Proyecto de curso
Control y Programación de Robots

Andrés Rosa Flores
Pedro López Japón
Norberto Cabello Gimeno
Alejandro Castillo Puerto

4º GIERM
Curso 2022/23

Índice

Introducción

En esta memoria de proyecto se presenta el resultado de nuestro trabajo en el ámbito de la simulación y control de un robot ROSbot 2.0 mediante el método de persecución pura (pure pursuit).

El objetivo principal de nuestro proyecto ha sido diseñar, construir y simular un sistema de control que permitiese al modelo del ROSbot seguir una trayectoria precalculada de manera precisa y eficiente alcanzando una meta predefinida. Para lograrlo, se hace uso de un algoritmo de Dijkstra para generar la trayectoria global del sistema y el método de “pure pursuit” para seguir la ruta realizada a través del seguimiento de puntos objetivos.

A lo largo de esta memoria se explicarán de forma simple y orientativa cuáles han sido los paquetes utilizados, los nodos, el software de programación y simulación utilizados, así como los problemas a la hora de la simulación, ajuste del control propuesto, errores de seguimiento obtenidos en los experimentos, etc.

Software utilizado

Durante el desarrollo del proyecto se ha utilizado una combinación de herramientas y entornos de programación que han sido fundamentales para el logro de nuestros objetivos.

El primero de ellos es ROS Melodic. Sus librerías y herramientas nos facilitan el desarrollo de aplicaciones robóticas. Con la ayuda de ROS, hemos creado y coordinado los diferentes nodos del sistema, permitiendo la comunicación y el intercambio de información entre ellos.

El desarrollo de nodos y algoritmos se codifica a través de Microsoft Visual Studio. Con este IDE hemos podido escribir, depurar y administrar el código C++ de forma eficiente.

Para la simulación del entorno en el cual el robot se desplaza, se ha recurrido al entorno Gazebo, así como la herramienta ‘Building Editor’ de este para crear los mapas físicos que se usarán para los distintos experimentos. Este ha permitido colocar el modelo del robot en el mapa, así como calcar las paredes en 3D tal y como se establecen en las dimensiones del mapa de costes.

Este mapa de costes y la trayectoria calculada se visualizan en RViz, un programa incluido en la suite de ROS. En él se muestra en tiempo real el seguimiento del robot y los datos más importantes que no se ven en el entorno de simulación de Gazebo.

Por último, se acude a MATLAB para representar los resultados de una forma visual clara. Datos como la odometría, las trayectorias o los errores son los que se representan durante toda esta memoria.

Modelo del robot

Lo primero que se hizo fue instalar el modelo del robot. En nuestro caso era fácil porque el robot que se nos solicita es el ROSbot 2.0 de Husarion, el mismo que se ha usado en las prácticas de la asignatura. Tras clonar el repositorio "*rosbot_description*" de GitHub en la primera práctica, era inmediato hacer una copia para nuestro espacio de trabajo y usar lo que nos interese del modelo.

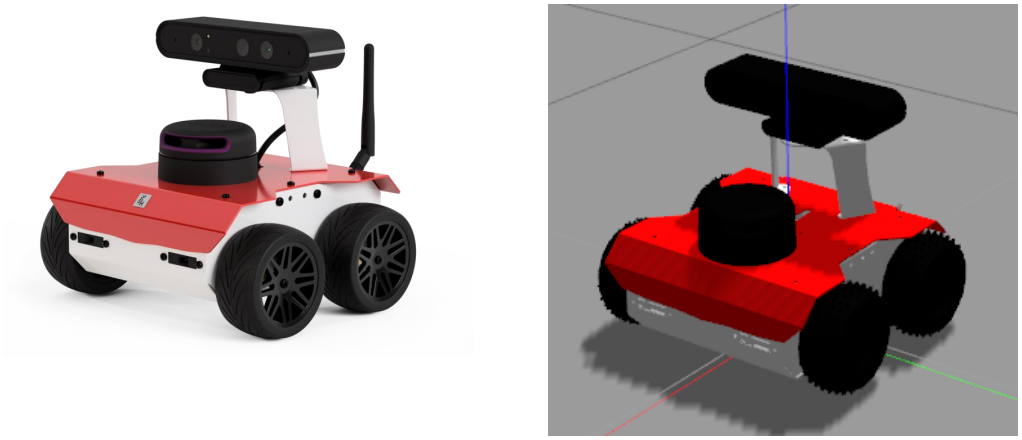


Imagen 1. Modelo del robot

Este modelo contiene principalmente una carpeta de recursos llamada src acompañada de unos archivos Shell que sirven para lanzar las distintas aplicaciones usadas y matar los procesos en ejecución para asegurarse de no pisar software previamente abierto.

Dentro de la carpeta src se localizan tres carpetas llamadas *rosbot_description*, *rosbot_gazebo* y *rosbot_navigation,* que serán esenciales para la correcta puesta a punto de nuestro controlador local.

El paquete *rosbot_description* contiene varias carpetas claves para la descripción y simulación del robot. La carpeta URDF contiene los archivos homónimos que definen la estructura física del Robot como las partes del cuerpo, los joints, los sensores y los actuadores. La carpeta meshes contiene las mallas tridimensionales utilizadas para representar visualmente el robot y sus componentes como las ruedas, el chasis o los sensores. Sin esta carpeta y la anterior no se podría ver el robot físico en el entorno de simulación. La carpeta RViz contiene configuraciones básicas que definen la forma en la que se representan los datos en RViz y la carpeta de scripts contiene archivos de Python relacionados con la comunicación. Finalmente, dentro de la carpeta launch, tenemos los archivos de lanzamiento de distintas herramientas útiles de RViz, así como este programa en sí y de Gazebo.

En el directorio *rosbot_gazebo* nos dividimos en las carpetas launch y worlds. La carpeta worlds contiene los mundos creados por nosotros. Se han creado 3 mundos en Gazebo basándonos en los mapas de costes programados para RViz. Estos mundos se explicarán más adelante, pero adelantamos que tienen distintas

formas, pasillos y dimensiones para exprimir el potencial del controlador y estudiar sus límites. Por otro lado, la carpeta launch, lanzará estos mundos en Gazebo.

Finalmente, se incluye la carpeta *rosbot_navigation*, la cual se ha acabado inutilizando porque todo lo relativo a la navegación se encuentra en la carpeta del pure-pursuit.

Si ejecutamos la orden rqt_graph en la consola de comandos justo después de lanzar el *rosbot_description* como base del proyecto, se obtendrá un esquema como el mostrado a continuación.

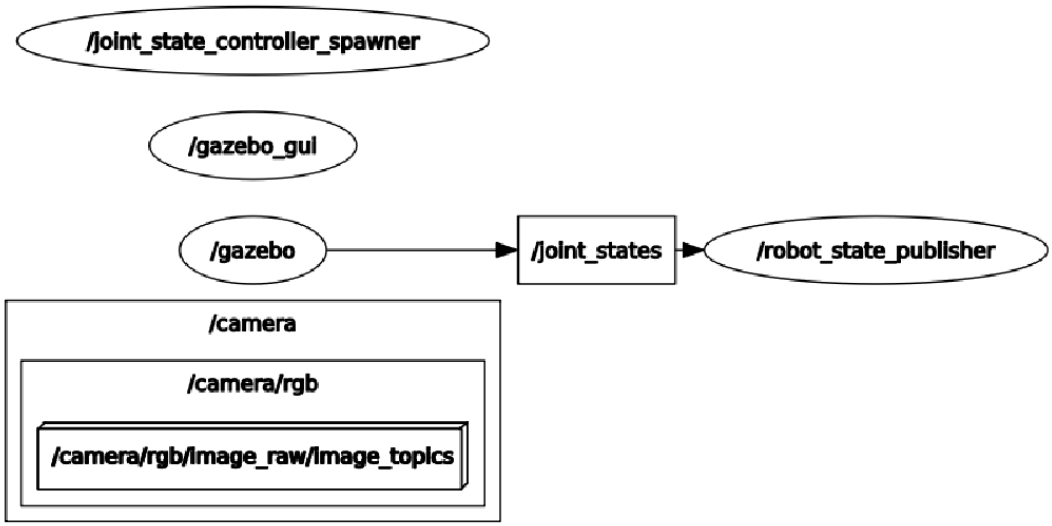


Imagen 2. Conexiones básicas del Rosbot 2.0

Costmap_node y mundos de gazebo

Este nodo de ROS se trata de un mapa de costes a partir del cual se establece el valor de cada celda en los distintos mapas creados.

Lo primero es crear un objeto publicador con el que se manda esta información anterior al topic */map*, el cual tiene un tamaño de cola de 10. Los mensajes que se envían son del tipo “nav_msgs/OccupancyGrid”. A continuación, se crea un objeto "nav_msgs::OccupancyGrid" llamado "costmap" que representa un mapa de ocupación. Por último, en el encabezado del nodo, tenemos un conjunto de parámetros característicos para cada mapa. Algunos de ellos son la resolución, el origen de coordenadas o las dimensiones del mapa.

Posteriormente pasamos a la lectura de los distintos mapas para saber qué valor de ocupación tiene cada celda. Esto se hará dentro de un bucle en el que se definen los obstáculos del mapa, para verificar con posterioridad si la celda actual está dentro de alguno de los obstáculos, asignando así un valor a la celda correspondiente en "costmap.data". Si la celda está dentro de un obstáculo, se asigna un valor de 50, y de lo contrario, se asigna un valor de 0.

Este proceso se realizará para cada uno de los mapas que describiremos más adelante.

Por último, se publica en el topic */map* el coste de cada celda para que pueda ser leído por el planificador global y así crear una ruta predefinida.

Una vez explicado este nodo, llamado *“costmap_node.cpp”,* vamos conocer los tres mapas que hemos creado para así poder hacer distintas pruebas y ver cómo funciona el algoritmo en cada caso.

El primer mapa se trata de una base de 100m² en el que se añaden cuatro vigas cuadradas en los vértices de un cuadrado imaginario de 50m². En este caso hemos hecho que nuestro ROSbot circule de un extremo a otro esquivando estos obstáculos.

El segundo mapa es un laberinto con paredes estrechas y espaciadas en el que el ROSbot tiene suficiente espacio para girar.

Por último, el tercer mapa se ha hecho con caminos estrechos y curvas pronunciadas para poder comprobar los casos límites del algoritmo.

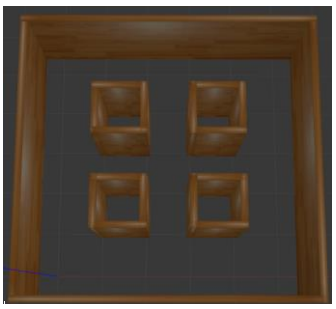


Imagen 3. Mundo 1

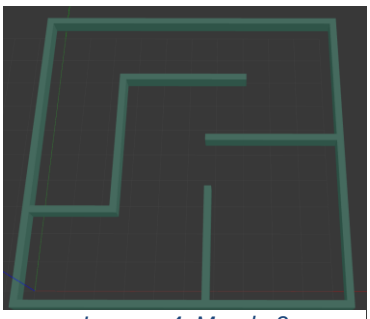


Imagen 4. Mundo 2

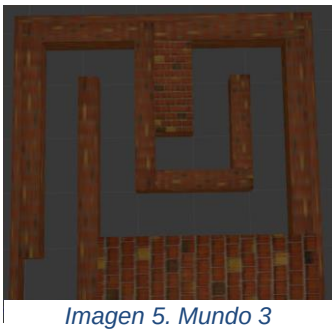


Imagen 5. Mundo 3

Dijkstra

Este nodo es el encargado de calcular la trayectoria global que debe seguir el robot, implementando un algoritmo tipo Dijkstra o de caminos mínimos. Este algoritmo calcula, para un grafo ponderado, el camino más corto entre un punto inicio y un punto objetivo.

A grandes rasgos, este comienza dado una distancia infinita a todos los puntos, que llamaremos nodos, a excepción del origen, Además añade todos los nodos a una lista de “no visitados”. A continuación, se selecciona el origen como nodo actual y se comprueba si existe una ruta más corta a cada uno de ellos, comparándola con la distancia acumulada del nodo actual y actualizando la tabla de distancias del grafo. Repitiendo este proceso se marcan todos los nodos como “visitados” y se tiene una tabla con la distancia más corta entre ellos. Una vez explorado todo el grafo se puede reconstruir la mejor ruta desde el origen al destino.

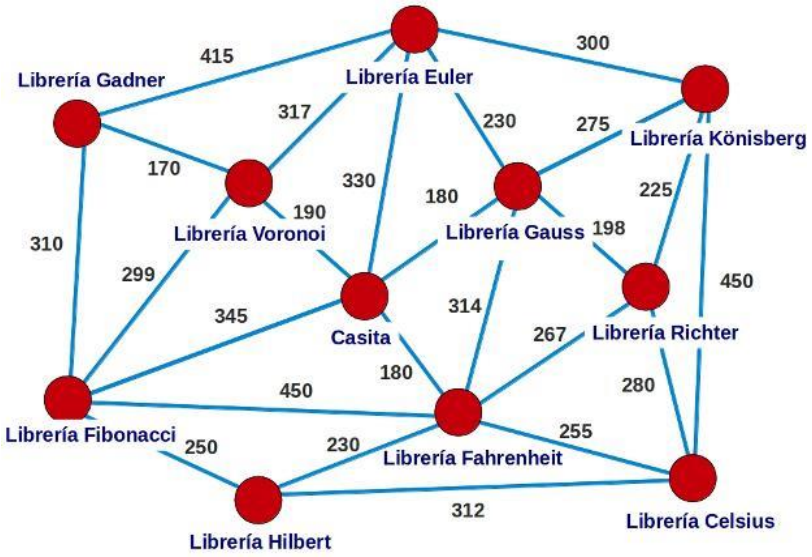


Imagen 6. Grafo del algoritmo Dijkstra

La implementación de este algoritmo se ha desarrollado en un nodo de ROS que se suscribe al topic */map,* del que recibirá el mapa de costes a partir del cual podrá generar un grafo sobre el que trabajar. También recibe como parámetros del archivo “.launch” los puntos de origen y destino. El cálculo de la trayectoria se realiza dentro de una función *mapCallback* que se ejecutará una vez recibido el mapa de costes y la almacenará en la variable *path,* un array de la clase nav_msgs. Para el cálculo de esta se comienza tomando la información del mapa y de los puntos de origen y destino, creando una tabla de distancias, una cola de prioridad y una matriz de puntos vecinos. A continuación, se explorará de manera iterativa el mapa hasta llegar al destino o hasta vaciar la cola. Una vez hecho esto reconstruimos la trayectoria desde el punto final y la invertimos para que la orientación sea correcta. Una vez calculada, se publicará en el topic */path,* de donde podrá tomarla el planificador local.

Pure-Pursuit

Este nodo es el encargado de realizar el control de nuestro robot de manera local, es decir, es el encargado de seguir una trayectoria que recibe del planificador global detallado en el apartado anterior. Con este fin hemos implementado un controlador de trayectorias tipo Pure Pursuit que establecerá las velocidades adecuadas para que el robot siga la trayectoria.

Funcionamiento general de un controlador pure pursuit.

A nivel general, este tipo de controladores calculan la velocidad angular que debemos aplicar a un robot para alcanzar un punto Look-Ahead de la ruta que se sitúa delante del mismo. Una vez alcanzado este punto, con más o menos precisión según el controlador sea implementado, este será recalculado. De manera iterativa conseguiremos mover al robot a través del camino deseado hasta el punto objetivo.

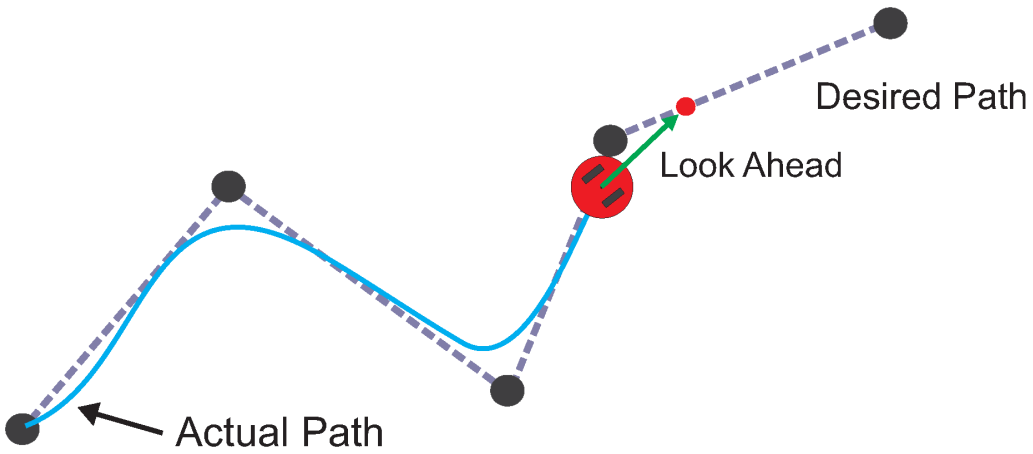


Imagen 7. Funcionamiento general del método de persecución pura

Para hacer funcionar un algoritmo de esta índole debemos pasar al mismo una lista de waypoints que darán forma a la ruta y una distancia de Look-Ahead, que será la distancia máxima entre el robot y el siguiente punto al que se desplace el mismo. Además, podemos limitar las velocidades lineales y angulares máximas que desarrolle nuestro robot para llegar al punto indicado, suponiendo que la primera se mantendrá constante. Eligiendo adecuadamente estos 3 parámetros podemos ajustar el movimiento del robot a las especificaciones del mismo, del entorno y a las posibles exigencias y limitaciones que se nos planteen.

El cálculo de las velocidades será específico para el sistema de movimiento que tenga nuestro vehículo, siendo en nuestro caso un sistema de 4 ruedas motrices diferencial (aplica velocidades distintas a sus motores para realizar trayectorias curvas).

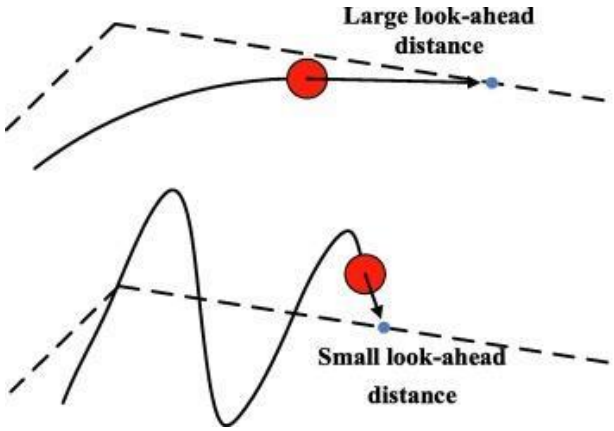


Imagen 8. Influencia del parámetro 'look-ahead'

Aplicación del controlador a nuestro caso:

Con el objetivo de aplicar esto a nuestro robot se ha desarrollado el nodo "pure_pursuit_node.cpp" desde el que se llevará a cabo la tarea del control local. Para ello, nuestro nodo estará suscrito a los topics */odom* y */path*. Del primero recibe la pose del robot (objeto *pose* de la clase *geometry_msgs*) en todo momento, mientras que del segundo recibirá un array (objeto *path* de la clase *nav_msgs*) con las posiciones que conforman la trayectoria a seguir. Para cada una de las suscripciones tendremos una función *callback* que almacenará los datos recibidos en sus correspondientes variables, las cuales mantendrá actualizadas conforme se vaya recibiendo nueva información. Además, para la odometría se extrae el ángulo de guiñada (rotación respecto del eje z) del cuaternión recibido con la orientación.

Al mismo tiempo el nodo publicará en el topic */cmd_vel* las velocidades lineales y angulares que debe tener el robot en todo momento, pasándolas como una estructura *Twist* de la clase *geometry_msgs*.

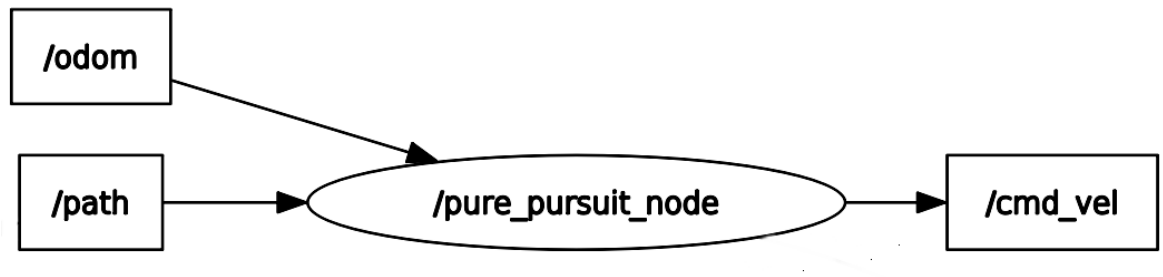


Imagen 9. Conexiones del nodo

Respecto al controlador, este estará implementado dentro de la clase *PurePursuitController,* en la que se crea un objeto pasando a su constructor los parámetros del controlador. Dentro de esta clase, la función *getCmdVel()* llevará a cabo el cálculo de las velocidades apoyándose en las siguientes funciones para obtener un punto objetivo:

- getDist(): devuelve la distancia entre dos puntos como el módulo de un vector entre ambos.
- getWaypoint(): devuelve el punto de la trayectoria global, por delante del robot, más cercano a este.
- getTarget(): a partir del waypoint más cercano, calcula el punto de la trayectoria global más alejado que puede alcanzar dentro de la *look-ahead distance* que hayamos elegido.

Una vez calculado el punto objetivo, se comprobará si este es el punto final de la trayectoria global, para en caso contrario calcular las velocidades a aplicar de la siguiente manera:

- Velocidad lineal: se establecerá al su valor máximo.
- Velocidad angular: se calcula como la máxima velocidad angular por el seno del ángulo entre el robot y el punto objetivo, asegurando una velocidad menor conforme el robot se acerca al punto deseado.

Con esto, para hacer funcionar nuestro nodo, sólo se debe esperar a recibir una trayectoria a través del tópico */path,* para llamar así a la función *getCmdVel()* y publicar el resultado en el tópico */cmd_vel*. Realizando esto de manera recursiva dentro de un bucle en la función principal, se logrará que el robot se desplace de forma satisfactoria al punto objetivo.

Para realizar un análisis de los resultados obtenidos, desde la función *callback* de la odometría, se deben escribir en un archivo de texto las coordenadas del robot, su orientación y las coordenadas de los puntos de la trayectoria global. Este archivo será interpretado con posterioridad utilizando *MatLab.*

Lanzamiento de sistema

Para poder ejecutar nuestro sistema tenemos que lanzar nodos de 2 paquetes diferentes, el paquete *rosbot_description* y el paquete *pure_pursuit*. Con este motivo se ha confeccionado un archivo *pure_pursuit.launch* desde el cual indicamos a ROS cuales son los nodos que debe ejecutar y algunos parámetros extra que nos serán de utilidad.

Desde este archivo ".launch" se lanza en primer lugar el modelo del robot, así como el entorno de gazebo en que este se moverá, pudiendo elegir desde aquí cuál de los 3 mundos queremos utilizar. A continuación, se lanzan los nodos correspondientes al controlador local y global de trayectorias, el generador de mapas de costes y el visualizador RVIZ.

Además de los propios nodos, se indican una serie de argumentos mediante el comando *param* entre los que destacan el mapa de costes a utilizar (existiendo uno asociado a cada mundo de gazebo), los puntos de origen y destino respecto a los que se calcula la trayectoria y los parámetros del controlador (look-ahead distance y velocidades máximas). De esta forma se tendrá la posibilidad de modificar rápidamente la mayoría de parámetros que rigen el funcionamiento de nuestro sistema sin tener que recompilar cada vez, permitiendo así hacer pruebas de manera sencilla.

Para lanzar el sistema de esta manera basta con abrir un terminal y dentro de la carpeta donde se ubiquen nuestro proyecto ejecutar el comando:
roslaunch pure_pursuit pure_pursuit.launch

De esta forma, se obtendrá el siguiente esquema final para nuestro robot.

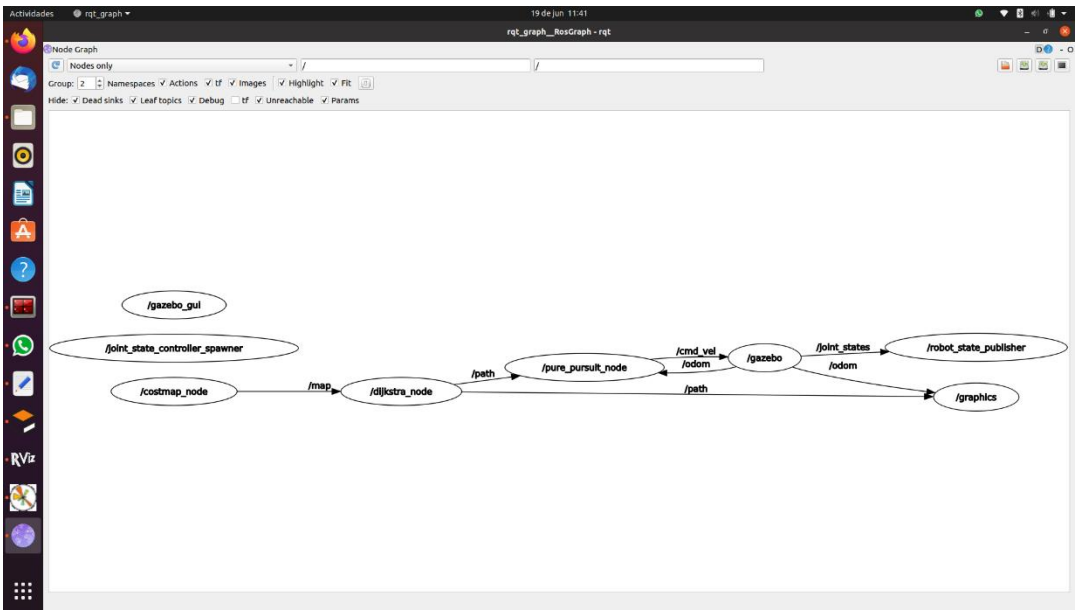


Imagen 10. Esquema final de conexiones del robot

Resultados

En este apartado se hará una breve exposición de los distintos experimentos realizados con nuestro controlador, así como la finalidad de estos y los datos más significativos para obtener conclusiones sobre los límites de funcionamiento del método de persecución pura empleado.

Como se ha comentado con anterioridad, se realizarán simulaciones sobre tres mapas distintos, que permitirán observar como el movimiento del robot se ajusta a las distintas trayectorias dependiendo de las condiciones del entorno que esté a su alrededor.

Análisis de experimentos en el primer mundo

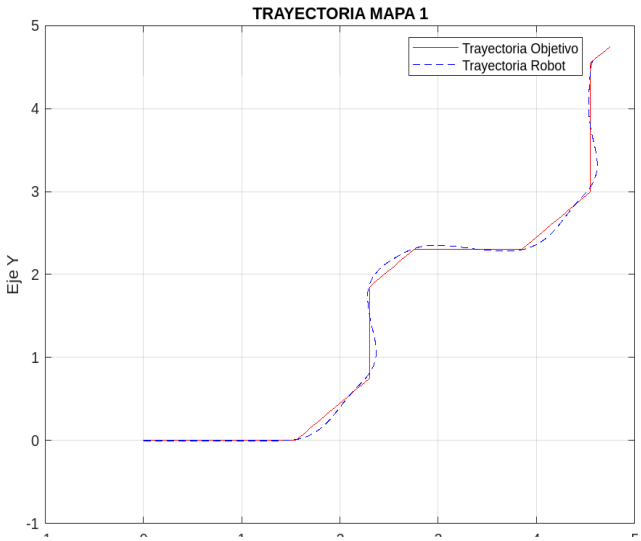
Como se ha visto en apartados anteriores, este entorno de simulación creado no es más que una habitación con 4 obstáculos colocados de forma simétrica. El objetivo de este mapa es poder realizar unas primeras pruebas del nodo programado para el seguimiento de trayectorias, de forma que se puedan ir ajustando los distintos parámetros del controlador para que este haga seguir al robot con la mayor precisión posible la trayectoria global generada por el planificador con algoritmo de Dijkstra.

Este mapa permite gran versatilidad a la hora de hacer experimentos para un ajuste de parámetros, ya que, aunque las trayectorias generadas sean prácticamente iguales independientemente del punto objetivo, permite determinar en un rango satisfactorio los valores de las distintas variables para que el robot siga la trayectoria de forma aceptable.

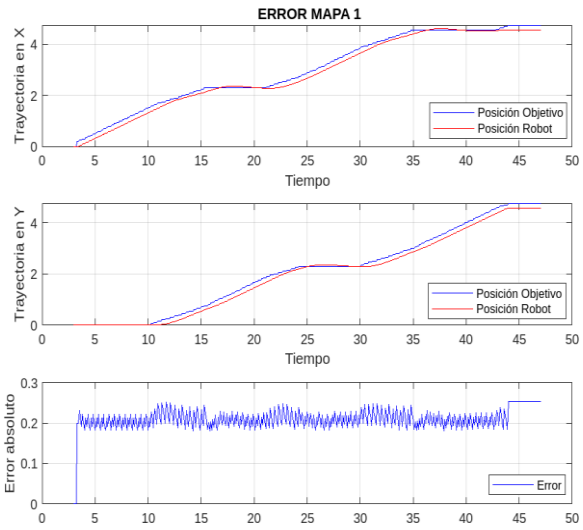
En primer lugar, se han ajustado los parámetros de la siguiente forma:

- Look-ahead = 0.2
- Max-linear = 0.2
- Max-angular = 1.0

Obteniendo el siguiente resultado:



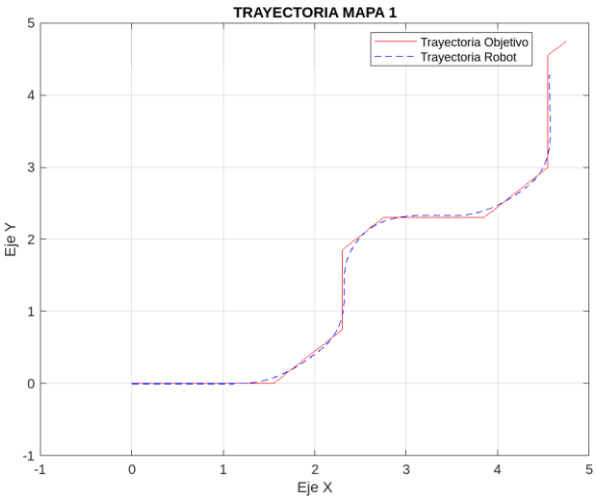
Gráfica 1. Trayectoria experimento 1



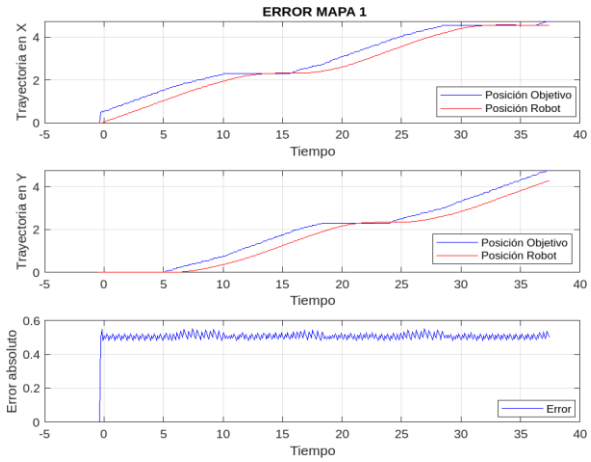
Gráfica 2. Errores experimento 1

Como se puede observar, el robot sigue la trayectoria correctamente. Sin embargo, hay un error grande a la hora de realizar giros, ya que, al tener una velocidad angular baja, el robot tarda mucho en recuperar la posición que debería.

Para el segundo experimento, se quiso probar a aumentar la variable look-ahead, para ver si así evitaba los giros obteniendo puntos más alejados de este. No obstante, los resultados no fueron los esperados, ya que la precisión disminuyó en gran medida y la distancia final al punto objetivo se hizo más grande.



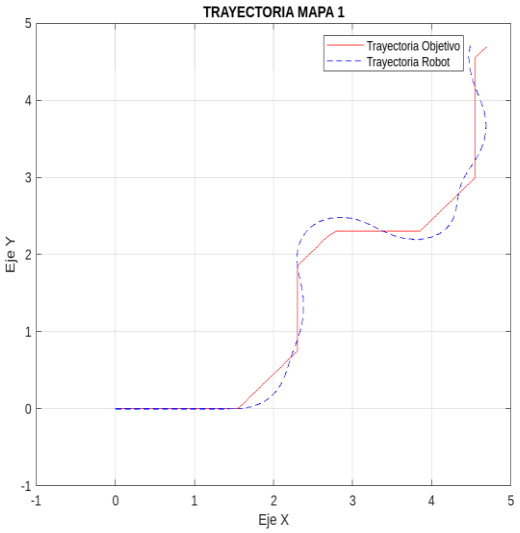
Gráfica 3. Trayectoria experimento 2



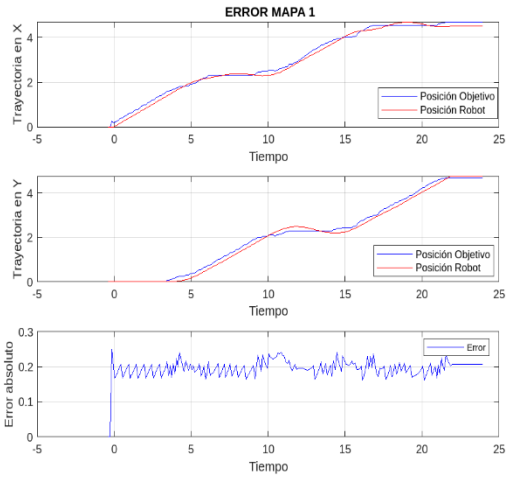
Gráfica 4. Errores experimento 2

Como ya se ha comentado antes, en la segunda gráfica se observa como el error de las coordenadas del robot es bastante mayor al del apartado anterior.

El siguiente pasó fue aumentar el valor de la velocidad lineal, para así poder obtener conclusiones de como afectaban los distintos valores del control al funcionamiento del robot. De esta forma, se aumentó a un valor de 0.5 la velocidad lineal, obteniendo así los siguientes resultados:



Gráfica 5. Trayectoria experimento 3



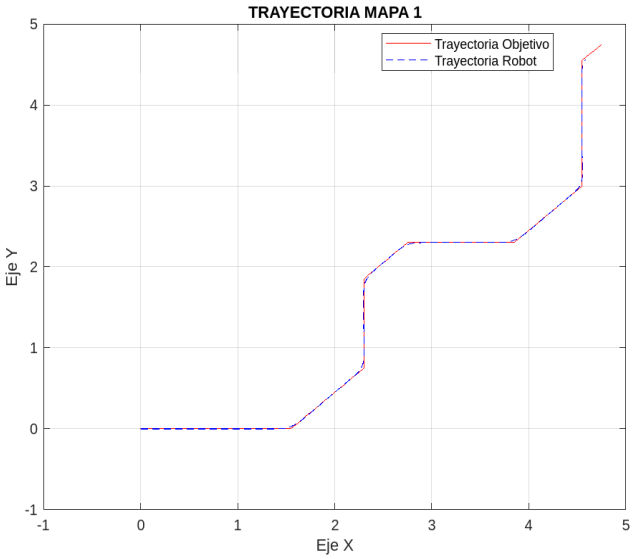
Gráfica 6. Errores experimento 3

Es bastante apreciable como un valor de velocidad demasiado elevado no permite al robot realizar giros consecutivos con precisión, ya que este no dispone del tiempo suficiente como para poder seguir la trayectoria ideal.

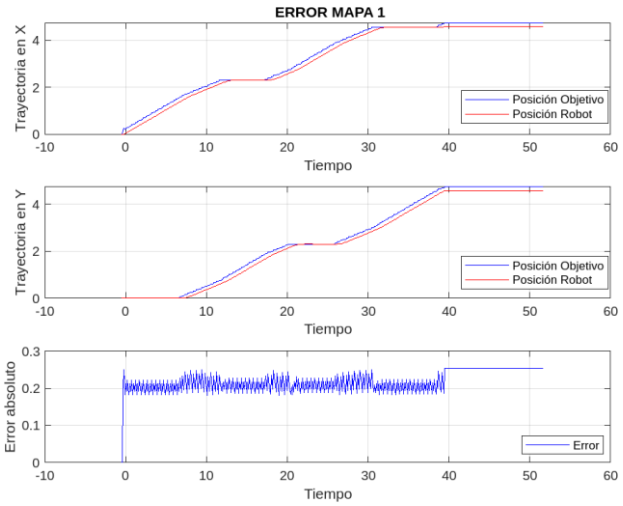
Tras realizar numerosas simulaciones ajustando los parámetros de distintas formas, se llegó a la conclusión de que los siguientes valores eran los que permitían el funcionamiento más correcto para el seguimiento de trayectorias.

- Look-ahead = 0.2
- Max-linear = 0.2
- Max-angular = 2.5

Con estos valores, los resultados finales fueron los siguientes.



Gráfica 7. Trayectoria experimento 4



Gráfica 8. Errores experimento 4

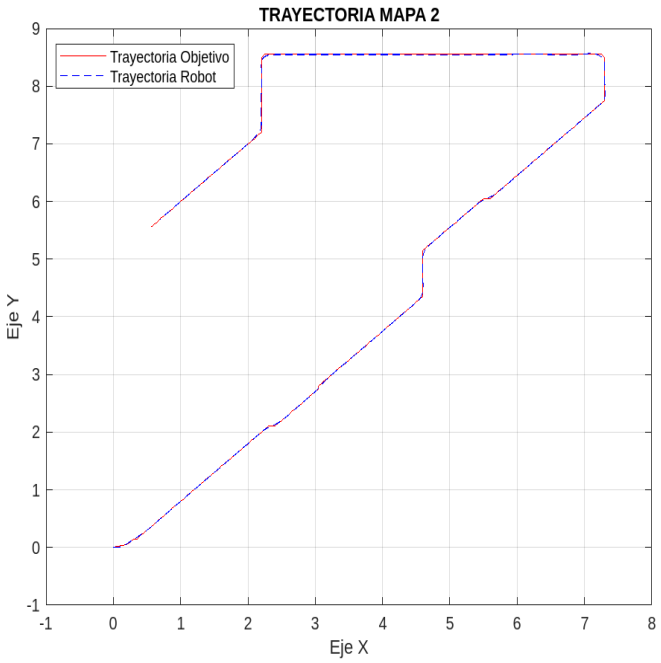
Es bastante esclarecedor ver cómo el ajuste a la trayectoria es bastante bueno, teniendo errores ínfimos tanto en trayectos lineales como en los giros de las esquinas. Sin embargo, no se podría concluir si estos parámetros son los mejores para este controlador, ya que sería necesario observar cómo se podría trabajar con un entorno que permita programar distintas trayectorias. Para ello, se mostrarán más experimentos realizados sobre dos mapas adicionales, logrando así realizar un número de pruebas suficiente como para obtener conclusiones válidas de nuestro proyecto.

Análisis de experimentos en el segundo mundo

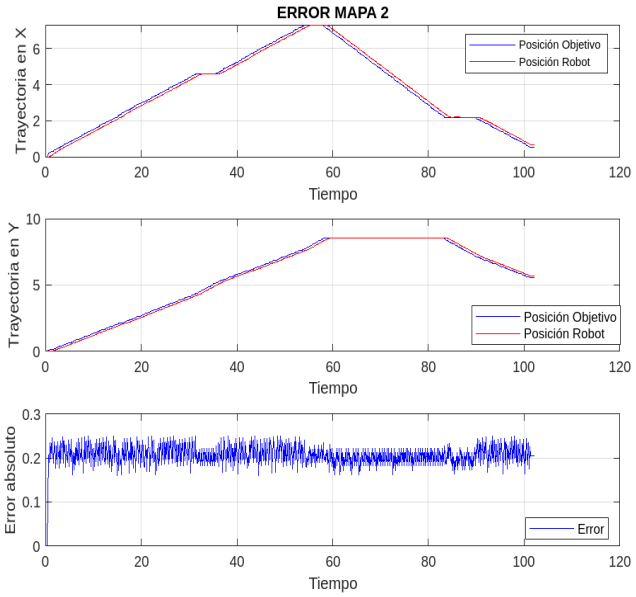
Con anterioridad se ha mostrado como este segundo entorno se basaba en el mapa de una habitación dividida por zonas más grandes, de forma que los giros del robot permitan un mayor margen de choque sin que este colisione con alguna pared. Además, el principal objetivo de este nuevo mapa será probar distintos tipos de trayectoria, ya que ofrece una mayor variedad a la hora de planificar caminos que el ejemplo anterior.

En primer lugar, se ha realizado una prueba con los parámetros finales que se obtuvieron sobre el anterior mapa, pudiendo así verificar que estos valores permiten que el controlador funcione de forma correcta independientemente de la trayectoria planificada o del entorno en el que se encuentre el robot.

Los resultados obtenidos para el experimento pueden verse en las siguientes gráficas.

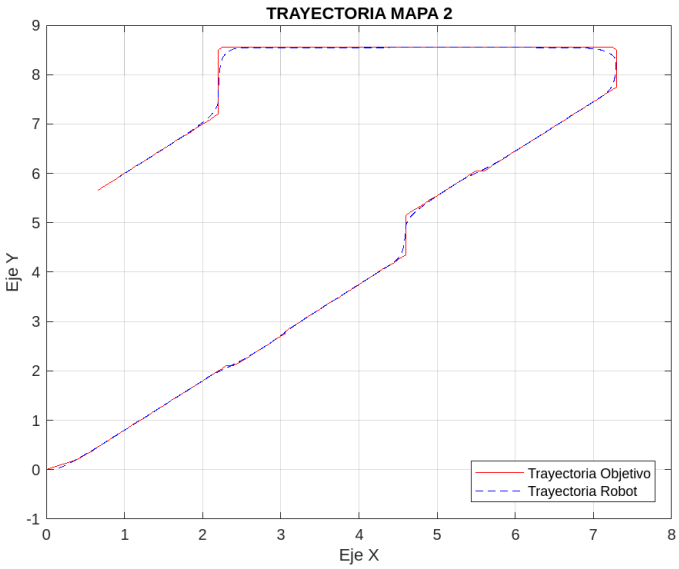


Gráfica 9. Trayectoria experimento 5

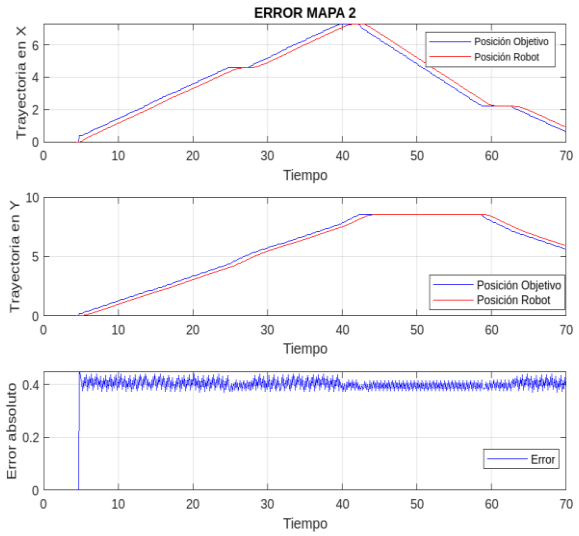


Gráfica 10. Errores experimento 5

Como se puede apreciar, el robot vuelve a seguir la trayectoria planificada de forma satisfactoria, además de hacerlo con un error pequeño y que se mantiene bastante estable. No obstante, se vio la necesidad de intentar realizar pruebas con una velocidad lineal mayor, que permitiese recorrer este mapa de grandes dimensiones en un rango de tiempo más acortado.

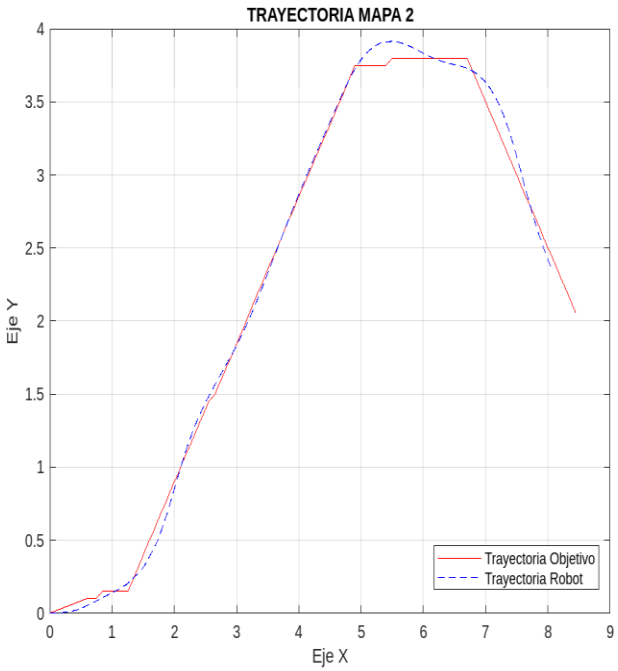


Gráfica 11. Trayectoria experimento 6

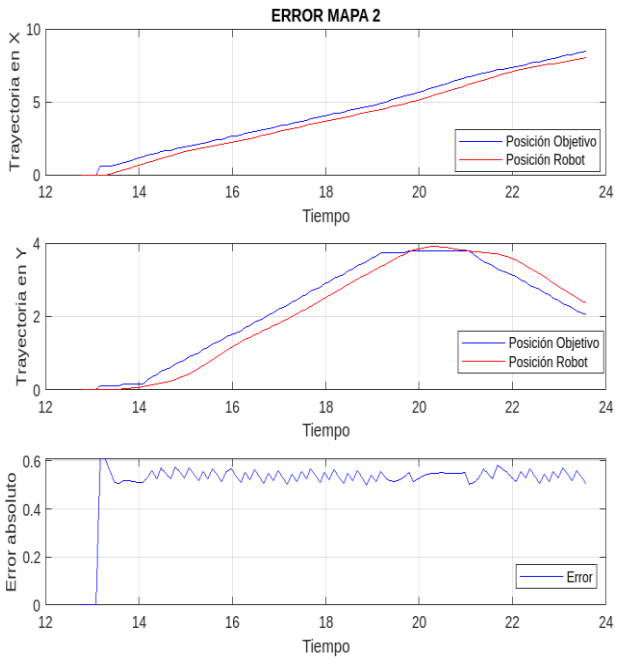


Gráfica 12. Errores experimento 6

Los resultados mostrados son mejores a lo esperado inicialmente, permitiendo así ver cómo disponiendo de trayectorias más sencillas, es posible aumentar la velocidad del robot de forma moderada manteniendo un error que no sea excesivamente elevado. Sin embargo, ¿dónde se encuentra este límite de ajuste de parámetros?, ¿hasta dónde puede parecer válido el ajuste del robot a la trayectoria? No parece algo fácil, pero se ha intentado hacer una prueba elevando aún más el parámetro de velocidad lineal, además de cambiar el punto objetivo para así poder disponer de una trayectoria lo más simple posible, de forma que se han obtenido los resultados mostrados a continuación.



Gráfica 13. Trayectoria experimento 7



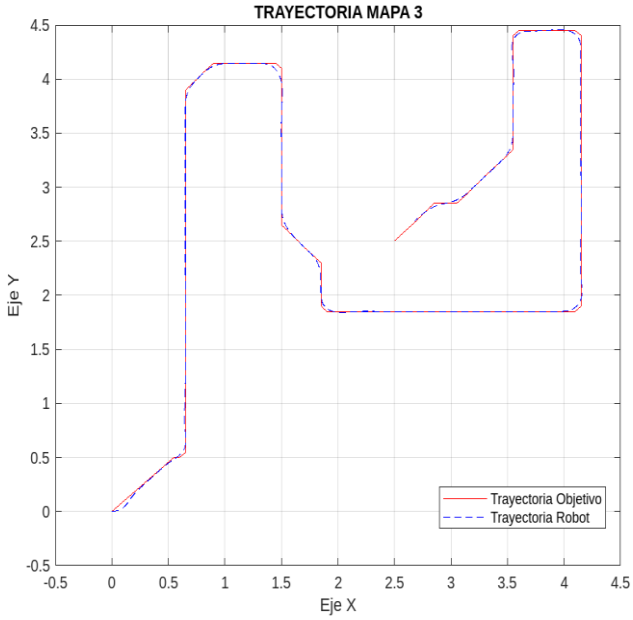
Gráfica 14. Errores experimento 7

Como se puede observar, dando un valor igual a uno a la velocidad, los errores máximos obtenidos al realizar giros son muy pronunciados. Por tanto, a pesar de seguir la trayectoria de forma más o menos correcta, es mejor hacer uso de los valores que se han obtenido en el último experimento realizado sobre el primer mapa, ya que son los que permiten un ajuste más fino al camino planificado, además de ser esta la mejor forma de evitar colisiones en giros muy cerrados donde el espacio sea reducido.

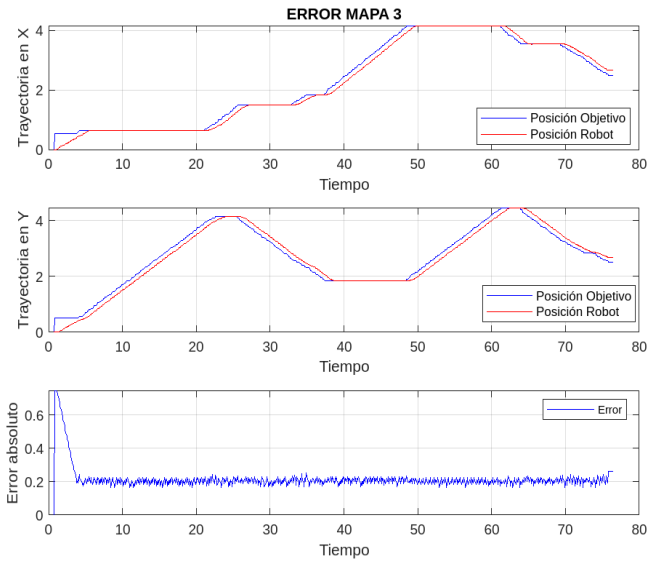
Análisis de experimentos en el tercer mundo

Con todos los resultados anteriores, ya se puede disponer de la gran mayoría de conclusiones del proyecto, pero, ¿cuál es el límite de aplicación de este controlador? Es decir, ¿puede trabajar bien en cualquier tipo de entorno?

Para ello, se ha creado este último mapa, donde se ha realizado una simulación con los parámetros que mejor funcionan para nuestro controlador, obteniendo así las siguientes gráficas.



Gráfica 15. Trayectoria experimento 8



Gráfica 16. Errores experimento 8

Como se puede ver, a pesar del reducido espacio con el que cuentan estos pasillos, el robot mantiene el seguimiento de trayectorias con un error mínimo, evitando también colisiones en el momento de realizar giros cerrados sobre esquinas donde se dispone de un espacio muy reducido. Sin embargo, en el momento en el que se cambia cualquier valor del controlador, el robot colisiona y nunca termina de realizar el camino que se le da, permitiendo así ver como influye también el entorno donde se encuentre en la validez del funcionamiento de este tipo de seguimiento.

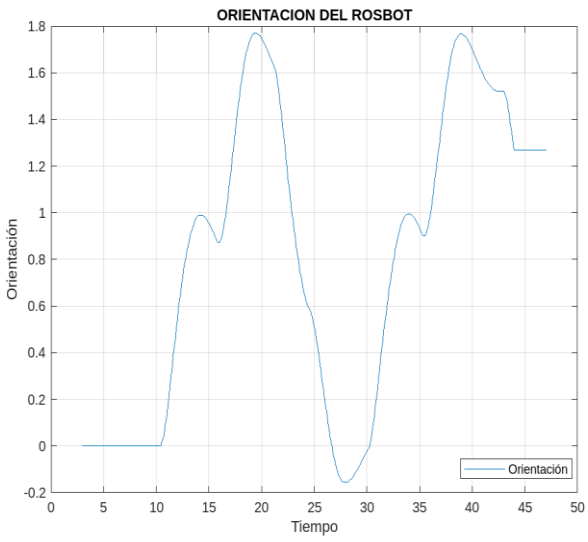
Conclusiones

Tras la realización de todos estos experimentos, es interesante recopilar también una tabla comparativa con los errores obtenidos a lo largo de las pruebas. Por ello, se puede observar a continuación.

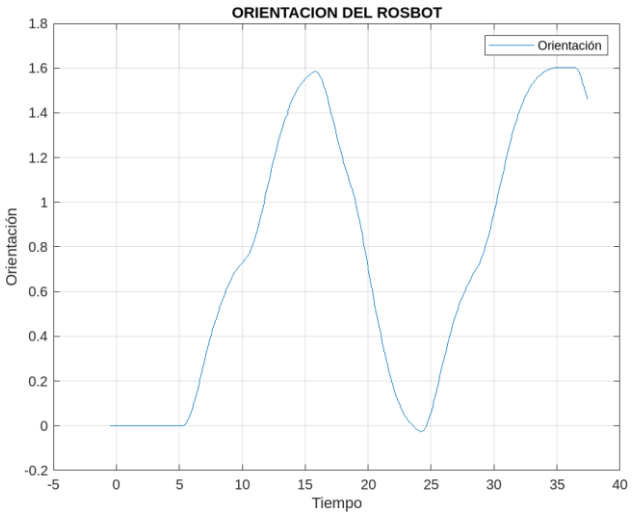
	MUNDO 1_1	MUNDO 1_2	MUNDO 1_3	MUNDO 1_4	MUNDO 2_1	MUNDO 2_2	MUNDO 2_3	MUNDO 3_1
error max X	0,2319	0,55	0,25	0,25	0,2301	0,4147	0,5999	0,55
error max Y	0,231	0,5272	0,2297	0,227	0,2312	0,4167	0,4828	0,5003
error medio X	0,1281	0,2964	0,1231	0,1313	0,1424	0,2787	0,4043	0,0876
error medio Y	0,1288	0,3031	0,1181	0,1376	0,1187	0,2305	0,2847	0,1554
error min X	1,26E-06	8,03E-06	3,44E-06	8,02E-06	8,03E-06	1,48E-06	7,62E-06	1,08E-05
error min Y	2,17E-06	3,68E-06	1,68E-06	2,90E-06	0	5,39E-06	1,33E-06	0

Tabla 1. Comparativa de errores en las distintas pruebas

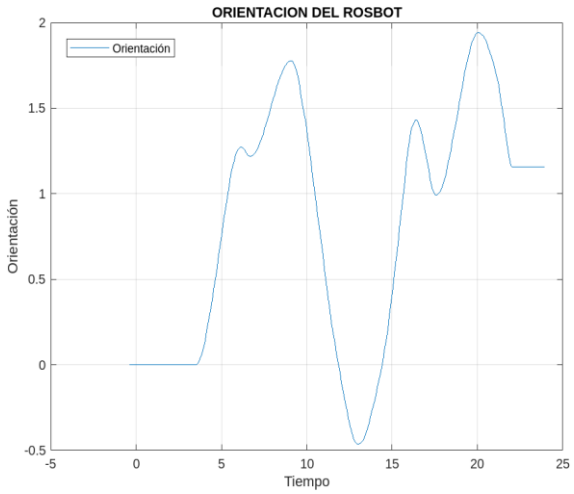
Además, también es algo interesante observar como dependiendo de la velocidad del robot y del número de puntos que este debe tomar para realizar el seguimiento, este puede llegar al punto objetivo con una orientación distinta a la esperada. Para reflejar esto, se procede a adjuntar una comparación de la evolución de la orientación del robot en cada uno de los 4 experimentos realizados sobre el primer mapa.



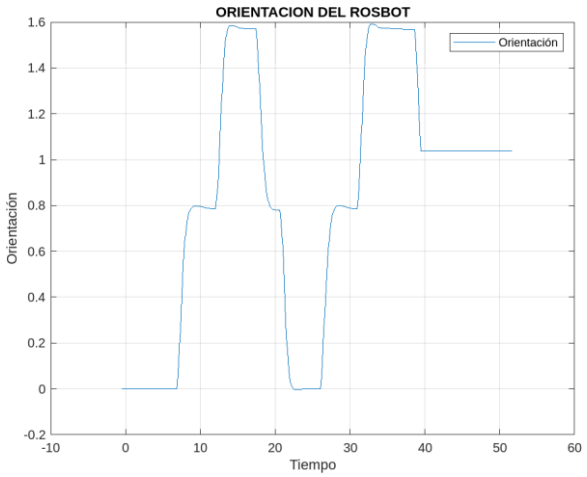
Gráfica 17. Orientación en experimento 1



Gráfica 18. Orientación en experimento 2



Gráfica 19. Orientación en experimento 3



Gráfica 20. Orientación en experimento 4

Como se puede apreciar, el valor del parámetro 'look-ahead' influye enormemente en la orientación del robot, ya que, en el único experimento dónde este cambia, el robot no llega con una orientación constante al punto objetivo. Por otra parte, también se observa como el cambio de velocidad también influye en la orientación del robot, así como el valor final con el que este llega al punto objetivo.

Tras las numerosas pruebas realizadas, es fácil obtener muchas conclusiones sobre este tipo de control que se ha programado para nuestro proyecto, como cuáles son los valores que permiten una mayor precisión a la hora de realizar el seguimiento o cómo afecta cada uno de ellos a la evolución de la trayectoria que hace el robot. Además, se ha podido comprobar también sobre qué tipos de entornos se puede implementar este control de forma satisfactoria.

Sin embargo, ni mucho menos esto es todo lo que se podría exprimir de este proyecto, ya que existe un abanico de posibilidades muy amplio para poder poner a prueba el método de persecución pura. Por ejemplo, ¿qué pasaría si la trayectoria fuese planificada con otro algoritmo? Y es que, es bastante claro que quizás un algoritmo 'A estrella' o cualquier otro pueda permitir un mayor rango de funcionamiento del robot, ya que, el algoritmo de Dijkstra calcula trayectorias muy restrictivas, que no permiten un gran margen de cambio en los parámetros del control, como se ha visto con anterioridad.

Por otra parte, también existe la posibilidad de cambiar el tipo de superficie por que el que deslice el robot, teniendo así suelos con mayor o menor rozamiento, como por ejemplo el hielo. Otro factor que se podría tener en cuenta es cómo podría afectar por ejemplo una fuerte ráfaga de viento al movimiento del robot, que quizás, podría evitar que este volcase o no pudiese seguir correctamente la trayectoria planificada.

Con los puntos propuestos anteriormente, es bastante claro que existe un amplio horizonte para seguir realizando pruebas sobre el método de persecución pura a pesar de la simpleza de este algoritmo. Sin embargo, las conclusiones esenciales más importantes y esenciales se han obtenido, habiéndose cumplido el propósito de este proyecto. Y es que, a pesar de la simpleza de este método de seguimiento de trayectorias, es posible que este pueda ser usado para algunas aplicaciones que no requieran un control muy complicado.

Por último, aunque hubiese sido muy interesante analizar la robustez de este control ante más situaciones, se ha comprendido como afectan los distintos parámetros a la evolución de la trayectoria seguida por el robot, así como el funcionamiento general de como se debe trabajar en primera instancia con este tipo de robots, quedando satisfechos tanto con los resultados obtenidos como con el proceso de trabajo realizado.