Universidad Zaragoza
1542

Trabajo de Fin de Grado

“Sliding Balloon” 2D: técnica de navegación para robots terrestres.

“Sliding Balloon” 2D: navigation technique for ground robots.

Autor

Adrián Montero Vitoria

Director

José Luis Villaroel Salcedo

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Año Académico: 2018-2019

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe entregarse en la Secretaría de la EINA, dentro del plazo de depósito del TFG/TFM para su evaluación).

D./Dª. Adrian Montero Vitorria ,en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de (Grado/Máster) Electrónica y Automática (Título del Trabajo)

"Sliding Balloon" 2D: técnica de navegación para robots terrestres.
Sliding Balloon 2D: navigation technique for ground robots.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 21 de Noviembre 2019

Fdo: Adrián Montero Vitoria

Resumen

Este objetivo principal de este proyecto es el desarrollo teórico y experimental de una nueva técnica de navegación 2D conocida como “Sliding Balloon”, cuyo propósito principal es la navegación por túneles y galerías. La navegación se basa en crear un area circular (o globo, de ahí el nombre) que ocupe el máximo espacio del entorno, de forma que el robot avance buscando el centro de dicho globo.

En una primera fase se realiza un estudio de las técnicas de navegación ya existentes, que nos aporta diferentes puntos de vista a la hora de desarrollar nuestra navegación. Junto a ello también se hace una revisión de las herramientas disponibles para la experimentación, con el fin de marcar un punto de inicio para el proyecto.

A continuación se procede al desarrollo teórico de “Sliding Balloon”, es decir la descripción paso a paso del procesamiento de la información realizado por la técnica.

Finalmenete, una vez definida la técnica se procede al desarrollo de los algoritmos y su prueba. Para ello se hace uso de Matlab y ROS, siendo el primero un programa mas sencillo que nos permite realizar las pruebas pertienentes para que la programación sobre ROS sea mas sencilla, ya que este es un programa mas complejo pero muy eficaz para la prueba de comportamiento robot.

Con ayuda de estas dos herramientas y las trazas obtenidas por un sensor de haz plano real somos capaces de probar el correcto funcionamiento de la navegación “sliding Balloon”, cumpliendo así el objetivo principal.

Tabla de contenido

Tabla de figuras

1. Introducción

Este proyecto no está basado en ningún otro trabajo previo, por lo que inicia una nueva temática de estudio. Esta temática entra dentro del amplio marco del proyecto "ROBOCHALLENGE" que se lleva a cabo en la universidad de Zaragoza. En él se plantea la investigación teórica y experimental en el ámbito de la intervención y la exploración de entornos desafiantes mediante robots aéreos y terrestres, donde entorno desafiante se refiere a aquellos lugares donde fracasan o no funcionan correctamente técnicas robóticas actuales o en vías de desarrollo. En estos entornos se pretende demostrar la aplicabilidad de las nuevas técnicas que se desarrollen. El presente TFG afronta el desarrollo de una nueva técnica de navegación en 2D para robot terrestres, cuyo enfoque principal es el tránsito en túneles y galerías

Actualmente existen múltiples técnicas de navegación para robot autónomos móviles, las cuales permiten a estos moverse con total autonomía, dada su capacidad para evitar obstáculos (desde los primitivos algoritmos que detectan un obstáculo y detienen al robot a corta distancia de éste a fin de evitar una colisión, hasta llegar a los algoritmos más sofisticados que le permiten al robot rodear al obstáculo). En operaciones sin supervisión, es decir, autónomas, la planificación de trayectorias es fundamental. El problema es encontrar una trayectoria óptima, libre de colisiones, entre una posición inicial y otra final en un entorno cerrado o acotado.

Fig. 1 "Pioneer P3AT" con sensor laser.

En este trabajo se propone el desarrollo de una innovadora técnica de navegación en tiempo real denominada: *"*Sliding Balloon*"*. La cual recibe este nombre debido a su modo de la planificación de caminos, haciendo uso de un círculo virtual, que es lanzado por delante del robot ocupando el espacio máximo de forma que calcula una trayectoria segura evitando los posibles obstáculos. Esta técnica será probada sobre un escenario real, utilizando un robot "Pioneer P3AT" (Fig. 1).

1.1. Objetivos

El trabajo plantea los siguientes objetivos:

- Desarrollo e implementación de una técnica de navegación de robots denominada "Sliding Balloon" en 2D. Esta técnica se basa en un círculo virtual que modela el espacio de seguridad de movimientos de un robot y que se desplaza por delante del robot estableciendo una trayectoria segura. La técnica usará los datos aportados por un sensor láser de haz plano.
- Desarrollo y prueba de algoritmos 2D en *MATLAB*.
- Implementación de los algoritmos en ROS ("Robot Operating System").
- Se pondrá a punto en simulación con datos reales y se probará en un escenario real con un robot "PIONEER". Los algoritmos utilizados para las pruebas de simulación y la prueba real son los desarrollados en ROS.

1.2. Planificación

Para llevar a cabo los objetivos planteados, se establecen las siguientes etapas en el desarrollo del proyecto:

- En la **fase de análisis del estado del arte** se realiza una revisión bibliográfica de los diferentes factores que son relevantes para la realización del proyecto: análisis de las técnicas de navegación, robot "Pioneer P3AT" con sensor laser de haz plano, Matlab, ROS y el entorno de simulación Gazebo.
- En la **primera fase** del proyecto se realiza el desarrollo de los algoritmos 2D, que recrean el comportamiento de la técnica de navegación, sobre Matlab (entorno de programación). Para ello se requerirán técnicas de tratamiento de nubes de puntos.
- En la **segunda fase** del proyecto los algoritmos obtenidos en la primera fase se implementaran en ROS, desarrollando así una técnica de navegación robusta y utilizable por una gran variedad de plataformas robots.
- En la **tercera fase** se realizará la puesta a punto sobre el entorno de simulación Gazebo haciendo uso de datos reales y finalmente se probara la técnica de navegación sobre un escenario real (robot "Pioneer").
- En la **última fase** del proyecto se interpretaran los datos obtenidos, se obtendrán conclusiones y la viabilidad de implementación de algoritmos futuros.

2. Análisis del estado del arte

2.1. Análisis de trabajos previos

El propósito principal de cualquier técnica de navegación para robots autónomos es alcanzar el destino final, evitando por el camino cualquier obstáculo que pueda aparecer. El diseño de una buena técnica de navegación requiere de dos competencias principales: planificación y reacción [1]. La planificación nos otorga un objetivo y una trayectoria a seguir para alcanzar dicha meta, mientras que la reacción nos permite modificar la trayectoria ante los problemas que surjan durante el desplazamiento debido a eventos desconocidos (como pueden ser obstáculos). La planificación es un campo ampliamente estudiado previo a la aparición del robot autónomo, dado su interés para la planificación de robots manipuladores, por contrario la evitación de obstáculos aparece por necesidad para la realización de tareas que pueden ser realizadas por robots autónomos móviles.

Por ello vamos a realizar un pequeño análisis de algunas de las técnicas de navegación con evitación de obstáculos existentes, las cuales pueden depender de diferentes tipos de sensores (ultrasónicos, telémetro láser, cámara de vídeo).

- **"Bug Algorithm":** Probablemente el método más simple, el cual consiste en rodear los obstáculos que aparezcan a lo largo de la trayectoria. Existen dos versiones, la primera rodea completamente el obstáculo y después se separa desde el punto más cercano al objetivo, en la segunda versión el robot sigue el contorno del obstáculo y se separa de este cuando puede avanzar directamente hacia el objetivo. [1]

- **Campos de potencial:** Este método se basa en una serie de fuerzas imaginarias que actúan sobre el robot. La fuerza aceleradora final es el resultado de sumar las fuerzas repulsivas ejercidas por los obstáculos y la fuerza atractiva del objetivo. Mediante una formula gaussiana se ponderan incidencia de cada elemento para crear un campo de fuerzas virtual. La dirección a tomar se obtiene mediante el cálculo del gradiente, el cual determina el camino con mayor pendiente. De esta forma podemos decir que el robot se mueve a través del campo virtual de fuerzas como un balón en una pendiente, es decir buscando el camino en el cual las fuerzas ejercidas sobre este son las menores del mapa. [1][2]

- **Campo de fuerzas virtuales (VFF):** El VFF ("Virtual Force Field") permite un control robusto de movimiento continuo. Este método hace uso de una grilla-histograma actualizada en el tiempo, en la cual se representa el nivel de certeza, que es el nivel confianza que el algoritmo tiene de que existe un obstáculo en esa ubicación. Para controlar el movimiento del vehículo, se hace uso del método de campo de potencial a través de las lecturas que realiza el sensor sobre un pequeño área de la grilla-histograma (ver Fig 2). En cada lectura solamente se incrementa una celda en la grilla-histograma, creando una distribución de probabilidad de pequeño gasto computacional. Así pues, la información es actualizada en la grilla lo antes posible, de forma que el próximo cálculo de la fuerza resultante tome en cuenta estos datos, ambos cálculos son asíncronos. Haciendo uso de ambos conceptos se obtiene una

navegación capaz de responder ante obstáculos desconocidos (método reactivo) pudiendo ser usado para altas velocidades. Pero este método tiene unos problemas los cuales son difíciles de solucionar, cuando hay baja visibilidad entre dos obstáculos, la naturaleza discreta de la grilla-histograma y la navegación por largos pasillos puede generar un movimiento oscilatorio e inestable por las fuerzas ejercidas por las paredes.[1][3]

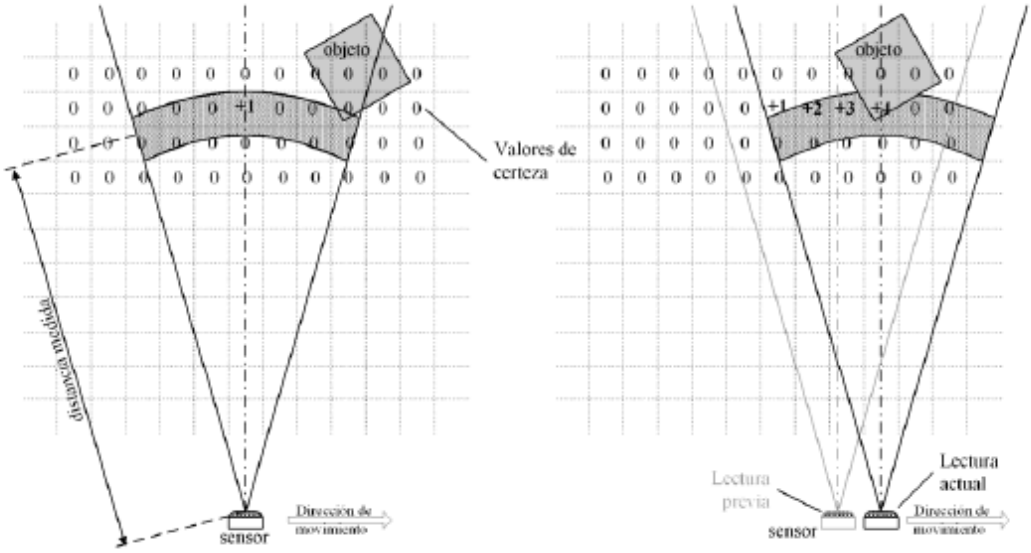

Fig. 2 Grilla-Histograma

- **Histograma del campo vectorial (VFH):** Los problemas del método VFF derivan de una reducción excesiva de datos, por ello se desarrolla un nuevo método denominado VFH ("Vector Field Histogram") que realiza una reducción de datos en dos estados, lo cual, es más simple que la técnica empleada por VFF. Además existen tres niveles de representación de los datos.
  a) El nivel más alto contiene la información detallada del entorno del robot. La grilla-histograma es actualizada continuamente, la diferencia es que en este método es absoluta. Sin embargo, con el movimiento del robot también se mueve una pequeña ventana cuadrada que se superpone a la grilla-histograma, denominada región actica.
  b) En el nivel intermedio se crea un histograma polar alrededor del centro momentáneo del robot, el cual comprende n sectores de ancho α. El eje-x representa el ángulo en el cual ha sido encontrado un obstáculo y el eje-y representa la probabilidad de que realmente halla un obstáculo en esa dirección.
  c) En el último nivel se obtienen los datos de salida que son los valores de dirección y velocidad del vehículo. Basándose en la gráfica de Densidad Polar de Obstáculos (ver Fig. 3) y en el valor umbral obtenemos estas referencias, las zonas que superen el valor umbral se consideran prohibidas mientras que las zonas que no superen este valor se consideran zonas transitables.

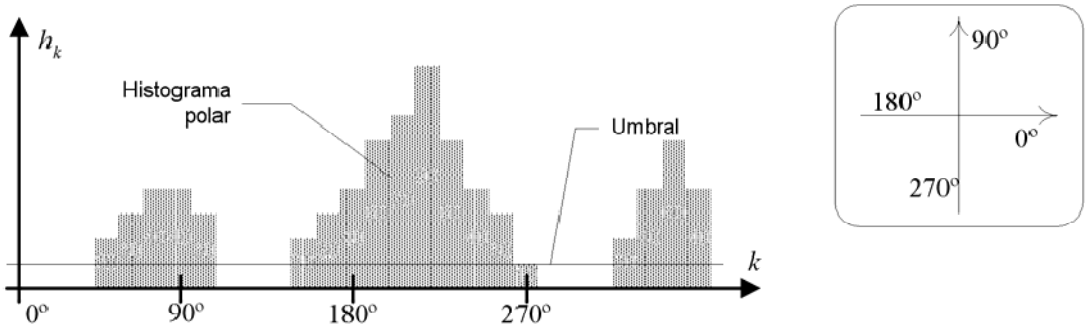

Fig. 3 Gráfica de Densidad Polar de Obstáculos

Pero este método también presenta algunas desventajas que son la velocidad de procesamiento está relacionada con la velocidad máxima del robot y la necesidad de gran capacidad de almacenamiento. [4][5]

- **"Nearnes Diagram" (ND):** Este método tiene por objetivo encontrar la región de mayor tamaño, entre los obstáculos detectados por el sensor, en la dirección principal de movimiento. Mediante el sensor se detectan dichos obstáculos dentro de un radio, para así identificar los huecos formados entre ellos y la distancia mínima que los separa, de forma que podemos asegurar si el robot cabe por el hueco. Esta técnica hace uso de dos herramientas para analizar la relación entre el robot, la distribución de los obstáculos y la meta, además el espacio es dividido en n sectores de anchura α. Las herramientas son el PND y el RND. El PND representa la cercanía de los obstáculos medida desde el centro del robot y el RND representa la cercanía de los obstáculos medida desde el extremo del robot. Haciendo uso de las herramientas se detectan huecos, que se producen por una discontinuidad entre dos sectores, regiones, que son dos huecos consecutivos y por último el área transitable, que es la región transitable más cercana al objetivo (ver Fig. 4). [6]

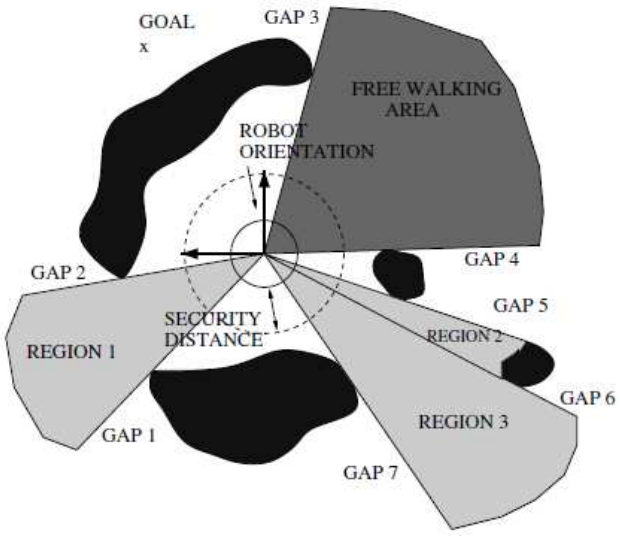

Fig. 4 Huecos, regiones y área transitable calculadas

- **“Bubble band technique”:** Este método se basa en una banda elástica para modelar una trayectoria global hasta el objetivo y en una serie de “burbujas” a lo largo de esta trayectoria, las cuales definen el espacio libre de movimientos para el robot. La planificación produce una trayectoria global evitando los diferentes obstáculos conocidos, utilizando para ello dos fuerzas, la fuerza de contracción y la fuerza externa de repulsión. Estas fuerzas también permiten manejar posibles cambios en el entorno del robot, como obstáculos que provocaran el cambio de la banda elástica hasta una nueva posición de equilibrio. Las burbujas representan subconjuntos de espacio libre para el robot, obtenidos por una función cuyo resultado es la distancia mínima entre el robot y el obstáculo. Por tanto, tal como se puede ver en la Fig. 5, la banda está representada por una serie de burbujas, estas burbujas se superponen con sus vecinas para asegurar que, mientras que el camino a seguir esté dentro de estas, el robot no sufrirá ningún choque.

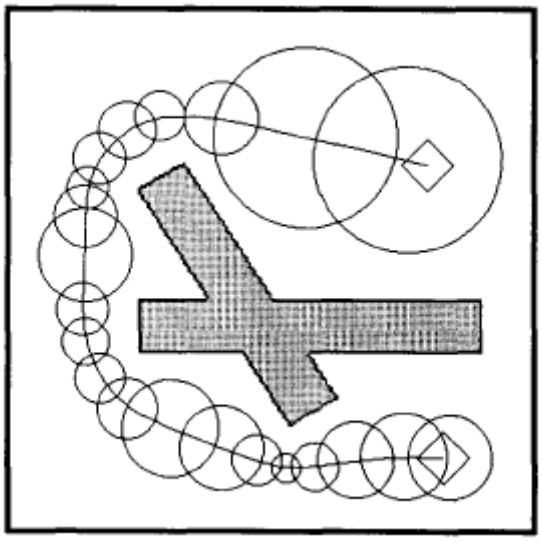

Fig. 5 Burbujas generadas a lo largo de la trayectoria (banda elástica)

Después de esta breve introducción sobre algunas de las técnicas de navegación podemos concluir que el movimiento del robot es el resultado de la lectura actual o previa de los sensores disponibles, su objetivo y la posición relativa del robot.

2.2. Descripción de recursos

En este apartado se expondrán los componentes y herramientas utilizadas para el desarrollo de este Trabajo de Fin de Grado.

2.2.1. Hardware

- **Ordenador:** Se considera el nodo central del sistema, en él se ejecutaran todas las simulaciones de los algoritmos desarrollados en Matlab, se implementaran estos algoritmos en ROS y se realizará la puesta a punto con datos reales en el simulador Gazebo. El ordenador es de la maraca ASUS modelo F555L, con un procesador Intel Core i5 y 8GB de RAM.

- **"Pioneer P3AT":** es una plataforma robótica todo terreno altamente versátil, que permite un alto grado de personalización. Es un robot popular para proyectos en exteriores o en terrenos difíciles.

2.2.2. Software

- **Matlab:** Es una herramienta de software matemático con un entorno de desarrollo integrado (IDE) y con un lenguaje de programación propio que dispone de numerosas aplicaciones desde resolución de problemas matemáticos, representación de datos y funciones, implementación de algoritmos y comunicación con otros lenguajes y dispositivos. En este TFG se utilizará la *toolbox* para tratamiento de nubes de puntos.

- **ROS ("Robot Operating System"):** ROS, es un marco de código abierto para la programación de robots, sirve como plataforma de "software" para aquellos que realizan proyectos o actividades con robots. Consiste en una serie de herramientas, librerías y convenciones cuyo objetivo es simplificar la creación de comportamientos robots robustos y complejos. Este "software" fácilmente se puede implementar en cualquier lenguaje de programación moderno, como son *C++*, *Python*, *Lisp*, *Java* o *Lua*. Por lo que, al eliminar lenguajes exclusivos de cada marca, ayuda a que pueda ser utilizado por un mayor número de robots y al ser "software" libre, está en constante evolución y desarrollo. Las ventajas de utilizar este método de programación son numerosas ya que permite que el mismo código pueda ser utilizado por diferentes robots de diferentes marcas, dotado de una gran cantidad de herramientas para depurar, visualizar y simular (Rviz, Gazebo).

- **Gazebo:** Es un simulador robot que permite testear rápidamente algoritmos, diseño de robots, entrenar sistemas de inteligencia artificial, etc. Todo ello utilizando escenarios realistas, tanto interiores como exteriores. Es decir, Gazebo es un motor de físicas robusto, con gráficos de alta calidad, interfaces programables y gráficas convenientes.

- **RVIZ:** Es una herramienta de visualización que permite recopilar y mostrar información tanto del robot como de sus sensores. RVIZ recoge la información de los *messages* publicados en los *topics* usados en el programación en ROS.

3. Técnica de navegación: “Sliding Balloon”

3.1. Idea fundamental

La técnica de navegación que se presenta, se denomina “Sliding Balloon”, que traduciendo literalmente al español significa Globo Deslizante. El nombre de esta técnica viene dado por su semejanza con un globo, el cual tiene la capacidad de inflarse y desinflarse. Su utilización está centrada esencialmente a espacios cerrados en los que puede haber obstáculos puesto que se trata de una navegación reactiva, aunque también se podría utilizar en espacios abiertos.

El algoritmo crea una trayectoria segura para el robot a través de un área circular (el globo), que avanza un paso por delante del robot buscando la dirección de avance más segura. Este área aumenta o se reduce ocupando el máximo espacio posible hasta encontrar dos puntos del entorno dentro del círculo virtual. De esta forma, el camino consiste en avanzar repetidamente en dirección hacia el centro del próximo área circular válido. Sin embargo, si el radio de la siguiente circunferencia es inferior a un valor establecido el robot se detiene, puesto que se produciría un choque del robot con el entorno.

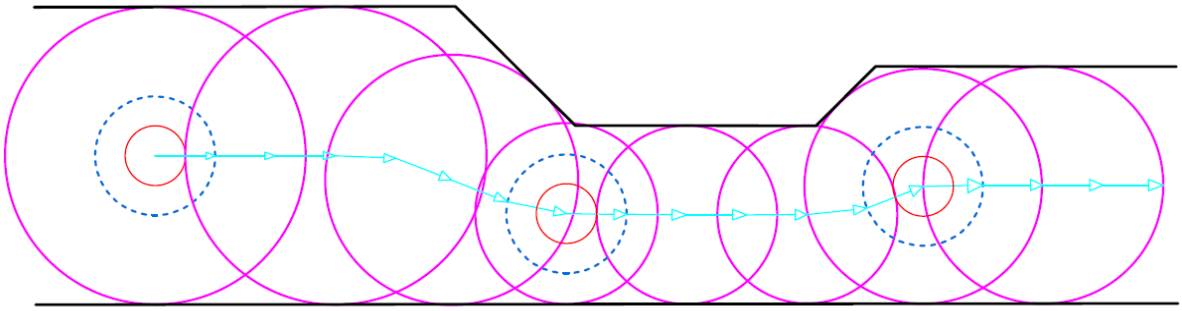

Fig. 6 Ejemplo simplificado de la técnica “Sliding Balloon”.

En la fig. 6 se muestra un esquema simplificado del funcionamiento de esta técnica. Se presentan 5 colores: el rojo que representa al robot, el circulo azul con línea discontinua representa los diferentes puntos alcanzables por el robot (circunferencia con radio la distancia de avance), el color magenta representa el globo, en azul claro está la dirección que sigue en cada paso de avance el robot y en negro el entorno Observamos como el área circular en color magenta aumenta o disminuye en función del espacio disponible, por lo que el centro también varía de posición, lo cual hace cambiar la dirección del robot.

3.2. Funcionamiento: paso a paso

En este subapartado se procede a explicar más detalladamente el funcionamiento principal de la técnica "Sliding Balloon".

1. Calculamos el objetivo (**P_o** , cuya posición será precisada en los siguientes pasos) mediante la dirección inicial (**Dir_{avance}**), el punto de partida del robot (**P_P**) (o la última dirección) y el radio de avance **R_{avance}** como observamos en la ecuación (3.1).

$$P_O = P_P + \overrightarrow{Dir_{avance}} * R_{avance} \tag{3.1}$$

2. Buscamos el punto más cercano del entorno al objetivo calculado en el paso anterior (**$P_{Nearest}$**, ecuación (3.3)) Distancia entre ambos puntos es igual al radio **$R_{Balloon}$** del globo o area de seguridad, como vemos en la ecuación (3.2).

$$R_{Balloon} = \min\left(\left\|\overrightarrow{P_O P_i}\right\|\right) \ \forall \ P_i \ \in Puntos_{Entorno} \tag{3.2}$$

$$P_{Nearest} = \left\{ P_i \mid \min\left(\left\|\overrightarrow{P_O P_i}\right\|\right) \right\} \forall \ P_i \ \in Puntos_{Entorno} \tag{3.3}$$

3. Comprobamos si este punto es válido, calculando si dentro de la circunferencia de radio **$R_{Balloon}$** (mas un pequeño diferencial, ecuación (3.4)) y centro en **P_o**, hay otro punto del entorno además del más cercano como muestra la ecuación 3.5.

$$R_{Balloon} \ = \ R_{Balloon} + Diferencial \tag{3.4}$$

$$Puntos_{\ Rdif} = Puntos_{Rdif} \ \cup P_{IN}$$
$$P_{IN} = \left\{ P_j \mid \left\|\overrightarrow{P_O P_j}\right\| < R_{Balloon} \right\} \forall \ P_j \ \in Puntos_{Entorno} \tag{3.5}$$

4. **P_{IN}** y **$P_{Nearest}$** deben encontrarse en lados contrarios de la dirección de avance (**Dir_{avance}**), como se muestra en las ecuaciones (3.6), (3.7) y (3.8), esta operación se repite para todos los puntos que del conjunto **$Puntos_{Rdif}$** (ecuación (3.5)). Si ambos puntos se encuentran en distinto lado con respecto del vector **Dir_{avance}** continuamos en el paso 6, de lo contrario corregiremos la posición.

$$\overrightarrow{v_1} = \overrightarrow{P_O P_{Nearest}} \ , \ \overrightarrow{v_2} = \overrightarrow{P_O P_{IN}} \tag{3.6}$$

$$\overrightarrow{Xv_1} = \overrightarrow{Dir_{avance}} \ x \ \overrightarrow{v_1} \ , \ \overrightarrow{Xv_2} = \overrightarrow{Dir_{avance}} \ x \ \overrightarrow{v_2} \tag{3.7}$$

$$Distinto \ lado \ \Leftrightarrow \ \overrightarrow{Xv_1}[2] \ * \ \overrightarrow{Xv_2}[2] < 0 \tag{3.8}$$

5. Corrección del objetivo:
   5.1. Con centro $\mathbf{P_{Nearest}}$ y radio $\mathbf{R_{Balloon}}$ (ecuación (3.4)), calculamos un nuevo $\mathbf{P_O}$ mediante la intersección de este círculo y el círculo con radio $\mathbf{R_{avance}}$ y centro en $\mathbf{P_P}$ (corte entre dos circunferencias origina dos puntos, que se obtienen de despejar la ecuación (3.9) y la ecuación (3.10)). Entre los dos puntos obtenidos tomamos como correcto el más próximo al objetivo previo, para ello hacemos uso de la ecuación 3 pero en este caso el conjunto está compuesto únicamente por dos puntos $\mathbf{P_1}$ y $\mathbf{P_2}$.

$$(x - P_{Nearest}[0])^2 + (y - P_{Nearest}[1])^2 = {R_{Balloon}}^2 \quad (3.9)$$

$$(x - P_P[0])^2 + (y - P_P[1])^2 = {R_{avance}}^2 \quad (3.10)$$

   5.2. Repetimos la operación del paso 4, comprobamos si $\mathbf{P_{IN}}$ y $\mathbf{P_{Nearest}}$ se encuentran en lados distintos con respecto de $\mathbf{Dir_{avance}}$.
   5.3. Si se encuentran en el mismo lado se repite el punto 5 hasta obtener el objetivo correcto que cumpla las condiciones del paso 4.
6. Finalmente antes de dar por válido el objetivo comprobamos que con centro en $\mathbf{P_o}$, no hay ningún punto del entorno dentro del radio de seguridad ($\mathbf{R_{seguridad}}$), para ello hacemos uso de nuevo de la ecuación (3.5).

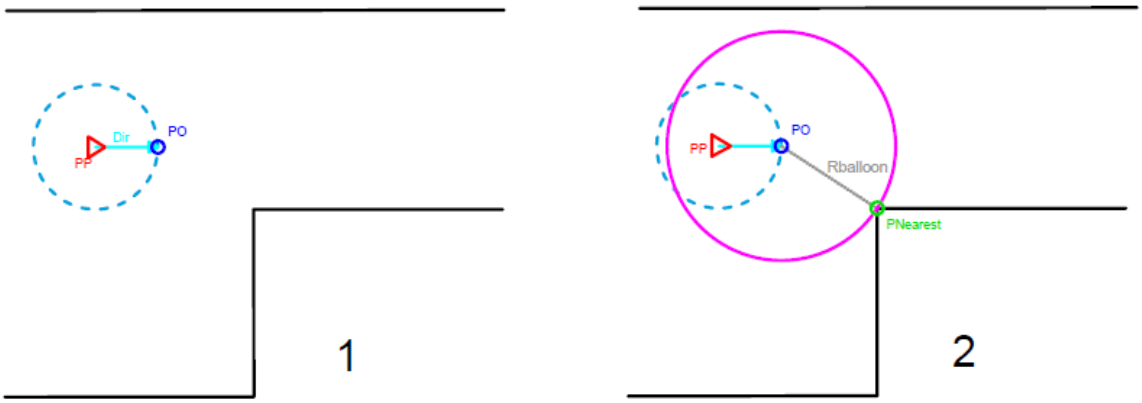

Fig. 7 "Sliding Balloon" paso a paso I

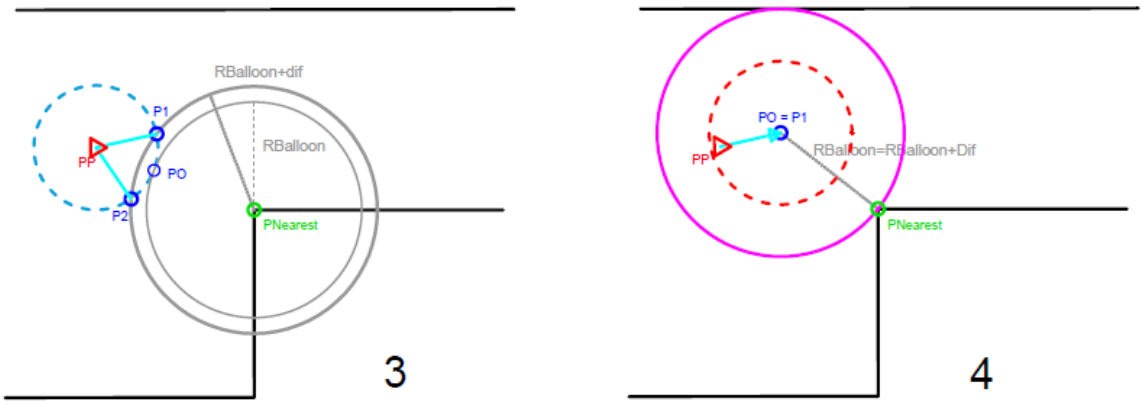

Fig. 8 "Sliding Balloon" paso a paso II

En la Fig. 7 y 8 está representado cada uno de los pasos de la técnica "Sliding Balloon". En color negro tenemos el entorno del robot, en rojo la posición del robot (PP) y el área de seguridad (círculo con línea discontinua), en azul claro la dirección que toma el robot (la cual se va actualizando) y el radio de avance del robot, en azul oscuro (PO, P1,P2) la meta a alcanzar (la cual también se actualiza paso a paso), en magenta el globo, en verde el punto más cercano al objetivo calculado y en gris elementos utilizados en el método.

En la imagen 1 de la Fig. 7 se encuentra el cálculo del punto objetivo provisional (Paso 1). En la imagen 2 se comprueba si el punto calculado en el paso anterior es correcto (Paso 2, 3 y 4). Sobre la imagen 3 de la Fig. 8 se calculan los dos puntos que se originan de la intersección entre dos circunferencias (Paso 5) y finalmente en la imagen 4 comprobamos que el nuevo PO, elegido entre P1 y P2 (siendo este el más próximo al anterior PO), es correcto y que no hay ningún punto dentro del área de seguridad (Paso 6).

3.3. Versiones implementadas

Las versiones que se desarrollan en este TFG dependen de la información que se tiene del entorno, es decir, si se conoce el lugar dado un mapa y la posibilidad de que haya o aparezcan obstáculos. Por tanto estamos hablando de planificación previa y de navegación en tiempo real. La diferencia entre ambas, es que el primero recibe inicialmente una nube de puntos la cual conforma el mapa del entorno que rodeará al robot. De esta forma el algoritmo desarrolla una trayectoria con anterioridad al comienzo de la marcha. Con este funcionamiento suponemos un entorno controlado en el que se conocen datos del entorno. Mientras que en el segundo se desconoce la posibilidad de que aparezcan obstáculos, por tanto depende únicamente de las lecturas de sus sensores para recibir información sobre el entorno.

En ambos casos se utilizan las dos mismas técnicas para el cálculo de la posición:

1. **“Sliding Balloon”**: Esta técnica es la principal y da nombre a este trabajo, la cual se basa en la idea fundamental anteriormente explicada.
2. **“Wall Sliding Balloon”**: Esta técnica, derivada de la idea fundamental, calcula el camino seguro manteniendo una distancia constante con una de las dos paredes. El robot recibe una distancia que determina el radio del globo, busca la pared deseada y finalmente comienza a avanzar. Durante todo el trayecto el globo está en contacto con la pared elegida.

- Paso a paso de “Wall Sliding Balloon”:

Inicialmente se fija un radio (**R_{dist}**) que determinará la distancia constante a la pared elegida (izquierda o derecha). Lo primero que se debe realizar es un posicionamiento, es decir, colocar el robot próximo a la pared elegida. A partir de este momento el funcionamiento es similar a la técnica fundamental.

1. Calculamos el objetivo (**P_o** , cuya posición será precisada en los siguientes pasos) mediante la dirección inicial (**Dir_{avance}**), el punto de partida del robot (**P_P**) y el radio de avance **R_{avance}** como observamos en la ecuación (3.1) (o en su defecto la última dirección tomada, la posición del robot y el radio de avance , si ya estamos en movimiento).
2. Corrección del objetivo
   2.1. Buscamos el punto más cercano del entorno (**$P_{Nearest}$**) al objetivo calculado previamente, ecuación (3.2), y comprobamos que este puto se encuentre en el lado correcto del sentido de avance, derecho o izquierdo, ecuación (3.11) y (3.12).

$$\overrightarrow{v_1} = \overrightarrow{P_{Nearest}P_O} \ , \ \overrightarrow{Xv_1} = \overrightarrow{Dir_{avance}} \ x \ \overrightarrow{v_1} \tag{3.11}$$

$$lado\ derecho \Leftrightarrow \overrightarrow{Xv_1}[2] < 0, \ lado\ izquierdo \Leftrightarrow \overrightarrow{Xv_1}[2] > 0 \tag{3.12}$$

   2.2. Si no es así giramos el vector dirección 2 grados sobre el eje z, con sentido de giro hacia la pared deseada como muestra la ecuación (3.13).

$$\overrightarrow{Dir_{avance}} = Matriz_{giro} * (\overrightarrow{Dir_{avance}})^t \tag{3.13}$$

   2.3. Se repite el proceso del punto 2.1 y 2.2 hasta conseguir que **$P_{Nearest}$** se encuentre en el lado correcto con respecto a **Dir_{avance}**.
   2.4. Una vez el punto se encuentra en el lado deseado, calculamos la intersección entre ambas circunferencias, teniendo el primero centro en **$P_{Nearest}$** y radio la distancia fijada al comienzo (**R_{dist}**), mientras que la segunda circunferencia tiene centro en la **P_P** y radio **R_{avance}**
3. Finalmente, antes de dar por válido el objetivo, comprobamos que con centro en **P_o**, no hay ningún punto del entorno dentro del radio de seguridad (**$R_{seguridad}$**), para ello hacemos uso de nuevo de la ecuación (3.5).

4. Algoritmos en Matlab

Una vez explicados los principales conceptos que se van a emplear a la hora de desarrollar esta nueva técnica de navegación, hacemos uso de la herramienta de programación de Matlab la cual nos proporciona un marco flexible y sencillo para programar y poder probar los algoritmos. Los programas aquí obtenidos nos permiten estudiar con mayor facilidad las dificultades a la hora de realizar la navegación.

Para el desarrollo de los algoritmos correspondientes a las versiones que se van a implementar, comentadas en el apartado anterior, se han utilizado las diferentes toolbox ofrecidas por Matlab. Junto con los ficheros principales se han definido diferentes funciones como `SameSide`, `SolveCirclesInterseccion`, `OnTheRightSide` o `NearestSol` que nos permiten computar las condiciones y resolver las ecuaciones del apartado 3, anexo 8.1.5.

En los siguientes subapartados, cada una de las técnicas se explica en detalle tanto con planificación previa como en tiempo real y se muestran imágenes que muestran el comportamiento de los algoritmos.

4.1. Planificación previa

Para el correcto funcionamiento de este tipo de navegación suponemos un entorno controlado, por lo que si el programa recibe un mapa a través de una nube de puntos, el robot no necesita de información adicional sobre posibles obstáculos que no estén reflejados en este mapa. De esta forma podemos calcular la totalidad de la trayectoria con seguridad de que el entorno no cambiará.

Para la lectura del mapa se hace uso de la "toolbox Computer Vision" [12] que ofrece Matlab para procesar archivos de nubes de puntos (Fig. 9). La función más usada de esta toolbox es `findNearestNeighbors` [13], permite ordenar en función de la distancia los puntos del entorno.

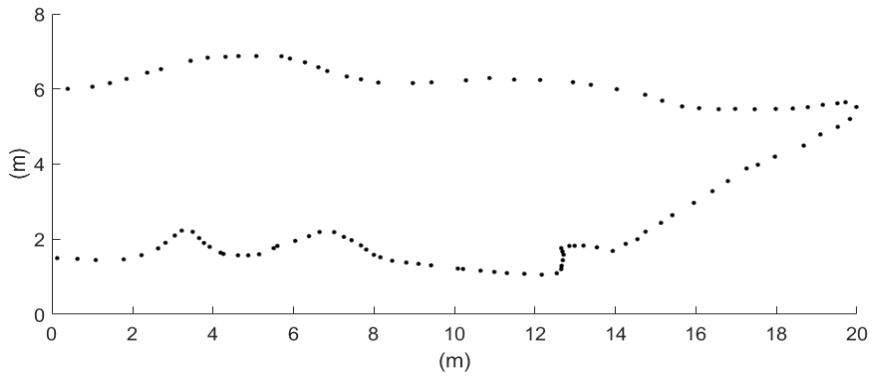

Fig. 9 Ejemplo de mapa formado por nube de puntos

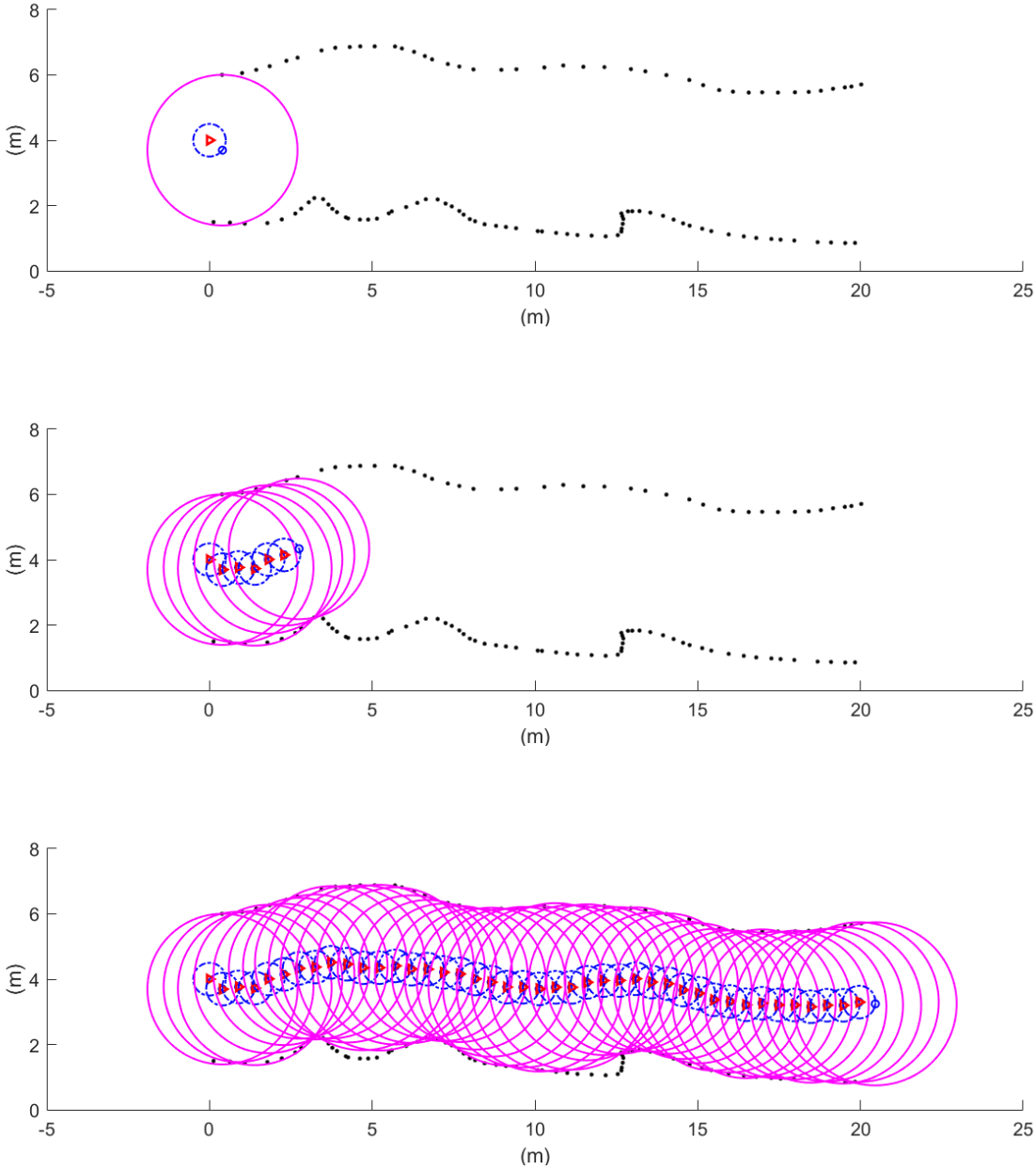

Fig. 10 "Sliding Balloon" con Planificación previa

- **"Sliding Balloon"** con planificación previa: En la Fig. 10 tenemos en tres gráficas que muestran como la trayectoria es obtenida por la técnica "Sliding Balloon". Como observamos en la primera gráfica, el algoritmo muestra el mapa con la información obtenida de la nube de puntos representado por los puntos negros, la posición del robot a través del triángulo rojo, el globo o área libre de obstáculos con la línea magenta y el destino de cada iteración con el pequeño círculo azul contenido en el radio de avance del robot representado con el círculo azul de línea discontinua. En las siguientes dos gráficas podemos comprobar como el algoritmo busca el máximo espacio disponible aumentando y disminuyendo el globo o área libre de obstáculos,

adaptándose en todo momento a la forma del entorno. Al final de la galería el robot se detiene puesto que gracias a la información contenida en la nube de puntos sabe cuando ha alcanzado el punto más alejado. Para revisar el código ver el anexo 8.1.1.

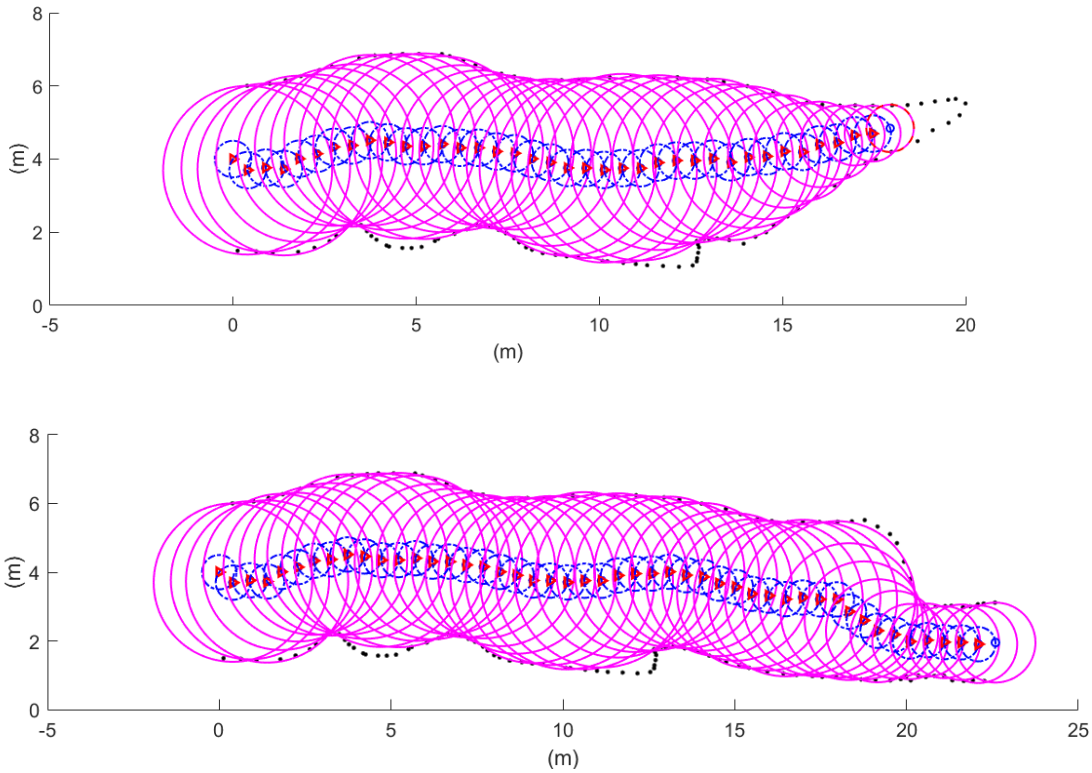

Fig. 11 Planificación Previa en diferentes galerías usando "Sliding Balloon"

En la Fig. 11 podemos ver de nuevo el resultado final de aplicar el algoritmo "Sliding Balloon" en dos nuevas galerías, creadas para probar el comportamiento en diferentes situaciones en función del espacio disponible. En el primer caso vemos como la galería se estrecha en la parte final hasta cerrarse completamente, de forma que la navegación detecta que el dentro del radio de seguridad se encuentra un punto del entorno por lo que detiene el avance del robot, el radio de seguridad del robot se representa con una línea roja discontinua. Sin embargo en el segundo caso observamos como la galería se estrecha pero no llega a cerrarse en ningún momento, por lo tanto sigue habiendo espacio disponible para avanzar de forma segura y por tanto el robot continúa su avance.

- **"Wall Sliding Balloon"** con planificación previa: Para mostrar un ejemplo se ha elegido la pared derecha de la galería como lado deseado. En la Fig. 12 observamos de nuevo como se lleva a cabo la planificación previa en la totalidad del mapa. Igualmente el magenta representa el globo, el triángulo rojo la posición del robot en el mapa, en negro los puntos del mapa, y en azul la meta y el radio de avance. Para el "Wall Sliding

Balloon" vemos como primero busca acercarse a la pared deseada para luego comenzar el avance hasta el final del mapa imitando el relieve de la pared casi a la perfección, el globo tiene en todo momento un radio constante manteniendo así una distancia fija con la pared. Todo el código queda reflejado en el anexo 8.1.2.

Puesto que estamos limitando la distancia a una pared, la distancia a la pared contraria no se tiene en cuenta, únicamente cuando se detecta un punto dentro del radio de seguridad se detiene el movimiento.

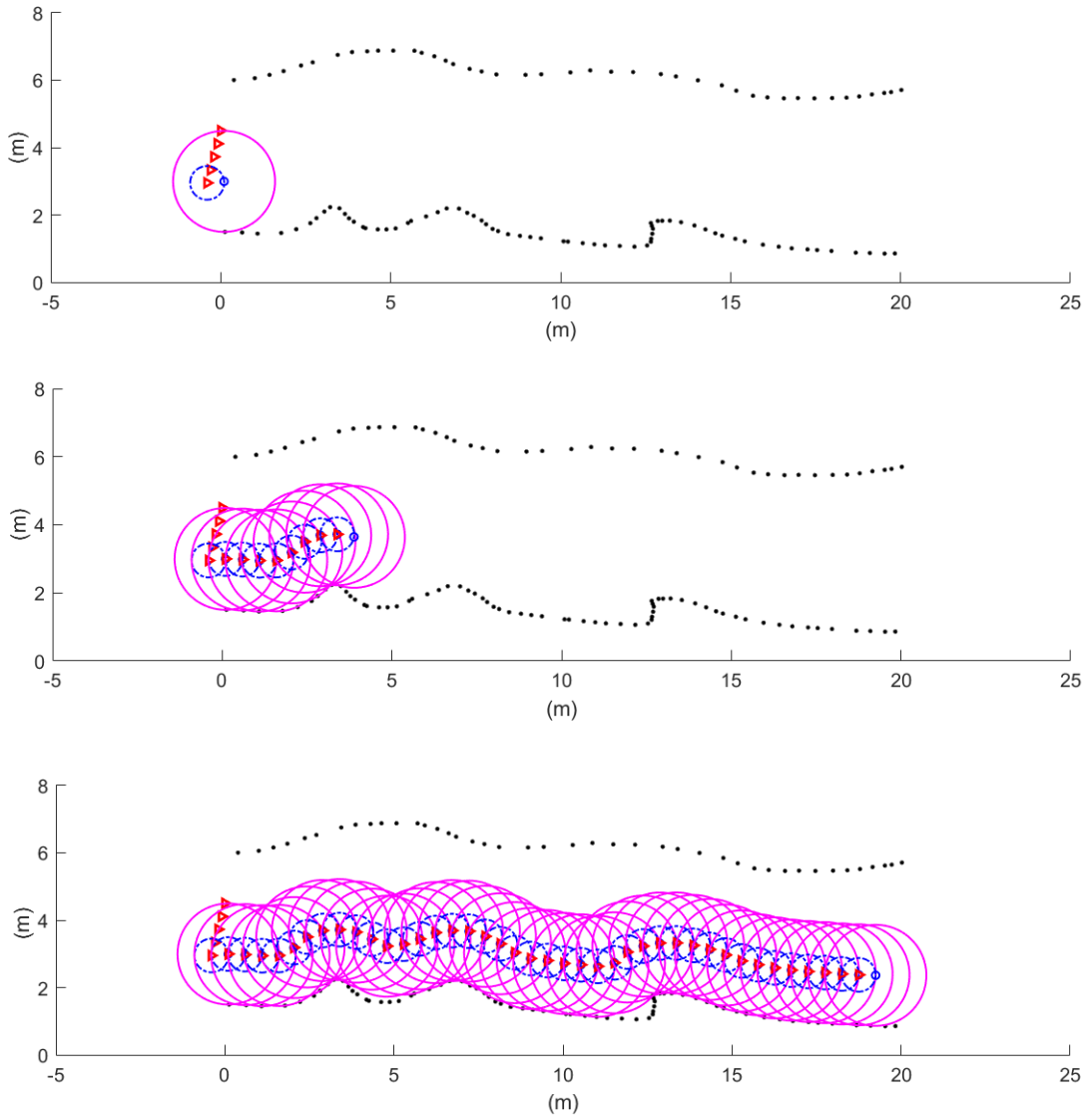

Fig. 12 "Wall Sliding Balloon" con planificación previa

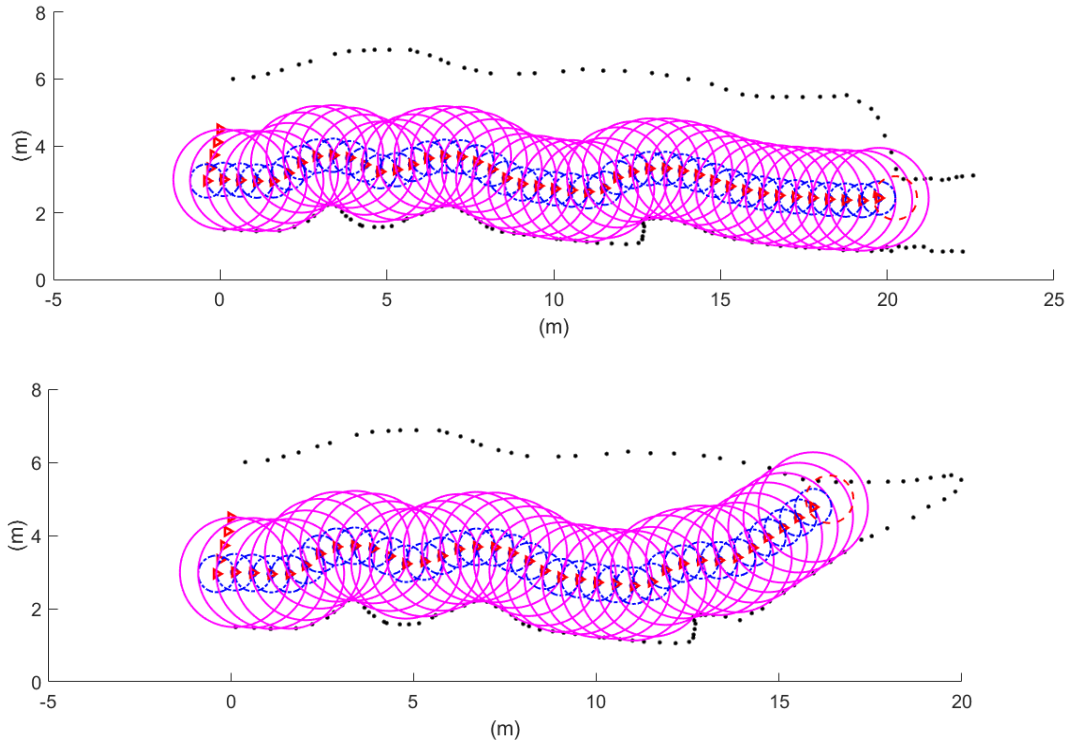

Fig. 13 Planificación Previa en diferentes galerías usando "Wall Sliding Balloon"

Podemos observar en la Fig. 13 el resultado final de aplicar el algoritmo "Wall Sliding Balloon" en dos nuevas galerías, creadas para probar el comportamiento en diferentes situaciones en función del espacio disponible. En ambas galerías vemos como el movimiento se produce correctamente, manteniendo la distancia constante. Sin embargo, el robot se detiene en ambas galerías cuando se detecta un punto dentro del radio de seguridad, el cual la mayoría de los casos se encontrará en el lado contrario a la pared elegida o por delante del robot. A diferencia del "Sliding Balloon" que sí que es capaz de avanzar a través de la galería de la primera gráfica, con el "Wall Sliding Balloon" el robot no es capaz de pasar por el reducido espacio. Cambiando las condiciones de distancia sobre la pared, el robot alcanzaría el final.

4.2. Tiempo real

En este caso no se tiene información sobre el entorno, o si se conoce, no se puede asegurar que haya o aparezcan nuevos obstáculos. De forma que para el correcto funcionamiento se deben incluir sensores capaces de adquirir información del espacio que rodea al robot, en nuestro caso tenemos un sensor láser de haz plano. Para realizar las pruebas en tiempo real en Matlab, puesto que no se realiza una conexión directa con el robot, se ha recogido con anterioridad una serie de lecturas realizadas por el sensor en un entorno real, en este caso las trazas se corresponden con las del túnel de Canfranc. Por consiguiente si el algoritmo es capaz de realizar su trayectoria con los datos que recibiría en tiempo real podremos asegurar que ante una prueba real el robot se comportará igual que en la simulación.

El laser recoge información continuamente, estas lecturas son transformadas a nubes de puntos creando así un mapa local del entorno, con el cual somos capaces de trabajar usando la "toolbox Computer Vision" [12] de Matlab, mencionada anteriormente. Este mapa se crea mediante la función `CreateMapa` (anexo 8.1.5), la cual para cada punto calcula sus coordenadas dados el ángulo y la distancia, datos recogidos por el sensor, con respecto del robot.

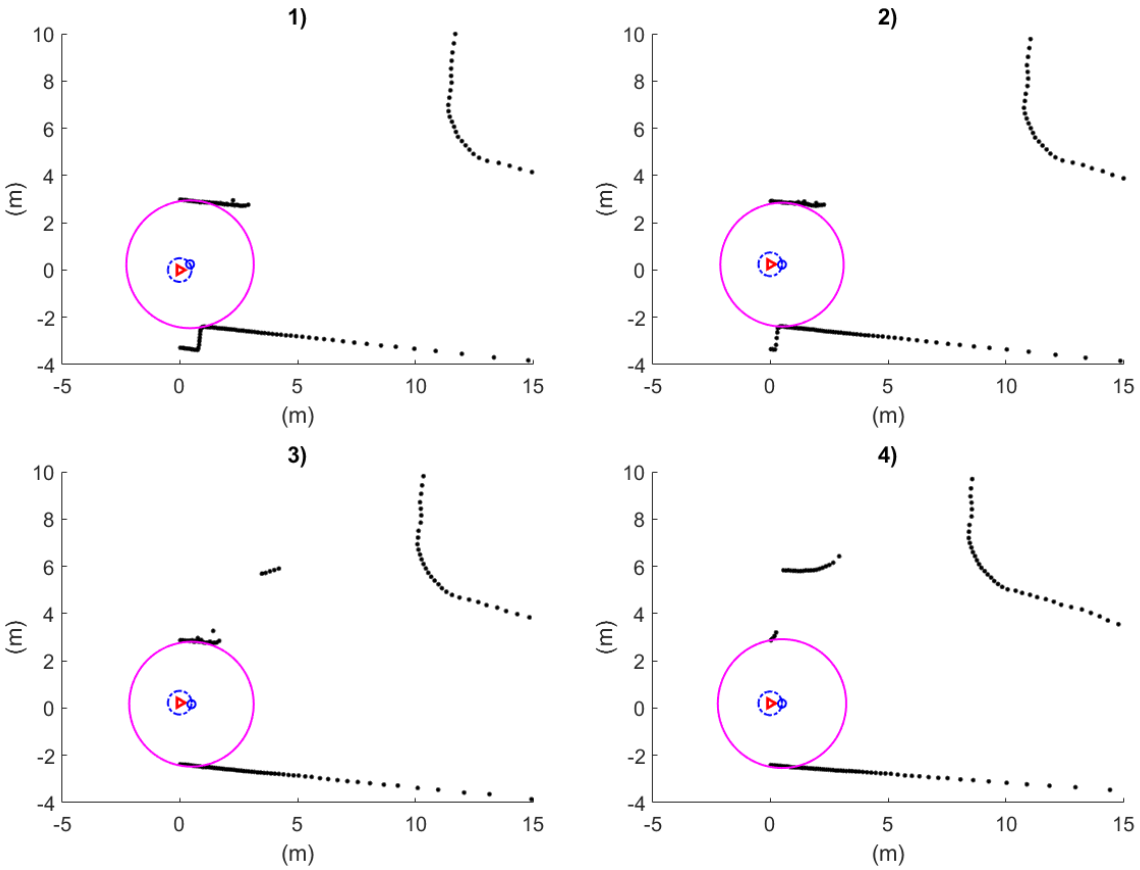

Fig. 14 "Sliding Balloon" en tiempo real I

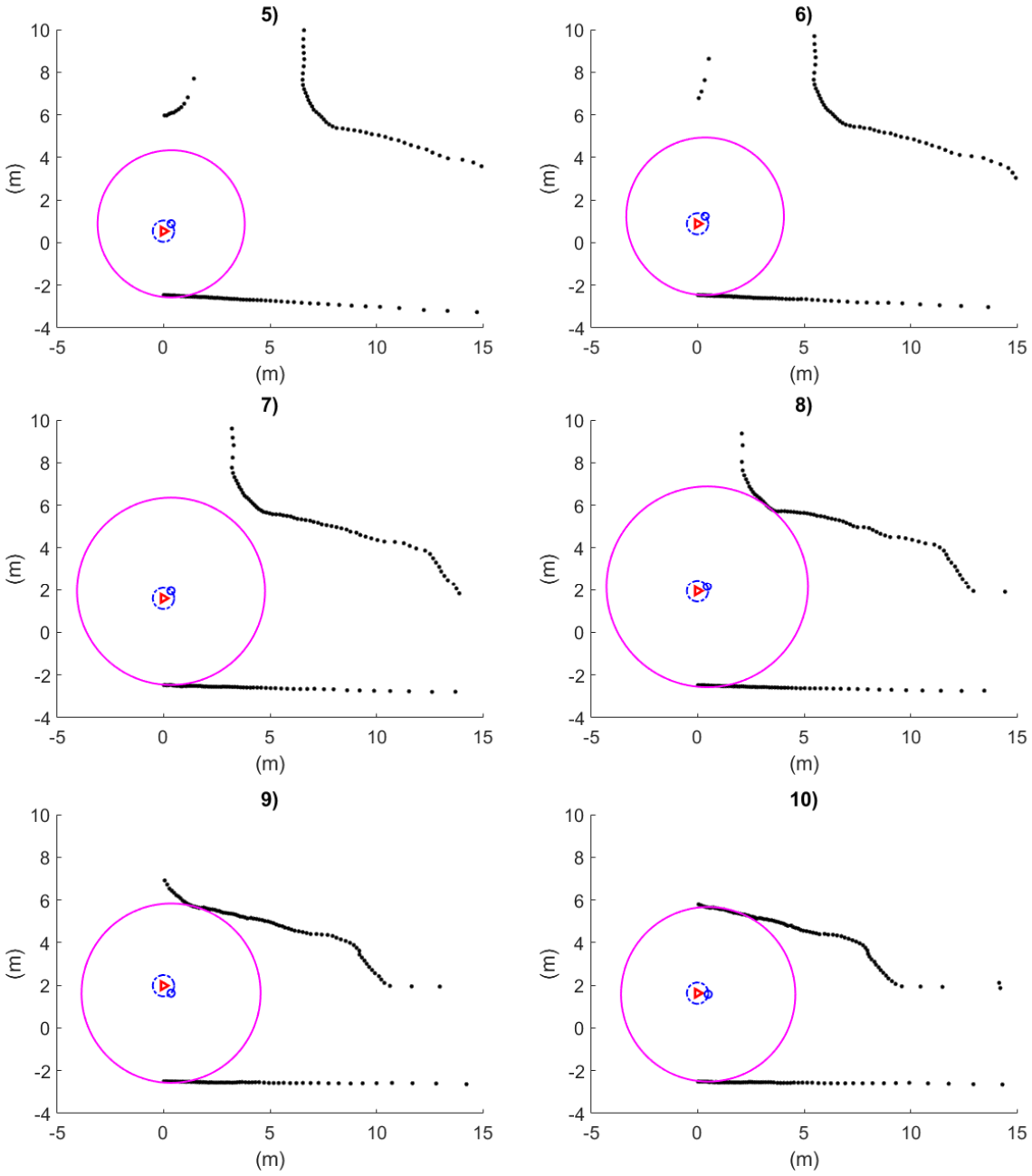

Fig. 15 “Sliding Balloon” en tiempo real II

- **“Sliding Balloon”** en tiempo real:

  Como observamos en la Fig. 14 y 15 la técnica “Sliding Balloon” utiliza en cada iteración la lectura que recibe del sensor para calcular la próxima meta. A lo largo de las 12 gráficas contenidas entre las dos imágenes (todos los pasos no son sucesivos entre si, se han escogido las figuras más relevantes para reflejar el comportamiento), vemos como el robot avanza a lo largo de un túnel, aproximadamente unos 17 metros. El mapa está representado por los puntos en color negro, el círculo magenta representa el globo, el triángulo rojo la posición del robot, el círculo azul el radio de avance y el círculo pequeño azul la próxima meta. En este ejemplo observamos un comportamiento curioso, ya que cuando aparece una galería en el lado izquierdo el globo no aumenta enormemente su diámetro. Esto se debe a que queremos avanzar en la dirección del túnel y no desviarnos, por lo que obligamos a avanzar en línea recta cuando el globo intenta aumentar enormemente ya que este aumento se debe a la aparición de una galería lateral. Todo el código queda reflejado en el anexo 8.1.3.

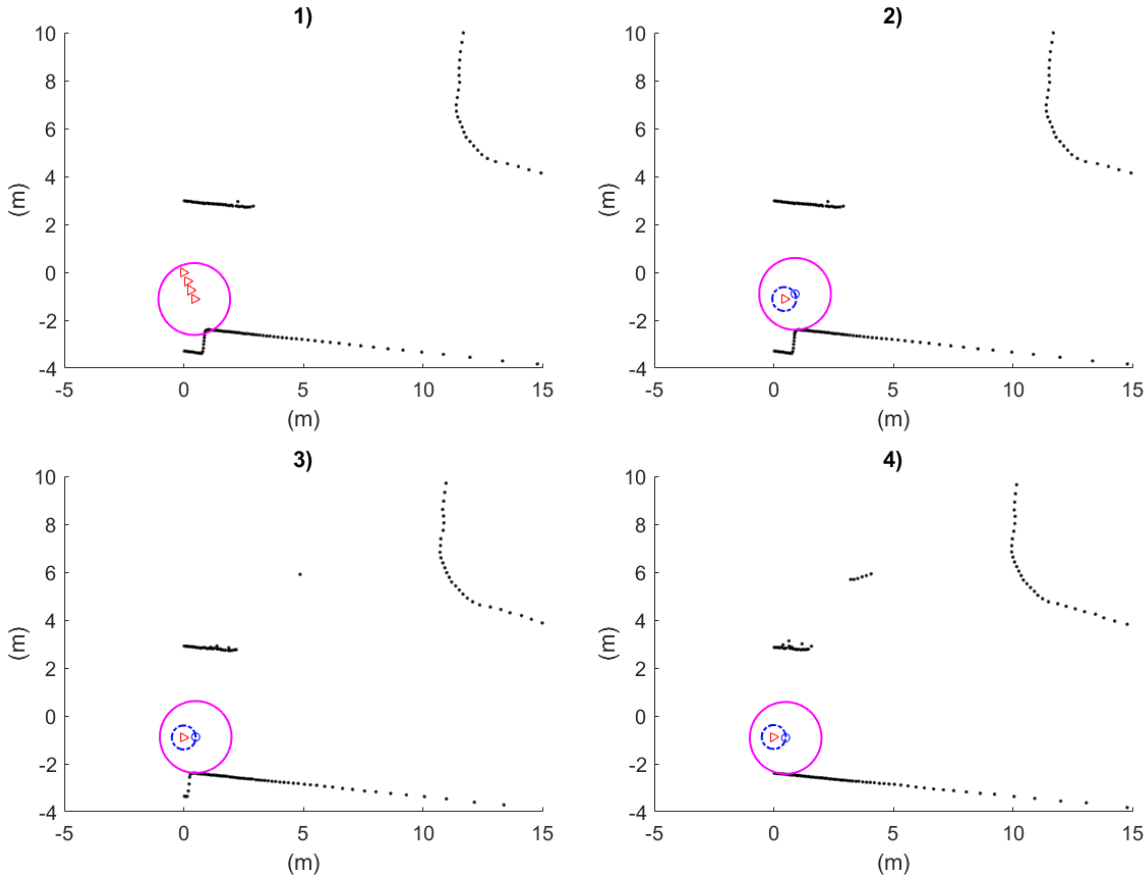

Fig. 16 “Wall Sliding Balloon” en tiempo real I

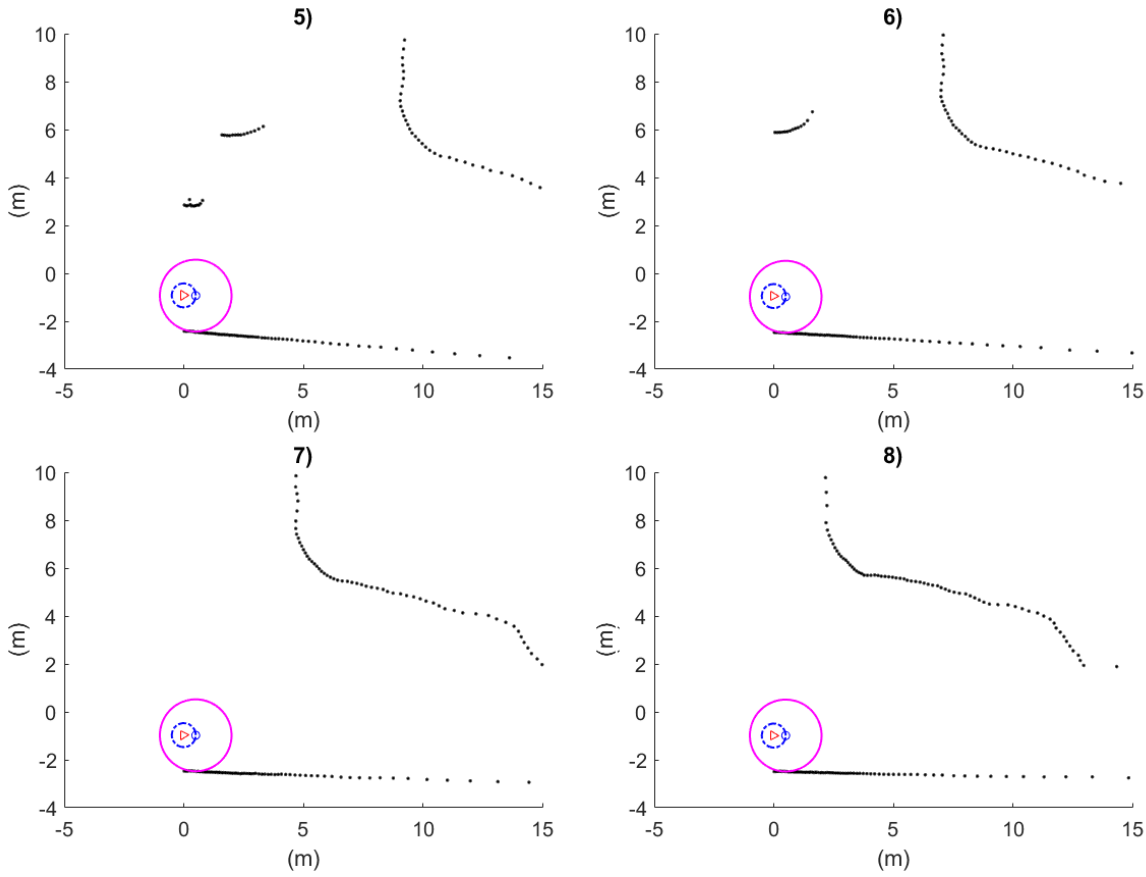

Fig. 17 "Wall Sliding Balloon" en tiempo real II

- **"Wall Sliding Balloon"** en tiempo real:

  En la Fig. 16 y 17 observamos el comportamiento de la técnica "Wall Sliding Balloon" en tiempo real, cada gráfica corresponde con una iteración de la navegación, donde el mapa mostrado corresponde con la última lectura del sensor láser. Igual que en las figuras anteriores el mapa se representa en color negro, en magenta el globo, en azul el radio de avance, en rojo la posición actual del robot y con un pequeño círculo azul la meta a alcanzar. En este caso, al igual que en planificación previa, también se ha elegido la pared derecha para mostrar los resultados. Observamos un correcto funcionamiento de la navegación, inicialmente se acerca a la pared deseada para posteriormente avanzar. En este ejemplo apenas hay variación del relieve puesto que se trata de las lecturas realizadas en un túnel recto, por lo que una vez posicionado a la distancia fijada, la dirección de avance es prácticamente fija. Cabe mencionar también que dada la amplitud del túnel que se está transitando, difícilmente se producirá la detención del robot a no ser que sea producida por un obstáculo que se encuentra en el túnel. Todo el código en el anexo 8.1.4.

Como podemos observar ambas versiones de la técnica de navegación se comportan a la perfección ante las lecturas realizadas por el sensor, a pesar de que esta simulación en tiempo

real no es estricta como su nombre expresa, por lo que podemos asegurar que el comportamiento de la técnica de navegación será la adecuada en un entorno real.

5. Algoritmos en ROS

ROS o “robot operating System”, sistema operativo robot en español, nos proporciona un “framework” flexible para la creación de “software” para robot. ROS no es más que un conjunto de herramientas, librerías dedicadas y convenciones, lo cual nos permite recrear el comportamiento de un robot en diferentes plataformas robóticas.

El funcionamiento de ROS es fundamentalmente modular, su principal objetivo es trabajar con pequeños programas denominados nodos, que solo realizan una o varias funciones, y que estos programas se comuniquen entre sí. Un sistema en ROS está conformado por muchos nodos ejecutándose simultáneamente y comunicándose a través de mensajes (no siendo necesaria que ambos nodos se encuentren en ejecución en la misma máquina).

Una de las formas de comunicación entre los programas es a través de un *topic*. Un *topic* es una “corriente de información” en la cual los nodos *Publishers* depositan *messages* y estos son transportados a los nodos *Subscribers*, es decir, un nodo puede publicar y/o recibir información a través de un *topic*. Este tipo de comunicación es instantánea, en cuanto la información es publicada esta es recibida por los nodos suscritos los cuales ejecutan una función *callback* automáticamente. La información publicada en un *topic*, denominada como *messages*, la cual debe ser siempre del mismo tipo definida en el fichero *message*. Pero también existe otro tipo de comunicación entre los nodos que es a través de los llamados *Services*, y a diferencia de los *topics*, estos se caracterizan por ejecutarse a través de llamadas síncronas. Esta diferencia los hace adecuados para la ejecución de funciones que solamente se ejecutan ocasionalmente. El servidor que proporciona este servicio define la función *callback* a ejecutar una vez que recibe una solicitud por el servicio que anuncia [8] [9].

Para entender mejor un sistema de ROS y como se comunican los nodos entre sí, se utiliza el gráfico que ROS proporciona. Este gráfico muestra todos los nodos que se están ejecutando, con que nodos se comunica y a través de que *topics* lo hace.

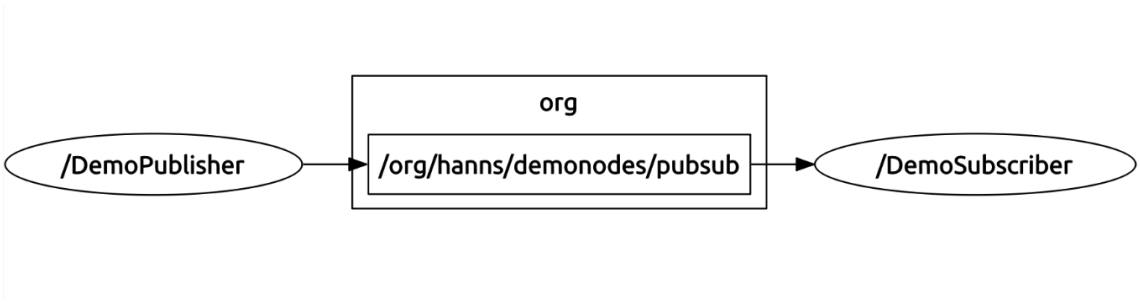

Fig. 18 Ejemplo ROS graph

Como se observa en la Fig. 18 el sistema de ROS representado se compone de dos nodos y un *topic* a través del cual intercambian mensajes, en este caso la información solo sigue un sentido puesto que /DemoPublisher publica en el *topic* /org/hanns/demonodes/pubsub y /DmoSubscriber recibe esta información para trabajar con ella.

Mediante el uso de este “framework” y apoyados en un simulador robótico denominado Gazebo, somos capaces de crear los algoritmos que finalmente podrán ser utilizados en cualquier robot real terrestre que haga uso de ROS con solamente unos pequeños cambios. Previamente a introducir el algoritmo en el robot, los algoritmos son fácilmente probados y testeados en dicho simulador, el cual nos proporciona un entorno preciso y eficiente donde obtener resultados de robots en entornos complejos tanto en interiores como en exteriores.

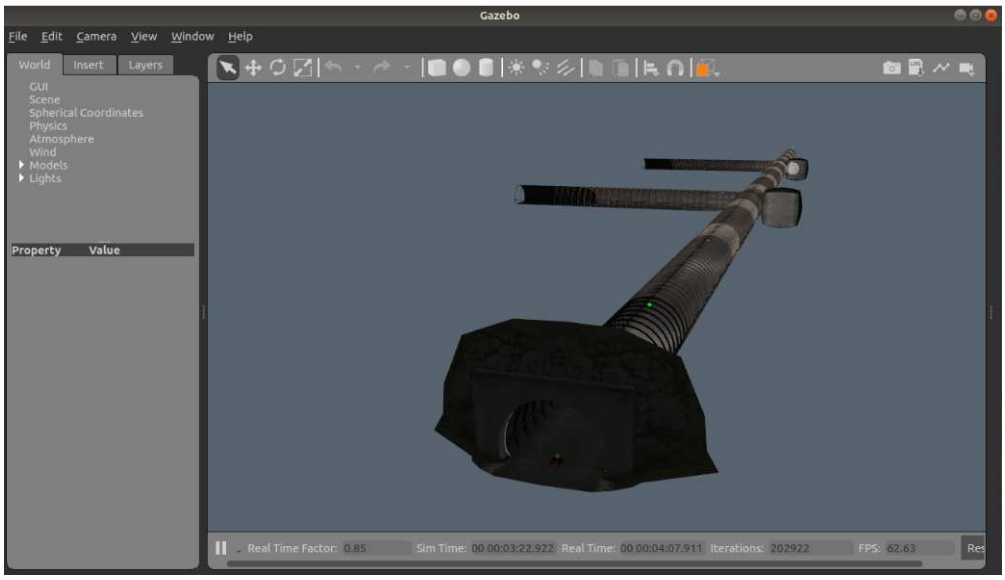

Fig. 19 Simulador Gazebo I

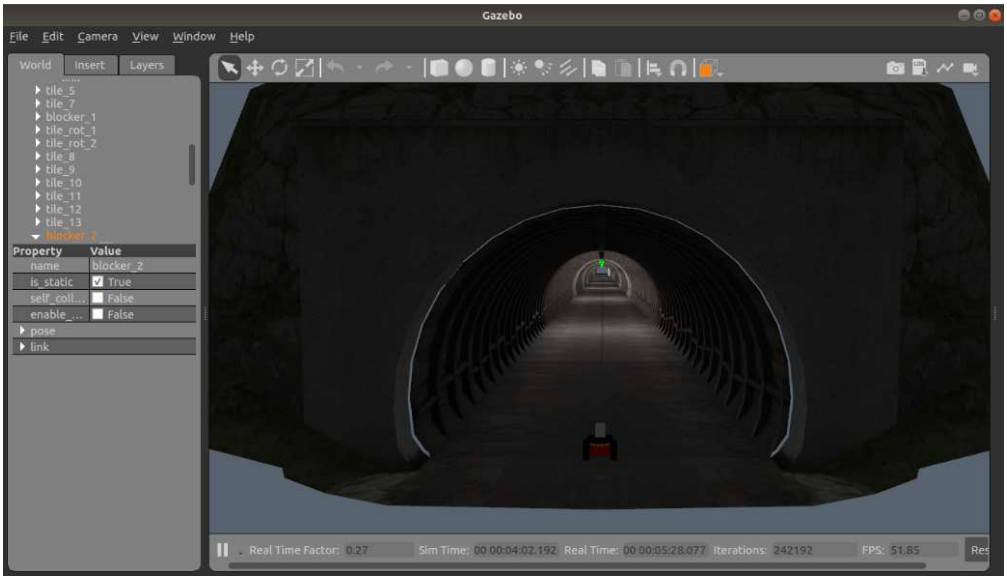

Fig. 20 Simulador Gazebo II

En la Fig. 19 y en la Fig. 20 observamos el entorno del simulador diseñado para probar el algoritmo de navegación. Se trata de un túnel recto con galerías a los lados. El terreno es fácilmente intercambiable para facilitar o dificultar el avance, así como también es fácil introducir obstáculos. También observamos el modelo del robot que se usara para las simulaciones.

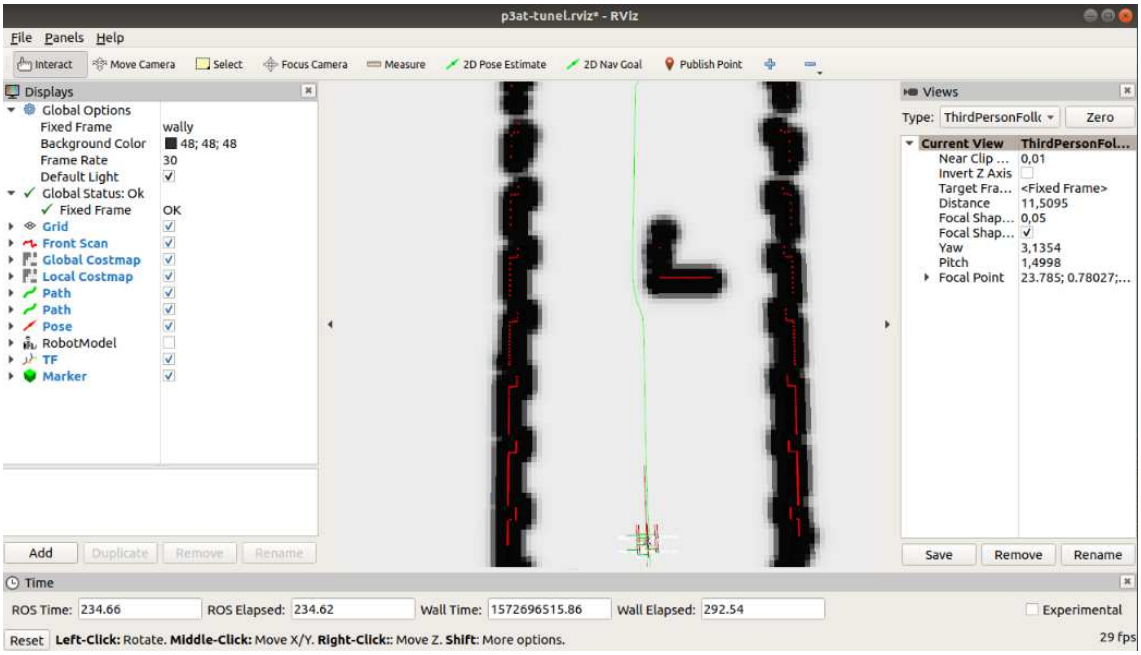

Fig. 21 RVIZ

En la Fig. 21 podemos ver la información propia del robot recogida por *RVIZ*, es decir sus ejes, así como también el entorno. La línea roja representa la información detectada por el haz laser y en negro el entorno, toda esta información se actualiza conforme el robot avanza por el túnel. Esta herramienta de visualización es propia de ROS.

Para explicar los algoritmos nos fijaremos únicamente en los realizados en tiempo real, ya que la planificación previa es más sencilla al no tener en cuenta la información recibida por el sensor. De esta forma los subapartados siguientes serán únicamente los dos modelos del algoritmo.

Puesto que la teoría del algoritmo ya ha sido explicada en apartados anteriores, en los siguientes subapartados explicaremos únicamente detalles más genéricos de la programación en ROS, dentro de ROS se ha decidido programar en c++. La programación en c++ es más robusta lo cual implica cambiar el tratamiento de la información con respecto a *Matlab*. Las operaciones se realizan con datos del mismo tipo, por lo que es necesario tener mayor cuidado a la hora de operar.

5.1. "Sliding Balloon"

ROS se basa en comunicación entre nodos a través de "topics" y "services", lo cual ya ha sido explicado al principio de este apartado. Para el desarrollo de este TFG y más en concreto esta versión del algoritmo contamos con siete nodos ("SlideBalloon", "RVIZ", "move_base", "Laser_shaper", "robot_state_publisher", "gallery" y "wally"). Hacemos una breve descripción de los cuatro más importantes.

- ***SlideBalloon*:** se trata del nodo principal del sistema de navegación ya que almacena la fichero principal que procesa, el cual se muestra en el anexo 8.2.1 del algoritmo de navegación y todas las funciones que definen la forma de realizar los cálculos geométricos, igualmente han sido recogidas en el anexo 8.2.3.
- ***Move_base*:** este nodo se encarga de mover las diferentes ruedas del robot con fin de alcanzar la pose determinada una vez que recibe el objetivo. Además busca realizar el camino más corto entre la meta y el robot, es decir que se mueve siempre en dirección a la meta actual, aunque la meta anterior no se haya alcanzado de forma que los giros no son tan bruscos.
- ***Laser_shaper*:** encargado de modelar el funcionamiento del sensor laser utilizado.
- ***Rviz*:** En este nodo se realizan los cálculos con toda la información recogida. Sirve también como herramienta de visualización de la información recogida por el robot (muestra el laser y las paredes detectadas por este), al igual que la posición de sus ejes y la meta alcanzar con su respectiva trayectoria a seguir.

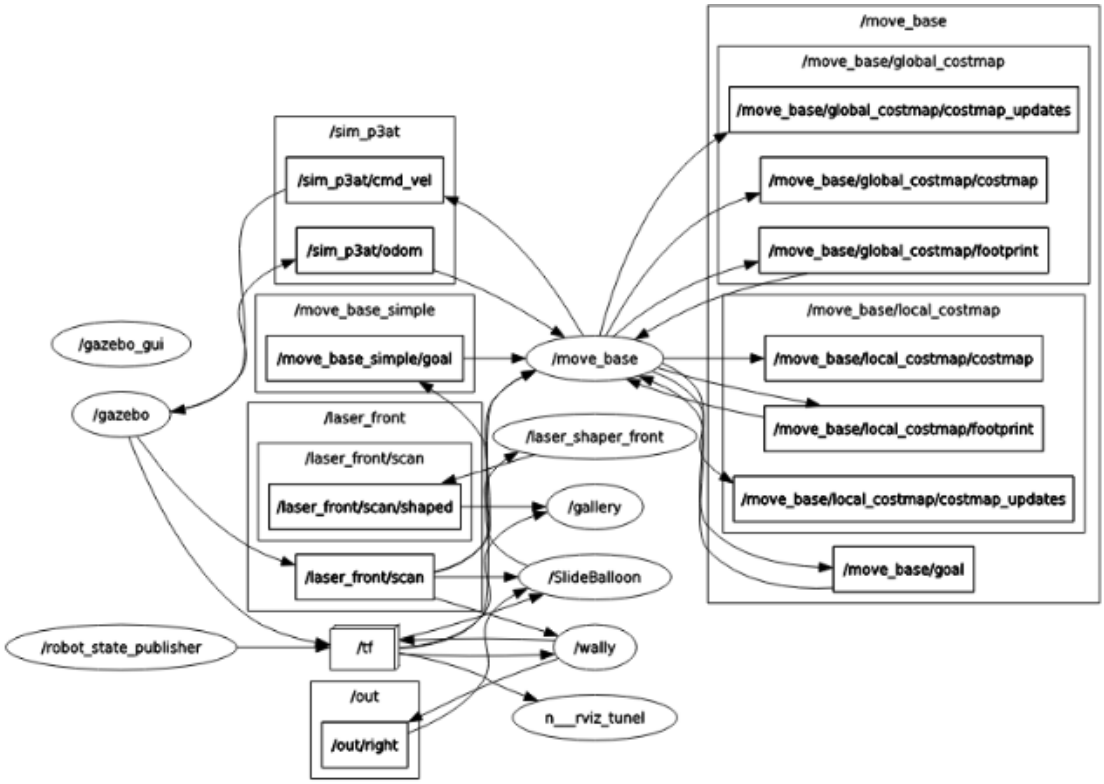

Fig. 22 ROS graph de "Sliding Balloon"

Una vez explicado el funcionamiento básico de cada nodo, pasamos a explicar en detalle el nodo que concierne al estudio de este trabajo, “Slide Balloon”. Este nodo está compuesto por varios ficheros, los cuales almacenan las diferentes funciones, es esencial para determinar la trayectoria de avance que el robot seguirá en el entorno en función de la información recogida por el robot a través del resto de nodos. En la Fig. 22 podemos ver el grafo con todos los nodos que se encuentran en funcionamiento, también podemos ver a través de que *topics* se comunican entre si.

En este caso nuestro nodo principal está suscrito a dos “topics” (/laser_front/scan y /out/right), dichas subscripciones ejecutan una función de llamada/regreso. Una de ellas “`SlideBalloon::scan_callback`”, anexo 8.2.4, se encarga de actualizar la nube de puntos que el robot ve en cada instante, y la segunda “`SlideBalloon::wallRightCallback`”, anexo 8.2.4, se encarga de referenciar el robot respecto al entorno y la meta. Al mismo tiempo también publica información en otros dos “topics” (goalStamped y SlideBalloon/alive), dicha información es el objetivo a alcanzar y un mensaje que avisa de que el nodo continua en funcionamiento. Finalmente el nodo cuenta con un servicio “`SlideBalloon::enableCallback`” el cual permite publicar la meta en el “topic”, es decir da la orden de comienzo.

Debido a la velocidad de lecturas ofrecidas por el laser se ha decidido calcular una meta para cada una de estas lecturas. Lo que quiere decir que el algoritmo no espera a alcanzar una meta para calcular la que va a continuación, de esta forma obtenemos una serie de metas consecutivas que el robot irá buscando alcanzar, consiguiendo así un movimiento más suave evitando giros bruscos ya que el propio navegador busca el mejor camino para concatenar la posición del robot y la del objetivo.

A diferencia que en Matlab, que tiene diferentes “toolbox” que te permiten hacer uso de funciones ya creadas, en ROS debemos programar desde cero todas las funciones que sean necesarias. Estas funciones que “SlideBalloon” hace uso son:

- **`NearestPoint`:** Calcula el punto más cercano del set de puntos detectado por el láser.
- **`SameSide`:** Nos permite discernir si dos puntos se encuentran en el mismo lado que un vector dirección.
- **`IntersectionCircles`:** Devuelve el punto más cercano al punto objetivo, de entre los dos calculados de la intersección entre dos círculos.
- **`PointsInRadius`:** devuelve un vector con los puntos cuya distancia al centro es menor al radio.
- **`FarestPoint`:** Devuelve el punto más alejado del set de puntos.

Todas las funciones ahora explicadas se muestran en el anexo 8.2.3, dentro del fichero “geometry”.

Para mostrar el correcto funcionamiento del algoritmo se ha probado en dos mapas diferentes en los cuales se pueden encontrar obstáculos a la altura que sensa el haz plano.

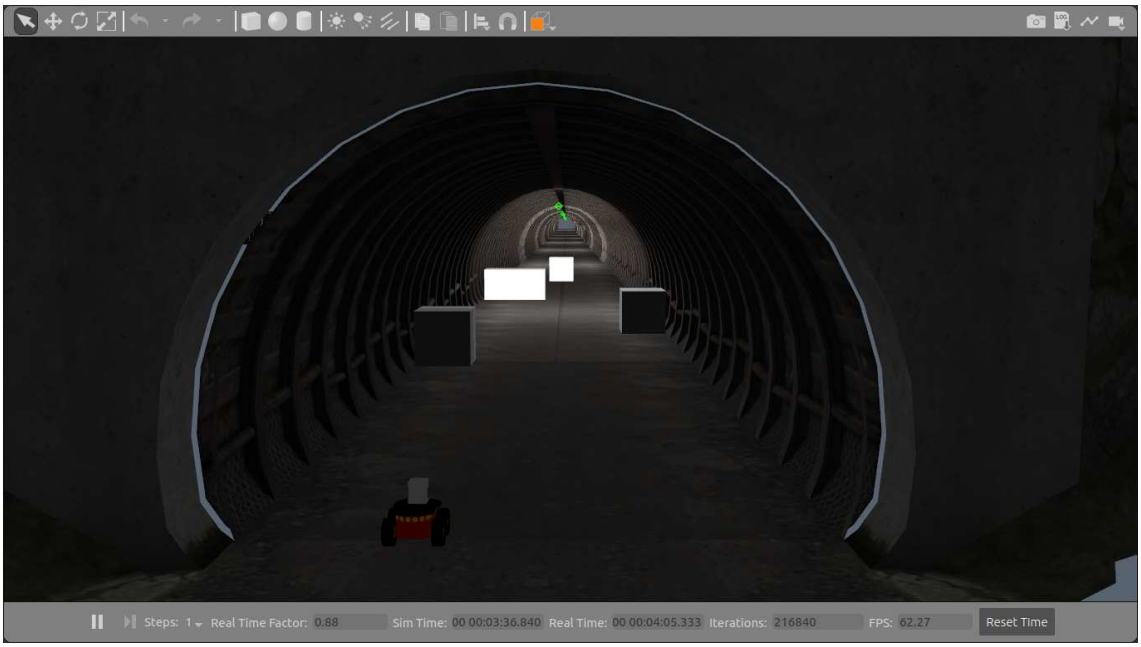

Fig. 23 Vista 3D del túnel recto (mapa I)

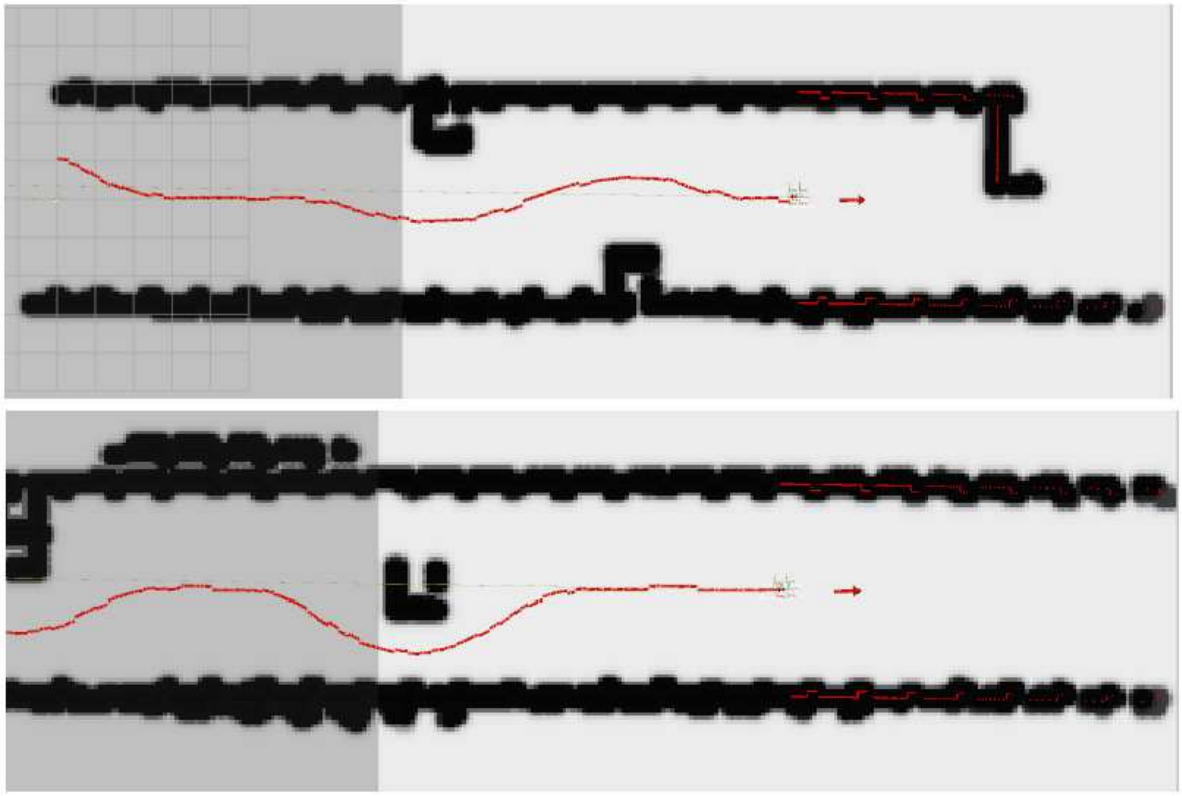

Fig. 24 Trayectoria seguida por el robot, en el mapa I, usando "Sliding Balloon"

En la Fig. 23 y 24 observamos que la primera prueba se ha realizado en un túnel recto con varios obstáculos a ambos lados y uno centrado en mitad del plano. La Fig. 23, muestra el mapa creado en Gazebo y la localización de los obstáculos mediante una vista en tercera persona. Sin embargo la Fig. 24, al ser un algoritmo 2D, nos basta con la vista aérea para interpretar la información. En esta imagen, obtenida en RVIZ, podemos identificar diferentes elementos como el mapa que se va actualizando según las lecturas del sensor, representadas por las líneas rojas sobre las paredes, la trayectoria seguida por el robot a través del túnel, la meta actual y los ejes del robot. Por la trayectoria seguida podemos afirmar que en todo momento el robot busca la zona central del mapa, siendo esta la más segura y de mayor distancia a los obstáculos.

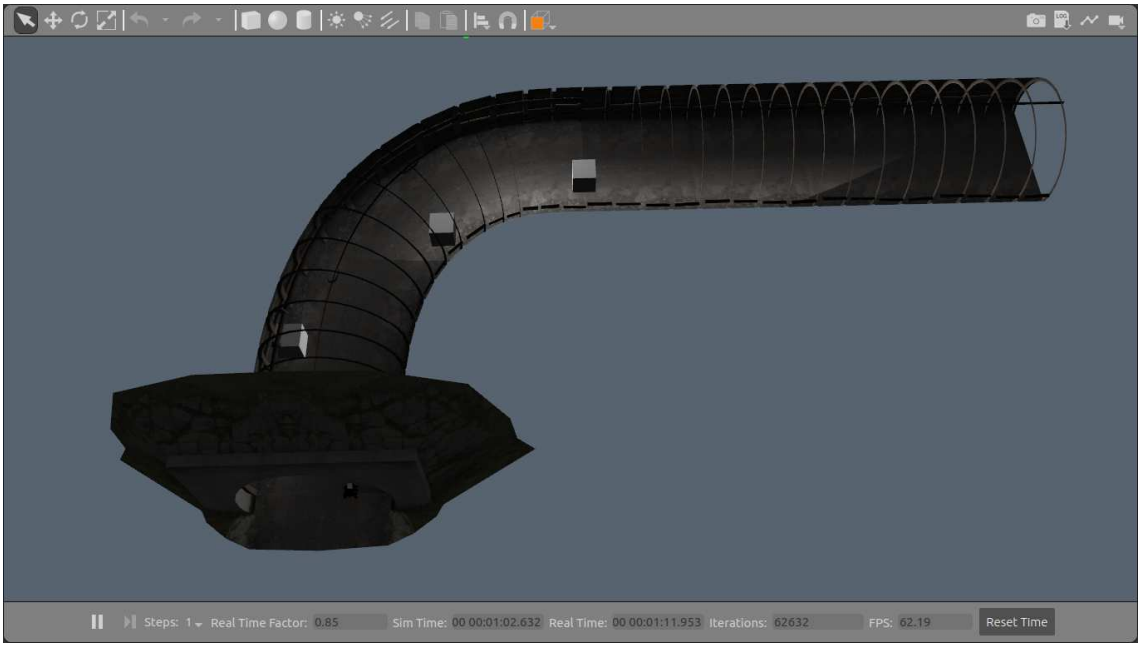

Fig. 25 Vista 3D del túnel curvo (mapa II)

Al igual que en el caso anterior en la Fig. 25 y 26 tenemos representado el segundo mapa utilizado en la simulación, que en este caso cuenta con una desviación hacia la derecha al comienzo. A pesar de contener una curva se puede ver como el robot es capaz de seguir la trayectoria del túnel a la par que esquiva los óbstaculos que se encuentra por el camino.

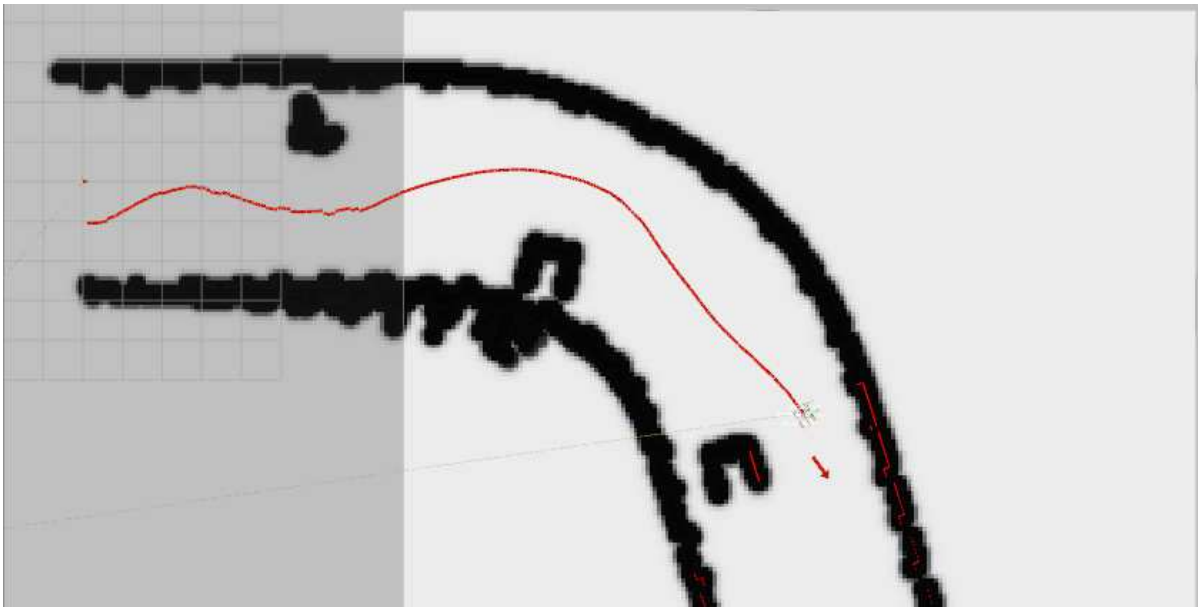

Fig. 26 Trayectoria seguida por el robot, en el mapa II, usando "Sliding Balloon"

5.2. "Wall Sliding Balloon"

Como ya hemos explicado anteriormente "Wall Sliding Balloon" es una variante de "Sliding Balloon" de forma que su programación en ROS es muy similar. De forma que también hace uso los mismos nodos, los siete que han sido enumerados en el subapartado anterior, cambiando en este caso el nodo principal.

El nodo "WallSlideBalloon" se compone de los mismos ficheros, ya que ambas versiones comparten funciones, el único cambio resulta del procesamiento de la información en el fichero principal.

Al igual que "SlideBalloon"," WallSlideBalloon" está suscrito a dos "topics" (/laser_front/scan y /out/right), estas subscripciones ejecutan una función de llamada/regreso. La primera "`WallSlideBalloon::scan_callback`", anexo 8.2.4, se encarga de actualizar la nube de puntos, y la segunda "`WallSlideBalloon::wallRightCallback`", anexo 8.2.4, se encarga de referenciar el robot respecto al entorno y la meta. Al mismo tiempo también publica información en otros dos "topics" ("goalStamped" y "SlideBalloon/alive"), dicha información es el objetivo a alcanzar y un mensaje que avisa de que el nodo continua en funcionamiento. Sin embargo como gran diferencia "WallSlideBalloon" cuenta con dos servicios "`SlideBalloon::enableCallback`" el cual permite publicar la meta en el "topic" para que "move_base" lo reciba, es decir, da la orden de comienzo y "`SlideBalloon::sideCallback`" la cual nos permite decidir que pared seguirá el robot (con true para el lado derecho y false para el izquierdo), anexo 8.2.5.

Debido a la velocidad de lecturas ofrecidas por el laser se ha decidido calcular una meta para cada una de estas lecturas. Lo que quiere decir que el algoritmo no espera a alcanzar una meta para calcular la que va a continuación, de esta forma obtenemos una serie de metas consecutivas que el robot irá buscando alcanzar, consiguiendo así un movimiento más suave

evitando giros bruscos ya que el propio navegador busca el mejor camino para concatenar la posición del robot y la del objetivo.

A diferencia que en Matlab que tiene diferentes "toolbox" que te permiten hacer uso de funciones ya creadas, en ROS debemos programar desde cero todas las funciones que sean necesarias, las cuales quedan recogidas geometry.cpp del anexo 8.2.3. Estas funciones que "WallSlideBalloon" hace uso son (además de las ya explicadas `NearestPoint`, `PointsInRadius`, `IntersectionCircles`):

- **`OnTheRight`:** Calcula si un punto está a la derecha del punto objetivo dado el vector dirección.

Las pruebas para comprobar el comportamiento de "Wall Slide Balloon" se han realizado en los mismos mapas, de forma que en las pruebas ambas versiones de la navegación están bajo las mismas condiciones. Para mostrar que funciona para ambas paredes para la primera prueba se ha elegido la pared izquierda y la derecha para la segunda prueba.

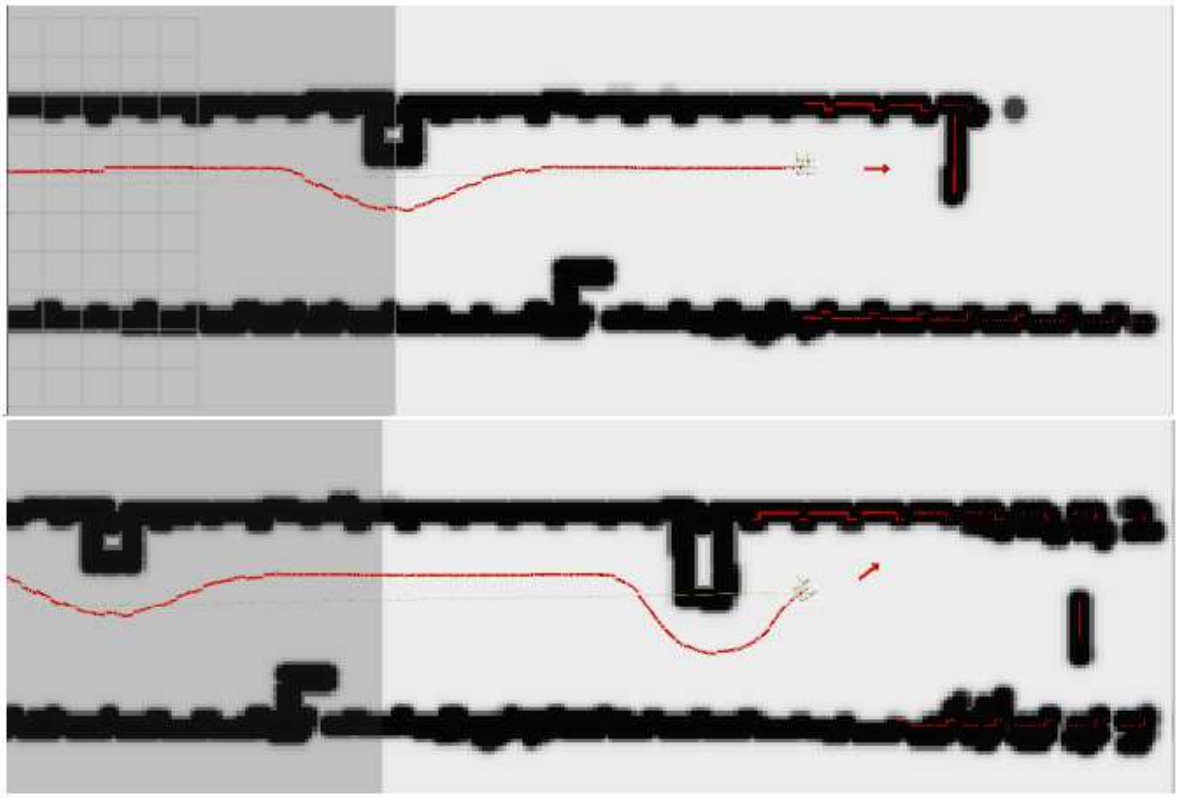

Fig. 27 Trayectoria seguida por el robot, en el mapa I, usando "Wall Sliding Balloon"

En la Fig. 22 y 26 observamos de nuevo el túnel recto con obstáculos a ambos lados. La Fig. 22 muestra el mapa creado en Gazebo y la localización de los obstáculos mediante una vista en tercera persona. Sin embargo la Fig. 26, al ser un algoritmo 2D, nos basta con la vista aérea para interpretar la información. En esta imagen, obtenida en RVIZ, se pueden identificar los

diferentes elementos ya mencionados. Por la trayectoria seguida podemos confirmar que el robot busca la pared izquierda manteniendo siempre una distancia fija.

Al igual que en el caso anterior en la Fig. 25 y 28 tenemos representado el segundo mapa utilizado en la simulación, que en este caso cuenta con una desviación hacia la derecha al comienzo. A pesar de contener una curva se puede ver como el robot es capaz de seguir la trayectoria del túnel centrándose en el relieve de la pared derecha a la par que esquiva los obstáculos que se encuentra por el camino.

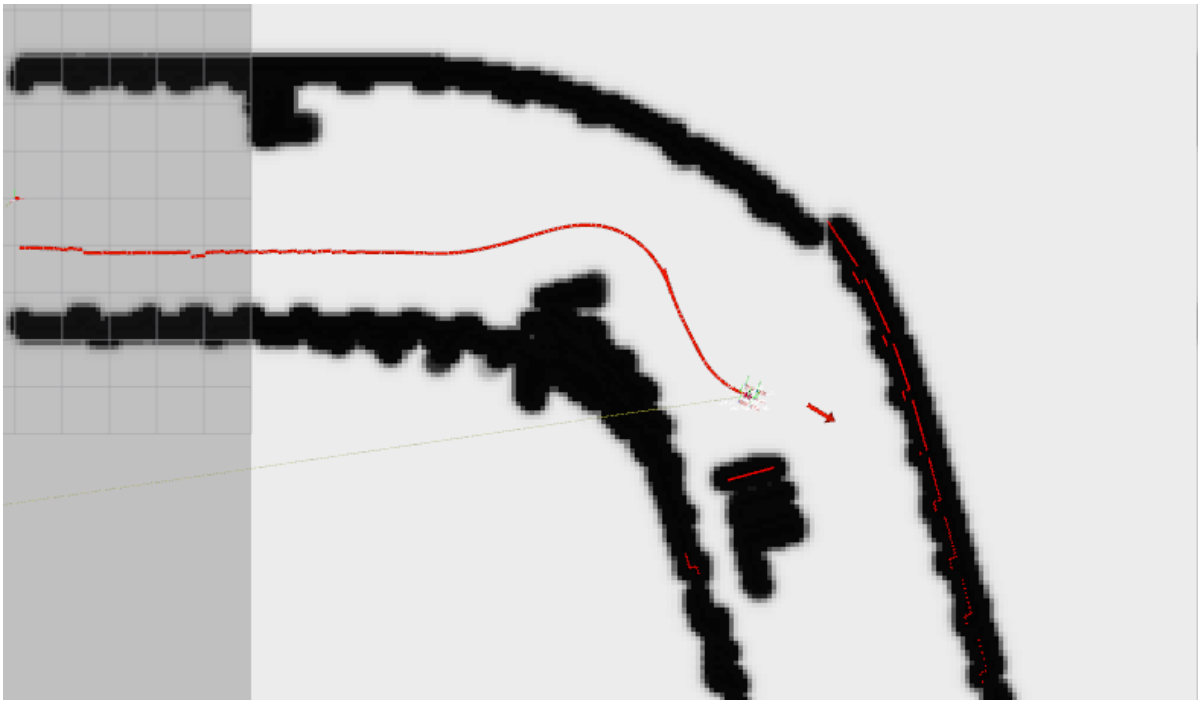

Fig. 28 Trayectoria seguida por el robot, en el mapa II, usando "Wall Sliding Balloon"

6. Conclusiones y trabajo futuro

Este proyecto ha consistido en el desarrollo una técnica de navegación 2D terrestre denominada “Sliding Balloon” cuyo fin es ayudar en la búsqueda de nuevos métodos para navegar en entornos desafiantes demostrando la valided de este tipo de técnicas. El proyecto se ha compuesto de dos fases bien diferenciadas. La primera ha consistido en un estudio y documentación de técnicas similares a la par que del desarrollo de la idea fundamental mediante pruebas simuladas en un entorno sencillo de programación como es matlab. Mientras que la segunda se ha basado en el aprendizaje de la plataforma ROS y del desarrollo del algoritmo en dicha programa.

Basándose en la idea principal se ha desarrollado también una versión adicional que aporta una mayor versatilidad a la hora de elegir el movimiento que deseamos realice el robot. De esta forma la versión principal siempre busca avanzar por la parte central del espacio en el que se encuentra y detecta. La segunda versión busca moverse siguiendo el relieve de una de las paredes manteniendo siempre una distancia fija y con un área de seguridad. Durante este trabajo ha quedado demostrado que ambos algoritmos funcionan correctamente, pero aun así queda margen de mejora principalmente a la hora de toma de decisiones.

Dichos algoritmos se han centrado en la navegación en un entorno similar a un túnel, con el objetivo de ser capaces de evitar obstáculos, pero dicho código tiene sus limitaciones como son bifurcaciones (en principio seguiría la dirección principal del túnel sin opción a decidir), recovecos (puesto que el robot se detendría, Fig.28) y zonas amplias para el “Sliding Balloon”. Por el otro lado las limitaciones de “Wall Slide Balloon” son zonas con poca distancia entre las paredes (lo cual sumado a los obstáculos puede dar lugar a confusión entre paredes), obstáculos que se encuentran a una distancia un poco superior a la fija con respecto a la pared (si entra en radio de seguridad se detiene), como se puede observar en la Fig.29.

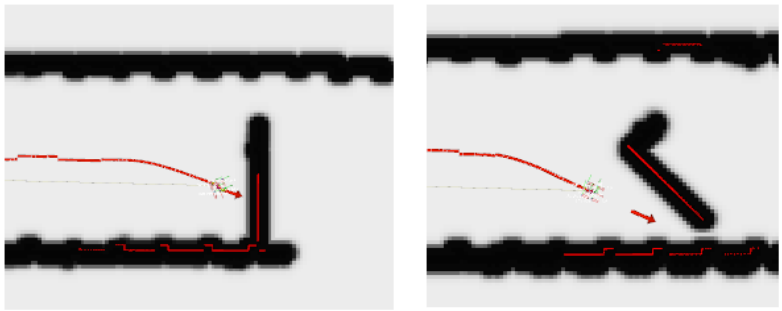

Fig. 29 Ejemplo de fallos por “Slide Balloon”

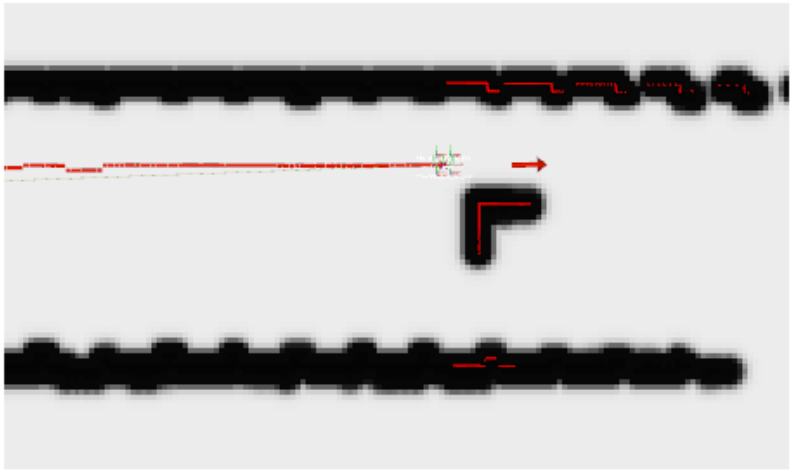

Fig. 30 Ejemplo de fallos por "Wall Slide Balloon"

El principal objetivo de este proyecto era demostrar la valided de esta técnica a la hora de la navegación en 2D. Una vez demostrado su funcionamiento, se abre la puerta a través de futuros trabajos, al desarrollo de los algoritmos para la toma de decisiones ante espacios más complejos, desarrollo de la técnica en 3D para robot terrestre e incluso para drones los cuales pueden resultar en una temática de estudio muy interesante.

7. Bibliografía

[1] I.R. Nourbakhsh, R. Siegwart. “Introduction to Autonomous mobile robots.” Bradford Books, 2004.

[2] Bambino, I. “Una Introducción a los Robots Móviles” , 2008.

[3] J. Borenstein and Y. Koren, “Real-time obstacle avoidance for fast mobile robots” IEEE Trans. Syst., Man, Cybern., vol. 19, pp. 1179–1187, May 1989.

[4] J. Borenstein and Y. Koren, “Real-time Obstacle Avoidance for Fast Mobile Robots in Cluttered Environments” IEEE Trans. Syst., Man, Cybern., vol. 19, pp572-577, May 1990.

[5] J. Borenstein and Y. Koren, “The Vector Field Histogram-Fast Obstacle Avoidance for Mobile Robots” IEEE Trans. Robotics and Automation, vol. 7, pp278-288, June 1991.

[6] Javier Minguez and Luis Montano. Nearness Diagram (ND) Navigation: Collision Avoidance in Troublesome Scenarios. IEEE Transactions on Robotics and Automation, 20(1), 2004.

[7] Khatib, O., Quinlan, S., “Elastic Bands: Connecting, Path Planning and Control,” in Proceedings of IEEE International Conference on Robotics and Automation, Atlanta, GA, May 1993.

[8] Lentin Joseph. “Mastering ROS for Robotics Programming”, Packt> 2015.

[9] Morgan Quigley, Brian Gerkey and William D. Smart. “Programing robots with ROS”, O’Reilly Media 2015.

[10] http://wiki.ros.org/

[11] http://www.cplusplus.com/

[12] https://es.mathworks.com/help/vision/index.html

[13] https://es.mathworks.com/help/vision/ref/pointcloud.findnearestneighbors.html?s_tid=doc_ta

8. Anexos

8.1. Código Matlab

8.1.1. "Sliding Balloon" planificación previa

```
while true
    plot (pp(1), pp(2), 'r>') ;
    AdvanceAnterior=AdvanceVector;
    BallonCenter = pp + AdvanceAnterior*AdvanceStep ; % punto al
que nos dirigimos, tomando la última dirección
    BallonCenterSupuesto=BallonCenter;%almacenamos el punto al que
nos dirigiriamos inicialmente para calcular ángulos
    EndInflate = false ;

   % plot (BallonCenter(1), BallonCenter(2), 'r*') ;
    circle (pp, AdvanceStep);

    while not(EndInflate) % mientras que no se ocupe el espacio
disponible ...

        % obtenemos EL punto más cercano a BallonCenter
        [indices,dists] =
findNearestNeighbor(ptCloud,BallonCenter,1);
        NearestPoint = ptCloud.Location(indices,:) ;

        % r : distancia/radio entre BallonCenter y NearestPoint
        r = dists(1) ;
        % l: distancia entre punto partida y NearestPoint
        l = norm (pp - NearestPoint) ;

        if (l+AdvanceStep < r+d)
          break %stop si el radio es mayor a l
        end

        %obtenemos los dos puntos, dentro del radio de movimiento,
que
        %interseccionan con la circunferencia de radio r+d
(r=distancia entre
        %BallonCenter y NearestPoint, d=pequeño diferencial)
tomando como centro
        %NearestPoint
        [p1 p2 Sol] = SolveCircleIntersection (pp, AdvanceStep,
NearestPoint, r+d) ;

        % BallonCenter es ahora el punto más cercano de los dos
calculados
        BallonCenter = NearestSol (BallonCenter, p1, p2) ;
        BallonRadius = r+d ;

        % calculamos el vector dirección a el nuevo BallonCenter y
lo
        % "normalizamos"
        AdvanceVector = BallonCenter - pp ;
        AdvanceVector = AdvanceVector / norm(AdvanceVector) ;
```

```
        %Calculamos el ángulo entre la dirección anterior y la
nueva

AB=AdvanceVector(1)*AdvanceAnterior(1)+AdvanceVector(2)*AdvanceAntr
ior(2)+AdvanceVector(3)*AdvanceAnterior(3);
     ProdNorm=norm(AdvanceVector)*norm(AdvanceAnterior);
        ang=acos((AB/ProdNorm));
        ang=180*ang/pi;
        %si el ángulo es superior a un valor máximo limitamos su
giro
        if (ang>angMaxGiro)
            if(BallonCenter(2)>BallonCenterSupuesto(2))%si el punto
calculado está más arriba que el inicialmente calculado giro a la
izquierda
                AdvanceVector=(RIzq * AdvanceAnterior')';
                BallonCenter = pp + AdvanceVector*AdvanceStep ;
                BallonRadius=norm(NearestPoint-BallonCenter);
                break
            else  %si el punto calculado está más abajo que el
inicialmente calculado giro a la derecha
                AdvanceVector=(RDcha * AdvanceAnterior')';
                BallonCenter = pp + AdvanceVector*AdvanceStep ;
                BallonRadius=norm(NearestPoint-BallonCenter);
                break
            end
        end

        % Comprobamos si dentro del radio actual hay algún punto
        [indices,dists] =
findNeighborsInRadius(ptCloud,BallonCenter,BallonRadius) ;
        nv = NearestPoint - BallonCenter ;

        %comprobamos si al hinchar la esfera encontramos algún
punto además de nearestpoint,
        %si es así comprobamos si ambos puntos están al mismo lado
de
        %la dirección de avance, si cada uno está a un lado dejamos
de hinchar.
        for i = 1:length(indices)
            iv = ptCloud.Location(indices(i),:) - BallonCenter ;
            if not (SameSide (AdvanceVector, nv, iv))
                EndInflate = true ;
                break
            end
        end
    end

    plot (BallonCenter(1), BallonCenter(2), 'bo') ;
    circle (BallonCenter, BallonRadius);
    %detectamos si ha llegado al final o es demasiado estrecho como
para pasar
    if (pf(1)-BallonCenter(1)<0)
        disp(mensaje1);
        break
    elseif (findNeighborsInRadius(ptCloud,BallonCenter,SRadius))
        circle (BallonCenter, SRadius);
        disp(mensaje2);
        break
    end

    %actualizamos el punto inicial para el próximo movimiento
```

```
        pp = BallonCenter ;
        AdvanceAnterior=AdvanceVector;
        pause ;
    end ;
```

8.1.2. "Wall Sliding Balloon" planificación previa

```
while true
    plot (pp(1), pp(2), 'r>', 'LineWidth', 2) ;
    LastVector = AdvanceVector;
    BallonCenter = pp + AdvanceVector*AdvanceStep ;
    Fin = false ;

    %plot (BallonCenter(1), BallonCenter(2), 'r*') ;
    circle (pp, AdvanceStep);


    while not (Fin)

        BallonCenter = pp + AdvanceVector*AdvanceStep ;

        %Actualizamos el NearestPoint
        [indices,dists] = findNearestNeighbors
(ptCloud,BallonCenter,1) ;
        NearestPoint = ptCloud.Location(indices(1),:) ;

        %Comprobamos que el nearestPoint está a la derecha, si no
lo está
        %giramos la dirección con intención de encontrar uno más
cercano que se
        %encuentre a la derecha
        if not (OnTheRightSide (BallonCenter, AdvanceVector,
NearestPoint))
            AdvanceVector = (Rz * AdvanceVector')' ;
        else
            Fin = true ;
        end ;

    end ;


    [p1 p2 Sol] = SolveCircleIntersection (pp, AdvanceStep,
NearestPoint, BallonRadius) ;

    if (Sol==true)
        BallonCenter = NearestSol (BallonCenter, p1, p2) ;

        %av: vector de avance actual
        AdvanceVector = BallonCenter - pp ;
        AdvanceVector = AdvanceVector / norm(AdvanceVector) ;
    else
        AdvanceVector = LastVector ;
        AdvanceVector = AdvanceVector / norm(AdvanceVector) ;
    end;

    plot (BallonCenter(1), BallonCenter(2), 'bo', 'LineWidth', 1.5)
;
    circle (BallonCenter, BallonRadius) ;
```

```
        %Comprobamos si ha llegado al final del túnel o ha vuelto al
    principio
        if (pf(1)-BallonCenter(1)<1)
            disp(mensaje1);
            break
        elseif(BallonCenter(1)-pi(1)<-1)
            disp(mensaje2);
            break
        end;

        pp = BallonCenter ;
        pause ;

end ;
```

8.1.3. "Sliding Balloon" tiempo real

```
for i = 6500:20 :length(ranges(:,1))
    %Para las lecturas del laser a partir de la 6500

    disp (i) ;
    advance=AdvanceVector;
    advance = advance / norm(advance) ;
    %pasamos información de punto actual y dirección para calcular
el mapa
    %alrededor del robot
    mapa=CreateMapa(i);
    %una vez calculado el mapa lo guardamos y cargamos
    save mapa.mat
    load('mapa');
    ptCloud = pointCloud(mapa);
    figure;
    hold on;
    plot (pp(1), pp(2), 'r>',  'LineWidth', 2) ;
    plot (ptCloud.Location(:,1), ptCloud.Location(:,2),'k.',
'MarkerSize', 10) ;

    AdvanceAnterior=AdvanceVector;
    BallonCenter = pp + AdvanceAnterior*AdvanceStep ; % punto al
que nos dirigimos, tomando la última dirección
    BallonCenterSupuesto=BallonCenter;%almacenamos el punto al que
nos dirigiríamos inicialmente para calcular ángulos
    EndInflate = false ;

%     plot (BallonCenter(1), BallonCenter(2), 'r*') ;
    circle (pp, AdvanceStep);

    while not(EndInflate) % mientras que no se ocupe espacio
disponible...

        % obtenemos EL punto más cercano a BallonCenter
        [indices,dists] =
findNearestNeighbors(ptCloud,BallonCenter,1) ;
        NearestPoint = ptCloud.Location(indices,:) ;

        % r : distancia/radio entre BallonCenter y NearestPoint
        r = dists(1) ;
     % l: distancia entre punto partida y NearestPoint
        l = norm (pp - NearestPoint) ;

        if (l+AdvanceStep < (r+d))
             AdvanceVector = [1 0 0] ;
             BallonCenter = pp + AdvanceAnterior*AdvanceStep ;

         break %stop si el radio es mayor a l
        end

     %obtenemos los dos puntos, dentro del radio de movimiento, que
     %interseccionan con la circunferencia de radio r+d
(r=distancia entre
```

```
     %BallonCenter y NearestPoint, d=pequeño diferencial) tomando
como centro
     %NearestPoint
        [p1 p2 Sol] = SolveCircleIntersection (pp, AdvanceStep,
NearestPoint, r+d) ;

     % BallonCenter es ahora el punto más cercano de los dos
calculados
        BallonCenter = NearestSol (BallonCenter, p1, p2) ;
        BallonRadius = r+d ;

     % calculamos el vector dirección a el nuevo BallonCenter y lo
     % "normalizamos"
        AdvanceVector = BallonCenter - pp ;
        AdvanceVector = AdvanceVector / norm(AdvanceVector) ;

     %Calculamos el ángulo entre la dirección anterior y la nueva

AB=AdvanceVector(1)*AdvanceAnterior(1)+AdvanceVector(2)*AdvanceAnte
rior(2)+AdvanceVector(3)*AdvanceAnterior(3);
        ProdNorm=norm(AdvanceVector)*norm(AdvanceAnterior);
        ang=acos((AB/ProdNorm));
        ang=180*ang/pi;
     %si el ángulo es superior a un valor máximo limitamos su giro
        if (ang>angMaxGiro)
            if(BallonCenter(2)>BallonCenterSupuesto(2))%si el punto
calculado está más arriba que el inicialmente calculado giro a la
izquierda
                AdvanceVector=(RIzq * AdvanceAnterior')';
                BallonCenter = pp + AdvanceVector*AdvanceStep ;
                BallonRadius=norm(NearestPoint-BallonCenter);
                break
            else  %si el punto calculado está más abajo que el
inicialmente calculado giro a la derecha
                AdvanceVector=(RDcha * AdvanceAnterior')';
                BallonCenter = pp + AdvanceVector*AdvanceStep ;
                BallonRadius=norm(NearestPoint-BallonCenter);
                break
         end
        end

    % Comprobamos si dentro del radio actual hay algun punto
        [indices,dists] =
findNeighborsInRadius(ptCloud,BallonCenter,BallonRadius) ;

        nv = NearestPoint - BallonCenter ;

    %comprobamos si al hinchar la esfera encontramos algún punto
además de nearestpoint,
    %si es así comprobamos si ambos puntos están al mismo lado de
    %la dirección de avance, si cada uno está a un lado dejamos de
hinchar.
        for i = 1:length(indices)
            iv = ptCloud.Location(indices(i),:) - BallonCenter ;
            if not (SameSide (AdvanceVector, nv, iv))
                EndInflate = true ;
                break
         end
        end
```

```
    end

    plot(BallonCenter(1), BallonCenter(2), 'bo', 'LineWidth', 1.5);
    circle (BallonCenter, BallonRadius);

    %detectamos si ha llegado al final o es demasiado estrecho como
para pasar

    if (findNeighborsInRadius(ptCloud,BallonCenter,SRadius))
        circle (BallonCenter, SRadius);
        disp(mensaje2);
        break
    end

    %actualizamos el punto inicial para el próximo movimiento
    pp = BallonCenter ;
    pp(1)=0;
    position=BallonCenter;

end ;
```

8.1.4. "Wall Sliding Balloon" tiempo real

```
for i = 6500:22:length(ranges(:,1))

    disp (i) ;
    advance=AdvanceVector;
    advance = advance / norm(advance) ;
    %pasamos información de punto actual y dirección para calcular
el mapa
    %alrededor del robot
    mapa=CreateMapa(i);
    %una vez calculado el mapa lo guardamos y cargamos
    save mapa.mat
    load('mapa');
    ptCloud = pointCloud(mapa);
    figure;
    hold on;
    plot (pp(1), pp(2), 'r>') ;
    plot (ptCloud.Location(:,1), ptCloud.Location(:,2),'k.') ;

    AdvanceAnterior=AdvanceVector;
    BallonCenter = pp + AdvanceAnterior*AdvanceStep ; % punto al
que nos dirigimos, tomando la última dirección
    BallonCenterSupuesto=BallonCenter;%almacenamos el punto al que
nos dirigiríamos inicialmente para calcular ángulos
    EndInflate = false ;

    %plot (BallonCenter(1), BallonCenter(2), 'r*') ;
    circle (pp, AdvanceStep);

    while not (EndInflate)

        BallonCenter = pp + AdvanceVector*AdvanceStep ;

        %Actualizamos el NearestPoint
        [indices,dists] = findNearestNeighbors
(ptCloud,BallonCenter,1) ;
        NearestPoint = ptCloud.Location(indices(1),:) ;

%Comprobamos que el nearestPoint está a la derecha, si no lo esta
%giramos la dirección con intención de encontrar uno más cercano
que se
        %encuentre a la derecha
        if not (OnTheRightSide (BallonCenter, AdvanceVector,
NearestPoint))
            AdvanceVector = (Rz * AdvanceVector')' ;
        else
            EndInflate = true ;
        end ;

    end ;

    [p1 p2 Sol] = SolveCircleIntersection (pp, AdvanceStep,
NearestPoint, BallonRadius) ;

    if (Sol==true)
        BallonCenter = NearestSol (BallonCenter, p1, p2) ;
```

```
        %av: vector de avance actual
        AdvanceVector = BallonCenter - pp ;
        AdvanceVector = AdvanceVector / norm(AdvanceVector) ;
    else
        AdvanceVector = AdvanceAnterior;
        AdvanceVector = AdvanceVector / norm(AdvanceVector) ;
    end;

    plot (BallonCenter(1), BallonCenter(2), 'bo') ;
    circle (BallonCenter, BallonRadius) ;

 pp = BallonCenter ;
    pp(1)=0;
    position=BallonCenter;

end ;
```

8.1.5. Funciones Matlab

- NearestSol:

```
function psol = NearestSol(p, p1, p2)
      %Elige el punto más cercano a p
      %   Detailed explanation goes here

      if norm(p1-p) > norm(p2-p)
         psol = p2 ;
      else
        psol = p1 ;
      end
end
```

- CreateMapa:

```
function [mapa] = CreateMapa(j)
      load('ranges2.mat');
      load('angles2.mat');

      for i =1:1:length(angles)
      %giro sobre z, para obtener la dirección de punto
      Rz = [cos(angles(1,i)) -sin(angles(1,i)) 0 ; sin(angles(1,i))
cos(angles(1,i)) 0 ; 0 0 1] ;

      %dirección del robot al punto a calcular
      direccionPunto = Rz * [1 0 0]';
       direccionPunto=direccionPunto/norm(direccionPunto);

      %posición en la que se encuentra el punto calculado
       posicionPunto=[0 0 0]'+direccionPunto*ranges(j,i);

            if (posicionPunto(1)>30)
                mapa(i,1)= 30;
             else
                   mapa(i,1)= posicionPunto(1);
            end

             if (posicionPunto(2)>30)
                   mapa(i,2)= 30;
             else
                  mapa(i,2)= posicionPunto(2);
             end

       mapa(i,3)= 0;
      end
end
```

- Circle:

```
function h = circle(p,r)
      th = 0:pi/50:2*pi;
      xunit = r * cos(th) + p(1);
      yunit = r * sin(th) + p(2);
      if (r==0.65)
          h = plot(xunit, yunit,'r--', 'LineWidth', 1.5);
      elseif(r==0.5)
          h = plot(xunit, yunit,'b-.', 'LineWidth', 1.5);
      else
          h = plot(xunit, yunit,'m', 'LineWidth', 1.5);
      end
end
```

- OnTheRightSide:

```
function result = OnTheRightSide (origin, divisorv, point)
  v1 = point - origin ;
  xv1 = cross (divisorv, v1) ;
  if xv1(3) < 0
      result = true ;
  else
      result = false ;
  end
end
```

- SameSide

```
function result = SameSide (divisorv, v1, v2)
%Decide si dos vectores están en el mismo lado de una divisoria
%   divisorv: vector que define la divisoria
%   v1, v2: los dos vectores

  xv1 = cross (divisorv, v1) ;
  xv2 = cross (divisorv, v2) ;
  if xv1(3)*xv2(3) > 0
      result = true ;
  else
      result = false ;
  end
end
```

- SolveCircleIntersection

```
function [ p1 p2 Sol ] = SolveCircleIntersection(c1, r1, c2, r2)
      % Obtiene los dos puntos de intersección de dos circunferencias
      %   Circunferencia1: centro c1, radio r1
      %   (x - xc1)^2 + (y - yc1)^2 = r1^2

      xc1 = c1 (1) ;
      yc1 = c1 (2) ;

      %   Circunferencia2: centro c2, radio r2
      %   (x - xc2)^2 + (y - yc2)^2 = r2^2

      xc2 = c2 (1) ;
```

```
        yc2 = c2 (2) ;

        epsilon = 0.001 ;

        if (xc1 - xc2)^2 < epsilon
            yp1 = (r1^2 - r2^2 + yc2^2 - yc1^2)/(2*(yc2 - yc1)) ;
            yp2 = yp1 ;

            xp1 = xc1 + sqrt(r1^2 - (yp1 - yc1)^2) ;
            xp2 = xc1 - sqrt(r1^2 - (yp1 - yc1)^2) ;
        else

            % x = -y*M + N

            M = (yc2 - yc1)/(xc2 - xc1) ;
            N = (r1^2 - r2^2 + xc2^2 - xc1^2 + yc2^2 - yc1^2)/(2*(xc2 -
        xc1)) ;

            % yp1 = (-b + sqrt(b^2 - 4*a*c))/(2*a)

            a = M^2 + 1 ;
            b = 2*(M*xc1 - M*N - yc1) ;
            c = N^2 + xc1^2 - 2*N*xc1 + yc1^2 - r1^2 ;

            yp1 = (-b + sqrt(b^2 - 4*a*c))/(2*a) ;
            xp1 = -yp1*M + N ;

            yp2 = (-b - sqrt(b^2 - 4*a*c))/(2*a) ;
            xp2 = -yp2*M + N ;
        end

        if (isreal(xp1) & isreal(xp2) & isreal(yp1) & isreal(yp2))
            p1 = [xp1 yp1 0] ;
            p2 = [xp2 yp2 0] ;
            Sol=true;
        else
            p1=[0 0 0] ;
            p2=[0 0 0] ;
            Sol=false;
        end;

    end
```

8.2. Código ROS

8.2.1. "Slide Balloon"

```
#include "SlideBalloon.h"
#include <boost/tuple/tuple.hpp>
#include <std_msgs/Int8.h>

    double max_laser_range= 30.0, laser_offset = 0, pos_ = -100;
    bool start = false;
    int count = 0;
    std::vector<tf::Vector3> points_;
    tf::Vector3 Nearest_Point;
    tf::Vector3 pp_(0,0,0);
    tf::Vector3 AdvanceVector_ (1,0,0);
    tf::Vector3 AdvanceVector_Anterior_ (1,0,0);
    tf::Vector3 BalloonCenter_(0,0,0);

SlideBalloon::SlideBalloon(ros::NodeHandle n) :
    n_(n)
{
    ros::NodeHandle private_node("~");
    private_node.param("mode", mode_, 1) ;
    private_node.param("timer_active", timer_active_, false);
    private_node.param("diferencial", diferencial_, 0.05);
    private_node.param("AdvanceStep", AdvanceStep_, 1.20);
    private_node.param("Security_Radius", Security_Radius_, 0.65);
    private_node.param("Ang_MaxGiro", Ang_MaxGiro_, 45.0);
    private_node.param("EndInflate", EndInflate_, false);
    private_node.param("distance", wall_distance_, 0.0);

    private_node.param<std::string>("global_reference_frame",
global_reference_frame_, "odom");
    private_node.param<std::string>("local_reference_frame",
local_reference_frame_, "laser_front");
    private_node.param<double>("freq_goal", freq_goal_, double(5));

    ros::NodeHandle public_node;

//Declaración de SUBSCRIBERS ----------------------------------------
    scan_subscriber_ = public_node.subscribe("/laser_front/scan",  1,
&SlideBalloon::scan_callback, this);
    if (mode_ == 0|| ! timer_active_){
        wall_subscriber_ = public_node.subscribe("/out/right", 10,
&SlideBalloon::wallRightCallback, this);
    }
//-------------------------------------------------------------------

// Declaración de PUBLISHERS ----------------------------------------
    goal_publisher_ =
public_node.advertise<geometry_msgs::PoseStamped>("goalStamped", 10);
    tick_publisher_ =
public_node.advertise<std_msgs::Int8>("SlideBalloon/alive", 10);
//-------------------------------------------------------------------

    sleep(2);
    if (not tf_.waitForTransform(global_reference_frame_,
local_reference_frame_, ros::Time(0), ros::Duration(15)))
    {
```

```
        ROS_ERROR_STREAM("Unable to find " << global_reference_frame_
<< " -> " << local_reference_frame_ << " transform");
        throw ros::Exception("Unable to find needed transform");
    }

//Declaración de SERVICES ------------------------------------------
    enable_service = n.advertiseService("SlideBalloon/enable",
&SlideBalloon::enableCallback, this);
//-----------------------------------------------------------------------

    have_goal_ = false;
    enabled_ = false;
    direction_straight_ = true;

    std::cerr << "Waiting transform" << std::endl;

//Obtenemos the initial pose
    try
    {
        tf::StampedTransform stamped_initial;
        tf_.lookupTransform(global_reference_frame_,
local_reference_frame_, ros::Time(0), stamped_initial);
        initial_pose_ = stamped_initial;
        current_pose_ = initial_pose_;
    } catch(tf::TransformException& e)
    {
        ROS_ERROR("%s", e.what());
        throw ros::Exception("Unable to set initial transform");
    }
    std::cerr << "Waiting for transform done." << std::endl;

}

//Función que actualiza objetivo
void SlideBalloon::publishGoalNoTimer()
{
    ROS_INFO("publish goal no timer");
    updatePose();

    mtx_callback.lock();

    geometry_msgs::PoseStamped goal_stamped_msg;
    goal_stamped_msg.header.stamp = ros::Time::now();

    if (mode_ == 0)
    {
      ROS_INFO("mode 0");
        goal_stamped_msg.header.frame_id = local_reference_frame_;
        geometry_msgs::Pose goal_msg;
        tf::poseTFToMsg(current_goal_, goal_msg);
        goal_stamped_msg.pose = goal_msg;
    }
    else if (mode_ == 1)
    {
      ROS_INFO("mode 1");

      goal_stamped_msg.header.frame_id = global_reference_frame_;

//obtenemos el quaternio del mensaje para obtener la orientación del
robot
      tf::Quaternion q = current_pose_.getRotation();
```

```
      tf::Matrix3x3 m (q);
      double roll, pitch, yaw;
      m.getRPY(roll, pitch, yaw);

      AdvanceVector_[0] = cos(yaw);
      AdvanceVector_[1] = sin(yaw);
      AdvanceVector_[2] = 0;
      AdvanceVector_ = AdvanceVector_ / AdvanceVector_.length();
      printf("AdvanceVector_: %f\n",AdvanceVector_[0]);
      printf("AdvanceVector_: %f\n",AdvanceVector_[1]);
      printf("AdvanceVector_: %f\n",AdvanceVector_[2]);

//obtenemos la posición actual del robot
      pp_ = current_pose_.getOrigin();
      pp_[2] = 0;
      printf("pp: %f\n",pp_[0]);
      printf("pp: %f\n",pp_[1]);
      printf("pp: %f\n",pp_[2]);

//calculamos el objetivo provisional
      BalloonCenter_ = pp_ + AdvanceVector_ * AdvanceStep_;
      printf("BalloonCenter_: %f\n",BalloonCenter_[0]);
      printf("BalloonCenter_: %f\n",BalloonCenter_[1]);
      printf("BalloonCenter_: %f\n",BalloonCenter_[2]);

      tf::Vector3 BalloonCenterSupuesto = BalloonCenter_;
      bool endInflate = false;

      do{
//Calculamos el punto más cercano al punto objetivo (BalloonCenter)
del entorno
            Nearest_Point = geometry::nearestPoint(points_,
BalloonCenter_);
            double r = BalloonCenter_.distance(Nearest_Point);
//distancia entre Balloocenter y nearestpoint
            double l = pp_.distance(Nearest_Point);

//condición que determina parada de hinchado de globo
            if(l+AdvanceStep_ < r+diferencial_)
            {
                  tf::Vector3 farest_point_ =
geometry::farestpoint(pp_, points_);
                  AdvanceVector_ = BalloonCenter_ - farest_point_;
                  AdvanceVector_ = AdvanceVector_ /
AdvanceVector_.length();

                  BalloonCenter_ = pp_ + AdvanceVector_ *
AdvanceStep_;
                  break;
            }

            double BalloonRadius = r+diferencial_;
//Calculamos intersección entre círculos para corregir la posición
objetivo
            BalloonCenter_ = geometry::intersectionCircles(pp_,
AdvanceStep_, Nearest_Point, BalloonRadius, BalloonCenter_);
            AdvanceVector_ = BalloonCenter_ - pp_;
            AdvanceVector_ = AdvanceVector_ / AdvanceVector_.length();
```

```
//comprobamos si al hinchar la esfera encontramos algún punto además
de nearestpoint, si es así comprobamos si ambos puntos están al mismo
lado de la dirección de avance, si cada uno está a un lado dejamos de
hinchar.
                  std::vector<tf::Vector3> PuntosEnRadio =
geometry::pointsInRadius(points_, BalloonCenter_, BalloonRadius);

            tf::Vector3 nv = Nearest_Point - BalloonCenter_;
            for (unsigned int i = 0; i < PuntosEnRadio.size(); i++)
            {
                 if (PuntosEnRadio[i] != Nearest_Point)
                 {
                        tf::Vector3 iv = PuntosEnRadio[i] -
BalloonCenter_;
                        if (not geometry::sameside (AdvanceVector_,
nv, iv))
                  {
                  endInflate = true;
                  break;
                  }
                 }
            }
      }while(not endInflate);

//Comprobamos si hay algún punto dentro del radio de seguridad
       std::vector<tf::Vector3> PuntosEnRadio =
geometry::pointsInRadius(points_, BalloonCenter_, Security_Radius_);
         if ( not (PuntosEnRadio.empty()))
         {
            enabled_ = false;
            printf("Demasiado cerca de la pared como para avanzar\n");
         }
//introducimos la información en el mensaje goal_stamped_msg que se va
a publicar como próxima meta a alcanzar
            printf("BalloonCenter_: %f\n",BalloonCenter_[0]);
            printf("BalloonCenter_: %f\n",BalloonCenter_[1]);
            printf("BalloonCenter_: %f\n",BalloonCenter_[2]);
            printf("\n");

            printf("Nearest_Point: %f\n",Nearest_Point[0]);
            printf("Nearest_Point: %f\n",Nearest_Point[1]);
            printf("Nearest_Point: %f\n",Nearest_Point[2]);
            printf("\n");

      goal_stamped_msg.pose.position.x = BalloonCenter_[0];
      goal_stamped_msg.pose.position.y = BalloonCenter_[1];
      goal_stamped_msg.pose.position.z = BalloonCenter_[2];

      tf::Quaternion q2 (AdvanceVector_, 0);
      tf::quaternionTFToMsg(q2, goal_stamped_msg.pose.orientation);

            printf("AdvanceVector_: %f\n",AdvanceVector_[0]);
            printf("AdvanceVector_: %f\n",AdvanceVector_[1]);
            printf("AdvanceVector_: %f\n",AdvanceVector_[2]);
      AdvanceVector_Anterior_ = AdvanceVector_;
    }
    if (enabled_) {
      goal_publisher_.publish(goal_stamped_msg);
            ROS_INFO("objetivo publicado");
    }
    std_msgs::Int8 tick;
```

```
    tick_publisher_.publish(tick);
    mtx_callback.unlock();
}

bool SlideBalloon::updatePose()
{
    ROS_INFO_ONCE("update pose");
    try
    {
        tf::StampedTransform stamped_current;
        tf_.lookupTransform(global_reference_frame_,
local_reference_frame_, ros::Time(0), stamped_current);
        current_pose_ = stamped_current;
    }
    catch(tf::TransformException& e)
    {
        ROS_WARN("%s", e.what());
        return false;
    }
    return true;
}
```

8.2.2. "Wall Slide Balloon"

```
#include "WallSlideBalloon.h"
#include <boost/tuple/tuple.hpp>
#include <std_msgs/Int8.h>

    double max_laser_range= 30.0, laser_offset = 0;
    std::vector<tf::Vector3> points_, right_points_, left_points_;
    tf::Vector3 pp_(0,0,0);
    tf::Vector3 Nearest_Point;
    tf::Vector3 AdvanceVector_(0,0,0);
    tf::Vector3 AdvanceVector_Anterior_(0,0,0);
    tf::Vector3 BalloonCenter_(0,0,0);
    tf::Vector3 farest_point_ (0,0,0);

WallSlideBalloon::WallSlideBalloon(ros::NodeHandle n) :
    n_(n)
{
    ros::NodeHandle private_node("~");

    private_node.param("mode", mode_, 1) ;
    private_node.param("timer_active", timer_active_, false);

    private_node.param("AdvanceStep", AdvanceStep_, 1.5);
    private_node.param("Balloon_Radius", Balloon_Radius_,1.5);
    private_node.param("Security_Radius", Security_Radius_,0.65);
    private_node.param("Ang_MaxGiro", Ang_MaxGiro_, 45.0);
    private_node.param("EndInflate", EndInflate_, false);

    private_node.param<std::string>("global_reference_frame",
global_reference_frame_, "odom");
    private_node.param<std::string>("local_reference_frame",
local_reference_frame_, "laser_front");
    private_node.param<double>("freq_goal", freq_goal_, double(5));

    ros::NodeHandle public_node;

//Declaración de SUBSCRIBERS ----------------------------------------
    scan_subscriber_ = public_node.subscribe("/laser_front/scan",  1,
&WallSlideBalloon::scan_callback, this);
    if (mode_ == 0|| ! timer_active_){
        wall_subscriber_ = public_node.subscribe("/out/right", 10,
&WallSlideBalloon::wallRightCallback, this);
    }
//-------------------------------------------------------------------

//Declaración de PUBLISHERS -----------------------------------------
    goal_publisher_ =
public_node.advertise<geometry_msgs::PoseStamped>("goalStamped", 10);
    tick_publisher_ =
public_node.advertise<std_msgs::Int8>("WallSlideBalloon/alive", 10);
//-------------------------------------------------------------------


    sleep(2);

    if (not tf_.waitForTransform(global_reference_frame_,
local_reference_frame_, ros::Time(0), ros::Duration(15)))
    {
```

```
        ROS_ERROR_STREAM("Unable to find " << global_reference_frame_
<< " -> " << local_reference_frame_ << " transform");
        throw ros::Exception("Unable to find needed transform");
    }

//Declaración de SERVICES ------------------------------------------
    side_service = n.advertiseService("WallSlideBalloon/side",
&WallSlideBalloon::sideCallback, this);
    enable_service = n.advertiseService("WallSlideBalloon/enable",
&WallSlideBalloon::enableCallback, this);
//-----------------------------------------------------------------------

    enabled_ = false;
    side_ = true;
    side_changed = false;
    have_goal_ = false;
    direction_straight_ = true;
    std::cerr << "Waiting transform" << std::endl;

//Obtenemos the initial pose
    try
    {
        tf::StampedTransform stamped_initial;
        tf_.lookupTransform(global_reference_frame_,
local_reference_frame_, ros::Time(0), stamped_initial);
        initial_pose_ = stamped_initial;
        current_pose_ = initial_pose_;
    } catch(tf::TransformException& e)
    {
        ROS_ERROR("%s", e.what());
        throw ros::Exception("Unable to set initial transform");
    }
    std::cerr << "Waiting for transform done." << std::endl;
}

//Función que actualiza objetivo
void WallSlideBalloon::publishGoalNoTimer()
{
    ROS_INFO("publish goal no timer");
    updatePose();

    mtx_callback.lock();

    geometry_msgs::PoseStamped goal_stamped_msg;
    goal_stamped_msg.header.stamp = ros::Time::now();

    if (mode_ == 0)
    {
      ROS_INFO("mode 0");
        goal_stamped_msg.header.frame_id = local_reference_frame_;
        goal_stamped_msg.pose.position.x =
current_pose_.getOrigin().x() + AdvanceStep_;
      tf::Quaternion q = tf::createIdentityQuaternion();
        tf::quaternionTFToMsg(q,goal_stamped_msg.pose.orientation);
    }
    else if (mode_ == 1)
    {
      ROS_INFO("mode 1");
         goal_stamped_msg.header.frame_id = global_reference_frame_;
```

```
//obtenemos el quaternio del mensaje para obtener la orientación del
robot que va a AdvanceVector
         tf::Quaternion q = current_pose_.getRotation();

         tf::Matrix3x3 m(q);
         double roll, pitch, yaw;
         m.getRPY(roll, pitch, yaw);

         AdvanceVector_[0] = cos(yaw);
         AdvanceVector_[1] = sin(yaw);
         AdvanceVector_[2] = 0;
         AdvanceVector_ = AdvanceVector_ / AdvanceVector_.length();
         printf("AdvanceVector_: %f\n",AdvanceVector_[0]);
         printf("AdvanceVector_: %f\n",AdvanceVector_[1]);
         printf("AdvanceVector_: %f\n",AdvanceVector_[2]);

//obtenemos la posición actual del robot
         pp_ = current_pose_.getOrigin();
         pp_[2] = 0;
         printf("pp: %f\n",pp_[0]);
         printf("pp: %f\n",pp_[1]);
         printf("pp: %f\n",pp_[2]);

 //calculamos el objetivo provisional
         BalloonCenter_ = pp_ + AdvanceVector_ * AdvanceStep_;
         printf("BalloonCenter_: %f\n",BalloonCenter_[0]);
         printf("BalloonCenter_: %f\n",BalloonCenter_[1]);
         printf("BalloonCenter_: %f\n",BalloonCenter_[2]);
         bool endInflate = false;

         do{
//Volvemos a calcular eL objetivo en caso de que la direción haya
cambiado
            BalloonCenter_ = pp_ + AdvanceVector_ * AdvanceStep_;
            Nearest_Point = geometry::nearestPoint(points_,
BalloonCenter_);

//Comprobamos que pared estamos usando como guía
            if(side_)
            {
//Comprobamos si nearest point esta a la derecha del objetivo
                bool ontheRIGTH =
geometry::onTheRightSide(AdvanceVector_, BalloonCenter_,
Nearest_Point);

//Si no está a la derecha corregimos la dirección girando hacia la
derecha buscando que nearest este a la derecha.
               if (not ontheRIGTH)
                   {
                  double AngularStep = -0.0349 ;   // 2 grados a la
derecha
                  double cs = cos(AngularStep);
                  double sn = sin(AngularStep);
                  tf::Matrix3x3 Rdcha_(cs, -sn, 0, sn,  cs, 0, 0, 0,
1);
                  tf::Matrix3x3 AdvVector ( AdvanceVector_[0], 0, 0,
AdvanceVector_[1], 0, 0,  AdvanceVector_[2], 0, 0 );

                  Rdcha_ *= AdvVector;
                  AdvanceVector_[0] = Rdcha_[0][0];
                  AdvanceVector_[1] = Rdcha_[1][0];
```

```
                    AdvanceVector_[2] = Rdcha_[2][0];
                      }
            }
            else
            {

//Comprobamos si nearest point está a la derecha del objetivo
                    bool ontheRIGTH =
geometry::onTheRightSide(AdvanceVector_, BalloonCenter_,
Nearest_Point);

//Si está a la derecha corregimos la dirección girando hacia la
izquierda buscando que nearest esté a la izquierda.
                if (ontheRIGTH)
                {
                  double AngularStep = 0.0349 ;   // 2 grados a la izq
                  double cs = cos(AngularStep);
                  double sn = sin(AngularStep);
                  tf::Matrix3x3 Rizq_ (cs, -sn, 0, sn, cs, 0, 0, 0,1);
                  tf::Matrix3x3 AdvVector (AdvanceVector_[0], 0, 0,
AdvanceVector_[1], 0, 0,  AdvanceVector_[2], 0, 0 );

                  Rizq_ *= AdvVector;
                  AdvanceVector_[0] = Rizq_[0][0];
                  AdvanceVector_[1] = Rizq_[1][0];
                  AdvanceVector_[2] = Rizq_[2][0];
            }
         }while(not endInflate);
//Una vez que nearest se encuentra en el lado correcto calculamos el
objetivo que resulta de la intersección entre los círculos de avance y
ballon_radius que la distancia fija a la pared

         BalloonCenter_ = geometry::intersectionCircles(pp_,
AdvanceStep_, Nearest_Point, Balloon_Radius_, BalloonCenter_);

            printf("BalloonCenter_: %f\n",BalloonCenter_[0]);
            printf("BalloonCenter_: %f\n",BalloonCenter_[1]);
            printf("BalloonCenter_: %f\n",BalloonCenter_[2]);

         AdvanceVector_ = BalloonCenter_ - pp_;
         AdvanceVector_ = AdvanceVector_ / AdvanceVector_.length();

            printf("AdvanceVector_: %f\n",AdvanceVector_[0]);
            printf("AdvanceVector_: %f\n",AdvanceVector_[1]);
            printf("AdvanceVector_: %f\n",AdvanceVector_[2]);
//Comprobamos que en el radio de seguridad no haya ningun punto del
entorno
         std::vector<tf::Vector3> PuntosEnRadio =
geometry::pointsInRadius(points_, BalloonCenter_, Security_Radius_);
         if ( not (PuntosEnRadio.empty()))
         {
            enabled_ = false;
         }
    }
//publicamos la información del objetivo
      goal_stamped_msg.pose.position.x = BalloonCenter_[0];
      goal_stamped_msg.pose.position.y = BalloonCenter_[1];
      goal_stamped_msg.pose.position.z = BalloonCenter_[2];

      tf::Quaternion q2 (AdvanceVector_, 0);
      tf::quaternionTFToMsg(q2, goal_stamped_msg.pose.orientation);
```

```
    if (enabled_) {
        goal_publisher_.publish(goal_stamped_msg);
    }
    std_msgs::Int8 tick;
    tick_publisher_.publish(tick);
    mtx_callback.unlock();
}


bool WallSlideBalloon::updatePose()
{
    ROS_INFO_ONCE("update pose");

    try
    {
        tf::StampedTransform stamped_current;
        tf_.lookupTransform(global_reference_frame_,
local_reference_frame_, ros::Time(0), stamped_current);
        current_pose_ = stamped_current;
    }
    catch(tf::TransformException& e) {
        ROS_WARN("%s", e.what());
        return false;
    }
    return true;
}
```

8.2.3. Funciones de ROS "geometry"

```
#include "geometry.h"

tf::Vector3 geometry::intersectionCircles(const tf::Vector3& c1,
double r1, const tf::Vector3& c2, double r2, const tf::Vector3& p)
{
//Calcula los dos puntos que se originan de la intersección entre dos
puntos
    double base, exp;
//Obtiene los dos puntos de intersección de dos circunferencias
//Circunferencia1: centro c1, radio r1
//(x - xc1)^2 + (y - yc1)^2 = r1^2
    double xc1 = c1 [0];
    double yc1 = c1 [1];
//Circunferencia2: centro c2, radio r2
//(x - xc2)^2 + (y - yc2)^2 = r2^2
    double xc2 = c2 [0];
    double yc2 = c2 [1];
    double epsilon = 0.001;

    double yp1, yp2, xp1, xp2, M, N, a, b, c;
    std::complex<double> yyp1, xxp1, yyp2, xxp2;
    base = xc1 - xc2;

    if ( pow(base,2) < epsilon)
    {
         yp1 = (pow(r1,2) - pow(r2,2) + pow(yc2,2) - pow(yc1,2)
)/(2*(yc2 - yc1));
         yp2 = yp1;

       base = yp1 - yc1;

         xp1 = xc1 + sqrt(pow(r1,2) - pow(base,2));
         xp2 = xc1 - sqrt(pow(r1,2) - pow(base,2));
    }
    else
    {
//x = -y*M + N

         M = (yc2 - yc1)/(xc2 - xc1) ;
         N = (pow(r1,2) - pow(r2,2) + pow(xc2,2) - pow(xc1,2) +
pow(yc2,2) - pow(yc1,2) )/(2*(xc2 - xc1));

//yp1 = (-b + sqrt(b^2 - 4*a*c))/(2*a)

         a = pow(M,2) + 1 ;
         b = 2*(M*xc1 - M*N - yc1) ;
         c = pow(N,2) + pow(xc1,2) - 2*N*xc1 + pow(yc1,2) - pow(r1,2);

         yp1 = (-b + sqrt( pow(b,2) - 4*a*c))/(2*a);
         xp1 = -yp1*M + N ;

         yp2 = (-b - sqrt( pow(b,2) - 4*a*c))/(2*a);
         xp2 = -yp2*M + N ;
    }
    tf::Vector3 p1 (xp1,yp1,0);
    tf::Vector3 p2 (xp2,yp2,0);
    tf::Vector3 psol;
    if ( p1.distance(p) > p2.distance(p) )
```

```
    {
          psol = p2;
    }
    else
    {
          psol = p1;
    }
    return psol;
}

bool geometry::sameside(const tf::Vector3& divisor, const tf::Vector3&
v1, const tf::Vector3& v2)
{
    tf::Vector3 xv1 = divisor;
    tf::Vector3 xv2 = divisor;

    xv1.cross (v1);
    xv2.cross (v2);
    bool result;

    if (xv1[2]*xv2[2] > 0)
    {
        result = true;
    }
    else
    {
        result = false;
    }
    return result;
}

tf::Vector3 geometry::nearestPoint(const std::vector<tf::Vector3>&
points,const tf::Vector3& center)
{
//Cálculo del punto más cercano a "center" del set de puntos
    double distancia=100;
    tf::Vector3 NearestPoint;
    for (unsigned int i = 0; i < points.size(); i++)
    {
      tf::Vector3 p = points[i];
      double x = center.distance( p );
      if (x < distancia )
      {
          distancia = x;
          NearestPoint = points[i];
        }
    }
    return NearestPoint;
}

std::vector<tf::Vector3> geometry::pointsInRadius(const
std::vector<tf::Vector3>& points, const tf::Vector3& center, double
radio)
{
//Devuelve los puntos dentro de un radio
     std::vector<tf::Vector3> puntos;
     for (unsigned int i = 0; i < points.size(); i++)
     {
      tf::Vector3 p = points[i];
      double x = center.distance( p );
      if (x <= radio)
```

```
      {
          puntos.push_back(p);
        }
      }
      return puntos;
}

bool geometry::onTheRightSide(const tf::Vector3& divisor, const
tf::Vector3& origin, const tf::Vector3& point)
{
//Decide si un punto está a la derecha de una recta definida por un
punto y un vector director
//divisorv: vector director
//origin: punto de la recta
//point: punto a investigar

      tf::Vector3 v1 = point - origin;
      double valor;
      tf::Vector3 divisor1 = divisor;
      valor = divisor1[0]*v1[1] - divisor1[1]*v1[0];
      bool result;
      printf("componente z de la mult.: %f\n",valor);
      if (valor < 0)
      {
          result = true;
      }
      else
      {
          result = false;
      }
  return result;
}

tf::Vector3 geometry::farestpoint(const tf::Vector3& center, const
std::vector<tf::Vector3>& points)
{
//Calcula el punto más lejano del set de puntos
      double distancia=0;
      tf::Vector3 maslejano;
      for (unsigned int i = 0; i < points.size(); i++)
      {
       tf::Vector3 p = points[i];
       double x = center.distance( p );
       if (x > distancia)
       {
           distancia = x;
           maslejano = p;
         }
      }
      return maslejano;
}
```

8.2.4. Funciones de subscripción a “topic”

```
//Callback que actualiza la posición de los puntos del entorno
respecto del robot

void WallSlideBalloon::scan_callback(const sensor_msgs::
LaserScan::ConstPtr& msg)
{
    mtx_callback.lock();

    points_.clear();

    for ( int i = 0; i < msg->ranges.size(); i++) {
            double angle = msg->angle_min + double(i) * msg-
>angle_increment - laser_offset;
            double x = msg->ranges.at(i) * cos(angle);
            double y = msg->ranges.at(i) * sin(angle);
       tf::Vector3 p;
   if (x < max_laser_range && y < max_laser_range) {
       p[0] = x;
       p[1] = y;
       p[2] = 0;
       points_.push_back(p);
        }
    }
    mtx_callback.unlock();
}

void WallSlideBalloon::wallRightCallback(const wally::LineVector&
msg)
{
    if (not updatePose())
        return;
ROS_INFO("cb2");
    double _AdvanceStep_ = direction_straight_? AdvanceStep_:
AdvanceStep_;

    if (msg.lines.size() > 0) //my wall is visible
    {
        ROS_ERROR("See the wall");
        assert (-std::abs(msg.lines[0].angle) < M_PI/4);
        tf::Pose wXr, wXg, lXg;

        wXg.setOrigin(tf::Point(_AdvanceStep_, wall_distance_,
0.0));
        wXg.setRotation(tf::createIdentityQuaternion());

        wXr.setOrigin(tf::Point(0.0, -msg.lines[0].distance, 0.0));
        wXr.setRotation(tf::createQuaternionFromYaw(-
msg.lines[0].angle));
        lXg.setRotation(tf::createQuaternionFromYaw(-
msg.lines[0].angle));

        mtx_callback.lock();
        current_goal_ = lXg;

        have_goal_ = true;
        mtx_callback.unlock();
    }
    else
    {
```

```
            ROS_ERROR("Cannot see the wall");
        }

        if (!timer_active_ )
                publishGoalNoTimer();
    }
```

8.2.5. Funciones de “Services”

```
//elección del lado
bool WallSlideBalloon::sideCallback(std_srvs::SetBool::Request
&req,std_srvs::SetBool::Response &res){
    side_ = req.data;
    res.success = true;
    res.message = side_?"RIGHT":"LEFT";
    side_changed = true;
    return true;
}

//permitir navegación WallSlideBalloon/SlideBalloon
bool WallSlideBalloon::enableCallback(std_srvs::SetBool::Request
&req,std_srvs::SetBool::Response &res){
    enabled_ = req.data;
    res.success = true;
    res.message = enabled_?"ENABLED":"DISABLED";
    return true;
}
```