Universidad Zaragoza
1542

Trabajo Fin de Grado

# Control de una formación de "platooning" usando un equipo de robots

## Control of a platooning formation using a team of robots


Autor

**Víctor Gallardo Sánchez**

Directores

Rosario Aragüés Muñoz

Luis Riazuelo Latas

ESCUELA DE INGENIERÍA Y ARQUITECTURA



2024

# Resumen

Un robot según la RAE es **una máquina o ingenio electrónico programable que es capaz de manipular objetos y realizar diversas operaciones.** Estos autómatas están muy presentes en la sociedad actual, ya sea en fábricas, en servicios de ocio, servicios militares, medicina y en cualquier campo que podamos imaginar. Es por esta creciente importancia que la investigación en este ámbito aumenta y va generando un mayor interés en la población. Hay muchos tipos de robots y son sistemas tan complejos que dan lugar a un sinfín de ramas que se pueden estudiar. En nuestro caso, vamos a centrarnos en los **robots móviles**.

La robótica móvil es una parte de la robótica que engloba aquellos robots que son capaces de trasladarse en un entorno dado. Es decir, robots que tienen la capacidad de moverse en su entorno. Dentro de esta rama, existe el concepto de "platooning" que es **una agrupación de vehículos que siguen una misma trayectoria**. Esto se puede ver, por ejemplo, en una serie de camiones que realizan el mismo trayecto o un conjunto de robots transportadores en una fábrica.

Este TFG pretende que varios robots móviles realicen una misma trayectoria conformando un *platooning*. Para ello se estudiará el artículo [5] y se realizará una implementación adaptada en lenguaje C++ de dos de las trayectorias concretas que proponen en él. Se utilizará el entorno ROS [1] de trabajo para llevar a cabo las tareas correspondientes. Se añadirá el uso de un sensor láser de percepción para realizar una detección de obstáculos. Además, se probará la implementación desarrollada en el simulador STAGE y en un entorno real con robots móviles diferenciales "Turtlebots".

# Agradecimientos

En primer lugar, quiero dar las gracias a mis profesores, Rosario y Luis. Gracias por todo el trabajo y esfuerzo que habéis dedicado a este proyecto. Sin vosotros nada de esto habría sido posible.

También quiero dar las gracias a mis padres y a mi hermana, por haberme apoyado incondicionalmente durante todo este tiempo y por estar ahí siempre que los he necesitado.

Gracias a todos.

# Índice

# Capítulo 1

# Introducción

## 1.1. Motivación

El estudio de la conducción autónoma y la coordinación de vehículos ha evolucionado enormemente estos últimos años en el ámbito de la robótica. En particular, el concepto de *platooning* se considera un avance significativo debido a que presenta unos beneficios enormes en áreas como el transporte, seguridad vial o reducción de emisiones.

Actualmente se utilizan conjuntos de robots en muchos de los sectores ya nombrados. Esto se debe a que tener una flota de vehículos que realizan una misma trayectoria mejora de manera muy eficiente los recursos demandados por las empresas. Además, agiliza el transporte y reduce gastos económicos.

Sin embargo, el control de formaciones de vehículos plantea desafíos muy complejos puesto que implica muchos aspectos a tener en cuenta como la convergencia de los diferentes robots en la misma trayectoria, la navegación de dichos vehículos dentro de ella, las fuerzas de repulsión que mantienen al *platooning* unido o incluso los obstáculos que se generan entre los propios robots, donde podría haber peligro de choque entre ellos.

En resumen, esta rama de la robótica tiene un gran potencial por delante. No obstante, apenas se acaba de empezar a estudiar. No hay demasiada información al respecto y no es un tema que esté para nada asentado, lo que la hace una disciplina muy interesante donde investigar al respecto.

Por todo esto, este TFG pretende acercarse un poco más al mundo de la robótica y el control de *platooning*, investigando el artículo [5] y realizando una implementación adaptada de las ecuaciones que proponen en él. De esta manera se conseguirá un control de varios robots en una misma trayectoria.

## 1.2. Objetivos

El proyecto que se lleva a cabo quiere desarrollar un sistema donde varios robots realicen una misma trayectoria conformando un *platooning*. Para ello hay varios objetivos y problemas claros que hay que abordar:

- **Aprendizaje previo:**
  - Aprendizaje de ROS
  - Pruebas con STAGE
  - Entendimiento de sensores de percepción (láser)
- **Análisis y entendimiento de las ecuaciones matemáticas.**
  - Entendimiento de las ecuaciones generales de movimiento
  - Extracción de las ecuaciones importantes para trayectoria *Circular*
  - Extracción de las ecuaciones importantes para trayectoria *Lissajous*
  - Adaptación de las ecuaciones a un entorno discretizado.
- **Implementación de los métodos multi-robot para la navegación**
  - Implementación de lógica de movimiento.
  - Implementación de detección de obstáculos mediante el láser.
- **Implementación del algoritmo matemático de las trayectorias**
  - Implementación de trayectoria *Circular*.
  - Implementación de trayectoria *Lissajous*
- **Desarrollo de una arquitectura que permita la comunicación entre las diferentes partes**
- **Validación en entorno simulado (*STAGE*)**
- **Validación en entorno real (*Turtlebots*)**

## 1.3. Metodología

Para el desarrollo de los objetivos que se han comentado en la sección 1.2 se ha utilizado el lenguaje de programación C++ y el entorno de desarrollo Robot Operating System (ROS) [1]. Este entorno trata de un conjunto de herramientas y bibliotecas de

software de código abierto diseñado para ayudar en el desarrollo de software de robots. Además, se ha utilizado el simulador STAGE [12] para llevar a cabo las pruebas y validaciones. En cuanto al entorno real, se han utilizado Turtlebots como robots del *platooning*.

El trabajo realizado se ha dividido en varias fases. A continuación, se muestra un resumen de cada fase de trabajo:

En primer lugar se ha **aprendido el entorno con el que se va a trabajar (ROS)** realizando varios tutoriales para entender las herramientas y distintas partes que lo conforman. Asimismo, se ha estudiado el simulador STAGE y el láser para la detección de obstáculos.

En segundo lugar, **se ha estudiado el artículo** [5] con el fin de entender las ecuaciones que este plasma. De él, se han extraído las ecuaciones más importantes y se han adaptado para la implementación que se iba a realizar.

Una vez hecho el análisis matemático, se pasa a la **fase de implementación**. Esta ha sido, sin duda, la fase que más tiempo ha ocupado, ya que requiere la creación de una arquitectura que permita la comunicación entre los nodos, la implementación de la navegación de los robots, la implementación de la trayectoria *Circular* y de la trayectoria *Lissajous*.

Posteriormente, se ha pasado a una **fase de pruebas y validación** donde se ha evaluado el sistema en el simulador STAGE, realizando diversas pruebas y observando diversos efectos sobre el *platooning*. En esta fase se ha depurado totalmente el código realizado arreglando pequeños *bugs* y problemas que han surgido.

Por último, se ha probado la implementación en un **entorno real** (mediante Turtlebots) para ver como respondía fuera del simulador. Para ello, se ha adaptado el código para que se conecte con los Turtlebots. Asimismo, se ha creado un mapa y se ha utilizado un filtro de partículas para localizar al robot. Los pasos seguidos y la explicación detallada se encuentran en el capítulo 7.1.

## 1.4. Estructura de la memoria

En esta sección se va a explicar como está dividida la memoria, en qué apartados y de qué trata cada apartado.

*Capítulo 1: Introducción.* Este es el apartado actual. Consta de la motivación del trabajo, los objetivos que se han planteado, la metodología a seguir y la estructura de la memoria.

*Capítulo 2: Análisis matemático del método de platooning.* Este capítulo resume el

artículo [5] que se ha estudiado. Posteriormente, resalta las ecuaciones más importantes que hay que tener en cuenta y por último, explica los cambios que se han hecho en dichas ecuaciones para adaptarlas a los ejemplos concretos que se han implementado.

*Capítulo 3: Herramientas y trabajo previo a la implementación.* En este capítulo se resume las herramientas que se han tenido que aprender para realizar este TFG. Se explicará principalmente ROS, de qué manera se ha aprendido y otros conceptos como el sensor de percepción láser. Como es una explicación extensa, se añade un Anexo B con toda la información relativa a este trabajo previo.

*Capítulo 4: Arquitectura del sistema.* Este capítulo empieza a explicar la implementación llevada a cabo. Se mostrará cómo es el sistema implementado, cómo se comunican los nodos y qué mensajes se han creado. Es fundamental para entender el funcionamiento de los diferentes nodos en conjunto.

*Capítulo 5: Definición del algoritmo e implementación* Este capítulo explica completamente la implementación realizada. En él se verá el nodo de movimiento implementado, la trayectoria *Circular* y la trayectoria *Lissajous*.

*Capítulo 6: Resultados en simulación.* Este capítulo expone los resultados principales que se han obtenido en la simulación. Además da otras pruebas donde se ve muy bien el efecto de diversos parámetros sobre el *platooning*.

*Capítulo 7: Experimentos en un entorno real.* Muestra los pasos realizados para adaptar parte de los resultados al entorno real. Asimismo, añade vídeos y fotos donde se ve el funcionamiento de los Turtlebots. Se referencia el Anexo A en este capítulo para la explicación del filtro de partículas.

*Capítulo 8: Conclusiones.* Resumen y conclusiones de lo aprendido en este trabajo. Añade una valoración subjetiva del autor del mismo y futuras líneas de trabajo. Referencia el Anexo C con un diagrama de Gantt del trabajo realizado.

Como enlaces de importancia para el lector se destacan la lista de reproducción de los vídeos que se van a mostrar durante este trabajo [8] y el enlace al repositorio de GitHub que contiene el código desarrollado [3].

# Capítulo 2

# Análisis matemático del método de *platooning*

En este capítulo se va a explicar a fondo el artículo que se ha estudiado para la realización de este TFG [5].

Antes de comenzar, cabe destacar que es un artículo bastante denso a nivel matemático, en el cual se generalizan demasiado las ecuaciones y no se centra tanto en casos concretos de aplicación. Por ello, se hará primero un resumen de la parte teórica del artículo, es decir, conceptos importantes que hay que saber y como desarrollan el problema. Toda esta parte teórica se encuentra en la sección 2.1.

Posteriormente, se explicará como se ha modificado la función de repulsión que proponen en el artículo para adaptarla a la implementación que se ha realizado. En la sección 2.2 se encuentra las explicaciones correspondientes.

Por último, se entrará en detalle con los casos prácticos (secciones 2.3.1 y 2.3.2). De esta manera, se adaptará toda la parte teórica que se ha visto a los casos concretos de aplicación que ellos mismos proponen en él.

## 2.1. Resumen del artículo

El artículo [5] propone un algoritmo de campo vectorial guiado distribuido (en inglés, *distributed guiding vector-field*) para la ordenación espontánea de un *platooning* de robots que se mueven en una trayectoria predefinida. A partir de ahora, a este algoritmo lo nombraremos como *DGVF*. Un campo vectorial guiado es un espacio donde, mediante una serie de vectores, se representan direcciones o intensidades de magnitudes como pueden ser las fuerzas. La figura 2.1 representa como se vería un campo vectorial en distintas trayectorias.

Este artículo está dividido en 3 partes principalmente. En esta sección se verán las 2 primeras partes que son:

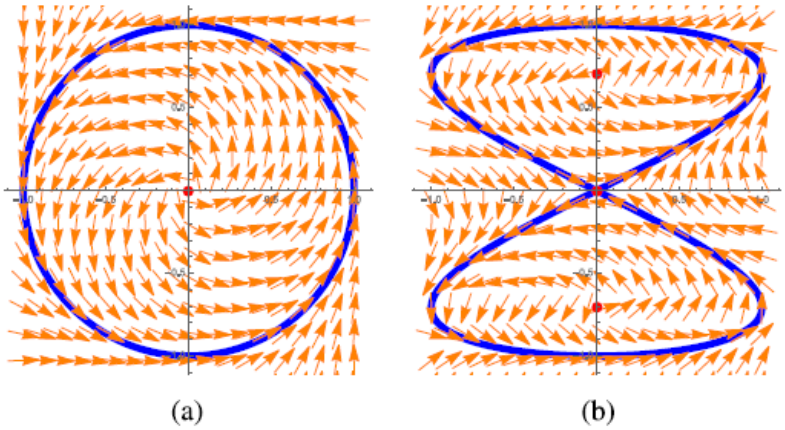



Figura 2.1: Campos vectoriales para distintas trayectorias (obtenida del artículo [21]).

– Conceptos previos que se definen, *preliminares*. Es decir, conceptos que hay que saber y tener en cuenta para el desarrollo del problema.

– Desarrollo del problema, *ecuaciones teóricas importantes*. En esta sección se presentarán los resultados que ha obtenido el artículo de sus estudios.

Con todo esto se pretende conseguir una ordenación espontánea del *platooning* como se observa en la figura 2.2. De esta forma, se consigue que no haya un robot que sea siempre el líder, sino que dependiendo de las posiciones iniciales uno u otro hagan de líder. En la figura 2.2 se aprecia perfectamente este aspecto, en el que en dos experimentos distintos se ordenan de manera distinta.

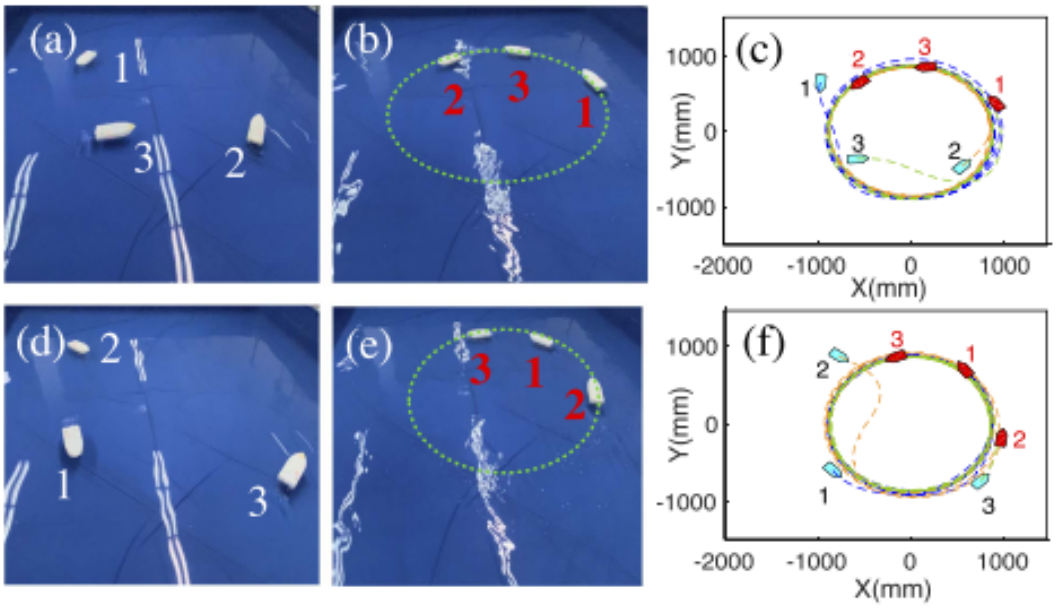



Figura 2.2: Ordenación espontánea en el círculo (obtenida del artículo [5]).

### 2.1.1. Preliminares

Supongamos que un camino deseado de n dimensiones en el espacio Euclídeo se representa como

$$P := \{\sigma \in \mathbb{R}^n \,|\, \phi(\sigma) = 0\}$$

donde cada $\sigma$ es una coordenada representada por una serie de funciones implícitas.

El artículo propone añadir una dimensión más que vamos a llamar $\omega$. Esta $\omega$ es una coordenada virtual que representa la posición de un robot en la trayectoria.

De esta manera, se puede definir el camino deseado como un camino con **una coordenada extra**:

$$P^{hgh} := \{\xi_i \in \mathbb{R}^{n+1} \,|\, \phi_{i,j}(\xi_i) = 0, i, j \in \mathbb{Z}_1^n\} \tag{2.1}$$

siendo $\xi_i := [\sigma_{i,1}, \ldots, \sigma_{i,n}, \omega_i]^T$ y $\phi_{i,j}(\xi) := \sigma_{i,j} - f_{i,j}(\omega), i, j \in \mathbb{Z}_1^n$ las funciones que miden el error del camino. El subíndice $i$ representa el robot y el subíndice $j$ la coordenada. De esta manera, por ejemplo, $\sigma_{i,1}$ sería la coordenada 1 del robot $i$. Además, $f_{i,j}(\omega)$ es la función de la trayectoria para la coordenada $w$. Se verá más adelante en ecuaciones de las trayectorias propuestas. Esto es la base principal de todo lo que se va a ver a partir de ahora. Lo que pretenden conseguir con este nuevo camino es el efecto que se aprecia en la figura 2.3, es decir, pasar de un camino en 3D estirado a un camino en 2D donde la $w$ marcará la posición de cada robot en dicha trayectoria.

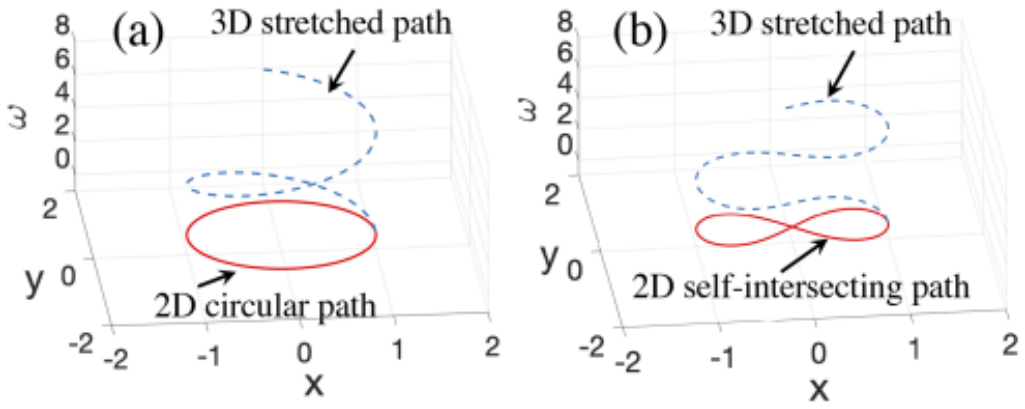



Figura 2.3: Efecto que se busca con el nuevo camino $P^{hgh}$ (obtenido del artículo [5]).

Asimismo, comenta que se va a trabajar con un *single integrator* para el movimiento de los robots el cual se representa como

$$\dot{x}_i = u_i + d_i, \quad i \in V \tag{2.2}$$

$V$ es el conjunto de robots $V = \{1, 2, \ldots, N\}$, $x_i$ representa la posición del robot i y $u_i$ representa la entrada que se le da al robot i (es decir, la acción que realiza). El término

$d_i$ representa las perturbaciones externas, sin embargo, en este trabajo no se va a tener en cuenta. La complejidad de este término y el cómo afectan agentes externos en los robots podría ser materia de una investigación muy diferente.

El artículo también **define un concepto de vecindario** que se va a utilizar de aquí en adelante para el *platooning*. El concepto se representa de la siguiente manera:

$$N_i(t) := \{k \in V, k \neq i \,|\, |\omega_{i,k}(t)| < R\} \tag{2.3}$$

donde R representa una distancia máxima de seguridad y $\omega_{i,k} := \omega_i - \omega_k$.

Por lo tanto, un vecindario de un robot son todos los robots que no sean él mismo, cuyas diferencias de posición virtual $\omega$ estén a menos distancia que la distancia máxima $R$. Este concepto será muy importante, como se verá posteriormente, para decidir que fuerzas se aplican en la repulsión y mantener al *platooning* unido.

Para concluir con los conceptos previos que hay que conocer, se definen 4 propiedades que, de manera teórica, se tienen que cumplir (ya que en la implementación se verá que no se tienen que cumplir exactamente así). Las propiedades son las siguientes:

1. $\lim_{t\to\infty} \phi_{i,j}(p_i(t)) = 0, \quad \forall i \in V, j \in \mathbb{Z}_1^n$
2. $\lim_{t\to\infty} \dot{\omega}_i(t) = \lim_{t\to\infty} \dot{\omega}_k(t) \neq 0, \quad \forall i \neq k \in V$
3. $r < \lim_{t\to\infty} \left|\omega_{s[k]}(t) - \omega_{s[k+1]}(t)\right| < R, \quad \forall k \in \mathbb{Z}_{N-1}^1$
4. $|\omega_{i,k}(t)| > r \quad \forall t \geq 0, \quad \forall i \neq k \in V,$

La **propiedad 1** explica que todos los robots acaban convergiendo al path deseado, es decir que acaban siguiendo la trayectoria que se busca. La **propiedad 2** representa que todos los robots se mueven a lo largo de la trayectoria, ya que $\dot{\omega}_i = u_i^w$, es decir, es la acción de la w para un robot i. Por lo tanto, esta propiedad se refiere, a que en tiempo infinito, la acción va a seguir siendo distinta de 0. Esto hace que el robot se haya movido respecto a la posición anterior con la misma velocidad que llevan sus vecinos.

**La propiedad 3 y 4 van ligadas.** La 3 indica que la diferencia de posición entre los robots siempre está entre una distancia mínima r y una distancia máxima R. Mientras que la 4 dice que dos robots siempre están a más de r de distancia.

Estas 2 últimas propiedades son las que se verá en la implementación que no tienen porque cumplirse exactamente, ya que dependen de las fuerzas de repulsión y el acercamiento a un concepto que veremos más adelante que es el objetivo virtual

($\omega^*$). Además como cambiaremos la función de repulsión, añadirá algún parámetro nuevo que también hace que se modifique la distancia de los robots.

### 2.1.2. Ecuaciones teóricas importantes

Una vez entendidos los conceptos preliminares que se han explicado, el artículo propone las ecuaciones en las que se basa el comportamiento deseado. Partiendo de la base de que se utiliza el movimiento de *single integrator* que se veía en la ecuación (2.2), el artículo define la acción de cada dimensión como:

$$u_{i,j} = (-1)^n \partial f_{i,j} - k_{i,j}\phi_{i,j} + \hat{d}_{i,j} \quad \forall j \in \mathbb{Z}_1^n$$

donde la i indica el robot, $i \in V$, la j indica la coordenada(es decir, en 2D j=1,2 lo que significa que j representa la coordenada x o la coordenada y). $\partial f_{i,j}$ es $\frac{\partial f_{i,j}(\omega_i)}{\partial \omega_i}$, $k_{i,j}$ es la ganancia para cada coordenada $k_{i,j} \in \mathbb{R}^+$ y $\phi_{i,j}$ es el error que comete el robot i en la coordenada j.

Se recuerda que en la implementación realizada **no se ha tenido en cuenta las perturbaciones externas** por lo que la fórmula quedaría de la siguiente manera:

$$u_{i,j} = (-1)^n \partial f_{i,j} - k_{i,j}\phi_{i,j} \quad \forall j \in \mathbb{Z}_1^n \tag{2.4}$$

Es decir, la entrada (o acción) que se realiza en un robot i para una dimensión j es la derivada de la función que representa el comportamiento de dicho robot menos el error cometido en dicha dimensión escalado con una constante.

Por otra parte, el artículo también define la acción de la coordenada virtual de cada robot como:

$$u_i^\omega = (-1)^n + \sum_{j=1}^{n} k_{i,j}\phi_{i,j}\partial f_{i,j} - c_i(\omega_i - \hat{\omega}_i) + \eta_i \tag{2.5}$$

donde $c_i \in \mathbb{R}^+$ y es una ganancia al igual que lo son $k_{i,j}$. La fórmula se trata de un sumatorio de los errores cometidos en cada dimensión proyectados sobre la derivada de la función que define la trayectoria deseada. A dicha fórmula se le añaden dos conceptos importantes nuevos.

En primer lugar, $\hat{w}_i$ es la estimación de la coordenada virtual objetivo($w^*$).Es una coordenada que se va a estar moviendo por el camino deseado como si se tratase de un robot virtual. Este es el que marca el ritmo que tiene que seguir el *platooning*, pero no existe realmente. Es simplemente un objetivo que va a estar dando vueltas en la trayectoria con error 0, $\phi_j^* = 0$.

En segundo lugar, se presenta el concepto de repulsión $\eta_i$. Esto será la repulsión del robot i respecto a su vecindario. Dicha repulsión la representan como:

$$\eta_i = \sum_{k \in N_i} \alpha(|\omega_{i,k}|) \frac{\omega_{i,k}}{|\omega_{i,k}|} \tag{2.6}$$

donde $\frac{\omega_{i,k}}{|\omega_{i,k}|}$ representa la dirección sobre la que actúa la fuerza de repulsión en robot i menos el robot k. Esto se multiplica por un término que es $\alpha(|\omega_{i,k}|)$ y que llamaremos *función de repulsión*, esta función da valores de repulsión dependiendo de la distancia de 2 robots entre ellos y la definen como:

$$\alpha(s) = \begin{cases} \frac{1}{s-r} - \frac{1}{R-r} & \text{si } r < s \leq R \\ 0 & \text{si } s > R \end{cases} \tag{2.7}$$

Es importante destacar que se ha modificado la función y se puede encontrar la explicación del porque y de sus correspondientes cambios en la sección 2.2.

## 2.2. Nueva función de repulsión

En esta sección se va a explicar la nueva función de repulsión que se ha implementado. Antes de nada, cabe destacar, que se llegó a implementar la función de repulsión (2.7) y fue ahí cuando se detectaron las limitaciones que esta tenía para nuestra aplicación.

La función de repulsión que se propone en el artículo **no gestiona el caso en el que la distancia es menor que r**. A nivel teórico, se entiende que no lo ponen porque, según sus suposiciones, esto no va a ocurrir. Sin embargo, cuando se implementa un modelo sobre un entorno real o simulado hay muchos aspectos que pueden influir en el acercamiento de los robots. Es por esto, que se necesita sí o sí tener controlada la repulsión que se ejerce cuando los vehículos están muy cerca.

Se ha mostrado la forma que tiene la función mediante un gráfico de dispersión dándole valores de 0 a 2R para ver que devuelve en cada caso. Esta función se puede apreciar en la figura 2.4. Para representarla se ha tenido que poner un valor concreto para los casos menores que r ya que, si no, como se ha comentado, no se define el valor. Este valor se ha decidido que sea 10. En el eje X, se muestra los valores, transformados a grados, para su mejor entendimiento. Se recuerda que estos valores representan el valor absoluto de la diferencia de $w_i$ y $w_k$, siendo $i$ y $k$ robots vecinos. En el eje Y se muestra la intensidad de la repulsión que devuelve la función dependiendo del acercamiento de los robots.

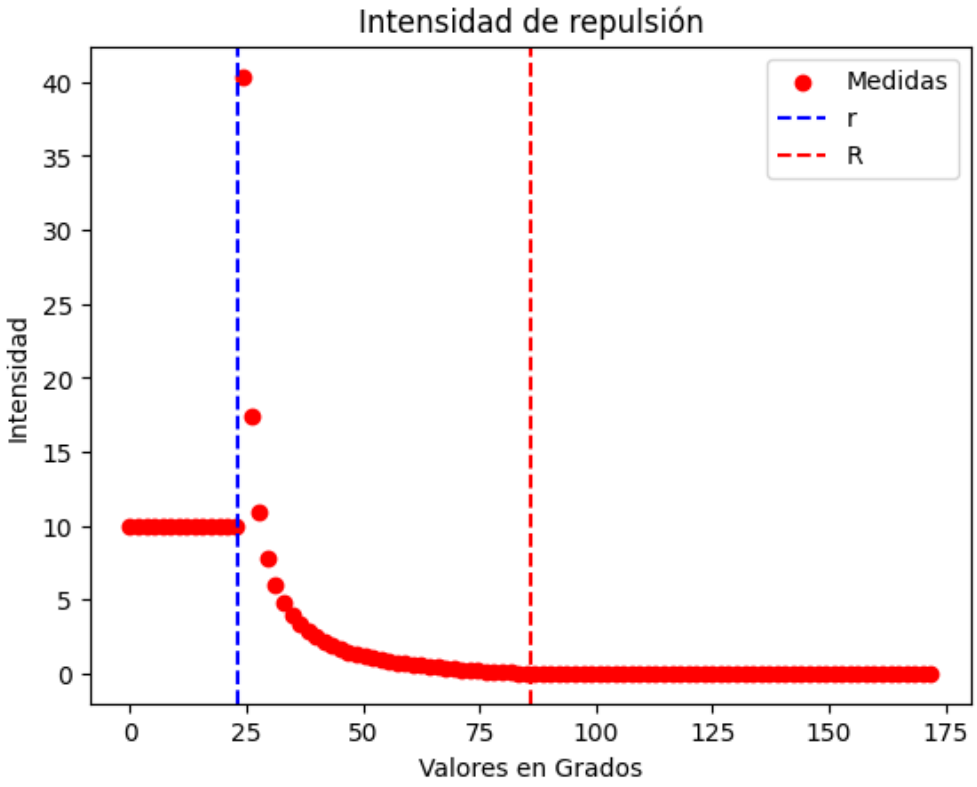



Figura 2.4: Función de repulsión propuesta en el artículo para R=1.5 y r=0.4 radianes.

Asimismo, otro problema que presenta dicha función de repulsión es que **si la distancia entre 2 robots se acerca mucho a r, el término** $1/(s-r)$ **se hace infinito**. Nuevamente, a nivel teórico se entiende que no presente problemas. Sin embargo, a nivel práctico lo que esto produce es un aumento drástico de la repulsión que devuelve unos valores totalmente desorbitados. Estos valores, al ser tan altos, anulaban el resto del comportamiento de los robots y hacían que los robots se moviesen a posiciones no deseadas de golpe.

Dependiendo de la r y R introducidas se puede ver muy bien este efecto. En la figura 2.5 se aprecia perfectamente. El valor producido por la función de repulsión cuando $s$ se acerca a $r$ es de 500, mientras que los valores normales que está dando en el resto de la función no pasan de 10.

En resumen, al no tratar el caso de $s < r$ y además tener esos cambios tan drásticos de valor debido al término $1/(s-r)$ hacía que su funcionalidad a nivel práctico fuese inviable.

Fue por todo esto que se decidió modificar la función para conseguir una **nueva función de repulsión** que tuviese el comportamiento que se buscaba. Para ello se añadió una nueva variable $r_{\text{tope}}$ para evitar este término $1/(s-r)$ y que representa un nuevo mínimo. Es decir, se busca una función que sea realizable (finita siempre).

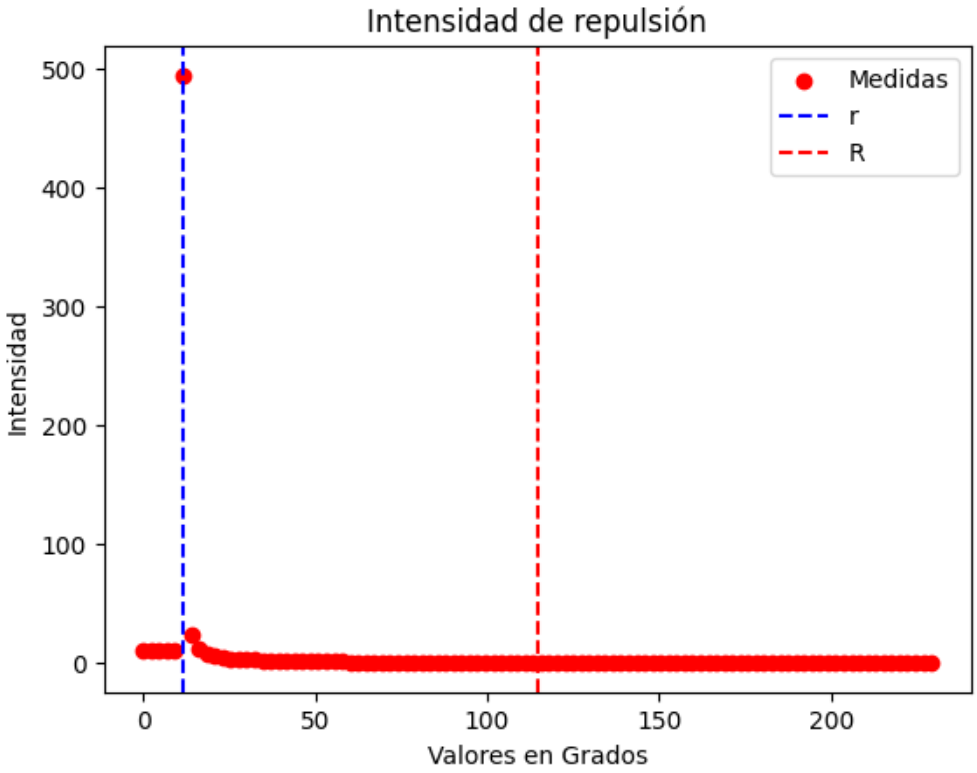



Figura 2.5: Función de repulsión propuesta en el artículo para R=2 y r=0.2 radianes.

Además, debe tener el efecto máximo a una distancia r y no se debe tener en cuenta robots que estén a distancias mayores que R.

De esta manera, la función que se propuso fue la siguiente:

$$\alpha(s) = \begin{cases} \frac{1}{r - r_{tope}} * (r - r_{tope}) & \text{si } r \geq s \\ \frac{R-s}{(s-r_{tope})*(R-r)} * (r - r_{tope}) & \text{si } r < s \leq R \\ 0 & \text{si } s > R \end{cases} \tag{2.8}$$

Así, se representan los 3 casos que buscamos:

1. Caso en el que los robots están muy cerca.
2. Caso en el que los robots están a la distancia que se quiere
3. Caso en el que están muy lejos. En este caso entre ellos no hay fuerza de repulsión.

Si ahora representamos la nueva función con los mismos valores de entrada que antes, se observa un resultado mucho más controlado (ver figura 2.6). Ahora no va a haber valores muy altos que produzcan resultados inesperados, sino que se va a empezar desde un valor de repulsión, para cuando estén muy cerca, que depende de r y $r_{tope}$ e irá decreciendo ese valor conforme se alejen los vehículos. Además, el término $r - r_{tope}$ que está multiplicando en ambos casos, sirve para normalizar los valores de intensidad entre 0 y 1 y conseguir un resultado todavía más controlado.

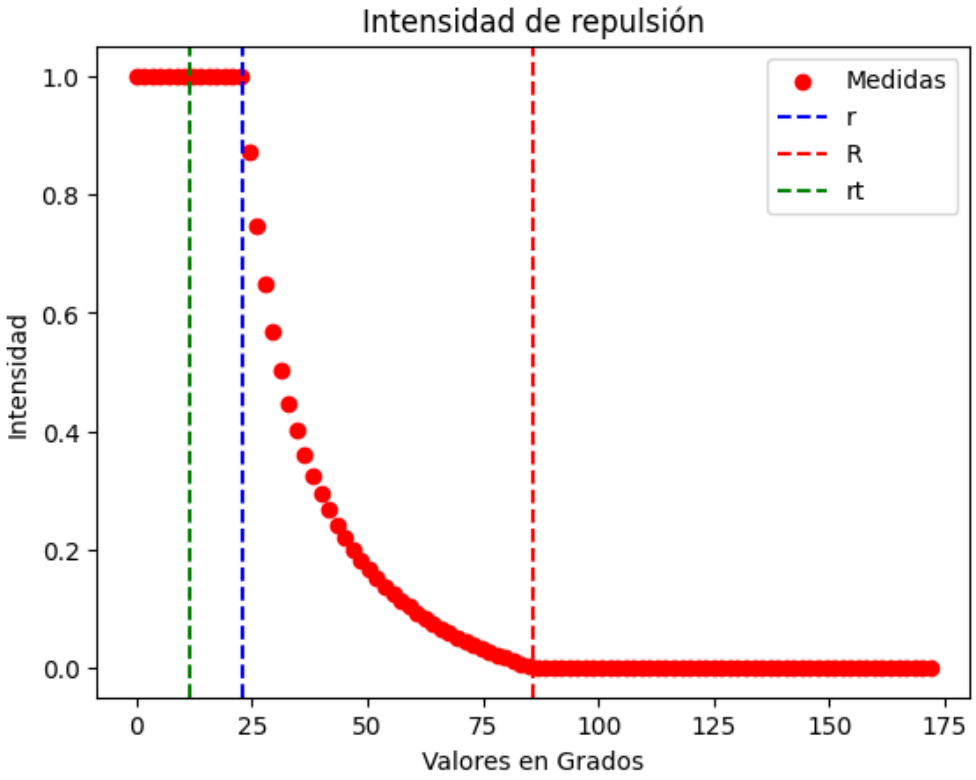



Figura 2.6: Función de repulsión nueva para R=1.5, r=0.4, $r_{tope}$ = 0.2 radianes.

## 2.3. Casos concretos de aplicación

Una vez entendida toda la teoría y vistos las modificaciones correspondientes sobre la misma, se va a pasar a explicar como se han obtenido las ecuaciones para los casos concretos de aplicación que proponen en el artículo. Se han llevado a cabo los casos en 2 dimensiones de la trayectoria *Circular* y de la trayectoria en forma de ocho que denominan como *Lissajous*.

### 2.3.1. Adaptación de las ecuaciones para trayectoria *Circular*

Para obtener las ecuaciones que se implementarán en el código realizado más adelante, se parte de la ecuación de *single integrator* que veíamos en 2.2 y que recordamos a continuación (se recuerda que no se tiene en cuenta las perturbaciones externas para el trabajo realizado):

$$\dot{x}_i = u_i, \quad i \in V$$

Esta ecuación significa que la nueva posición de un robot depende únicamente de su entrada. De esta manera se puede definir la entrada o acción para un tiempo $t$ como:

$$\frac{x_{i,1}(t+1) - x_{i,1}(t)}{T} = u_{i,1}(t) \tag{2.9}$$

Esto es, la acción para un tiempo t es igual a la diferencia de posiciones del robot dividido para un periodo. Este periodo $T$ se elige de manera arbitraria.

Partiendo de esta base, el artículo nos da los siguientes datos para la trayectoria *Circular*:

$$\begin{cases} x_{i,1} = 800 \cdot \cos(\omega_i) \\ x_{i,2} = 800 \cdot \sin(\omega_i) \\ k_{i,1} = 3{,}5, \quad k_{i,2} = 3{,}5 \\ c_i = 2 \end{cases} \tag{2.10}$$

Con todo esto, es hora de sacar las acciones para cada coordenada según las fórmulas (2.4) y (2.5). Se recuerdan estas fórmulas a continuación:

$$u_{i,j} = (-1)^n \partial f_{i,j} - k_{i,j}\phi_{i,j} \quad \forall j \in \mathbb{Z}_1^n$$

$$u_i^\omega = (-1)^n + \sum_{j=1}^{n} k_{i,j}\phi_{i,j}\partial f_{i,j} - c_i(\omega_i - \hat{\omega}_i) + \eta_i$$

Para ellas se necesita la derivada $\partial f_{i,j}$ y el error $\phi_{i,j}$ que se obtienen de la siguiente manera:

$$\partial f_{i,1} = \frac{\partial f_{i,1}}{\partial w_i} = \frac{\partial x_{i,1}}{\partial w_i} = -800 * \sin w_i$$

$$\partial f_{i,2} = \frac{\partial f_{i,2}}{\partial w_i} = \frac{\partial x_{i,2}}{\partial w_i} = 800 * \cos w_i$$

$$\phi_{i,1} = x_{i,1}(t) - 800 \cdot \cos(\omega_i)$$

$$\phi_{i,2} = x_{i,2}(t) - 800 \cdot \sin(\omega_i)$$

Por lo tanto las acciones quedan como:

$$u_{i,1}(t) = (-1)^n \cdot (-800 \cdot \sin w_i) - k_{i,1} \cdot x_{i,1}(t) + k_{i,1} \cdot 800 \cdot \cos w_i \tag{2.11}$$

$$u_{i,2}(t) = (-1)^n \cdot (800 \cdot \cos w_i) - k_{i,2} \cdot x_{i,2}(t) + k_{i,2} \cdot 800 \cdot \sin w_i \tag{2.12}$$

$$\begin{aligned} u_i^\omega(t) = {} & (-1)^n + k_{i,1} \cdot (x_{i,1}(t) - 800 \cdot \cos(\omega_i(t)) \cdot (-800 * \sin w_i(t)) \\ & + k_{i,2} \cdot (x_{i,2}(t) - 800 \cdot \sin(\omega_i(t)) \cdot (800 * \cos w_i(t)) \\ & - c_i(\omega_i - \hat{\omega}_i) + \sum_{k \in N_i} \alpha(|\omega_{i,k}|) \frac{\omega_{i,k}}{|\omega_{i,k}|} \end{aligned} \tag{2.13}$$

### 2.3.2. Adaptación de las ecuaciones para trayectoria *Lissajous*

Para obtener las acciones que se implementarán en el código hay que seguir la misma lógica que se ha seguido en el apartado anterior 2.3.1. Sin embargo, esta vez el artículo nos da otros datos como funciones de $x$, $y$ y como constantes:

$$\begin{cases} x_{i,1} = \frac{800 \cdot \cos(\omega_i)}{1+0{,}3 \cdot (\sin \omega_i)^2} \\ x_{i,2} = \frac{800 \cdot \sin(\omega_i) \cdot \cos(\omega_i)}{1+0{,}3 \cdot (\sin \omega_i)^2} \\ k_{i,1} = 2, \quad k_{i,2} = 2 \\ c_i = 2 \end{cases} \tag{2.14}$$

Por lo tanto las derivadas y errores ahora son distintos también. Para realizar las derivadas se utiliza la regla del cociente $u'(x) = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{[g(x)]^2}$

$$\partial f_{i,1} = \frac{\partial f_{i,1}}{\partial w_i} = \frac{\partial x_{i,1}}{\partial w_i} = \frac{800 \cdot (-\sin \omega_i (1 + 0{,}3 \cdot (\sin \omega_i)^2) - 0{,}6 \cdot (\cos \omega_i)^2 \cdot \sin \omega_i)}{(1 + 0{,}3 \cdot (\sin \omega_i)^2)^2}$$

$$\partial f_{i,2} = \frac{\partial f_{i,2}}{\partial w_i} = \frac{\partial x_{i,1}}{\partial w_i} = \frac{800 \cdot (\cos(2\omega_i)(1 + 0{,}3 \cdot (\sin \omega_i)^2) - 0{,}6 \cdot (\cos \omega_i)^2 \cdot (\sin \omega_i)^2)}{(1 + 0{,}3 \cdot (\sin \omega_i)^2)^2}$$

$$\phi_{i,1} = x_{i,1}(t) - \frac{800 \cdot \cos(\omega_i)}{1 + 0{,}3 \cdot (\sin \omega_i)^2}$$

$$\phi_{i,2} = x_{i,2}(t) - \frac{800 \cdot \sin(\omega_i) \cdot \cos(\omega_i)}{1 + 0{,}3 \cdot (\sin \omega_i)^2}$$

De esta manera, las acciones se vuelven a obtener según las fórmulas (2.4) y (2.5).

$$u_{i,1}(t) = (-1)^n * \frac{800 \cdot (-\sin \omega_i (1 + 0{,}3 \cdot (\sin \omega_i)^2) - 0{,}6 \cdot (\cos \omega_i)^2 \cdot \sin \omega_i)}{(1 + 0{,}3 \cdot (\sin \omega_i)^2)^2} \tag{2.15}$$

$$- k_{i,1} \cdot x_{i,1}(t) + k_{i,1} \cdot \frac{800 \cdot \cos(\omega_i)}{1 + 0{,}3 \cdot (\sin \omega_i)^2} \tag{2.16}$$

$$u_{i,2}(t) = (-1)^n * \frac{800 \cdot (\cos(2\omega_i)(1 + 0{,}3 \cdot (\sin \omega_i)^2) - 0{,}6 \cdot (\cos \omega_i)^2 \cdot (\sin \omega_i)^2)}{(1 + 0{,}3 \cdot (\sin \omega_i)^2)^2}$$

$$- k_{i,2} \cdot x_{i,2}(t) + k_{i,2} \cdot \frac{800 \cdot \sin(\omega_i) \cdot \cos(\omega_i)}{1 + 0{,}3 \cdot (\sin \omega_i)^2} \tag{2.17}$$

La acción de la coordenada virtual se abreviará sin poner el resultado de las derivadas. En su lugar, se pondrá $\partial f_{i,1}$ y $\partial f_{i,2}$ para representarlas. De esta manera se consigue una mejor lectura de la fórmula, ya que si no queda muy compleja.

$$
\begin{aligned}
u_i^{\omega}(t) = (-1)^n &+ k_{i,1} \cdot \left( x_{i,1}(t) - \frac{800 \cdot \cos(\omega_i)}{1 + 0{,}3 \cdot (\sin \omega_i)^2} \right) \cdot \partial f_{i,1} \\
&+ k_{i,2} \cdot \left( x_{i,2}(t) - \frac{800 \cdot \sin(\omega_i) \cdot \cos(\omega_i)}{1 + 0{,}3 \cdot (\sin \omega_i)^2} \right) \cdot \partial f_{i,2} \\
&- c_i(\omega_i - \hat{\omega}_i) + \sum_{k \in N_i} \alpha(|\omega_{i,k}|) \frac{\omega_{i,k}}{|\omega_{i,k}|}
\end{aligned} \tag{2.18}
$$

Por último, cabe destacar que en ambas trayectorias (tanto *Circular* como *Lissajous*) se utiliza la fórmula (2.9) para actualizar las coordenadas $x$, $y$ y $w$ de los robots.

Estas fórmulas son las que se implementarán como se indica en el resto de capítulos (ver 5.2) para llevar a cabo el las simulaciones y experimentos del *platooning*.

# Capítulo 3

# Herramientas y trabajo previo a la implementación

En el capítulo que se presenta, se va a hacer un resumen del trabajo previo que se realizó. Como se trata de una explicación bastante extensa, se referencia al lector al Anexo B para entender bien los detalles de todo el trabajo que se llevó a cabo.

Se utilizó ROS para la realización de este trabajo, concretamente ROS Noetic. Para aprenderlo se realizaron 2 prácticas que correspondían a las prácticas 1 y 2 de la asignatura *Autonomus Robots* en el *Máster Universitario en Robótica, Gráficos y Visión por Computador* de UNIZAR.

La primera era una parte más teórica de ROS: explicaba conceptos como los nodos, los topics y los comandos. La segunda proponía la realización de un algoritmo para conseguir mover un robot a ciertas posiciones. Esta práctica requirió el desarrollo de una implementación que se explica detalladamente en el Anexo B.

Por último, se estudió el sensor de percepción que se iba a utilizar, el láser. Para ello se entendió correctamente los datos que devolvía mediante la realización de una detección básica de obstáculos.

# Capítulo 4

# Arquitectura del sistema

Antes de comenzar con la implementación del algoritmo, hay que tener claro el sistema que se ha implementado. Esto consiste en conocer las partes que forman el sistema, qué nodos en ROS se han creado, qué mensajes hay y cómo se comunican entre ellos.

En este capítulo se entrará en detalle en todos estos aspectos, mostrando imágenes y diagramas que van a ayudar a entender la estructura y el funcionamiento del sistema que se ha desarrollado. Se recuerda al lector que el código desarrollado se encuentra en el repositorio de GitHub [3].

## 4.1. Definición de nodos y estructura del sistema

Para comenzar, se tiene que hablar de la parte más importante de la arquitectura del sistema: los nodos (o módulos) que la componen. Estos son:

- moveRobot: es el nodo de movimiento del robot, implementa toda la lógica de movimiento hacia una meta, la detección de obstáculos y la comunicación por su parte con los demás.

- circular2DTrajectory: este nodo es el calculador de la trayectoria *Circular*. Se encarga de calcular los puntos correspondientes para realizar el *platooning* en una trayectoria *Circular* en 2D. Tiene 2 modos de trabajo. El modo "DEBUG" calcula todos los puntos para I iteraciones (sirve para mirar numéricamente ciertos aspectos) sin comunicación con el nodo de movimiento y el modo "RUN" se comunica con el nodo de movimiento (*moveRobot*) pasándole cuando requiera un nuevo punto (meta).

- lissajous2DTrajectory: sigue exactamente la misma lógica que el nodo de la trayectoria *Circular*. Es decir, es un calculador de la trayectoria *Lissajous* que

puede comunicarse con el *moveRobot* o simular para I iteraciones lo que hacen los robots numéricamente.

– generarGraficos: módulo que genera ciertos gráficos para entender diversos aspectos de la trayectoria *Circular*.

– generarGraficosLis: módulo que genera ciertos gráficos para entender diversos aspectos de la trayectoria *Lissajous*.

Una vez vistos los nodos más importantes, se va a plasmar mediante un diagrama de paquetes como se han organizado y distribuido en ROS. Este diagrama se puede apreciar en la figura 4.1.

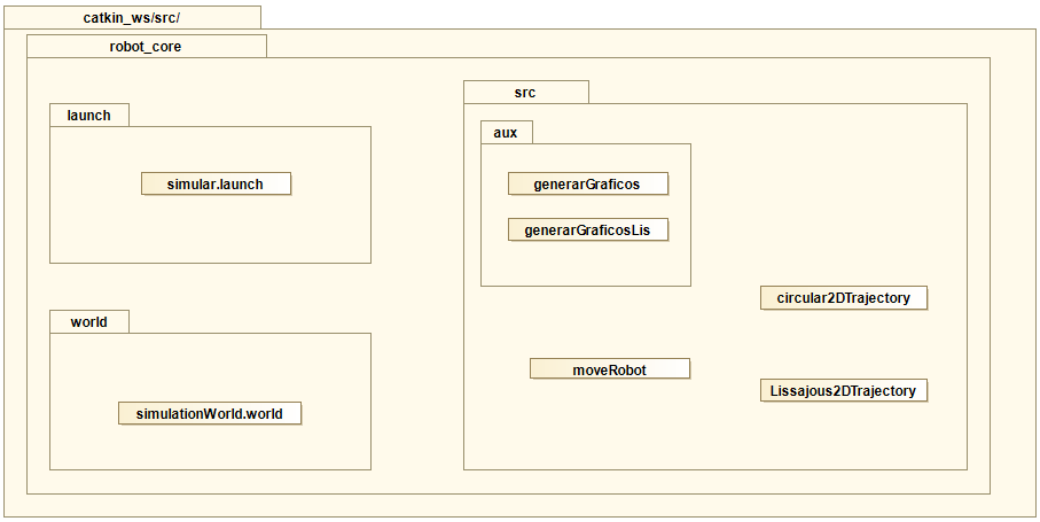



Figura 4.1: Diagrama de paquetes del sistema.

Como se observa en dicho diagrama, se está trabajando en un entorno de ROS que se ha llamado *catkin_ws* . Este entorno tiene un paquete llamado *robot_core* donde se han implementado todos los nodos comentados anteriormente. Se encuentra una carpeta *src* que contiene todo el código. En dicha carpeta encontramos los nodos de lógica y movimiento mientras que los nodos de generación de gráficos se encuentran en un nivel más de profundidad (directorio *aux*).

Asimismo, en *robot_core* se encuentran dos paquetes que no se han comentado todavía. El primero se trata de el módulo *launch*, que contiene un fichero ".launch" para lanzar la simulación. Este fichero se explicará mejor más adelante(ver sección 6.1). Por otra parte, el paquete *world* que contiene el mundo que se utiliza en la simulación.

Entrando más en detalle con el paquete *src*, se puede crear un diagrama de clases (fig 4.2) con los nodos que hemos visto y sus relaciones. Como el código es complejo

y los nodos tienen muchas variables y funciones, en el diagrama se van a poner las fundamentales de cara a explicar la arquitectura del sistema.

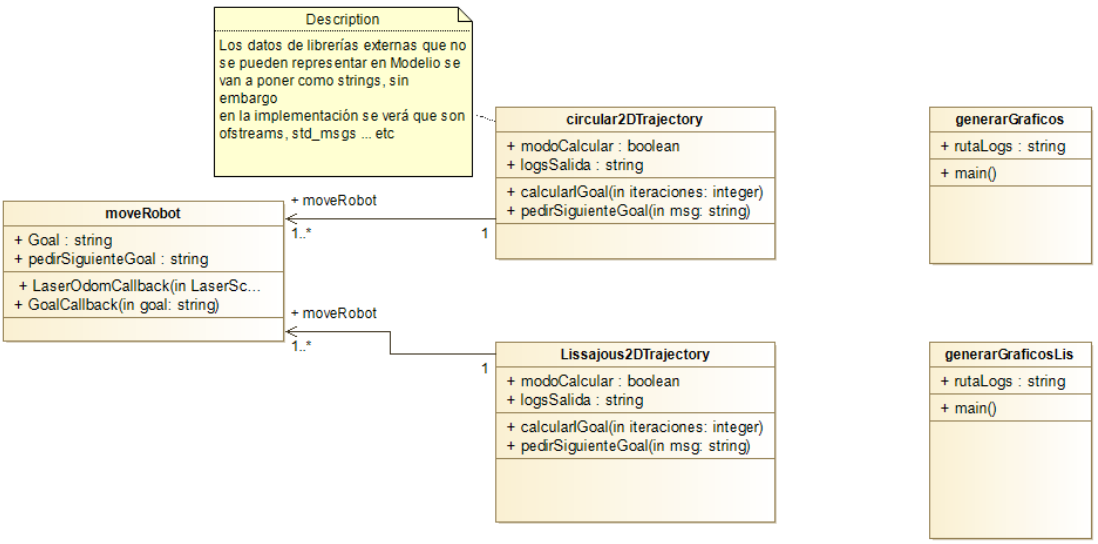



Figura 4.2: Diagrama de clases del paquete *src*.

En este diagrama de clases se pueden apreciar varios aspectos importantes del sistema:

1. Como se ha comentado antes, **los calculadores de trayectorias tienen 2 modos**. Si se decide utilizar el modo "DEBUG", se ejecuta el método calcularIGoal donde se le pasan un número de iteraciones y ejecuta numéricamente la trayectoria que seguiría el *platooning* en esas iteraciones.

   Si se decide utilizar el modo "RUN" se pone en modo comunicación ejecutando pedirSiguienteGoal. La comunicación exacta se explicará en las siguientes secciones.

2. Las clases de los calculadores siempre son 1 sola, pero tienen relación 1 a N con el movimiento del robot, es decir, depende de los robots que haya habrá más nodos instanciadaos de *moveRobot* pero el calculador siempre será el mismo. De esta manera el calculador puede soportar N nodos cualesquiera de movimiento de robots.

3. Si se decide ejecutar el cálculo de trayectoria para I iteraciones, el nodo calculador crea varios ficheros de logs con la información necesaria para que posteriormente se puedan mostrar varios gráficos importantes. Estos gráficos mostrarán más o menos el resultado que se puede obtener si se simula.

Por último, cabe destacar que el código perteneciente al algoritmo se ha escrito en C++. Sin embargo, los generadores de gráficos están escritos en Python por simplicidad del uso de la librería *matplotlib*.

## 4.2. Comunicación entre nodos

### 4.2.1. Mensajes y topics definidos

Se han definido 3 tipos de topics para la comunicación entre el nodo calculador de la trayectoria y el nodo de movimiento. Antes de nada, cabe destacar que la comunicación entre el nodo de movimiento y ambos calculadores (*circularTrajectory* y *LissajousTrajectory*) es idéntica.

Estos 3 tipos de mensajes son los siguientes:

| Topic | Tipo de mensaje | Dirección |
|---|---|---|
| /goal | geometry_msgs PoseStamped | Circular o Lissajous → moveRobot |
| /pedirSiguienteGoal | String | moveRobot → Circular o Lissajous |
| /hayObstaculo | String | moveRobot → Circular o Lissajous |

Tabla 4.1: Topics definidos para la comunicación entre el nodo de movimiento y el calculador de una trayectoria.

Estos mensajes tienen la siguiente funcionalidad:

- **/Goal**: es un canal por el que se pasa un tipo de dato geometry_msgs PoseStamped, es decir, se pasa una posición. Por este canal, se van a pasar las metas a las que tiene que ir un robot calculadas por el calculador de trayectoria.

- **/pedirSiguienteGoal**: es un canal por el que se pasa un mensaje de que un robot ha llegado a su meta con su odometría en ese momento. Se pasa un tipo de dato *String* que utiliza el siguiente formato:

  $$id\_robot, x, y$$

  De esta manera, se sabe que robot ha llegado a su meta y con que odometría exacta. Así si se requiriese se podría corregir la odometría en caso de haber un error.

- **/hayObstaculo**: este canal se utiliza para que el robot le mande un mensaje al calculador si ha encontrado un obstáculo. Simplemente le manda un *String* con su id para que el calculador sepa que robot ha sido. Como se verá en la implementación del movimiento (ver sección 5.1) la detección de obstáculos

simplemente para los vehículos un determinado tiempo por lo que no haría falta pasar este mensaje al calculador. Sin embargo, se ha decidido dejarlo implementado, por si se prueban otras variantes como dar distintas metas a los robots cuando detecten obstáculos.

Cabe destacar que realmente en la práctica, hay varios topics de /goal y /pedirSiguienteGoal, concretamente, uno por robot. De esta manera, como el calculador es una única instancia, escucha a diferentes topics para tener comunicación directa y aislada con cada robot. Mientras tanto, el robot al ser un nodo generalizado que se instancia varias veces, se le asigna un id de robot y configura sus comunicaciones para escuchar y publicar en los topics correspondientes.

Por ejemplo, si yo tengo 3 robots realizando una trayectoria *Circular*, hay 1 nodo instanciado como circular2DTrajectory y 3 nodos instanciados como moveRobot cada uno con un id de robot 0, 1 y 2 respectivamente. De esta forma, los topics que se crearían serían los que se pueden ver en la siguiente figura 4.3. En esta figura aparecen más topics

```
/clock
/hayObstaculo
/robot_0/base_pose_ground_truth
/robot_0/base_scan
/robot_0/cmd_vel
/robot_0/goal
/robot_0/odom
/robot_0/pedirSiguienteGoal
/robot_1/base_pose_ground_truth
/robot_1/base_scan
/robot_1/cmd_vel
/robot_1/goal
/robot_1/odom
/robot_1/pedirSiguienteGoal
/robot_2/base_pose_ground_truth
/robot_2/base_scan
/robot_2/cmd_vel
/robot_2/goal
/robot_2/odom
/robot_2/pedirSiguienteGoal
```

Figura 4.3: Topics que se definen en trayectoria *Circular* de 3 robots.

que los que hemos comentado, estos son importantes a la hora de que el robot funcione en la simulación(por ejemplo cmd_vel sirve para introducirle velocidad al robot, odom

es su odometría...). Sin embargo, como no son relevantes para la comunicación entre los nodos no se va a profundizar más en ellos.

En cuanto a los *topics* que sí hemos explicado se ve como se crean /goal y /pedirSiguienteGoal para cada robot. Si el calculador le quisiese pasar su meta al robot 1 se lo pasaría por robot_1/goal mientras que el robot 1 le pediría su siguiente meta mediante robot_1/pedirSiguienteGoal.

A continuación (fig 4.4) se puede ver un diagrama de secuencia que muestra la comunicación que se ha ido explicando con el ejemplo de 3 robots en una trayectoria *Circular*.

Como se aprecia, la comunicación es un bucle en el que cada robot le pide una meta mediante pedirSiguienteGoal y cuando el calculador ha recibido todas (quiere decir que todos los robots ya han llegado a su meta actual) calcula las nuevas metas y las manda. Si el calculador ha recibido menos de 3 metas no manda nada, cuando reciba la tercera es cuando calcula las metas nuevas. Así se asegura de que todos los robots van sincronizados.

Si durante el movimiento los robots detectan un obstáculo, se lo hacen saber al calculador mandando un mensaje, como se ha explicado anteriormente, por el topic *hayObstaculo*.

Cabe destacar que las funciones calcularMetas e irAMeta del diagrama de secuencia no se llama exactamente así en el código implementado. Se han puesto con esos nombres para mayor claridad del diagrama.

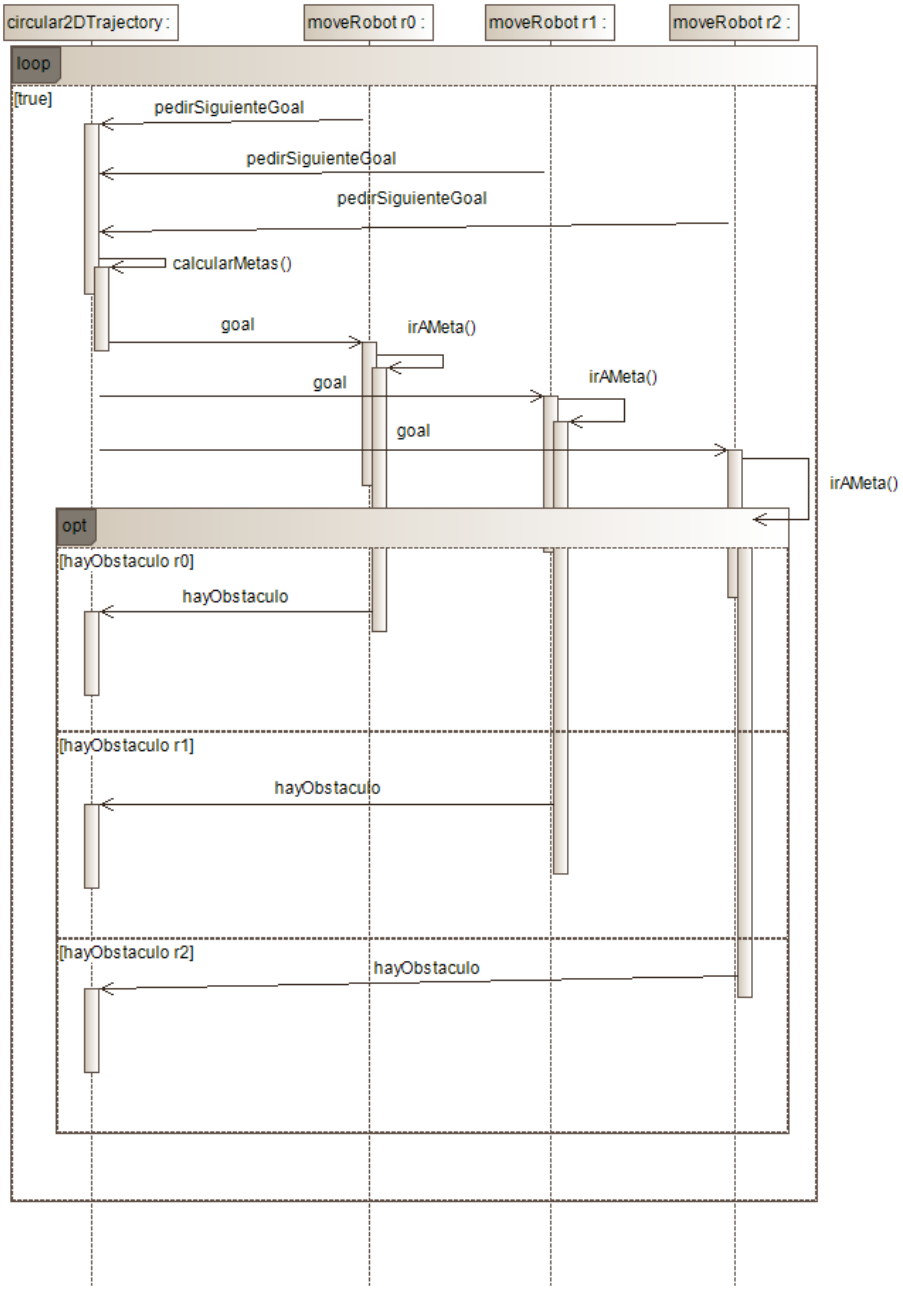



Figura 4.4: Diagrama de secuencia de 3 robots en trayectoria *Circular*.

# Capítulo 5

# Definición del algoritmo e implementación

Se ha visto ya los apartados correspondientes al análisis matemático y a la arquitectura del sistema que se ha implementado. En ellos, se explicaban aspectos muy importantes a la hora de implementar el *platooning*. Sin embargo, no se explicaba la implementación del *platooning* como tal.

En este capítulo veremos como se ha realizado dicha implementación. Primeramente se explicará como se ha implementado el nodo de movimiento de los robots basándose en el código que ya se desarrolló para los tutoriales (ver sección B.1). Posteriormente, se verá como se implementaron las trayectorias circulares y *Lissajous* 2D que aparecían en el artículo [5] y que tanto se han comentado hasta ahora.

## 5.1. Nodo de movimiento

En primer lugar, como se ha comentado, se va a explicar la implementación del nodo de movimiento. Este se ha realizado en un fichero C++ que se llama *moveRobot* y que se explica en la arquitectura del sistema 4.1.

Este nodo es el encargado de realizar el movimiento de los robots hasta una meta dada. Como ya se sabe, se comunica con los calculadores de metas (bien *Circular* o bien *Lissajous*) y éstos le van pasando metas conforme las requiera. De esta manera el nodo va hacia esas metas.

Asimismo, implementa una detección de obstáculos básica mediante el láser que se explica en la sección de trabajo previo a la implementación B.2.

### 5.1.1. Lógica de movimiento

La parte más importante del nodo de movimiento de los robots es el cómo se mueven, es decir, qué lógica siguen para realizar el movimiento. Para ello, hay que entender que

se ha implementado una lógica *go_to_goal*.

Este término se refiere a ir hacia una meta, es decir, el robot sabe donde está él y sabe a donde tiene que llegar. Para ello, hace diferentes operaciones que le llevan hasta dicha meta. Estas operaciones las realiza cada vez que tiene información sobre el láser y sobre la odometría, ya que son los datos que utiliza. El robot espera a tener las dos informaciones disponibles y cuando las tiene comienza a realizar el movimiento.

Este *go_to_goal* se ha implementado basándose en el ángulo del robot siguiendo los siguiente pasos:

- Se conoce la odometría del robot (posición x, posición y, posición z, orientación w). Como es una trayectoria en 2D no se va a necesitar la posición z.
- Se conoce la meta a la que hay que llegar (posición x, posición y).
- Se calcula la diferencia de coordenadas entre la meta y el robot. A estas diferencias las llamamos *ex* y *ey*.
- Se calcula la orientación que debe tener el robot para ir a la meta mediante la función atan2.
- El robot gira sobre si mismo hasta que consigue esta orientación. Este giro se ha adaptado para hacer de manera "inteligente". Gira a la derecha o a la izquierda depende cual sea el camino mas corto.
- El robot va hacia delante una vez tiene la orientación correcta hasta llegar a la meta. En este momento, utiliza el láser para detectar obstáculos.

Una vez entendidos los pasos que hace, se va a entrar en detalle en cada uno de ellos.

**En cuanto a calcular la orientación del ángulo,** esto se hace, como se ha visto, mediante la diferencia de posiciones en x y posiciones en y de la meta y el robot. Esto se debe a que la operación arcotangente de dos parámetros (atan2) devuelve el ángulo entre el eje positivo de las x y la recta formada desde origen (0,0) hasta el punto (x,y). Este ángulo siempre es el mismo que el ángulo que hay entre el robot y la meta. Dicho efecto se puede observar perfectamente en un ejemplo (figura 5.1) hecho con la herramienta GeoGebra [2].

En este ejemplo, vamos a imaginar que el robot está en la posición (1,1) y quiere ir a la meta (6,3). La diferencia de posiciones es:

$$ex = 6 - 1 = 5$$

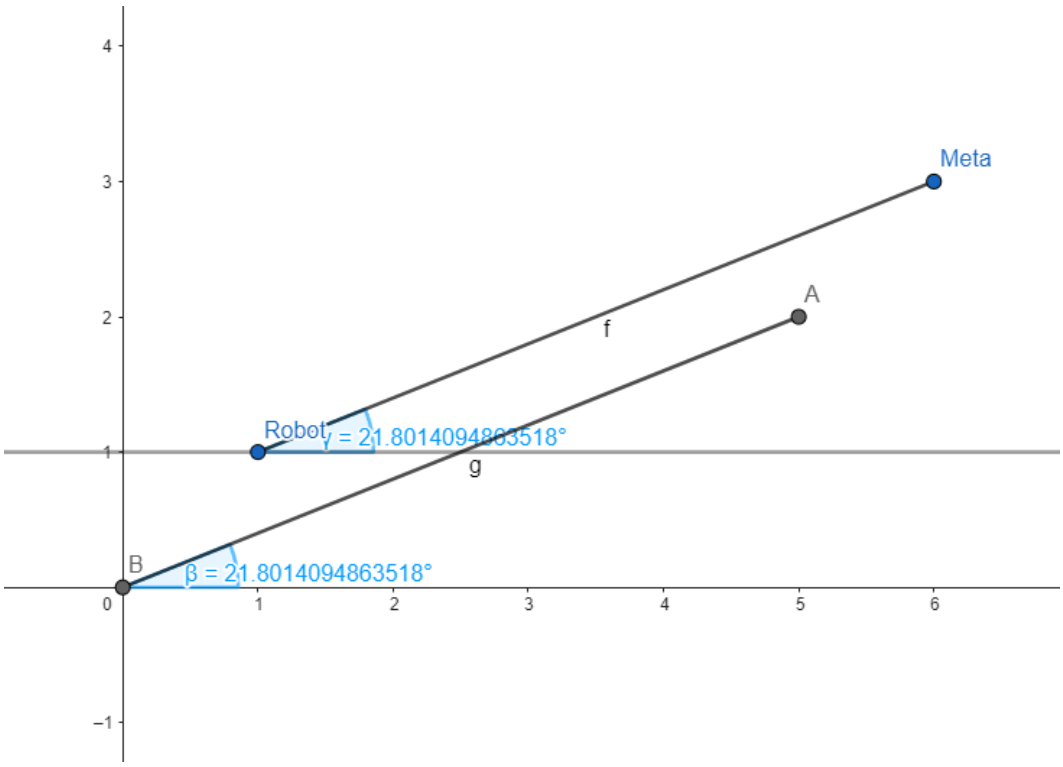



Figura 5.1: Utilización de la función atan2.

$$ey = 3 - 1 = 2$$

Este punto (5,2) es el punto A en el gráfico. Como se observa, el ángulo que calcula atan2 $\beta = 21{,}801°$ es el mismo que el que hay entre el robot y la meta $\gamma = 21{,}801°$

Una vez que el robot sabe la orientación correcta hacia la meta, **empieza a girar sobre sí mismo** para conseguirla. Esto se hace restando la orientación del robot menos la de la meta y aplicando únicamente velocidad angular si todavía no es la misma. El robot gira de manera "inteligente", es decir, mira la diferencia de orientaciones y así sabe si tiene que girar hacia la izquierda o hacia la derecha depende de donde quede más cerca. Se trata de un añadido que se puso al esqueleto de movimiento de los tutoriales ya que si no perdía demasiado tiempo siempre girando hacia el mismo lado.

Cuando consigue la orientación correcta **empieza a ir hacia delante**, aquí es donde activa la detección de obstáculos de la que hablaremos posteriormente. El ir hacia delante es muy sencillo, simplemente es quitarle la velocidad angular que tenía el robot y darle velocidad lineal. A continuación se añade, con el fin de entender mejor toda esta lógica, una parte del código en C++ que implementa todo esto.

```
    //Calcular la orientacion utilizando atan2
    float ex = Goal.pose.position.x - estimate_pose->pose.pose.position.x;
    float ey = Goal.pose.position.y - estimate_pose->pose.pose.position.y;
    float beta = atan2(ey, ex);

    //Obtener la orientacion actual del robot
    float current_yaw = tf::getYaw(q);

    float alpha = beta- current_yaw;

    //Asegurarse que alpha esta entre -pi y pi
    if (alpha > M_PI) {
        alpha -= 2 * M_PI;
    } else if (alpha < -M_PI) {
        alpha += 2 * M_PI;
    }
    geometry_msgs::Twist input;

    float tolerance = 0.1;  // Tolerancia para detener el giro
                            //cuando este cerca del objetivo
    //Alpha indica si el objetivo esta a la izquierda o a la derecha
    if (alpha > tolerance) {
        //Gira a la izquierda
        //Asigna una velocidad angular positiva para girar a la izquierda
        input.linear.x = 0;
        input.angular.z = 0.5;
    } else if (alpha < -tolerance) {
        //Gira a la derecha
        //Asigna una velocidad angular negativa para girar a la derecha
        input.linear.x = 0;
        input.angular.z = -0.5;
    } else {
        input.linear.x = 0.5;
        input.angular.z = 0;
```

Como se observa, *alpha* es la diferencia entre orientaciones de la que se hablaba. Hay que asegurarse de que está entre $-\pi$ y $\pi$.

Posteriormente se va a comparar con un valor que se llama de *tolerancia*. Este valor es el umbral que se acepta como que la orientación del robot está suficientemente cerca de la orientación deseada. Se utiliza un valor de tolerancia ya que el robot calcula esto cada vez que coordina los datos de la odometría y del láser. Por lo tanto, es difícil que si está girando de mientras justo se obtenga el valor 0 (ambas orientaciones son iguales). Es por esto que se acepta un valor parecido a 0 (menor que 0.1 radianes) como suficientemente cerca.

### 5.1.2. Detección de obstáculos

La detección de obstáculos funciona en el momento en el que el robot se mueve hacia la meta. Como se ha explicado en la sección anterior, primero el robot busca la orientación correcta y luego se mueve. Además, esta detección está basada en la

detección que se desarrolla en los tutoriales B.2 para aprender sobre ROS y el láser.

La detección es muy sencilla, si recordamos como funciona el láser, este lanza muchos rayos en un rango que va desde *angle_min* a *angle_max*. Lo que se ha cambiado respecto a la detección del tutorial es lo siguiente: ahora se miran todos los rayos, si alguno devuelve un valor muy cercano (menor que un valor que se pone de umbral), se deduce que hay un obstáculo y se para el robot (se le asignan velocidades 0).

Al ser el campo de visión del robot 180°ahora detecta mucho mejor los obstáculos, ya que en la versión primitiva del tutorial solo se miraban los 3 rayos centrales. En la figura 5.2 se puede observar una ilustración con valores arbitrarios de lo que podía ser la diferencia antes versus ahora. En esta figura se observa una ilustración donde

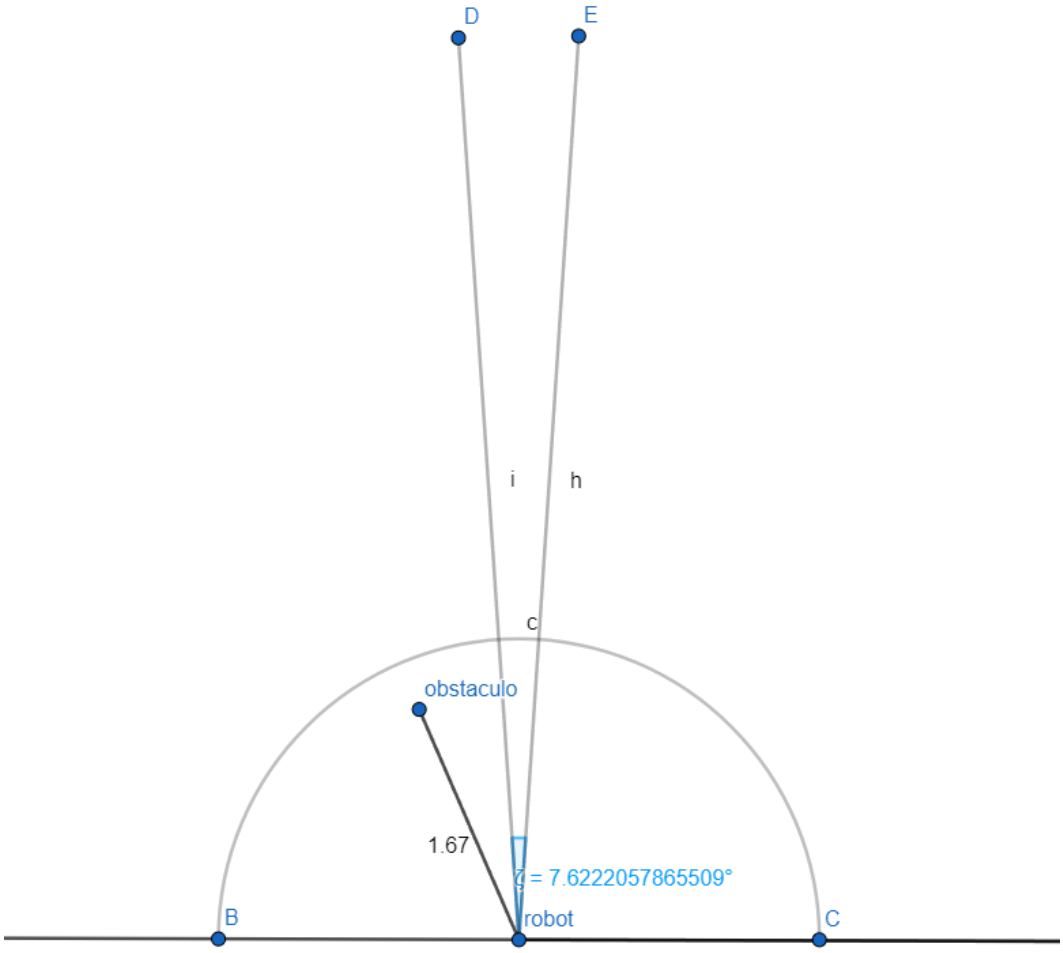



Figura 5.2: Ilustración de lo que detectaba antes y lo que detecta ahora el láser.

las rectas i y h representan 2 de los 3 rayos centrales, concretamente los extremos. De esta manera, antes detectaba únicamente obstáculos en el ángulo que estos forman. Sin

embargo, ahora se miran todos los rayos que lanza el láser (lanza en 180°) por lo que si ahora se tiene el obstáculo que se aprecia en la ilustración si lo detectaría.

Se repite que los valores de esta figura no son reales, simplemente se ha añadido para entender mejor la magnitud de lo que antes detectaba y ahora detecta.

Para ilustrar el comportamiento explicado, se añade un vídeo [19] donde se aprecia perfectamente como detectaba el láser antes y la detección que hace ahora.

Tal y como se ha explicado en la figura 5.2 se ve que la detección que se hacía antes pasaba por el obstáculo que se le pone sin problema ya que este obstáculo no está justo delante sino en un lateral. A priori, esto parecería un comportamiento correcto ya que el robot no va a chocar con el obstáculo. Sin embargo, en el contexto en el que se está trabajando, al ser un *platooning* de robots y al tener todos los vehículos movimiento se ha decidido implementarlo como se ha explicado.

De esta forma, aunque el robot pare si el obstáculo no esta estrictamente enfrente suyo, se evitan bastantes choques laterales y comportamientos no deseados.

### 5.1.3. Generalización para N robots

Una vez implementado y entendido el movimiento de los robots, se quiere replicar esto para N robots puesto que en el *platooning* tenemos varios. Para ello se ha tenido que implementar una manera de poder tener N robots con la misma lógica de movimiento.

Como bien se sabe ya, toda la lógica de movimiento se ha desarrollado en el fichero *moveRobot.cpp* por lo cual, se ha adaptado este para poder lanzar N nodos similares. Esto se ha hecho basándose en las siguientes ideas:

- Se necesita un id de robot para cada uno.
- Se necesita que el robot escuche por sus topics correspondientes según su id, como se explico en el capítulo de arquitectura 4.1.
- Se necesita vincular que ROS sepa qué robot es cada uno en la simulación.

Para conseguir esto se ha añadido un parámetro que se le pasa al programa y que tiene que ser un entero. Este parámetro indica el id del robot que estamos lanzando. Como el lanzamiento de la simulación se hace mediante un fichero .launch que lo automatiza (este se explicará en el capítulo de simulación 6.1) se ha adaptado nuestro código para que funcione con este fichero. En el fichero de lanzamiento se le pasa a cada nodo que se quiere instanciar su id correspondiente(desde 0 a N). De esta manera, el propio ROS se encarga automáticamente de vincular estos robots con sus topics básicos para asignar velocidades, odometría, láser...etc.

Ahora solo falta asignar los topics creados por nosotros a cada robot. Se ha decidido que el topic *pedirSiguienteGoal* se remapea desde el fichero de lanzamiento en el topic *robotx/pedirSiguienteGoal* dependiendo de que robot estamos lanzando. Sin embargo, el topic *robotx/goal* se crea en el propio nodo *moveRobot.cpp* siendo x el argumento que se le ha pasado como id del robot.

En resumen, para poder utilizar nuestro moveRobot para N robots hay que tener un fichero de lanzamiento que les pase como argumentos sus id correspondientes y que remapee el topic *pedirSiguienteGoal*. Además hay que hacer que el robot X se suscriba al topic *robotx/goal*.

## 5.2. Trayectoria *Circular*

Partiendo del análisis matemático que se hizo y las adaptaciones de las ecuaciones de la trayectoria *Circular* (sección 2.3.1) es más o menos sencillo realizar la implementación en C++ como se verá a continuación. En primer lugar, se van a explicar ciertos conceptos importantes de la implementación que hay que conocer. Posteriormente, se hablará un poco de cómo se han adaptado los parámetros y constantes que venían en el artículo [5] para conseguir el comportamiento deseado.

Por último, se comentará los cambios que se han hecho para conseguir que el calculador soporte N robots.

### 5.2.1. Implementación de las ecuaciones matemáticas

Para entender la implementación de las ecuaciones para la trayectoria *Circular* es fundamental entender la adaptación que se hizo de las mismas en la sección 2.3.1. Concretamente, nos basamos en las ecuaciones de las acciones vistas en las ecuaciones (2.11), (2.12) y (2.13). Estas ecuaciones son las que se han implementado de manera idéntica en C++. Cada acción, es decir, la entrada que hay que darle a un robot en su coordenada $x$, $y$ y $w$ se calculan de esta manera.

Asimismo, todo esto se junta con el bucle de control que se especifica en esa sección y que corresponde con la ecuación (2.9). De esta manera cada iteración del bucle de control lo que se va a calcular es:

$$x_{i,1}(t+1) = x_{i,1}(t) + u_{i,1}(t) * T \tag{5.1}$$

Esto es lógico, la nueva posición se calcula sumando la anterior posición más la acción que se ha realizado por el periodo. Este periodo es la manera de discretizar el entorno

del tiempo continuo que tienen en cuenta en el artículo. Así, en cada iteración del bucle de control está pasando un tiempo T.

De cara a la implementación, también es importante destacar que hay dos modos de funcionamiento:

- Se le pasa al calculador el argumento "calcular". Este calcula numéricamente para I iteraciones (e.g 10000) como se comportan los robots en el tiempo.
- Si no se le pasa ningún argumento, el nodo calculador se queda a la espera de comunicarse con los nodos del movimiento de los robots como se explicó en la sección de arquitectura 4.1. Concretamente, la comunicación exacta que sigue se puede ver en el diagrama de secuencia 4.4.

### 5.2.2. Ajuste de parámetros utilizados

Una vez comentada la implementación que se ha realizado, hay que hablar de los parámetros que el artículo proponía y que se han modificado por que no funcionaban o porque no se ajustaban correctamente.

Para comenzar, hay que decir, que los parámetros que ellos proponían para la trayectoria *Circular* en el artículo [5] no funcionan con nuestra implementación. Esto no quiere decir que dicho artículo sea incorrecto, simplemente hay que tener en cuenta que, al realizar una implementación de algo teórico en un entorno discretizado, hay aspectos que cambian y se necesitan distintas constantes.

Además no eran fácilmente ajustables, es decir, tal vez con ciertos parámetros muy bien buscados se conseguía una trayectoria buena, pero si luego se quería ver como variaba esa trayectoria con el cambio de los parámetros dejaba de funcionar.

Por ello, se han añadido y modificado algunos parámetros que se van a comentar a continuación:

- **Radio de la circunferencia:** Este es el parámetro que peor se adaptaba a la implementación ya que el valor que proponía el artículo era 800 (milímetros). Sin embargo las ecuaciones trabajan con metros por lo cual se ha cambiado a valores entre 1 y 3 más o menos. Esto se corresponde al espacio disponible en el que se pueden hacer los experimentos en el entorno real.
- **Nueva forma de relacionar los parámetros de la repulsión**. Esto es un punto muy importante: se han añadido tres parámetros D, $K_\omega$ y $d_{\text{opt}}$ nuevos. Estos junto a la $c_i$ que se tenía (que se pasa a llamar C en esta explicación) se relacionan entre sí para poder modificar la repulsión de manera más sencilla.

Ahora, C es la atracción y D la repulsión entre 2 robots. $K_\omega$ es un parámetro que ayuda a escalar las repulsiones y atracciones y $d_{\text{opt}}$ es la distancia en radianes (es decir su w) a la que queremos que se quede un robot de su vecino.

Para obtener la relación entre C y D, se va a pensar en el caso en el que queremos una separación final $d_{\text{opt}}$. Como se sabe que la w* tiende asintóticamente a acabar en el centro del *platooning*, tenemos que la atracción en el caso peor (para N robots, el robot más alejado) estará a una distancia de $K_\omega \cdot C \cdot \frac{n-1}{2} * d_{\text{opt}}$. En cuanto a la repulsión será $K_\omega \cdot D \cdot \frac{(r-r_{\text{tope}})\cdot(R-d_{\text{opt}})}{(d_{\text{opt}}-r_{\text{tope}})\cdot(R-r)}$, según la nueva función de repulsión que se explicó para que se quede entre r y R. Esto ocurre si sólo uno de los vecinos ejerce repulsión. Es decir, el vecino más cercano. Para asegurarse de esto, basta con poner una R lo suficientemente pequeña (se ha elegido $R = 1{,}5 * d_{\text{opt}}$).

Por lo tanto, se quiere que los robots estén en la configuración final en la que se mantienen unidos porque las fuerzas suman 0, es decir, la repulsión y la atracción son iguales. Esto es $C \cdot \frac{(n-1)}{2} \cdot d_{\text{opt}} = D \cdot \frac{(r-r_{\text{tope}})\cdot(1{,}5\cdot d_{\text{opt}}-d_{\text{opt}})}{(d_{\text{opt}}-r_{\text{tope}})\cdot(1{,}5\cdot d_{\text{opt}}-r)}$

De esta manera, se obtiene que la relación entre la repulsión y la atracción tiene que ser:

$$C = \frac{D \cdot (r - r_{\text{tope}})}{(n-1) \cdot (d_{\text{opt}} - r_{\text{tope}}) \cdot (1{,}5 \cdot d_{\text{opt}} - r)}$$

En resumen, se tiene ahora una forma de modificar la repulsión y atracción de los robots para que se queden a la distancia óptima que se desea. Por ello se pueden elegir la R, r, $r_{\text{tope}}$ y $d_{\text{opt}}$ para modificar el *platooning*. El efecto de los parámetros sobre el *platooning* se podrá ver mejor en la sección de Simulación 6.1.

### 5.2.3. Generalización para N robots

Los nodos calculadores, tanto el *Circular* como el *Lissajous* se instancian una sola vez, por lo que tienen que ser capaces de gestionar N robots simultáneamente. Para ello, se ha implementado una manera muy sencilla que gestione N robots.

**En primer lugar,** se guardan las posiciones de los robots en un vector de N componentes, donde cada componente es un robot. Es un vector de posiciones, por lo que si se accede a la posición 1, se tendrá la información de la posición del robot 1.

**Por otra parte,** para cada robot se suscribe el nodo a los topics *robotx/pedirSiguienteGoal* y *robotx/goal* siendo x el id del robot correspondiente. Esta es la manera en la que se comunicarán el nodo calculador con los N nodos de movimiento de los robots.

Con todo esto y la generalización para N robots del nodo de movimiento (ver sección 5.1.3) ya seríamos capaces de lanzar N robots que hagan una trayectoria *Circular* con el modelo de *platooning* diseñado.

## 5.3. Trayectoria *Lissajous*

Antes de comenzar con la explicación de la trayectoria *Lissajous*, cabe destacar que es prácticamente idéntica a la implementación realizada para la trayectoria *Circular*, por lo que si se ha entendido ésta, será muy sencillo entender la que se va a comentar ahora. Esto se debe a que son dos calculadores de trayectorias que se comportan igual, tienen los mismos modos de ejecución, se comunican de manera idéntica con los nodos de movimiento, etc. Lo único que cambia es la trayectoria que calcula cada uno, es decir, las ecuaciones implementadas. Concretamente, se implementan las ecuaciones 2.16, 2.17 y 2.18. El bucle de control sigue siendo el mismo que en el otro calculador:

$$x_{i,1}(t+1) = x_{i,1}(t) + u_{i,1}(t) * T \tag{5.2}$$

Para entender el funcionamiento de este nodo o ver el resto de características de la implementación, se emplaza al lector a la sección 5.2 donde se explica más a fondo dicha implementación.

**En cuanto al ajuste de parámetros realizado**, es el mismo que en la trayectoria *Circular*. Se ha conseguido dar valores de repulsión y atracción mediante la modificación de la distancia óptima y las distancias r, R y $r_{\text{tope}}$.

Lo único que cabe destacar como diferencia a la trayectoria *Circular*, es que en esta trayectoria es importante que el *platooning* vaya junto. Esto se debe a que si fuesen demasiado separados podrían chocarse en el punto de intersección de *Lissajous*.

Por último, **la generalización para N robots** en la trayectoria *Lissajous* es idéntica a la explicada en la sección 5.2.3.

# Capítulo 6

# Resultados en simulación

En esta sección se van a mostrar los resultados de las trayectorias de *platooning* implementadas. Para ello se va a explicar principalmente dos pruebas distintas. Estas tratan de un *platooning* de 3 robots en trayectoria *Circular* y *Lissajous*. Dichas pruebas van a ser los resultados principales de la implementación que se van a analizar a fondo. Asimismo, se va a explicar como lanzar la simulación para conseguir estos resultados.

Una vez explicadas las pruebas principales, se hará un estudio del impacto que tiene variar ciertos parámetros en las trayectorias.

Todos estos resultados irán acompañados de gráficas que ayudan a entender ciertos comportamientos.

## 6.1. Forma de lanzar la simulación

Se ha preparado un fichero *.launch* para lanzar de manera automática una simulación para 3 robots, primero para que hagan la trayectoria *Circular* y después otra simulación con *Lissajous*.

Esta simulación es muy fácil lanzarla: simplemente hay que ejecutar *simular.launch* que se encuentra en la carpeta *launch* (ver código en GitHub [3]). Este fichero carga un mundo de 20x20, donde va a emplazar 3 robots en posiciones iniciales elegidas. Se verá más adelante el efecto de cambiar las posiciones iniciales.

Es decir, lo que hace es instanciar 3 nodos de movimiento de robot asignándoles un "id" a cada uno (para que sepan que robot son). Cuando se lanza la simulación se puede ve el mundo que aparece en la figura 6.1

## 6.2. Pruebas principales

En esta sección se van a estudiar dos pruebas principales. Estas pruebas son una trayectoria *Circular* para 3 robots y una trayectoria *Lissajous* para 3 robots. Los vídeos

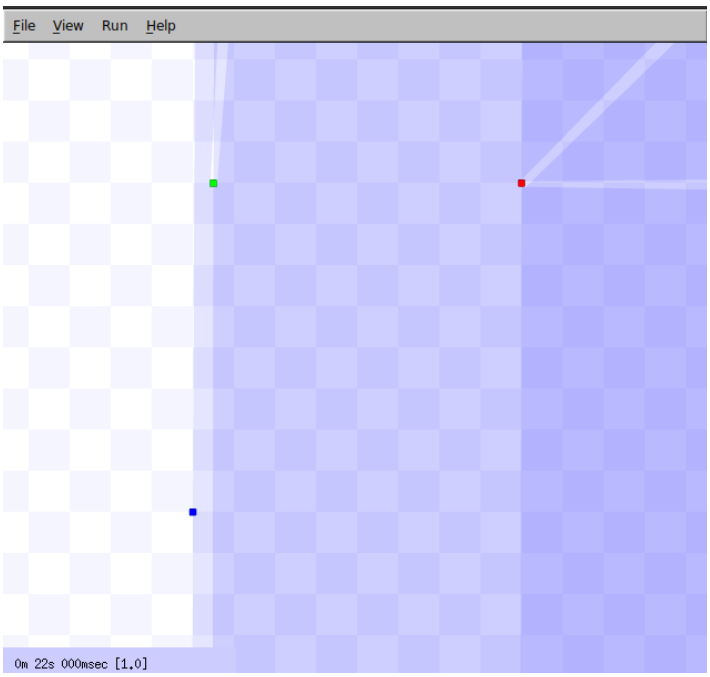



Figura 6.1: Mundo lanzado por la simulación.

de los resultados de las pruebas se pueden ver en [17] para el círculo y en [20] para el *Lissajous*.

Vamos a explicar primero **la trayectoria *Circular*** [17]. Como se puede observar en el vídeo, los 3 robots parten de posiciones cualesquiera en el mapa y acaban conformando un *platooning* de una trayectoria *Circular*. La circunferencia que realizan es de radio 1.6 metros y se pueden apreciar diversos aspectos de interés durante la simulación.

- En primer lugar, los robots parten de unas posiciones cualesquiera y buscan la circunferencia, que en este caso es 1.6 metros de radio desde el (0,0) del mapa.
- Una vez entran los 3 vehículos en la trayectoria *Circular*, están inicialmente más separados. Éstos se van a ir juntando poco a poco como se ve en el vídeo, debido a las fuerzas de repulsión y atracción, para acabar conformando el *platooning* con una distancia óptima de separación.
- Más o menos a partir del minuto y medio de simulación, se puede observar que los 3 robots han conseguido la distancia óptima, es decir, sus repulsiones se han igualado con la atracción al centro del *platooning*. A partir de este momento van a mantener esta distancia entre vecinos para el resto de la simulación.

A continuación se muestran varias gráficas que explican el comportamiento del vídeo de manera numérica.

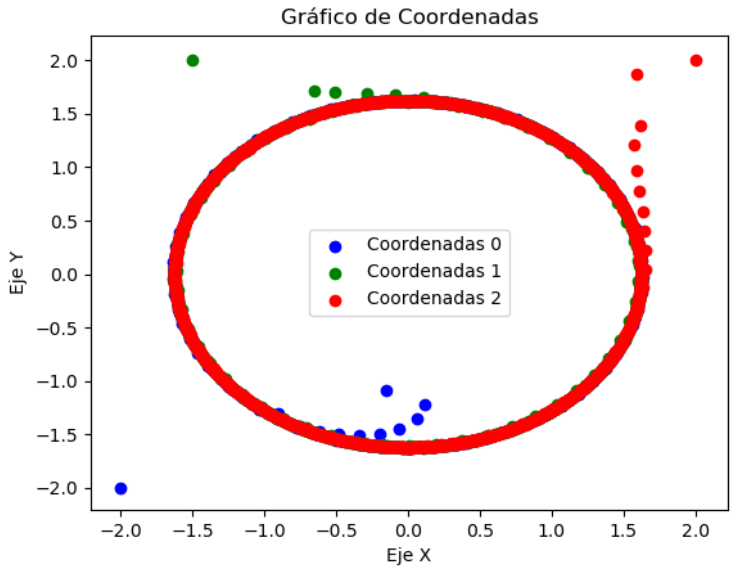
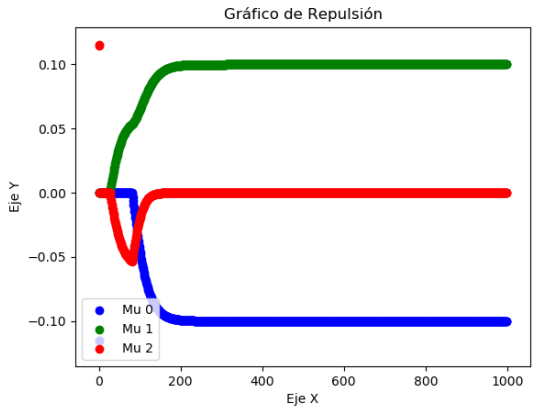



Figura 6.2: Trayectoria *Circular* de 3 robots en la simulación.

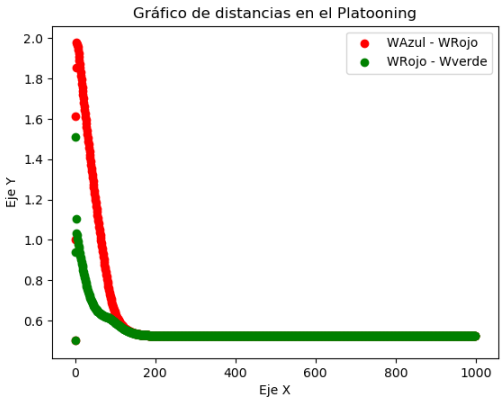
La primera figura que se aprecia 6.2 es la trayectoria que hacen los robots graficando sus posiciones en cada momento. De esta manera, se ve cómo los 3 robots parten de posiciones alejadas del círculo y finalmente acaban todos realizando una circunferencia bastante exacta.

Por otra parte, se tiene los valores de repulsión 6.3a que muestra como los 3 robots convergen en 3 repulsiones distintas pero complementarias. Esto es lógico, cuando los vehículos han alcanzado el *platooning* perfecto, los valores de repulsión son siempre los mismos, ya que se han conseguido ordenar. Por ejemplo, en nuestro caso al tener 3 vehículos, cuando alcancen el *platooning* con distancia óptima, la repulsión es la que se observa. Es decir, los dos vehículos de ambos extremos tendrán repulsiones en direcciones opuestas y el vehículo del centro tendrá repulsión 0.

Este mismo efecto se puede observar en la figura adyacente a la que acabamos de comentar 6.3b. Esta figura muestra la evolución de la diferencia de la coordenada W entre un vehículo y su vecino. Es decir, muestra según el sistema que se tiene de posición en la trayectoria como varía la separación entre los vehículos. Por eso, se puede ver que los vehículos parten inicialmente muy separados pero acaban convergiendo en una distancia. Esta es la distancia óptima. En el caso de esta simulación, se eligió 30 grados como distancia óptima, que son 0.52 radianes. Como se aprecia en la figura los vehículos se quedan justo a ese valor de separación.

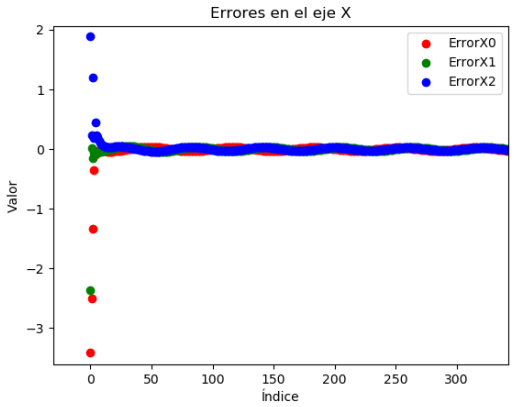



(a) Valores de repulsión para 3 robots en la trayectoria *Circular*.

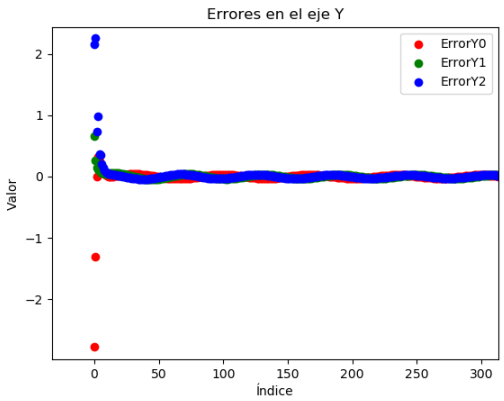



(b) Valores de W entre vecinos para 3 robots en la trayectoria *Circular*.

Figura 6.3: Gráficas de repulsión y distancia entre vecinos para 3 robots en trayectoria *Circular*.




(a) Errores en el eje X para 3 robots en la trayectoria *Circular*.




(b) Errores en el eje Y para 3 robots en la trayectoria *Circular*.

Figura 6.4: Errores en ejes X e Y para trayectoria *Circular*.

Por último, se ven los errores en el eje X y eje Y (figs 6.4a y 6.4b). Éste es otro hecho que respalda ver que se ha conseguido un *platooning* correcto. Los robots empiezan teniendo un error muy alto y acaban convergiendo a un error prácticamente nulo. Esto es nuevamente porque convergen en la circunferencia y ahí no se van prácticamente nada del punto que les corresponde.

Una vez vista la trayectoria *Circular*, es muy sencillo entender los resultados para **la trayectoria *Lissajous***. Se recuerda al lector que el vídeo correspondiente es [20].

En este vídeo se ve el mismo comportamiento que había para la trayectoria *Circular*, los robots parten de posiciones iniciales cualesquiera y se juntan en un *platooning* de una trayectoria *Lissajous*.

Se muestran a continuación las mismas gráficas que se mostraron pero para la trayectoria *Lissajous*.

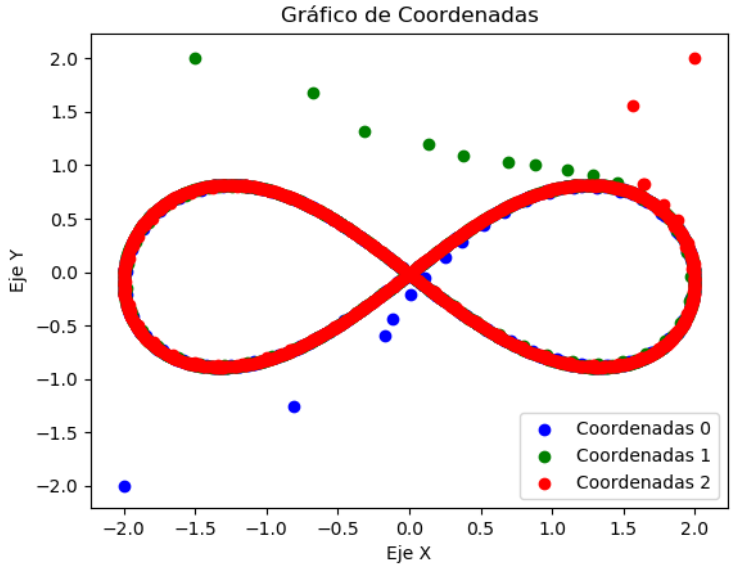
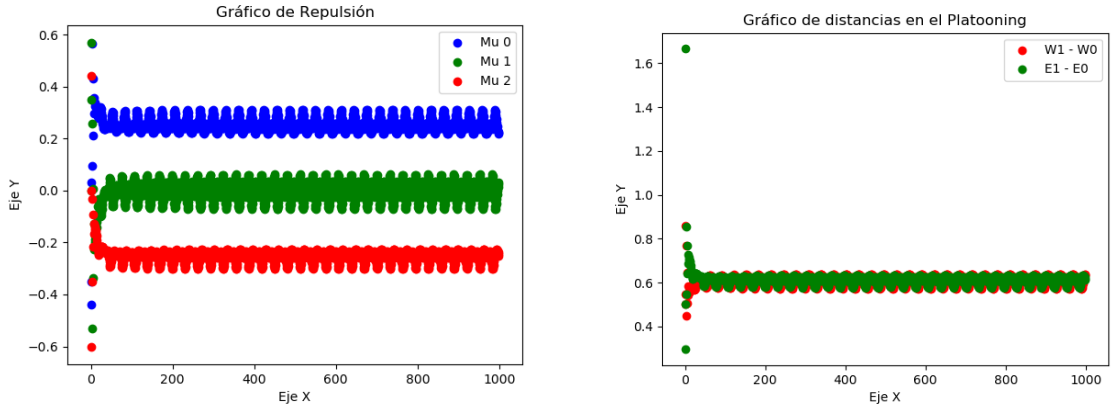



Figura 6.5: Trayectoria Lissajous de 3 robots en la simulación.



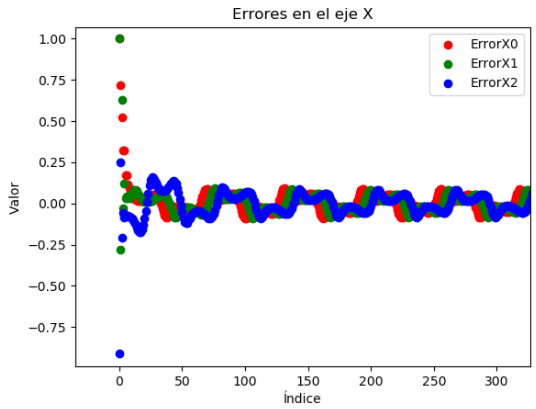

(a) Valores de repulsión para 3 robots en la trayectoria Lissajous.

(b) Valores de W entre vecinos para 3 robots en la trayectoria Lissajous.

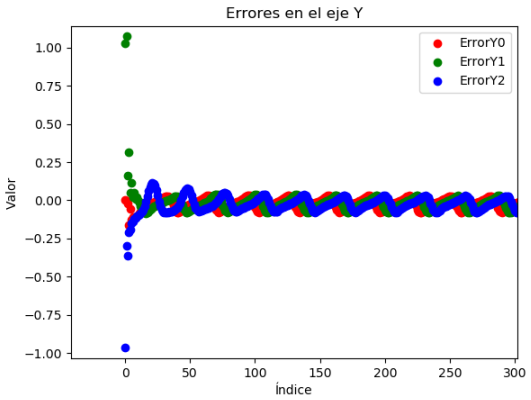
Figura 6.6: Gráficas de repulsión y distancia entre vecinos para 3 robots en trayectoria *Lissajous*.

Estas gráficas muestran el mismo comportamiento que mostraban las del círculo y respaldan por tanto, los resultados que se ven en el vídeo. Es decir, se alcanza un *platooning* correcto en la trayectoria *Lissajous*.

Una de las diferencias notables respecto a las gráficas que se vieron del círculo son los valores de repulsión y W ( figs 6.6a y 6.6b). Estos no convergen en un solo valor, esto se debe a que al ser una trayectoria más compleja (es decir, tiene giros




(a) Errores en el eje X para 3 robots en la trayectoria Lissajous.




(b) Errores en el eje Y para 3 robots en la trayectoria Lissajous.

Figura 6.7: Errores en ejes X e Y para trayectoria *Lissajous*.

y distintos cambios de ángulo) los valores de repulsión convergen en un conjunto de valores distintos. Para que se entienda, en la trayectoria circular, una vez alcanzado el *platooning* las fuerzas podrían ser el mismo valor y los vehículos seguirían siempre el mismo camino. Sin embargo, en *Lissajous* son una serie de valores que se repiten todo el rato para mantener el *platooning* unido. Este "rizado" de los valores se podría corregir bajando T o $K_\omega$. Sin embargo tardaría más en converger. Es por esto que se ha decidido utilizar estos valores.

Asimismo, se observa este efecto en los errores de los ejes X e Y. Para la trayectoria *Lissajous*, al ser una trayectoria más compleja y por los valores de T y $K_\omega$ , tarda más en converger y mantiene el "rizado" que se ha comentado.

En resumen, lo único que cambia es la complejidad de la trayectoria, en ambas trayectorias se ven los mismos efectos y ambas conforman un *platooning* correcto, es decir, mantiene las propiedades del *platooning* que conocemos.

## 6.3. Otras pruebas. Estudio de parámetros

Una vez vistas dos trayectorias correctas de *platooning*, se va a pasar a comentar ciertos aspectos que se han observado mediante la modificación de algunos parámetros.

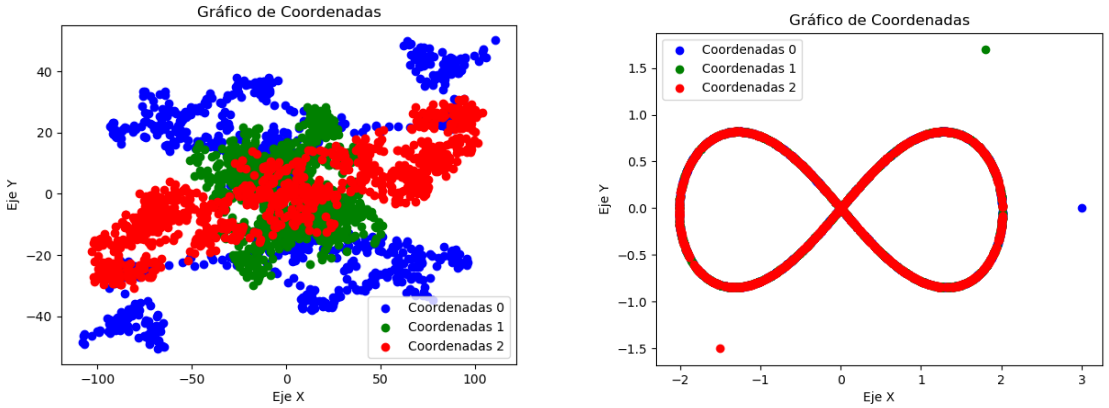
### 6.3.1. Constantes k1, k2 y radio circunferencia

Se ha apreciado que las constantes k1 y k2 son las que le dan principalmente la forma a la trayectoria. Estas son las ganancias que se aplican en el eje X y en eje Y de las ecuaciones que conforman ambas trayectorias (tanto *Circular* como *Lissajous*). Se

pueden observar estas constantes en las fórmulas que se veían en el análisis matemático de la trayectoria *Circular* (2.11) o de *Lissajous* (2.16). Es **muy importante** elegirlas bien puesto que, si se eligen mal, no va a funcionar nada. Elegir unas constantes óptimas requiere mucho tiempo y, como se va a ver, si se aumenta el radio de la circunferencia en gran medida, se necesita buscar unas nuevas constantes. Este estudio se ha hecho mediante el modo "DEBUG" del código implementado. De esta manera, se ha podido observar el efecto en las trayectorias que resultan de modificar dichas constantes.

En la figura 6.8a se aprecia qué es lo que ocurre si, para la trayectoria *Lissajous* que se enseñó en los resultados principales, se aumentan enormemente las constantes. Antes se utilizaba constantes con valor 3 y ahora se ha aumentado a valor 20.

Esto produce un efecto totalmente distorsionado de la trayectoria. Se aprecia una nube de puntos que intenta tener forma de *Lissajous* pero, al ser valores tan altos, no lo consigue. Si estas constantes se bajan un poco pero siguen siendo altas (e.g valor 10) se aprecia el efecto de la figura 6.8b. Se obtiene la trayectoria pero se pierden muchos puntos de aproximación de los vehículos a la misma, es decir, se sobreajusta de tal manera que pierde realismo. Es decir, la trayectoria planificada difiere mucho de la realmente ejecutada por los robots.

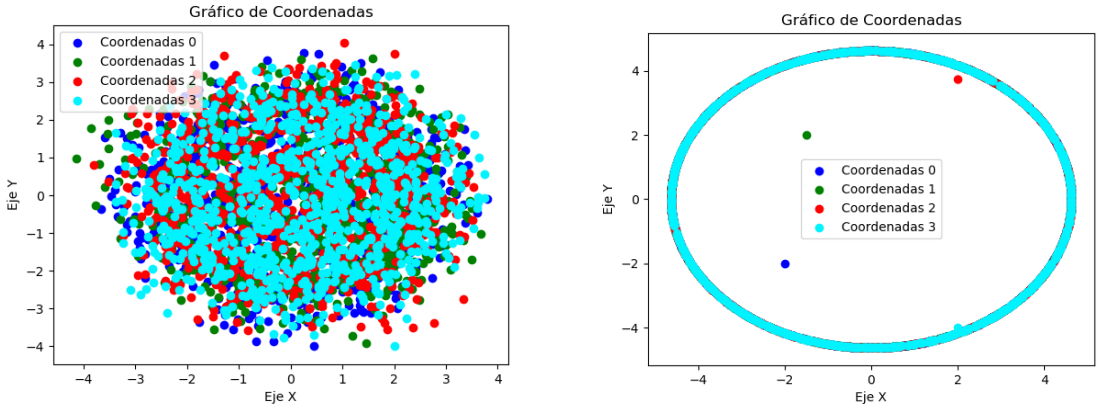



(a) k1 y k2 muy altas *Lissajous*.

(b) k1 y k2 sobreajustadas *Lissajous*.

Figura 6.8: Trayectorias *Lissajous* para distintos valores de k1 y k2.

El mismo efecto se aprecia al cambiar el radio y no las constantes. Si ahora utilizamos las constantes que se tenían para la trayectoria *Circular* de los resultados principales 6.2 y le aumentamos considerablemente el radio (e.g 4.6 de radio), se forma una nube de puntos que intenta imitar una trayectoria *Circular* pero al tener constantes tan bajas no lo consigue (fig 6.9a). Si buscamos unas nuevas constantes que se adapten a nuestra nueva circunferencia se consigue una trayectoria *Circular*, algo sobreajustada, pero con buena forma (fig 6.9b).




(a) Radio de circunferencia 4.6 y constantes k1 y k2 = 3

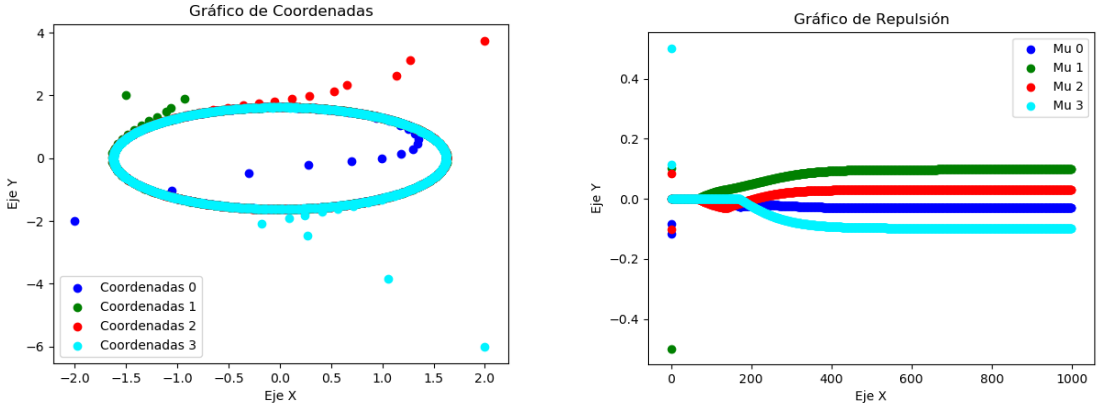
(b) Radio de circunferencia 4.6 y constantes k1 y k2 = 10

Figura 6.9: Trayectorias *Circular* para radio 4.6 metros

En resumen, se ve una relación entre el radio de la circunferencia y las constantes k1 y k2. Se sabe que si se aumenta el radio de la circunferencia considerablemente, se debe aumentar también las constantes. Sin embargo, no hay una relación matemática entre ellas. Dependerá de la trayectoria y el resto de parámetros.

### 6.3.2. Número de robots

El número de robots se puede cambiar con facilidad aunque resulta una tarea tediosa de simular. Por ello se ha hecho un vídeo con 4 robots para que se vea como sigue funcionando correctamente. Es la trayectoria *Circular* con 4 robots y se puede encontrar aquí [16]. Se puede observar la trayectoria qué estos hacen y cómo las repulsiones se nivelan (recordemos que esto quiere decir que se ha alcanzado un *platooning* correcto) en las figuras 6.10a y 6.10b.

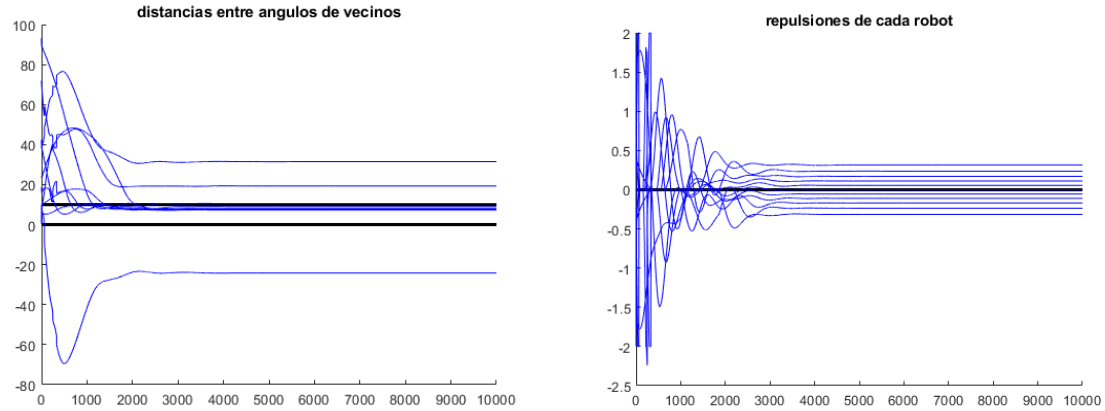



(a) Trayectoria *Circular* 4 robots.

(b) Valores repulsión 4 robots.

Figura 6.10: Trayectoria y repulsiones para círculo de 4 robots.

Como simular más robots es una tarea tediosa, se va a introducir 2 gráficas (figs 6.11a y 6.11b) que muestran la evolución de la repulsión y la diferencia de las posiciones entre vecinos para un número de robots grande (e.g 11). De esta manera, aunque no se simule, se demuestra numéricamente que se alcanza un *platooning* correcto independientemente del número de robots siempre y cuando quepan en la trayectoria. Cabe destacar una línea que se aprecia con valor negativo en la figura 6.11a (por debajo de -20 grados). Esto simplemente ocurre ya que la coordenada virtual $w$ puede tomar valores entre $-\pi$ y $\pi$. Es por ello que es posible que haya diferencias negativas. Lo importante, es que estas diferencias siguen manteniendo la distancia óptima entre vecinos. Estas gráficas muestran, como se aprecia, que las repulsiones de los 11 robots




(a) Diferencia W vecinos para 11 robots. (b) Evolución repulsiones para 11 robots.

Figura 6.11: Diferencia de vecinos y repulsiones para 11 robots.

tienden a quedarse estables (se alcanza un *platooning* correcto) y que las distancias entre vecinos nunca bajan de la distancia óptima, en este caso 10 grados.

### 6.3.3. Distancia óptima entre vecinos

La distancia óptima (parámetro $d_{\text{opt}}$) se puede cambiar para que los robots se queden a la distancia que se quiera entre ellos. En este caso se ha probado a realizar una simulación de *Lissajous* con una distancia óptima mucho mayor que la simulación de los resultados principales. Se le ha puesto 50 grados de separación entre ellos. Se puede ver esta simulación en el vídeo [18].

El vídeo muestra como se quedaban en los resultados principales y como se quedan en esta prueba. Se ve una notable diferencia de distancia entre los robots. Cabe destacar que en el *Lissajous* concretamente, es importante no darles una separación excesiva ya que podrían ir tan separados que se chocasen en el punto de intersección.

### 6.3.4. Importancia de w*. Velocidad

La $w^*$ es la coordenada virtual objetivo, esta se explicó en la sección de análisis matemático (ver 2.1.2). Resumiendo, es un "robot virtual" que no existe y va realizando la trayectoria a una velocidad asignada. De esta manera marca el ritmo al que tiene que ir el *platooning*.

En este caso se ha probado a bajar esa velocidad (de normal es 1 radián por segundo) a mucho menos (0.1 radianes por segundo). El efecto que se consigue con esto es que todo el *platooning* vaya extremadamente lento, tanto que no es ni práctico. Los robots comenzarán a su velocidad normal hasta que converjan en la trayectoria. Una vez dentro de la trayectoria es cuando empieza a hacer efecto este ritmo que marca la coordenada virtual objetivo, por lo tanto en cuanto entren en la trayectoria reducirán su velocidad enormemente.

Se ha confeccionado un vídeo en el que se ve primero como van los robots a su velocidad normal y posteriormente como se van con el cambio de velocidad a 0.1. Este vídeo puede verse en [7]

Este efecto se traduce en que se tiene un parámetro que permite ajustar la velocidad del *platooning*. Es muy práctico ya que se puede subir o bajar la velocidad a la que van los robots conforme nos convenga.

# Capítulo 7

# Experimentación en un entorno real

Es conocido por todo el mundo que el entorno real plantea una serie de dificultades que no surgen a nivel matemático o incluso en las simulaciones. Una vez realizado el groso del trabajo principal se ha querido ver si era posible transferir parte de este trabajo a un entorno real. Para ello, se han hecho varias pruebas en un entorno real como se verá durante este capítulo.

En dicho capítulo, se explicará como se ha adaptado una prueba de una trayectoria *Circular* con 1 robot a un entorno real. Se comentará que hardware se ha utilizado, cuales son los pasos que se han seguido para poner todo en marcha y se verán resultados de la prueba.

## 7.1. Pasos seguidos para las pruebas

Antes de comenzar a explicar las adaptaciones que se han llevado a cabo, es importante destacar que los robots móviles "Turtlebots" utilizados se pueden encontrar en [13]. Lo único que no se ha utilizado ha sido la cámara *Kinect*. En su lugar, se ha utilizado el sensor láser Hokuyo UST 20LX [6].

Se observa una imagen del aspecto del Turtlebot en el entorno real en la figura 7.1. Lo primero que se ha hecho para poder realizar las pruebas ha sido **generar un mapa**. Para ello se ha utilizado el módulo de ROS Gmapping [4]. De esta manera se ha ido moviendo el robot por la zona en la que íbamos a trabajar para que fuese reconociendo lo que había alrededor suyo y fuese generando el mapa correspondiente. El mapa corresponde a un espacio diáfano en la primera planta del edificio Ada Byron en la Escuela de Ingeniería y Arquitectura. Tiene el siguiente aspecto (ver fig 7.2). Consiste en una serie de celdas que pueden ser de color blanco, gris o negro. El color blanco representa una celda libre, el color negro una celda ocupada y el color gris una celda desconocida. De esta manera, cuando el robot está localizado en el mapa, sabe por donde puede ir o no.


Figura 7.1: Turtlebot utilizado en entorno real.

Se ha elegido el (0,0) como el centro de la sala, para que el robot realice la trayectoria a su alrededor y tenga espacio suficiente.

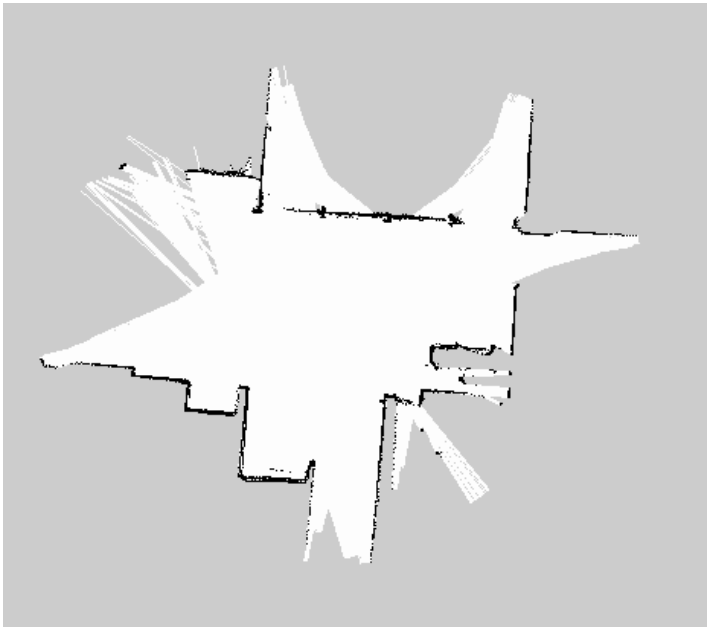


Figura 7.2: Mapa utilizado en entorno real.

Una vez generado el mapa simplemente se ha puesto al robot en una posición inicial dentro del mapa (en concreto se ha puesto en la posición -2,-2). El robot utiliza el nodo AMCL de ROS para localizarse en el mapa [10]. Este nodo implementa un filtro de partículas cuya explicación más detallada puede verse en el Anexo A.

Posteriormente, se han cambiado los topics de la implementación para que, en vez de comunicarse con la simulación, mandasen los mensajes al Turtlebot. Éste tiene unos topics concretos donde tiene la velocidad, odometría y láser.

Se ha utilizado el programa *MobaXterm* [9] para conectarse con el Turtlebot y pasarle los ficheros de la implementación. Es importante destacar que se tiene que tener acceso desde el Turtlebot para poder conectarse, es decir, tiene que tener tu IP el Turtlebot para permitirte la conexión.

Una vez realizados todos estos pasos, se ha procedido a lanzar las pruebas. Se ha probado una trayectoria *Circular* con 1 robot para ver que convergía correctamente y la hacía bien.

En las figuras que se muestran más abajo 7.3, se puede ver cómo el robot parte de una posición inicial y acaba dando vueltas a 1.6 metros del centro de la circunferencia.

Asimismo, se añade un vídeo en [15] de la prueba real que se comenta. En dicho vídeo se muestra también mediante el programa RViz [11], que es un visualizador gráfico de la información que nos proporcionan por los topics los nodos de ROS, la trayectoria que está haciendo el robot en tiempo real. Se puede observar que hace una circunferencia de 1.6 metros como se ha dicho (cada baldosa de Rviz corresponde a 1 metro, por lo que hace algo menos de 2 baldosas).

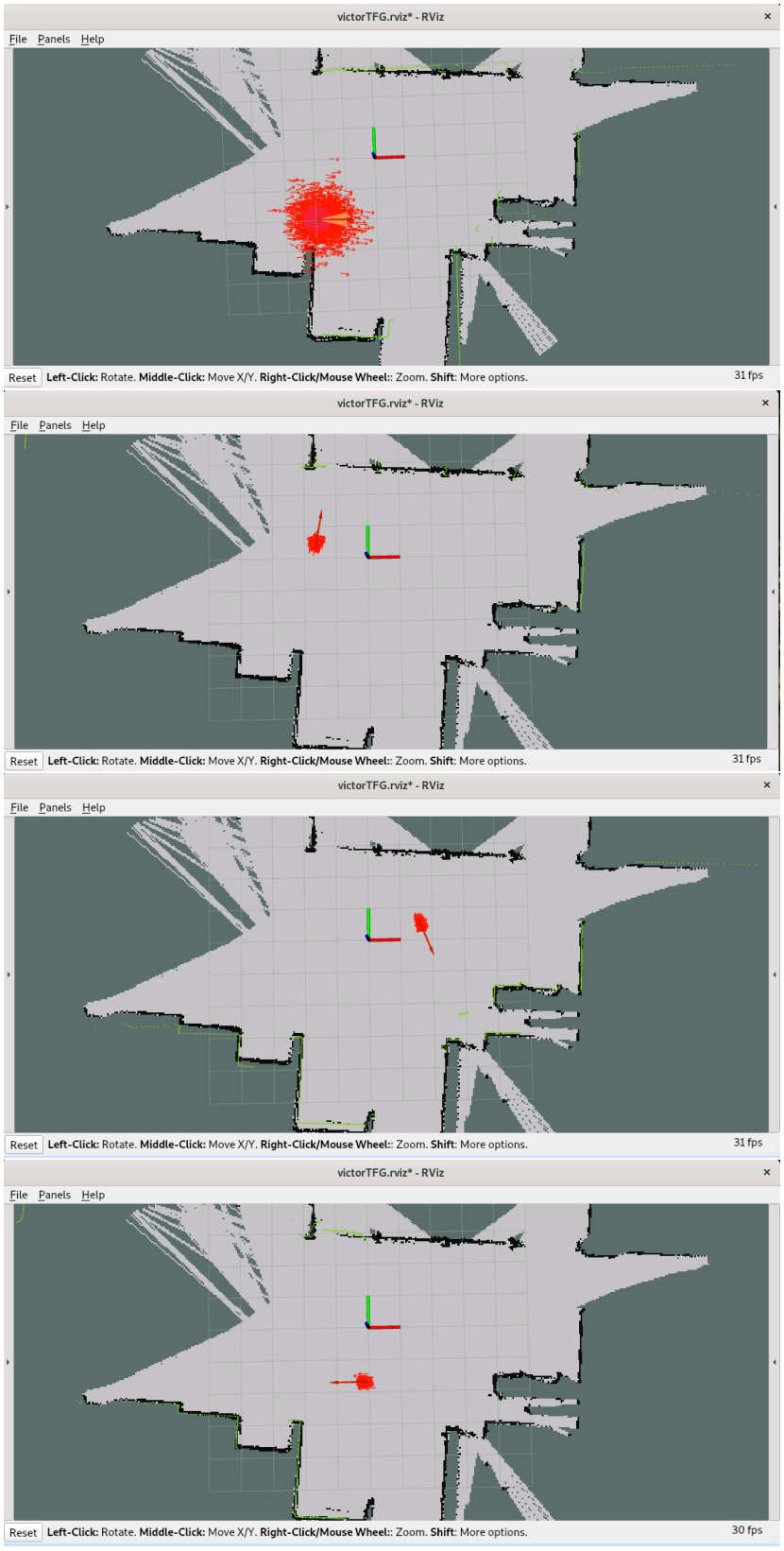



Figura 7.3: Distintas posiciones del robot en entorno real vistas desde Rviz.

## 7.2. Lecciones aprendidas

Realizando el paso al entorno real se ha visto que surgen muchos problemas que no aparecen en la simulación. Concretamente se destacan estos puntos:

- Como el nodo de movimiento del robot no acelera sino que da velocidades enteras de golpe, esto se traduce en el entorno real con que a veces el *Turtlebot* da pequeños "botes" ya que cambia drásticamente su velocidad. Esto es un aspecto que no se puede ver en un entorno simulado tan fácilmente ya que funciona bien, sin embargo al pasar al entorno real se aprecia mejor. Se propone una alternativa de implementación como línea futura de trabajo en el capítulo *Conclusiones* (ver 8.1).

- Hay diferencias entre como funciona el Turtlebot y como funciona el simulador STAGE a nivel de comunicación. Por ejemplo, en la odometría en el simulador se publica constantemente los datos de la posición del robot. Sin embargo, en el Turtlebot solo actualiza el filtro de particulas y publica nuevas posiciones si se ha movido considerablemente respecto a la anterior posición. Este aspecto se ha solucionado publicando continuamente la posición en el Turtlebot también.

# Capítulo 8

# Conclusiones

Para finalizar con el trabajo que se ha presentado en esta memoria, se van a comentar las conclusiones a las que se ha llegado realizándolo. Antes de nada, cabe destacar que se han cumplido todos los objetivos que se propusieron en el capítulo de Introducción. Se ha conseguido entender perfectamente las matemáticas que modelan el *platooning* y se han conseguido implementar varias trayectorias totalmente funcionales con las características que se buscaban. Además, se ha probado a fondo en simulación y en un entorno real. Para mayor información de las fases de trabajo realizadas se referencia al lector al diagrama de Gantt del Anexo C.

Como ya se sabía, el entorno de la robótica es un entorno complejo que requiere tener unas bases bastante claras de cómo funciona para poder entenderlo. Concretamente, en este trabajo, se ha partido de un artículo bastante denso a nivel matemático, lo cual ha dificultado las cosas. No hablaban tanto de casos concretos de aplicación sino más de generalizaciones matemáticas para formar un *platooning*. Siempre acompañado de poca explicación de palabra y muchas ecuaciones que a veces no aportaban demasiado.

Asimismo, se ha visto la diferencia de trabajar con un artículo a nivel teórico y de implementarlo. En la implementación siempre surgen aspectos que no se han tenido en cuenta y se tienen que resolver, como por ejemplo: la función de repulsión que hubo que cambiar, los parámetros que hubo que calcular, la discretización del tiempo mediante el periodo T, etc.

Otro aspecto importante que cabe destacar es la diferencia entre la simulación y la realidad. Una vez implementado todo y hechas las pruebas en STAGE, se vio que no era nada fácil adaptarlo al entorno real ya que surgen problemas de hardware, de comunicación, etc. Esto nos enseñó que el entorno real siempre cuesta más puesto que surgen problemas donde pensabas que no había.

Resumiendo, y como experiencia personal del autor, pienso que es un campo muy interesante y bonito el de la robótica pese a todos los problemas y enigmas que contiene. El poder realizar un trabajo sobre esto ayuda a darnos cuenta de la dificultad que conlleva y a acercarnos un poco más a este mundo.

## 8.1. Futuras líneas de trabajo

Con el TFG realizado se abren varias líneas de trabajo sobre las cuales se podría seguir estudiando:

- Una de ellas podría ser el adaptarlo mucho más a un entorno real partiendo de las bases que ya se han hecho en este trabajo. Se podría probar a realizar otras trayectorias, añadir más robots, realizar pruebas reales de transportes de mercancías o incluso pruebas en el entorno de la seguridad vial.

- Por otra parte, también se podría seguir investigando el *platooning* de otras trayectorias. Obtener las ecuaciones matemáticas que las componen y realizar los mismos estudios e implementaciones que se ha hecho para la trayectoria *Circular* o *Lissajous*. Con lo implementado en este TFG, si supiesemos describir una trayectoria (por muy compleja que sea) mediante ecuaciones matemáticas, podría ser posible implementar un movimiento de un *platooning*.

- Realizar un algoritmo más sofisticado de navegación. Se podría combinar las velocidades líneales y angulares o incluso trabajar con aceleraciones para evitar los "botes" de los que se hablaba en el capítulo del entorno real (ver 7.1).

# Bibliografía


[1] ROS Contributors. *Documentation - ROS Wiki*. Accessed: 2023-12-27. 2021. URL: https://wiki.ros.org/Documentation.

[2] *GeoGebra*. Accessed: 2023-01-03. URL: https://www.geogebra.org/.

[3] *GitHub del código realizado*. Accessed: 2023-01-16. URL: https://github.com/viictorgallardo/TFG-Robotics.git.

[4] *Gmapping para mapa entorno real*. Accessed: 2023-01-16. URL: http://wiki.ros.org/gmapping.

[5] Bin-Bin Hu et al. "Spontaneous-Ordering Platoon Control for Multirobot Path Navigation Using Guiding Vector Fields". En: *IEEE Transactions on Robotics* 39.4 (2023), págs. 2654-2668. DOI: 10.1109/TRO.2023.3266994.

[6] *Láser utilizado en el entorno real*. Accessed: 2023-01-16. URL: https://www.roscomponents.com/es/lidar-escaner-laser/ust-20lx.

[7] *Lissajous rápido versus lento*. Accessed: 2023-01-16. URL: https://youtu.be/JR4oJXvxSFc.

[8] *Lista de reproducción de vídeos*. Accessed: 2023-01-16. URL: https://www.youtube.com/playlist?list=PL2pZRSxEnFj6d33bT_PiCEe7uN6T_1Rx7.

[9] *MobaXTerm program*. Accessed: 2023-01-16. URL: https://mobaxterm.mobatek.net/.

[10] *Paquete de localización del Turtlebot*. Accessed: 2023-01-16. URL: http://wiki.ros.org/amcl.

[11] *Rviz ROS Documentation*. Accessed: 2023-01-16. URL: http://wiki.ros.org/rviz.

[12] *STAGE simulator ROS*. Accessed: 2023-01-05. URL: http://wiki.ros.org/stage.

[13] *Turtlebot utilizado en el entorno real*. Accessed: 2023-01-16. URL: https://clearpathrobotics.com/turtlebot-2-open-source-robot/#:~:text=TurtleBot%202%20is%20the%20world's,Pro%20Sensor%20and%20a%20gyroscope..

[14] *Tutorial obstáculos*. Accessed: 2023-01-16. URL: https://youtu.be/rzKAFZyYBqk.

[15] *Vídeo de 1 Turtlebot en entorno real*. Accessed: 2023-01-16. URL: https://youtu.be/_aJrMpj7cY4.

[16] *Vídeo de 4 robots trayectoria Circular*. Accessed: 2023-01-16. URL: https://youtu.be/1K_MbpV7DX4.

[17] *Vídeo de círculo principal*. Accessed: 2023-01-16. URL: https://youtu.be/fanZT8_dp3o.

[18] *Vídeo de diferencias con $d_{opt}$*. Accessed: 2023-01-16. URL: https://youtu.be/XVjk1HjMg_M.

[19] *Vídeo de obstáculos antes vs ahora*. Accessed: 2023-01-16. URL: https://youtu.be/6h7BQFnZrs0.

[20] *Vídeo Lissajous principal*. Accessed: 2023-01-16. URL: https://youtu.be/xMGpJPUws4Y.

[21] Weijia Yao et al. "Singularity-Free Guiding Vector Field for Robot Navigation". En: *IEEE Transactions on Robotics* 37.4 (2021), págs. 1206-1221. DOI: 10.1109/TRO.2020.3043690.

# Lista de Figuras

# Lista de Tablas

# Anexos

# Anexos A

# Filtro de partículas

El filtro de partículas es un mecanismo para la localización de un robot móvil en un entorno real. Como ya se sabe, la odometría como tal acumula mucho error, por lo que es necesario un mecanismo más sofisticado para localizar al robot en cada momento. En este caso se ha utilizado AMCL, que es un método de localización probabilístico para un robot que se mueve en un espacio 2D. Utiliza un filtro de partículas para buscar la posición de un robot frente a un mapa conocido.

La función principal del filtro de partículas es estimar la posición de un robot como se ha comentado. Para ello, se ayuda de varios mecanismos, como son el mapa que ha generado, el láser y la odometría del robot.

Como se observa en la figura A.1 el robot empieza en una posición inicial con una gran incertidumbre. Todas las flechas rojas que se ven, son posibles lugares donde podría estar el robot ya que aún no lo sabe concretamente. Conforme vaya moviéndose y consiguiendo datos de su entorno mediante el láser y la odometría, irá precisando su posición.

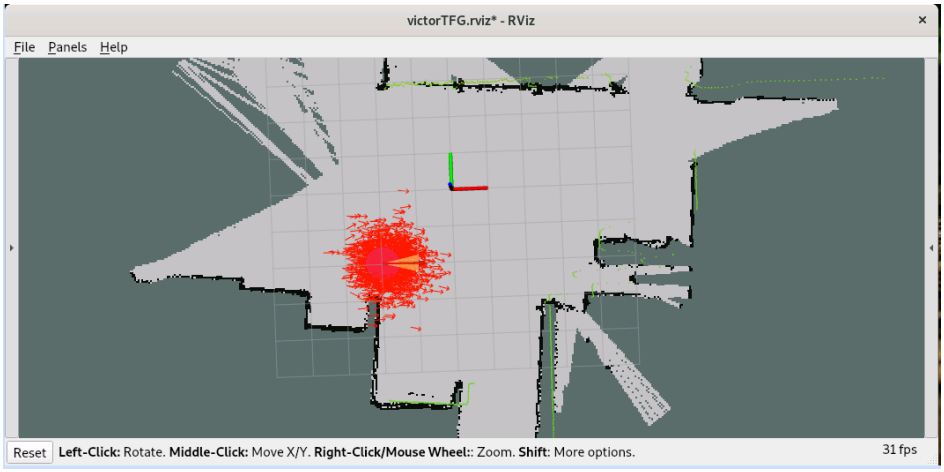



Figura A.1: Filtro de partículas para la posición inicial del robot.

Ahora el robot comienza a moverse y ajusta un poco más su posición. Para ello utiliza la odometría junto con lo que se ha movido y el láser junto con las paredes y elementos del mapa que ya conoce. El láser se puede ver como las líneas verdes de la imagen A.2.

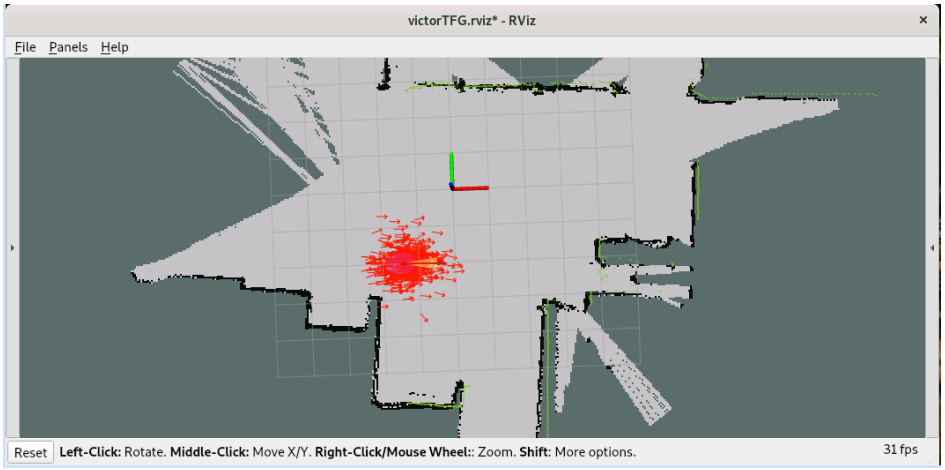



Figura A.2: Filtro de partículas para la posición 2 del robot.

El filtro de partículas sigue convergiendo como se aprecia en las figuras A.3 y A.4. Se apoya de los datos que obtiene junto a la odometría, el láser y su entorno. De esta manera va ajustando cada vez más la nube de incertidumbre donde podría estar el robot. En la figura A.4 se aprecia muy bien como reduce mucho la incertidumbre al dar un giro, ya que el robot se da cuenta de que ha cambiado su orientación y por tanto las posibilidades de donde está se reducen mucho.

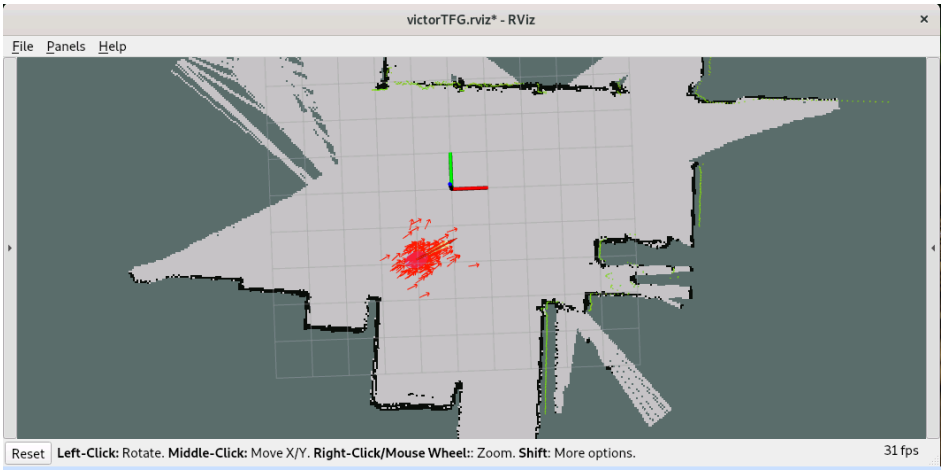



Figura A.3: Filtro de partículas para la posición 3 del robot.

Por último, se muestra un figura tras varias vueltas a la trayectoria *Circular*. El filtro de partículas ya se ha ajustado perfectamente y por lo tanto el robot está muy bien localizado (ver A.5).

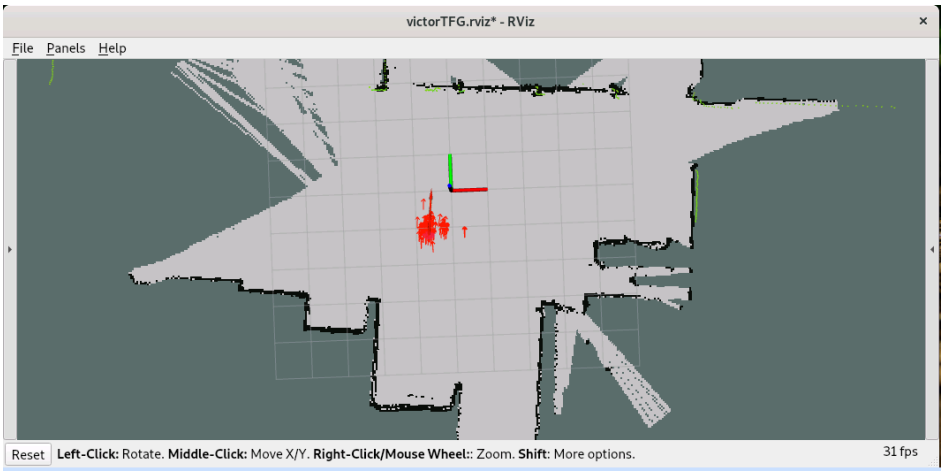



Figura A.4: Filtro de partículas para la posición 4 del robot.

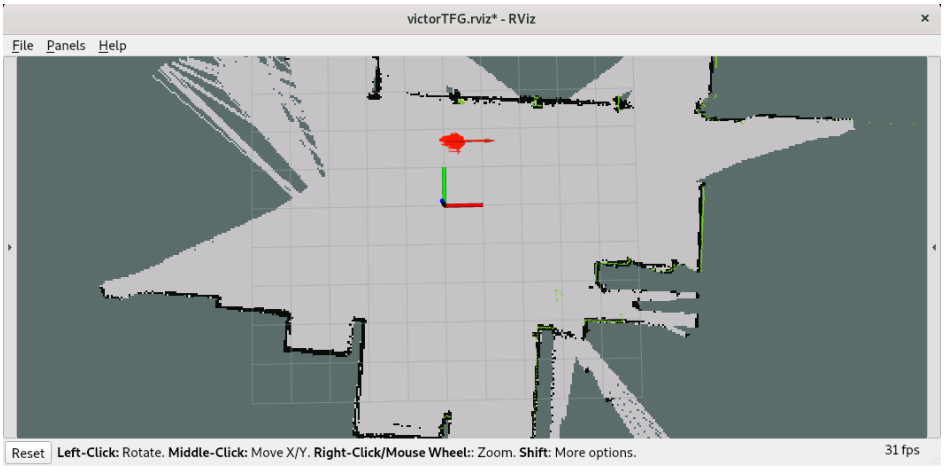



Figura A.5: Filtro de partículas para la posición final del robot.

# Anexos B

# Herramientas y trabajo previo

En este Anexo, se va a explicar que trabajo previo se hizo con las herramientas que se iban a utilizar. Antes de comenzar con la implementación, era muy importante entender bien los conceptos principales de ROS. La distribución que se iba a utilizar era ROS Noetic. Para ello se hicieron 2 tutoriales que abarcaban la información más importante que se necesitaba. Estos tutoriales corresponden a las prácticas 1 y 2 de la asignatura *Autonomus Robots* en el *Máster Universitario en Robótica, Gráficos y Visión por Computador* de UNIZAR. Al ser unas prácticas privadas no se van a referenciar, sin embargo, se puede encontrar la documentación de ROS que resumen estas prácticas en [1].

Asimismo, se estudió el tipo de dato *LaserScan* que viene en ROS para obtener los datos del sensor láser y realizar detección de obstáculos. Se hizo una detección de obstáculos sencilla que se utilizará muy parecida en la implementación del *platooning*.

## B.1. Aprendizaje de ROS

Como se ha comentado en la introducción de este Anexo, se realizaron 2 prácticas que resumen los conceptos más importantes que se pueden encontrar en [1].

**La primera práctica** explicaba conceptos básicos como:

- Nodos, topics, como compilar...
- Manejo de los comandos básicos de terminal
- ROS en Visual Studio Code
- Simulador STAGE

Proponía varios ejercicios que consistía en publicar mensajes, escucharlos, entender como se comunican los nodos y como se lanzan.

**La segunda práctica** ya entraba más de lleno en la programación en ROS. Se pretendía realizar un programa que leyese unas metas de un fichero y las publicase en un *topic*. A su vez, otro nodo (del que daban el esqueleto) tenía que leerlas e ir hacia ellas.

Esto se implementó y se lanzó en el simulador STAGE. Se puede ver una imagen en la figura B.1 del comportamiento implementado. El mundo tiene 4 objetivos (cuadrados

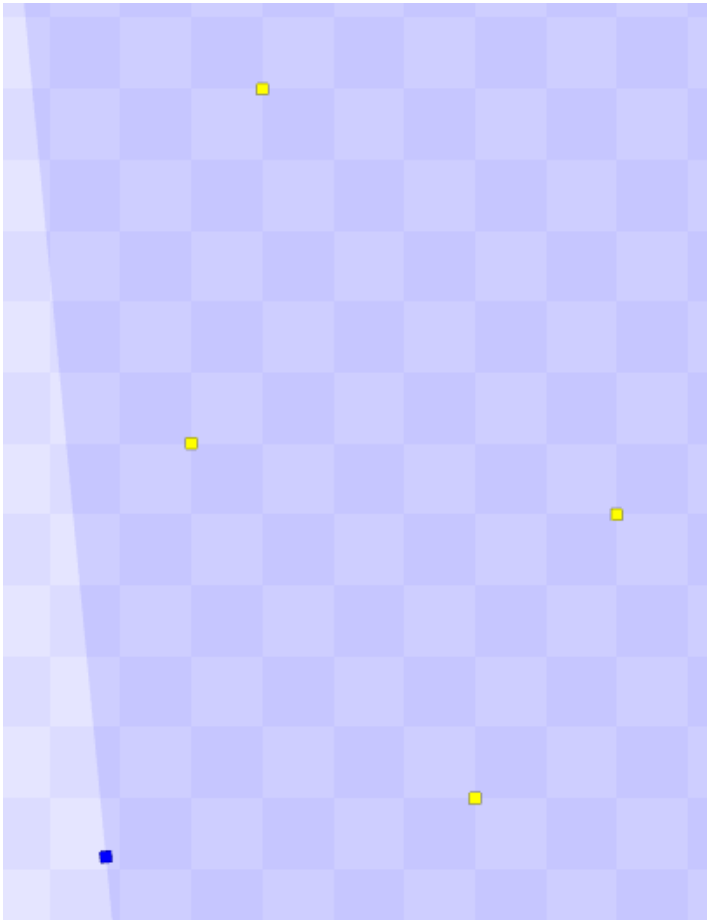


Figura B.1: Tutorial 2 lanzado en el simulador STAGE.

amarillos) que son los mismos que lee el nodo *followTargets*. Este nodo los lee y los publica en el topic *goal* conforme el nodo de movimiento va llegando a las metas.

La práctica pedía completar la lógica de movimiento del robot. En resumen, lo que hace el robot es girar sobre sí mismo hasta que se queda con la misma orientación que la dirección en la que está la meta. Una vez conseguida esa orientación va recto hacia delante.

Se entrará en mas detalles de la lógica de movimiento cuando se explique la implementación (ver sección 5.1) ya que se ha basado en lo desarrollado en esta práctica para el movimiento de los robots en el *platooning*.

Finalmente, esta práctica buscaba que el robot recorriese las 4 metas, por lo que empezaba yendo a la primera como se ha visto en la anterior figura y trás recorrer todas llegaba a la última como se aprecia en la siguiente B.2.

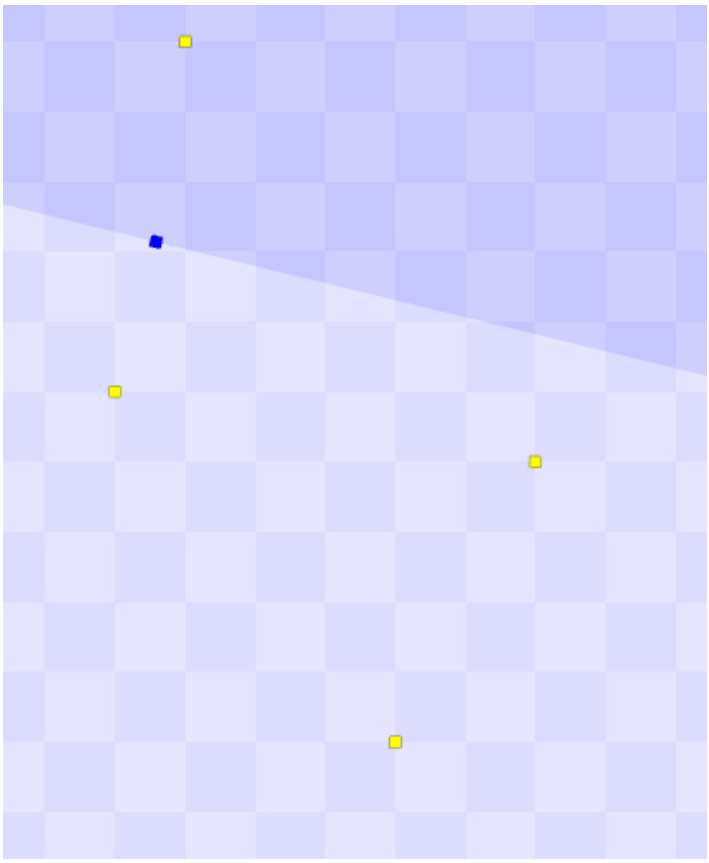


Figura B.2: Tutorial 2 lanzado en el simulador STAGE llegando a la meta.

Como trabajo extra, se probó a introducir 2 robots en el mismo mundo. Uno era este que ya se había creado que iba recibiendo metas y yendo hacia ellas y el otro era uno que se manejaba por el teclado mediante un nodo que daban en la práctica *teleop_twist_keyboard*. De esta manera, se iban moviendo dos robots en el mismo mundo. Esto servía para ir acercándose a lo que sería la implementación del *platooning*. Los 2 robots se pueden ver en la fig B.3. Son el azul (es el que va a las metas que ya estaba antes) y el verde (se mueve por teclado).

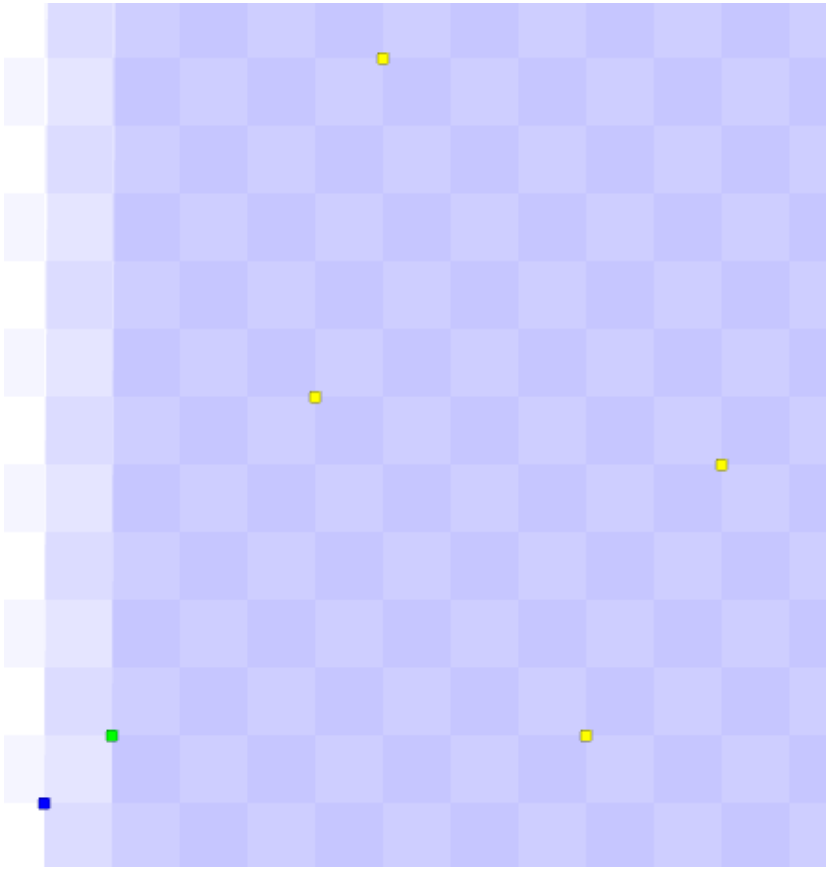


Figura B.3: 2 Robots en el mismo mundo.

## B.2. Aprendizaje del sensor láser

Por otra parte, se tuvo que entender también como funcionaba el sensor láser mediante ROS. El láser es un sensor de rango que mide la distancia a las objetos proporcionando una medida de profundidad. Estas medidas del entorno son proporcionadas mediante ROS por medio de un mensaje que contiene la siguiente información (ver figura B.4). Su funcionamiento se puede resumir en que el escáner del robot va lanzando láseres y va publicando la información en un topic que se llama *scan*.

Este láser funciona de la siguiente manera: escanea desde el angulo mínimo hasta el máximo con una separación entre cada rayo que lanza de *angle_increment* radianes. Estas mediciones que obtiene las guarda en el vector de rangos, es decir, guarda la distancia de cada rayo lanzado al primer objetivo que encuentra.

Una vez entendido el tipo de dato se implementó una detección básica en la que si los 3 rayos centrales del láser detectaban algo lo suficientemente cerca parasen. Esto se añadió al código que se había hecho en los otros tutoriales. De esta manera, el robot iba yendo a cada meta pero si se le ponía un obstáculo delante paraba hasta que desapareciese el obstáculo. Este efecto se aprecia en el vídeo [14] perfectamente.

```
Header header            # timestamp in the header is the acquisition time
                         # of the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your
                         # scanner is moving, this will be used in
                         # interpolating position of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min
                         #or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].
                         # If your device does not provide intensities,
                         # please leave the array empty.
```

Figura B.4: Información de LaserScan.

De esta forma se comprendió correctamente el uso del sensor láser y se hizo un modelo sencillo que después se utilizaría cambiando alguna línea para la detección en el *platooning*.

# Anexos C

# Diagrama de Gantt

Se presenta el cronograma mediante un diagrama de Gantt de las fases de trabajo realizadas. Se han desglosado algunas en sub-fases para su mejor entendimiento.


Figura C.1: Diagrama de Gantt del trabajo realizado.