Universidad Zaragoza
1542

TRABAJO DE FIN DE GRADO

Diseño e implementación de la técnica de navegación “Sliding Balloon” mejorada para un barco autónomo.

Design and implementation of the improved “Sliding Balloon” navigation technique for an autonomous boat.

Autora

Lidia Sánchez Olalla

Director

José Luis Villarroel Salcedo

Ingeniería Electrónica y Automática, curso 2021-2022

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Resumen

El propósito de este proyecto es mejorar una técnica de navegación, diseñada para robots terrestres, para su adaptación a un barco autónomo. Esta técnica, llamada “Sliding Balloon”, se basa en el cálculo de un globo virtual justo delante del robot con el que detectar posibles obstáculos en el entorno y así evitarlos buscando siempre el camino más seguro.

El principal interés de este proyecto está en hacer que un pequeño barco sea capaz de recorrer, de manera autónoma, galerías subterráneas inundadas y pozos. Con ello, y gracias a un sensor LiDAR, se podrá obtener una reconstrucción 3D en forma de nube de puntos sin que una persona necesite entrar en la galería.

Para comenzar con esta tarea, primero se realiza un estudio de algunas de las técnicas de navegación con evitación de obstáculos ya existentes y se hace hincapié en el estudio y comprensión de la técnica “Sliding Balloon”, de la que se va a partir.

A continuación, se propondrán mejoras a algunos de los problemas e inconvenientes principales de la técnica original, además de adaptarla al movimiento de un barco. La técnica final desarrollada, “Sliding Balloon V2”, será descrita por pasos y mediante algunas imágenes para su mejor comprensión.

Tras desarrollar la técnica por pasos, se probará su correcto funcionamiento mediante simulaciones en MATLAB y, posteriormente, se utilizará la herramienta ROS para poder utilizar el algoritmo sobre un barco real.

Por último, habrá unas pruebas finales donde se pondrá en funcionamiento la técnica mejorada, siendo probada en entornos reales.

Tabla de contenido

Tabla de figuras

1. Introducción

Este Trabajo Fin de Grado ha sido realizado en el seno del Grupo de Robótica, Percepción y Tiempo Real (RoPerT) de la Universidad de Zaragoza y dentro del marco de los proyectos Autonomous or semi-autonomous Robot Deployment for Underground Applications (ARDUA, PID2019-105390RB-I00, Ministerio de Ciencia e Innovación, Proyectos de I+D+i Retos Investigación) y "Estudio e Intervención en Cueva Román (CLUNIA, Burgos)" financiado por la Diputación Provincial de Burgos.

El objetivo de este trabajo es utilizar la técnica de navegación a tiempo real, "Sliding Balloon", sobre un barco autónomo para el estudio de lagunas o ríos subterráneos.

En concreto, la cueva Román, en Burgos, que se caracteriza por tener un terreno hostil, lleno de barro, estrecheces y derrumbes. Esta cueva cuenta con restos arqueológicos muy antiguos que podrían verse afectados por la entrada del ser humano en la cueva. Así, se buscan maneras de acceder a la cueva de la forma menos invasiva posible, con la doble finalidad de preservar los restos y mapear la cueva de forma segura.

Esta técnica de navegación, pensada originalmente para robots terrestres, será la base de la que partir en este trabajo. Se propondrán soluciones para algunos de los casos particulares más importantes de mal funcionamiento y se adaptará la técnica con el fin de convertirse en una técnica de navegación apta para barcos autónomos.

1.1. Objetivos

Los objetivos definidos para realizar este trabajo son los siguientes:

- Estudio en profundidad de la técnica de navegación "Sliding Balloon" 2D, tanto sus ventajas, como sus principales problemas.
- Desarrollo e implementación de mejoras para la técnica.
- Adaptación y simulación del nuevo algoritmo de navegación sobre MATLAB.
- Implementación del algoritmo en C y ROS.
- Prueba del algoritmo sobre escenarios reales.

1.2. Recursos utilizados

Para llevar a cabo este trabajo han sido necesarios los siguientes recursos tanto hardware como software:

- Ordenador

- USV RoboBoat: Se trata de un pequeño barco de cebo (Carp Madness XXL) con un casco de plástico cubierto por una capa de fibra de carbono. Está dotado de dos hélices diferenciales alimentadas por sendos motores con regulador de velocidad que funcionan bajo señales de tipo PWM. En el casco se han integrado distintos módulos de alto nivel (Fig. 1):
  - IMU Phidgets Spatial 3/3/3, se trata de una unidad de medida de inercia, que cuenta con brújula, giróscopo y acelerómetro en los tres ejes y permite obtener la medida de aceleraciones, giros e inclinación del barco con precisión.

- VELODYNE Puck LITE LiDAR, un sensor de haz de luz con un rango de hasta 100m que permitirá la toma de datos del entorno en todo momento.

- JETSON NANO, placa que hace de soporte para el sistema operativo (Linux UBUNTU 18.04.5 LTS) y en la que se lanzará el programa ROS Melodic.

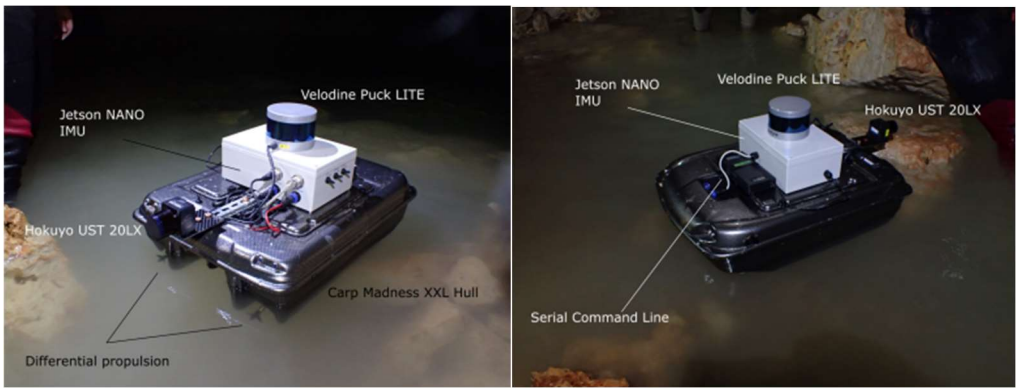

FIG. 1 VISTAS TRASERA Y FRONTAL DEL BARCO USV ROBOBOAT Y SUS PARTES

- MATLAB: Es un entorno de desarrollo integrado muy utilizado en el ámbito de la ingeniería. Esta herramienta se utilizará en este trabajo para la implementación de algoritmos, cálculos, representaciones gráficas y simulaciones. Todo ello haciendo uso de algunas de sus *toolboxes*, diseñadas para el tratamiento de nubes de puntos y robótica, en concreto la *Computer Vision Toolbox* y la *Robotics Toolbox*.

- ROS (Robot Operating System): Es un software de programación de código abierto dedicado principalmente al desarrollo de aplicaciones para robots. Cuenta con multitud de librerías dedicadas a la robótica y con varias herramientas muy útiles para la visualización de datos y recopilación de información.

- RVIZ: Es una de las herramientas con las que cuenta el software ROS. Con RVIZ se lograrán visualizar, en tiempo real, los datos tomados por el LiDAR del entorno, así como otras medidas dadas por el algoritmo de navegación. Además, esta herramienta permite la recopilación de datos de interés durante el funcionamiento, como pueden ser las velocidades, localizaciones… y cualquier dato que se pueda obtener.

2. Análisis del estado del arte

Hoy en día todo buen robot móvil autónomo debe contar con una técnica de navegación. La navegación autónoma requiere dos aspectos esenciales que son la planificación y la reacción.

La planificación de trayectorias se basa en buscar la ruta más eficiente y segura que debe seguir un robot para llegar a su objetivo. Este es un campo que ya ha sido estudiado extensamente, dado el interés en robots de tipo manipulador. Sin embargo, el interés en que los robots sean capaces de moverse en entornos más dinámicos hace necesario un estudio más profundo en la reacción. La rección es la capacidad de un robot para detectar posibles obstáculos en su recorrido y poder evitarlos, recalculando la ruta a seguir para llegar a su meta original.

La técnica que se va a desarrollar a lo largo de este trabajo se centrará en utilizar un método de evitación de obstáculos centrado en el seguimiento de muros.

Algunas de las técnicas de navegación existentes dedicadas a la evitación de obstáculos son las siguientes:

- **"Bug Algorithm"**: Con este algoritmo, un robot es capaz de evitar los obstáculos que se encuentra circunnavegándolos. En una primera versión, cuando el robot se encuentra con el obstáculo, lo rodea por completo y parte posteriormente desde el punto que más cerca esté de su meta. Es una segunda versión, algo más eficiente, el robot rodea el obstáculo hasta encontrar un punto desde el que partir de forma segura hacia la meta sin tener que circunnavegar el obstáculo por completo. [1]

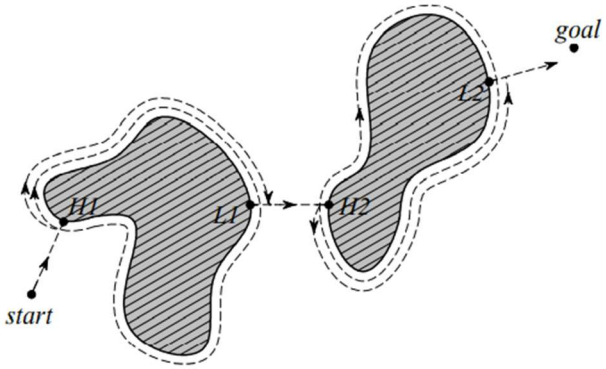

FIG. 2 EJEMPLO DEL BUG ALGORITHM

- **Campos de potencial**: En este método, se crea artificialmente un campo de potencial de acuerdo con las características del entorno. Siendo los obstáculos y las paredes los puntos con mayor potencial y la meta, el mínimo. Se trata de que el robot se mueva desde donde esté siguiendo una trayectoria de potenciales descendientes, como una pelota que rueda por una colina. De esta manera se dice que el robot se ve atraído por potenciales bajos (objetivo) y repelido por potenciales altos (obstáculos). [1]

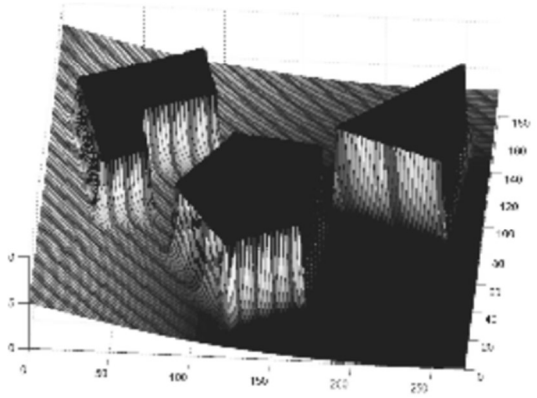

FIG. 3 EJEMPLO DE CAMPO DE POTENCIAL CON OBSTÁCULOS

- **"Nearness Diagram"**: Estos "diagramas de cercanía" son utilizados en conjunto con otras técnicas de navegación para encontrar las relaciones entre la posición del robot, los obstáculos y el objetivo. Este método se basa en la estrategia de "dividir y conquistar" ya que se divide el espacio en varias regiones (Fig. 4) según los obstáculos detectados por el robot en cada momento. De esta manera se busca el camino más seguro hacia el objetivo tomando decisiones sobre las acciones según un radio de seguridad y dependiendo siempre de la situación de los obstáculos. [2]

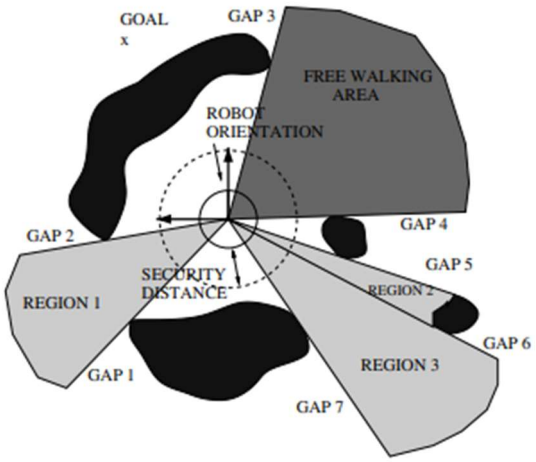

FIG. 4 EJEMPLO REGIONES NEARNESS DIAGRAM

- **"Bubble Band Technique"**: Se basa en definir una "cadena de burbujas" que configuran el espacio en el que el robot podría moverse libremente sin colisionar con el entorno. Al encontrar obstáculos en la trayectoria, las burbujas se minimizarían, reduciendo así el espacio de movimiento disponible y evitando las posibles colisiones. [1]

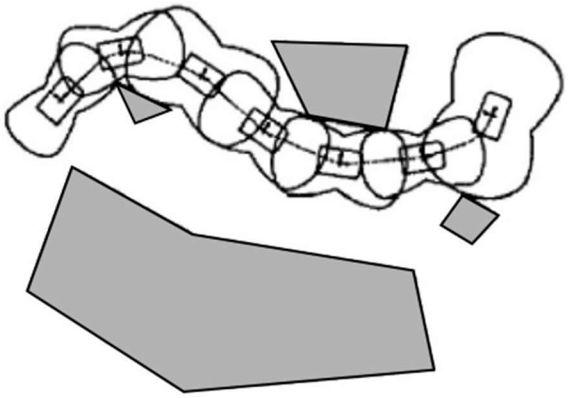

FIG. 5 EJEMPLO DE LA TÉCNICA BUBBLE BAND

- **"Sliding Balloon"**: Este es el método sobre el que se realizarán las modificaciones que son objeto de este trabajo. Esta técnica de navegación autónoma consiste en la creación de un "globo" cercano al robot hacia cuyo dentro debe dirigirse el robot. Este globo se calcula teniendo siempre en cuenta las características del entorno que rodea al robot en todo momento y permite una navegación segura evitando posibles colisiones. [3]

3. La técnica de navegación “Sliding Balloon”

3.1. Fundamentos y funcionamiento de la técnica.

La técnica de navegación “Sliding Balloon” (en español, globo deslizante), es una técnica de navegación 2D planteada en primera instancia para su aplicación en robots terrestres. Consiste principalmente en el cálculo de una circunferencia o globo de seguridad, “lanzado por delante del robot”, con el fin de evitar obstáculos o colisiones con el entorno en el que se mueve.

Desde el inicio, se le da al robot una dirección a seguir, un radio de seguridad y un radio de avance, que será la distancia que deba avanzar en cada paso. Con esta dirección y radio de avance, un punto, delante del robot, será el supuesto centro del globo. A partir de este punto y teniendo en cuenta las características del entorno más cercanas a él, se calcula, sobre el radio de avance que rodea al robot, el globo de mayor tamaño posible. El centro de este globo será el siguiente punto al que avanzar y el vector que une el robot con este punto, la nueva dirección a seguir.

Así, conforme el robot avanza en su trayectoria, este globo se infla y se desinfla de acuerdo con las características del entorno y posibles obstáculos, calculando en cada iteración el siguiente punto del entorno al que deberá desplazarse (Fig. 6).

De esta manera se busca siempre la trayectoria más segura, teniendo siempre en cuenta el radio de seguridad mínimo como medida de seguridad excepcional. Si algún obstáculo o zona del entorno quedase dentro de este radio, el robot se detendría por completo.[3]

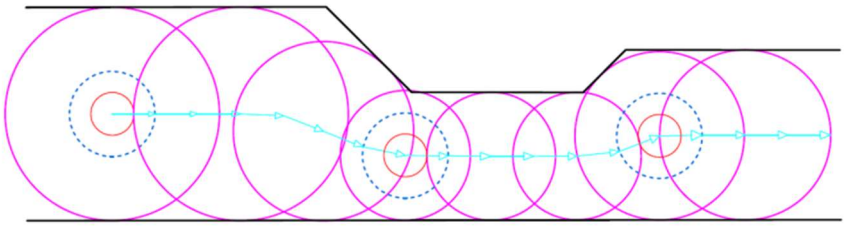

FIG. 6 EJEMPLO SENCILLO DE LA TÉCNICA “SLIDING BALLOON” [3]

Existen dos formas de utilizar esta técnica, la primera “Sliding Balloon”, consiste en, como se ha comentado anteriormente, buscar siempre el espacio más amplio por el que pasar, buscando el globo más grande posible (Fig. 7).

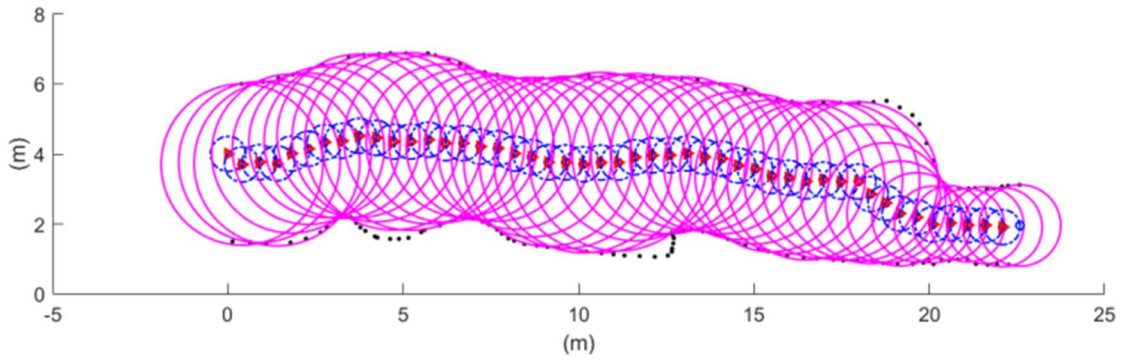

FIG. 7 EJEMPLO DE "SLIDING BALLOON" [3]

La segunda variante, que será en la que más se centrará este trabajo es el "Wall Sliding Balloon", que es en esencia la misma técnica, pero fijando en todo momento una distancia con la pared. En este caso puede verse como el robot (triángulo rojo) a pesar de comenzar en medio del túnel, se desplaza primero hasta que la distancia con la pared es la indicada antes de comenzar su trayectoria (Fig. 8).

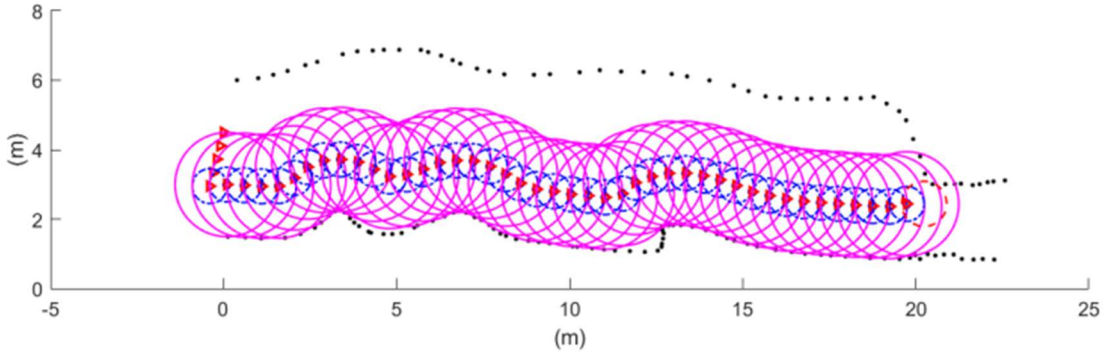

FIG. 8 EJEMPLO DE "WALL SLIDING BALLOON" [3]

3.2. Problemas que mejorar y adaptación a un barco.

El algoritmo original funcionará correctamente siempre que no se dé algún caso particular. Por ejemplo, si el robot llegase a una situación de estrechamiento o recoveco en el entorno (Fig. 8), cuyo espacio es menor que el radio de seguridad del robot, el robot se detendría por completo. Una mejora fundamental en estos casos sería hacer que el robot fuera capaz de dar media vuelta antes de llegar a un punto muerto y proseguir por otro camino hacia su meta.

Con esta mejora puede que no fuera necesario hacer nada más en el caso de un robot terrestre. Sin embargo, lo que se pretende con este trabajo es aplicar esta técnica a un barco autónomo.

A diferencia de un robot con ruedas, un barco tiene inercia al moverse en el agua. En un robot terrestre, bastará con parar los motores para detener su movimiento por completo. Sin embargo, en un barco, será necesario aplicar una velocidad totalmente contraria para pararlo en el menor tiempo posible.

En algunas pruebas en las que se ha aplicado la técnica original sobre el barco, se ha visto que, aunque el barco sea capaz de detectar los obstáculos y recalcular la trayectoria, la inercia provoca colisiones en muchos casos. Para evitar esto, se intentaba frenar el barco aplicando una velocidad máxima negativa lo que hacía sufrir mucho a los motores y provocaba trayectorias bruscas.

3.3. Soluciones planteadas.

Al intentar adaptar el algoritmo original a un barco, está claro que el principal problema es la anticipación. La anticipación que otorga el algoritmo original a un robot terrestre no es suficiente para el caso de un barco, ya que, como se ha comentado anteriormente, la inercia obliga a tener un mayor nivel de anticipación.

Para solventar esto se plantearon diferentes soluciones, siendo una de ellas cambiar la figura del globo, una circunferencia, por una elipse. Una elipse permitiría al barco detectar obstáculos o paredes a mayor distancia de lo que los puede detectar una circunferencia. Sin embargo, al tener la elipse más parámetros (focos, longitud de ejes…) y ser por tanto más complicado de implementar, se optó por construir un segundo globo.

Tras probar distintas alternativas para construir un segundo globo, se llegó a la conclusión de que la mejor opción sería construir este segundo globo sobre la circunferencia que forma el primero.

Así, la técnica de navegación “Sliding Balloon” mejorada (o “Sliding Balloon V2”) se basa en el cálculo de no uno, si no dos globos consecutivos. Al utilizar dos globos se consigue una mayor anticipación y percepción de las características del entorno y, por lo tanto, se logra corregir la trayectoria con la suficiente antelación, evitando así posibles choques no deseados.

Por otro lado, ya que lo que se pretende con este método es recopilar información sobre el entorno donde se va a mover el barco, se ha adaptado la forma del “Wall Sliding Balloon”. En lugar de mantener al barco a una distancia fija a la pared, se le dará un vector director que apunte siempre hacia la orilla. De esta manera, se resuelve el problema de que el barco se detuviese al entrar en un estrechamiento como se veía en la figura 8.

3.4. “Sliding Balloon V2”.

Tras evaluar cuáles serían las mejores soluciones para mejorar el algoritmo de “Sliding Balloon” y poder adaptarlo a un barco, se ha llegado al algoritmo explicado por pasos a continuación.

Antes de comenzar, se definen distintos parámetros que entrarán en juego a lo largo de los pasos del algoritmo que son: Radio de avance (*AdvanceStep*), sobre este radio se calculará el primer globo; radio máximo (*Rmax*) que pueden llegar a tener los globos y radio mínimo o de seguridad (*Rmin*) que deben tener los globos calculados para seguir una trayectoria normal; vector de avance de misión (*AdvanceVectorMission*), será el vector director que se pretende seguir en cada momento.

Una vez definidos los parámetros principales, el algoritmo comienza a iterar con los siguientes pasos:

1. Para comenzar, el barco toma los puntos del entorno como una nube de puntos 3D. Esta nube es filtrada para que queden únicamente aquellos puntos que interesan (Figs. 9 y

10). Es decir, se trabajará con aquellos puntos que estén a una altura similar a la del barco y a una distancia relativamente cercana.

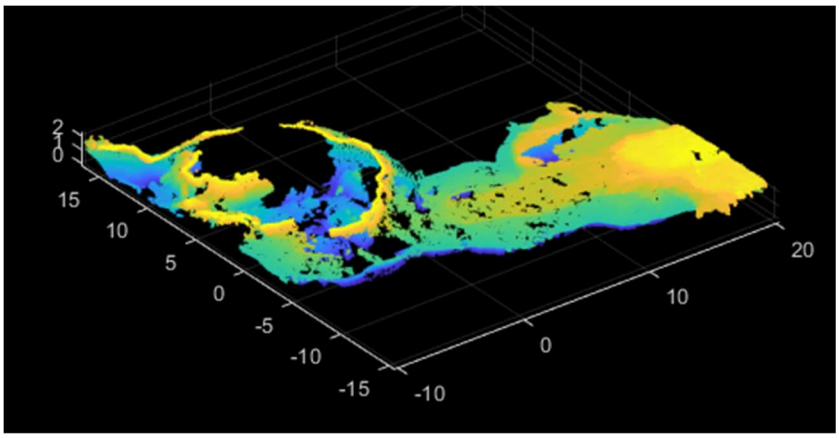

FIG. 9 NUBE DE PUNTOS SIN FILTRAR

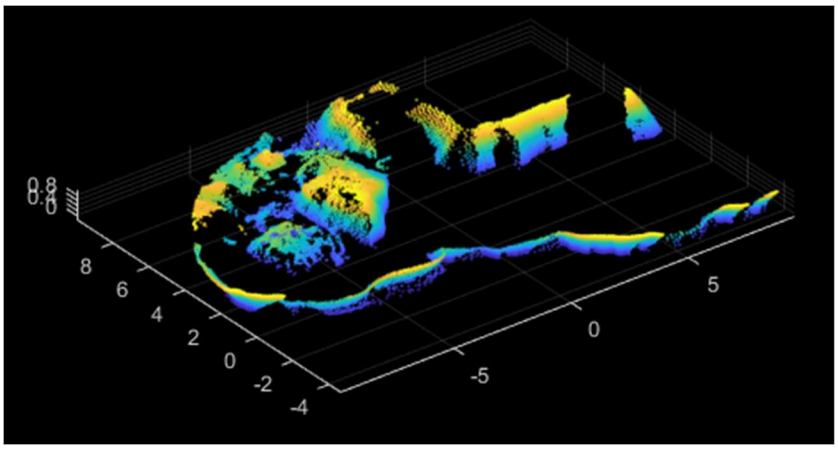

FIG. 10 NUBE DE PUNTOS FILTRADA

2. A continuación, se calculan los globos, calculando sus centros y sus radios.
   2.1. Siempre se tomará, la posición del barco como el origen de coordenadas. Se calcula un primer punto al que avanzar con radio *AdvanceStep* y dirección *AdvanceVector*. Se toma este primer punto como centro provisional del primer globo (*BalloonCenter1*). Únicamente en la primera iteración, el vector de avance *AdvanceVector* tendrá el mismo valor que *AdvanceVectorMission*.

$$BalloonCenter1 = AdvanceStep * \overrightarrow{AdvanceVector}$$

   2.2. Dado el punto *BalloonCenter1*, se busca, en la nube de puntos del entorno, el punto más cercano. La distancia a este punto será el radio del primer globo.

$$BalloonRadius1 = \min\left(\left\|\overrightarrow{BalloonCenter1\ Pi}\right\|\right) \forall\ Pi\ \in Puntos\ del\ entorno$$

$$NearestPoint = \left\{Pi \;\middle|\; \min\left(\left\|\overrightarrow{BalloonCenter1\ Pi}\right\|\right)\right\} \forall\ Pi\ \in Puntos\ del\ entorno$$

Fig. 11: En rojo, el barco en su posición inicial (P0); en azul, el radio de avance, en azul celeste, la dirección de avance; en azul oscuro, el supuesto centro del primer globo; en verde, el punto más cercano del entorno al punto anterior; en rosa, el supuesto primer globo.

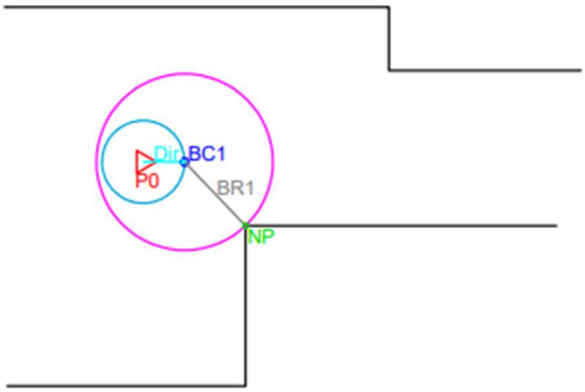

FIG. 11 PASOS 2.1 Y 2.2

2.3. Sumándole al radio obtenido un pequeño diferencial *d*, se comprueba si el globo ya es lo suficientemente grande o si la desviación es máxima, en cuyos casos, se terminaría de hinchar el globo. La condición de máxima desviación se da cuando el punto *BalloonCenter1* está los más lejos posible del punto *NearestPoint*. Por otro lado, *P0* es el punto en el que se encuentra el barco, que para este caso siempre será el origen de coordenadas.

$$BalloonRadius1 = BalloonRadius1 + d$$

$$l = \left\|\overrightarrow{NearestPoint\ P0}\right\|$$

$$\begin{cases} BalloonRadius1 > Rmax \ \rightarrow 1º\ globo\ terminado, ir\ al\ paso\ 2.7 \\ l + AdvanceStep < BalloonRadius1 \ \rightarrow 1º\ globo\ terminado, ir\ a\ paso\ 2.7 \\ Cualquier\ otro\ caso \ \rightarrow Continuar\ con\ paso\ 2.4 \end{cases}$$

2.4. Una vez calculado el radio del primer globo (*BalloonRadius1*), se busca cual será el centro del globo. Para ello, habrá que resolver la intersección entre la circunferencia formada por el radio de avance del barco (con centro en el origen) y la circunferencia formada por el radio del primer globo calculado (con centro en *NearestPoint*). Esto nos dará dos puntos diferentes como soluciones, por lo que se deberá escoger aquel que esté más cerca del primer centro, calculado en el paso 2.1. Este punto escogido será el nuevo *BallonCenter1*. El sistema de ecuaciones a resolver será el siguiente:

$$\begin{cases} (x - NearestPoint[0])^2 + (y - NearestPoint[1])^2 = BalloonRadius1 \\ (x - P0[0])^2 + (y - P0[1])^2 = AdvanceStep \end{cases}$$

2.4.1. Obtenidos los puntos *P1* y *P2* del sistema anterior, se busca cuál es más cercano a *BallonCenter1*.

$$BalloonCenter1 = \{P \mid \min(\|\overrightarrow{BalloonCenter1\ P}\|)\}\ \forall\ P\ \in \{P1, P2\}$$

Fig. 12: En gris, los radios del primer globo y el mismo radio sumando un diferencial; en azul oscuro, los puntos de intersección de ambas circunferencias, uno de estos puntos será el nuevo centro del primer globo.

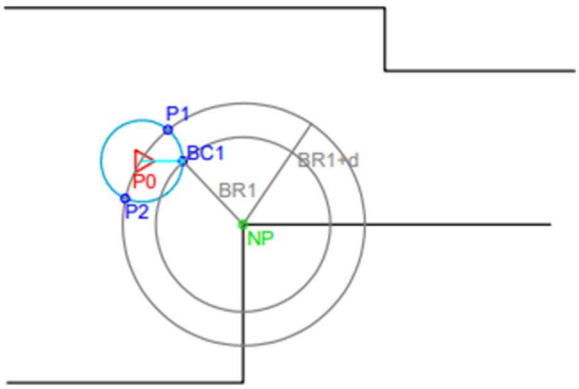

FIG. 12 PASOS 2.3 Y 2.4

2.5. Este nuevo centro no coincide con el *AdvanceVector* anterior, por lo que habrá que actualizarlo.

$$\overrightarrow{AdvanceVector} = \overrightarrow{P0\ BallonCenter1}$$

2.6. Con este nuevo centro, obtenido en el punto 2.4.1, se buscan, dentro del radio *BalloonRadius1*, puntos que puedan estar dentro del globo. Esto se hace para comprobar si se puede continuar hinchando el globo.

$$P_{IN} = \{P_j \mid \|\overrightarrow{BalloonCenter1\ P_j}\| < BalloonRadius1\}\ \forall\ P_j \in Puntos\ del\ entorno$$

2.6.1. Para dejar de hinchar el globo, estos puntos del entorno P_{IN} y el punto *NearestPoint*, deben encontrarse en lados separados por el nuevo vector de avance del barco, obtenido en el punto 2.5.

$$\overrightarrow{v_1} = \overrightarrow{P0\ NearestPoint}, \qquad \overrightarrow{v_2} = \overrightarrow{P0\ P_{IN}}$$

$$\overrightarrow{Xv_1} = \overrightarrow{AdvanceVector} * \overrightarrow{v_1}, \qquad \overrightarrow{Xv_2} = \overrightarrow{AdvanceVector} * \overrightarrow{v_2}$$

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

Si se cumple que los puntos están en distintos lados del vector de avance, se puede continuar con el siguiente globo (paso 2.7). Si no se cumple, habrá que volver al paso 2.2 para seguir hinchando el globo. Cada vez que se vuelve al paso 2.2, tanto los valores de *BalloonRadius1* y *BalloonCenter1* como el vector de avance, se han actualizado. Lo que produce un nuevo *NearestPoint* y permite calcular un globo cada vez más grande

(hasta alcanzar *Rmax* o que se cumpla la condición anterior), de esta manera se consigue que la trayectoria a seguir sea por el camino más seguro posible.

Fig. 13: En azul celeste, el nuevo vector director (*AdvanceVector*); en azul oscuro, el nuevo centro para el primer globo; en rosa, el primer globo con su nuevo radio terminado, ya que existen puntos del entorno a ambos lados del vector director.

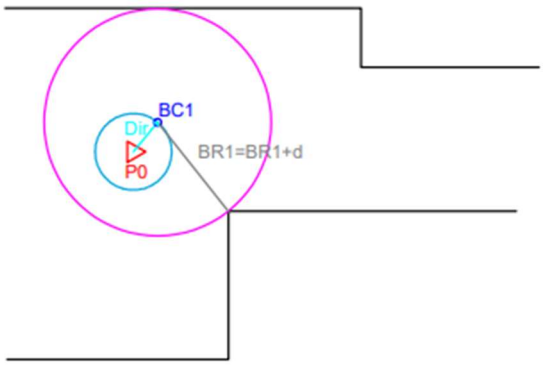

FIG. 13 PASOS 2.5 Y 2.6

Una vez se ha obtenido el primer globo, se procede al cálculo del segundo, cuyos valores característicos serán *BalloonCenter2* y *BalloonRadius2*. Este segundo globo permitirá una mayor anticipación, lo que conllevará en una trayectoria final más estable y segura para el barco.

2.7. Primero se calcula, al igual que con el primer globo, el centro para el segundo globo. Este vendrá dado por el vector *AdvanceVector* actualizado antes. En este caso el segundo globo se formará sobre el primero, al igual que el primero se ha formado sobre el radio de avance del barco.

$$BalloonCenter2 = (AdvanceStep + BalloonRadius1) * \overrightarrow{AdvanceVector}$$

2.8. Para comprobar que la trayectoria calculada por el primer globo es segura, se comprueba si en una circunferencia del mismo radio que *BalloonRadius1* y centro en *BalloonCenter2* hay puntos del entorno que obstaculizarían el paso.

$$P_{IN} = \{P_j \mid \|\overrightarrow{BalloonCenter2\ P_j}\| < BalloonRadius1\}\ \forall\ P_j \in Puntos\ del\ entorno$$

2.8.1. En el caso de que no existan puntos dentro de esta circunferencia, será ésta la que conforme el segundo globo que tendrá como centro *BalloonCenter2* y como radio *BalloonRadius2*, con el mismo valor que *BalloonRadius1*. Si, en cambio, existen puntos del entorno dentro de esta circunferencia, habrá que continuar con el cálculo del segundo globo.

$$\begin{cases} \exists\ Puntos \in P_{IN} \rightarrow Continuar\ en\ paso\ 2.9 \\ \nexists\ Puntos \in P_{IN} \rightarrow Globo\ 2\ terminado \end{cases}$$

Fig. 14: Comprobación de dirección segura, en rosa el primer y segundo globo. Como dentro de la zona que conforma el segundo globo hay puntos del entorno, habrá que recalcular este segundo globo para corregir la dirección de avance.

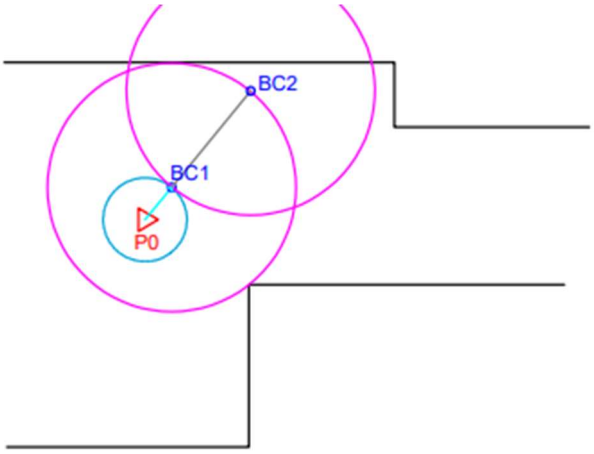

FIG. 14 PASOS 2.7 Y 2.8

2.9. Para construir el segundo globo, se busca de nuevo el punto del entorno más cercano al supuesto *BalloonCenter2*.

$$BalloonRadius2 = \min\left(\left\|\overrightarrow{BalloonCenter2\ Pi}\right\|\right) \forall\ Pi\ \in Puntos\ del\ entorno$$

$$NearestPoint = \left\{Pi \,\middle|\, \min\left(\left\|\overrightarrow{BalloonCenter2\ Pi}\right\|\right)\right\} \forall\ Pi\ \in Puntos\ del\ entorno$$

Fig. 15: De nuevo en verde, el punto más cercano al supuesto centro de, esta vez, el segundo globo (BC2, en azul oscuro).

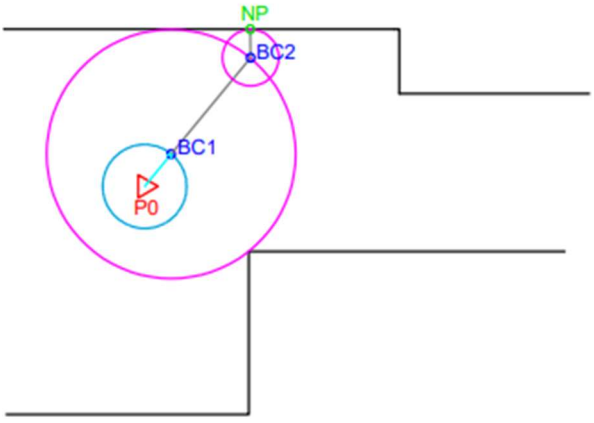

FIG. 15 PASO 2.9

2.10. Sumándole al radio obtenido un pequeño diferencial *d*, se comprueba si el globo ya es lo suficientemente grande o si la desviación es máxima, en cuyo caso, se

terminaría de hinchar el globo. Para comprobar si la desviación es máxima se toma como referencia el supuesto centro del segundo globo.

$$BalloonRadius2 = BalloonRadius2 + d$$

$$l = \left\|\overrightarrow{NearestPoint\ BalloonCenter2}\right\|$$

$$\begin{cases} BalloonRadius2 > Rmax \ \rightarrow 2º\ globo\ terminado \\ l + BalloonRadius1 < BalloonRadius2 \ \rightarrow 2º\ globo\ terminado \\ Cualquier\ otro\ caso \ \rightarrow Continuar\ con\ paso\ 2.11 \end{cases}$$

2.11. Una vez calculado el radio del segundo globo, se busca cuál será su centro. Igual que para el primer globo, se resuelve la intersección entre la circunferencia del primer globo y la circunferencia formada por el radio del segundo globo calculado (con centro en *NearestPoint*). De nuevo habrá dos puntos como posibles soluciones y se escogerá aquel que esté a menor distancia de *BalloonCenter2*. Este punto escogido será el nuevo *BallonCenter2*. El sistema de ecuaciones a resolver será el siguiente:

$$\begin{cases} (x - NearestPoint[0])^2 + (y - NearestPoint[1])^2 = BalloonRadius2 \\ (x - BalloonCenter1[0])^2 + (y - BalloonCenter1[1])^2 = BalloonRadius1 \end{cases}$$

2.11.1. Obtenidos los puntos *P1* y *P2* del sistema anterior, se busca cuál es más cercano a *BallonCenter2*.

$$BalloonCenter2 = \left\{P \,\middle|\, \min\left(\left\|\overrightarrow{BalloonCenter2\ P}\right\|\right)\right\} \forall\ P\ \in \{P1, P2\}$$

Fig. 16: En gris, el radio del supuesto segundo globo y el mismo más un diferencial; en azul oscuro, los puntos P1 y P2, resultado de la intersección de ambas circunferencias. Uno de ellos, el más cercano a BC2 será el nuevo centro para el segundo globo.

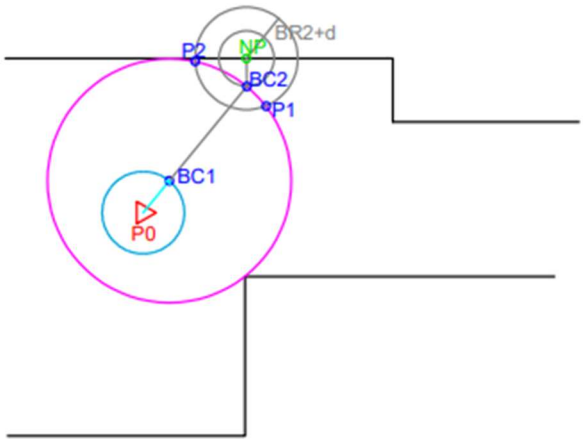

FIG. 16 PASOS 2.10 Y 2.11

2.12. Para continuar con el siguiente paso en el que se comprobará si el globo calculado es el mejor, se necesita el vector que une los centros de ambos globos.

$$Vector1\_2 = \overrightarrow{BallonCenter1\ BalloonCenter2}$$

2.13. Con el centro del segundo globo obtenido en el punto 2.11.1, se buscan, dentro del radio *BalloonRadius2*, puntos que puedan estar dentro del globo. De esta manera se comprueba si se puede seguir hinchando el globo.

$$P_{IN} = \{P_j \mid \|\overrightarrow{BalloonCenter2\ P_j}\| < BalloonRadius2\}\ \forall\ P_j \in Puntos\ del\ entorno$$

2.13.1. Para dejar de hinchar el globo, estos puntos del entorno P_{IN} y el punto *NearestPoint*, deben encontrarse en lados separados por el vector calculado en el paso 2.12.

$$\overrightarrow{v_1} = \overrightarrow{BalloonCenter2\ NearestPoint}, \qquad \overrightarrow{v_2} = \overrightarrow{BalloonCenter2\ P_{IN}}$$

$$\overrightarrow{Xv_1} = \overrightarrow{Vector1\_2} * \overrightarrow{v_1}, \qquad \overrightarrow{Xv_2} = \overrightarrow{Vector1\_2} * \overrightarrow{v_2}$$

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

Fig. 17: En rosa, el segundo globo, con centro en BC2 (en azul oscuro). Existen puntos del entorno tanto a izquierda como a derecha del vector que une el centro del primer globo con el centro del segundo, por lo que el globo puede dejar de hincharse.

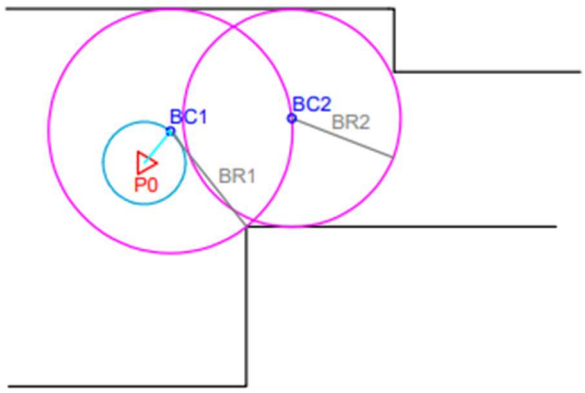

FIG. 17 PASOS 2.12 Y 2.13

Si se cumple que los puntos están en distintos lados del vector, se habrá obtenido el segundo globo con centro y radio *BalloonCenter2* y *BalloonRadius2* respectivamente. Si no se cumple habrá que volver al paso 2.9. para seguir hinchando el segundo globo.

Una vez obtenidos los centros y radios de los dos globos, se puede continuar con el algoritmo.

3. Tras la creación de ambos globos, el rumbo a seguir (un ángulo sobre el eje x, ya que la posición del barco está definida como el eje de coordenadas en todo momento) será

aquel marcado por la bisectriz del ángulo formado por las líneas que unen el barco con cada centro de los globos.

$$\theta = atan2(BalloonCenter1[2], BalloonCenter1[1])$$

$$\varphi = atan2(BalloonCenter2[2], BalloonCenter2[1])$$

$$Rumbo = \frac{\varphi - \theta}{2} + \theta$$

Fig. 18: Cálculo del rumbo (en verde), dados los centros de los globos (BC1 y BC2, en azul oscuro)

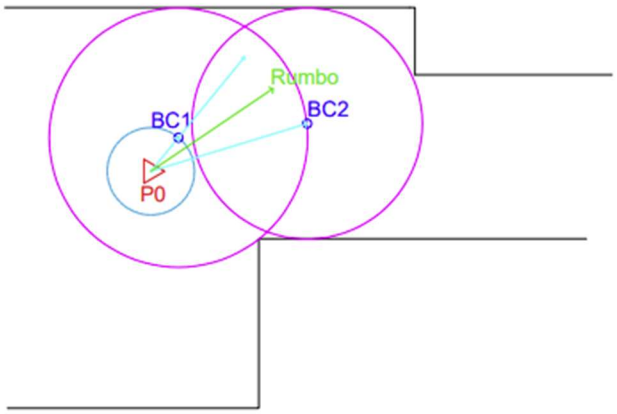

FIG. 18 CÁLCULO DEL RUMBO

4. Se comprueba entonces si el primer globo es mayor que un mínimo. Si este mínimo no se cumple, quiere decir que el barco está demasiado cerca de una pared u obstáculo. Por lo tanto, en la siguiente iteración el AdvanceVector cambiará, para que los siguientes globos sean lanzados en otra dirección, lejos de la pared.

5. A continuación, entra en juego una función de anticipación que indica si el barco sería capaz de continuar con el rumbo calculado en el apartado 3.
   5.1. Dentro del segundo globo, se buscan si hay o no puntos del entorno. Si no hay, el barco puede continuar sin problemas en el paso 6. En caso contrario, se sigue en el paso 5.2.

$$P_{IN} = \{P_j \mid \|\overrightarrow{BalloonCenter2\ P_j}\| < BalloonRadius2\} \ \forall\ P_j \in Puntos\ del\ entorno$$

$$\begin{cases} \exists\ Puntos \in P_{IN} \ \rightarrow Continuar\ en\ paso\ 5.2 \\ \nexists\ Puntos \in P_{IN} \ \rightarrow Continuar\ en\ paso\ 6 \end{cases}$$

   5.2. Si existen puntos dentro del segundo globo, se determina cuales quedan a izquierda y derecha del vector que determina el rumbo. Para cada punto del conjunto P_{IN} se calcula un vector que va desde el origen hasta cada punto. El signo tercer miembro

del producto vectorial entre este vector y el vector que determina el rumbo indicará si el punto está a la derecha, la izquierda o sobre el mismo vector de Rumbo.

$$\overrightarrow{\imath v} = \overrightarrow{P0\ P_{IN}}$$

$$\overrightarrow{Rumbo\_\imath v} = \overrightarrow{Rumbo} * \overrightarrow{\imath v}$$

$$\begin{cases} \overrightarrow{Rumbo\_\imath v}[3] > 0 \ \rightarrow El\ punto\ está\ a\ la\ izquierda\ del\ vector,\ P_{IN} \in izqda \\ \overrightarrow{Rumbo\_\imath v}[3] < 0 \ \rightarrow El\ punto\ está\ a\ la\ derecha\ del\ vector,\ P_{IN} \in dcha \\ \overrightarrow{Rumbo\_\imath v}[3] = 0 \ \rightarrow El\ punto\ está\ sobre\ el\ vector \end{cases}$$

5.3. Una vez que se han clasificado todos los puntos encontrados a izquierda y a derecha, se busca la distancia mínima entre pares de puntos con un bucle iterativo.

$$minDist = \min\left(\left\|\overrightarrow{dcha_\imath\ izqda_\jmath}\right\|\right)$$

5.4. Si la mínima distancia es mayor que el radio de seguridad del barco, éste puede continuar con su avance con *AdvanceVectorMission*. En caso contrario, el *AdvanceVector* para la próxima iteración cambiará, lanzando los globos en otra dirección.

$$\begin{cases} minDist > Rmin \ \rightarrow AdvanceVector = AdvanceVectorMission \\ minDist < Rmin \ \rightarrow AdvanceVector\ de\ desviación \end{cases}$$

En el ejemplo visto en las figuras anteriores (Fig. 18), los puntos del entorno que entran dentro del segundo globo están separados una distancia suficiente como para que el barco pase sin peligro entre ellos, por lo que, en la siguiente iteración, el vector de avance tomará el valor de AdvanceVectorMission.

6. Una vez calculado el rumbo y el *AdvanceVector* para la siguiente iteración, se calcula la velocidad de referencia para el barco. Esta velocidad variará según el tamaño de los globos, y el ángulo del rumbo, siendo máxima en los casos de una trayectoria recta y globos de tamaño máximo y mínima en caso de rumbos muy desviados del original o globos demasiado pequeños.

7. Un controlador calculará entonces la diferencia de velocidades que ha de haber entre las hélices para alcanzar la orientación del Rumbo deseado.

8. Con esta diferencia y la velocidad calculada en el paso 6 se determinará la acción final a aplicar sobre los motores para alcanzar tanto el rumbo como la velocidad consignados.

4. Algoritmos en MATLAB

Para comprobar que el nuevo algoritmo funciona correctamente se utiliza el programa MATLAB, una herramienta de programación en la que se pueden definir todos los pasos anteriores y observar una simulación sencilla.

Para llevar a cabo algunos de los pasos del algoritmo se hace uso de las mismas funciones con las que contaba el algoritmo original, además de algunas nuevas. Estas funciones son las siguientes:

- *SolveCircleIntersection*: Devuelve los puntos de intersección entre dos circunferencias, dados sus centros y sus radios. Se utiliza para calcular los nuevos centros de los globos.
- *NearestSol*: Dados dos puntos (los devueltos por la función SolveCircleIntersection) devuelve el más cercano a otro punto. Se utiliza para actualizar los valores de los centros de los globos.
- *SameSide*: Dado un vector de separación y otros dos vectores, de vuelve verdadero o falso si estos dos vectores quedan a derecha e izquierda del primer vector. Esta función se utiliza para saber si existen puntos a izquierdas y a derechas del vector de avance y saber, en definitiva, si ya se puede dejar de hinchar el globo.
- *Bisectriz*: Devuelve la bisectriz de un ángulo dados dos puntos. Esta función se utiliza en el cálculo del rumbo.
- *Paso*: Dados un radio un centro y un vector, devuelve verdadero o falso en función de si existe paso seguro o no. Esta función comprende todo el paso 5 del algoritmo. (Ver sección 3.4.)

Todos los pasos que conllevan él cálculo de los globos son recogidos en la función *SB2D_stepV2*, que dados una nube de puntos y un vector de avance (*AdvanceVector*), devuelve la posición de los centros y el valor de los radios de cada globo.

Por otro lado, para poder simular y comprobar el correcto funcionamiento del nuevo algoritmo, se hace uso de la función *pointCloud*, nube de puntos. Los puntos del mapa son archivados en esta "nube" y pueden ser tratados mediante algunas funciones incluidas en la *Computer Vision Toolbox* que proporciona el propio MATLAB. Entre otras, las funciones a utilizar más relevantes son *FindNeighborsInRadius*, que busca puntos dentro de una circunferencia definida, y *FindNearestNeighbours*, que busca el punto o puntos más cercanos más cercanos a otro determinado. [4]

Después, con la función *RRumbo*, se calcula la diferencia de velocidades que ha de haber entre las hélices para que el barco tome el rumbo que se ha calculado con los globos. Es en esta función donde aparece el control.

Por último, para observar una simulación de la trayectoria del barco en un mapa, se han definido algunas funciones de visualización. Estas funciones son *boat*, que define las orientaciones y posiciones que va adoptando el barco a lo largo del algoritmo y, por último, *PlotRobot3*, que dibuja sobre la simulación la figura del barco.

El algoritmo al completo queda recogido entonces en el módulo *TresPozosBoat.m*. En este módulo se definen todos los parámetros que se utilizan en el algoritmo, se hace el filtrado de la nube de puntos y se realiza el cálculo de velocidades. Por último, al ejecutar este módulo se

puede observar la simulación de la trayectoria que toma el barco sobre un mapa previamente cargado.

4.1. Comparativa de funcionamiento entre el algoritmo original y el mejorado.

Tras implementar todo el algoritmo en MATLAB se ha hecho una comparativa, a nivel visual, de las trayectorias seguidas por el barco tanto con el algoritmo original como con el algoritmo mejorado.

En un mapa creado a mano, los resultados son los siguientes. Con el algoritmo original se observa que el barco es incapaz de salir de la curva cerrada en la que ha entrado (Fig. 19). Esto se debe principalmente a que un único globo no proporciona la previsión suficiente como para rectificar antes de llegar a quedarse atorado. Por otro lado, con el algoritmo mejorado, se ve como el barco es capaz de salir (Fig. 20). En este caso, al disponer de dos globos, la previsión es mucho mayor. El recoveco se ve con anterioridad y la función *Paso* determina que el barco no va a ser capaz de pasar por allí por lo que, en la siguiente iteración, lanza los globos en otra dirección, consiguiendo así saltarse el recoveco y continuar su trayectoria. Además, puede observarse como en el caso del algoritmo mejorado, la trayectoria es mucho más suave gracias a la anticipación que proporciona disponer de dos globos.

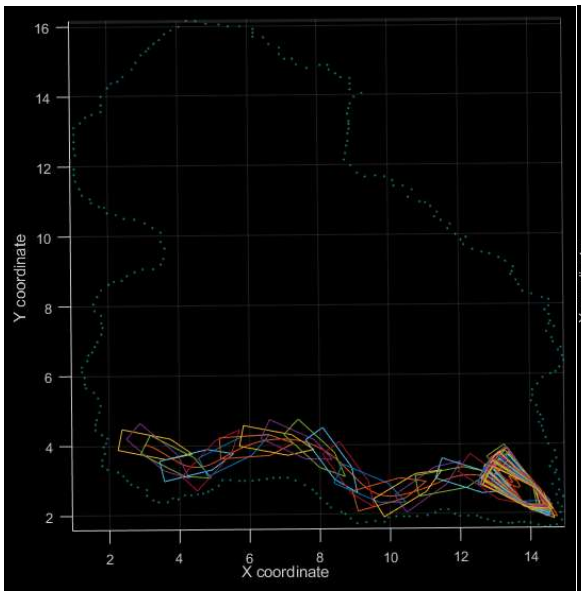

FIG. 19 PRUEBA DE MAPA, ALGORITMO ORIGINAL

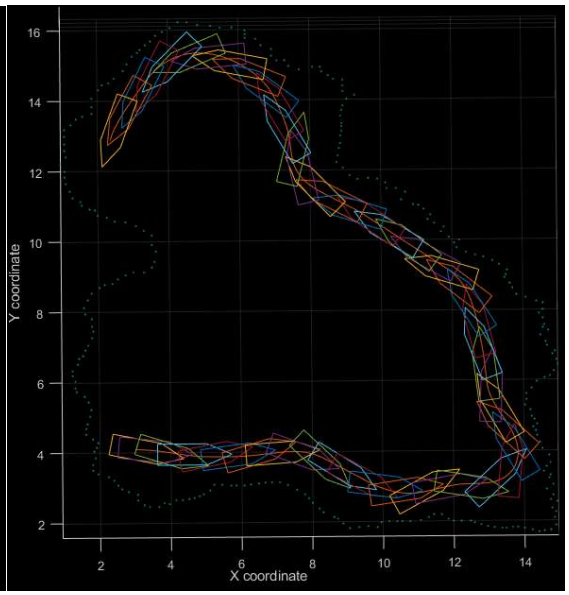

FIG. 20 PRUEBA DE MAPA, NUEVO ALGORITMO

De manera similar, en un mapa que representativo de una caverna real, se ve como con el algoritmo mejorado (Fig. 22) se consiguen trayectorias más suaves que con el algoritmo original (Fig. 21)

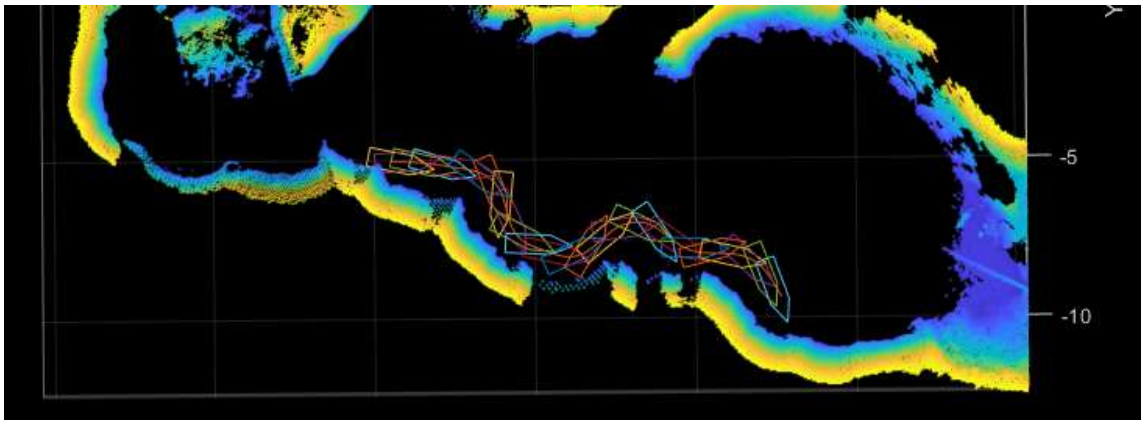

FIG. 21 PRUEBA DE MAPA REAL, ALGORITMO ORIGINAL

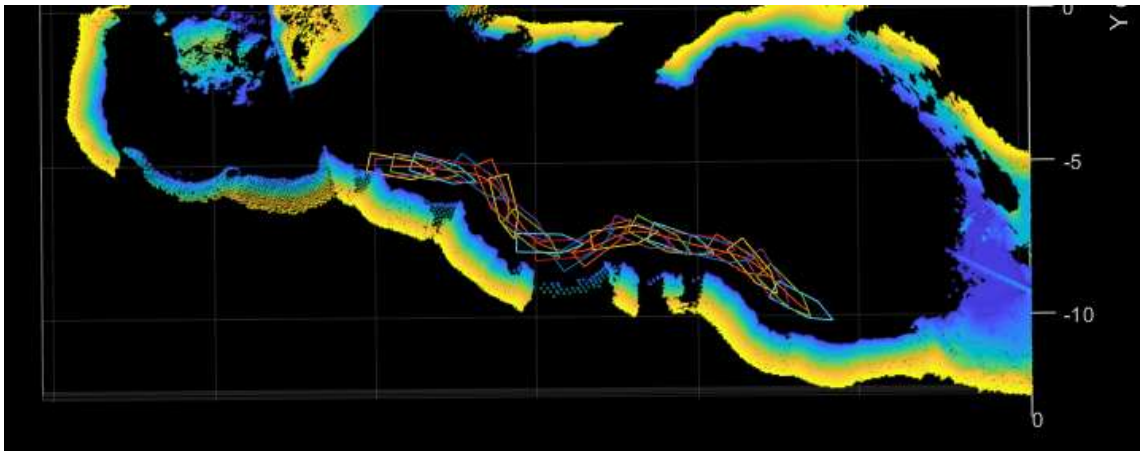

FIG. 22 PRUEBA DE MAPA REAL, NUEVO ALGORITMO

5. Algoritmos en C y ROS

Una vez visto en MATLAB que el algoritmo funciona correctamente en simulaciones, es momento de aplicar el nuevo método "Sliding Balloon V2" en el barco real. Para ello es necesario escribir todo el algoritmo en lenguaje C y hacer uso de la herramienta ROS, dedicada a la programación de robots.

El algoritmo se ha definido en diferentes módulos con la finalidad de simplificar el código. Por un lado, está el módulo de funciones geométricas, *geometry.cpp/.h*, en el que se han definido todas las funciones auxiliares que utiliza el método "Sliding Balloon V2":

- *Distance*: Calcula la distancia entre dos puntos dados.
- *findNearestNeighbor*: Dada una nube de puntos calcula el o los puntos más cercanos a otro punto.
- *findNeighborsInRadius*: Determina qué puntos de una nube de puntos están dentro de una circunferencia dada.
- *SolveCircleIntersection*: Calcula los puntos de intersección entre dos circunferencias dadas.
- *SameSide*: Determina si dos vectores están en el mismo lado de una línea divisoria dada por otro vector.
- *NearestSol*: Determina entre dos puntos cuál es el más cercano a un tercero.
- *Norm*: normaliza un vector.
- *Bisectriz*: Calcula la bisectriz dados dos puntos, de la misma manera que se explicó anteriormente.

Todas estas funciones son similares a las implementadas en MATLAB, sin embargo, funciones como *Distance*, *findNearestNeighbor*, *findNeighborsInRadius* y *norm*, que ya venían predefinidas en MATLAB han tenido que ser reescritas en C para poderse aplicar.

Por otro lado, está la clase C++ *Lolargo_SB.cpp/.h*. En ella se ha definido una función que engloba toda la parte del algoritmo en la que se calculan los dos globos, haciendo uso de algunas de las funciones descritas en *geometry.cpp*. También se ha incluido en esta clase la función Paso, ya que no se trata exactamente de una función de geometría. Así en esta clase reside la función SB2D_step V2, que, de forma similar a la función implementada en MATLAB, devuelve los centros y radios de los globos calculados de acuerdo con una nube de puntos y un vector de avance dados.

Por último, está *RoboBoatV2.cpp*. Se trata de un programa C++ que implementa un nodo ROS de navegación con el método. Con este programa se pondrá en funcionamiento el sistema de ROS con el que se podrá hacer navegar al barco gracias a los cálculos del algoritmo implementado. Además, el sistema ROS permitirá en todo momento ver en tiempo real los datos del barco, como velocidades lineales y angulares, posiciones, orientaciones… Todo ello sobre la herramienta de visualización RVIZ, en la que se puede ver la trayectoria que va siguiendo el barco y cómo los globos se van adaptando según las características del entorno en las que se navega.

6. Pruebas en entornos reales

Después de ver el correcto funcionamiento del algoritmo sobre MATLAB y haberlo descrito en lenguaje de programación de robots, es el momento de ver cómo se desenvuelve el barco con la técnica de navegación mejorada en un entorno real.

6.1. Piscina

Con el fin de ver si realmente el algoritmo funciona fuera de las simulaciones, se prueba el programa sobre el barco en una piscina. Al tener la piscina una forma rectangular, con líneas y ángulos rectos, se trata de la peor situación en la que podría encontrarse el barco a la hora de navegar de manera autónoma. Por otro lado, la piscina es un entorno totalmente controlado en el que se puede rescatar el barco si algo llegara a fallar.

Otro objetivo de esta prueba es el ajuste de parámetros tanto de los globos como de velocidades máximas de las hélices. Para que el barco navegue sin peligro de choques es necesario definir tanto su radio de seguridad como el radio máximo y mínimo que pueden tener los globos. Además, también se revisarán los parámetros que gobiernan el comportamiento del regulador que se utiliza en el control de velocidad.

Partiendo de unos parámetros ya definidos en pruebas con el algoritmo original, se hace una primera prueba.

FIG. 23 PRUEBAS EN PISCINA SIN CAMBIAR PARÁMETROS

En esta primera prueba se observó a simple vista y con ayuda de la herramienta RVIZ que los globos se calculaban correctamente y el barco era capaz de navegar por el borde de la piscina. Sin embargo, se vio también que el barco, por la inercia, se acercaba demasiado a los bordes a pesar de conseguir un cambio de trayectoria (Fig. 23).

Para evitar esta situación de casi choque, se cambian los tamaños que pueden tener los globos buscando un equilibrio entre el máximo y el mínimo. Se aumenta el máximo con el fin de obtener mayor anticipación al entorno y se aumenta el mínimo para que dé tiempo a cambiar de dirección antes de llegar al límite.

De nuevo se vuelve a poner el barco en movimiento. En esta segunda prueba se observó que, en efecto, aumentando los radios de los globos se conseguía corregir la trayectoria con mayor antelación, evitando situaciones peligrosas de escaso espacio entre el barco y la orilla (Fig. 24).

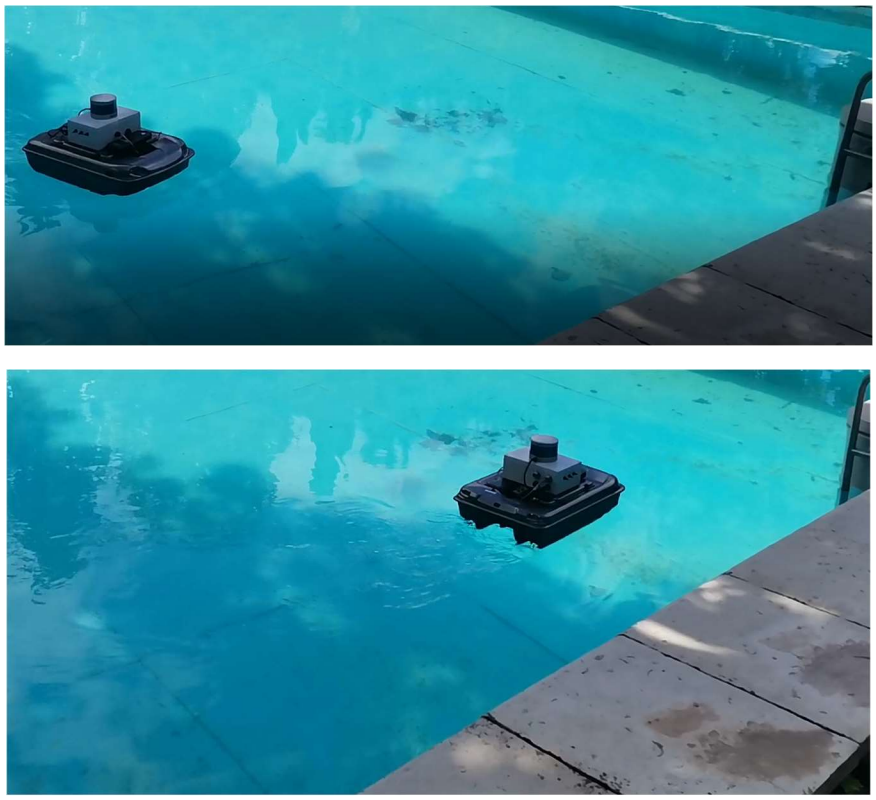

FIG. 24 PRUEBA EN PISCINA CAMBIANDO PARÁMETROS

Tras esta prueba de espacio se decidió aumentar la velocidad máxima de navegación con el fin de que el barco realizase el recorrido en menor tiempo. Tras aumentar este parámetro se observó que el barco seguía realizando trayectorias suaves y manteniendo suficiente espacio con las orillas.

En una última prueba se decidió cambiar los parámetros del regulador. Se probó con un regulador PI (proporcional integral), pero al no ver demasiada diferencia con respecto al regulador original, se optó por dejarlo como estaba, un regulador de tipo proporcional.

Por último, se muestra en las figuras 25 y 26 una comparativa de la reacción al entorno con el algoritmo original y con el nuevo algoritmo.

En el algoritmo original (Fig. 25), el barco seguía su trayectoria hasta encontrarse demasiado cerca de la pared. En ese momento, frenaba en seco y rotaba 90° sobre sí mismo para continuar por otro camino. El frenado en seco se consigue aplicando a las hélices una velocidad máxima negativa. Este cambio tan brusco de consigna puede ser válido, hasta cierto punto, en casos particulares de choque inmediato, sin embargo, no debería ocurrir cada vez que el barco se topa con una orilla. El hecho de que en este caso el barco no corrija su trayectoria hasta que está tan cerca de la orilla se debe principalmente a que solo se utiliza un globo. Este globo se lanza inmediatamente delante del barco y se va haciendo más pequeño conforme el barco se acerca

a la orilla. Es por esto que, cuando el radio de este globo ha alcanzado un tamaño mínimo, es demasiado tarde para rectificar la trayectoria con una curva, haciéndose necesaria la parada en seco.

Con el nuevo algoritmo (Fig. 26), los dos globos permiten ver la orilla con suficiente antelación. A esta distancia de detección, el algoritmo aplica nuevas consignas a las hélices que varían de forma gradual. Así, se evitan las paradas en seco y los cambios bruscos de consigna. Además, como el barco empieza a dar la curva mucho antes, el resultado es una trayectoria muchísimo más suave que los giros bruscos de 90° grados que se obtenían con el algoritmo original.

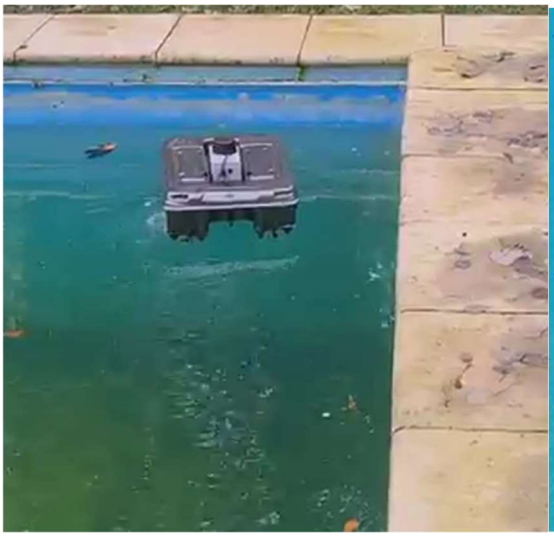

FIG. 25 PRUEBA PISCINA ALGORITMO ORIGINAL

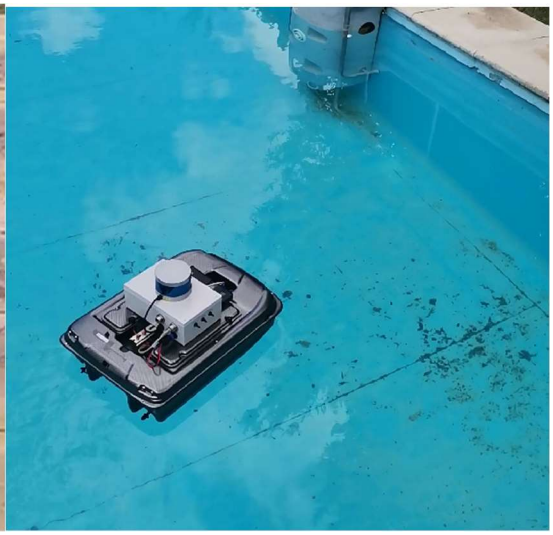

FIG. 26 PRUEBA PISCINA NUEVO ALGORITMO

6.2. Cueva Román, Clunia, Burgos

Como se comentaba en la introducción de este trabajo, uno de los objetivos principales de este trabajo era adaptar la técnica de navegación “Sliding Balloon” a un barco autónomo. Tras todo el trabajo realizado es el momento de poner a prueba el algoritmo para lo que ha sido diseñado.

La Cueva Román, situada al sur de la provincia de Burgos, es un conjunto de galerías inundadas y pozos subterráneos utilizados por la antigua ciudad romana de Clunia para el abastecimiento de agua (Fig. 27). Esta cueva, descubierta sobre el año 1900, es una cueva natural modificada en algunas zonas por los romanos, lo que la convierte en un yacimiento arqueológico de gran interés. En ella se han encontrado todo tipo de restos arqueológicos desde inscripciones en las paredes y figuras, hasta huesos humanos. [5]

Actualmente, está en marcha un proyecto para digitalizar todo lo que hay en estas galerías con el fin de estudiar todo lo encontrado sin necesidad de entrar a la cueva. Se trata de un entorno difícil, lleno de barro, techos bajos, estrecheces… En el que es complicado moverse y existe peligro de estropear restos arqueológicos de interés.

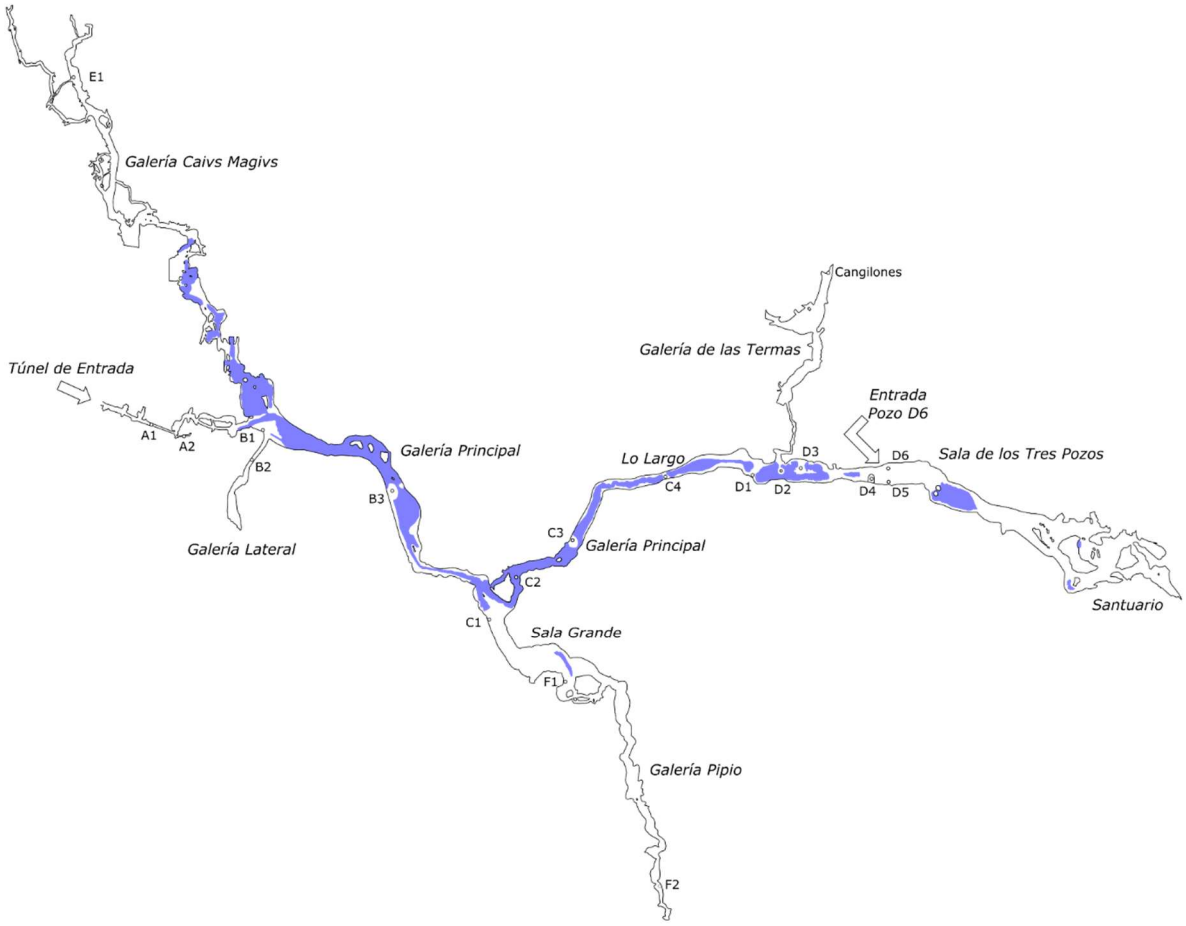

FIG. 27 PLANO DE LA CUEVA ROMÁN

Este trabajo se centra sobre todo en la parte de escaneado de la cueva, con el objetivo principal crear una reconstrucción digital de los túneles y galerías y para explorar huecos que aún no se hayan encontrado o en los que el acceso sea difícil.

Es aquí donde se pone en marcha el nuevo algoritmo "Sliding Balloon V2", sobre el barco RoboBoat, para el escaneo de la galería inundada "Lo Largo" de la figura 27.

Una vez puesto en marcha el barco, la única manera de visualizar su movimiento fue mediante la aplicación RVIZ. Durante todo el recorrido, se pudo observar la forma de la cueva en forma de una nube de puntos captada por el sensor láser LiDAR y cómo iban cambiando los globos, tanto en posición como en tamaño, respecto a la posición del barco y su entorno (Figs. 28 y 29). Además de esta vista gráfica, se podía ver en tiempo real los valores de velocidad que llevaba el barco y el valor de tamaño de los globos.

Tanto en la figura 28 como en la figura 29 se puede observar cómo los globos van adaptándose al entorno. El primero marca una primera dirección a seguir y el segundo aplica la corrección, haciendo que el barco pueda entrar en estrechamientos y evitando entrar en caminos sin salida o demasiado estrechos.

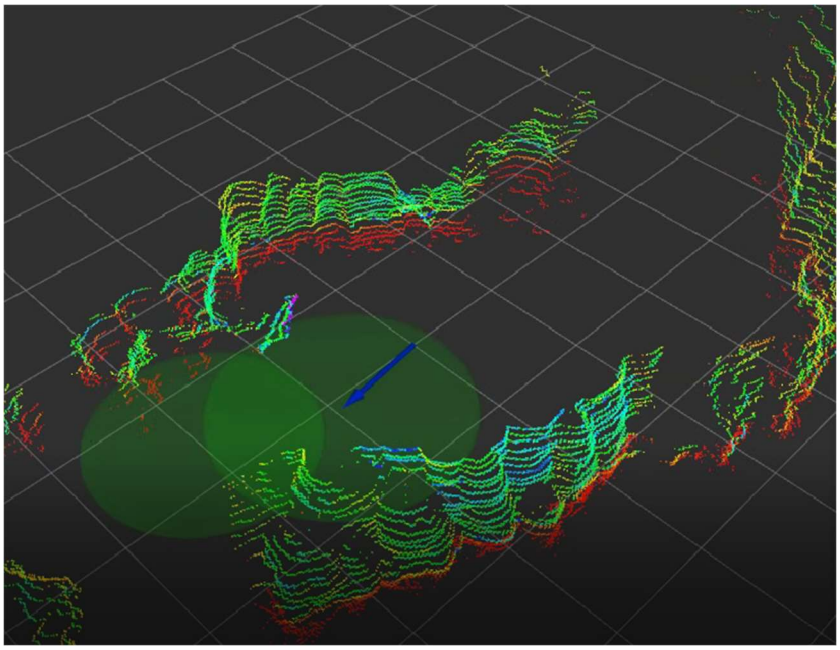

FIG. 28 VISTA "SLIDING BALLOON V2" EN RVIZ (1)

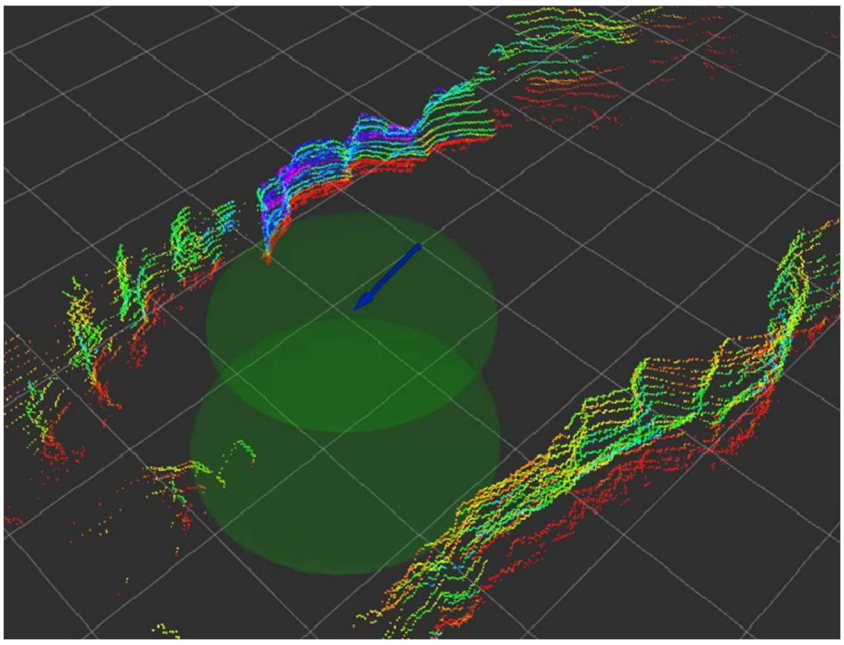

FIG. 29 VISTA "SLIDING BALLOON V2" EN RVIZ (2)

Para finalizar, con todos los datos recogidos por el barco durante su recorrido, se ha realizado sobre MATLAB un mapa de nube de puntos junto con la trayectoria seguida.

En la figura 30 se puede observar el recorrido hecho por el barco. El barco salió de la zona superior derecha y navegó hacia la zona inferior izquierda siguiendo la pared que quedaba a su derecha. Se puede ver cómo el barco circula por el lado derecho de la cueva y cómo logra evitar los huecos considerados demasiado pequeños o que no tenían paso posible al mismo tiempo que realiza una trayectoria de giros suaves durante todo el recorrido.

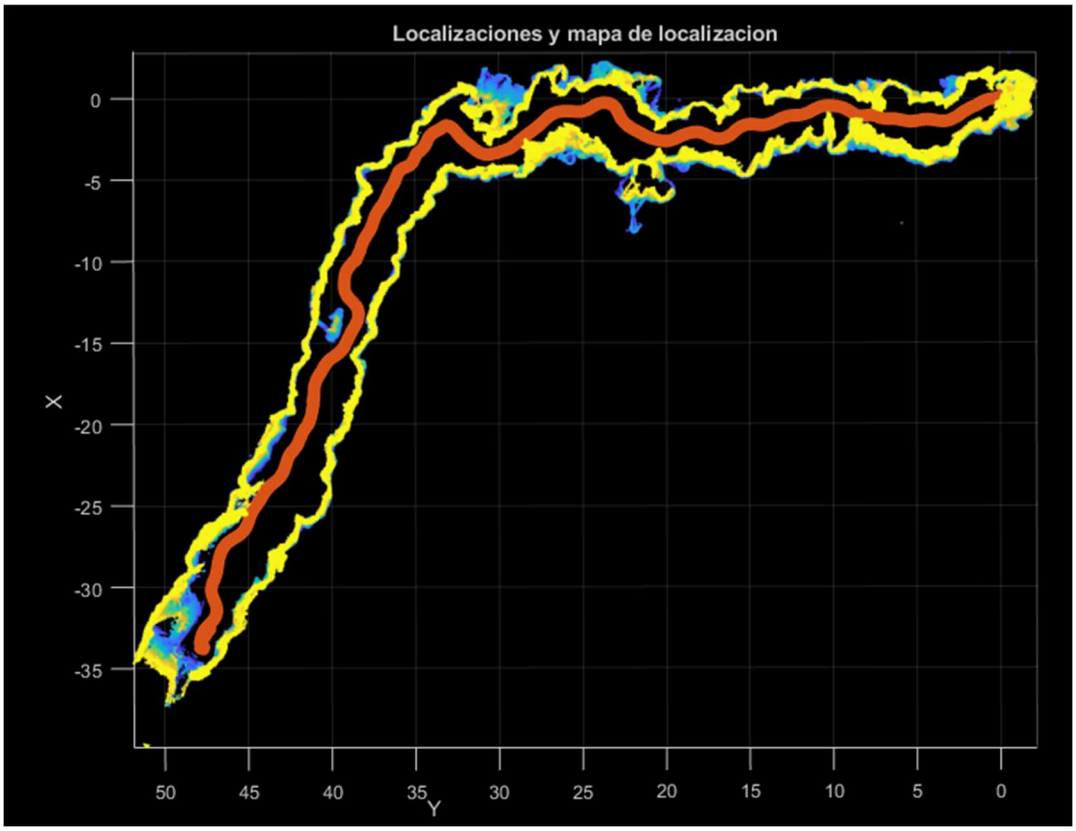

FIG. 30 TRAZA Y NUBE DE PUNTOS

Gracias a la técnica de navegación “Sliding Balloon V2” se ha logrado reconstruir mediante un modelo en 3D de nube de puntos las dimensiones de una de las zonas que componen la cueva Román.

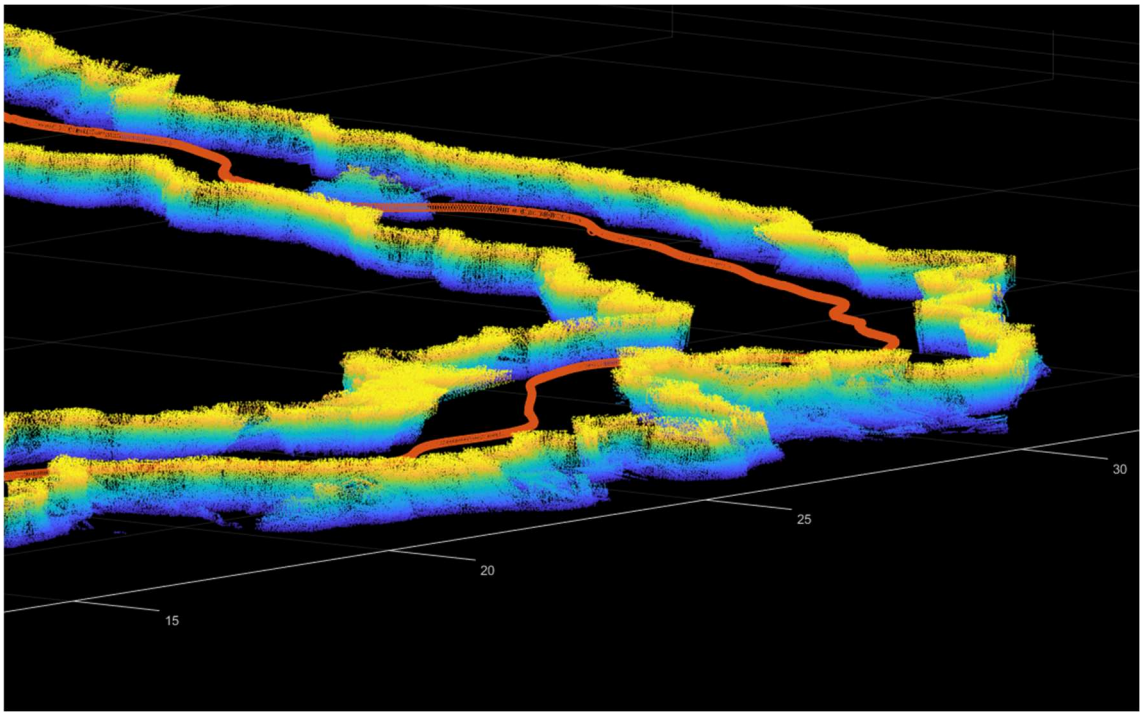

FIG. 31 VISTA EN PERSPECTIVA DE LA CUEVA

7. Conclusiones

Este trabajo se ha centrado en el diseño y mejora de la técnica de navegación “Sliding Balloon” con el propósito de aplicar esta técnica, creada y pensada para robots de tipo terrestre, a un barco autónomo.

A lo largo del proceso de adaptación del algoritmo “Sliding Balloon” al algoritmo “Sliding Balloon V2” ha habido dos principales desafíos. Por un lado, lograr aumentar el grado de percepción de los elementos del entorno, necesario para evitar los problemas causados por la inercia. Y, por otro lado, corregir algunas de las limitaciones con las que contaba la técnica de navegación original.

Tras estudiar en profundidad la técnica de navegación “Sliding Balloon” y desarrollar una nueva técnica basada en esta, ha quedado constancia tanto mediante simulación en MATLAB, como en entornos reales de que este nuevo método funciona correctamente. Se han conseguido salvaguardar los principales problemas que tenía la técnica original al aplicarse a un barco autónomo ya que esta nueva técnica cuenta con un mayor nivel de anticipación y hace que el barco lleve trayectorias suaves.

Por otro lado, es importante comentar la importancia de los resultados de la prueba sobre la cueva de Clunia. No solo se ha conseguido que el barco navegue de manera autónoma en una zona de características cuanto menos hostiles, si no que se ha conseguido una navegación segura y óptima a lo largo de todo el recorrido. Esto abre un frente en el que utilizar esta técnica con el fin de explorar entornos similares a los de la cueva Román y con el objetivo de poder reconstruir el espacio recorrido mediante un modelo digital en 3D. Además, el uso de esta técnica evita el que una persona tenga que recorrer la cueva poniendo en riesgo tanto su seguridad como la de los restos arqueológicos que en ella pudiera haber, como ha sido el caso de este proyecto.

Así pues, queda demostrado el funcionamiento de la técnica de navegación “Sliding Balloon V2”, diseñada para barcos autónomos y que conlleva un paso adelante en el ámbito de la exploración de entornos hostiles como son las cuevas subterráneas.

8. Bibliografía

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots, Second Edition*, vol. 23. 2011.

[2] J. Minguez and L. Montano, "Nearness diagram (ND) navigation: collision avoidance in troublesome scenarios," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 1, 2004, Accessed: Jul. 12, 2022. [Online]. Available: https://ieeexplore-ieee-org.cuarzo.unizar.es:9443/document/1266644/

[3] A. Adrián Montero Vitoria D. José Luis Villaroel Salcedo, "'Sliding Balloon' 2D: técnica de navegación para robots terrestres. 'Sliding Balloon' 2D: navigation technique for ground robots. Trabajo de Fin de Grado".

[4] "Point Cloud Processing - MATLAB & Simulink - MathWorks España." https://es.mathworks.com/help/vision/point-cloud-processing.html?s_tid=CRUX_lftnav

[5] "Clunia Subterránea." http://www.clunia.es/descubre-clunia/subterranea/

9. Anexos

9.1. Código de MATLAB

9.1.1. Función para el cálculo de los globos

```
function [Local_Goal_1,Balloon_Radius_1, Local_Goal_2, Balloon_Radius_2] =
SB2D_stepV2(Velodyne_Cloud,AdvanceVector)
%SB2D_STEPV2 Second Version for the Sliding Balloon technique
%Calculates the next two points of the trajectory using Sliding Balloon
%   Velodyne_Cloud: Point CLoud obtained from Velodyne in the local
%                   reference of the boat
%   AdvanceVector: Initial advance direction

global Rmin ;            % Radio mínimo de seguridad de la esfera
global Rmax ;            % Inflado maximo de la esfera
global AdvanceStep ;     % Distancia a la que se pone el objetivo local
global Precision ;       % Precision del metodo expresada en metros

% Planchado nube de puntos
ptCloud = pointCloud ([Velodyne_Cloud.Location(:,1)
Velodyne_Cloud.Location(:,2) 0.0*Velodyne_Cloud.Location(:,3)]) ;

% Inicializacion proceso
BalloonCenter1 = AdvanceVector*AdvanceStep ; %Este balloon center es
provisional (P0)
EndInflate = false ;
d = Precision / 2.0 ;

% Cálculo del primer globo como un Sliding Balloon original
while not(EndInflate)
    [indices,dists] = findNearestNeighbors(ptCloud,BalloonCenter1,1) ;

    % NearestPoint : nearest point
    NearestPoint = ptCloud.Location(indices,:) ;

    % r : distancia NearestPoint, BalloonCenter
    r = dists(1) ;

    l = norm (NearestPoint) ;
    Balloon_Radius_1 = r+d ;

    if (Balloon_Radius_1 > Rmax) break ; end ;


    % Condicion de maxima desviacion
    % Centro de globo está lo mas lejos posible de NP
    if (l+AdvanceStep < Balloon_Radius_1)  break ; end ;


    [p1 p2] = SolveCircleIntersection ([0.0 0.0 0.0], AdvanceStep,
NearestPoint, Balloon_Radius_1) ;

    BalloonCenter1 = NearestSol (BalloonCenter1, p1, p2) ;

    % Nuevo vector de avance según el primer globo calculado
    AdvanceVector = BalloonCenter1 ;
    AdvanceVector = AdvanceVector / norm(AdvanceVector) ;
```

```
    [indices,dists] =
findNeighborsInRadius(ptCloud,BalloonCenter1,Balloon_Radius_1) ;

    nv = NearestPoint - BalloonCenter1 ;
    EndInflate = false ;

    for i = 1:length(indices)
        iv = ptCloud.Location(indices(i),:) - BalloonCenter1 ;
        if not (SameSide (AdvanceVector, nv, iv))
            EndInflate = true ;
            break
        end
    end

end

Local_Goal_1 = BalloonCenter1 ;

% Cálculo del segundo globo sobre el primero

% Inicialización de variables para el segundo globo
% Lanzamos globo provisional sobre el primer globo con su mismo radio
BalloonCenter2 = AdvanceVector*(Balloon_Radius_1 + AdvanceStep) ;
EndInflate = false ;
d = Precision / 2.0 ;

% Buscar si hay puntos dentro del radio del segundo globo
[indices,dists] =
findNeighborsInRadius(ptCloud,BalloonCenter2,Balloon_Radius_1) ;
if isempty(indices)
    % No hay puntos, se puede seguir con el rumbo dado por el primer globo
    Local_Goal_2 = BalloonCenter2;
    Balloon_Radius_2 = Balloon_Radius_1;
    AdvanceVector2 = AdvanceVector;
else
    % Cálculo del segundo globo sobre el primero
    while not(EndInflate)
        [indices,dists] = findNearestNeighbors(ptCloud,BalloonCenter2,1);

        % NearestPoint : nearest point
        NearestPoint = ptCloud.Location(indices,:) ; % esta localizacion es
en coord del robot

        % r : distancia NearestPoint, BalloonCenter2
        r = dists(1) ;

        l = norm (NearestPoint - BalloonCenter2) ;
        Balloon_Radius_2 = r+d ;

        if (Balloon_Radius_2 > Rmax) break ; end ;

        % Condicion de maxima desviacion
        if (l+Balloon_Radius_1 < Balloon_Radius_2)  break ; end ;

        [p1 p2] = SolveCircleIntersection (BalloonCenter1, Balloon_Radius_1,
NearestPoint, Balloon_Radius_2);
        BalloonCenter2 = NearestSol (BalloonCenter2, p1, p2) ;
```

```
        % Vector que une los centros de los dos globos
        Vector12 = BalloonCenter2-BalloonCenter1;
        Vector12 = Vector12/norm(Vector12);

        % Vector que une el barco con el centro del segundo globo
        AdvanceVector2 = BalloonCenter2;
        AdvanceVector2 = AdvanceVector2 / norm(AdvanceVector2);

        [indices,dists] =
findNeighborsInRadius(ptCloud,BalloonCenter2,Balloon_Radius_2) ;

        nv = NearestPoint - BalloonCenter2 ;
        EndInflate = false ;

        for i = 1:length(indices)
            iv = ptCloud.Location(indices(i),:) - BalloonCenter2 ;
            if not (SameSide (Vector12, nv, iv))
                EndInflate = true ;
                break
            end
        end
    end
    Local_Goal_2 = BalloonCenter2;
end

end
```

9.1.2. Script para la simulación

```
clear all ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                                    ROBOBOAT                                   %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

parar = false ;
ErrObj = 0.2 ;

% Parametros del barco

d = 0.12 ;            % Distancia helice al eje del barco (m)
Ib = 0.10 ;           % Momento de innercia (Kg/m2)
Kx = 6.125 ;          % Cte resistencia al avance (Ns2/m2)
Ky = 15.3 ;           % Cte resistencia avance lateral (Ns2/m2)
Kz = 0.8 ;            % Cte resistencia giro (Nms2)
m = 5.0 ;             % Masa del barco (Kg)
T = 0.1 ;             % Periodo de muestreo (s)

Param = [d Ib Kx Ky Kz m T] ;

% Vector de estado inicial en la referencia ABS

Vx0 = 0.0 ;           % Velocidad lineal
Vy0 = 0.0 ;
W0 = 0.0 ;            % Velocidad angular
Px0 = 0.5 ;           % Posicion inicial (0.5, -5.0)
Py0 = -5.0 ;
Rumbo0 = -0.2 ;        % Rumbo

X0 = [Vx0 Vy0 W0 Px0 Py0 Rumbo0]' ;

Vref_max = 0.3 ;
Vref_min = 0.2 ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                          Laguna Tres Pozos                                    %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

load ('LagunaTresPozos.mat') ;
Galeria3D = pointCloud (Galeria3D.Location) ;
ABS_T_PARK = [ 0 -1  0  68.69 ;
    1  0  0  47.70 ;
    0  0  1  29.82 ;
    0  0  0   1.0 ] ;
Galeria3D = pctransform(Galeria3D,affine3d(ABS_T_PARK'));

indices = find (Galeria3D.Location(:,3) < 1.0) ;
Lim_Gal = select(Galeria3D, indices) ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                          Sliding Balloon                                      %
```

```
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Parametros del metodo

global Rmin ;            % Radio mínimo de seguridad de la esfera
Rmin = 0.5 ;
global Rmax ;            % Inflado maximo de la esfera
Rmax = 0.95 ;
global AdvanceStep ;     % Distancia a la que se pone el objetivo local
AdvanceStep = 0.75 ;
global Precision ;       % Precision del metodo expresada en metros
Precision = 0.1 ;

% Vector de avance

AdvanceVector_Mission = [1 -0.2 0] ;
AdvanceVector_Mission = AdvanceVector_Mission/norm(AdvanceVector_Mission) ;
AdvanceVector = AdvanceVector_Mission ;

ABS_T_BOAT = [ 1  0  0  Px0 ;
    0  1  0  Py0 ;
    0  0  1  0 ;
    0  0  0  1 ] ;

quat = tform2quat(ABS_T_BOAT) ;
plotTransforms(ABS_T_BOAT(1:3,4)',quat) ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
%                        Bucle simulación                               %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Fl = 1.0 ;            % Fuerza helice izquierda (N)
Fr = 1.0 ;            % Fuerza helice derecha (N)

U0 = [Fl Fr]' ;

X = X0 ;
U = U0 ;
Trayectoria = [] ;
SB_traza = [] ;

for i= 1:100

    % Nube de puntos del Velodyne
    BOAT_T_ABS = ABS_T_BOAT^-1 ;

    % Filtrado de la nube de puntos
    Vel = pctransform(Galeria3D,affine3d(BOAT_T_ABS'));
    [indices,dists] = findNeighborsInRadius(Vel,[0 0 0],10.0) ;
    Vel = select(Vel,indices) ;
    indices = find (Vel.Location(:,3) < 1.0) ; %Planchado de la nube coord z
< 1.0
    Vel = select(Vel,indices) ;
    indices = find (Vel.Location(:,3) <
0.27*sqrt(Vel.Location(:,1).^2+Vel.Location(:,2).^2)) ;
```

```
    Vel = select(Vel,indices) ;

    % Aplicacion del metodo Siliding Ballon V2

     [LG1, BR1, LG2, BR2] = SB2D_stepV2(Vel, AdvanceVector);
     Circle(LG2,BR2);
    % Cálculo del angulo que define el rumbo RRef
    if (abs(LG1(1))<0.01)
        RRef = LG1(2)/abs(LG1(2))*pi/2.0 ;
    else
        RRef = Bisectriz(LG1,LG2);
    end ;

    if BR1 < Rmin  %Cambio de direccion por estar demasiado cerca de la pared
        AdvanceVector = [0.0 1.0 0] ;
        LG1 = [0.0 1.0 0.0] ;
        %         U = [0 ; 0] ;
    else
        AdvanceVector = AdvanceVector_Mission ;
    end ;

    %Comprobación de paso
    PASA = Paso(RRef, LG2, BR2, Vel);
    if not(PASA)
        j = 0;
    end

    % Si no PASA se aplica otro AV durante unas pocas iteraciones, permite
    % salir de recovecos
    if (j < 5)
        AdvanceVector = [0.866 0.5 0.0] ; % AdvanceVector para los siguientes
globos +30 grados
        j = j + 1;
    end

    if (abs(RRef)<pi/4.0)
        if (BR1 > Rmax && BR2 > Rmax) %Si ambos globos son maximos, velocidad
maxima
            Vref = Vref_max ;
        else
            % Calculo de la velocidad correspondiente con interpolación
            Vref = (BR1 - Rmin)/(Rmax - Rmin) * (Vref_max - Vref_min) +
Vref_min ;
            Vref = Vref*(1 - abs(RRef)/pi/4); %Reduccion de velocidad segun
angulo
        end ;
    else
        if (abs(RRef)<pi/2.0)
            Vref = Vref_max * (pi/4.0 - abs(RRef))/(pi/4.0) ;
        else
            Vref = -Vref_max ; % marcha atrás de golpe porque se va a chocar
        end ;
    end ;

    if RRef < 0.0
        RRef = RRef + 2*pi ; %Hace el ángulo positivo, luego se normaliza en
RRumbo
    end ;
```

```
    Vb = X(1)*cos(X(6)) + X(2)*sin(X(6)) ;
    P = [X(4) ; X(5)] ;

F = Vref ;

    dF = RRumbo (RRef, 0.0) ; %(Rumbo que quiero tener, rumbo que lleva el
barco)
    U = [F-dF ; F+dF] ;       %RRumbo calcula diferencia de velocidades entre
helices

    % Simulación dinámica RoboBoat

    X = boat (X, U, Param) ;

    % Nueva referencia BOAT
    Eje_X = [cos(X(6)) sin(X(6)) 0] ;
    Eje_X = Eje_X/norm (Eje_X) ;
    Eje_Y = [-Eje_X(2) Eje_X(1) 0] ;
    Eje_Z = [0 0 1] ;
    % Matriz rotacion
    R = [Eje_X' Eje_Y' Eje_Z'] ;
    % Traslacion
    T = [X(4) X(5) 0.0]' ;
    ABS_T_BOAT = [[R T] ; [0 0 0 1]] ;

    Trayectoria = [Trayectoria ; X(1:6)'] ;
    SB_traza = [SB_traza ; [LG1, BR1]] ;


end ;

    figure (1) ;
    pcshow (Lim_Gal) ;
    hold on ;
    xlabel ('X coordinate') ;
    ylabel ('Y coordinate') ;

plot3 (Trayectoria(:,4),Trayectoria(:,5),0*Trayectoria(:,5)) ;
for i=1:20:length(Trayectoria)
    PlotRob3(Trayectoria(i,4), Trayectoria(i,5), Trayectoria(i,6)) ;
end ;

hold off ;
```

9.9.3. Funciones añadidas

```
function [bisec] = Bisectriz(punto1,punto2)
%BISECTRIZ Calcula el angulo medio que forman dos puntos con el origen
%sobre el eje x (en radianes)
tita = atan2(punto1(2),punto1(1));
phi = atan2(punto2(2),punto2(1));
bisec = ((phi-tita)/2) + tita;
end
```

```
function result = Paso(rumbo, punto2,radio2,Velodyne_Cloud)
%PASO Calcula si el barco puede pasar dado el rumbo, el segundo globo y una
nube de puntos
%   rumbo: rumbo de avance calculado
%   punto2: centro del globo 2
%   radio2: radio del globo 2
%   Velodyne_Cloud: nube de puntos del velodyne

global Rmin;
% punto que se encuentra en la trayectoria que define el rumbo
vector=[1 sin(rumbo) 0];
% vector que define el rumbo
vector = vector/norm(vector);
% inicializacion de minima distancia a un valor alto
minDist = 10.0;
% planchado de nube de puntos y busqueda de puntos en radio
ptCloud = pointCloud ([Velodyne_Cloud.Location(:,1)
Velodyne_Cloud.Location(:,2) 0.0*Velodyne_Cloud.Location(:,3)]) ;
[indices, dists]=findNeighborsInRadius(ptCloud,punto2,radio2) ;

if isempty(indices)
    %Si no hay puntos en la direccion dada, se puede pasar directamente
    result = true;
else
    %Separar los puntos que estan a izqa y dcha del vector director
    result = true; %se presupone que se va a poder pasar
    izq = [];
    dcha = [];
    for i = 1:length(indices)
        iv = ptCloud.Location(indices(i),:);
        xv = cross(vector,iv);
        if xv(3) < 0
            %el punto esta a la derecha del vector
            dcha = [dcha; ptCloud.Location(indices(i),:)];
        elseif xv(3) > 0
            %el punto esta a la izquierda del vector
            izq = [izq; ptCloud.Location(indices(i),:)];
        else
            % el punto esta en la misma dirección
            result = false;
            break;
        end
    end

    if result
        for i = 1:length(dcha)
            for j = 1:length(izq)
```

```
                % Buscar la minima distancia entre los puntos de un lado y
otro
                distance = norm(izq(j)-dcha(i));
                % Comparación con el radio mínimo
                if distance < Rmin
                    result = false;
                    break;
                % ESTE ULTIMO IF NO SERÍA NECESARIO
                elseif distance < minDist
                    minDist = distance;
                end
            end
        end
    end
end
```

9.2. Código de C/ROS

9.2.1. Módulo de funciones geométricas.

9.2.1.1. geometry.h

```
/*********************************************************************
******/
/*project  :ROBOBOAT                                                 */
/*filename:geometry.h                                                 */
/*version  :2                                                         */
/*date     :27/05/2022                                                */
/*author   :Jose Luis Villarroel + Lidia                          */
/* description : Basic functions to manage geometry in
SlidingBalloon        */
/*********************************************************************
******/

#ifndef GEOMETRY_H_
#define GEOMETRY_H_

#define pi 3.1416

typedef float Vector2[2] ;

typedef Vector2 Point ;
typedef Vector2 Vector ;

typedef struct {
    unsigned int Count;
    Point Location[450];
} PointCloud2D;

typedef float HomogeneousTransformation[3][3] ;

float Distance (Point P1, Point P2) ;
float findNearestNeighbor (PointCloud2D *ptCloud, Point P, Point
Neighbor) ;
void SolveCircleIntersection (Point c1, float r1, Point c2, float r2,
Point s1, Point s2) ;
char SameSide (Vector divisorv, Vector v1, Vector v2) ;
void NearestSol (Point Obj, Point p1, Point p2, Point Sol) ;
void findNeighborsInRadius(PointCloud2D *ptCloud, Point P, float Radius,
PointCloud2D *sol) ;
void CloudCopy (PointCloud2D *ptCloud, PointCloud2D *ptCopy) ;
void Polar2Cart (PointCloud2D *ptCloud) ;
float norm (Vector V) ;
float Bisectriz (Point P1, Point P2);

#endif /* GEOMETRY_H_ */
```

9.2.1.2. geometry.cpp

```
/*****************************************************************
******/
/* project  :
ROBOBOAT                                                        */
/*filename:geometry.c                                            */
/*version  :2                                                    */
/*date     :27/05/2022                                           */
/*author   :Jose Luis Villarroel + Lidia                            */
/* description : Basic functions to manage geometry in Sliding
BAlloon        */
/*****************************************************************
******/

/*****************************************************************
******/
/*              Used modules                                      */
/*****************************************************************
******/

#include "math.h"
#include "geometry.h"
#include <cstring>
#include <iostream>
using namespace std;

/*****************************************************************
******/
/*               Exported functions                               */
/*****************************************************************
******/

float Distance (Point P1, Point P2) {
    return sqrt(pow((P1[0] - P2[0]),2) + pow((P1[1] - P2[1]),2)) ;
}

float findNearestNeighbor (PointCloud2D *ptCloud, Point P, Point
Neighbor) {

    float dist = 0.0, min_dist = 10000.0 ;
    unsigned int i ;
    Point PN ;

    for (i=0; i<ptCloud->Count; i++) {
        dist = Distance (ptCloud->Location[i],P) ;
        if (dist < min_dist) {
            PN[0] = ptCloud->Location[i][0] ;
```

```
            PN[1] = ptCloud->Location[i][1] ;
            min_dist = dist ;
        }
    }
    Neighbor[0] = PN[0] ;
    Neighbor[1] = PN[1] ;
    return min_dist ;
}

void SolveCircleIntersection (Point c1, float r1, Point c2, float r2,
Point s1, Point s2) {

    float a, b, c, epsilon ;
    float M, N ;
    float xc1, xc2, yc1, yc2 ;
    float xp1, xp2, yp1, yp2 ;

    // Obtiene los dos puntos de interseccion de dos circunferencias
    //   Circunferencia1: centro c1, radio r1
    //   (x - xc1)^2 + (y - yc1)^2 = r1^2

    xc1 = c1 [0] ;
    yc1 = c1 [1] ;

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

    xc2 = c2 [0] ;
    yc2 = c2 [1] ;

    epsilon = 0.001 ;

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

        xp1 = xc1 + sqrt(pow(r1,2) - pow(yp1 - yc1, 2)) ;
        xp2 = xc1 - sqrt(pow(r1,2) - pow(yp1 - yc1, 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 ;
    }

    s1[0] = xp1 ;
    s1[1] = yp1 ;
    s2[0] = xp2 ;
    s2[1] = yp2 ;

    return ;
}

char SameSide (Vector divisorv, Vector v1, Vector v2) {
//Decide si dos vectores estan en el mismo lado de una divisoria
//   divisorv: vector que define la divisoria
//   v1, v2: los dos vectores

    float zcross1, zcross2 ;

  zcross1 = divisorv[0]*v1[1] - divisorv[1]*v1[0] ;
  zcross2 = divisorv[0]*v2[1] - divisorv[1]*v2[0] ;
  if (zcross1*zcross2 > 0) return 1 ;
  else return 0 ;
}

void NearestSol (Point Obj, Point p1, Point p2, Point Sol) {
    if (Distance(Obj,p1) > Distance(Obj,p2)) {Sol[0] = p2[0] ; Sol[1] =
p2[1] ;}
    else {Sol[0] = p1[0] ; Sol[1] = p1[1] ;}
}


void findNeighborsInRadius(PointCloud2D *ptCloud, Point P, float Radius,
PointCloud2D *sol) {

    unsigned int i, j = 0 ;
    float distance ;

    sol->Count = 0 ;
    for (i=0;i<ptCloud->Count;i++) {
```

```
        distance = sqrt(pow(P[0] - ptCloud->Location[i][0],2) + pow(P[1]
- ptCloud->Location[i][1], 2)) ;
        if (distance <= Radius) {
            sol->Location[j][0] = ptCloud->Location[i][0] ;
            sol->Location[j][1] = ptCloud->Location[i][1] ;
            sol->Count ++ ; j++ ;
        }
    }
    return ;
}

void CloudCopy (PointCloud2D *ptCloud, PointCloud2D *ptCopy) {

    std::memcpy (ptCloud->Location, ptCopy->Location, ptCloud->Count*8) ;
    ptCloud->Count = ptCopy->Count ;
}


void Polar2Cart (PointCloud2D *ptCloud) {
    float angle, range ;
    unsigned int i ;

    for (i=0;i<ptCloud->Count;i++) {
        angle = ptCloud->Location[i][1] ;
        range = ptCloud->Location[i][0] ;
        ptCloud->Location[i][0] = range * cos(angle) ;
        ptCloud->Location[i][1] = range * sin(angle) ;

    }
}


float norm (Vector V) {
    return sqrt(pow(V[0],2) + pow(V[1],2)) ;
}


float Bisectriz (Point P1, Point P2){

    float tita, phi;
    //Calcula el angulo medio que forman dos puntos con el origen (en
radianes)
    tita = atan2(P1[1],P1[0]);
    phi = atan2(P2[2],P2[0]);
    return ((phi - tita)/2) + tita;
}
```

9.2.2. Clase C++ que implementa el “Sliding Balloon”

9.2.2.1. LoLargo_SB.h

```
#ifndef Lolargo_SB_H
#define Lolargo_SB_H

#include "geometry.h"

class Lolargo_SB
{
public:
//    Lolargo_SB() ;
    void Configure (double AdvanceStep_, double Precision_, double Rmin_,
double Rmax_);

    void SB2D_stepV2(PointCloud2D* ptCloud, Vector AdvanceVector, Vector
LocalGoal1, float* BalloonRadius1, Vector LocalGoal2, float*
BalloonRadius2);  // SB v2

    char Paso (float Rumbo, Point P, float Radius, PointCloud2D
*ptCloud);

private:

    //parameters
    double AdvanceStep = 0.5;
    double Precision = 0.1;
    double Rmin = 0.2;
    double Rmax = 0.5;

    //local variables
    bool EndInflate;

    Point RobotCenter = {0.0, 0.0} ;
    Point BalloonCenter ;
    Point BalloonCenter1, BalloonCenter2;   // SB v2

    Point p1, p2 ;
    Point NearestPoint ;

    PointCloud2D Sbscriber_scan, Protected_scan, NCloud ;
    Vector nv, iv ;

    float d, r, dist, l ;
    unsigned int i, j ;

};

#endif // Lolargo_SB_H
```

9.2.2.2. LoLargo_SB.cpp

```
#include "Lolargo_SB.h"
#include <math.h>
#include <iostream>

using namespace std;

void Lolargo_SB::Configure(double AdvanceStep_, double Precision_, double
Rmin_, double Rmax_)
{
    AdvanceStep = AdvanceStep_;
    Precision = Precision_;
    Rmin = Rmin_;
    Rmax = Rmax_;
}


void Lolargo_SB::SB2D_stepV2 (PointCloud2D* ptCloud, Vector
AdvanceVector, Vector LocalGoal1, float* BalloonRadius1, Vector
LocalGoal2, float* BalloonRadius2){
    //Segunda version del Sliding Balloon 2D, calcula los dos siguientes
puntos de la trayectoria usando dos globos
    //  ptCLoud: Nube de puntos obtenida por un laser plano en la
referencia local del robot
    //  AdvanceVector: Direccion inicial de avance

    unsigned int max_iter, n_iter = 0;

    BalloonCenter1[0] = AdvanceVector[0]*AdvanceStep; BalloonCenter1[1] =
AdvanceVector[1]*AdvanceStep;
    EndInflate = false;
    d = Precision / 2.0;
    max_iter = (unsigned int)(Rmax/d);

    //Calculo del primer globo, es un SB normal
    while(!EndInflate){

        if (ptCloud->Count == 0){
            *BalloonRadius1 = Rmax;
            break;
        }

        n_iter ++;
        if (n_iter > max_iter){
            *BalloonRadius1 = 0.0;
            break;
        }

        dist = findNearestNeighbor(ptCloud,BalloonCenter1,NearestPoint);
```

```
        r = dist;
        l = norm (NearestPoint);
        *BalloonRadius1 = r + d;

        if (*BalloonRadius1 > Rmax) break;
        if (l + AdvanceStep < *BalloonRadius1) break; //Condicion de
maxima desviacion

        SolveCircleIntersection(RobotCenter, AdvanceStep, NearestPoint,
*BalloonRadius1, p1, p2);
        NearestSol(BalloonCenter1, p1, p2, BalloonCenter1);

        findNeighborsInRadius(ptCloud,BalloonCenter1,*BalloonRadius1,&NCl
oud) ;

        nv[0] = NearestPoint[0] - BalloonCenter1[0] ; nv[1] =
NearestPoint[1] - BalloonCenter1[1] ;
        EndInflate = false ;

        //Se busca si hay puntos del entorno a ambos lados del vector
director
        for (i=0;i<NCloud.Count;i++) {
            iv[0] = NCloud.Location[i][0] - BalloonCenter1[0] ;
            iv[1] = NCloud.Location[i][1] - BalloonCenter1[1] ;
            if (!SameSide (AdvanceVector, nv, iv)){
                EndInflate = true ;
                break ;
            }
        }
    }

    LocalGoal1[0] = BalloonCenter1[0];
    LocalGoal1[1] = BalloonCenter1[1];

    //Calculo del segundo globo sobre el primero

    BalloonCenter2[0] = BalloonCenter1[0]*(*BalloonRadius1 +
AdvanceStep)/norm(BalloonCenter1);
    BalloonCenter2[1] = BalloonCenter1[1]*(*BalloonRadius1 +
AdvanceStep)/norm(BalloonCenter1);

        *BalloonRadius2 = *BalloonRadius1;

    EndInflate = false;
    n_iter = 0; //Reinicio el contador de iteraciones

    //Vemos si donde se lanza el segundo globo hay puntos que interfieran
en la trayectoria
```

```
    findNeighborsInRadius(ptCloud,BalloonCenter2,*BalloonRadius1,&NCloud)
;
    if(NCloud.Count == 0){
        LocalGoal2[0] = BalloonCenter2[0];
        LocalGoal2[1] = BalloonCenter2[1];
        *BalloonRadius2 = *BalloonRadius1;
    }

    //Si no es el caso, se construye el segundo globo
    else{
        while(!EndInflate){
            if (ptCloud->Count == 0){
                *BalloonRadius2 = Rmax;
                break;
            }

            n_iter ++;
            if (n_iter > max_iter){
                *BalloonRadius2 = 0.0;
                break;
            }

            dist =
findNearestNeighbor(ptCloud,BalloonCenter2,NearestPoint);
            r = dist;

            Point laux;
            laux[0] = NearestPoint[0] - BalloonCenter2[0];
            laux[1] = NearestPoint[1] - BalloonCenter2[1];
            l = norm (laux);

            *BalloonRadius2 = r + d;

            if (*BalloonRadius2 > Rmax) break;
            if (l + *BalloonRadius1 < *BalloonRadius2) break; //Condicion
de maxima desviacion

            SolveCircleIntersection(BalloonCenter1, *BalloonRadius1,
NearestPoint, *BalloonRadius2, p1, p2);
            NearestSol(BalloonCenter2, p1, p2, BalloonCenter2);

            findNeighborsInRadius(ptCloud,BalloonCenter2,*BalloonRadius2,
&NCloud) ;

            nv[0] = NearestPoint[0] - BalloonCenter2[0] ; nv[1] =
NearestPoint[1] - BalloonCenter2[1] ;
            EndInflate = false ;

            for (i=0;i<NCloud.Count;i++) {
```

```
                iv[0] = NCloud.Location[i][0] - BalloonCenter2[0] ;
                iv[1] = NCloud.Location[i][1] - BalloonCenter2[1] ;
                if (!SameSide (AdvanceVector, nv, iv)){
                    EndInflate = true ;
                    break ;
                }
            }
        }
    }

    LocalGoal2[0] = BalloonCenter2[0];
    LocalGoal2[1] = BalloonCenter2[1];


}

char Lolargo_SB::Paso (float Rumbo, Point P, float Radius, PointCloud2D
*ptCloud){
    // Calcula si el barco podria pasar dados el segundo globo y el rumbo
calculado

    float minDist, distance, position;
    PointCloud2D dcha, izda ;

    float vec[] = {1, sin(Rumbo)};
    // Busqueda de puntos dentro del radio
    PointCloud2D puntos;


    findNeighborsInRadius(ptCloud, P, Radius, &puntos);

    // Si no hay puntos se puede seguir
    if (puntos.Count == 0){
        return 1;
    }
    //Si los hay, se debe comprobar el paso
    else{
        for (i=0;i<puntos.Count;i++){
            iv[0] = puntos.Location[i][0];
            iv[1] = puntos.Location[i][1];
            position = (vec[0]*iv[1])-(vec[1]*iv[0]);

            // Separación de puntos a izqda y dcha del vector director
            if (position < 0){
                dcha.Location[i][0] = puntos.Location[i][0];
                dcha.Location[i][1] = puntos.Location[i][1];
                dcha.Count ++ ;
            }
            else if (position > 0){
```

```
                izda.Location[i][0] = puntos.Location[i][0];
                izda.Location[i][1] = puntos.Location[i][1];
                izda.Count ++ ;
            }
            else{
                return 0; //Algun punto esta encima del mismo vector
            }
        }

        //Búsqueda de la mínima distancia entre puntos de izqda y dcha
        for (i=0;i<dcha.Count;i++){
            for (j=0;j<izda.Count;j++){
                distance = Distance(dcha.Location[i],izda.Location[j]);
                if (distance < minDist){
                    minDist = distance;
                }
            }
        }
        if (minDist < Rmin){
            return 0; //La menor distancia es poca para poder pasar
        }
        else{
            return 1; //Se puede pasar
        }
    }
}
```

9.2.3. Programa C++ que implementa un nodo ROS de navegación con el método (RoboBoatV2.cpp)

```
/*********************************************************************
******/
/*project  :ROBOBOAT                                                 */
/*filename: RoboBoat.cpp                                             */
/*version  :3                                                        */
/*date     :27/05/2022                                               */
/*author   :Jose Luis Villarroel + Lidia                             */
/* description : This module implements a ROS navigation node to be used
in   */
/*               Lo Largo (Cueva Roman, CLUNIA). Based on Sliding
Balloon.    */
/*********************************************************************
******/

/*********************************************************************
******/
/*                  Used modules                                      */
/*********************************************************************
******/
#include <math.h>
#include <ros/ros.h>
#include <serial/serial.h>
#include <visualization_msgs/Marker.h>
#include <std_msgs/String.h>
#include <std_msgs/Empty.h>
#include <sensor_msgs/LaserScan.h>
#include <sensor_msgs/PointCloud.h>
#include <laser_geometry/laser_geometry.h>

#include <sstream>
#include <string>

#include "Lolargo_SB.h"
#include "geometry.h"

/*********************************************************************
******/
/*                  Typedefs and structures                           */
/*********************************************************************
******/

using namespace std ;

#define PI 3.1416
```

```
/*********************************************************************
******/
/*                  Global variables                                 */
/*********************************************************************
******/

//ros stuff
ros::Subscriber scan_subscriber;
ros::Publisher goal_publisher1;
ros::Publisher goal_publisher2;
ros::Publisher cloud_publisher;
visualization_msgs::Marker marker;
geometry_msgs::Point BC_msg;
sensor_msgs::PointCloud cloud;
laser_geometry::LaserProjection projector_;
double max_laser_range= 30.0, laser_offset = -3.14159274101/2.0 ;


// Parameters of Sliding Balloon
double Rmin = 0.3 ;
double Rmax = 0.6 ;
double Advance_Step = 1.2 ;
double Precision = 0.1 ;
Vector AdvanceVector_Mission = {1.0, -0.2} ;

// Variables of Sliding Balloon
PointCloud2D scan;
Vector Advance_Vector;
Vector Local_Goal1, Local_Goal2;
float Balloon_Radius1, Balloon_Radius2, Balloon_Radius_Min, nam;
char PASA = 0 ;
unsigned char j = 5;

// Sliding Balloon instance
Lolargo_SB LLSB;

// RoboBoat
serial::Serial ser;

float Vref, dF, RRef ;

float Vref_min = 0.2 ;
float Vref_max = 0.4 ;

float K = 0.2 ;
float Kd = 0.0 ;
float T = 1 ;

unsigned int n = 0 ;
```

```
/*****************************************************************
******/
/*           Local function prototypes                           */
/*****************************************************************
******/

void scan_callback(const sensor_msgs::LaserScan::ConstPtr& msg);
void ConfigureMarker (void) ;
float RRumbo (float Ref) ;


/*****************************************************************
******/
/*                 main code                                     */
/*****************************************************************
******/

int main(int argc, char** argv) {
    ros::init(argc, argv, "RoboBoat_NodeV2");
    ros::NodeHandle nh;

    scan_subscriber = nh.subscribe("/horizontal_scan", 1, scan_callback);
    goal_publisher1 =
nh.advertise<visualization_msgs::Marker>("Balloon1", 1);
    goal_publisher2 =
nh.advertise<visualization_msgs::Marker>("Balloon2", 1);
    cloud_publisher = nh.advertise<sensor_msgs::PointCloud>("Entorno",
1);

    nh.getParam ("/RoboBoat_NodeV2/Rmin", Rmin) ;
    nh.getParam ("/RoboBoat_NodeV2/Rmax", Rmax) ;
    nh.getParam ("/RoboBoat_NodeV2/Advance_Step", Advance_Step) ;
    nh.getParam ("/RoboBoat_NodeV2/Precision", Precision) ;

    ROS_INFO("Rmin = %f, Rmax = %f, Advance_Step = %f, Precision = %f",
Rmin, Rmax, Advance_Step, Precision) ;
    LLSB.Configure (Advance_Step, Precision, Rmin, Rmax) ;

    nh.getParam ("/RoboBoat_NodeV2/Vref_min", Vref_min) ;
    nh.getParam ("/RoboBoat_NodeV2/Vref_max", Vref_max) ;
    nh.getParam ("/RoboBoat_NodeV2/K", K) ;
    nh.getParam ("/RoboBoat_NodeV2/Kd", Kd) ;

    ROS_INFO("Vref_min = %f, Vref_max = %f, K = %f, Kd = %f", Vref_min,
Vref_max, K, Kd) ;

    nh.getParam ("/RoboBoat_NodeV2/Advance_Vector_x",
AdvanceVector_Mission[0]) ;
```

```
    nh.getParam ("/RoboBoat_NodeV2/Advance_Vector_y",
AdvanceVector_Mission[1]) ;

    nam = norm(AdvanceVector_Mission);
    AdvanceVector_Mission[0] = AdvanceVector_Mission[0] / nam;
AdvanceVector_Mission[1] = AdvanceVector_Mission[1] / nam;
    Advance_Vector[0] = AdvanceVector_Mission[0]; Advance_Vector[1] =
AdvanceVector_Mission[1];

    ROS_INFO("Advance_Vector_x = %f, Advance_Vector_y = %f",
AdvanceVector_Mission[0], AdvanceVector_Mission[1]) ;


    ConfigureMarker() ;

    try
    {
        ser.setPort("/dev/ttyRB0");
        ser.setBaudrate(9600);
        serial::Timeout to = serial::Timeout::simpleTimeout(50);
        ser.setTimeout(to);
        ser.open();
    }
    catch (serial::IOException& e)
    {
        ROS_ERROR_STREAM("Unable to open port ");
        return -1;
    }

    if (ser.isOpen()) {
        ROS_INFO_STREAM("Serial Port initialized");
    }
    else {
        return -1;
    }

    ros::spin();
}




/*********************************************************************
******/
/*           Local function code                                     */
/*********************************************************************
******/
```

```
void scan_callback(const sensor_msgs::LaserScan::ConstPtr& msg)
{
   if (msg->ranges.size() < 200) {
      projector_.projectLaser(*msg, cloud, 3.0);
      cloud_publisher.publish(cloud);
      for (int i = 0; i < cloud.points.size(); i++) {
         scan.Location[i][0] = cloud.points[i].x;
         scan.Location[i][1] =  cloud.points[i].y;
      }
      scan.Count = cloud.points.size();

      try
      {
        LLSB.SB2D_stepV2(&scan, Advance_Vector, Local_Goal1,
&Balloon_Radius1, Local_Goal2, &Balloon_Radius2);
        ROS_INFO("LG_x1 = %f, LG_y1 = %f, BR1 = %f",
Local_Goal1[0],Local_Goal1[1],Balloon_Radius1) ;
        ROS_INFO("LG_x2 = %f, LG_y2 = %f, BR2 = %f",
Local_Goal2[0],Local_Goal2[1],Balloon_Radius2) ;
      }
      catch (int e)
      {
         Balloon_Radius2 = 0.0 ;
         ROS_INFO("SB2D exception") ;
      }

        marker.pose.position.x = Local_Goal1[0];
        marker.pose.position.y = Local_Goal1[1];
        marker.scale.x = 2.0*Balloon_Radius1;
        marker.scale.y = 2.0*Balloon_Radius1;

        goal_publisher1.publish(marker);

        marker.pose.position.x = Local_Goal2[0];
        marker.pose.position.y = Local_Goal2[1];
        marker.scale.x = 2.0*Balloon_Radius2;
        marker.scale.y = 2.0*Balloon_Radius2;

        goal_publisher2.publish(marker);



      if (Balloon_Radius2 < Rmin) {
         Advance_Vector[0] = 0.866 ; Advance_Vector[1] = 0.5 ;
         Local_Goal1[0] = Advance_Vector[0] ; Local_Goal1[0] =
Advance_Vector[1] ;
         j = 0 ;
      }     // 30 grados izq
```

```
      if (Balloon_Radius1 < Rmin) {
         Advance_Vector[0] = 0.0 ; Advance_Vector[1] = 1.0 ;
         Local_Goal1[0] = Advance_Vector[0] ; Local_Goal1[0] =
Advance_Vector[1] ;
         j = 0 ;
      }      // Todo a la izquierda

      if (j < 5) {j++ ; ROS_INFO("j = %i", j) ; }
      else Advance_Vector[0] = AdvanceVector_Mission[0];
Advance_Vector[1] = AdvanceVector_Mission[1];

      if (fabsf(Local_Goal1[0])<0.01) RRef =
Local_Goal1[1]/fabsf(Local_Goal1[1])*PI/2.0 ;
      else RRef = atan2(Local_Goal1[1],Local_Goal1[0]) ;

      if (Balloon_Radius1 < Balloon_Radius2) Balloon_Radius_Min =
Balloon_Radius1 ;
      else Balloon_Radius_Min = Balloon_Radius1 ;

      if (fabsf(RRef)<PI/4.0) {
         if (Balloon_Radius_Min > Rmax) Vref = Vref_max ;
         else Vref = (Balloon_Radius_Min - Rmin)/(Rmax - Rmin) *
(Vref_max - Vref_min) + Vref_min ;
         Vref = Vref*(1 - fabsf(RRef)/PI/4);
      }
      else if (fabsf(RRef)<PI/2.0) Vref = Vref_max * (PI/4.0 -
fabsf(RRef))/(PI/4.0) ;
      else Vref = -Vref_max ;


      if (RRef < 0.0) RRef = RRef + 2.0*PI ;

      dF = RRumbo (RRef) ;

      std::stringstream ss ;
      ss << "#SRVW " << Vref << "," << dF << "," << n << "\r\n" ;
      std::string order = ss.str() ;
      char ch[80] ;
      order.copy(ch,order.length(),0);

      ROS_INFO_STREAM(ss.str());
      ser.write((uint8_t *)ch, (size_t)order.length()) ;
   }
}

void ConfigureMarker (void)
{
   marker.header.frame_id = "velodyne";
   marker.header.stamp = ros::Time::now();
```

```
    marker.ns = "my_namespace";
    marker.id = 0;
    marker.type = visualization_msgs::Marker::SPHERE;
    marker.action = visualization_msgs::Marker::ADD;
    marker.pose.position.x = 0.0;
    marker.pose.position.y = 0.0;
    marker.pose.position.z = 0.0;
    marker.pose.orientation.x = 0.0;
    marker.pose.orientation.y = 0.0;
    marker.pose.orientation.z = 0.0;
    marker.pose.orientation.w = 1.0;
    marker.scale.x = 1.0;
    marker.scale.y = 1.0;
    marker.scale.z = 0.1;
    marker.color.a = 1.0; // Don't forget to set the alpha!
    marker.color.r = 0.0;
    marker.color.g = 1.0;
    marker.color.b = 0.0;
    marker.lifetime = ros::Duration();
}

float RRumbo (float Ref)
{
    // Calcula la diferencia de fuerza de las helices para corregir el
rumbo
    //   RRef : referencia de rumbo
    //   accion_R : diferencia de fuerza en las helices

    float accion_R ;
    static float error_R = 0 ;
    static float error_ant_R = 0 ;

    error_R = RRef - 0.0 ;
    if (abs(error_R) > PI)
        if (error_R < 0) error_R = error_R + 2*PI ;
        else error_R = error_R - 2*PI ;

    accion_R = K*error_R + Kd/T*(error_R - error_ant_R) ;

    error_ant_R = error_R ;

    return -accion_R ;
}
```