UNIVERSIDAD D SEVILLA 1505

Planificador de trayectorias basado en el diagrama de Voronoi

Robótica Móvil y de Servicios

Moreno Ordoñez, Santiago

Parra Montes, Jesús

Iniesta Fernández, José Manuel

Quarto Di Palo, Domenico

Máster en Ingeniería Electrónica, Robótica y Automática

Escuela Técnica Superior de INGENIERÍA DE SEVILLA

European Accreditation of Engineering Programmes EUR-ACE®

Introducción

En el ámbito de la robótica aérea, la planificación de trayectorias es una tarea fundamental para garantizar que los vehículos puedan navegar de manera segura y eficiente en entornos tridimensionales. Una técnica popular para abordar este desafío es la utilización de diagramas de Voronoi, los cuales permiten dividir el espacio en regiones que representan áreas accesibles desde diferentes puntos de referencia.

El propósito de este trabajo es implementar y evaluar una técnica de planificación de trayectorias basada en diagramas de Voronoi en un entorno tridimensional para robots aéreos. A diferencia de los métodos tradicionales que operan en un plano bidimensional, esta aproximación considera las restricciones adicionales impuestas por el movimiento en tres dimensiones, lo que resulta especialmente relevante en aplicaciones de vehículos no tripulados, como drones o vehículos aéreos no tripulados (UAV).

En este contexto, el proyecto se centrará en los siguientes aspectos:

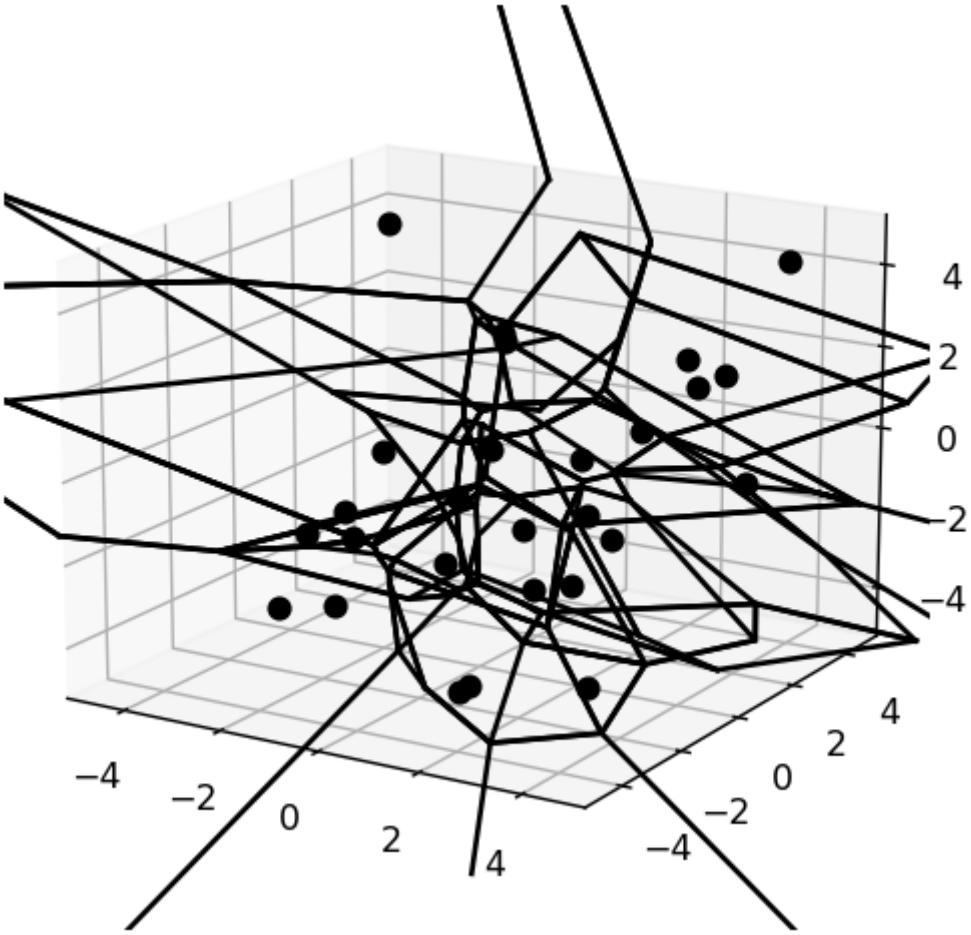
1. **Diagramas de Voronoi:** Se proporcionará una revisión de los principios subyacentes de los diagramas de Voronoi y cómo se pueden aplicar en el contexto de la planificación de trayectorias para robots aéreos.
2. **Desarrollo de Algoritmos:** Se desarrollarán algoritmos específicos para la generación y manipulación de diagramas de Voronoi en tres dimensiones.
3. **Planificadores de trayectoria:** Se verá la implementación de diagramas de Voronoi junto con el uso de distintos algoritmos usados para la creación de trayectorias.
4. **Implementación en un Entorno de Simulación:** Se implementará la técnica de planificación de trayectorias en un entorno de simulación tridimensional, utilizando herramientas como ROS (Robot Operating System) y Gazebo para modelar el comportamiento de los robots y su entorno.
5. **Evaluación y Experimentación:** Se llevarán a cabo experimentos y evaluaciones simuladas para estudiar el comportamiento del robot en distintas condiciones.
6. **Conclusiones**: Se resumirán los hallazgos obtenidos durante el proyecto, identificando posibles áreas de mejora y trabajo futuro para continuar avanzando en el desarrollo de técnicas de planificación de trayectorias para robots aéreos.

Diagramas de Voronoi

Los Diagramas de Voronoi son una herramienta matemática y geométrica que se utiliza en una variedad de aplicaciones, incluyendo la planificación de trayectorias y el control de movimiento de robots. Aquí se detallan los fundamentos teóricos de los Diagramas de Voronoi, junto con subpuntos relacionados con la planificación de trayectorias y los controladores de movimiento:

- **Definición:** Los Diagramas de Voronoi dividen un espacio en regiones, donde cada región está asociada a un punto de referencia y contiene todos los puntos del espacio que están más cerca de ese punto que de cualquier otro.
- **Características:** Estas regiones se conocen como celdas o polígonos de Voronoi, y representan áreas de influencia alrededor de los puntos de referencia.

Métodos de Generación: Los Diagramas de Voronoi pueden generarse utilizando algoritmos que encuentran las líneas equidistantes entre los puntos de referencia, construyendo así las fronteras de las celdas de Voronoi. En la siguiente imagen se adjunta un ejemplo de la generación del diagrama de Voronoi en un plano de 3 dimensiones:



Complejidad computacional: La generación eficiente de Diagramas de Voronoi requiere algoritmos que optimicen el cálculo de las fronteras de las celdas, especialmente en entornos tridimensionales. En este caso se ha implementado en código dándole más peso a los puntos más alejados de obstáculos.

Los Diagramas de Voronoi se utilizan para generar trayectorias eficientes y seguras para robots, ya que proporcionan una representación estructurada del espacio y ayudan a evitar obstáculos. Las fronteras de las celdas de Voronoi pueden servir como guías para planificar rutas que optimicen la distancia recorrida y minimicen el tiempo de navegación.

Los Diagramas de Voronoi pueden actualizarse dinámicamente en respuesta a cambios en el entorno o en las condiciones de vuelo. Esta capacidad de adaptación permite a los robots aéreos tomar decisiones en tiempo real y ajustar su ruta en función de la detección de obstáculos o cambios en la configuración del entorno.

En los siguientes puntos, se verá con más profundidad el uso de los diagramas de Voronoi junto con distintos planificadores de rutas para el uso en su conjunto con el robot.

Desarrollo de Algoritmos

El desarrollo de algoritmos para la generación de Diagramas de Voronoi implica la implementación de métodos eficientes que dividan el espacio en celdas basadas en la proximidad a un conjunto de puntos de referencia. Para generar dichas celdas, será necesario implementar el algoritmo con las coordenadas del sistema donde se va a ejecutar y con los puntos donde se encuentran los obstáculos.

Para garantizar que las trayectorias resultantes sean seguras y eviten colisiones, se ha implementado una función de costos, la cual se encarga de obtener los pesos de los distintos puntos que se encuentran en el mapa bajo estudio. Conforme los puntos se acerquen a un obstáculo, estos tendrán un mayor peso, de manera que la idea es ir recorriendo los puntos con menor peso para poder evitarlos.

A parte de esto, se ha implementado una función heurística, la cual proporciona una estimación del costo entre los puntos origen y destino para elegir la ruta más corta. Se han implementado ambas técnicas para garantizar que el robot siga la ruta más óptima donde no solo pase lo más alejado posible de los obstáculos, sino que también tenga en cuenta los pesos que tienen los puntos de las rutas más cortas. En el siguiente punto se hablará con más detenimiento sobre esta función de costos junto con el planificador de ruta.

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Una vez definidas las funciones de costos a implementar, se han seguido los siguientes pasos para llevar a cabo un desarrollo óptimo de la generación de trayectorias:

1. **Definición de Puntos de Referencia:** Se comienza definiendo un conjunto de puntos de referencia que representan ubicaciones clave en el espacio tridimensional. Estos puntos pueden ser posiciones de interés, como objetivos, áreas a evitar o puntos de partida y llegada para los robots.

2. **Cálculo de las celdas de Voronoi:** Utilizando un algoritmo eficiente, como el Algoritmo de Fortune o el de propagación de frentes, se calculan las celdas de Voronoi para los puntos de referencia. En este caso se ha usado el algoritmo de Fortune importando en python la librería “Voronoi” del paquete **scipy.spatial**

Durante este proceso, se asigna a cada punto en el espacio su celda de Voronoi correspondiente, que contiene todos los puntos más cercanos a ese punto en particular.

3. **Consideraciones de Obstáculos:** Se introduce la función de costos mencionada anteriormente, la cual penaliza la proximidad a los obstáculos en el espacio tridimensional.

4. **Consideración de la ruta más corta:** Se pasan los puntos calculados por la función heurística con la cual se obtienen pesos nuevos para los distintos puntos y se suman con los obtenidos anteriormente.

5. **Generación de Trayectorias Seguras:** Una vez se han sumado ambos costos, se eligen rutas que minimizan el costo total, lo que garantiza que se eviten los obstáculos y se prioricen las áreas seguras para la navegación a la vez que se tiene en cuenta las rutas más cortas.

6. **Implementación en Entornos Prácticos:** El algoritmo se implementa en entornos prácticos utilizando herramientas como ROS, aprovechando sensores y sistemas de percepción para detectar y evitar obstáculos en tiempo real, además, se integra con algoritmos de planificación de trayectorias y sistemas de control de movimiento para permitir la navegación autónoma de robots.

Planificadores de trayectorias mediante algoritmo A*

La planificación de trayectorias es un campo de estudio que se centra en encontrar una ruta óptima para que un robot se mueva desde su posición actual hasta un destino deseado, teniendo en cuenta la presencia de obstáculos y las restricciones del entorno. Uno de los algoritmos más ampliamente utilizados para este propósito es el algoritmo A* (A-star).

A* es un algoritmo de búsqueda heurística que utiliza una combinación de búsqueda exhaustiva y heurísticas para encontrar la ruta más corta entre dos puntos en un grafo ponderado. Su capacidad para encontrar soluciones óptimas en tiempos razonables lo hace muy popular en aplicaciones robóticas.

El proceso de A* comienza con un nodo inicial y utiliza una estructura de datos, comúnmente una cola de prioridad, para mantener los nodos pendientes de evaluación. En cada paso, se selecciona el nodo con el menor valor de la función de evaluación y se expanden sus nodos vecinos. Luego, se calcula el valor de la función de evaluación para cada nodo vecino y se actualizan sus costos acumulados y estimaciones heurísticas. Este proceso se repite hasta que se alcanza el nodo objetivo o se agota la lista de nodos pendientes.

Cabe comentar también que no solo se tiene en cuenta los costos obtenidos por la función heurística, sino que también los obtenidos por el diagrama de Voronoi como se ha dicho con anterioridad.

Una vez realizada la implementación de este planificador, se obtienen la serie de puntos por los que tiene que pasar el robot para ir desde el punto inicial al final.

Controladores de movimiento

Una vez que se ha generado una trayectoria planificada, el siguiente paso es convertirla en comandos de control específicos para el sistema de actuación del robot. Aquí es donde entran en juego los controladores de movimiento. Estos controladores toman la trayectoria deseada y la información del entorno para generar comandos de control que guían al robot a lo largo de la ruta planificada. Uno de los controladores más comunes y efectivos es el controlador de "pure pursuit".

El controlador de "pure pursuit" se basa en el concepto de seguir una ruta mediante la determinación de un punto objetivo a lo largo de la trayectoria deseada. Este punto objetivo se calcula considerando la posición actual del robot y la trayectoria planificada. El controlador ajusta continuamente la velocidad y la dirección del robot para dirigirlo hacia el punto objetivo, lo que resulta en un seguimiento suave y preciso de la ruta planificada. La simplicidad y eficacia del controlador de "pure pursuit" lo hacen muy popular en aplicaciones de robótica móvil.

Comparativa con los Controladores PID:

El controlador PID es otro método de control ampliamente utilizado en robótica. A diferencia del controlador Pure Pursuit, que se centra en seguir una trayectoria predeterminada, el controlador PID ajusta continuamente los comandos de control para minimizar el error entre la salida del sistema y un valor de referencia.

Los controladores PID son más generales y pueden adaptarse a una variedad de problemas de control, pero pueden requerir ajustes finos de los parámetros del controlador para obtener un rendimiento óptimo. A parte de esto, las ventajas que puede ofrecer un controlador Pure Pursuit en comparativa con un PID son:

1. El controlador Pure Pursuit es más simple de entender e implementar en comparación con un controlador PID. No requiere ajustes finos de los parámetros del controlador ni conocimientos profundos de teoría de control para funcionar efectivamente. Esto lo hace más accesible para personas sin experiencia en control de sistemas.
2. El controlador Pure Pursuit está diseñado específicamente para el seguimiento de trayectorias. Es altamente eficaz en entornos donde se dispone de una trayectoria de referencia predeterminada y el objetivo principal es seguir esta trayectoria con precisión. En comparación, los controladores PID pueden ser menos especializados y no están optimizados específicamente para el seguimiento de trayectorias.
3. El controlador Pure Pursuit puede ser menos sensible a perturbaciones y variaciones en el sistema en comparación con los controladores PID. Debido a su diseño específico para el seguimiento de trayectorias, el Pure Pursuit puede ser más robusto frente a cambios repentinos en las condiciones del entorno o en el comportamiento del sistema.

4. El controlador Pure Pursuit puede requerir menos ajuste de parámetros en comparación con los controladores PID. Una vez que se ha definido la trayectoria de referencia, el controlador Pure Pursuit puede funcionar efectivamente con pocos o ningún ajuste adicional de parámetros. En cambio, los controladores PID suelen requerir un ajuste cuidadoso de los parámetros para lograr un rendimiento óptimo, lo que puede ser más complejo y consumir más tiempo.
5. En aplicaciones donde el seguimiento de trayectorias es la tarea principal, como en la navegación de robots móviles o vehículos autónomos, el controlador Pure Pursuit puede ofrecer un rendimiento superior y más consistente en comparación con un controlador PID, que puede ser más generalista y menos especializado en esta tarea específica.

Por este motivo, aunque el controlador PID es una opción atractiva por su simplicidad y fácil implementación, se ha decidido usar el Pure Pursuit ya que en este tipo de controles las ventajas que ofrece son mayores.

ROS

ROS (Robot Operating System) es un conjunto de herramientas, bibliotecas y convenciones que tienen como objetivo simplificar el desarrollo de software para robots.

Características Principales de ROS:

- **Arquitectura Distribuida:** ROS está diseñado siguiendo una arquitectura distribuida, lo que significa que los distintos componentes de un sistema robótico pueden ejecutarse en diferentes nodos de manera independiente y comunicarse entre sí a través de un sistema de mensajería.
- **Sistema de Mensajería:** ROS proporciona un sistema de mensajería que permite a los nodos de un sistema robótico intercambiar datos de manera eficiente. Este sistema facilita la comunicación entre los diferentes componentes del robot, como sensores, actuadores, planificadores y controladores.
- **Gestión de Paquetes:** ROS utiliza un sistema de gestión de paquetes que facilita la instalación, distribución y gestión de software. Los desarrolladores pueden crear paquetes de software ROS que contienen módulos, bibliotecas y herramientas relacionadas con un aspecto específico de la funcionalidad del robot.
- **Herramientas de Desarrollo:** ROS ofrece una variedad de herramientas de desarrollo, incluyendo herramientas de visualización, simulación y depuración, que ayudan a los desarrolladores a diseñar, probar y depurar sus sistemas robóticos de manera más eficiente.
- **Compatibilidad con Diversos Hardware:** ROS es compatible con una amplia gama de hardware, lo que permite su uso en una variedad de plataformas robóticas, desde robots terrestres y aéreos hasta robots submarinos y brazos robóticos.

Beneficios de Utilizar ROS:

- **Reutilización de Código:** ROS fomenta la reutilización de código al proporcionar una amplia variedad de paquetes y bibliotecas de código abierto que pueden ser utilizados y adaptados para diferentes aplicaciones robóticas.

- **Desarrollo Colaborativo:** Al ser de código abierto y contar con una comunidad activa de desarrolladores, ROS facilita el desarrollo colaborativo de software para robots, permitiendo compartir conocimientos, herramientas y recursos entre la comunidad.
- **Facilidad de Integración:** Gracias a su arquitectura modular y su sistema de mensajería, ROS facilita la integración de diferentes componentes de hardware y software en un sistema robótico, lo que permite construir sistemas complejos y heterogéneos de manera más sencilla.

Hay diferentes versiones de ROS como kinetic, melodic o noetic. En un principio se iba a usar ROS melodic, pero tras encontrar errores en las dependencias de los paquetes que se querían usar, se decidió finalmente optar por ROS noetic.

La funcionalidad implementada para comunicarse entre los nodos desarrollados es la siguiente:

1. El primer nodo que se creó es el encargado de enviar los puntos obtenidos por el planificador de trayectorias al controlador del robot.
2. El segundo nodo, recibe esta información y pasa la posición actual del robot al primer nodo, de tal manera que se compara el punto actual con una pequeña incertidumbre para verificar que el robot ha llegado al punto deseado. Si es así, el primer nodo le enviará el segundo punto a ir al segundo nodo y así sucesivamente hasta llegar a la meta.

Implementación

Para la implementación en ROS de todo lo comentado en los puntos anteriores nos hemos decantado por el lenguaje de programación Python ya que es ampliamente elogiado por su elegante sintaxis, versatilidad y una comunidad activa. Aquí se enumeran algunas características por la cual destaca:

- Sintaxis Clara y Concisa: Python tiene una sintaxis fácil de leer y escribir, lo que hace que el código sea claro y comprensible. Esto facilita a los programadores escribir y mantener código.
- Versatilidad: Python es un lenguaje de propósito general, lo que significa que se puede utilizar para una amplia variedad de aplicaciones, desde desarrollo web hasta inteligencia artificial.
- Gran Comunidad y Soporte: Python tiene una comunidad activa y vibrante. Esto significa que hay una gran cantidad de recursos en línea, tutoriales, bibliotecas y frameworks que pueden facilitar el desarrollo.
- Biblioteca Estándar Rica: Python viene con una amplia biblioteca estándar que abarca desde manipulación de archivos hasta protocolos de red, lo que facilita la tarea de los desarrolladores al proporcionar funciones y módulos listos para usar.
- Facilidad de Aprendizaje: Es considerado como un buen lenguaje para principiantes debido a su sintaxis sencilla y su enfoque en la legibilidad del código. Esto ayuda a nuevos programadores a aprender rápidamente y a escribir código eficiente.
- Compatibilidad con Integración de Lenguajes: Puedes incorporar fácilmente código escrito en otros lenguajes como C y C++ en tu programa Python, lo que permite la integración de módulos de alto rendimiento cuando es necesario.
- Desarrollo Rápido de Prototipos: Python es conocido por su capacidad para facilitar el desarrollo rápido de prototipos, lo que lo convierte en una elección popular en entornos donde la agilidad y la iteración rápida son fundamentales.
- Soporte para Programación Orientada a Objetos y Funcional: Python admite múltiples paradigmas de programación, lo que brinda a los desarrolladores la flexibilidad de elegir el enfoque que mejor se adapte a sus necesidades.

Para la realización de este proyecto se han creado y desarrollado varios scripts en los que vamos a enumerar y explicar uno por uno.

- **point_publisher.py:**

  Este código está diseñado para realizar planificación de trayectorias en un entorno 3D mediante el uso del algoritmo A* (A estrella) y el diagrama de Voronoi.

En el que el se utiliza para diagrama de Voronoi para guiar la planificación de trayectorias en un entorno 3D, evitando obstáculos definidos en una matriz y optimizando el camino entre la posición inicial y la objetivo. La trayectoria calculada se publica en ROS para su uso en sistemas de control de robots móviles.

1. Definición de tipos y funciones auxiliares:

    - Se definen algunos tipos (`T`, `Location`, `GridLocation`) y funciones auxiliares para cálculos trigonométricos y conversiones de ángulos.

2. Generación de la cuadrícula y cálculo del diagrama de Voronoi:

    - La función `generate_grid` crea una cuadrícula 3D de puntos.

    - La función `calculate_voronoi` utiliza la biblioteca `scipy.spatial.Voronoi` para calcular el diagrama de Voronoi en 3D. Combina los obstáculos con la cuadrícula y devuelve el objeto `Voronoi`.

3. Clases `PriorityQueue` y `GridWithWeights`:

    - `PriorityQueue` es una cola de prioridad para el algoritmo A*.

    - `GridWithWeights` define un grafo 3D con obstáculos y funciones relacionadas.

4. Funciones de dibujo y visualización:

    - `draw_tile` y `draw_grid` son funciones auxiliares para visualizar la cuadrícula y los resultados del algoritmo.

5. Algoritmo A* y funciones relacionadas:

    - `heuristic` calcula la heurística entre dos ubicaciones.

    - `a_star_search` realiza la búsqueda A* en el grafo 3D, utilizando la cuadrícula generada y el diagrama de Voronoi para guiar la planificación de trayectorias.

    - `reconstruct_path` reconstruye el camino óptimo a partir de la información obtenida por el algoritmo A*.

6. Función `get_path`:

    - Utiliza la información del mapa de obstáculos y las posiciones inicial y objetivo para ejecutar el algoritmo A* y obtener la trayectoria óptima.

7. Clase `pointpublisher`:

    - Esta clase se encarga de publicar los puntos de la trayectoria en ROS.

8. Función `run` de `pointpublisher`:

    - En un bucle, obtiene la trayectoria llamando a `get_path` y la publica en el tópico `/point_topic` en ROS.

9. Inicialización y ejecución:

    - Se crea una instancia de `pointpublisher` y se ejecuta el método `run` para iniciar la publicación de puntos de trayectoria.

- **voronoi_path.py:**

Este código en Python es un nodo de ROS que suscribe a dos tópicos, "/pose" y "/point_topic", y publica en el tópico "/voronoi_pub" basándose en la información recibida. Aquí está un resumen esquemático del código:

1. Inicialización del Nodo y Suscripciones:

    - Se inicializa un nodo de ROS llamado "voronoi_pub" y se establece la frecuencia de publicación. Luego, se crea un objeto `voronoi_path`.

    - Se suscribe al tópico "/pose" para obtener la posición actual del robot.

    - Se suscribe al tópico "/point_topic" para obtener puntos de una trayectoria de Voronoi.

2. Publicación en el Tópico "/voronoi_pub":

    - La clase `voronoi_path` tiene un método `voronoi_publisher` que publica mensajes en el tópico "/voronoi_pub" utilizando el objeto `rospy.Publisher`. Cada mensaje contiene las coordenadas x, y, y theta.

3. Método de Verificación:

    - El método `check` verifica si la posición actual del robot está dentro de una tolerancia con respecto al punto de la trayectoria de Voronoi. Si es así, se considera que ha alcanzado ese punto y pasa al siguiente.

4. Método Callback para "/pose":

- El método “pose_cb” se activa cada vez que se recibe un mensaje en el tópico "/pose". Extrae las coordenadas de posición del mensaje y las almacena en la variable `self.pose`.

5. Método Callback para "/point_topic":

- El método “pose_voro” se activa cada vez que se recibe un mensaje en el tópico "/point_topic". Añade las coordenadas del punto a la variable “self.voronoi_target”.

6. Bucle Principal:

- En el método `run`, hay un bucle principal que publica puntos de la trayectoria de Voronoi en el tópico "/voronoi_pub". El bucle se ejecuta hasta que el nodo es apagado.

7. Ejecución del Programa:

- Se instancia un objeto `voronoi_path` y se ejecuta su método `run()`.

En resumen, este código crea un nodo ROS que, basándose en la posición del robot y los puntos de una trayectoria de Voronoi, publica información en un nuevo tópico "/voronoi_pub". Este nodo está diseñado para moverse a lo largo de la trayectoria de Voronoi recibida.

- **pose_reading.py:**

Este código en Python que se suscribe a tres tópicos ("/pose", "/voronoi_pub", "/point_topic") y publica comandos de velocidad en el tópico "/cmd_vel".

1. Inicialización del Nodo y Suscripciones:

- Se inicializa un nodo de ROS llamado "pose_sub".

- Se suscribe al tópico "/pose" para obtener la posición y orientación actual del robot.

- Se suscribe al tópico "/voronoi_pub" para obtener los puntos de la trayectoria de Voronoi.

- Se suscribe al tópico "/point_topic" para recibir mensajes de posición.

2. Publicación de Comandos de Velocidad:

   - La clase `pose_sub` tiene un método `publish_velocity` que publica comandos de velocidad en el tópico "/cmd_vel" usando el objeto `rospy.Publisher`.

3. Métodos de Callback:

   - `pose_cb`: Se activa cuando se recibe un mensaje en el tópico "/pose". Extrae las coordenadas de posición y orientación del mensaje y las almacena en las variables `self.pose` y `self.pose_orientation`.

   - `voronoi_cb`: Se activa cuando se recibe un mensaje en el tópico "/voronoi_pub". Extrae las coordenadas de posición y orientación del mensaje y las almacena en la variable `self.voronoi_target`.

   - `pose_voro`: Se activa cuando se recibe un mensaje en el tópico "/point_topic". Modifica la variable `self.test` para indicar que se ha recibido un mensaje.

4. Bucle Principal:

   - En el método `run`, hay un bucle principal que calcula los comandos de velocidad utilizando una función llamada `pure_pursuit` del módulo `purepursuit_controller`, y publica estos comandos en el tópico "/cmd_vel". El bucle se ejecuta hasta que el nodo es apagado.

5. Ejecución del Programa:

   - Se instancia un objeto `pose_sub` y se ejecuta su método `run()`.

En resumen, este código implementa un nodo de ROS que utiliza información de posición y trayectoria. La lógica de control y cálculos específicos se encuentran en un módulo externo llamado `purepursuit_controller`.

- **purepursuit_controller.py:**

  Este código está implementado el controlador de Pure Pursuit.

  1. Función `yaw_angle(pose_orientation)`:

- Calcula y devuelve el ángulo de orientación en el eje Z (yaw) a partir de la orientación del robot en cuaterniones.

2. Función pitch_angle(pose_orientation):

- Calcula y devuelve el ángulo de inclinación en el eje Y (pitch) a partir de la orientación del robot en cuaterniones.

3. Función pure_pursuit(voronoi_target, pose, pose_orientation):

- Implementa el algoritmo de control de "pure pursuit" para un robot móvil.

- Calcula la distancia y el ángulo entre la posición actual del robot y el punto objetivo en una trayectoria de Voronoi.

- Determina la velocidad lineal, velocidad angular y ajustes de orientación necesarios para seguir la trayectoria.

- Utiliza parámetros como la longitud del robot (`L`) y una constante de ajuste (`K`) para controlar la velocidad angular.

- Devuelve las velocidades lineal y angular, así como los ajustes de orientación.

Simulación con Gazebo y Rviz

Para la simulación del funcionamiento del robot, se ha usado la herramienta Gazebo y Rviz. Gazebo es un entorno de simulación tridimensional (3D) avanzado y de código abierto, alguna de sus características principales son:

- **Entorno 3D realista:** Gazebo proporciona un entorno de simulación 3D altamente realista donde se pueden modelar entornos complejos, como interiores de edificios, paisajes al aire libre, carreteras, entre otros.
- **Modelado de robots y vehículos:** Permite crear modelos precisos de robots, vehículos, drones y otros objetos móviles, incluyendo sus componentes físicos, sensores y actuadores. En este caso, se ha decidido hacer un mapa simple con la incorporación del robot y unos edificios para la selección de la ruta.
- **Física precisa:** Gazebo simula la física del mundo real, lo que permite modelar con precisión el comportamiento de los objetos en el entorno virtual, incluyendo la dinámica de movimiento, colisiones, fricción, gravedad, etc.
- **Interfaz de usuario intuitiva:** Ofrece una interfaz de usuario intuitiva y fácil de usar que permite configurar y controlar la simulación, así como visualizar datos y resultados en tiempo real.
- **API extensible:** Gazebo proporciona una API extensible que permite a los desarrolladores extender y personalizar la funcionalidad del simulador, así como integrarlo con otros sistemas y herramientas de software.

Rviz es una herramienta de visualización 3D que forma parte del ecosistema de ROS. Sus principales características son:

- **Visualización de Datos en 3D:** Rviz permite visualizar datos tridimensionales relacionados con el entorno del robot, como sensores, modelos 3D del robot, trayectorias planificadas y objetos en el entorno.
- **Interfaz Gráfica Intuitiva:** La interfaz gráfica de Rviz es fácil de usar, lo que facilita la configuración y la visualización de información relevante sobre el estado del robot.

- **Soporte para Múltiples Tipos de Datos:** Puede visualizar una amplia variedad de datos, incluyendo nubes de puntos, modelos URDF (Unified Robot Description Format), mapas 2D y 3D, trayectorias, datos de sensores como cámaras y láser, entre otros.
- **Configuración Dinámica:** Permite la configuración dinámica de la visualización, lo que significa que puedes ajustar parámetros y configuraciones en tiempo real para adaptarse a tus necesidades de visualización.
- **Colaboración con ROS:** Está integrado con ROS, permitiendo la fácil conexión con otros nodos y herramientas ROS. Puedes suscribirte a tópicos ROS y visualizar datos en tiempo real.
- **Herramienta de Depuración:** Rviz es una herramienta valiosa para la depuración y el análisis del comportamiento del robot. Puedes verificar la información de los sensores y la planificación de trayectorias para asegurarte de que el robot esté operando correctamente.
- **Soporte para Navegación y Planificación:** Es frecuentemente utilizado en aplicaciones de navegación y planificación de robots, ya que permite visualizar mapas, trayectorias planificadas y el estado de los sensores en tiempo real.
- **Extensibilidad:** Rviz es extensible y permite la integración de plugins adicionales para ampliar sus capacidades según las necesidades específicas del usuario.
- **Herramienta de Desarrollo y Pruebas:** Facilita el desarrollo y las pruebas de algoritmos de percepción, control y planificación en un entorno virtual antes de implementarlos en el robot físico.
- **Independiente de Plataforma:** Puede ejecutarse en sistemas operativos Linux, macOS y Windows, lo que facilita su uso en una variedad de entornos de desarrollo.

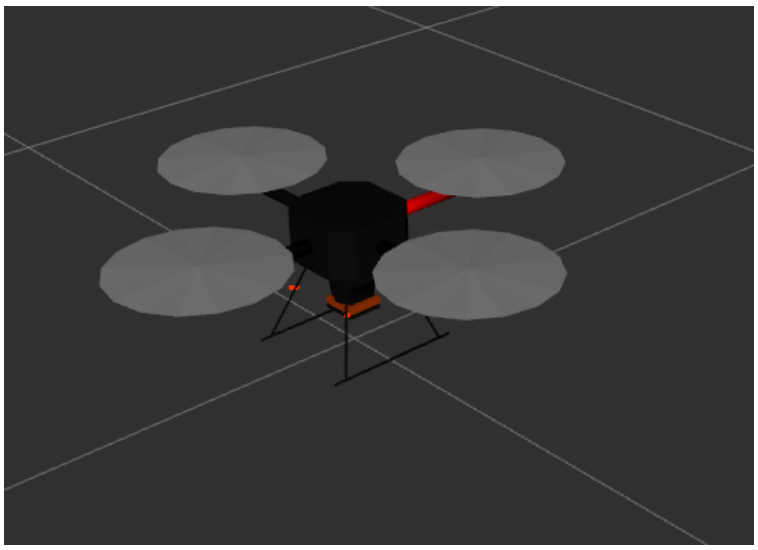
En resumen, Rviz es una herramienta esencial en el kit de herramientas de ROS, proporcionando una interfaz gráfica para visualizar y entender el comportamiento del robot en entornos 3D, lo que es crucial para el desarrollo y la depuración efectiva de aplicaciones robóticas.

Ya explicado nuestro entorno de simulación vamos a definir el modelo de vehículo aéreo que se ha utilizado que es el hector_quadrotor.

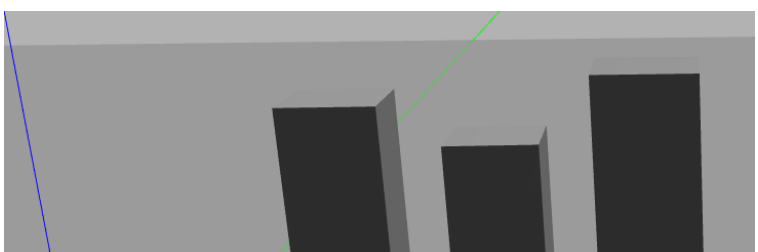
El hector_quadrotor es un modelo de robot cuadricóptero utilizado en el contexto de la robótica y la investigación en vehículos aéreos no tripulados (UAV, por sus siglas en inglés). Este modelo específico, a menudo referido como **Hector Quadrotor**, se ha destacado por su uso en el marco del **ROS** (Robot Operating System), que es un sistema operativo de código abierto utilizado para el desarrollo de software en robótica.

- **Definición:** el hector_quadrotor es un modelo específico de cuadricóptero utilizado como plataforma de simulación y desarrollo en el entorno de ROS.
- **Estructura:** es un cuadricóptero, lo que significa que tiene cuatro rotores dispuestos en una configuración cuadrada. La estructura básica consta de un chasis central y cuatro brazos, cada uno equipado con un rotor.

- **Propósitos:** se utiliza principalmente con fines de investigación, desarrollo y pruebas en el ámbito de la robótica aérea y los UAV. Proporciona una plataforma simulada para experimentación y desarrollo de algoritmos de control, planificación de trayectorias, percepción y otras aplicaciones relacionadas con UAV.
- **Simulación en ROS:** el modelo hector_quadrotor es compatible con ROS, lo que permite su integración fácil en el ecosistema ROS. Esto facilita la simulación, el control y la interacción con otros nodos y paquetes de software en el entorno de ROS.
- **Características Adicionales:** además de la simulación básica de movimiento y control, el modelo puede incluir características adicionales, como sensores simulados (cámaras, IMU, etc.), capacidad para planificación de trayectorias y simulación de entornos variados.
- **Herramienta de Desarrollo:** sirve como una herramienta valiosa para el desarrollo y prueba de algoritmos y aplicaciones relacionadas con vehículos aéreos no tripulados. Los investigadores y desarrolladores pueden utilizar el hector_quadrotor como una plataforma simulada antes de implementar y probar en un cuadricóptero físico.
- **Modularidad y Flexibilidad:** la modularidad y flexibilidad del modelo permiten a los usuarios adaptarlo a sus necesidades específicas de investigación y desarrollo. Pueden agregar sensores, modificar parámetros de simulación y ajustar la dinámica del cuadricóptero según los requisitos del proyecto.
- **Contribuciones a la Comunidad:** el modelo hector_quadrotor ha contribuido significativamente a la comunidad de desarrollo de ROS, proporcionando una plataforma común para la experimentación y la colaboración en el ámbito de la robótica aérea.

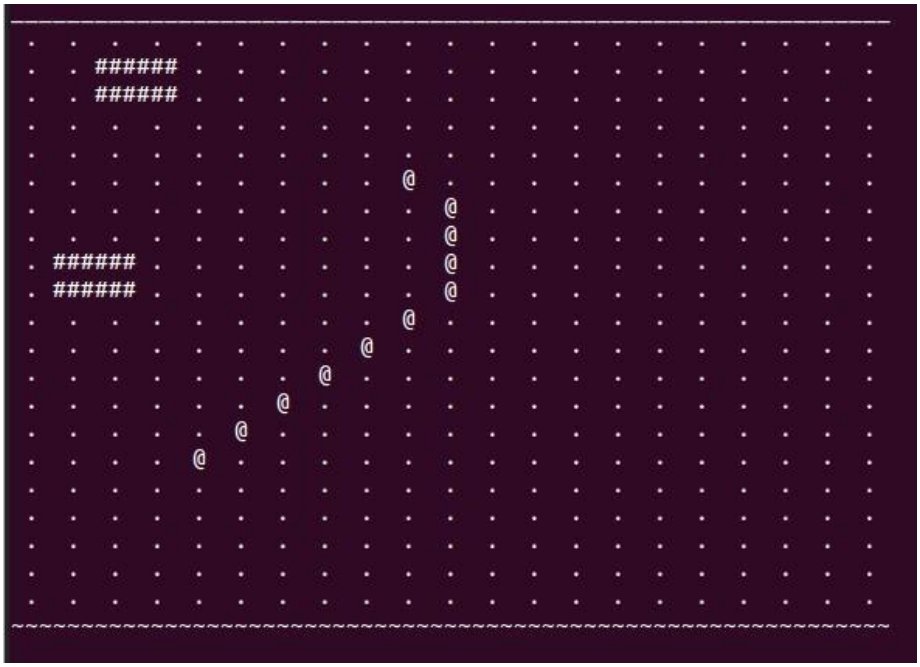


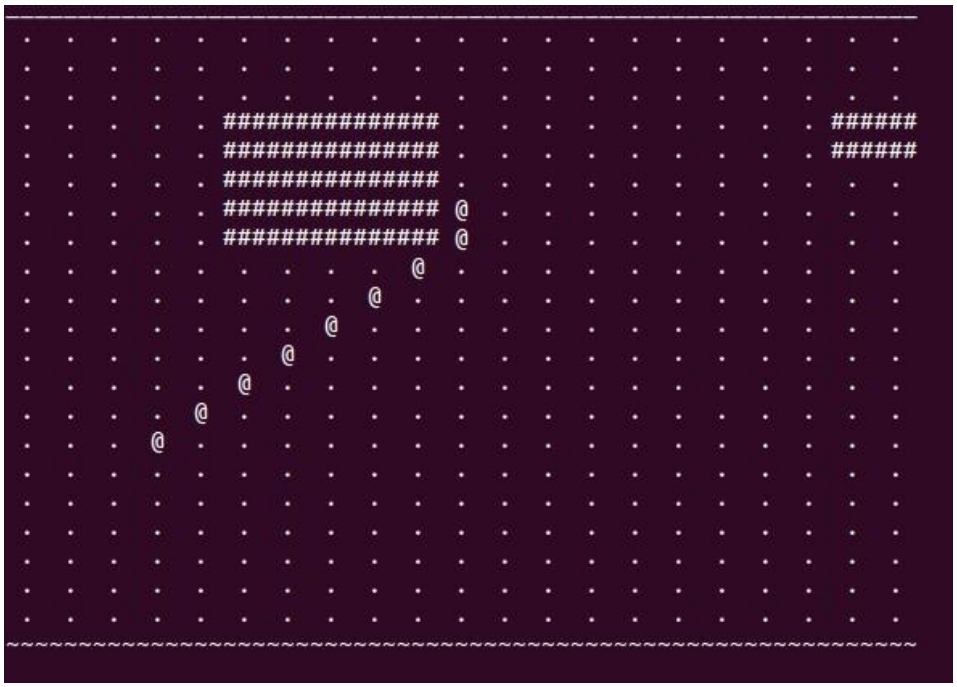
Ya, por último, se ha diseñado un mundo para la simulación del nuestro programa. Como podemos ver en la siguiente imágen:

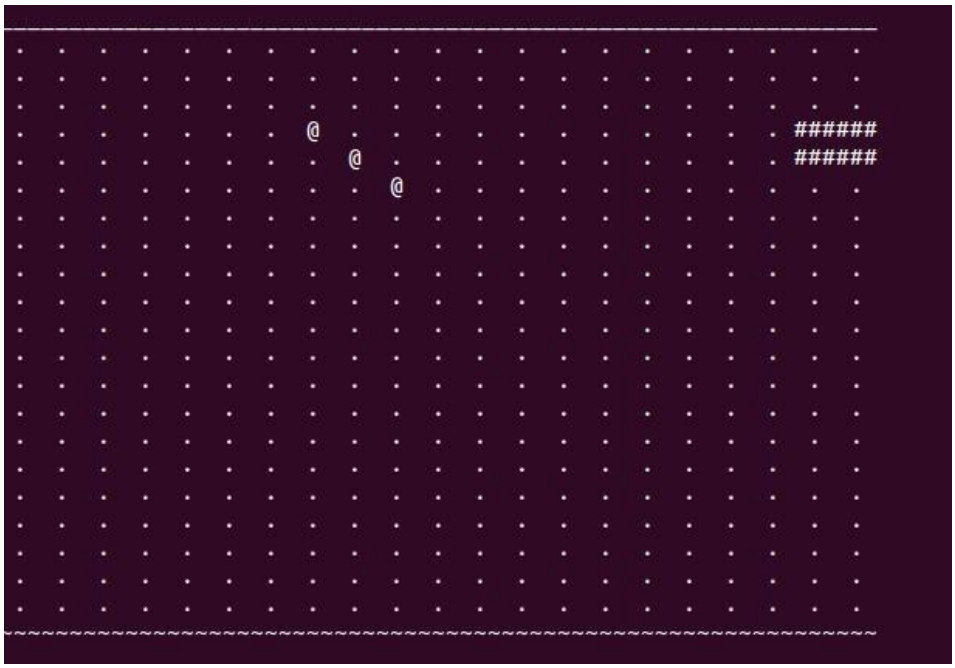


Resultados

En esta sección, procederemos a analizar los resultados derivados de la implementación realizada. Las siguientes imágenes exhiben el resultado bidimensional (2D) de la trayectoria seguida por nuestro vehículo aéreo no tripulado (dron) en distintos mapas en la simulación.







La observación de las trayectorias revela un desempeño notable al evitar obstáculos y seguir una ruta que integra efectivamente el diagrama de Voronoi, implementado en conjunto con una heurística que persigue la búsqueda del camino más corto. La eficacia de este enfoque se manifiesta en la capacidad del vehículo aéreo no tripulado (dron) para adaptarse y sortear obstáculos de manera efectiva, evidenciando una planificación de ruta coherente y eficiente.

El análisis detallado de las trayectorias trazadas resalta la calidad de la planificación del movimiento, donde la combinación de la información del diagrama de Voronoi y la heurística demuestra una capacidad excepcional para generar rutas que minimizan la longitud total del recorrido. Este enfoque integrado no solo cumple con el objetivo de evadir obstáculos de manera efectiva sino que también optimiza la eficiencia de la trayectoria, lo que se traduce en una navegación más precisa y económica en términos de distancia recorrida.

La sinergia entre la representación del espacio mediante el diagrama de Voronoi y la heurística aplicada refleja una estrategia planificada que maximiza la efectividad del dron en entornos complejos. Este análisis subraya la solidez del sistema implementado al gestionar situaciones desafiantes y proporciona un fundamento sólido para la confianza en la capacidad de planificación y ejecución del vehículo aéreo no tripulado en diversas condiciones y entornos operativos.

Conclusiones

En este trabajo se han aprendido conceptos generales sobre los planificadores de trayectoria, además de la amplia posibilidad de los mismos que se tiene adecuando el peso que se tiene en el mapa de costes.

También se ha aprendido a usar distintas herramientas tanto para la simulación de un planificador de trayectoria como para el control de un robot. Se destaca la dificultad añadida de que, tratándose de un robot aéreo, hay que adaptar la forma de abordar el problema a tres dimensiones.

Por otra parte, existen distintas mejoras al proyecto realizado, como podrían ser un control más preciso, ajustar mejor el mapa de costes del diagrama de Voronoi y el estudio del comportamiento con mayor variedad de condiciones ambientales.