas.

4.2 Experimentos de planificación

Han sido realizadas varias pruebas para ajustar los parámetros la nube de puntos y del algoritmo RRT-*Connect To Goal*. Se ha establecido inicialmente que, el valor de ε sea igual al tamaño del radio que le hemos dado a la esfera que define el robot y que el máximo número de iteraciones permitidas sea de 5000. Por lo tanto, nos centraremos en obtener la resolución adecuada modificando el tamaño del vóxel.

Un vóxel, es la unidad cúbica más pequeña de un espacio tridimensional, análoga al píxel en una imagen bidimensional. Al igual que un píxel, es la unidad mínima en una imagen digital que contiene información.

Para ello se han realizado pruebas con 3 escenas diferentes y 5 tamaños diferentes de vóxel (la medida se dará por la longitud de su arista).

Escena 1

En esta escena el Husky tiene que sobrepasar un árbol situado en el medio de un camino. Para cada tamaño de vóxel se han hecho 10 iteraciones del algoritmo.

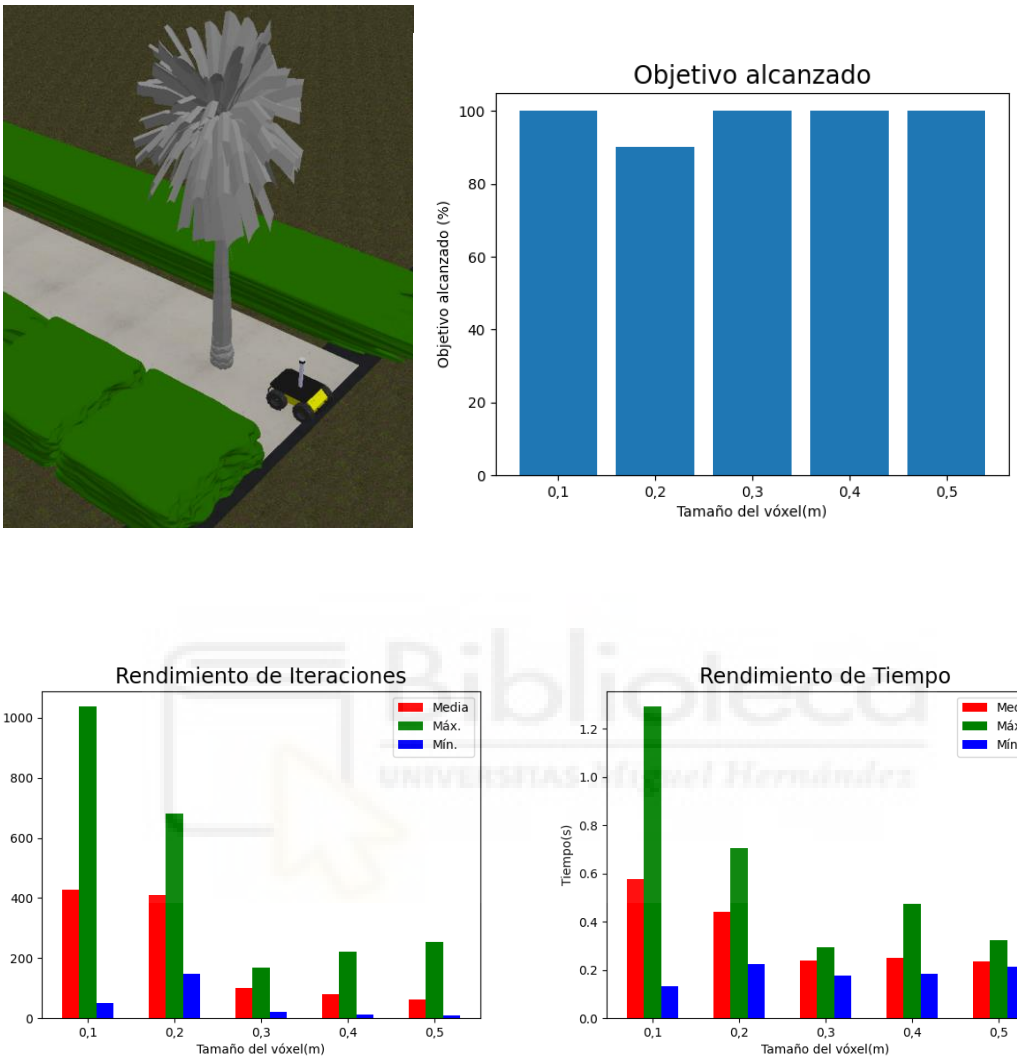


Figura 52 Escena 1. De arriba a la izquierda a abajo a la derecha: Escena de CoppeliaSim, porcentaje de acierto, rendimiento de iteraciones y rendimiento temporal en función del tamaño de vóxel.

Como podemos observar en el primer diagrama de la *Figura 52* Escena 1. De arriba a la izquierda a abajo a la derecha: Escena de CoppeliaSim, porcentaje de acierto, rendimiento de iteraciones y rendimiento temporal en función del tamaño de vóxel.se alcanza el objetivo casi todas las veces y como era esperado, se puede observar en los últimos diagramas de la *Figura 52* Escena 1. De arriba a la izquierda a abajo a la derecha: Escena de CoppeliaSim, porcentaje de acierto, rendimiento de iteraciones y rendimiento temporal en función del tamaño de vóxel.que, cuanto mayor es la resolución, los tiempos de procesamiento y el número de iteraciones aumenta.

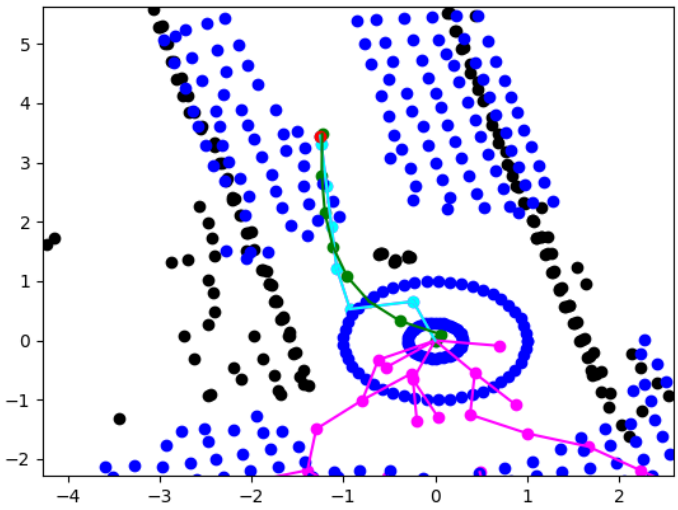


Figura 53 Solución encontrada para un tamaño de vóxel de 0.3 m en la escena 1.

Figura 54 Capturas del robot durante la navegación de la escena 1.

Una de las soluciones encontrada ha sido la de la *Figura 53* Solución encontrada para un tamaño de vóxel de 0.3 m en la escena 1.sin embargo también se han encontrado otras soluciones en las cuales el camino rodeaba el árbol por la parte derecha.

Escena 2

En esta escena el Husky tiene que sortear dos árboles realizando un zigzag. Para cada tamaño de vóxel se han hecho 10 iteraciones del algoritmo.

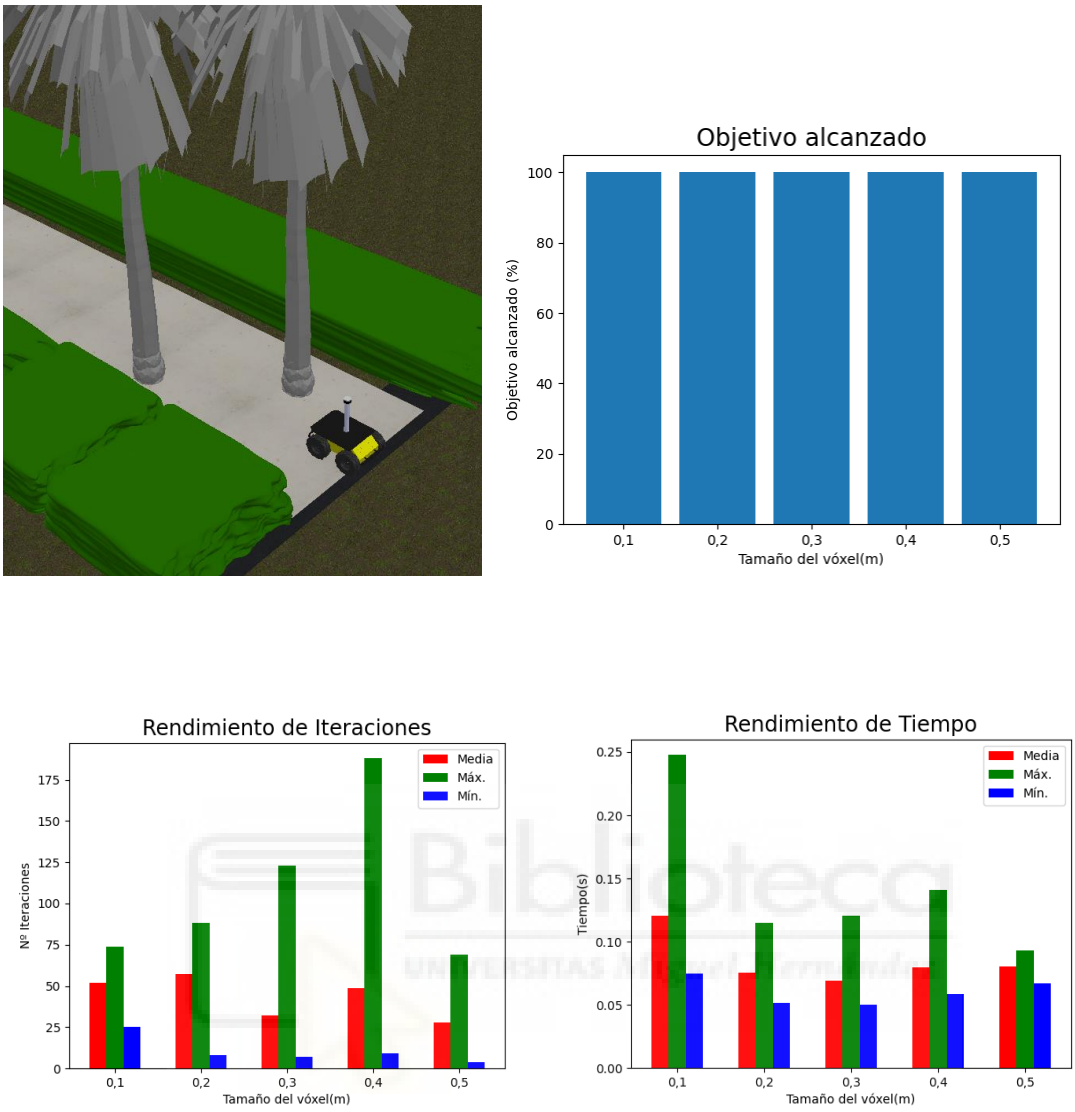


Figura 55 Escena 2. De arriba a la izquierda a abajo a la derecha: Escena de CoppeliaSim, porcentaje de acierto, rendimiento de iteraciones y rendimiento temporal en función del tamaño de vóxel.

Se ha conseguido alcanzar el objetivo el 100% de las veces para todos los tamaños de vóxel y además de una forma más eficiente que en la escena anterior, esto se debe a que cuando hay un obstáculo justo en el medio del camino y este no es muy ancho, el espacio por el que puede pasar sin que colisione con el árbol ni con los arbustos es mucho menor que cuando el árbol está situado a un lado del camino.

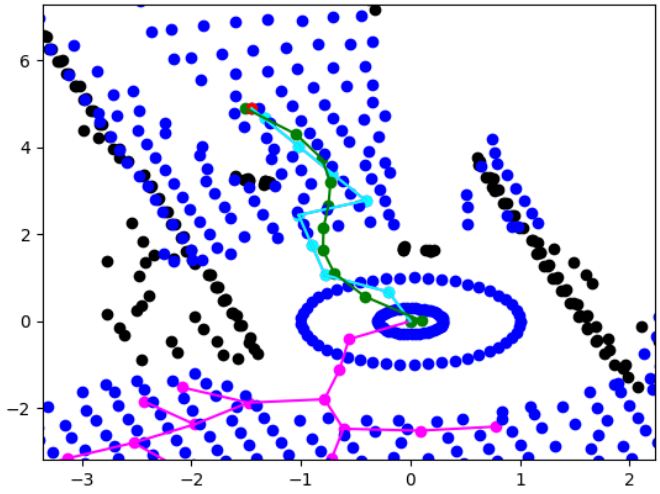


Figura 56 Solución encontrada para un tamaño de vóxel de 0.3 m en la escena 2.

Figura 57 Capturas del robot durante la navegación de la escena 2.

En esta ocasión todas las soluciones encontradas seguían la misma o parecida trayectoria en forma de zigzag sorteando los árboles como se puede observar en la *Figura 56* Solución encontrada para un tamaño de vóxel de 0.3 m en la escena 2.

Escena 3

En esta escena el Husky tiene que atravesar un camino complejo con varios árboles. Para cada tamaño de vóxel se han hecho 10 iteraciones del algoritmo.

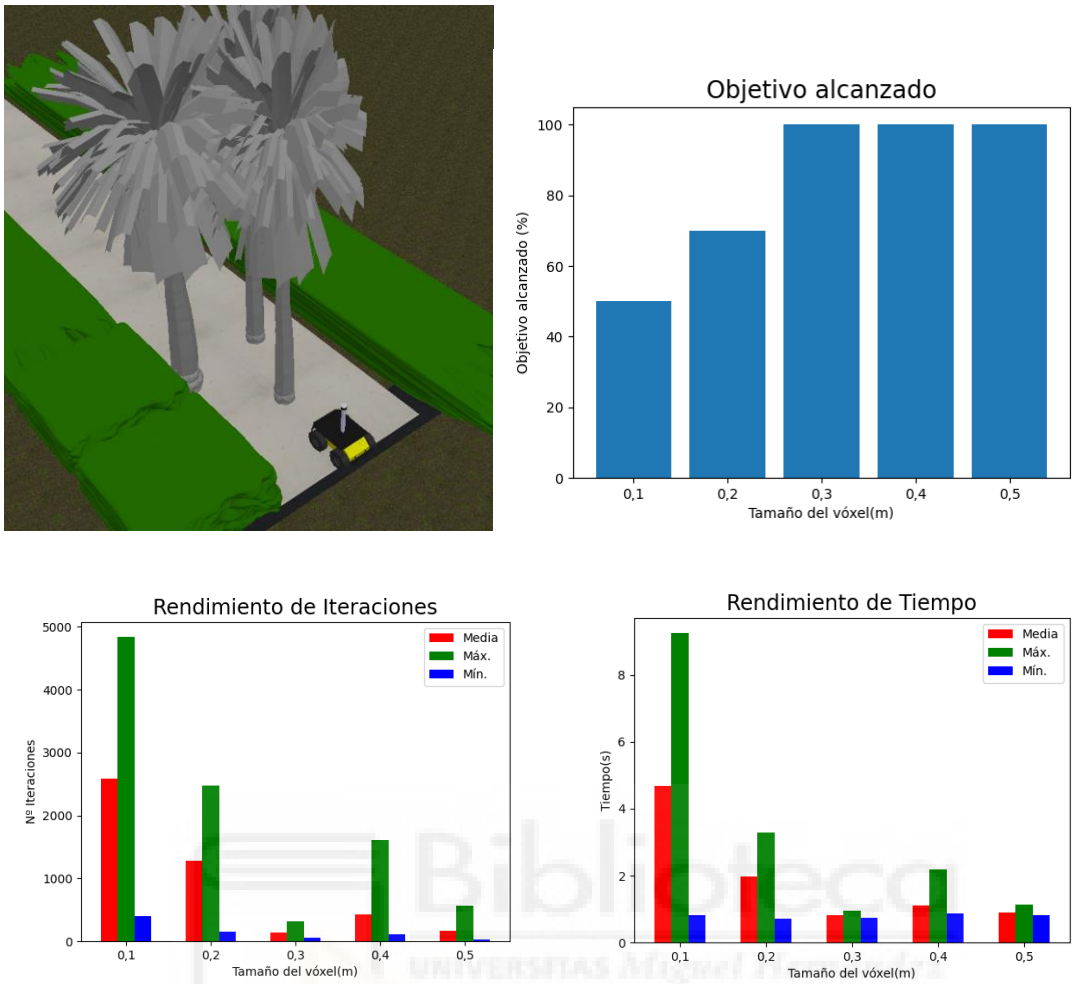


Figura 58 Escena 2. De arriba a la izquierda a abajo a la derecha: Escena de CoppeliaSim, porcentaje de acierto, rendimiento de iteraciones y rendimiento temporal en función del tamaño de vóxel.

La solución obtenida para esta escena para un tamaño de vóxel de 0.3 m es la que se puede encontrar en la *Figura 59* Solución encontrada para un tamaño de vóxel de 0.3 m en la escena 3., sin embargo, ocasionalmente también se ha hallado una solución en la cual se rodea el primer árbol por la derecha, pero esto ocasionaba que el Husky colisionara ligeramente con el siguiente árbol después del suavizado del camino debido al estrecho espacio por el que tenía que pasar. (Vídeo)

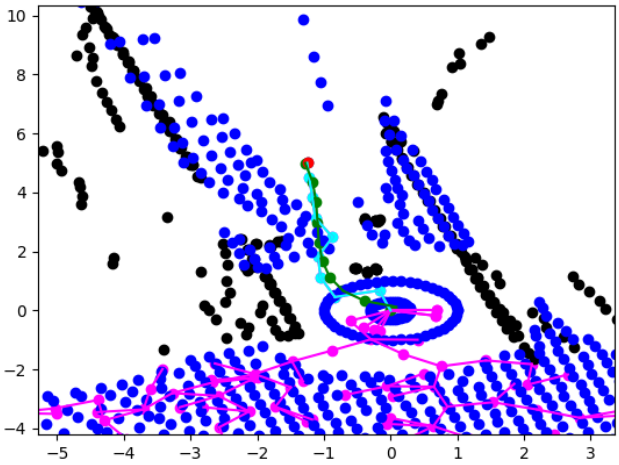


Figura 59 Solución encontrada para un tamaño de vóxel de 0.3 m en la escena 3.

Figura 60 Capturas del robot durante la navegación de la escena 2.

Juntando las 3 escenas obtenemos los siguientes resultados:

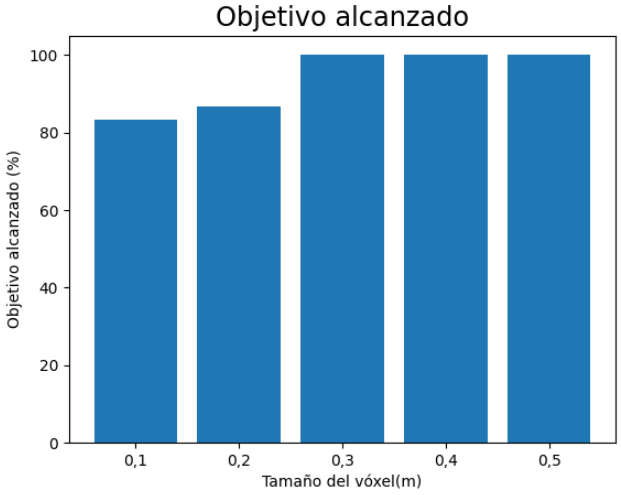


Figura 61 Porcentaje de acierto en función del tamaño de vóxel.

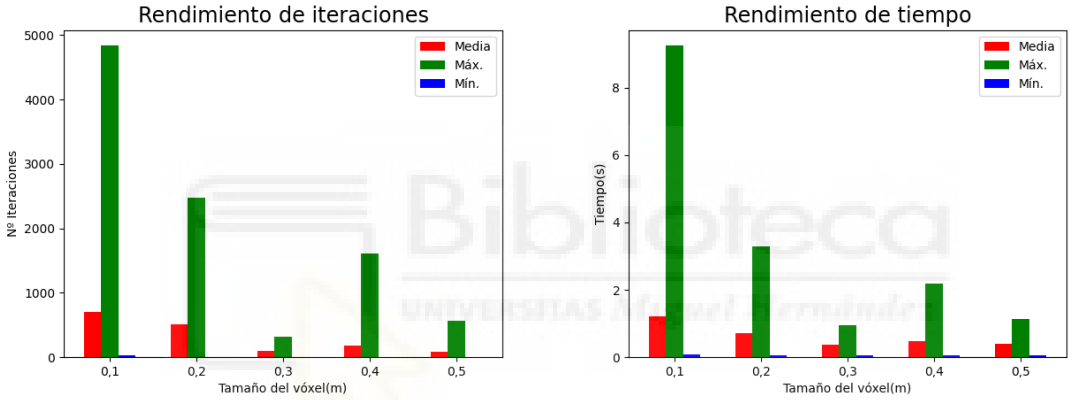


Figura 62 Rendimiento de iteraciones y rendimiento temporal en función del tamaño de vóxel.

Para las resoluciones más altas, en algunas ocasiones no se consigue alcanzar el objetivo, esto se debe a que cuando tenemos un obstáculo enfrente esto genera oclusiones detrás, no marcando el espacio como transitable y por tanto si al hacer el cálculo de colisiones el vóxel es muy pequeño y no logra conectar con el objetivo.

Como se ha comentado anteriormente, en la *Figura 62* Rendimiento de iteraciones y rendimiento temporal en función del tamaño de vóxel.se puede observar como la media de tiempo empleado y la media de iteraciones es mayor cuanto mayor es la resolución.

Como podemos observar, el mejor resultado se ha obtenido para un tamaño de vóxel de 0.3 *m* ya que se consigue llegar al objetivo el 100% de las veces, y los tiempos de procesamiento son bajos. Para un tamaño de vóxel de 0.3 m se obtiene una media de tiempo de 0.376 segundos y 93 iteraciones.

4.3 Pruebas de navegación en simulación

En este apartado se pondrá a prueba todo lo comentado anteriormente, a partir de las coordenadas de inicio y destino, se recorrerán los distintos caminos de la universidad ya sean caminos con una superficie irregular, aceras o delimitados por arbustos, además de obstáculos como palmeras, personas o bancos.

Prueba 1

En esta prueba el Husky tendrá que recorrer un camino delimitado por arbustos, este camino será lo más estrecho posible y el objetivo es que el Husky sea capaz de navegar a través de él sin colisionar en ningún momento con los arbustos.

- La coordenada de inicio será *38.276149, -0.6866967*.
- La coordenada de destino será *38.2763131, -0.6866021*.
- La velocidad comandada será de 0.3 *m/s*.
- El tamaño del vóxel será de 0.3 *m*.
- Caminos de 1.5 metros de ancho.

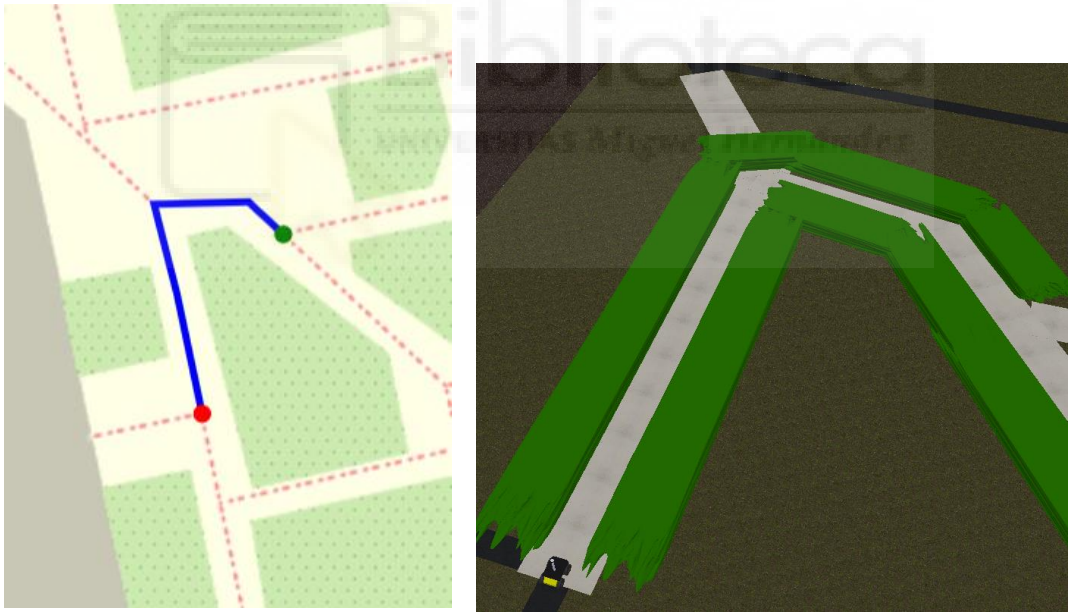


Figura 63 Izquierda: Trayectoria obtenida de OverpassTurbo. Derecha: Escena de CoppeliaSim de la prueba 1.

A continuación, se podrán ver algunas capturas del Husky mientras navega en simulación, se puede encontrar el vídeo completo aquí.

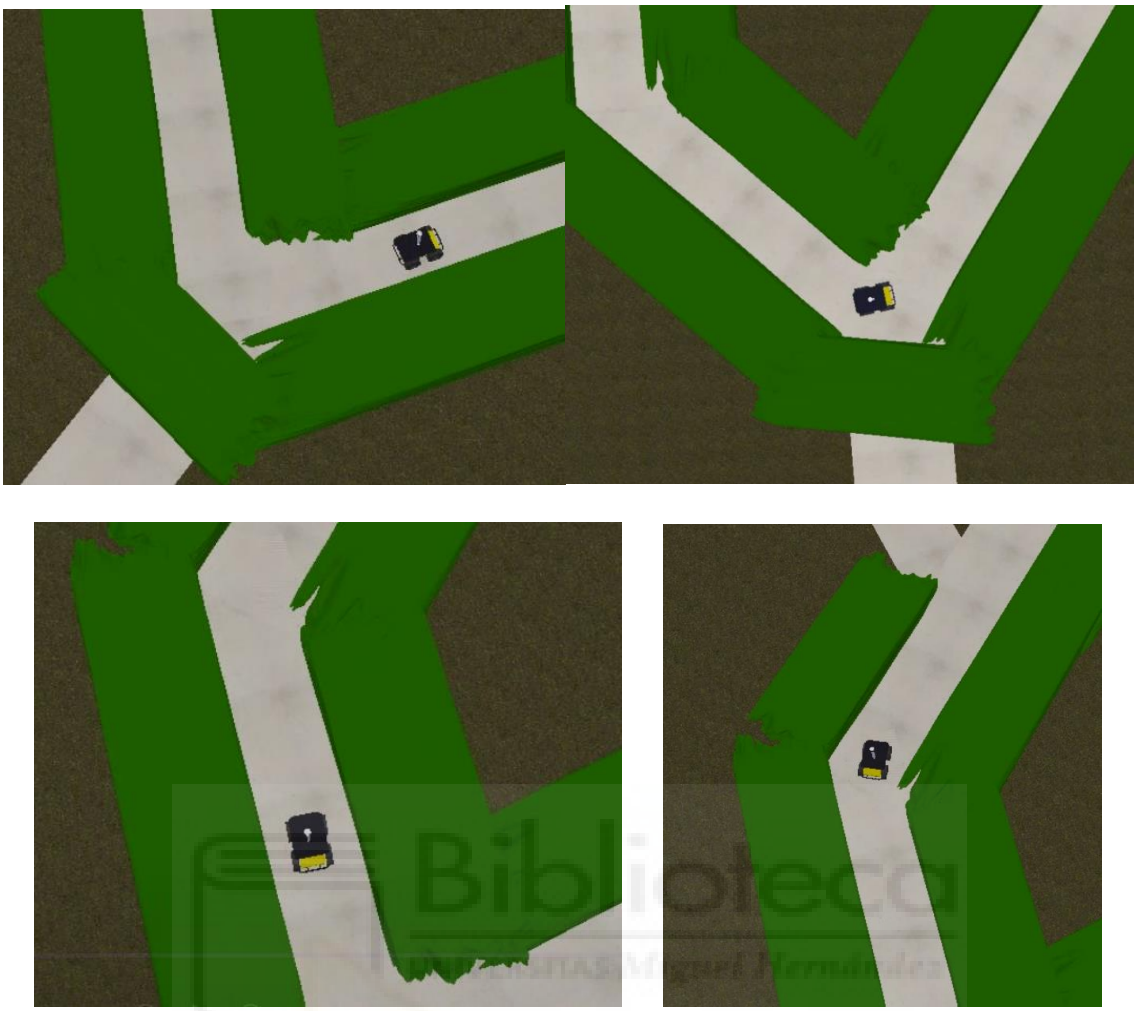


Figura 64 Capturas del robot durante la navegación de la prueba 1.

Las decisiones tomadas por el planificador local son las siguientes, donde los puntos azules son los puntos transitables y los negros los no transitables/obstáculos. Las líneas magenta representan intentos del algoritmo RRT, las cian la solución obtenida y las verdes la solución suavizada.

Como se puede ver en la segunda imagen *Figura* 65 Decisiones tomadas por el planificador local en la navegación de la prueba 1. al algoritmo le ha costado más encontrar la solución adecuada al girar ya que por los arbustos que delimitan el camino, el LiDAR no tenía visión del objetivo y no podía marcar sus alrededores como puntos transitables. Sin embrago se ha encontrado una solución cercana que nos permite seguir con la navegación.

Los nodos del camino inicialmente planteado estaban separados a una distancia de 3 metros, sin embargo, en la tercera imagen de la *Figura* 65 Decisiones tomadas por el planificador local en la navegación de la prueba 1. se puede observar que nuestro objetivo está a unos 6 metros de distancia, esto se debe a que ese nodo que estaba a 3 metros estaba en el rango de colisión con algún obstáculo, en este caso son los arbustos de los laterales.

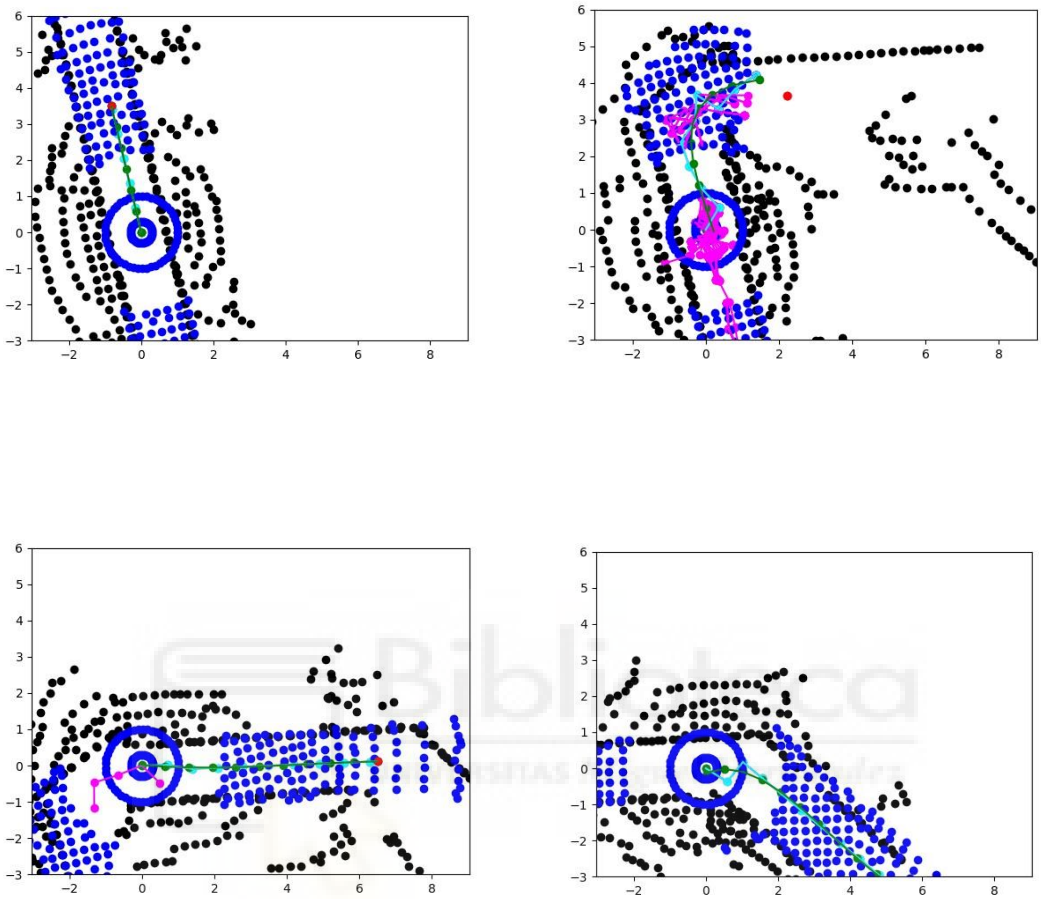


Figura 65 Decisiones tomadas por el planificador local en la navegación de la prueba 1.

En la *Figura* 66 se muestra el recorrido que ha hecho el robot, se puede observar que la trayectoria del Husky no es completamente igual a la trayectoria deseada, esto se debe a que los arbustos no están colocados de forma perfecta sobre el camino, sino que contienen un pequeño error de posición. Sin embargo, esto demuestra que el algoritmo cumple con el objetivo principal de esta prueba que era poder navegar por caminos delimitados y que el robot no colisionara aunque este camino no sea exactamente igual que la trayectoria deseada.

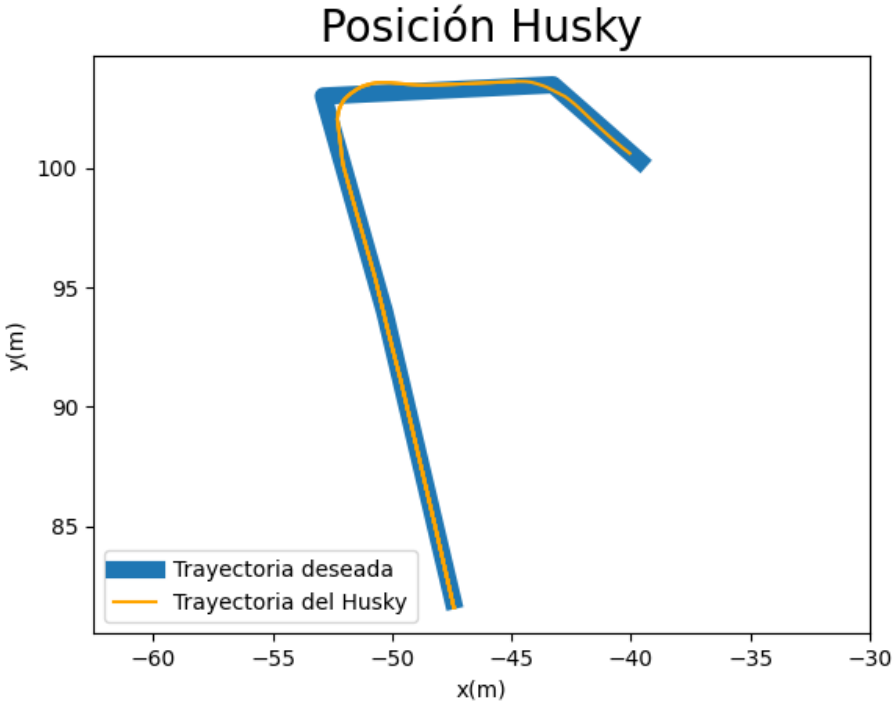


Figura 66 Posición del robot en comparación a la trayectoria deseada del robot durante la navegación de la prueba 1.

Prueba 2

En esta prueba el Husky tendrá que recorrer un camino delimitado por arbustos, este camino será más ancho que el anterior y tendrá varios obstáculos. El objetivo es que el Husky sea capaz de navegar a través de él sorteando los obstáculos eficazmente.

- La coordenada de inicio será *38.2759161*, *-0.6863394*.
- La coordenada de destino será *38.2763131*, *-0.6866021*.
- La velocidad comandada será de 0.3 *m/s*.
- El tamaño del vóxel será de 0.3 *m*.
- Caminos de entre 2 y 3 metros de ancho.

Figura 67 Trayectoria obtenida de OverpassTurbo para la prueba 2.

Figura 68 Escena CoppeliaSim de la prueba 2.

A continuación, se podrán ver algunas capturas del Husky mientras navega en simulación, se puede encontrar el vídeo completo aquí.

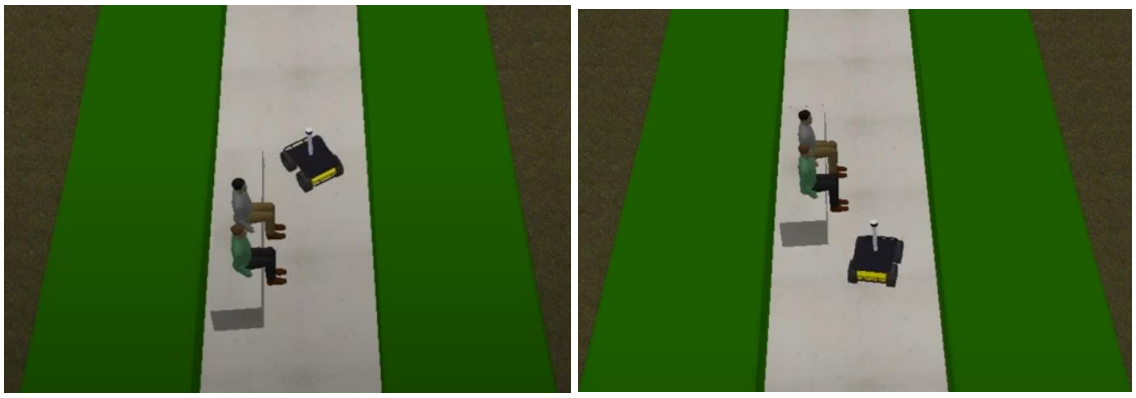
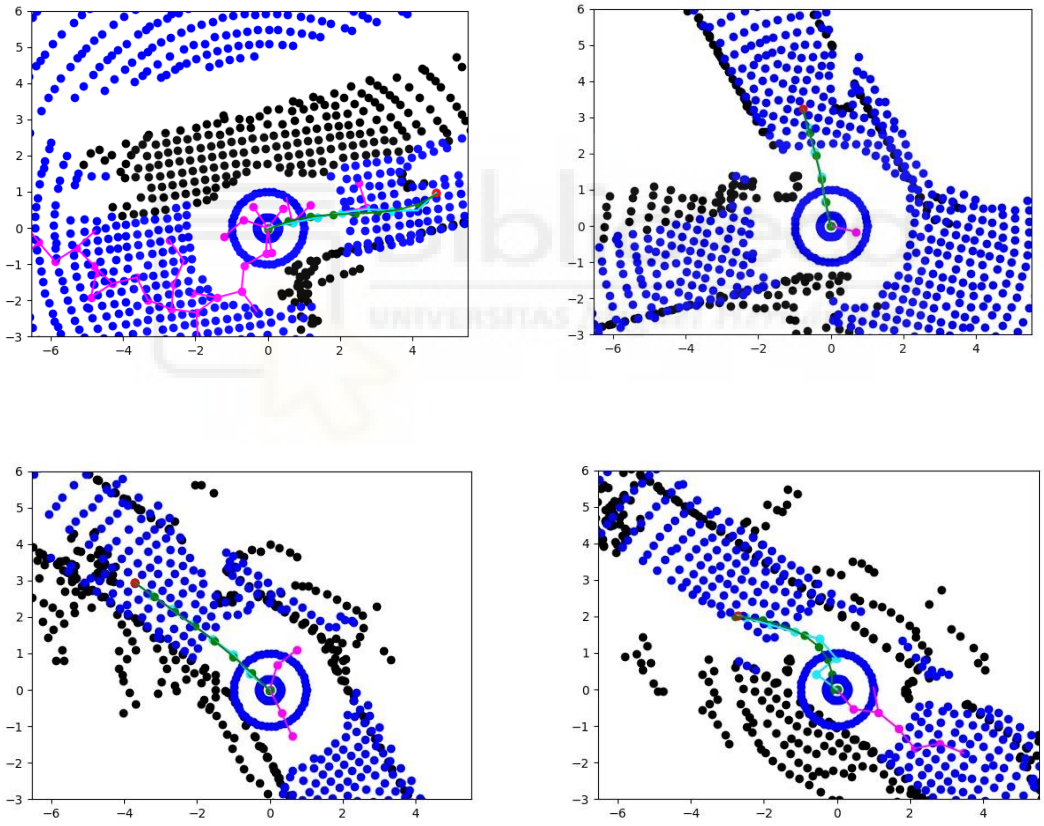


Figura 69 Capturas del robot durante la navegación de la prueba 2.

Algunas de las decisiones que ha tomado el planificador *RRT-Connect To Goal* para evitar los obstáculos son las siguientes: (Aquí vídeo mostrando todas de forma cronológica)



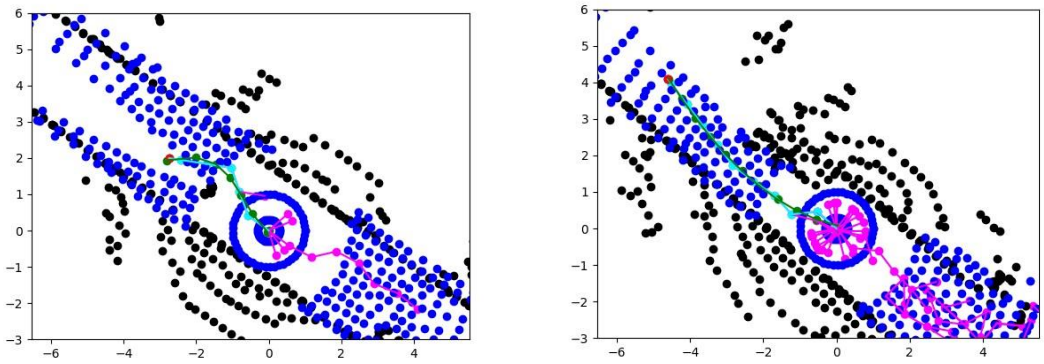


Figura 70 Decisiones tomadas por el planificador local en la navegación de la prueba 2.

Como se puede observar en las imágenes anteriores el algoritmo ha conseguido sortear los obstáculos en todos los casos, sin embargo, en la tercera imagen de la *Figura* 70 se puede ver que nuestro objetivo está bastante cerca de un obstáculo, lo que ha provocado que el robot tenga poco espacio para maniobrar y casi colisiona. (véase tercera imagen de la *Figura 69*)

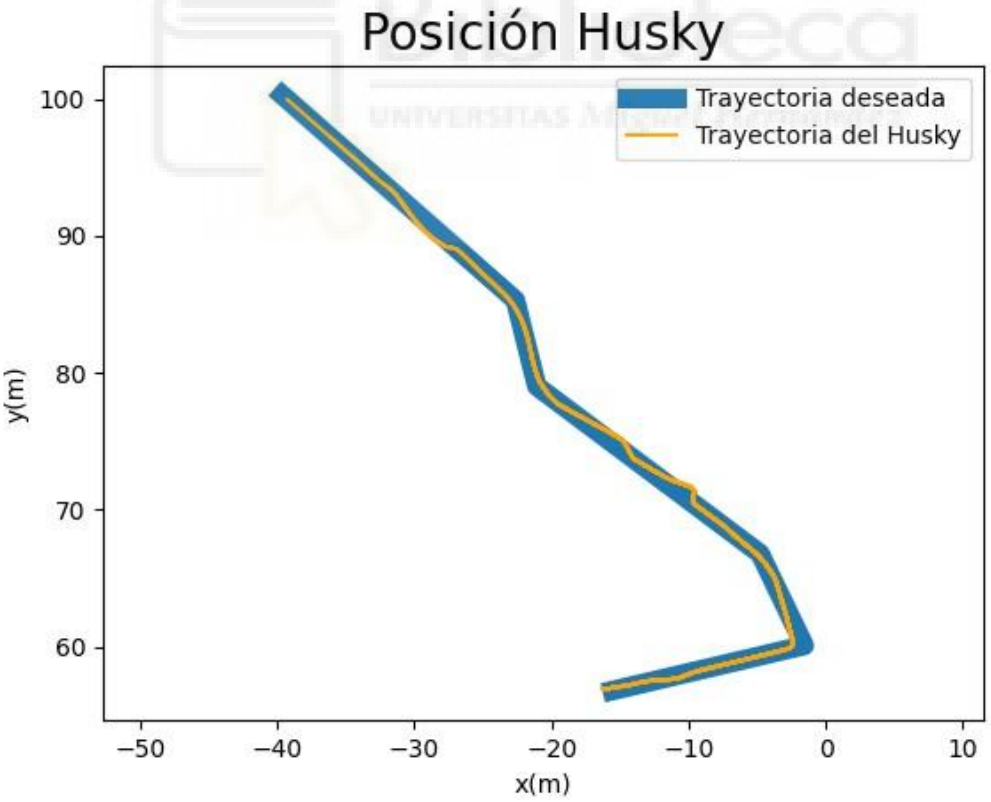


Figura 71 Posición del robot en comparación a la trayectoria deseada del robot durante la navegación de la prueba 2.

En la *Figura* 71 se puede observar que el Husky solo se ha desviado del camino cuando ha tenido que evitar algún obstáculo, el resto del tiempo ha sido capaz de seguir la trayectoria adecuadamente.

Prueba 3

En esta prueba el Husky tendrá que recorrer un camino semi delimitado en el cual hay una aglomeración de personas, este camino será más ancho que el anterior. El objetivo es que el Husky sea capaz de navegar a través sin colisionar con las personas.

- La coordenada de inicio será *38.2759026, -0.6857951*.
- La coordenada de destino será *38.2761001, -0.6855953*.
- La velocidad comandada será de 0.3 *m/s*.
- El tamaño del vóxel será de 0.3 *m*.
- Caminos anchos y semi delimitados.

Figura 72 Trayectoria obtenida de OverpassTurbo para la prueba 3.

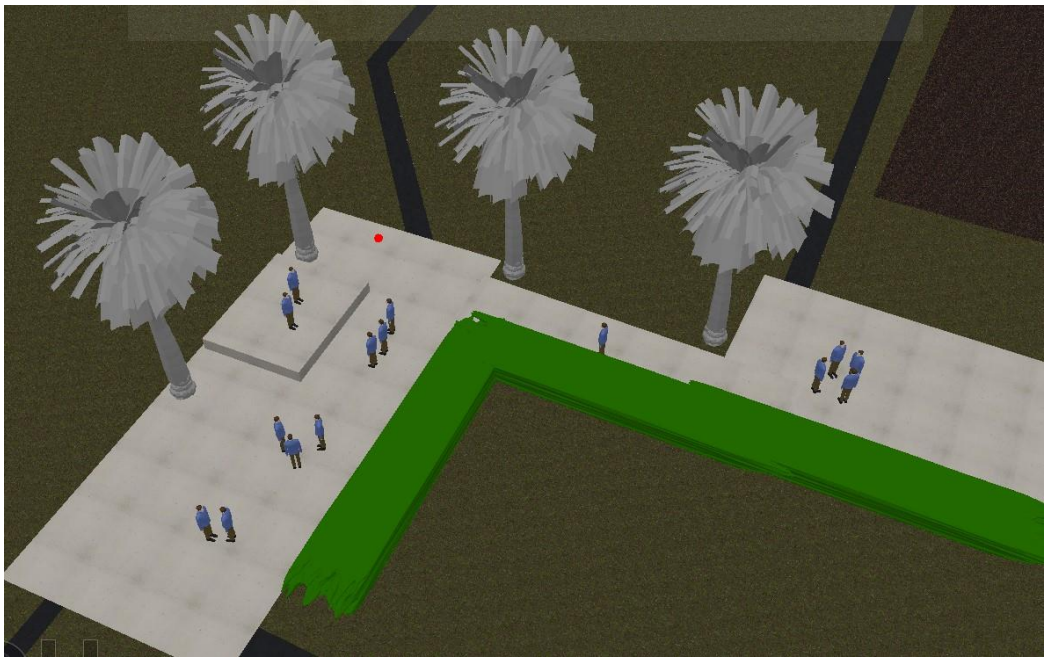


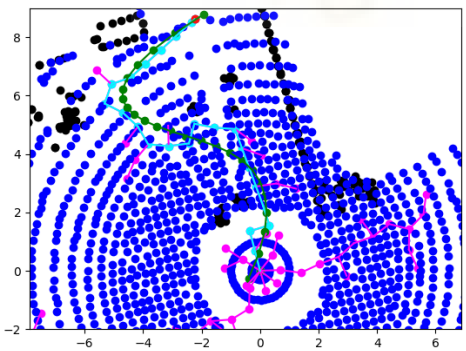
Figura 73 Escena CoppeliaSim de la prueba 3.

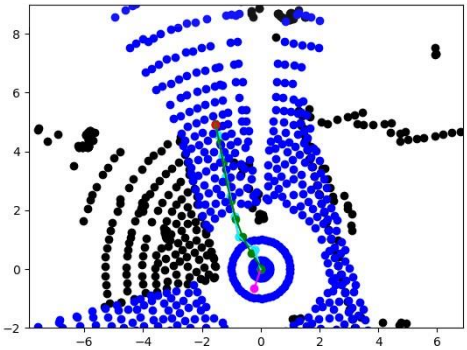
A continuación, se podrán ver algunas capturas del Husky mientras navega en simulación, se puede encontrar el vídeo completo aquí.




Figura 74 Capturas del robot durante la navegación de la prueba 3.

Como podemos observar en la tercera imagen de la *Figura* 74 hay un instante en el que el robot colisiona con el pie de una de las personas, esto se debe a que el pie de la persona no es representado como un obstáculo por su altitud y además como se puede ver en la primera imagen de la *Figura* 75, el robot venía con una orientación muy distinta, lo que ha hecho que no tenga suficiente espacio para maniobrar. Este es uno de los inconvenientes de este tipo de algoritmo y es que en ocasiones no se toma la decisión más lógica desde el punto de vista humano. En este caso lo más lógico quizá hubiera sido continuar con la trayectoria que llevaba y sobrepasar a las personas por detrás, logrando así disminuir ese ángulo tan grande de giro.





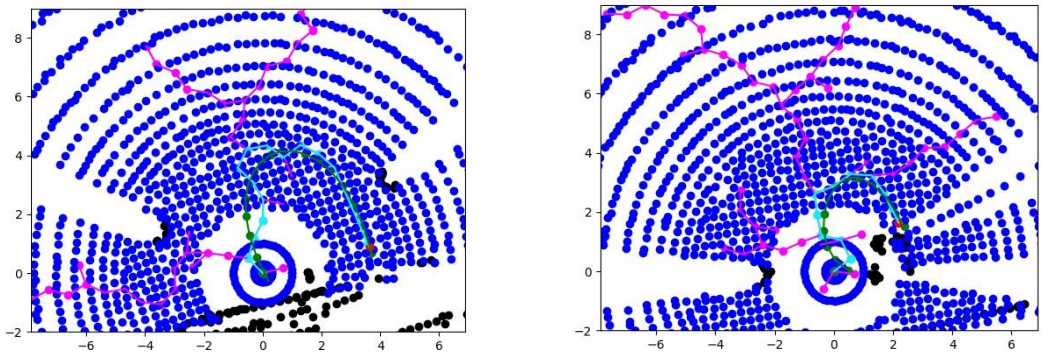


Figura 75 Decisiones tomadas por el planificador local en la navegación de la prueba 3.

Otro inconveniente del algoritmo lo podemos encontrar en la *Figura* 75 y *Figura* 76, como el camino no está completamente delimitado, cuando no hay ninguna diferencia de altura en los bordes, el LiDAR detecta todo el espacio como transitable lo que provoca que nos salgamos del camino momentáneamente a la hora de evitar un obstáculo.

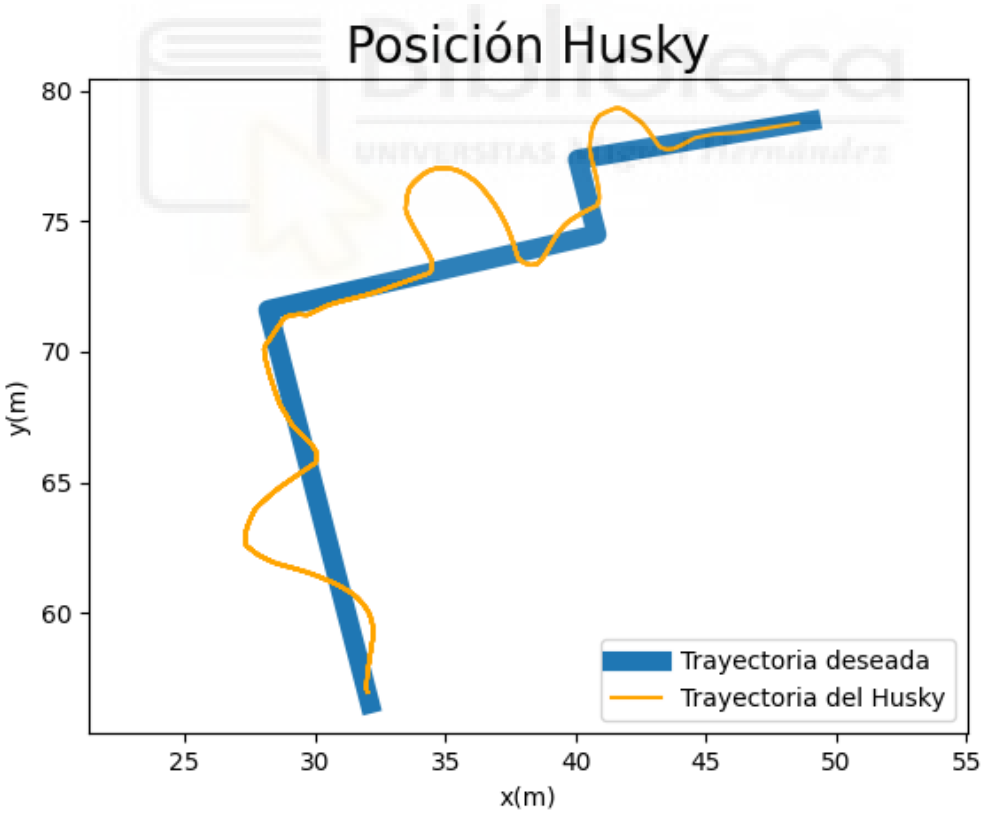


Figura 76 Posición del robot en comparación a la trayectoria deseada del robot durante la navegación de la prueba 3.

Prueba 4

En esta prueba el Husky tendrá que recorrer un camino por encima de una acera elevada en la que en los laterales no deberán ser transitables ya que hay una altura suficientemente alta. El objetivo es que el Husky sea capaz de navegar sin salirse de la acera ni colisionar con los obstáculos.

- La coordenada de inicio será *38.2769747, -0.6863856*.
- La coordenada de destino será *38.2767485, -0.6862136*.
- La velocidad comandada será de 0.3 *m/s*.
- El tamaño del vóxel será de 0.3 *m*.
- Caminos de alrededor de 2 metros de ancho a una altura de 10 cm.

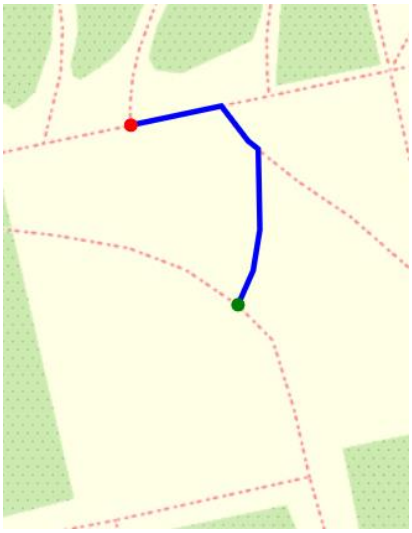


Figura 77 Trayectoria obtenida de OverpassTurbo para la prueba 4.

Figura 78 Escena CoppeliaSim de la prueba 4.

A continuación, se podrán ver algunas capturas del Husky mientras navega en simulación, se puede encontrar el vídeo completo aquí.

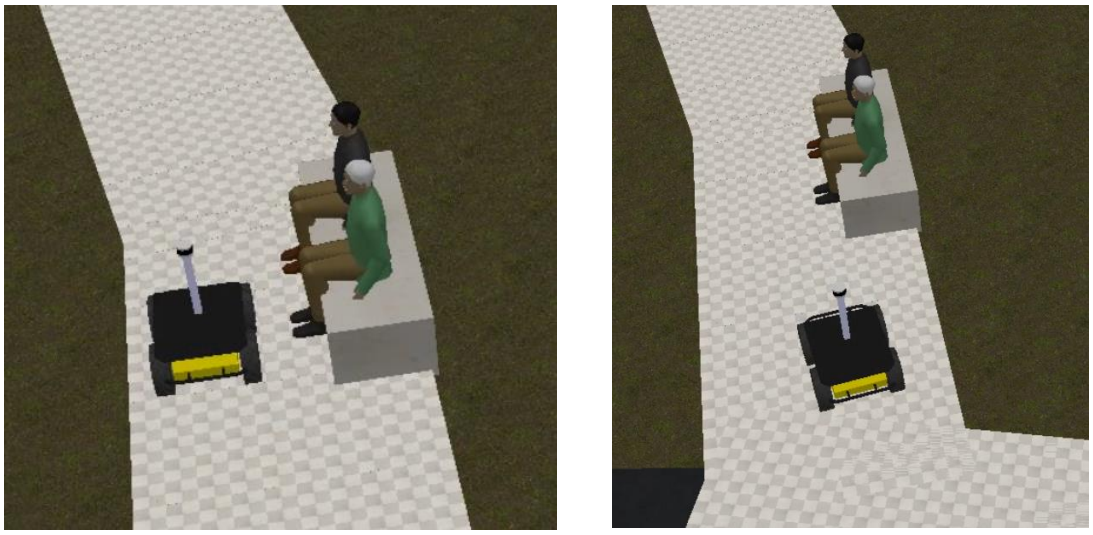


Figura 79 Capturas del robot durante la navegación de la prueba 4.

Como podemos observar en la *Figura* 80 el algoritmo es perfectamente capaz de identificar el camino y marca como espacio no transitable aquella superficie que no tenga la altura adecuada además de sortear los obstáculos y realizar la planificación sin salirse del camino.

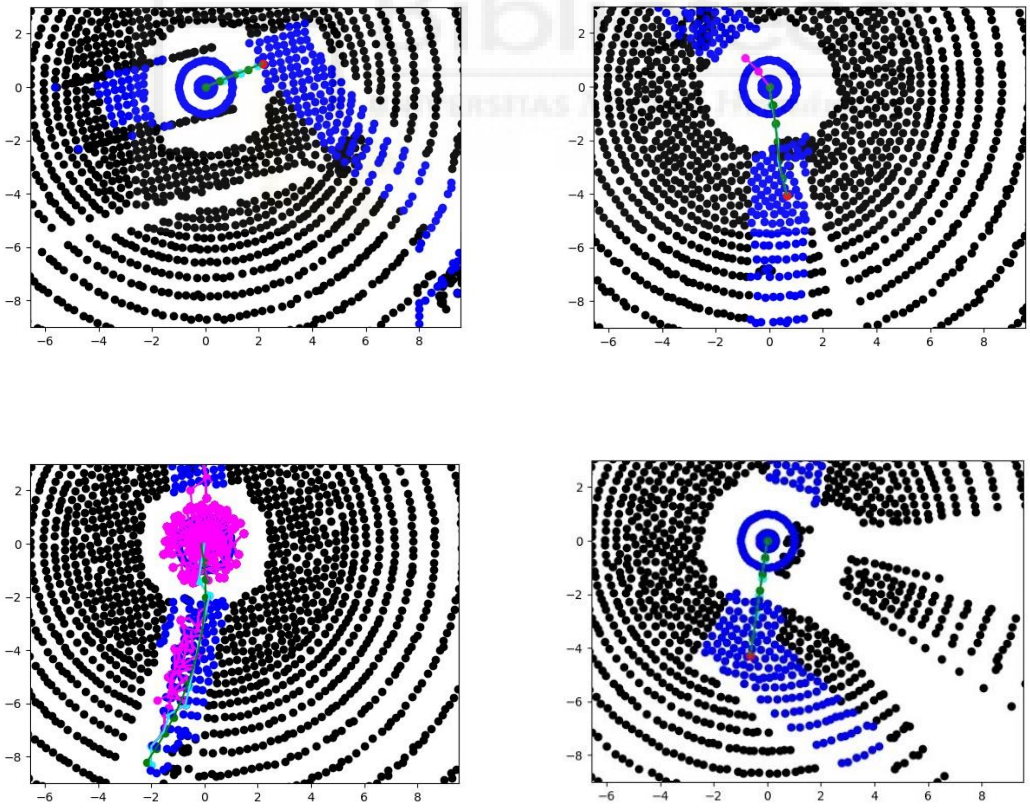


Figura 80 Decisiones tomadas por el planificador local en la navegación de la prueba 4.

Como en pruebas anteriores, podemos comprobar que, aunque la trayectoria deseada no concuerde con el camino a recorrer (véase *Figura* 81), el robot es capaz de llegar al destino evitando descarrilarse y salirse del camino real siempre y cuando este tenga los bordes bien definidos.

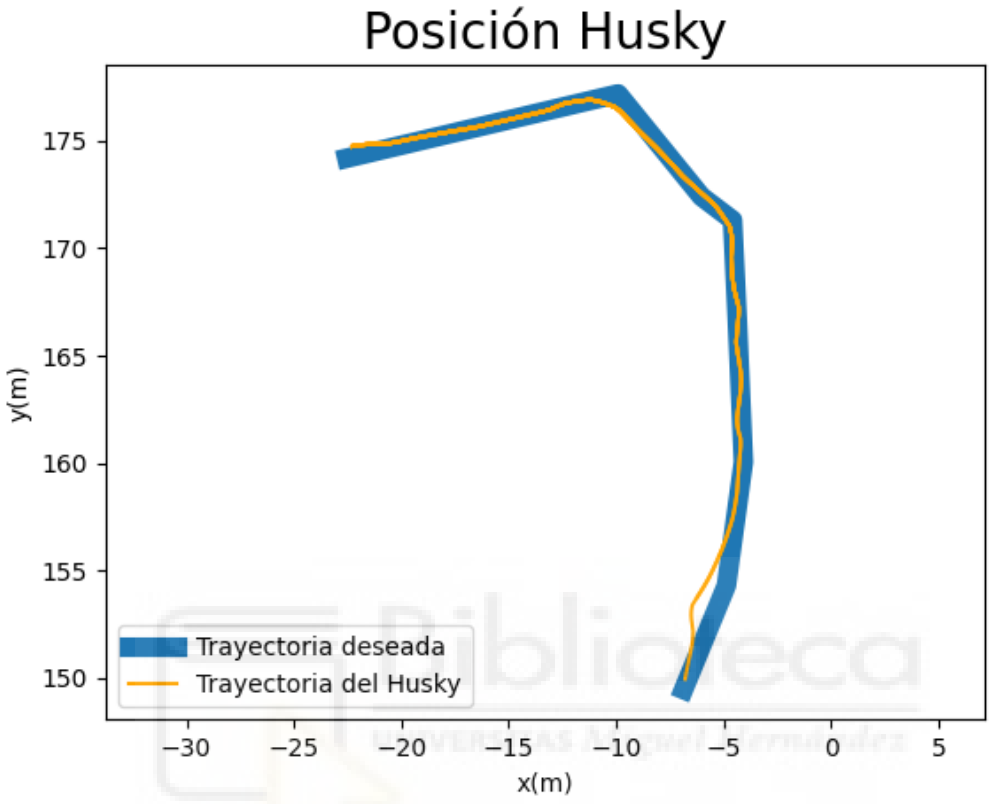


Figura 81 Posición del robot en comparación a la trayectoria deseada del robot durante la navegación de la prueba 4.

Prueba 5

En esta prueba el Husky tendrá que recorrer un camino irregular con pequeñas inclinaciones y obstáculos. El objetivo es que el Husky sea capaz de navegar y llegar al destino a pesar de las condiciones del terreno sin colisionar.

- La coordenada de inicio será *38.2770715, -0.6863711*.
- La coordenada de destino será *38.2773684, -0.6864113*.
- La velocidad comandada será de 0.3 *m/s*.
- El tamaño del vóxel será de 0.3 *m*.
- Caminos irregulares.

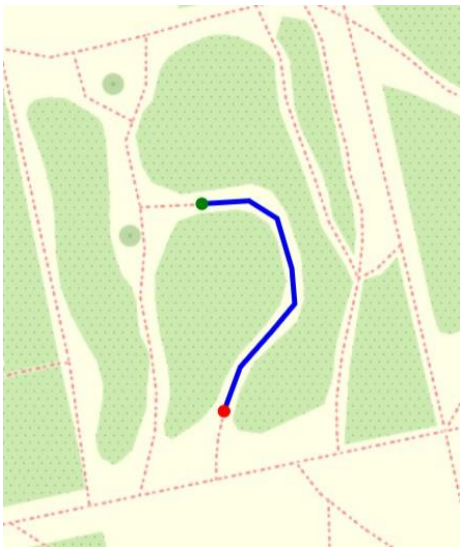


Figura 82 Trayectoria obtenida de OverpassTurbo para la prueba 5.

Figura 83 Escena CoppeliaSim de la prueba 5.

A continuación, se podrán ver algunas capturas del Husky mientras navega en simulación, se puede encontrar el video completo aquí.

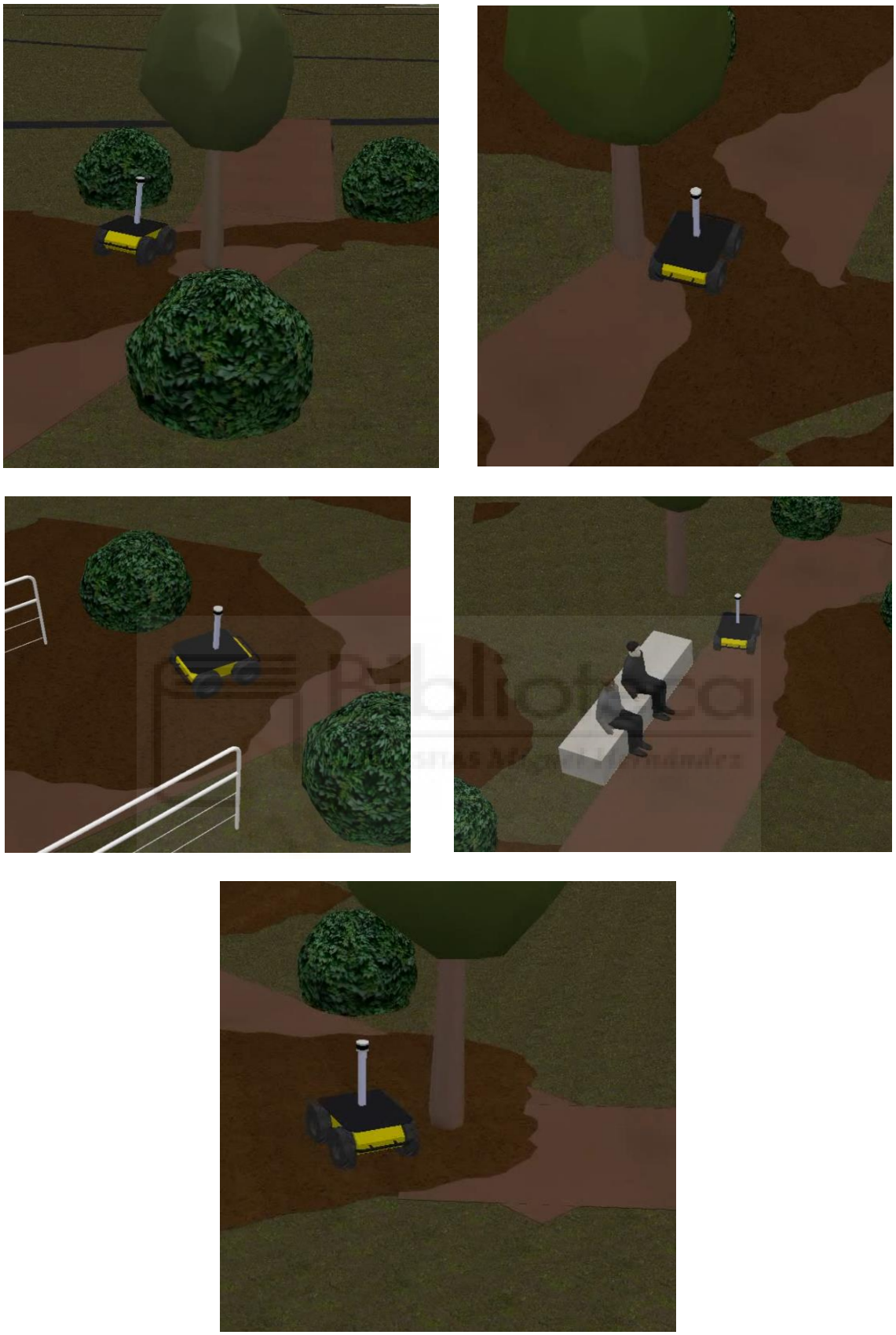


Figura 84 Capturas del robot durante la navegación de la prueba 5.

Cabe destacar que para este escenario se ha modificado el *threshold* de altura para el cual se detectan obstáculos. Esto se hace ya que como el terreno tiene altibajos, cuando estamos encima de uno, el terreno por debajo del robot podría ser considerado como obstáculo dificultando así la navegación. Es conveniente modificar este *threshold* dependiendo del terreno y el tipo de camino a navegar para obtener un rendimiento óptimo.

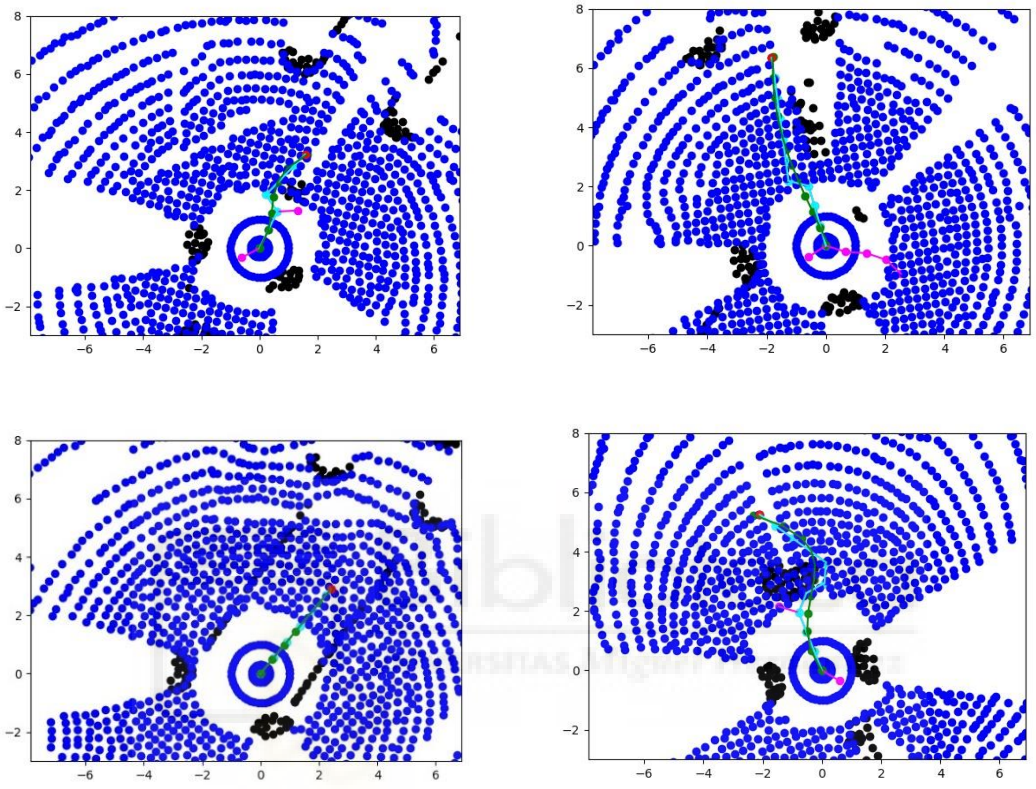


Figura 85 Decisiones tomadas por el planificador local en la navegación de la prueba 5.

Pese a los cambios de altura el Husky ha sido capaz de alcanzar el objetivo detectando y esquivando correctamente los obstáculos manteniéndose en la medida de lo posible sobre la trayectoria deseada comprobando así que ajustando los parámetros adecuados

el programa puede comportarse de manera satisfactoria en distintos entornos.

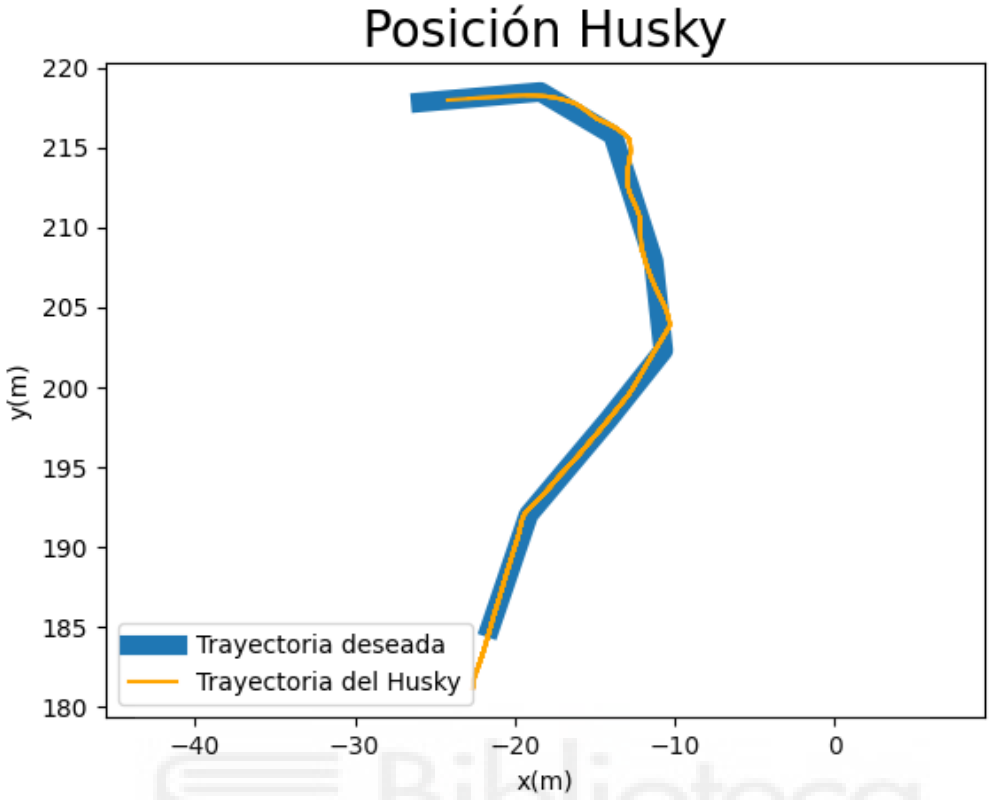


Figura 86 Posición del robot en comparación a la trayectoria deseada del robot durante la navegación de la prueba 5.

5. CONCLUSIONES

El trabajo presentado se centra en la implementación de un sistema de navegación autónomo para un robot móvil, específicamente el Husky, utilizando planificación de trayectorias basada en RRT y nubes de puntos LiDAR.

Se han probado diversas trayectorias, desde lineales hasta sinusoidales y circulares en ausencia de obstáculos. Estas pruebas han validado la capacidad del sistema para seguir trayectorias predefinidas con alta precisión, incluso cuando la orientación inicial del robot difiere de la dirección de la trayectoria.

En pruebas que simulan entornos más realistas, se ha añadido un módulo capaz de replanificar la trayectoria para la evasión de obstáculos basado en el algoritmo RRT-*Connect To Goal* sobre nubes de puntos. En los caminos de la universidad con obstáculos como árboles, palmeras, personas y bancos, el sistema ha demostrado una robusta capacidad de navegación, evitando colisiones y sorteando obstáculos de manera eficiente.

A pesar de los resultados positivos obtenidos, hay áreas donde se pueden realizar mejoras adicionales para aumentar la robustez y la versatilidad del sistema:

-Inclusión de algoritmos que permitan que el robot no colisione ante situaciones imprevistas, como algoritmos que hagan que el robot se detenga si está demasiado cerca de un obstáculo o un segundo cálculo de las colisiones siguiendo el camino después de este haber sido suavizado.

-Integración de sensores adicionales para poder discernir lo que es un camino y lo que no, ya que, si el camino no está delimitado por muros/arbustos o diferencias de altura, el algoritmo no es capaz de definir los límites de este.

- Para la transición del entorno de simulación a un entorno real, es crucial ajustar y calibrar los parámetros específicos del Husky, tanto aquellos relacionados con el control de velocidad y la fricción de las ruedas como los de la planificación en función del tipo de terreno a navegar. Esto garantizará que el rendimiento observado en la simulación se traduzca de manera efectiva al mundo real.

En conclusión, el trabajo realizado ha establecido una base sólida para la navegación autónoma de robots móviles utilizando RRT y LiDAR. Las mejoras en los algoritmos de planificación y control, así como las pruebas en simulaciones, demuestran el potencial de esta tecnología para aplicaciones reales en entornos complejos. Las futuras investigaciones y desarrollos deberían centrarse en la optimización continua y la validación en condiciones reales para alcanzar un sistema de navegación completamente autónomo y confiable.

6. ANEXOS

I. Código de ejemplo de ejecución del programa.

```
# Start simulation
simulation = Simulation()
simulation.start()
# Connect to the robot
robot = HuskyRobot(simulation=simulation)
robot.start(base_name='/HUSKY')
# Get robot parameters
robot.getParams()
# Simulate a LiDAR
lidar = Ouster(simulation=simulation)
lidar.start(name='/OS1')
# Simulate center of Husky robot
base = CoppeliaObject(simulation=simulation)
base.start(name='/CentroHusky')
# Coordinate 0
coord0 = [38.275401, -0.686178]
# Coordinate of our origin point
origin_point = [38.2763131, -0.6866021]
# Coordinate of our destination point
destination_point = [38.2763131, -0.6866021]
# Getting the nodes of the path found
data = getTrajNodes(origin_point, destination_point)
zh = 0.243  # Height of the center of the robot
traj = []
# We convert lat/long coordinates to x,y
for i in range(len(data)):
    x, y = tr.convertCoordinates(coord0, data[i])
    z = zh
    traj.append([x, y, z])
# Linear interpolation of the nodes
newtraj = tr.interpTraj(traj, 3)
# Spline interpolation on the curves to smoothen the path
goal = tr.SplinTraj(newtraj, traj)
voxel_size = 0.3

for i in range(len(goal)):
    # Convert nodes to robot's local coordinates
    obj = tr.CoordsToLocal(base, goal[i])
    # Get LiDar data
    data_lidar = get_lidar_data(lidar, base, voxel_size)
    # Find RRT path
    li_traj, valid, _, _ = find_path_connect_to_goalPC(data_lidar,
    obj, base.get_position(), voxel_size)
    if not valid:
        continue
    # Convert path to Global coordinates
    glb_traj = []
    for punto in li_traj:
        glb_traj.append(tr.CoordsToGlobal(base, punto))
    # Follow path found
    robot.goto_objective(puntos=glb_traj, base=base, velocity=0.3, skip=True)
simulation.stop()
```

7. BIBLIOGRAFIA

-Introducción

- **Introduction to Autonomous Mobile Robots**. MIT Press. Siegwart, R., Nourbakhsh, I.R.
- https://arxiv.org/abs/2206.03223
- https://www.intechopen.com/chapters/16161
- https://en.wikipedia.org/wiki/Mobile_robot
- https://robotnik.eu/mobile-robotics-see-the-advantages-it-brings-to-your-sector/
- https://docs.emlid.com/reachrs3/rtk-quickstart/rtk-introduction/

-Algoritmos de planificación

- **BTO-RRT: A rapid, optimal, smooth and point cloud-based path planning algorithm.** Zhaoliang Zheng, Member, IEEE, Thomas R. Bewley, Falko Kuester, and Jiaqi Ma, Member, IEEE.
- **RRT-Connect: An Efficient Approach to Single-Query Path Planning**. James J. Kuffner, Steven M. LaValle.
- **Rapidly-Exploring Random Trees: A New Tool for Path Planning**. Steven M. LaValle
- https://www.almabetter.com/bytes/tutorials/artificial-intelligence/a-star-algorithm-in-ai
- https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra

-Implementación

- https://coppeliarobotics.com/
- https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/
- https://wiki.ros.org/Robots/Husky
- https://en.wikipedia.org/wiki/Differential_wheeled_robot
- https://www.felix-beck.de/nyuad/teaching/doku.php?id=wiki:osm_3d_data
- https://ieeexplore.ieee.org/document/7068785
- https://ieeexplore.ieee.org/document/10111290