UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN


GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

# DISEÑO E IMPLEMENTACIÓN DE UN SIMULADOR 3D DE ROBOTS SUBMARINOS PARA LA OPTIMIZACIÓN DE ALGORITMOS DE CONTROL DISTRIBUIDOS

**Javier Madrigal Torija**

**2022**

# GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

## TRABAJO FIN DE GRADO

**Título:** Diseño e implementación de un simulador 3D de robots submarinos para la optimización de algoritmos de control distribuido.

**Autor:** D. Javier Madrigal Torija

**Tutor:** D. Rafael Sendra Arranz

**Ponente:** D. Álvaro Gutiérrez Martin

**Departamento:** Departamento de Tecnología Fotónica y Bioingeniería

## MIEMBROS DEL TRIBUNAL

**Presidente:** D. ……

**Vocal:** D. ……..

**Secretario:** D. ……..

**Suplente:** D. ……..

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de: ……

Madrid, a de de 20

# UNIVERSIDAD POLITÉCNICA DE MADRID

## ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN


## GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

### TRABAJO FIN DE GRADO

# DISEÑO E IMPLEMENTACIÓN DE UN SIMULADOR 3D DE ROBOTS SUBMARINOS PARA LA OPTIMIZACIÓN DE ALGORITMOS DE CONTROL DISTRIBUIDOS

**Javier Madrigal Torija**

**2022**

# RESUMEN

Los simuladores de robots son herramientas muy potentes y utilizadas, ya que permiten a los desarrolladores probar controladores en entornos seguros antes que en sistemas reales donde se arriesgue la integridad de los equipos. Sin embargo, el desarrollo de estos simuladores suele ir acompañado del crecimiento del sector robótico al que pertenecen. Es por ello que, la mayoría de los simuladores son de robots terrestres, ya que es el ámbito de la robótica más estudiado. Ello implica que en campos menos desarrollados como la robótica acuática no sean muy comunes. Por lo tanto, ante una necesidad generada por el reciente crecimiento en la economía azul, este Trabajo de Fin de Grado se enfoca en el diseño e implementación de un simulador de robótica acuática.

El medio acuático siempre ha supuesto muchos problemas a la robótica, ya que implica arriesgar los sistemas en cada uso, haciendo todavía más importante el desarrollo de estos simuladores que permitan probar los funcionamientos previamente. El simulador que se ha desarrollado plantea, en su versión inicial, un entorno implementado con Python que gestiona los gráficos mediante OpenGL, en que se es capaz de simular entornos con múltiples robots que interaccionen entre sí y con el medio. Estas interacciones se realizarán mediante los sensores con los que cuenta cada robot. Los sensores implementados en este TFG son un sensor GPS, seis sensores de distancia (uno en cada eje) y un radar en el plano $x$ e $y$. Finalmente, el simulador también cuenta con la capacidad de realizar ajustes rápidos por línea de ejecución y archivos de configuración en los que definir los escenarios de cada experimento. Además, también permite exportar datos para posteriormente analizarlos.

A fin de comprobar y mostrar el uso del simulador, en la segunda parte del TFG se han desarrollado una serie de experimentos para verificar el correcto funcionamiento de este. Siendo el ensayo principal, uno de los comportamientos más básicos que debe tener todo robot real, la capacidad de navegar evitando obstáculos. Para su desarrollo se han realizado experimentos secundarios en los que se investigue a cerca de las capacidades de movilidad que tiene el robot. En concreto se han analizado cómo afecta la inercia del medio al movimiento, cómo conseguir una flotabilidad estable y cómo adaptar la velocidad para mantenerla constante en el medio. Se ha obtenido así una visión general del funcionamiento y de la respuesta del propio medio en el que funciona el simulador.

# PALABRAS CLAVE

Robótica submarina, Simulador robótico, *BlueRov2*, Python, OpenGL, Robótica colectiva, Algoritmos de control distribuido, Control de velocidad, Control de flotabilidad, Navegación evitando obstáculos, Simulador robótico submarino.

# ABSTRACT

Robotics simulators are powerful and widely used tools as they allow developers to evaluate controllers in safe environments, rather than in real systems where the integrity of the equipment is at risk. However, these simulators are developed mostly because of the growth of the robotics sector, where they are needed. Therefore, most simulators are devoted to terrestrial robotics since this is the most studied subfield of robotics. Thus, underwater robotics simulators are not common at all, as expected. Nevertheless, in the last decade *blue economy* has been growing. This growth has been slowly creating the need of simulators in the community, so the purpose of this BSC Thesis is to fulfil that lack of development.

The aquatic environment has always implied several problems for robotics applications as it involves the risking at the systems in use, imposing as a requirement the previous assessment of the robots in a simulator. As an initial version an underwater robotics simulator has been develop using *Python* and *OpenGL* to manage the graphics. The simulator generates environments with multiple robots that interact with each other and with the environment. These interactions are carried out using the sensors of each robot, which are a GPS sensor, six distance sensors (one on each axis), and a radar in the x and y plane. Finally, the simulator also has the ability to make quick adjustments through command line and using configuration files that define different scenarios.

In order to test several the simulator, the second part of the BSC Thesis is devoted to the development of several experiments within the simulator. The main assessment validates the most basic behaviours that every real robot must-have, the ability to navigate avoiding obstacles. For the development of this, secondary experiments will be carried out to investigate: the mobility capabilities of the robot, how the inertia of the environment affects the movement, how to achieve a stable buoyancy, and how to adapt the speed to keep it constant in the environment. Thus, obtaining an overview of the operation and response of the environment in which the simulator operates.



# KEYWORDS

Underwater robotics, Robotic simulator, BlueRov2, Python, OpenGL, Collective robotics, Distributed control algorithms, Speed control, Buoyancy control, Navigation avoiding obstacles, Underwater robotic simulator.

# LISTA DE FIGURAS

# LISTA DE ACRÓNIMOS

**UNED:** Universidad Nacional de Educación a Distancia

**AUV:** *Autonomous Underwater Vehicles*

**GPS:** *Global Positioning System*

**URDF:** *Unified Robot Description Format*

**JSON:** *JavaScript Object Notation*

**CSV:** C*omma Separated Values*

**IOT:** *Internet of Things*

**API:** *Application programming interface*

**LED:** *Light Emitting Diode*

# ÍNDICE DEL CONTENIDO

# 1. INTRODUCCIÓN Y OBJETIVOS

## 1.1. INTRODUCCIÓN

En este trabajo de fin de grado se presenta el desarrollo general de un simulador 3d con físicas acuáticas para los robots *BlueROV2* [1], exponiendo un entorno sobre el cual los sensores del robot generen interacciones lo más realistas posibles. Se ha empleado un enfoque sensor-controlador-actuador que permita una experiencia lo más fiel posible, aplicando ciertas dinámicas que reflejen un entorno acuático**.** Además, se han implementado diferentes controladores que definen el comportamiento del robot ante diversas situaciones.

Este simulador es una parte de un proyecto mayor denominado NAUTILUS, creado por el ministerio de ciencia e innovación con el fin de explorar y desarrollar aplicaciones en los *BlueROV2* [1]. La idea surgió en base al gran auge que está teniendo la economía azul, junto al escaso desarrollo que tiene el ámbito robótico en los entornos submarinos. Se ha buscado crear el concepto de AUV para actividades coordinadas, es decir, robots que se puedan adaptar de forma autónoma al entorno acuático y realizar tareas definidas. A fin de llegar a este objetivo se ha segmentado el trabajo entre varias universidades españolas, donde, por ejemplo, la UNED facilitará los modelos físicos del robot en entornos acuáticos.

El retraso en la integración de robots dentro del ámbito acuático no es algo novedoso. Durante los últimos años, en los que cada día los robots están más integrados en nuestra sociedad, la mayoría de las tareas subacuáticas han seguido siendo realizadas de forma casi total por humanos. Principalmente se debe al riesgo que tiene el desarrollo de estos, donde un fallo durante un experimento en un entorno no controlado puede suponer la pérdida total del dispositivo. Pero también tiene que ver con el propio entorno acuático. Además de ser uno de los medios más complejos debido a las fuertes fuerzas acuáticas y a la poca exploración que ha tenido respecto a los demás entornos, también tiene unas aplicaciones menos directas para el ser humano respecto al medio terrestre o al medio aéreo. Sin embargo, eso no implica que no tenga gran utilidad, ya que cada vez cobra más peso la necesidad de limpiar los océanos, realizar tareas de búsqueda que para el ser humano son muy forzadas e incluso tareas de transporte o colectivas. Esto supone que, el principal obstáculo en el sector sea el alto riesgo que tiene su desarrollo, el cual puede ser mitigado con el uso de simuladores con alto grado de realismo. De esta forma podemos ser capaces de probar antes el funcionamiento y la adaptación del robot a ciertas tareas, sin poner en riesgo los mismos.

Esta necesidad es la que este TFG busca cubrir dentro del proyecto NAUTILUS. Para ello se ha desarrollado un simulador en el lenguaje de programación *Python* [2], apoyándose en los módulos *OpenGL* [3] y *pygame* [4] para la gestión visual del mismo. Dentro de esta versión inicial del simulador tendremos una aproximación del robot *BlueROV2* [1], que cuenta con tres tipos de sensores: un radar que permite hacer lecturas en modo barrido para ver la proximidad del robot a elementos del entorno, seis sensores de distancia (uno en cada eje) y un GPS, que nos indica la posición de este dentro del simulador. Finalmente, también se han implementado una serie de controladores a modo de experimentos para mostrar el comportamiento del robot ante ciertas situaciones y, de esta forma, poder demostrar el correcto funcionamiento del propio simulador. Estos experimentos van a dividirse en dos grupos: Algunos de ellos están basados en problemas muy específicos, como puede ser control de flotabilidad y de velocidad. Por otra parte, se ha propuesto un experimento final colectivo que ha requerido de los conocimientos adquiridos anteriormente, obteniendo un robot con la capacidad de navegar evitando obstáculos.

Por último, es importante mencionar que el simulador desarrollado para este documento es una versión inicial. El cual busca ser el fundamento de lo que eventualmente será el simulador de NAUTILUS, tal y como veremos en las líneas futuras expuestas en la Sec. 5.2.

## 1.2. OBJETIVOS

Para la gestión de objetivos del TFG vamos a segmentar los enfoques en dos campos, uno entorno al desarrollo del simulador y otro respecto al desarrollo de algoritmos en los diferentes experimentos.

En cuanto al primer ámbito cubrimos todo el desarrollo a nivel técnico del simulador, de forma que tengamos una visión clara de todos los niveles que se han cubierto. Esto supone una toma de decisión sobre el lenguaje que se va a utilizar, evaluando los principales puntos a favor y en contra de cada uno, además de seleccionar la mejor forma de renderizar objetos en 3d. En este caso se seleccionó *Python* [2] y *OpenGL* [3] junto a *pygame* [4] para la gestión del apartado gráfico. Una vez se tiene creado un entorno básico de visualización, debe tener la capacidad de implementar archivos externos que modelen objetos 3d, en este caso se han utilizado solo archivos *.obj*. Por otro lado, la gestión de la rotación de objetos por parte de *OpenGL* [3] es diferente a los estándares de robótica, lo que ha obligado a relacionar este modelo con los estándares *Roll, Pitch* y *Yaw*. Dentro del simulador 3d, se necesita tener la libertad de explorar el entorno por medio de una cámara. Es por ello que se ha implementado una cámara orbital con la capacidad de enfocar los diferentes robots dentro del simulador. En lo respectivo a la propia estructura del simulador, se ha decidido seguir los estándares sensor-controlador-actuador, ya que son los más fieles a la realidad, aislando completamente el controlador del simulador y solo dando acceso a los mismos que tendría un robot real. Por otro lado, en lo respectivo a las dinámicas acuáticas, se cuenta con modelos proporcionados por la UNED en *Matlab* [5], que han sido traducidos a *Python* [2] y adaptados al simulador para su utilización. Además, estas dinámicas se han separado del propio robot de forma que, en caso de querer incorporar más fuerzas externas, se puedan agregar de forma sencilla en un único cálculo. En lo respectivo a los sensores, ha sido necesario desarrollar la capacidad de ver los objetos del entorno desde dentro. Por ello se van a mapear los objetos mediante el uso de volúmenes de colisión, definidos por las superficies del objeto. Una vez creados estos volúmenes de colisión, se han podido implementar sensores GPS y de distancia e incluso, para tener un sensor más realista, se ha adaptado el sensor de distancia para funcionar como un radar, donde las lecturas se hacen a modo de barrido. Tras toda esta implementación, lo único que queda referente al entorno, es el desarrollo de mapas. En este caso, al tratarse de una versión inicial, simplemente se han incorporado mapas basados en piscinas con diferentes formas. Finalmente, para la comodidad del usuario final, es importante habilitar opciones que permitan la configuración rápida sin modificar código y exportar datos. Por ello, se ha implementado la capacidad de usar *flags* o ajustes en línea de ejecución y archivos de configuración que definan el estado inicial del entorno. Además de la capacidad de exportar datos sobre las sesiones realizadas en el simulador y así posteriormente analizarlos.

En lo referente al segundo ámbito, solo se estableció un objetivo, que va a contar con diferentes etapas antes de poder completarse. Este objetivo es el desarrollo de un algoritmo que permita a un robot moverse por el entorno esquivando obstáculos, lo que se denomina un experimento de *avoid obstacle*. Para poder lograr lo, tendrán que desarrollarse previamente ciertos experimentos secundarios, como la capacidad de mover al robot en cualquier eje de forma controlada, conseguir mantener una flotabilidad estable a lo largo de los experimentos y un algoritmo que controle la velocidad del robot, compensando automáticamente las inercias.

## 1.3. DISEÑO DEL DOCUMENTO

En lo correspondiente a la estructura del documento, se ha dedicado otro capítulo más a la introducción de la materia. El Cap. 2 que, desarrolla un estado del arte sobre el ámbito de los simuladores de robots y los diferentes enfoques que tienen hoy en día.

Durante el Cap. 3 se ha realizado una ligera introducción sobre las herramientas con las que se ha implementado el simulador, que son *Python* [2] y *OpenGL* [3], junto a una introducción matemática de los diferentes conceptos que se van a aplicar dentro de las diferentes etapas de la implementación. Posteriormente, se ha pasado a desarrollar los diferentes puntos del simulador, en los que se comienza con una visión general de la estructura del mismo y como se entiende su flujo de ejecución desde el punto de vista interno. Posteriormente se enumera cada apartado del simulador en detalle, recorriendo

desde las dinámicas, los volúmenes de colisión, mapas hasta los propios sensores y objetos que tiene el propio simulador.

A lo largo del Cap. 4 se ha tenido un enfoque más práctico del uso del simulador, tratando íntegramente el desarrollo de experimentos dentro del propio simulador centrándose en un experimento principal dedicado a generar un algoritmo que permita al robot navegar esquivando obstáculos del entorno. Dicho experimento a su vez ha desencadenado una serie de experimentos secundarios, como el control del movimiento del robot, la generación de un algoritmo que de una flotabilidad estable o uno que permita ajustar la velocidad para mantenerla constante a lo largo de la navegación. De forma que, según se han ido realizando los experimentos secundarios, se ha podido dar un mejor enfoque de experimento principal, hasta finalmente lograr el objetivo propuesto para este.

Una vez expuesto todo el apartado práctico del simulador, el Cap. 5 proporciona una serie de conclusiones que recopilan todos los puntos mencionados anteriormente, de forma que se dé una visión final de todo el proyecto. Finalmente, como última parte de este capítulo, también se han mencionan las posibles líneas futuras que tendrá el proyecto. Enfocando los puntos más importantes a desarrollar en el corto plazo y las posibilidades que este proyecto ofrece a futuro.

# 2. ESTADO DEL ARTE

Dentro del amplio abanico que ofrece el sector de los simuladores robóticos, hay principalmente dos enfoques: el primero de ellos tiene que ver con la simulación a la hora de crear un robot [6], permitiendo comprobar aspectos estructurales y dinámicos principalmente, y el segundo tipo son los simuladores cuya función es la recreación de un medio real en el que se puedan asegurar ciertos comportamientos del robot sin arriesgar su integridad. El simulador desarrollado en este TFG pertenece al grupo de simuladores que permiten evaluar comportamientos, y está centrado en el medio acuático. Se ha utilizado como lenguaje de programación principal *Python* [2] y se han gestionado los gráficos 3d mediante el uso de *OpenGL* [3]. Además, la estructura interna del simulador se basa en un enfoque sensor-controlador-actuador, asemejándose lo máximo a un modelo real y facilitando así el paso de los controladores del simulador a robots reales.

La mayoría de simuladores robóticos creados hace una década, están desarrollador en C++ como IRSIM [7], el cual está basado en una estructura sensor-controlador-actuador bastante parecida a la utilizada en este trabajo y una gestión de gráficos realizada por *OpenGL*. Se decidió usar este lenguaje de programación debido a su alta velocidad de ejecución, permitiendo así la ejecución de experimentos con un elevado grupo de robots. Sin embargo, esto también supone cierta pérdida de oportunidades en otros campos, ya que el desarrollo de algoritmos evolutivos que implementa en comparación al que podría tener con el uso de *Python* es mucho menor. Además de realizar en su mayoría un desarrollo desde cero de todos los elementos que incorpora.

En los simuladores robóticos implementados en Python, se encuentra la tendencia al uso de un módulo llamado *PyBullet* [8], el cual simplifica el desarrollo de ciertos comportamientos, además de tener un enfoque previo muy orientado a la robótica. Esto se puede apreciar en el simulador *Mereli* [9], el cual está desarrollado en *Python* y utiliza este módulo para la gestión de los gráficos y del entorno. Además, también sigue la estructura sensor-controlador-actuador que se ha mencionado. El uso del paquete permite implementar funciones más complejas de lo que se ha podido ver en otros simuladores, como pueden ser funciones de recogida de objetos, animaciones de alta complejidad y gestión de modelos URDF. También puede complementarse con otros módulos de *Python*, como puede ser *TensorFlow* [10], tal y como se puede ver en [11], donde se utiliza el entorno generado con *PyBullet* para la visualización y evaluación de objetivos y recompensas que deben tener los algoritmos de aprendizaje reforzado. Como se ha podido apreciar, la variedad y especialización de los módulos disponibles en *Python*, permiten explorar posibilidades muy complejas para otro tipo de lenguajes. El problema que surge con el uso de *PyBullet* es que no está preparado para el manejo de fluidos y simulaciones acuáticas, haciendo inviable su uso para el desarrollo del TFG, además de que a niveles de eficiencia hace inviable también el manejo de grandes cantidades de robots como si se pueden usar en IRSIM [7].

Por otro lado, se puede observar un claro desfase en el medio acuático respecto a los demás, pudiendo encontrarse entre los pocos ejemplos, el caso de Kelpie [12] que está desarrollado en ROS [13], lo cual permite una gran compatibilidad con los sistemas reales y un enfoque total en el ámbito robótico. Simula ciertas iteraciones con la superficie acuática y con el medio mediante el uso de un sonar, pero sin realmente interactuar de forma submarina como se busca en NAUTILUS, simplificando la poca expansión que hay sobre el medio acuático dentro de los simuladores robóticos.

En conclusión, se puede ver como la tendencia general es el uso de la estructura general de sensor-controlador-actuador (véase [7] [9]), ya que obliga al controlador a programarse de la misma forma que lo haría en un robot real. Por otra parte, en el ámbito del lenguaje, aunque no se pueda usar el módulo *PyBullet* [8] y se tenga que recurrir a *OpenGL* [3], sigue siendo junto a ROS [13] una de las alternativas más utilizadas hoy en día, además de la alta facilidad que proporciona para la gestión de redes neuronales. No obstante, no se debe olvidar las limitaciones que tiene en la velocidad de ejecución.

# 3. SIMULADOR

En este capítulo se va a desarrollar tanto un preámbulo teórico que cuenta con una introducción a los problemas matemáticos y como una introducción a *Python* [2] y *OpenGL* [3], como la estructura y funcionamiento del propio simulador, partiendo desde los niveles estructurales y flujos de ejecución, hasta finalmente los principales elementos del mismo. En lo referente a los problemas matemáticos se tratarán principalmente problemas vectoriales y de mapeado de superficies, ya que es lo que más difiere en *OpenGL* [3] respecto a los estándares de robótica más comunes. Por otro lado, en lo respectivo al simulador, esta versión inicial ya cuenta con la capacidad de ver a qué distancia se encuentra el robot de su entorno, mediante el uso de sensores permitiendo el desarrollo de algoritmia básica.

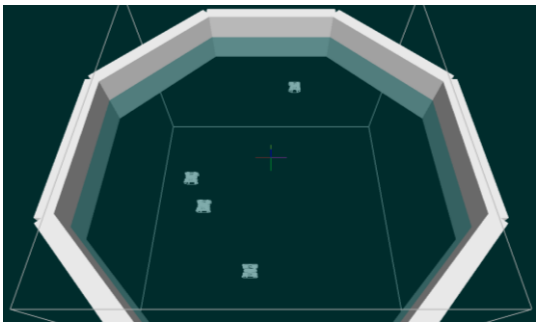

*Figura 1: Previsualización simulador*

## 3.1. PRELIMINARES MATEMÁTICOS

### 3.1.1. PITCH, YAW Y ROLL

El *Pitch*, *Yaw* y *Roll* son el estándar más utilizado para definir rotaciones en aeronaves y robots. Estos son tres vectores absolutos, uno por cada eje, sobre los cuales se realizan giros que alteran el estado del cuerpo sobre el que se realizan. Se puede ver más claro en el siguiente ejemplo:

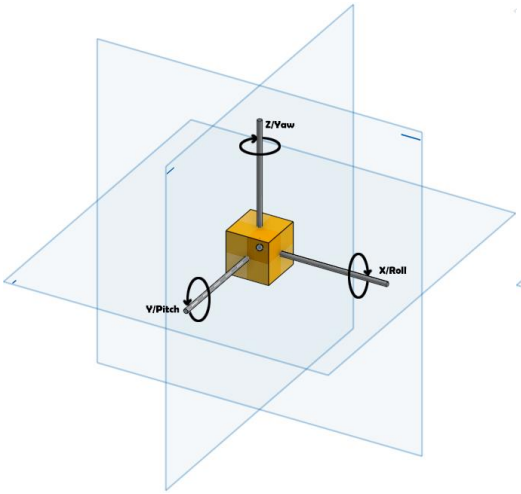


*Figura 2: Posición base Roll, Pitch y Yaw*

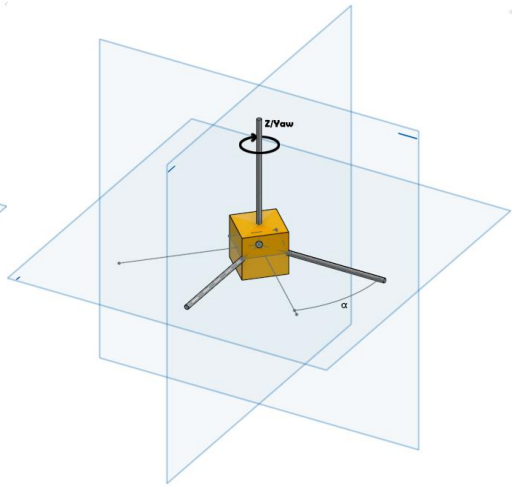


*Figura 3: Objeto con Yaw aplicado*

Finalmente, se debe tener una consideración en cuenta, como vemos en la *Figura 3*, se pueden diferenciar los ejes *x* e *y* rotados por la acción del *Yaw* y los ejes absolutos. Según los estándares que se han seguido, han utilizado los ejes absolutos, lo cual no necesariamente es lo que utilizan los diferentes módulos usados. Por ejemplo, en *OpenGL* [3] se emplean los ejes relativos por defecto,

mientras que para hacer los cálculos internos del simulador se pueden expresar directamente con los absolutos.

### 3.1.2. PROYECCIÓN DE VECTORES SOBRE BASES ROTADAS

En el caso de encontrar una herramienta que utilice los ejes relativos del *Pitch*, *Yaw* y *Roll* se tendrá que realizar una proyección de los vectores absolutos sobre los relativos para así conseguir el mismo efecto de rotación que tendríamos usando los absolutos. Esto va a suponer tres niveles:

- Para la primera rotación, podemos escoger cualquiera de los ejes. Se podrá realizar sobre el vector base absoluto $\overrightarrow{v_x} = (1, 0, 0)$, estableciendo como $\alpha$ el ángulo de la rotación.
- En la segunda rotación, los vectores $z$ e $y$ van a estar rotados $\alpha$ grados entorno a $x$. Por lo que se tiene que obtener el eje sobre el que realicemos la segunda rotación (expresado como combinación lineal de los rotados). En este caso, si se quisiera hacerlo sobre el eje $y$ sería de la siguiente forma: $\overrightarrow{v_y} = (0, \cos(\alpha), -\sin(\alpha))$, aplicando una rotación de $\varphi$ grados. Como se puede observar la coordenada en x se mantiene como 0 ya que este vector sigue siendo igual al absoluto.
- En la tercera rotación, todos los vectores tienen cierta modificación, por lo que la proyección del eje $z$ se va a expresar de la siguiente forma:

  $$\overrightarrow{v_z} = (-\sin(\varphi), sin(\alpha) \cdot \cos(\varphi), \cos(\alpha) \cdot \cos(\varphi))$$

  un ángulo $\theta$.

Como se ha podido observar, realmente lo que se ha hecho es redefinir los ejes de rotación de forma que se expresen sobre los vectores relativos con los que se trabaja. El resultado es igual al que obtendríamos en un sistema absoluto con los vectores base.

### 3.1.3. MAPEADO DE SUPERFICIES

A la hora de realizar cálculos internamente dentro del simulador, se ha necesitado una forma de expresar superficies para posteriormente calcular intersecciones con vectores. Es por ello que en esta versión inicial se supone que todos los elementos tienen volumen de colisión en forma de hexaedro. Para ello se han definido las superficies por medio de sus vértices, lo cual supone que estos puntos deben seguir las rotaciones y las variaciones de posición del propio objeto.

Para que los vértices hagan un correcto seguimiento de las rotaciones expresadas mediante el *Roll*, *Pitch* y *Yaw* se debe considerar que realmente son modificaciones de ángulo en coordenadas polares, modificando los planos según el eje de rotación. Se puede observar más claramente en el ejemplo de la *Figura 4*.

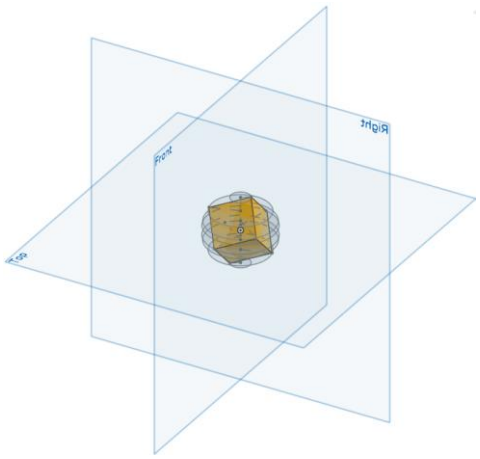

*Figura 4: Órbitas de los vértices al aplicar un Yaw*

En dicha figura, cada vértice del cubo mostrado se desplaza en su plano correspondiente el ángulo de rotación en polares. Y como las rotaciones se gestionan mediante conversiones a polares no tenemos los problemas de diferenciar bases absolutas y relativas que teníamos anteriormente con la gestión de *Roll*, *Pitch* y *Yaw*.

Finalmente, los cambios de posición se pueden implementar sencillamente desplazando los objetos entorno a su posición central. Esto supone que los vértices tienen una variación exactamente igual, de forma que se puede sumar directamente.

## 3.2. INTRODUCCIÓN PYTHON Y OPENGL

Python es un lenguaje de programación interpretado de alto nivel. Su filosofía principal se basa en una gran legibilidad del código y una gran versatilidad respecto a lenguajes más antiguos como C, C++ o Java. Además, cada vez cuenta con un mayor auge y popularidad por parte de la comunidad, haciendo que el mantenimiento sea constante y encontrando una gran variedad de módulos externos que pueden ser útiles. Es por ello por lo que se presenta como una buena opción para tener en cuenta. No obstante, si se busca de forma más específica a lo que nuestro proyecto necesita, encontrar proyectos similares que tengan que ver con simuladores robóticos 3d puede ser de gran ayuda. Dentro de estos proyectos similares hay un módulo bastante utilizado llamado *PyBullet* [8] el cual facilita multitud de operaciones para la creación de simuladores y su visualización. Sin embargo, este tiene una gestión muy complicada con los entornos acuáticos, por lo que no puede ser utilizado de forma directa. Pero si se puede utilizar como referencia e inspiración proyectos basados en este módulo como puede ser el caso de simulador *Mereli* [9], programado íntegramente en *Python* [2] y utilizando una estructura muy similar a la que buscamos. Por lo que finalmente, simplemente se necesita una buena gestión de un entorno visual 3d para poder desarrollar el proyecto en *Python* obteniendo así todos los beneficios mencionados anteriormente. Dentro de las múltiples herramientas que tiene *Python* para la gestión visual 3d, *OpenGL* [3] es de las más desacopladas y relevantes para simuladores robóticos. Esto es debido a que tiene múltiples versiones en diferentes lenguajes de programación y a su vez es de las más populares, por lo que será el utilizado para realizar esta tarea. Por último, una vez seleccionado el lenguaje de programación que se va a utilizar, es interesante hacer una reflexión sobre las limitaciones que puede tener. *Python*, como se ha visto, puede tener múltiples ventajas, pero tiene un claro impedimento en su velocidad de ejecución (siendo bastante más lenta que la otra posible alternativa C++). Aunque actualmente esto no supone un problema crítico, debe ser considerado para futuras versiones donde se busquen simulaciones con gran cantidad de robots, existiendo la posibilidad de traducir parte del código a otros lenguajes que proporcionen mejores tiempos de ejecución.

OpenGL [3] (*Open Graphics Library*) se puede definir como una API multilenguaje y multiplataforma, dedicada al manejo de gráficos 2d y 3d. Como se ha mencionado anteriormente, el hecho de que sea una API multilenguaje da una gran libertad para poder intercalar otros lenguajes más adelante. *OpenGL* proporciona una API con gran versatilidad, además de tener un largo desarrollo, mantenimiento y eficiencia. Pero tiene un problema: su curva de aprendizaje está bastante truncada, lo cual se debe a que el programador debe dictar los pasos exactos con los que realizar una escena, mientras que hoy en día la mayoría de API en este entorno son más descriptivas y de mayor nivel. Esto solamente se puede resolver mediante la prueba y error, junto al uso de documentación y desarrollando herramientas complementarias que permitan la automatización de ciertos procesos. Como hemos visto en el apartado de preámbulos matemáticos Sec. 3.1 se han tenido que estudiar ciertos métodos que van a tener que ser intercalados con el funcionamiento de *OpenGL* [3], para tener así un estándar más enfocado al ámbito robótico. El principal ejemplo de ello es que en *OpenGL* [3] se utilizan los vectores relativos para las rotaciones, tal y como hemos visto en el primer punto de los preliminares matemáticos Sec. 3.1.1. Por lo tanto, tendremos que usar las proyecciones explicadas en la Sec. 3.1.2 para realizar las rotaciones.

## 3.3. LÍNEA DE EJECUCIÓN

Tal y como se mencionó en los objetivos del Cap. 1, un concepto muy importante sobre el que se insistió desde el primer momento es la estructura sensor-controlador-actuador que debe tener el simulador. Esta estructura, además de estar bastante extendida y estandarizada, es la que hace una mejor representación de cómo funcionan realmente los robots en la vida real, es decir, completamente aislados del entorno, donde solo hay estímulos de entrada que se evalúan y generan una respuesta de salida. Ampliando este concepto al simulador entero, hay un paso extra se debe gestionar: la aplicación de las dinámicas. Estás dinámicas principalmente representan la acción del agua sobre el propio robot. No obstante, también se deben considerar posibles líneas futuras, en las que exista la necesidad de aplicar fuerzas externas como corrientes. Es por ello que se ha desacoplado toda la dinámica del robot, manejando el propio simulador en base al estado del robot y permitiendo así la gestión de más fuerzas si es necesario.

Finalmente, con este planteamiento, el flujo que sigue el programa a grandes rasgos es el siguiente. Cada robot tiene incorporado un controlador que define su funcionamiento, este controlador solo puede obtener datos mediante la lectura de los sensores con los que cuenta el robot y solo puede generar una respuesta mediante los actuadores que controlan los motores del robot. Posteriormente, el simulador evalúa el estado en el que se encuentra el robot y aplica las dinámicas correspondientes. Este proceso se repite con cada iteración del simulador, con un periodo de 10ms (o lo que es lo mismo una tasa de refresco de 100Hz).

## 3.4. ESTRUCTURA DEL SIMULADOR

A continuación, se exponen los elementos de la estructura del simulador que serán detallados en los siguientes apartados. La representación se muestra en la *Figura 5*.

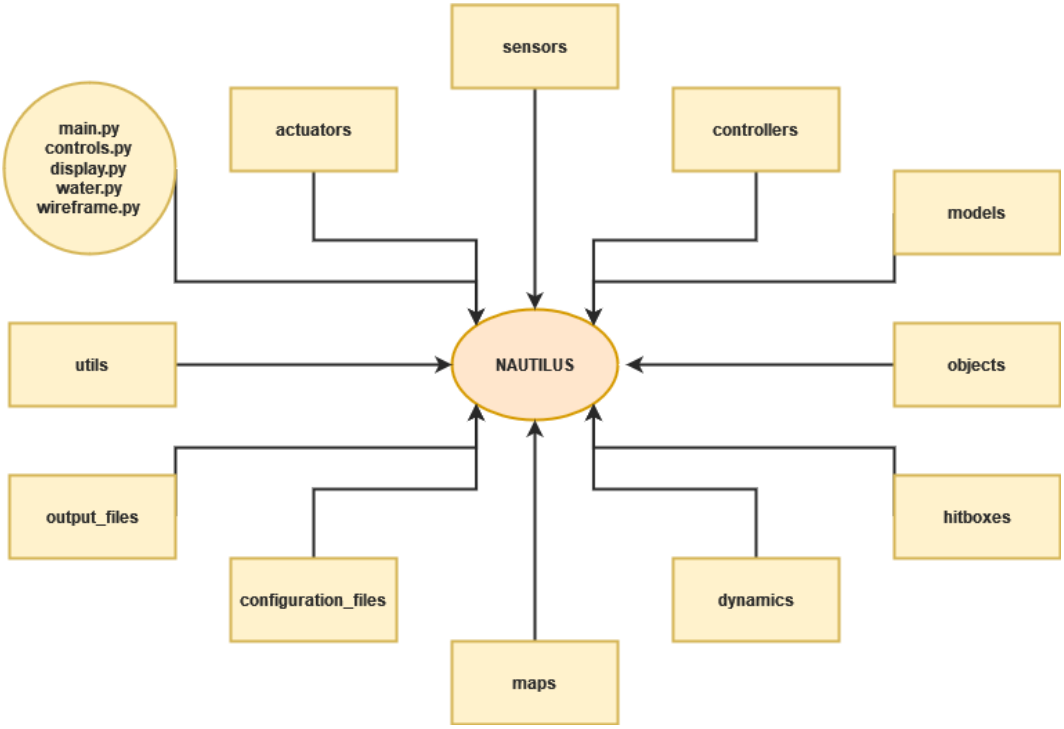


*Figura 5: Estructura donde los rectángulos son carpetas y el circulo archivos*

Dentro de las carpetas *actuators*, *sensors* y *controllers* se encuentran los archivos asociados a la estructura explicada en el punto anterior. Los actuadores son el medio que permite a los controladores cambiar el estado del robot y donde los sensores permiten obtener información específica del entorno, como veremos más en detalle en la Sec. 3.9.

En cuanto a los *models, objetcts, hitboxes y dynamics* se encuentran el grupo que modela los objetos mostrados en el simulador. La carpeta *models* almacena los propios modelos 3d que tendrán asociados los objetos para ser renderizados. Las *hitboxes* modelan dentro del simulador el objeto para que pueda ser evaluado por los sensores. Las *dynamics*, como ya se ha mencionado, son el modelado de las fuerzas aplicadas a los objetos, que necesitan ser asociadas al objeto para posteriormente ser evaluadas por el simulador. Finalmente, *objects* se encarga de almacenar toda esta información en objetos independientes según cada entidad del simulador.

Por otro lado, *maps* está destinada a almacenar los diferentes entornos que se vayan creando. En esta versión inicial hay dos mapas creados, como veremos más adelante. Por último, *configuration_files* y *output_files* son carpetas en las que su propio nombre indican el contenido. *configuration_files* almacena todos los archivos de configuración que podemos utilizar al arrancar el simulador y *output_files* almacena los archivos de salida generados por el simulador.

Tras ver las carpetas, se encuentran una serie de archivos externos alojados en la carpeta principal del simulador:

- *main.py:* Es el archivo principal que se debe de ejecutar para arrancar el simulador. Se encarga de configurar todo el simulador de forma inicial, de la gestión del hilo de ejecución y evaluar posibles *flags* de entrada para modificar algunos comportamientos como la generación de archivos de salida, cambios de mapa…
- *controls.py*: Este archivo se encarga de la gestión de los controles principales asociados al simulador. Tanto de lo relacionado con los controles de la cámara, como lo relacionado con los controles más específicos, como parar el simulador, mostrar sensores…
- *display.py:* Se encarga de la configuración inicial del *display* donde se va a mostrar el simulador. Este archivo tiene más relevancia de la que puede parecer, ya que su función es la de establecer resoluciones, luces y sombras, transparencia y capacidad de cambiar el tamaño de la ventana.
- *water.py* y *wireframe.py:* Son dos archivos encargados de renderizar elementos que siempre van a ser mostrados en el simulador. Si se configura el simulador para no mostrar ningún robot, estos dos elementos son los únicos que van a ser dibujados. El primero es un plano situado en $z = 0$ y el segundo son las aristas de un cubo con unas dimensiones de 10x10x10m y los ejes de la base central, para así tener una orientación base dentro del entorno.

Como se ha podido observar, el simulador está altamente segmentado. Al tratarse de un proyecto con tal calibre es importante que las zonas del código estén claramente aisladas y así en caso de introducir nuevo código poder encontrar errores de forma rápida y clara. Además, como es probable que nuevos desarrolladores se vayan incorporando al proyecto también ayuda a tener una visión más clara de todo de forma muy rápida.

## 3.5. DINÁMICAS

Dentro de los objetos del simulador se diferencian dos grupos. Un primer grupo que va a estar bajo los efectos del agua y otros que se van a mantener estáticos. Es por ello que en esta versión inicial se va a contar con el *modelo0* presentado por la UNED para simular las interacciones con el agua de los *BlueRov2* y una dinámica *static* que se aplicará a objetos del segundo grupo (objetos estáticos). Este *modelo0*, es un modelo dinámico que va a evaluar el estado inicial del robot, teniendo en cuenta velocidades lineales, angulares, estado de los motores, posición y rotación, para generar a su salida un nuevo estado, que haya simulado los efectos del medio sobre el robot, valorando parámetros como la inercia, el momento, la presión del agua y la flotabilidad del robot entre otros parámetros. Con cada modelo que se vaya presentando, la cantidad de parámetros que se van a tener en cuenta para evaluar el estado será mayor y con ello el funcionamiento final más realista.

Estas dinámicas están estructuradas dentro del simulador de forma que tienen como entrada el estado del objeto al que deben aplicarse y los controles de este mismo, lo que sería en el caso del robot los motores. Por otra parte, se genera a la salida el nuevo estado que va a tener esta entidad. Si usamos como ejemplo el modelo *static* se puede ver claramente que su función es devolver el mismo estado de forma que no altere nada al objeto. Esta estructura se ha definido de forma tan rígida para poder

realizar cambios de dinámicas rápidamente, ya que la UNED irá modificando su modelo y este debe actualizarse para ser cada vez más realista.

## 3.6. HITBOXES

Desde el punto de vista interno se debe ser capaz de *"ver"* los objetos, para ello se va a realizar un mapeo de las superficies que conforman dichos objetos. En esta versión inicial daremos por hecho que todos los objetos son tetraedros a ojos del simulador. Por lo tanto como se vio en la Sec. 3.1 sobre los antecedentes teóricos, es suficiente con definir los vértices que conforman cada superficie para determinar el objeto entero mediante la combinación de las superficies.

Gracias a la suposición de que todos los objetos tienen la misma estructura base, se va a poder generar todos los volúmenes de colisión o *hitboxes* partiendo de un cubo unitario, como se puede observar en la *Figura 6.*

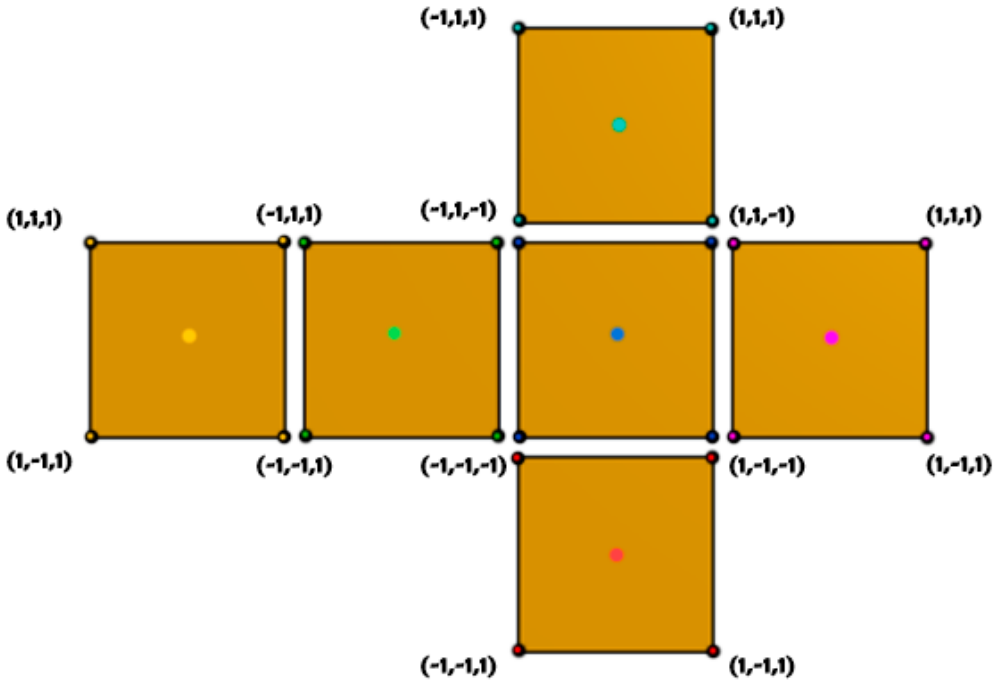


*Figura 6: Representación del hitbox unitario base del que parten todos los modelos*

$$basicSurface = \left[\begin{pmatrix} (1,1,1) \\ (1,1,-1) \\ (1,-1,-1) \\ (1,-1,1) \end{pmatrix}, \begin{pmatrix} (-1,1,1) \\ (-1,1,-1) \\ (-1,-1,-1) \\ (-1,-1,1) \end{pmatrix}, \begin{pmatrix} (1,1,1) \\ (-1,1,1) \\ (-1,1,-1) \\ (1,1,-1) \end{pmatrix}, \begin{pmatrix} (1,-1,1) \\ (-1,-1,1) \\ (-1,-1,-1) \\ (1,-1,-1) \end{pmatrix}, \begin{pmatrix} (1,1,1) \\ (-1,1,1) \\ (-1,-1,1) \\ (1,-1,1) \end{pmatrix}, \begin{pmatrix} (1,1,-1) \\ (-1,1,-1) \\ (-1,-1,-1) \\ (1,-1,-1) \end{pmatrix}\right]$$

Simplemente se tiene que dimensionar cada eje en función al tamaño del objeto y aplicar la rotación y posicionamiento, tal y como se vio en el preámbulo matemático dentro del mapeado de superficies. Respecto al apartado más técnico, la gestión de las *hitboxes* dentro del código se realiza de forma generativa, es decir, cada vez que se quiere conocer la *hitbox* de un objeto se crea y almacena en un generador de *Python* por motivos de eficiencia. Esto es debido a que se supone que las peticiones por iteración del simulador no van a ser excesivamente elevadas. En las *Figura 7 y Figura 8* se puede ver como se crea el generador que modela la *hitbox* y el resultado final aplicado al modelo 3d del robot.

```
return (
    tuple(
        Point(
            *tuple(c * s / 2 for c, s in zip(vertex, self.size)),
            *self.rotation,
            self.position
        ) for vertex in surface
    ) for surface in self.baseSurface
)
```

*Figura 7: Generador de hiboxes utilizado en el simulador*

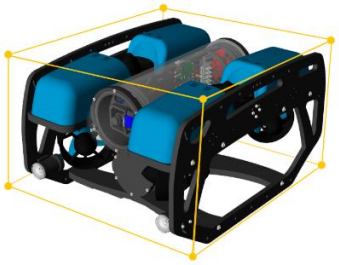

*Figura 8: Hitbox final sobre el robot*

## 3.7. ENTIDADES

Las entidades dentro del simulador se encargan de agrupar toda la información que se ha ido mencionando anteriormente dinámicas, *hitboxes* y modelos 3d, de forma que puedan realizar correctamente sus funciones asociadas. Para ello se ha definido una clase *Object*, la cual va a ser la clase base de la que heredan todas las entidades que va a gestionar el simulador. Esto permite definir una serie de atributos y métodos estandarizados para todos estos objetos. Estos atributos básicos van a ser la posición ($[x, y, z]$), la rotación ($[roll, pitch, yaw]$), un atributo *focusable* que indica si el objeto puede ser seguido por una cámara y otro atributo *showMoreInfo* que indica si se muestra la información extra asociada al objeto. En cuanto a los métodos que tienen por defecto, deben tener definidos un método *step* que gestione si el objeto debe sufrir alguna modificación, y un método *render*, que indica como dibujar el objeto dentro del simulador.

Este tipo de objetos son los más primitivos y dentro de ellos solo va a heredar de forma directa el objeto que funciona como cámara. Esta entidad va a ser a su vez la más diferente respecto a todas las demás ya que es la ventana entre el usuario y el simulador. Es por ello que además de programar propiamente el objeto, hay que definir como se quiere permitir al usuario mover la cámara y visualizar el entorno. En nuestro caso se ha decidido utilizar un rango de movimiento esférico/orbital, es decir, se podrá orbitar entorno a un punto al que la cámara siempre estará mirando. Se puede observar gráficamente en la *Figura 9*.

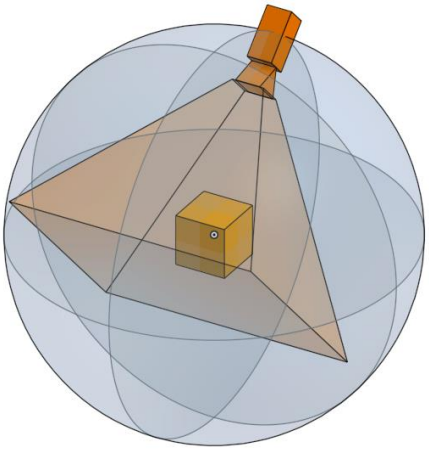

*Figura 9: Órbita y campo de visión cámara*

La pendiente que se genera en la pirámide que indica el campo de visión se denomina *pov* y es un parámetro que depende de la configuración realizada en el módulo *display.py*. La posición que tiene la cámara en la órbita depende de la rotación del objeto y el radio de la órbita lo definiremos como el *zoom* de la cámara. Finalmente, como última funcionabilidad diferente, agregaremos la opción de que la cámara pueda hacer que ese punto central sobre el que orbita puedan ser los diferentes objetos *focusable* del entorno.

Con todas las funciones del objeto definidas se podrá pasar a la caracterización de la clase *Camara*, que sigue la estructura mostrada en la *Figura 10.*

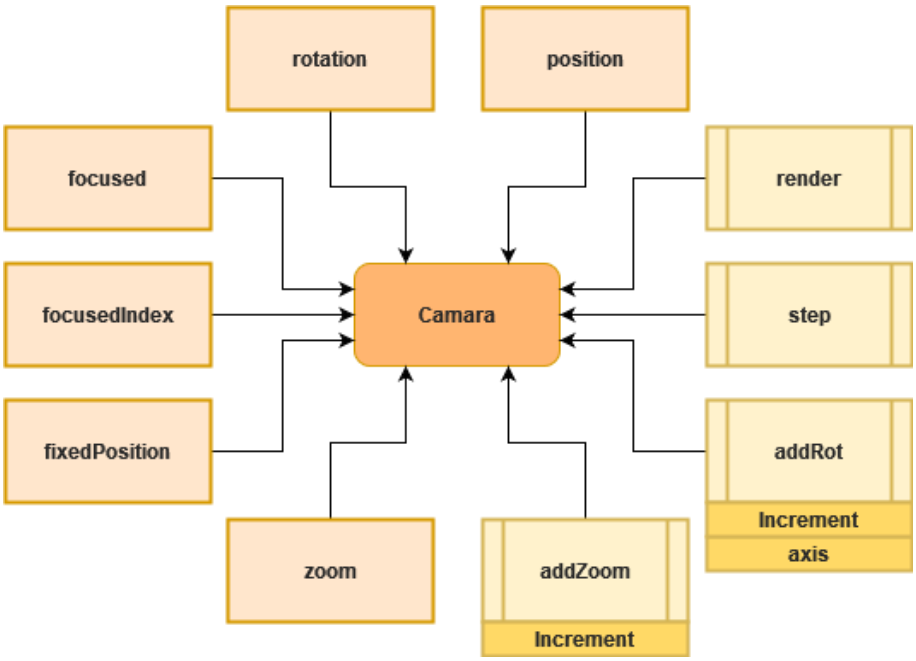


*Figura 10: Estructura objeto Camara*

Tal y como se encuentra representado, existen una serie de parámetros que permiten realizar las funciones que hemos definido anteriormente. Los métodos *addZoom y addRot* que nos permiten hacer las modificaciones pertinentes al estado. Por otro lado, en el caso de la cámara, el método *step* se encarga de, en caso de estar en modo seguimiento, actualizar el punto central a la nueva posición. El método *render* simplemente sitúa la cámara en la posición correspondiente, con el ángulo adecuado para enfocar al objeto. Algo a tener en cuenta es que la posición de la cámara, a diferencia del resto de objetos, se calcula en base a los ángulos, por lo que no se modifica de forma directa.

### 3.7.1. ENTIDADES DINÁMICAS

Dentro de las entidades que manejará el simulador, se encuentra un grupo que engloba lo que se podría denominar objetos físicos, los *DynamicObject*. Estos van a tener asociada una dinámica, un tamaño para definir la *hitbox* y un modelo 3d que será renderizado en el entorno dada su posición y rotación. Además, integraran nuevos métodos sumados a los que ya heredan de la clase *Objeto*, definiéndose *setState* que permite a la dinámica actualizar el estado del objeto una vez aplicada, *export* para manejar los datos reflejados en la carpeta *output_files* y una propiedad *state* que retorna el estado del objeto. Gracias a esta estructura se pueden enfocar los objetos manejados por el simulador con una estructura más parecida a la que ya mencionábamos anteriormente.

Partiendo como padre de este objeto encontramos dos objetos: las paredes y los robots. Dada la complejidad de los segundos primero explicaremos las paredes. Estas tienen una estructura base similar y nos permitirán ver el concepto de aplicación y uso de los *DynamicObject*. La pared es el objeto usado para definir mapas dentro del simulador. Se trata de un objeto estático que no debe verse afectado por las dinámicas acuáticas y que tiene como tamaño $(10m, 4m, 1m)$. En cuanto a su estado, este objeto tiene el estado más simple, formado por la posición y la rotación propia y al tratarse de un objeto fijo en el entorno no maneja ningún tipo de dato de salida y no tiene ninguna modificación en el método *step*. En lo respectivo a la información adicional que pueden mostrar este tipo de objetos se encuentran los vértices que forman la *hitbox*. Podemos ver esta estructura de forma más representativa en la *Figura 11*.

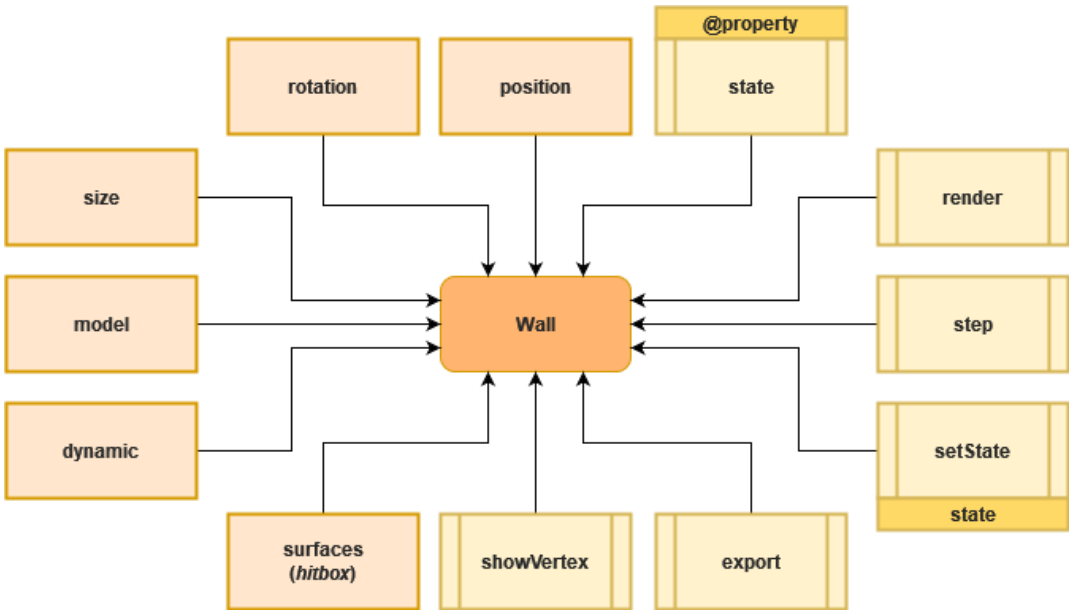


*Figura 11: Estructura objeto Wall*

Como se puede ver, realmente sigue la estructura básica de un *DynamicObject* de tipo estático. Finalmente, también es interesante mencionar el constructor de la propia clase, como podemos ver en la *Figura 12*.

```
class Wall(DynamicObject):
    def __init__(
            self,
            position,
            rotation=[90.0, 0.0, 0.0],
            size=(10, 4, 1),
            model="models/entities/wall/wall.obj",
            dynamic=static):

        def labmdaDynamic(): return dynamic(self.state)
        super().__init__(position, rotation, size, model, labmdaDynamic)
        self.surfaces = Thetrahedron(
            self.position, self.rotation, self.size
        )
# ...
```

*Figura 12: constructor objeto Wall*

Lo más destacable de la misma es la gestión de las superficies, es decir, la *hitbox*. Esta será actualizada cada vez que se realice un cambio de estado, y como el objeto pared es estático queda definida permanentemente. Cada vez se necesite la información almacenada, simplemente se tendrá que generar las superficies tal y como se ha visto anteriormente.

Entrando en lo referente al robot, vamos a seguir una estructura parecida a la usada con la pared, pero teniendo un objeto con dinámicas asociadas al *modelo0*. Que un objeto tenga asociada esta dinámica implica que su estado va a tener una serie de elementos más específicos, estos son: posición, rotación, velocidad lineal, velocidad angular y estado de los motores, lo cual se agrega a los requisitos que debe tener la clase por heredar de *DynamicObject*. A su vez, tal y como se ha explicado en la introducción al simulador, el robot sigue la estructura sensor-controlador-actuador, esto implica que va a tener un diccionario de sensores asociado, otro para los actuadores y un controlador único que defina su comportamiento, pudiendo cumplir así el flujo requerido. Por otro lado, al estar desacoplada la dinámica del robot, esta va a relacionarse con el actuador por medio de una variable llamada control que corresponde a la fuerza que aplican los motores en ese momento. En lo referente a los archivos de salida que se crearan para exportar el estado del robot, va a ser necesario poder diferenciarlos entre sí, es por ello que se le va a asignar un id. En base a esta serie de requisitos previos podemos definir una estructura entorno a la clase que modela el robot dentro del simulador. Esta clase se puede observar más en detalle en la siguiente *Figura 13*.

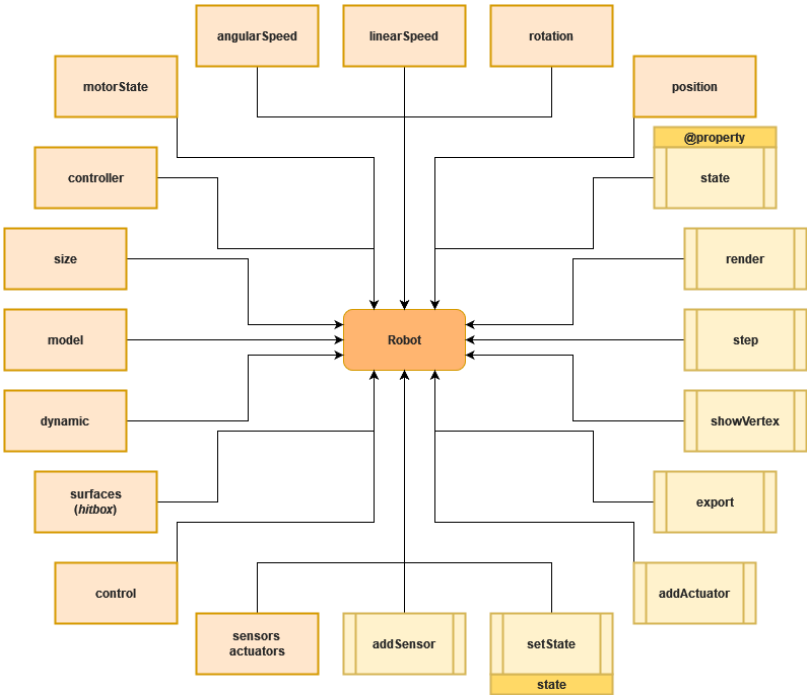


*Figura 13: Estructura objeto robot*

Dentro del gráfico se han omitido el *Id* del robot y una serie de configuraciones propias para los sensores que se realizan por defecto, como el *threshold* de los sensores de distancia que se define como 5m o el *enviroment* para que los sensores de distancia puedan hacer una evaluación del entorno.

En lo referente al flujo del programa, como ya se ha mencionado en repetidas ocasiones, la estructura sensor-controlador-actuador implica que el robot únicamente tiene contacto con su controlador, el cual se encarga de hacer las lecturas correspondientes de los sensores y de aplicar una respuesta dentro de los actuadores. Es por ello que el método *step* del robot únicamente realiza una iteración a su propio controlador. En caso de que el controlador quiera generar una salida modificando el control de los motores, lo hará por medio del actuador correspondiente al control, $self.actuators["control"]$.

Finalmente, dentro de la estructura de sensores con los que cuenta el robot en esta versión inicial se pueden a encontrar los siguientes campos:

- $self.sensors["\text{gps}"] \rightarrow [x, y, z]$: Un sensor GPS que permite a los controladores acceder a la posición global del robot dentro del entorno.
- $self.sensors["\text{distancia}"] \rightarrow [d_x, d_{-x}, d_y, d_{-y}, d_z, d_{-z}]$ : Un grupo de sensores de distancia situados en los ejes base y en ambos sentidos, para obtener una estimación de la distancia a la que se encuentran posibles obstaculos en las próximidades y en todas las direcciones.
- $self.sensors["\text{radar}"] \rightarrow [d_{+x}, d_{+x+y}, d_{+y}, d_{-x+y}, d_{-x}, d_{-x-y}, d_{-y}, d_{+x-y}]$: Un sensor radar que devuelve 8 lecturas de distancia en el plano $x$-$y$, con una velocidad de lectura de 1 lectura/tick.

Dentro del apartado de sensores se presenta más en detalle el funcionamiento de cada uno y los parámetros de configuración con los que cuentan.

## 3.8. MAPAS

Dentro de los diferentes elementos básicos que necesita el simulador, la creación de un entorno que permita llevar al límite las capacidades del mismo y a su vez experimentar las interacciones con los diferentes comportamientos tiene una gran relevancia**.** Para ello, se van a desarrollar diferentes mapas, permitiendo encontrar posibles fallos y explorar las interacciones de los sensores, además de dar versatilidad al desarrollo de controladores. Todo esto también debe adaptarse a la versión inicial en la

que se encuentra el simulador, teniendo claro que el objetivo final será la capacidad de incorporar entornos realistas de fondos marinos y que el simulador genere una interacción perfecta con el mismo. Sin embargo, de momento buscamos que los robots exploren dentro de diversas *piscinas* con diferentes perímetros que nos permitan explorar los puntos mencionados anteriormente. Con esta serie de requisitos se ha decidido crear dos tipos de arena: La primera una rectangular básica y la segunda un octógono que obligue a manejar interacciones con elementos diagonales. La profundidad de las *piscinas* se va a determinar como irrelevante, ya que en la mayoría de los casos trabajaremos con robots que tengan un comportamiento estable en el eje $z$.

Entorno a la primera piscina, la rectangular, se puede ver en las siguientes figuras tanto la función que genera su estructura, como un plano de la misma:

```python
def prismaArena():
    mapObjects = []
    alignOptions = {-1, 1}

    for level in range(0, 5):
        for mainAxisAlign in alignOptions:
            for secundaryAxisAlign in alignOptions:
                mapObjects.append(Wall(
                    position=[
                        mainAxisAlign * 5.0,
                        secundaryAxisAlign * 9.0,
                        -2.0 * level],
                ))
                mapObjects.append(Wall(
                    position=[
                        mainAxisAlign * 9.0,
                        secundaryAxisAlign * 5.0,
                        -2.0 * level],
                    rotation=[90, 0, 90]
                ))

    return mapObject
```

*Figura 14: Generador mapa rectángulo*

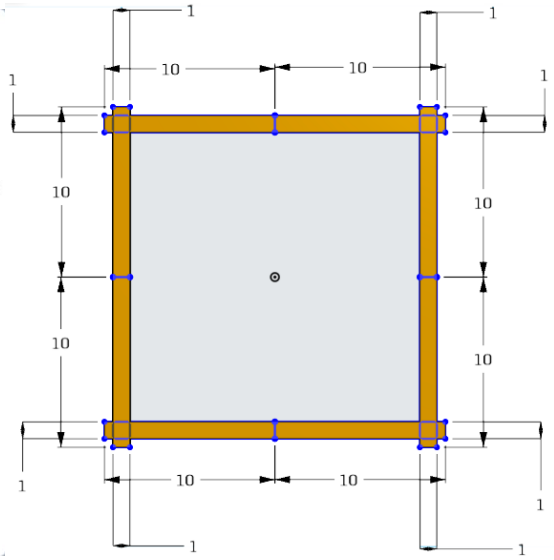


*Figura 15: Plano mapa rectángulo*

Este, aun siendo el más simple de los dos mapas, va a plantear cuestiones interesantes dentro del propio simulador. No tanto dentro de la interacción con sensores, sino en los comportamientos que se generen con los controladores. Los vértices de este mapa al ser un cuadrado se encuentran a 90°, esto en cuanto los robots no sigan trayectorias perpendiculares puede generar puntos donde el robot interprete que tiene más espacio del que realmente hay y deba realizar adaptaciones bastante rápidas. Veremos esto más en detalle en los propios experimentos dentro del Cap. 4.

Respecto al mapa octogonal, se observa en las siguientes figuras la función generadora del mapa y el plano del mismo:

```python
def cylinderArena():
    mapObjects = []

    for level in range(0, 1):
        for step in range(8):
            mapObjects.append(Wall(
                position=Point(
                    *[0.0, 12.0, -2.0 * level],
                    *[0, 0, 45 * step]
                ).coordinates,
                rotation=[90, 0, 45 * step]
            ))

    return mapObjects
```

*Figura 16: Generador mapa octógono*

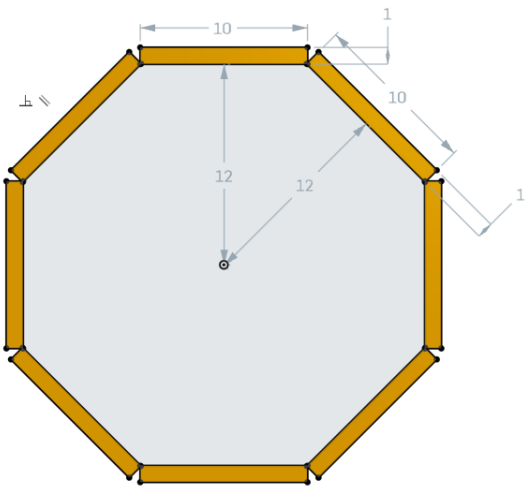


*Figura 17: Plano mapa octógono*

A diferencia del primer mapa, en este caso es más claro que el nivel de exigencia y el número de situaciones que pueden ser complejas de manejar son mayores. Por parte de los sensores, el número de interacciones y los ángulos de las paredes pueden generar, errores en los cálculos de distancia si no están bien desarrollados. Y por parte del comportamiento del robot, tener estímulos diagonales obliga a que se desarrolle una mayor capacidad de adaptación en el mismo, además de generar trayectorias más complejas. Todo ello hace que sea un mapa perfecto para los experimentos que cuentan con múltiples robots y evitadores de obstáculos.

Por último, aunque no se ha mencionado anteriormente existe un tercer mapa que se encuentra vacío. Se ha creado un mapa vacío para poder seleccionarlo sin modificar el flujo del programa principal del simulador. Este se encuentra enfocado para experimentos más centrados en trayectorias y pruebas de movimiento.

## 3.9. CONTROLADORES

Esta sección se centra en la estructura y funcionamiento de los controladores, sin entrar en ejemplos complejos ni en cómo funciona el desarrollo dentro de los mismos, ya que eso se desarrollará en el Cap. 4.

Todos los controladores parten de una clase padre llamada *Controller*, que establece los métodos y parámetros básicos que deben tener todos los hijos. El parámetro principal es el *controllerOwner*, que permitirá acceder al robot que este equipado por ese controlador, y así acceder a sus sensores y actuadores. Por otro lado, también define el método *step*, que gestiona cada ejecución del programa del controlador, y el método *export* que se combina con el método *export* que contiene cada robot. Esto es debido a que la lectura de los sensores se realiza desde el propio controlador. Además, es interesante mencionar la capacidad de exportar datos generados dentro del mismo, por lo que se puede devolver un diccionario que será agregado al *export* del robot. Se puede observar la estructura y un ejemplo de *export* en la *Figura 18.*

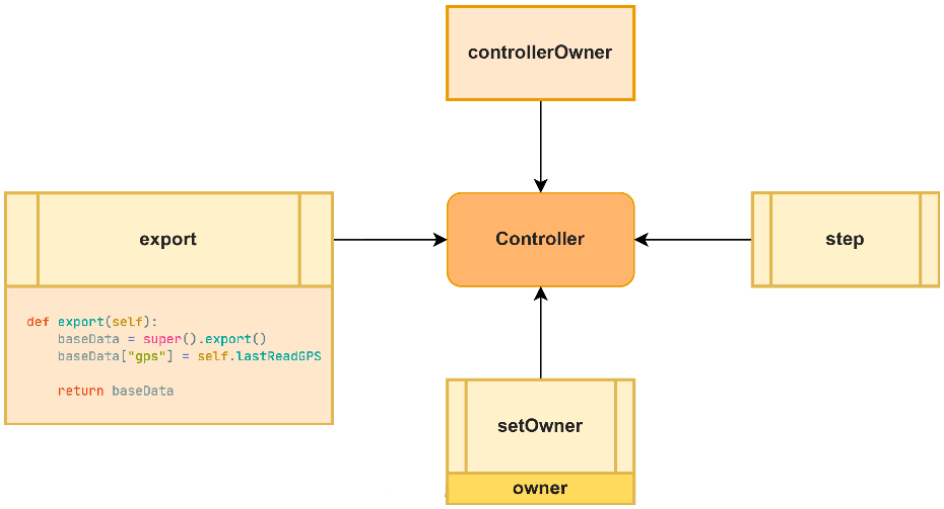


*Figura 18: Estructura de controlador*

El método *setOwner* no se ha sido mencionado anteriormente porque su única función es asignar el *controllerOwner*.

## 3.10. ACTUADORES

Los actuadores son el elemento más simple, cuya única función es modificar el estado del robot al que pertenecen. En nuestro caso el robot simplemente cuenta con actuadores de control que permiten modificar la fuerza de los motores.

La clase padre *Actuator* simplemente incorpora un atributo *actuatorOwner* que permite al actuador modificar el parámetro del robot que le corresponda y un método *apply* al que se le pasa como argumento el valor que se le quiera asignar.

Como se puede ver, la función de los actuadores no es tan fundamental dentro del simulador como lo es fuera de este. Como ya se ha mencionado, la filosofía sensor-controlador-actuador permite aislar el controlador del simulador haciendo más sencilla su extracción y traspaso a un robot real. De esta forma, en vez de modificar el parámetro vía *controllerOwner* dentro del controlador, se accede al actuador correspondiente y él lo modifica como ocurriría en un robot real.

## 3.11. SENSORES

En el apartado de sensores dentro del simulador, al igual que en la mayoría de los otros módulos, se va a encontrar un objeto principal *Sensor*, que va a definir ciertas características que estos objetos van a tener. En este caso, el atributo principal que va a requerir todo sensor es *sensorOwner*, de forma que en caso de necesitarlo pueda acceder a los datos del robot que posee ese sensor. Por otro lado, define un método *read* al que llamamos para obtener una lectura del sensor, un método *render* que permite visualizar información extra en algunos sensores si se activa mediante el parámetro *showMoreInfo*. Quedando la estructura general de los sensores de la siguiente manera:

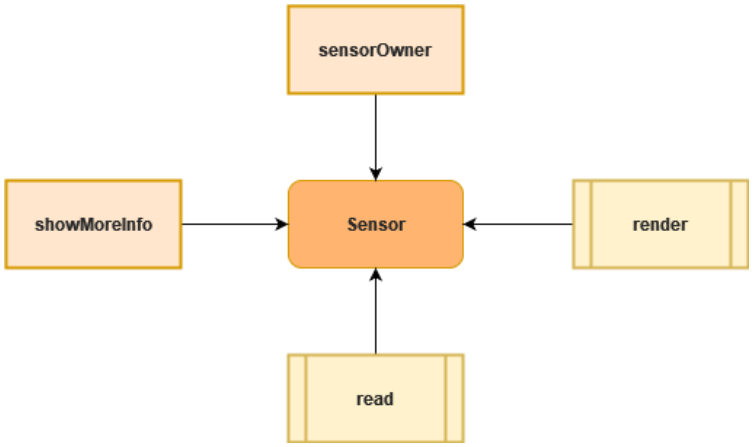


*Figura 19: Estructura de Sensor*

### 3.11.1. SENSOR GPS

El GPS (*Global Positioning System*) es un dispositivo que nos permite obtener información de la ubicación en la que se encuentra un objeto con un margen de error. Se utiliza sobre todo para tener la capacidad de viajar de un punto especifico a otro generando rutas y situar la posición dentro de un mapa. En este caso, dentro del simulador tendrá una función bastante similar, donde sus características principales van a ser situar al robot dentro del mapa y así posteriormente evaluar su trayectoria y comportamiento y que el robot tenga la capacidad de ir de un punto a otro o situarse en un punto concreto de cualquiera de los ejes. Algo que se debe tener en cuenta es que el robot ya cuenta con la posición global que tiene dentro del simulador y, de hecho, esta va a ser la que utilizaremos para simular el funcionamiento del GPS. Pero como ya se ha comentado, el robot no puede acceder a este tipo de variables globales ya que son parte del propio simulador y en un entorno real no se tendría acceso a las mismas. Por lo que el sensor de GPS proporciona acceso a esta variable global. Se sigue el flujo en el que el robot únicamente puede obtener estímulos recibidos de los sensores y queda completamente aislado del simulador.

Finalmente, en la *Figura 20* se puede ver una trayectoria genérica del robot viendo las diferentes lecturas del GPS.

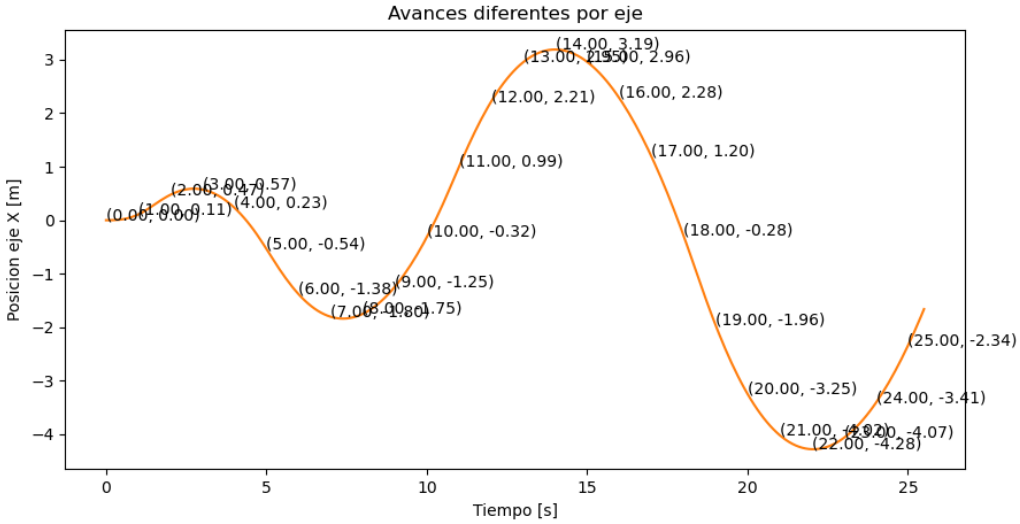


*Figura 20: Trayectoria con GPS*

### 3.11.2. SENSOR DISTANCIA

Dentro de los sensores disponibles, el desarrollo y la implementación del sensor de distancia supone un antes y un después en las capacidades del simulador. Una vez se ha implementado un sensor que permita al robot relacionarse directamente con el entorno, la complejidad de los controladores y los funcionamientos que realizan puede incrementarse de forma directa. Sin embargo, la implementación de los mismos también supone uno de los puntos más complejos en el propio simulador.

El sensor de distancia se establece como un vector unitario que define la dirección y el sentido de máxima sensibilidad del sensor. Además, se fija un umbral o *threshold* que definirá su módulo. De esta forma, cuando el vector atraviese una superficie del entorno dará un valor entre 0-1, siendo 0 que no hay intersección y 1 que hay colisión, que reflejará cómo de cerca se encuentra el robot del obstáculo. Para poder hacer estos cálculos, el sensor necesita tener acceso a un objeto *environment* que contenga todas las entidades del entorno. Por último, en el caso de los sensores de distancia, el parámetro *showMoreInfo* va a realizar una representación gráfica dentro del simulador del vector final de distancia que tiene el propio sensor como se puede ver en la *Figura 21*.

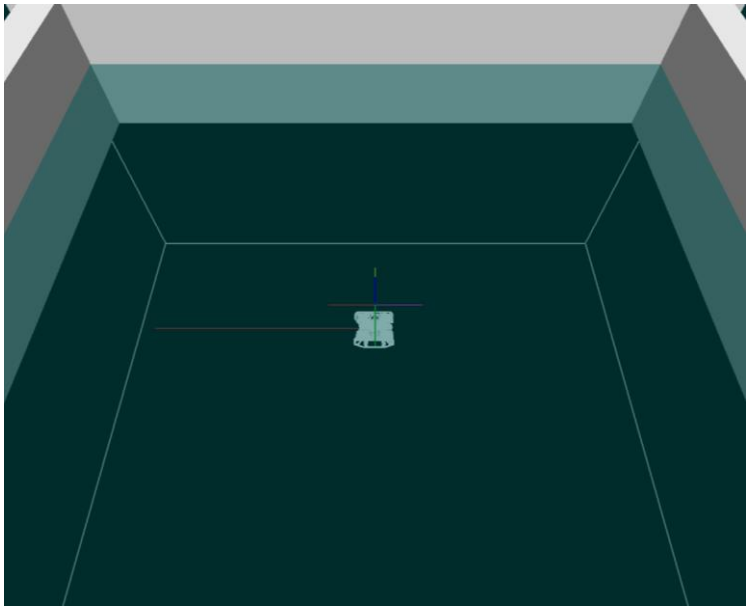

*Figura 21: Robot mostrando información del sensor de distancia*

Desde un punto de vista teórico, para ver si el vector tiene intersecciones con algún objeto del entorno, vamos a recurrir a las *hitboxes* que tiene asociada cada objeto. Se evalúa individualmente cada superficie de la *hitbox*, dando como lectura el mínimo valor de distancia, para posteriormente normalizarlo con la siguiente formula $nValue = \frac{threshold - value}{threshold}$. En las siguientes figuras se puede ver el proceso de forma más gráfica:

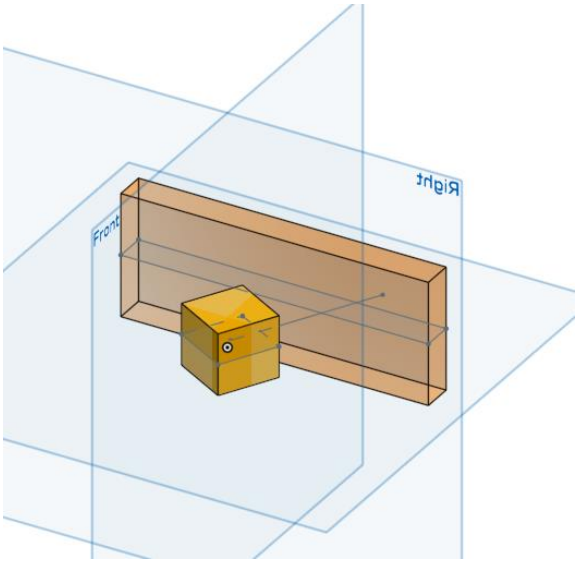


*Figura 22: Escenario base sensor distancia*

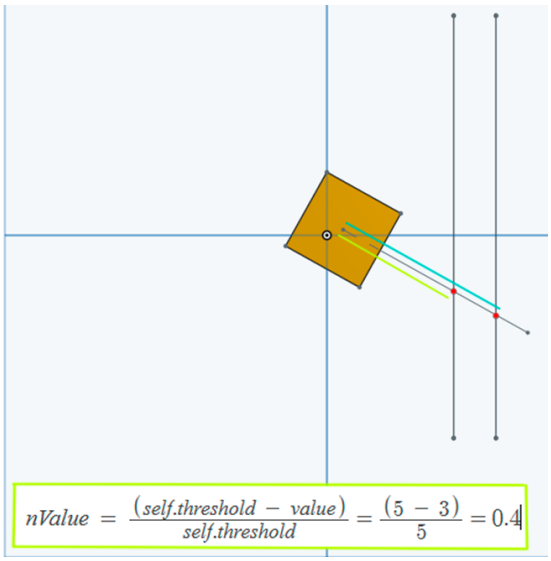


*Figura 23: Evaluación sensor de distancia*

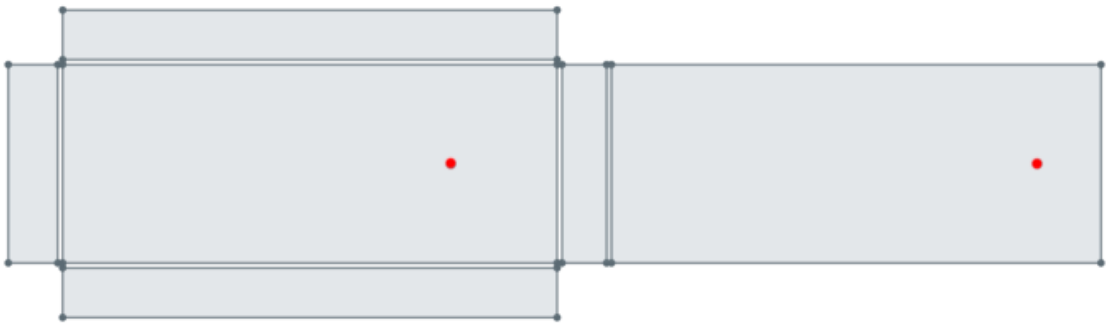

*Figura 24: Intersecciones sensor distancia con pared*

En lo respectivo al propio cálculo de intersecciones y distancias al centro del robot, se apoyará en una librería de Python llamada Geometry3D [14]. Esta librería nos va a facilitar la gestión de superficies y vectores para el cálculo de puntos de intersección entre ellos, gestionando los planos de las *hitboxes* como *ConvexPolygon* propios del paquete y el vector de distancia como una recta. La única consideración que debemos sopesar es que al usar el vector de distancia como una línea este no va a tener dirección, va a ser una línea que se va a extender por todo el espacio. Es decir, una vez calculado el punto de intersección, es necesario calcular el vector intersección y comprobar que tiene la misma dirección que el vector distancia. Se puede comprobar rápidamente de la siguiente forma:

$$\frac{x_d}{x_i} = \frac{y_d}{y_i} = \frac{z_d}{z_i} \rightarrow \text{misma dirección.}$$

Aunque la librería utilizada simplifique parte de los cálculos, se debe tener en cuenta que el coste de recursos que supone no es despreciable, ya que el peso computacional de cada evaluación del entorno por sensor de distancia es alto. Como se verá en las líneas futuras en la Sec. 5.2, hay ciertas optimizaciones que se pueden realizar de forma que solo se calculen las intersecciones una vez

comprobado que existen. Sin embargo, en esta versión inicial simplemente realizaremos un filtro de superficies sobre las cuales puede existir intersección. Este filtro únicamente comprueba que al menos uno de los vértices de la superficie se encuentre a menos distancia que el *threshold* definido, de forma que al menos exista la posibilidad de intersección.

Finalmente, también debemos tener en cuenta que el vector que define la dirección y el sentido del sensor está definido para un estado base del robot, en ausencia de movimiento y rotación. Es por ello que, para obtener los vectores reales de los sensores de distancia se debe aplicar la misma rotación y movimiento que tiene el robot. Esto se puede realizar con el mismo procedimiento aplicado en el Sec. 3.1.3, referente al mapeado de superficies, en el que se aplica una rotación y un movimiento a los vértices para situarlos en el mismo punto que se encuentra el robot, con la única salvedad es que en este caso se aplicar sobre vectores. Por otro lado, una vez ajustada la posición y la rotación del vector, solo queda adaptar su módulo para que coincida con el umbral, para ello se pueden seguir las siguientes ecuaciones que dan como resultado el vector de distancia final:

$$\text{ajust} = \frac{\text{threshold}}{\sqrt{x^2 + y^2 + z^2}} \rightarrow \overrightarrow{v_d} = \text{ajust} \cdot (x, y, z)$$

### 3.11.3. GRUPO DE SENSORES

No es de extrañar que según avance el desarrollo y la complejidad del simulador, los robots empiecen a usar muchos sensores de un mismo tipo, por lo que sería interesante crear un grupo de sensores que den una lectura conjunta. Por ejemplo, en el caso de los sensores de distancia, actualmente, si el usuario quisiera obtener la lectura de todos, tendría que ir llamando al método *read* en cada uno de los sensores individualmente. Es por ello que se tomó la decisión de incorporar un sensor llamado *GrupoSensores*, el cual realiza exactamente las mismas funciones que un sensor, pero sobre un conjunto de ellos. Por ejemplo, si ejecutamos el método *read*, este devolverá un *array* donde cada posición corresponde a una lectura de un sensor. O si llamamos al método *showMoreInfo* podemos hacer que se muestre más información en todos los sensores.

De momento, el único tipo de sensor que esta implementado como grupo de sensores es el sensor de distancia, tal y como se vio en la sección anterior. Así, simplemente accediendo a $self.sensors["distancia"]$ se obtendrá la lectura de todos los sensores de distancia con los que cuenta el robot, facilitando así el desarrollo de controladores que utilicen este tipo de sensores. Pero se debe tener en cuenta que el orden de las lecturas se da en el orden de incorporación de los sensores, de forma que hay que saber en las especificaciones del robot cuál es el orden de los vectores introducidos respectivamente para cada sensor de distancia.

### 3.11.4. RADAR

En un caso real, utilizar un sensor de distancia infrarrojo por cada eje principal del robot (como mínimo) va a suponer un gasto tan elevado que no va a ser realista. Por lo que una aproximación más realista sería un sensor capaz de realizar barridos a modo de radar, que fuera mostrando de forma menos instantánea la cercanía al entorno. Como se busca una versión inicial y funcional, no hay necesidad de hacer que este sensor tenga la capacidad de variar su plano de actuación. Por lo tanto, será restringido al plano $x$-$y$, pero implementado con visión de poder agregar, en caso de ser necesario, esta funcionalidad. Además, este sensor, va a heredar de la clase del sensor de distancia, ya que lo que es necesario es un sensor de distancia que vaya cambiando su posición dado un periodo. Los parámetros adicionales que se van a definir para el funcionamiento del radar son *readVel* y *numReads*: el primero define la cantidad de lecturas que se realizaran por iteración del simulador y el segundo define la cantidad de lecturas por vuelta que debe realizar el sensor. De esta forma tenemos

control total sobre la funcionalidad del sensor. En nuestro caso, el robot que utiliza el simulador tiene definido un total de 8 lecturas con una velocidad de 1 *lectura/tick* con un *threshold* de 5 metros al igual que los sensores de distancia. En la *Figura 25* se muestra una representación de los múltiples estados en los que se pueden encontrar el radar.

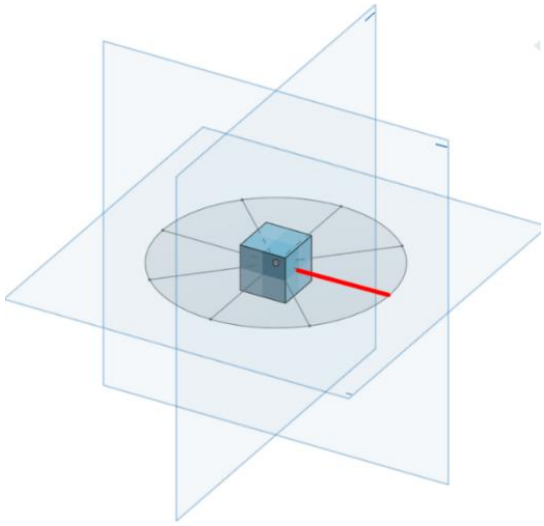

*Figura 25: Representación estados radar*

Por otro lado, también es destacable que, además de ser más realista, también es más eficiente. Esto es debido a que a diferencia del grupo de 6 vectores que tiene el sensor de distancia, los cuales realizan 6 lecturas simultaneas, este realiza una única por iteración. También genera una visión del entorno, aunque esta sea algo menos precisa. Se pueden observar comparativas detalladas en el Cap. 4, en el cual algunos de los experimentos se realizarán con ambos sensores evaluando las posibles diferencias. Respecto a la salida de los datos obtenidos, el sensor va a exportar una lista con las lecturas que tienen los estados y cada uno de estos se irá actualizando según se realice un nuevo barrido.

## 3.12. FLAGS Y ARCHIVOS DE CONFIGURACION

Una vez definidos todos los ámbitos estructurales que cubre el simulador, encontramos la necesidad de poder realizar modificaciones en la ejecución del script sin modificar código dentro del mismo. Estos ajustes externos dentro de la programación suelen estar diferenciados en dos grandes grupos: ajustes en línea de ejecución por medio de *flags* y configuraciones especificas generadas en archivos aparte. Habitualmente, la utilización de la segunda opción va de la mano del primer grupo, ya que para seleccionar estos archivos se utiliza un *flag*.

Dentro del ámbito de la programación, se definen los *flags* como opciones en línea de ejecución, es decir, ciertos argumentos que se especifican después del comando de ejecución del script:

```
python main.py # flags ...
```

*Figura 26: comando con flags*

En el simulador se van a poder utilizar los siguientes *flags* para realizar ajustes en línea de ejecución:

- *-novisual*: *flag* que permite ejecutar el simulador sin la parte visual. Es decir, se deshabilita todo el apartado gestionado por OpenGL y si queremos algún tipo de respuesta deberá ser o bien por vía terminal o bien mediante archivos de salida.
- *-c <controller>:* Este *flag* permite configurar un controlador directamente para todos los robots. El nombre del controlador deberá ser el mismo que tiene como clase dentro del simulador.
- *-o*: En este caso el *flag* configura dentro del simulador para que el robot llame a su función *export*, exportando también los datos definidos en el controlador tal y como se ha observado. En este caso

veremos más adelante cómo funciona y se estructuran los archivos de salida en el punto que tienen dedicado.

- *-m <arena>*: El siguiente *flag* nos permite configurar el mapa que se va a cargar en esa ejecución. El nombre de la arena deberá ser el que tiene su función generadora dentro del simulador.
- *-ef*: Nos permite habilitar el modo eficiente. Este modo lo que modifica es el modelo que utilizan diferentes elementos, como por ejemplo el robot, usando modelos 3d con menos polígonos y reduciendo así la carga que tiene el proceso de renderizado.
- *-cf <configuration_file>*: Este *flag* nos permite seleccionar un archivo de configuración perteneciente a la carpeta *configuration_files*. El nombre asociado al flag debe ser exactamente el mismo que el propio archivo.

Por otro lado, la otra opción de los archivos de configuración nos va a permitir hacer ajustes que serían imposibles por medio de *flags*. Estos corresponden al número de robots que se quieren utilizar en un experimento y, aún más importante, el estado inicial de los mismos. Para este tipo de configuraciones se ha decidido utilizar archivos con extensión *.json*, este tipo de archivos son muy utilizados para este tipo de tareas. *JSON* es el acrónimo de *JavaScript Object Notation*, y permite hacer descripciones de objetos como nuestros robots de forma muy rápida, legible y cómoda. Como se ha mencionado este tipo de archivo permite ajustar el estado inicial del robot, esto supone la capacidad para configurar todos los parámetros que tiene, en la *Figura 27* podemos ver una lista de los mismos en formato *JSON*.

```
{
  "position": [x, y, z],
  "rotation": [roll, pitch, yaw],
  "linearSpeed": [vx, vy, vz],
  "angularSpeed": [wx, wy, wz],
  "motorState": [m1, m2, _],
  "controller": "Example1"
}
```

*Figura 27: Robot estado inicial JSON*

Por otro lado, en la *Figura 28* se puede observar un ejemplo de un archivo de configuración completo.

```
{
    "robots": [
        {
            "position": [
                0.0,
                7.0,
                -1.0
            ]
        },
        {
            "position": [
                0.0,
                -7.0,
                -1.0
            ]
        },
        {
            "position": [
                7.0,
                0.0,
                -1.0
            ]
        },
        {
            "position": [
                -7.0,
                0.0,
                -1.0
            ]
        },
    ]
}
```

*Figura 28: Ejemplo archivo de configuración*

El anterior archivo de configuración permite crear un experimento con 4 robots, que tienen posiciones iniciales modificadas.

Como se puede observar, este tipo de estructura permite realizar un ajuste de forma bastante intuitiva y rápida, sin necesidad de modificar nada de código y pudiendo alternar entre diferentes archivos de configuración directamente en la línea de ejecución. Esto posibilita que en caso de estar probando un controlador que hemos desarrollado, se pueda visualizar diferentes configuraciones del entorno rápidamente y así agilizar el proceso de desarrollo.

Finalmente, como última consideración, dentro de los parámetros iniciales que tiene el robot que pueden ser configurados, se puede ver que se encuentra el controlador, también definido dentro de las *flags.* Esto es debido a que en la mayoría de los casos todos los robots tienen el mismo controlador y tienen el mismo comportamiento, pero en caso de querer configurar un experimento donde cada robot tenga un controlador, podremos hacerlo en el archivo de configuración y este solapará la configuración especificada en los *flags*.

### 3.12.1. ARCHIVOS DE SALIDA

Junto a la necesidad de configurar apartados del simulador sin modificar código, se encuentra también la de exportar datos del simulador para así poder realizar un estudio posterior y ver si realmente los comportamientos definidos se realizan satisfactoriamente, también es deseable poder almacenar trayectorias y evaluaciones en entornos concretos. Es por ello que hay una serie de métodos configurados para la gestión de datos a exportar y un *flag* dedicado a activarlos. Cada sesión con el guardado de datos activado crea una carpeta dentro de *output_files* con el nombre *EXP-YYYY_MM_DD-HH:MM:SS_PM*. A su vez cada robot exporta sus datos a un archivo de tipo *JSON* que se llama *Robot-id.json*. Se puede un ejemplo de diferentes sesiones en la *Figura 29*.

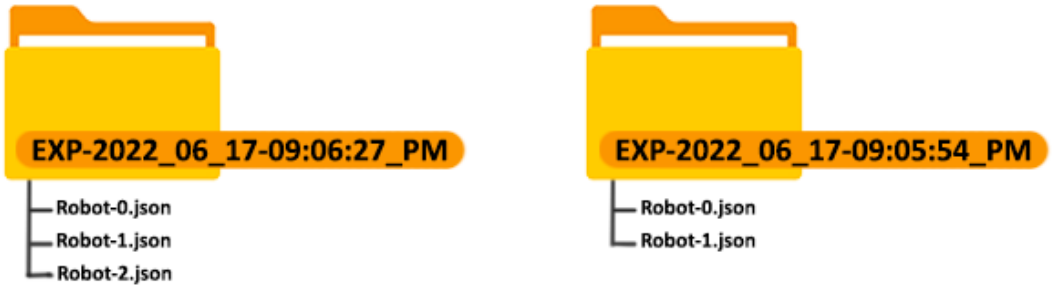


*Figura 29: Estructura de output files*

Respecto a los datos que se exportan de cada robot, como ya se mencionó anteriormente, son todos los datos del estado base en cada iteración almacenándolos en una lista. Todos estos datos, al igual que con los archivos de configuración, serán gestionados en archivos *JSON*. En este caso no es tanto por la comodidad y la legibilidad, sino por la versatilidad que este tipo de archivos ofrece. Al tratarse de datos que van a ser analizados o graficados, lo más importante es que sean muy versátiles y puedan leerse en cualquier lenguaje de programación y justo esta es una de las mayores ventajas que tiene *JSON* ya que está altamente estandarizado para el almacenamiento de datos y la mayoría de los lenguajes permiten leerlos y mapearlos a objetos propios. Otra de las posibles opciones usadas sobre todo en el sector del IOT sería el uso de archivos *CSV*. Sin embargo, esto en nuestro caso dificultaría el análisis de datos fuera de *Python* [2] o diversos lenguajes populares utilizados en el sector, además de que la principal ventaja de estos ficheros es la gran capacidad de manejar grandes bases de datos y poder procesarlas a altas velocidades.

En la *Figura 30* podemos ver un ejemplo de cómo quedaría exportado un experimento, en el que el *export* del controlador asigna únicamente las lecturas del sensor GPS que es el utilizado.


```
[
    {
        "position": [
            1.0,
            7.0,
            -1.0
        ],
        "rotation": [
            0.0,
            0.0,
            0.0
        ],
        "linearSpeed": [
            0.0,
            0.0,
            -0.013333333333333334
        ],
        "angularSpeed": [
            0.0,
            0.0,
            0.0
        ],
        "motorState": [
            0.0,
            0.0,
            0.0,
            0.0,
            0.16666666666666669,
            0.16666666666666669
        ],
        "control": [
            0,
            0,
            0,
            0,
            1.6666666666666667,
            1.6666666666666667
        ],
        "sensores": {
            "radar": [],
            "distancia": [],
            "gps": [
                1.0,
                7.0,
                -1.0
            ]
        }
    },

// ...
```


*Figura 30: Fichero salida de ejemplo*

# 4. EXPERIMENTOS

A lo largo de este capítulo se han desarrollado una serie de experimentos que pongan a prueba el simulador y a su vez den una visión general de las dinámicas y del comportamiento del robot en el medio. Para ello se puso un objetivo final, el cual era desarrollar un experimento de *avoid obstacle* el cual permita al robot moverse por el entorno esquivando los obstáculos que vaya encontrando, incluyendo otros robots. Este objetivo final se fue subdividiendo en diferentes experimentos más pequeños que acaben en su conjunto permitiendo el desarrollo de este, además de mostrar diversos comportamientos. Partiendo de esta premisa se presenta la siguiente estructura:

- Experimento 1: Análisis del movimiento que tiene el robot. Empezando por el propio control, que es el que genera el movimiento, y acabando por la interacción con el medio y las inercias que genera el mismo.
- Experimento 2: Para poder desarrollar el resto de los experimentos sin problemas se va a realizar una prueba enfocada en entender la flotabilidad del robot en el entorno y como compensar la misma para situarse a una altura determinada.
- Experimento 3: Primera aproximación de *avoid_obstacle* funcional, únicamente en un eje. Se basa en una compensación de inercia en tres etapas: movimiento libre, evitar y calma. Durante la calma se ejerce una fuerza que compensa el gran empuje generado para evitar el obstáculo, antes de volver a movimiento libre.
- Experimento 4: Ante los resultados del experimento 3, se cambia el enfoque buscando antes un control de velocidad que compense automáticamente la inercia y así solventar los problemas que tenía este.
- Experimento 5: Versión final de *avoid_obstacle* basada en un control de velocidad, que permite crear un entorno en una piscina con varios robots simultáneamente sin que ninguno tenga colisiones entre ellos o con otros elementos del entorno.

Como se puede ver los experimentos 1, 2 y 4 están centrados en representar ciertos funcionamientos propios del entorno y el simulador. Por otra parte, el experimento 3 y 5 buscan diferentes enfoques del controlador *avoid_obstacle*.

## 4.1. EXPERIMENTO 1: MOVIMIENTO E INERCIA

Tal y como se ha mencionado anteriormente, a diferencia de los simuladores de robots terrestres, este simulador al ser acuático va a tener una serie de detalles diferentes de alta importancia. El primero de ellos parte del propio robot, en este caso no se va a tener un actuador que active el motor de una rueda u orugas que muevan nuestro robot en un eje. El *BlueRov2* [1] va a contar con seis turbinas, cuatro de las cuales se encargarán del movimiento en el eje $x$ *e* $y$ y dos de ellas del movimiento en el eje $z$. Se pueden observar las ecuaciones finales de la fuerza total en cada eje en función del control asignado a cada turbina en las siguientes ecuaciones:

$$X_{motor} = \frac{\sqrt{2}}{2}(F_1 + F_2 + F_3 + F_4)$$

$$Y_{motor} = \frac{\sqrt{2}}{2}(F_1 - F_2 - F_3 + F_4)$$

$$Z_{motor} = F_5 + F_6$$

Como se puede observar, hay cierta interferencia entre el eje $x$ e $y$ de forma que, si no se realiza un estudio adecuado, se pueden descompensar las fuerzas y hacer que los ejes se anulen entre sí.

Otro punto importante acerca del movimiento del robot es la capacidad de rotar. Como estos comportamientos vienen dados por el modelo dinámico asociado al robot, en este caso el *modelo0*, de momento solo están implementados los giros entorno al eje $z$ o la capacidad de generar *Yaw*. Esto supone que, si las fuerzas no están compensadas en los controles del eje $x$ o el $y$, se van a generar

estas rotaciones, mientras que en el eje $z$ esto todavía no sucede. No obstante, puesto que es esperable que se implemente en futuros modelos, se tendrá en también en consideración.

A lo largo de este experimento se van a observar las configuraciones de los controles, con el fin de generar movimientos sin rotación en cualquiera de los ejes independientemente. En las siguientes figuras se presentan las trayectorias y los controles asociados a las mismas:

- Eje $x$:

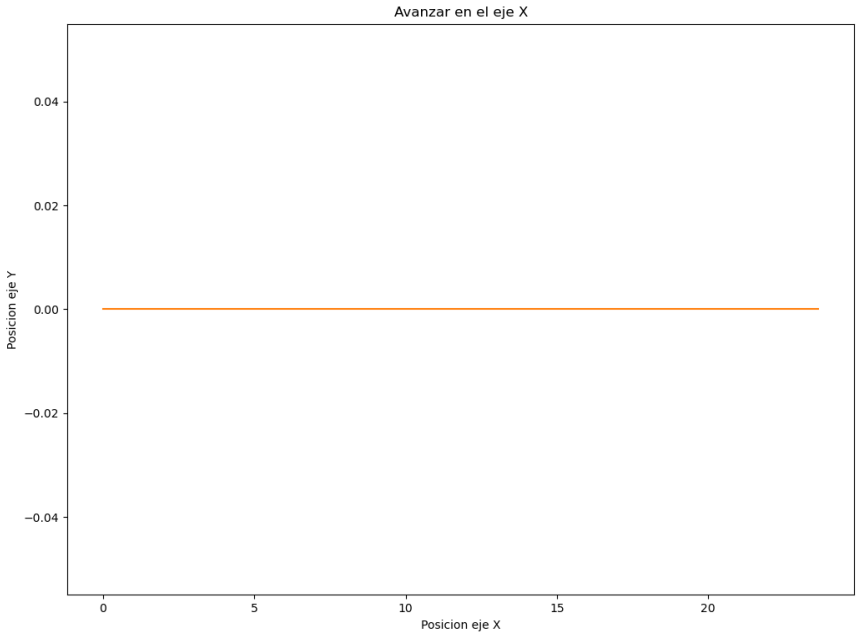


*Figura 31: Movimiento eje* $\boldsymbol{x}$*. Control = [1, 1, 1, 1, 0, 0]*

- Eje $y$:

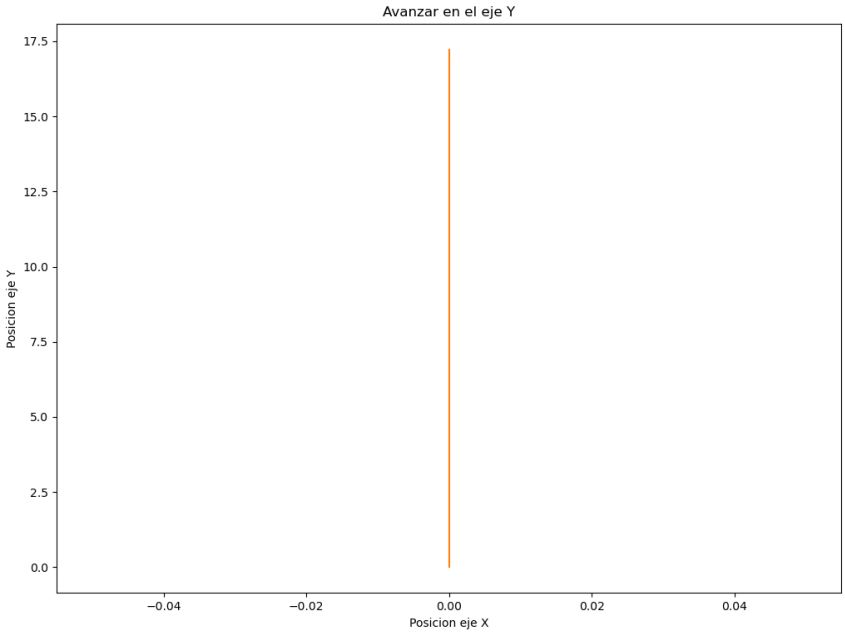


*Figura 32: Movimiento eje* $\boldsymbol{y}$ *y Control = [1, -1, -1, 1, 0, 0]*

- Eje $x$ e $y$:

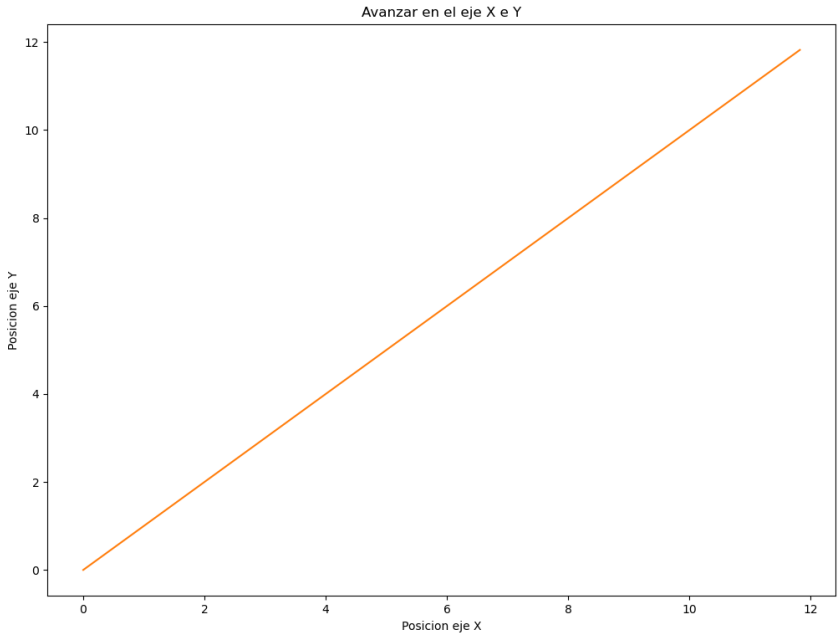


*Figura 33: Movimiento eje $\boldsymbol{x}$ e $\boldsymbol{y}$. Control = [1, 0, 0, 1, 0, 0]*

- Eje $x$ y $-y$:

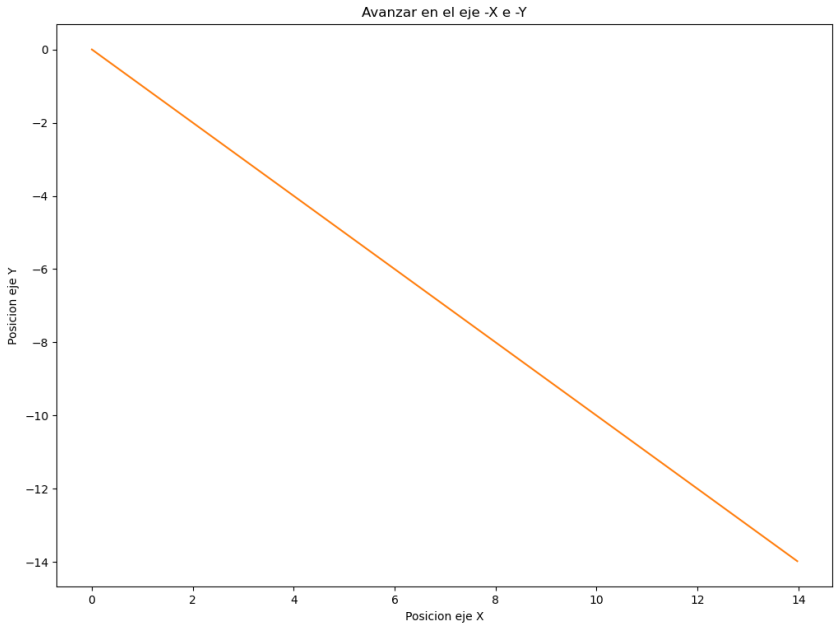


*Figura 34: Movimiento eje $\boldsymbol{x}$ y $\boldsymbol{-y}$. Control = [0, 1, 1, 0, 0, 0]*

A su vez, todas estas trayectorias tienen asociadas sus contrarias cambiando el signo de los controles, como se puede suponer.

Por otro lado, se puede ver que creando ciertas combinaciones con los controles se obtienen todos los grados de libertad que se requieren para poder mover en todos los ejes al robot. Además, al no buscar rotar el robot, es importante que las fuerzas aplicadas en los controles sean uniformes. Para ello se ha creado una función que se encarga de gestionar el control final dadas las fuerzas independientes de cada eje. Se puede observar en la *Figura 35.*

```python
def globalMove(self, x, y):
    # [1, 1, 1, 1] → avanza recto en el eje x sin girar
    # [-1, -1, -1, -1] → avanza rector en el eje -x sin girar
    # [1, -1, -1, 1] → avanza rector en el eje -y sin girar
    # [-1, 1, 1, -1] → avanza recto en el eje y sin girar
    # [2, 0, 0, 2] → avanza en diagonal de 45deg en el eje x e y sin girar
    # [-2, 0, 0, -2] → avanza en diagonal de 45deg en el eje -x e -y sin girar
    # [0, 2, 2, 0] → avanza en diagonal de 45deg en el eje x e -y sin girar
    # [0, -2, -2, 0] → avanza en diagonal de 45deg en el eje -x e y sin girar
    if y == 0:
        return [x] * 4
    elif x == 0:
        return [y, -y, -y, y]
    elif (x > 0 and y > 0) or (x < 0 and y < 0):
        return [(x+y), 0, 0, (x+y)]
    elif x > 0 and y < 0:
        return [0, (x-y), (x-y), 0]
    elif x < 0 and y > 0:
        return [0, -(y-x), -(y-x), 0]
    return [0] * 4
```

*Figura 35: Función generadora de controles*

Esta función también gestiona fuerzas diferentes en cada eje. Por lo que si se busca, por ejemplo, un empuje 3 veces mayor en el eje $x$ que en el eje $y$, se puede utilizar la configuración $Control = [2, 1,1,2,0,0]$, cuya trayectoria se muestra en la *Figura 36*.

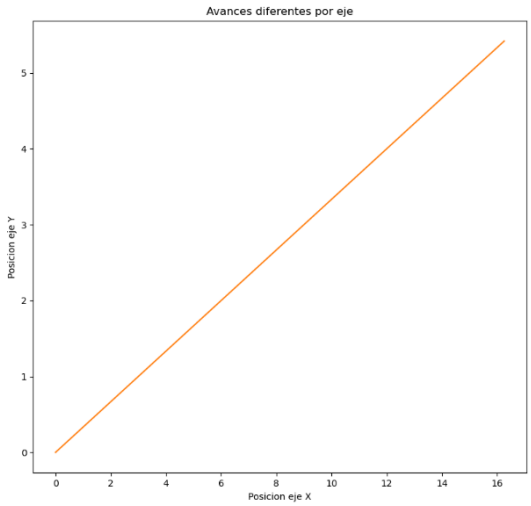


*Figura 36: Movimiento eje x tres veces mayor que eje y. Control = [2, 1, 1, 2, 0, 0]*

Por otra parte, para evaluar también las posibilidades que ofrece el modelo utilizando rotaciones y controles no compensados, se ha realizado un experimento sin utilizar esta función. La trayectoria resultante se puede observar en la *Figura 37*.

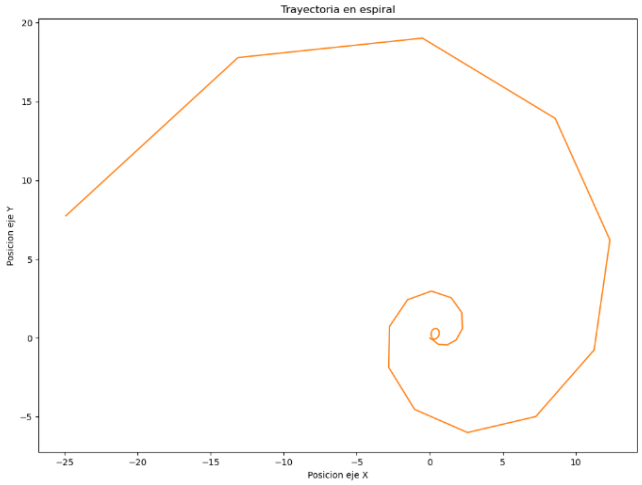


*Figura 37: Movimiento en espiral. Control = [-1, 1, 1, 1, 0, 0]*

Como se ha podido observar, una pequeña descompensación en las variables de control puede generar una trayectoria completamente diferente por esa ligera rotación producida. Al igual que con cierto control descompensado y variable podría intentarse conseguir una trayectoria circular, que provocaría al robot orbitar entorno a un punto.

Finalmente, otro aspecto importante a estudiar en este experimento es la diferencia con los simuladores terrestres de la inercia generada por el medio. En el resto de los simuladores, si el control se pone a 0 y los motores se frenan, el robot va a pararse casi inmediatamente. Sin embargo, en nuestro caso la inercia va a ser tal que nuestro robot reducirá su velocidad de forma lineal. Se puede ver este suceso con más detalle en la *Figura 38.*

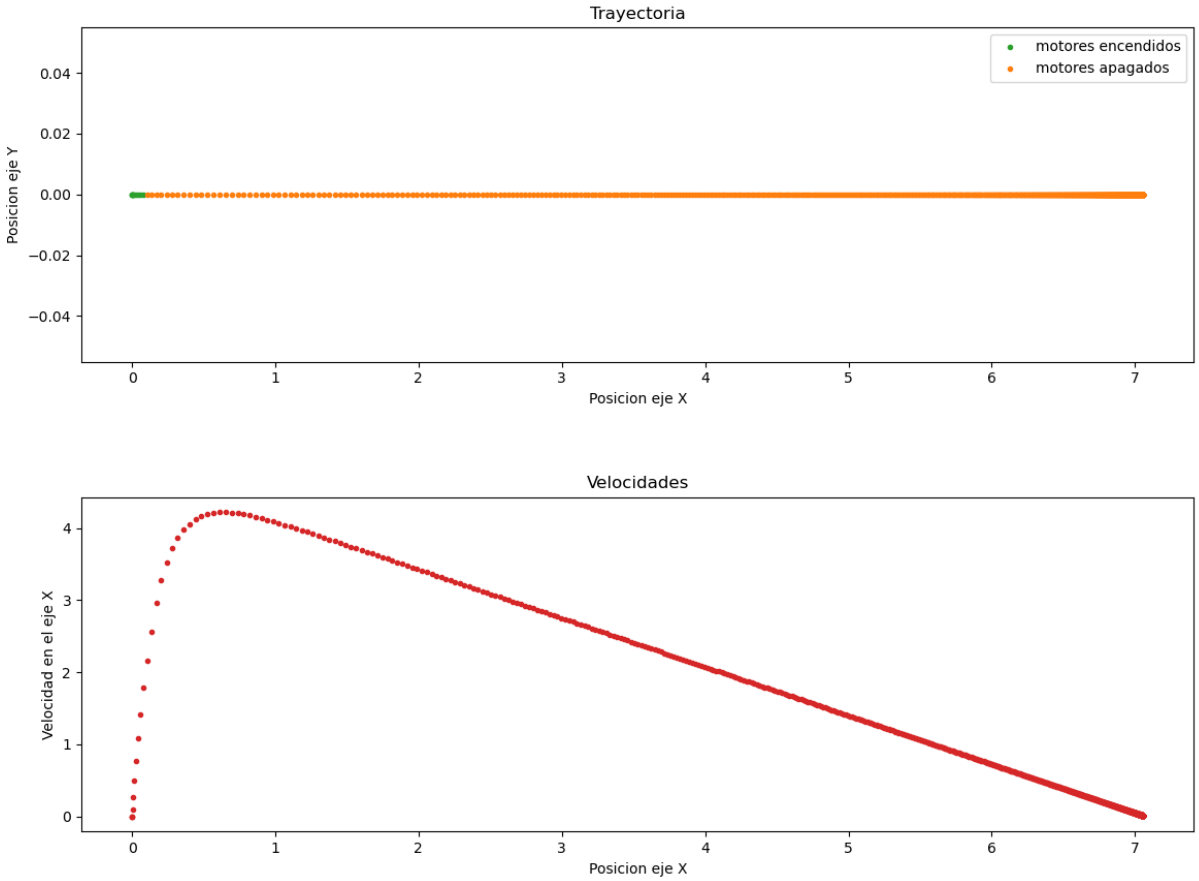


*Figura 38: Trayectoria con inercia y velocidad*

Tal y como se puede ver en la figura, según se apagan los motores el robot llega a su máxima velocidad a lo largo de la trayectoria. Tras esto, tiene lugar una fase de adaptación donde la velocidad empieza reducirse hasta entrar en un proceso de reducción de velocidad lineal que termina haciendo tender la velocidad a 0. Para el resto de los experimentos va a ser muy importante tener este factor en cuenta, ya que sobre todo para los experimentos principales esta inercia va a ser difícil de compensar en muchas situaciones e incluso compensarla puede suponer una fuerza todavía más grande en sentido contrario.

Como conclusiones del experimento, se tiene la capacidad de generar movimientos en cualquiera de los ejes sin implicar ningún tipo de rotación para el robot, aunque de necesitarse, se podría generar entorno al eje $z$. Se debe buscar una manera de compensar las inercias generadas por el entorno para así tener transiciones limpias entre estos movimientos, ya que si no se compensa la fuerza acumulada resultará en movimientos no deseados.

## 4.2. EXPERIMENTO 2: CONTROL DE FLOTABILIDAD

Tal y como se ha visto en el funcionamiento del control, el eje $z$ (a diferencia del $x$ e $y$), está separado completamente. Esto va a permitir realizar controles y adaptaciones en la altura de forma más rápida y aislada. Para evitar que los movimientos en el eje z sean un problema en el resto de los experimentos, se va a desarrollar un algoritmo que permita al robot mantener una altura cuasi estable y así poder centrar la atención en el funcionamiento propio del controlador en el plano $x$ e $y$.

Como norma general, los robots acuáticos cuentan con un factor de flotabilidad bastante alto, esto es debido a que en caso de tener algún problema de control o de batería, se busca que salga a flote para no tener que dedicar recursos a su búsqueda en el fondo marino (algo que puede ser muy costo). Pero en este caso, hay que enfocarlo desde el punto de vista contrario, en el que el robot busca estar a una altura fija sin ningún tipo de flotabilidad visible. Para ello se debe buscar una función que contrarreste esta fuerza de empuje hacia arriba, pero sin ser excesiva ya que se puede acabar generando el efecto contrario y que el robot se hunda. Otro factor notorio dentro de la búsqueda de esta flotabilidad estable es que en el simulador se podrá utilizar un valor inicial z y según sea de pequeño (siendo menor más profundidad) más fuerza generará el robot para salir a flote. Por lo que es de esperar que este control de flotabilidad solo funcione en cierto margen de profundidad. En la *Figura 39* se aprecian diversas gráficas de la trayectoria en el eje z de diferentes robots con el control a 0 y diferentes alturas iniciales, exponiendo de forma más representativa este fenómeno.

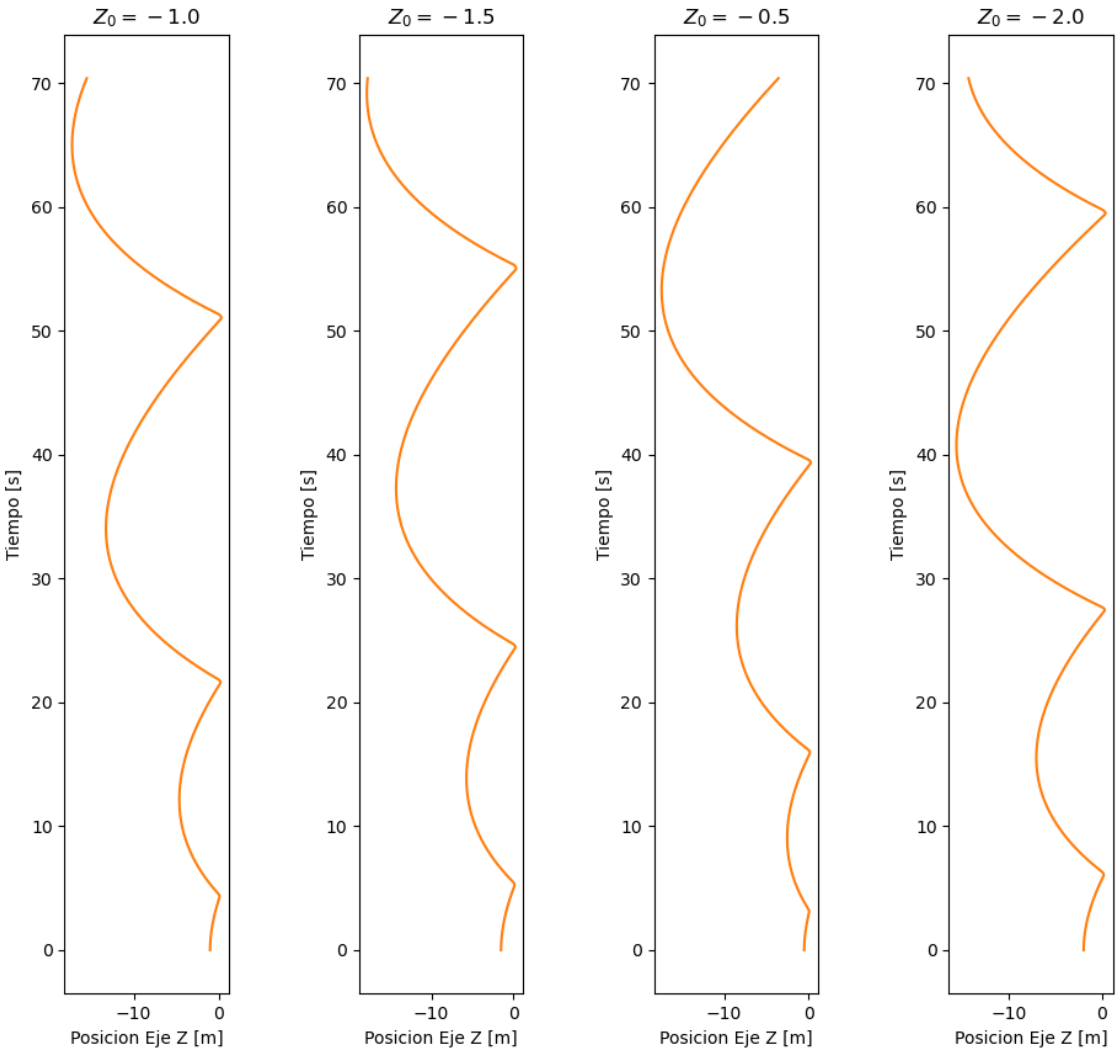


*Figura 39: Flotabilidad base para diferentes alturas iniciales*

Tal y como se puede observar, el robot en todos los casos sale a flote hasta que su centro de gravedad cruza el punto $z = 0$ y se hunde de nuevo, siendo cada vez mas importante esta fuerza de subida y por ende la de bajada. En consecuencia, se produce esa especie de oscilación que cada vez es más pronunciada en las gráficas.

Tras ver esta trayectoria en las gráficas, se puede concluir que el algoritmo que se busca va a tener que definir una altura z que sea la posición de equilibrio. Esta altura de equilibrio va a ser entorno a la cual oscilara el estado de flotabilidad y hundimiento, que va a depender de la altura inicial del robot. Además, se debe tener en cuenta que cuando la fuerza total sea mayor que la flotabilidad será cuando el robot entrará en un estado de hundimiento. Eso supone que aunque la fuerza no sea 0, se puede aplicar una fuerza que simplemente ralentice la fuerza de flotabilidad. Bajo esta premisa, se llega a la ecuación mostrada en la *Figura 40.*

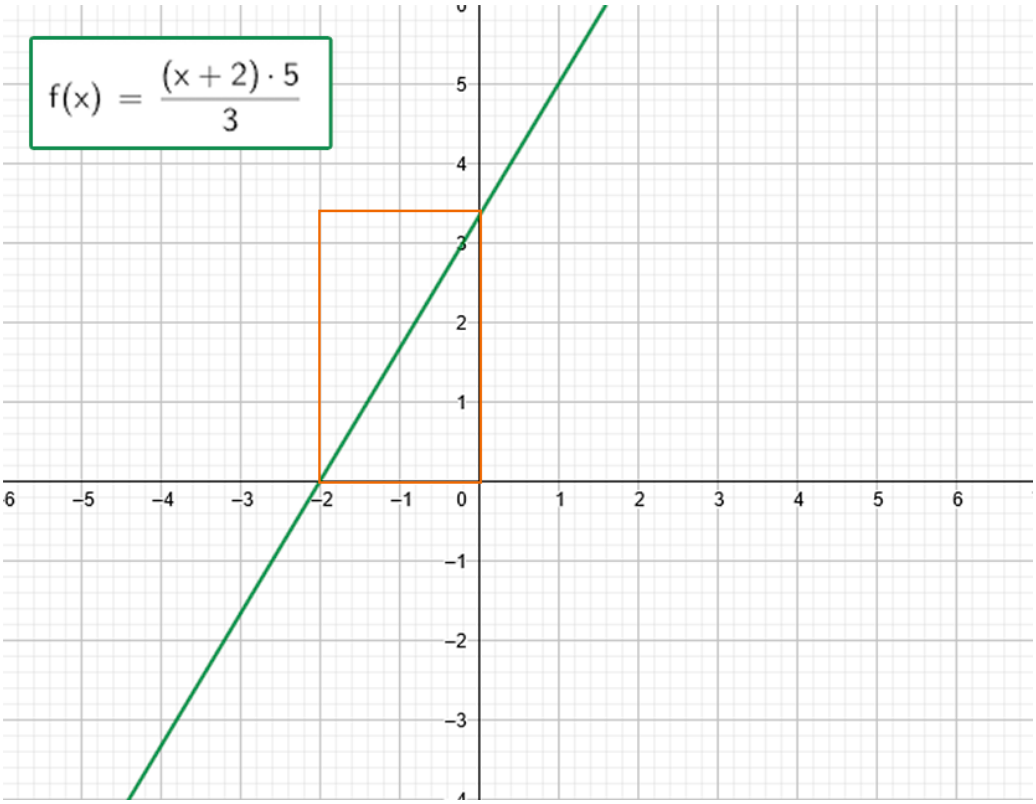


*Figura 40: Función de flotabilidad estable*[1]

La salida de la función refleja el valor de control que se le debe poner a cada una de las turbinas dedicadas al eje $z$. Como se puede observar es una función lineal de pendiente pronunciada, lo cual busca cambios bruscos ante ligeras variaciones de altura. Respecto a la constante que se suma a la altura, el *2*, lo que define es el rango de la zona estable. Se define la región estable como el rectángulo naranja de la función, siempre que la altura del dispositivo se mueva en esos valores (sin incluir los extremos para la altura inicial) se tendrá un proceso prácticamente estable.

En la *Figura 41* al igual que en la *Figura 40,* se puede ver la trayectoria de una serie de robots con alturas iniciales diferentes, sobre los cuales se ha aplicado el algoritmo de estabilidad explicado anteriormente. De esta forma, queda representada la variación de altura en el eje $z$ en función al tiempo, para así poder evaluar el funcionamiento del propio algoritmo y confirmar o desmentir las afirmaciones enunciadas partiendo únicamente de la expresión base.

[1] Siendo $x$ la altura y $f(x)$ el control de salida

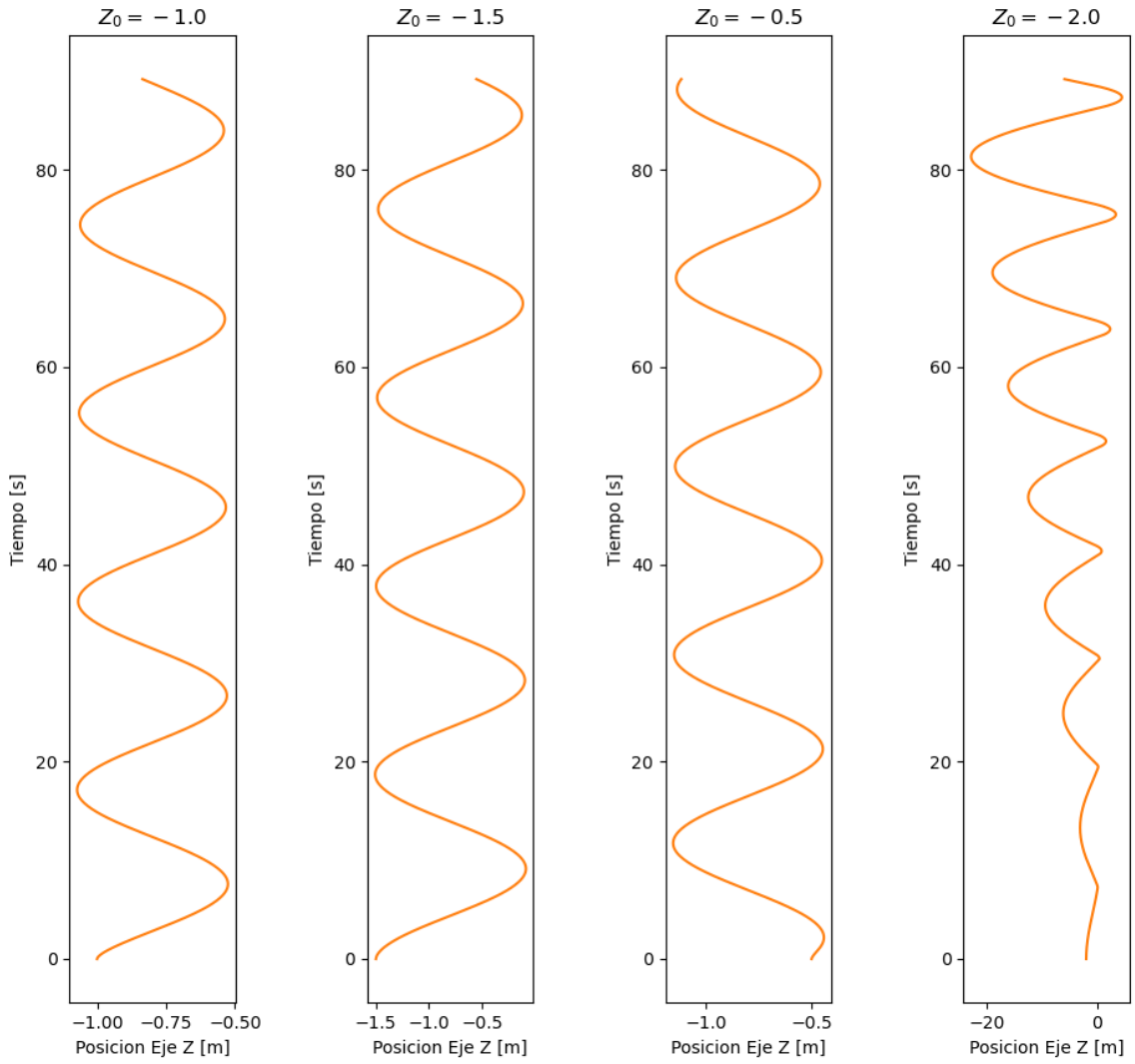


*Figura 41: Flotabilidad aplicando algoritmo de estabilidad con rango 2*

Tal y como se puede observar a primera vista, los robots que están dentro del rango de estabilidad han generado una trayectoria sinusoidal, mientras que el que se encontraba al límite de la zona se ha comportado de una forma más parecida a lo que se tenía en ausencia de control. Dentro de las tres primeras gráficas se puede ver que la que más variación tiene, o la que mayor amplitud en la sinusoide generada presenta, es la segunda, que es la que más cerca se encuentra del límite, y de la superficie. Por otro lado, la más alejada del límite inferior es la que menos rango de variación tiene, pero a su vez la que más cerca de la superficie se encuentra. Observando así, que no es una estrategia óptima aproximar todo lo que se pueda al cero. La opción inicial, que se encuentra en un punto medio respecto a los límites, genera una fluctuación que entra dentro de los márgenes aceptables y mantiene ambos límites a distancias asumibles.

Observando la posición y los rangos generados por la primera gráfica, se puede concluir que es la mejor opción para el resto de los experimentos. Sin embargo, a modo de comprobación adicional, se van a generar las trayectorias para un algoritmo de flotabilidad con una constante *3* para este caso. Con este valor, la posición inicial $z = -2$ entraría dentro de los márgenes estables y debería generar una trayectoria igual a las demás, pero con una gran variación por ser bastante cercana a los límites. Se observa estas trayectorias en la *Figura 42.*

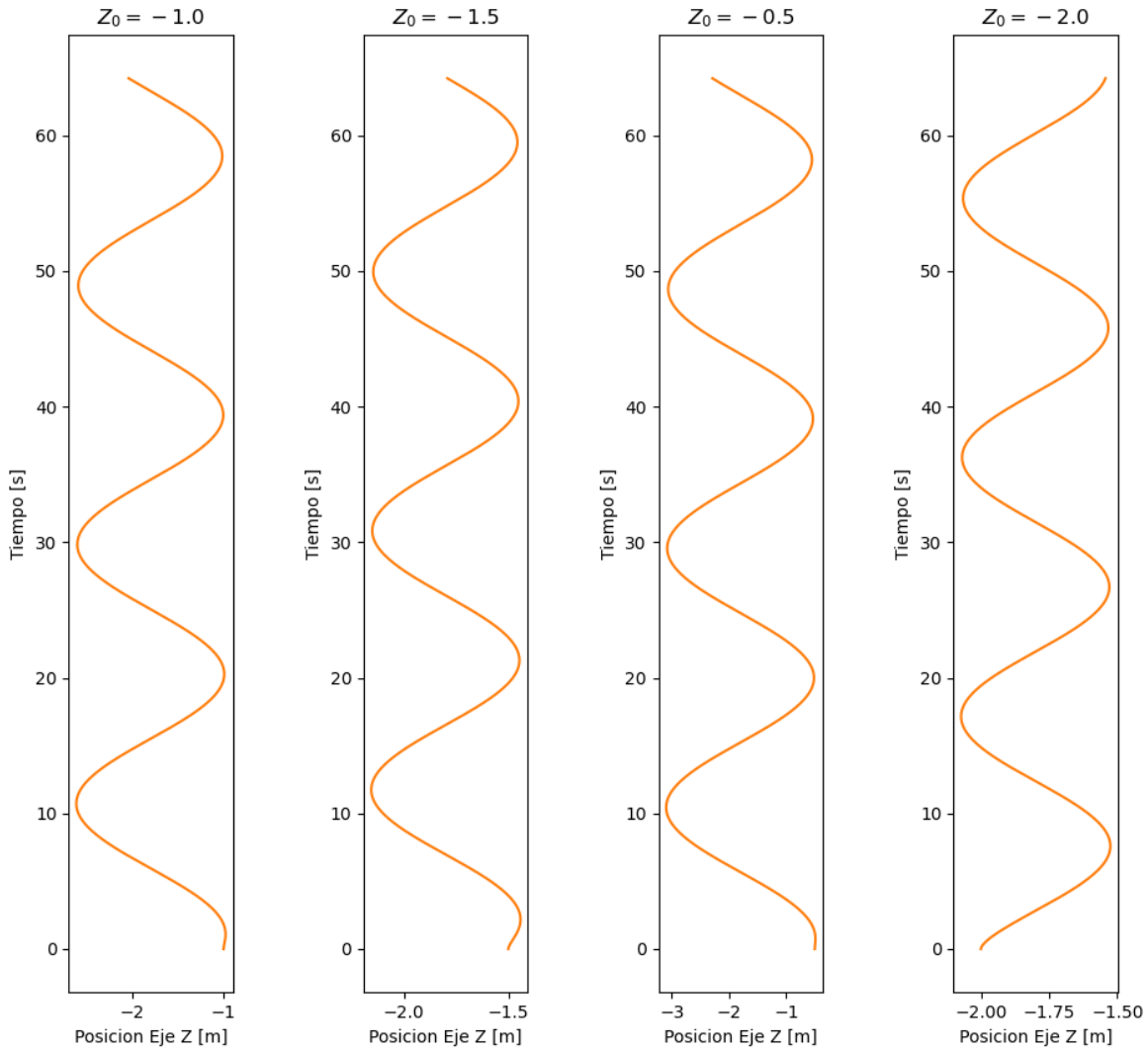


*Figura 42: Flotabilidad aplicando algoritmo de estabilidad con rango 3*

Se puede observar que, efectivamente, para estas condiciones la última gráfica también genera una posición estable. A su vez, la que mejor margen de variación y rango de posiciones tiene es la que se encuentra en el punto medio de esta región ($z = 1.5$). Incluso, la tercera gráfica de altura inicial $z = -0.5$, al ser la más próxima a los límites, es la que peor comportamiento presenta, generando una gran amplitud en la sinusoide y encontrándose muy cercana a los límites tanto superiores como inferiores.

Como conclusiones del experimento, se puede decir que se ha logrado un algoritmo que genera comportamientos estables en el eje *z* dentro del simulador, además de tener ciertos valores de ajuste para asegurar que el robot se encuentre siempre en la región estable.

## 4.3. EXPERIMENTO 3: AVOID OBSTACLE V1

Con los conocimientos incorporados en los anteriores experimentos, se puede hacer una aproximación al objetivo final de crear un algoritmo que permita navegar al robot esquivando obstáculos. Volviendo a los puntos iniciales, se sigue con el mayor de los problemas, la gestión de la inercia. El pensamiento más intuitivo es simplemente no tenerla en cuenta, y que al detectar un obstáculo se impulse con una fuerza proporcional a la cercanía con el mismo. Pero eso va a terminar generando una aceleración en el sentido contrario que va a hacer incapaz contrarrestarlo en el siguiente obstáculo encontrado,

generando un comportamiento no deseado. Es por ello que para gestionar esta inercia, que supone aplicar una fuerza muy grande, se va a utilizar una fase extra además de la propia de esquivar el obstáculo y navegar libre. Esta nueva etapa emplea una fuerza de frenada una vez se haya aplicado la fuerza suficiente para esquivar al obstáculo.

Entrando más en detalle dentro de cada etapa, a continuación se exponen las principales funciones de cada una. Durante la fase de navegación libre, el robot se encargará de navegar en la dirección de movimiento. En un estado inicial se parte de un avance en los ejes positivos. La fase esquivar hace que el robot generé una fuerza en sentido opuesto al movimiento y proporcional a la distancia a la que se encuentra del obstáculo. De forma que si aparece un objeto muy cerca del robot la fuerza generada por el mismo será también mucho mayor, para así poder esquivarlo adecuadamente. Por último, la fase de calma se encarga de aplicar una fuerza proporcional a la que ha sido necesaria para esquivar y así reducirla linealmente hasta que el robot acaba teniendo un movimiento con la inercia prácticamente compensada. Una vez conocidas las funciones de todas las fases, simplemente queda mencionar que existe un nivel de prioridades según el estado al aplicar las fuerzas. Si un eje tiene un estado de navegación libre y el otro de esquivar, no merece la pena hacer una media de las fuerzas y aplicarlas como ya se vio en la Sec. 4.1. Es más óptimo centrarse en la función esquivar, al igual que pasaría entre una fase calma y una de navegación libre, siendo la única situación en la que se reparten las fuerzas cuando ambos ejes se encuentran en la misma fase.

El sensor utilizado para estimar a qué distancia se encuentra el robot dentro del entorno es el radar, tal y como se explica en la Sec. 3.11, por su eficiencia y su mayor reflejo de la realidad. Sin embargo, en las pruebas finales que se realicen con múltiples robots, se utilizarán también sensores de distancia de forma que se pueda comparar el resultado con diferentes sensores y así evaluar las diferencias que ofrecen.

En lo referente a la implementación de este algoritmo, se va a realizar una incorporación parcial de la metodología aplicándola solo a un eje. De forma que se pueda comprobar la efectividad de esta y, en caso de ser correcta, expandirlo a los dos ejes. Como primer experimento para poner aprueba el algoritmo se va a introducir un único robot en la piscina cuadrada, situado en el punto central. De esta forma, el robot solo tendrá que esquivar las paredes de los laterales alternando las direcciones de movimiento y centrando el experimento en ver la compensación de inercia. Se puede ver la trayectoria que ha seguido dentro del eje x en la *Figura 43.*

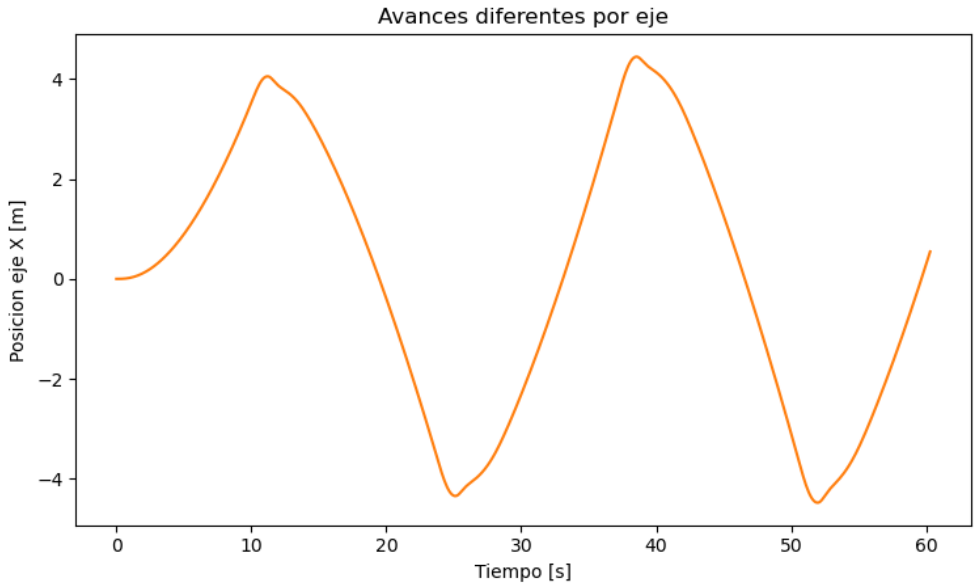


*Figura 43: Avoid Obstacle en un eje un solo robot*

Como se ha podido confirmar, el comportamiento del robot se encuentra dentro de lo esperado. Es capaz de compensar las inercias propias del estado esquivar y volver a un estado libre gracias al estado de calma. También se puede ver como la duración del estado calma dura más que el propio esquivar, lo que es debido a que para hacer una buena compensación de la inercia se debe utilizar fuerzas más pequeñas, pero de forma más prolongada en el tiempo. Esto puede ser un problema más adelante ya que es importante tener una buena capacidad de reacción y adaptación por parte del robot.

En la siguiente fase del experimento, para dar por efectivo el algoritmo en un solo eje, se va a realizar una prueba con dos robots en las mismas condiciones que el experimento anterior, pero cambiando las posiciones iniciales de los mismos para no solaparse. Se pueden observar las trayectorias de ambos robots en la *Figura 44*.

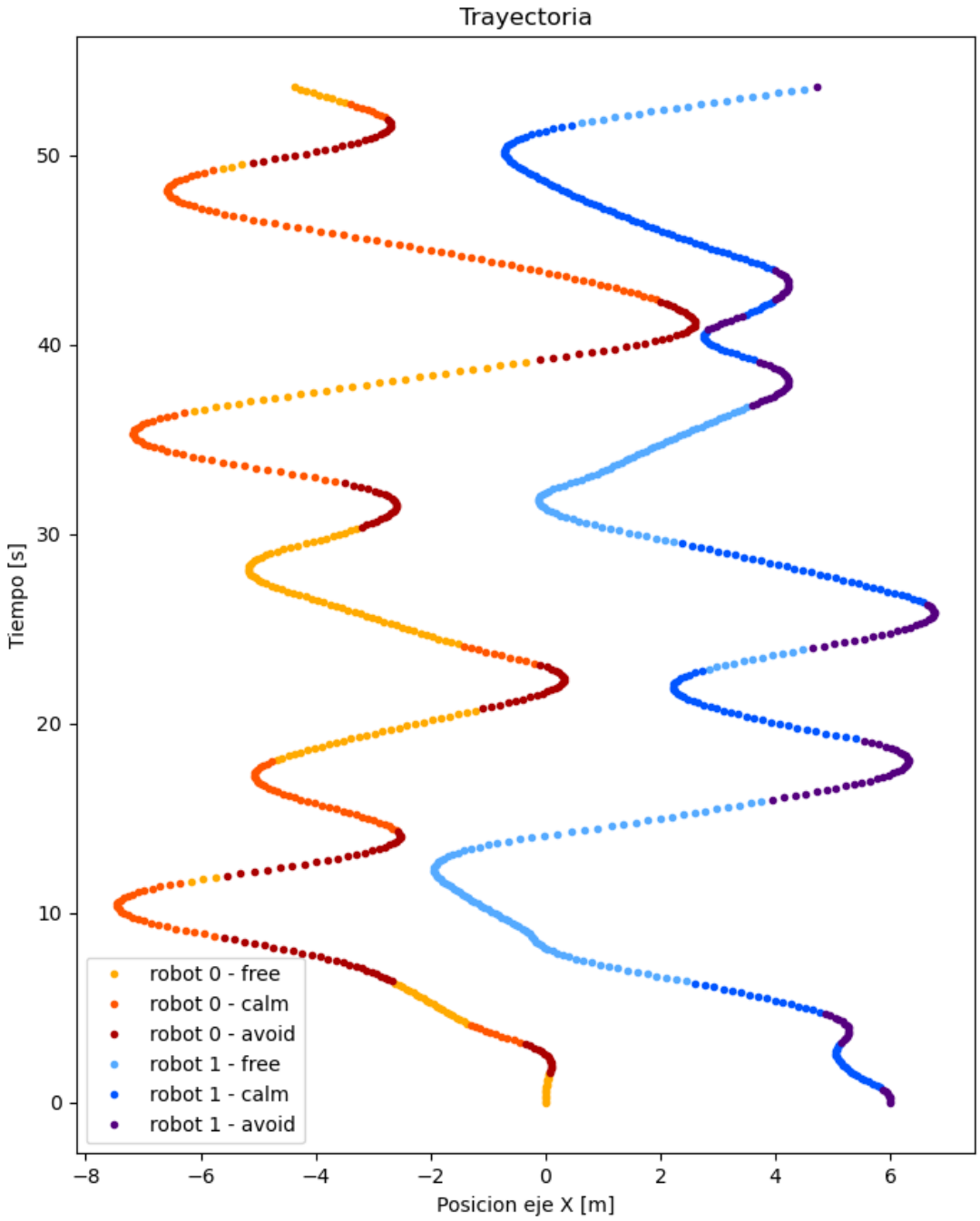


*Figura 44: Avoid Obstacle en un eje múltiples robots*

Se puede observar que, aunque las trayectorias no lleguen a juntarse, el tamaño del robot hace que realmente haya ciertas colisiones, aunque no lleguen al punto de atravesarse por completo los robots. Sobre todo, se pueden apreciar estos problemas cuando hay muchas fases de esquivar seguidas, que no permiten ajustar correctamente la inercia en el periodo de calma. Por lo que se puede decir que el algoritmo no es lo suficientemente bueno como para explorarlo en el plano entero. Sin embargo, sí que puede ser interesante ver la diferencia entre este mismo experimento que utiliza el sensor del radar y uno igual, pero utilizando los sensores de distancia. Se puede ver en la *Figura 45*.

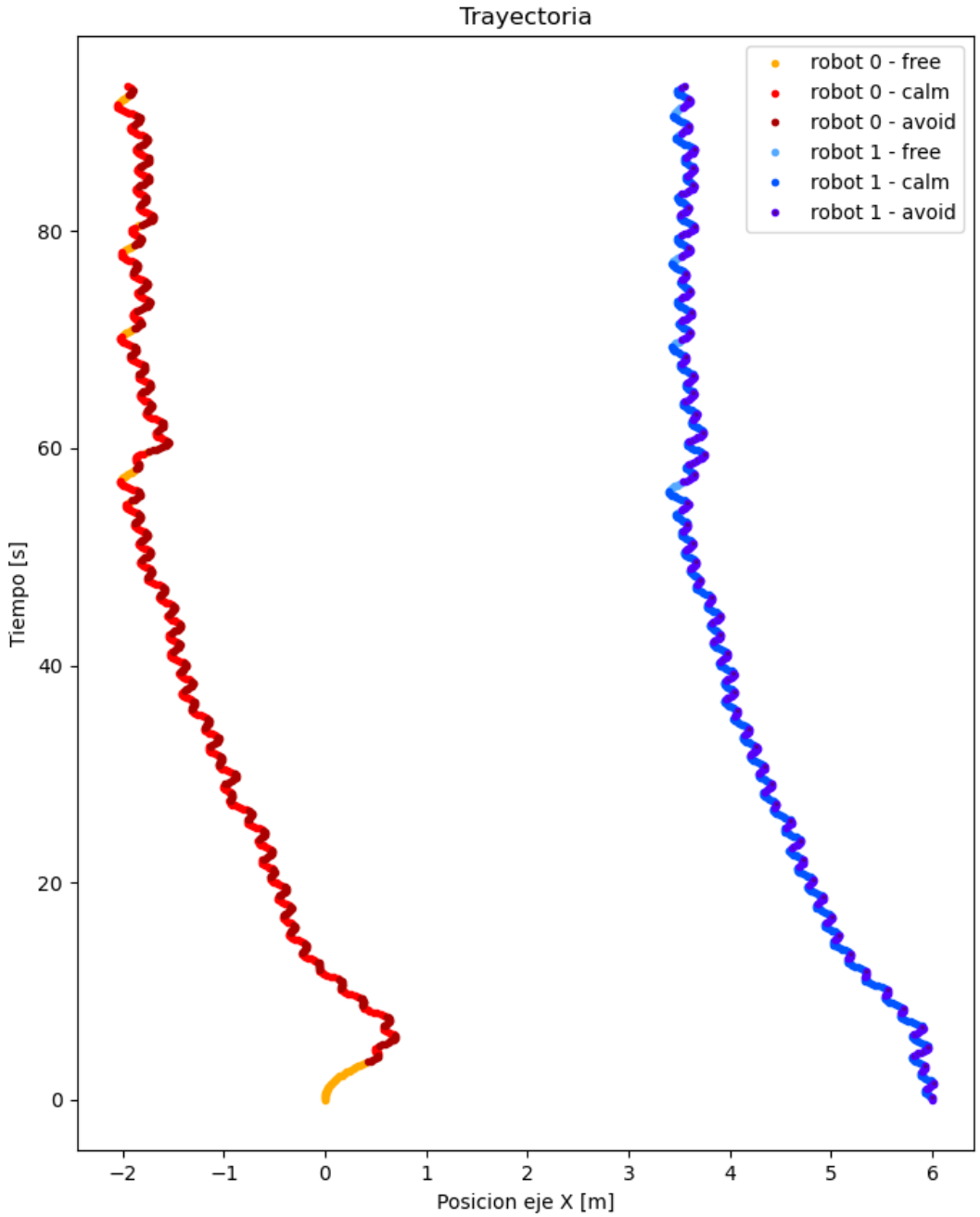


*Figura 45: Avoid Obstacle en un eje múltiples robots, utilizando los sensores de distancia*

Se observa que el resultado es completamente diferente, se puede ver como al tener lecturas constantes de la distancia en los ejes y al entrar un robot en estado esquivar se alterna la dirección en la que se mueve. El robot más cercano a la pared esquiva su obstáculo y antes de que el otro robot pueda alejarse lo suficiente este vuelve a generar que entren ambos en el estado esquivar, generando una trayectoria conjunta prácticamente idéntica. Esto va a evitar que los robots lleguen a colisionar, pero a su vez tampoco va a permitir que el funcionamiento sea el esperado.

En conclusión, la aproximación al algoritmo de navegación que permite esquivar obstáculos en tres fases es funcional, pero no permite gestionar etapas de esquive de forma muy seguida ya que no le da tiempo a cancelar la inercia. Otra posible solución sería tener un radar que tuviera más lecturas por iteración, de forma que detectara antes los obstáculos, ya que tal y como se ha visto en el caso de los sensores de distancia tener lecturas instantáneas sobre el entorno mitiga ese problema (aunque genera otros). También se podría considerar el cambio del enfoque general e implementar una compensación de inercia dinámica y automática durante todo el proceso.

## 4.4. EXPERIMENTO 4: CONTROL DE VELOCIDAD

Si se hace una reflexión sobre lo que realmente supone la inercia en el movimiento del robot, se puede llegar a la conclusión de que el campo que más se modifica es la velocidad. Si se le aplica una fuerza constante al robot, este tendrá una aceleración que se verá reflejada en un incremento progresivo de la velocidad. Mientras que, si se deja de aplicar esta fuerza constante de control, la deceleración no será igual de marcada que el cambio de fuerza, si no que decrecerá a un valor constante dependiente del medio hasta llegar a cero. Es por ello que si se quiere que el robot frene y baje su velocidad, el enfoque que se debe tener es el de aplicar una fuerza opuesta, no retirarla. De forma que estos pequeños ajustes se realicen de forma intercalada durante todo el proceso, no solo cuando haya una gran variación en las fuerzas y así tender siempre a la velocidad objetivo.

Para conseguir este objetivo, se puede utilizar el concepto expuesto en la Sec. 4.2, el cual propone adaptar las fuerzas en función a un objetivo. En ese caso, el objetivo era mantener la altura en un rango, mientras que en esta ocasión el objetivo es preservar la velocidad cerca de un valor concreto. Por suerte, en el primero se obligaba a trabajar en rangos por culpa de la flotabilidad, pero en el segundo, usando el mismo principio, se puede aproximar la velocidad directamente a una velocidad objetivo. La función que permite esto queda finalmente como:

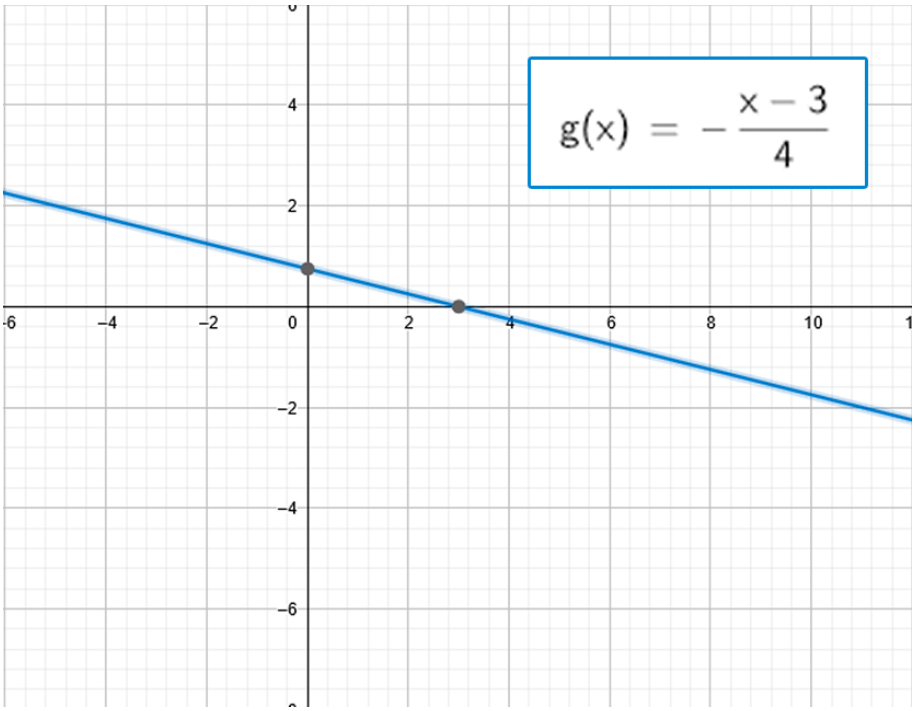


*Figura 46: Función para el control de velocidad*[2]

En el ejemplo de la función anterior, la velocidad objetivo es *3* y como se puede observar, en cuanto el robot tiene una velocidad superior al objetivo, se aplica una fuerza de frenado que crece cuanto más por encima se encuentre y, en el caso opuesto ocurre lo mismo cuanto más por debajo esta mayor es el

[2] Siendo $x$ la velocidad y $f(x)$ el control de salida

empuje. Se puede observar más claramente en un ejemplo dentro del simulador viendo la evolución de la velocidad, tal y como se refleja en la *Figura 47.*

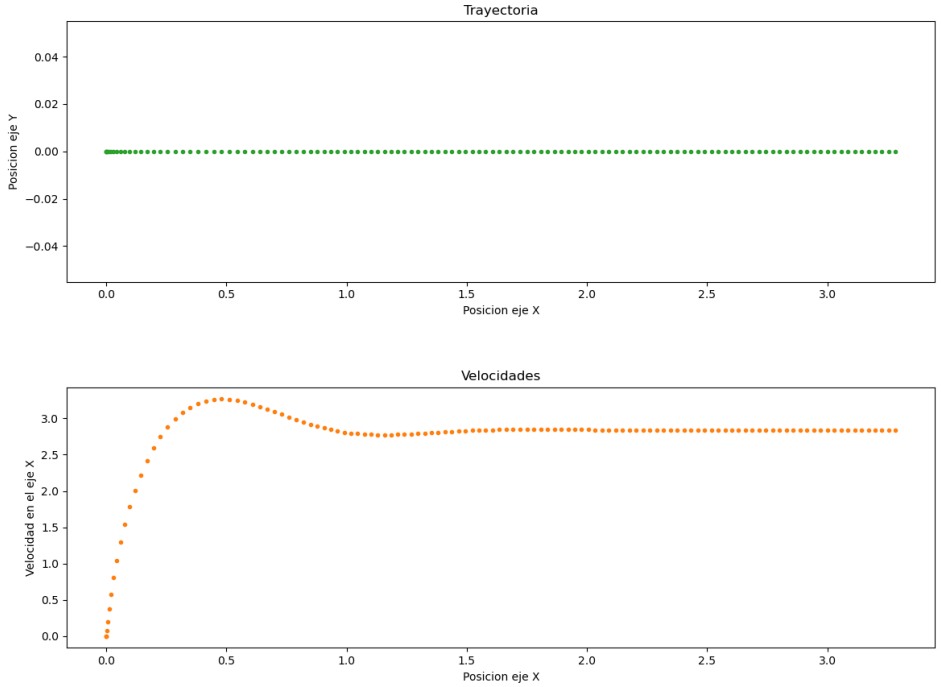


*Figura 47: Ejemplo aplicación adaptación de velocidad*

Según se puede apreciar, la velocidad tiene un incremento muy pronunciado al partir de parado el robot. Este cambio tan marcado genera una inercia, que una vez sobrepasada la velocidad objetivo se va compensando poco a poco con fuerzas intercaladas de frenado y aceleración. De ahí esa ligera oscilación después de la aceleración inicial, quedando después en un estado más estable con una velocidad ligeramente inferior que la objetivo. Dentro de estas adaptaciones, una vez se ha llegado a la región de estabilidad se podría modificar de nuevo la velocidad objetivo realizando una nueva compensación, que si requiere de menor cambio supondrá una variación menos pronunciada que en el caso anterior. Se puede ver representado en la *Figura 48.*

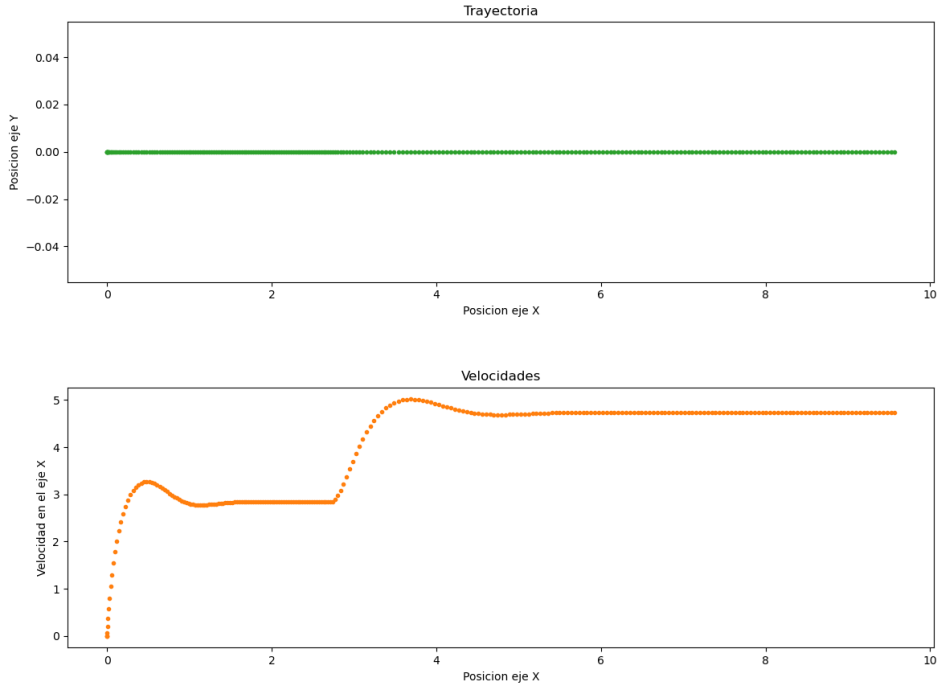


*Figura 48: Ejemplo de adaptación de velocidad doble*

Como se ha podido apreciar, esta segunda adaptación ha generado un rizado menos pronunciado debido a que ya contaba con una velocidad previa y la terminando con un estado de estabilidad entorno a la velocidad objetivo final, demostrando así la capacidad de adaptación del algoritmo.

Finalmente, como se ha mencionado al principio la fórmula, se aplica por ejes, y en los ejemplos anteriores solo se ha aplicado sobre un único eje. Este algoritmo debe de ser capaz de aplicar diferentes velocidades objetivo a cada eje y para ello utiliza la función definida en la Sec. 4.1 para aplicar controles independientes a los ejes $x$ e $y$ sin rotar el robot. Se puede observar la trayectoria y la velocidad por eje de la simulación en la *Figura 49.*

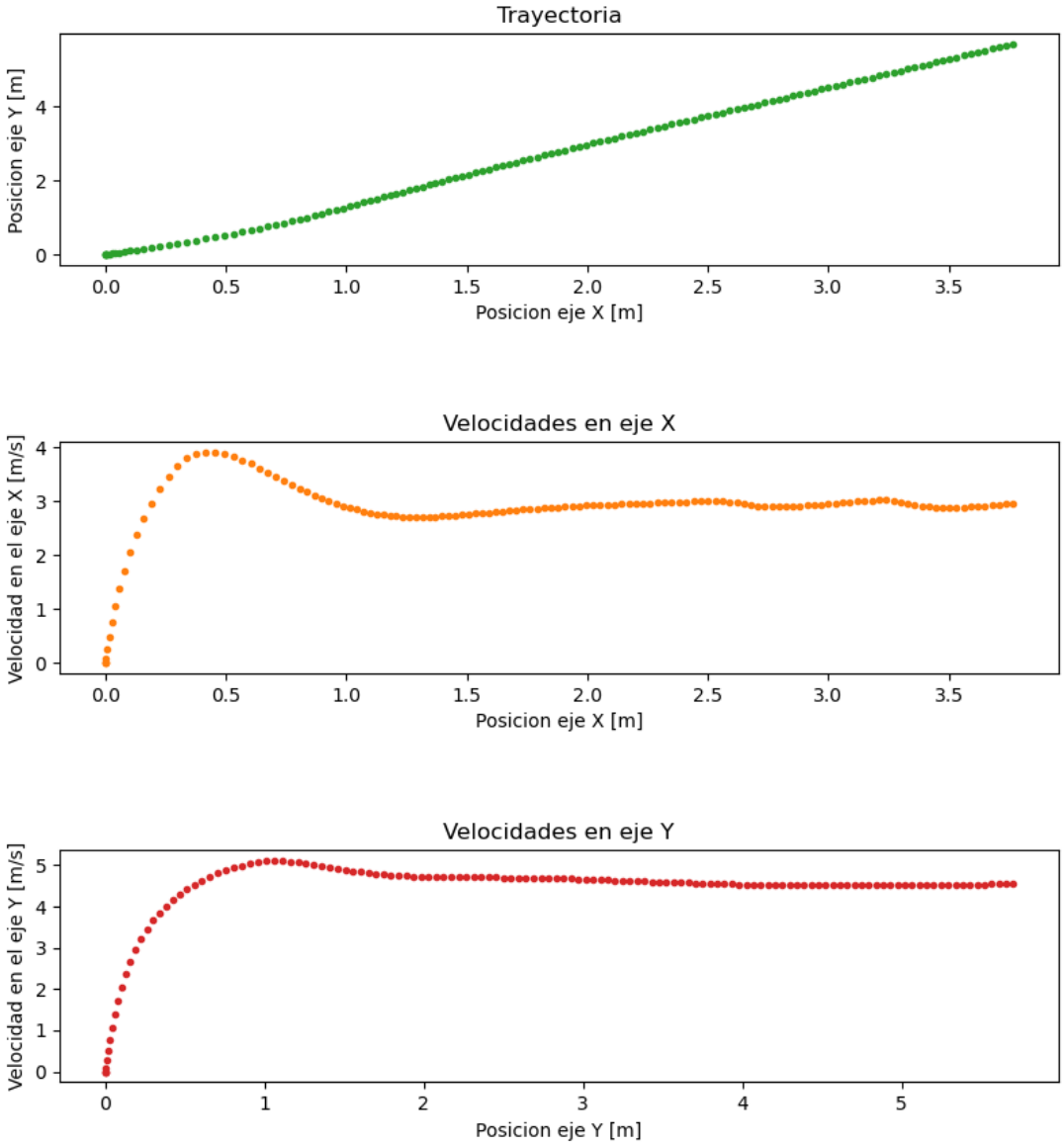


*Figura 49: Ejemplo de adaptación de velocidades en los dos ejes*

Se observa como en el eje $x$ la velocidad objetivo era 3 m/s y en el eje y 5 m/s. Se realiza una adaptación independiente, generando una trayectoria en la que el eje $y$ avanza 5/3 más que el $x$ tal y como se había supuesto que pasaría.

Como conclusión, se puede decir que el algoritmo de adaptación de velocidades desarrollado funciona correctamente para ambos ejes, partiendo de velocidades previas que no corresponden a la velocidad objetivo que se quiere llegar.

## 4.5. EXPERIMENTO 5: AVOID OBSTACLE V2

Teniendo la capacidad de contrarrestar la inercia de forma dinámica, se puede generar un nuevo enfoque para nuestro experimento final *avoid_obstacle*. Para ello, cuando un robot se acerque a un obstáculo ajuste su trayectoria para esquivarlo, y después adapte su velocidad al objetivo especificado, pero en la nueva dirección. La adaptación será más rápida y no se generarán situaciones tan forzadas como en el caso de la primera versión. De nuevo, utilizaremos de forma general el sensor del radar, pero para los experimentos finales se harán comparativas con los de distancia. Esta modificación del algoritmo anterior para la adaptación de velocidad va a suponer que la velocidad objetivo se ajuste a la dirección que el robot debe llevar y, a su vez, que sea capaz de gestionar obstáculos ejerciendo un empuje proporcional a la lectura del radar. En consecuencia, se crea un nuevo periodo de adaptación que involucra evitar el obstáculo y ajustar la velocidad en la nueva dirección. Se puede observar este comportamiento en la *Figura 50*, que representa evitado de una pared, en un movimiento a lo largo del eje $x$.

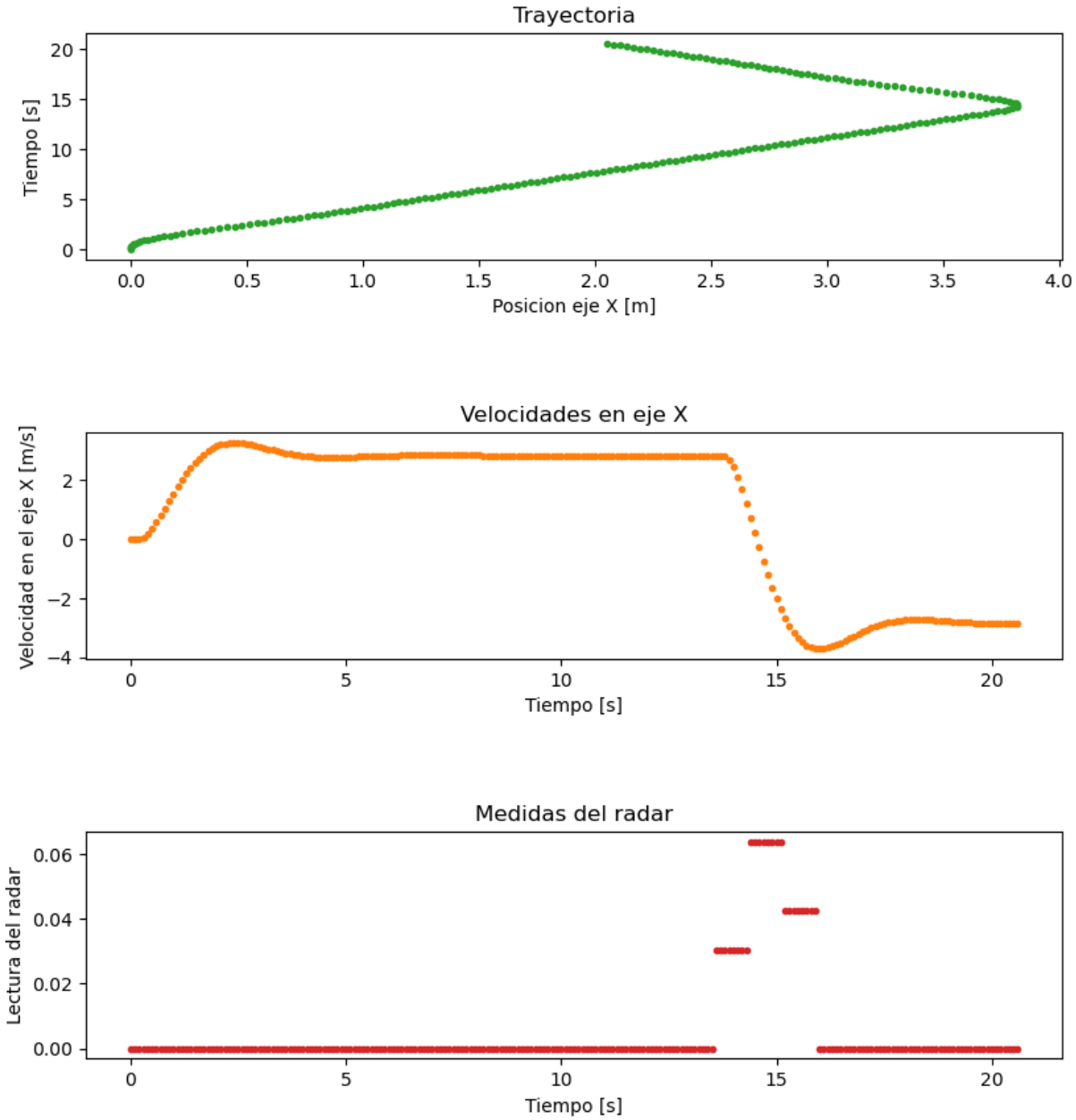


*Figura 50: Adaptación de velocidad durante avoid obstacle*

Una vez gestionada la adaptación de velocidades en un entorno más específico para el *avoid_obstacle*, se puede volver a plantear el experimento en un único eje. Como en la versión inicial ya se superó el problema de esquivar paredes en la piscina cuadrada dentro del eje x, se va a pasar directamente al caso en el que se introducen dos robots que se deben esquivar entre ellos también. Se puede ver el resultado del siguiente experimento en la *Figura 51*.

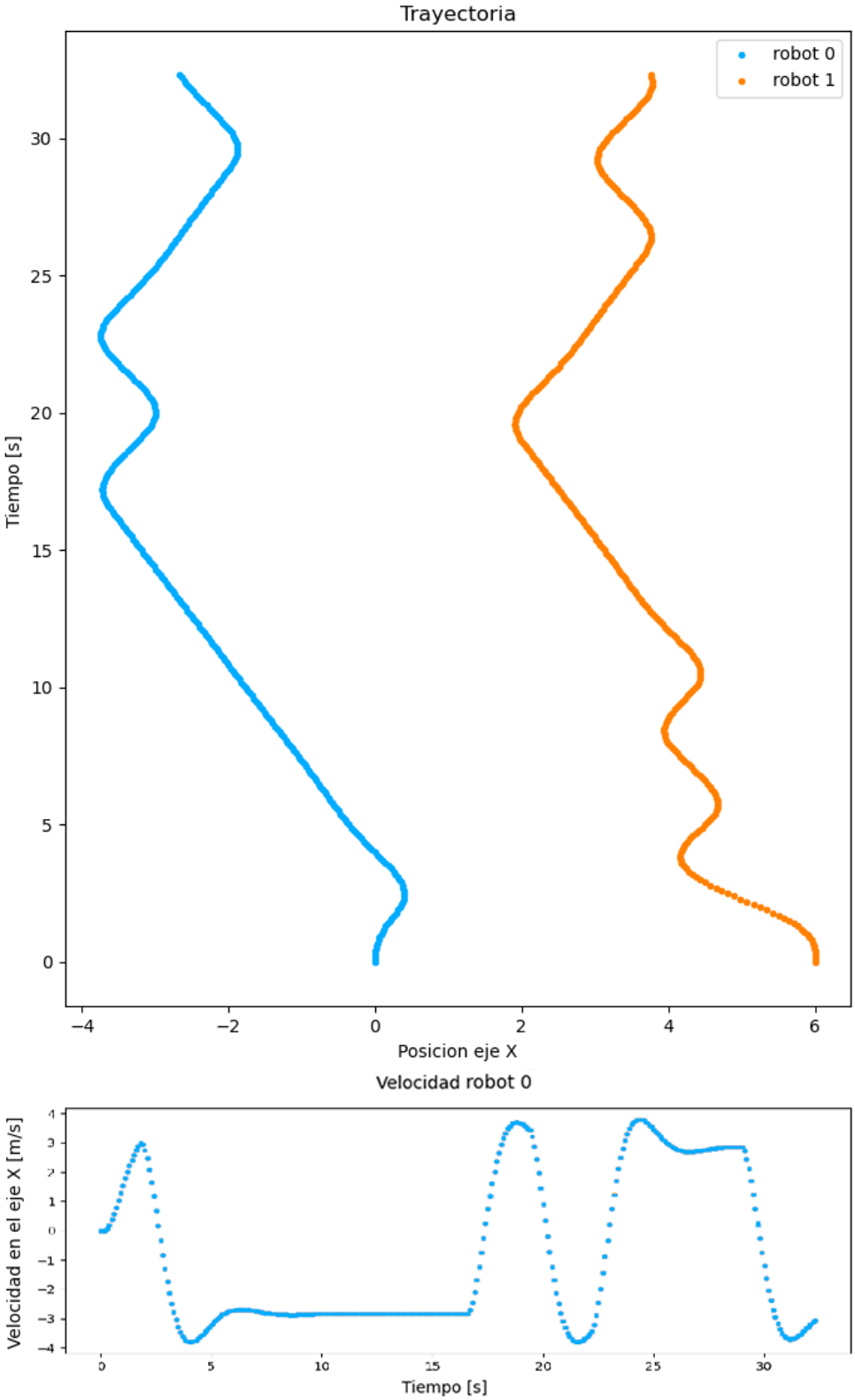


*Figura 51: Avoid obstacle v2 en un eje con dos robots*

Como se puede observar en las trayectorias, la distancia entre robots es mucho mayor gracias a la gran adaptación que ofrece el control de velocidad. Tal y como se puede reparar en la gráfica inferior, que nos representa la gestión de la velocidad en el primer robot, este enfoque no necesita un tiempo de calma con una duración determinada. Si se realiza un cambio de dirección o se empieza a evitar un obstáculo, este directamente se adapta a las nuevas velocidades requeridas hasta que entra en la fase de estabilidad.

Como conclusión en este apartado del experimento, esta nueva versión es mucho más versátil, permitiendo adaptaciones en menor tiempo. Lo cual va a proporcionar unos márgenes de seguridad ante colisiones mucho mayores que en el caso anterior. Sin embargo, esto no termina con el experimento. Para obtener el resultado que se buscaba, se debe tener este funcionamiento en ambos ejes y con más robots dentro de la piscina.

Para esta adaptación a dos dimensiones, al igual que en la Sec. 4.3, la gestión de cada eje va a ser independiente, tanto en el cálculo de velocidades y su adaptación, como la fase en la que se encuentra y se va a priorizar cuando uno de los ejes este esquivando, de forma que: si ambos están esquivando, la velocidad objetivo sea la determinada en ambos ejes, si uno está esquivando y otro no, la velocidad objetivo será solo la que busca el eje que esquiva y finalmente si ambos están en navegación libre, se busca también la velocidad determinada en cada eje. Con esta relación entre los ejes para probar su funcionamiento se van a situar cuatro robots en posiciones iniciales diferentes dentro de la piscina cuadrada. Tras un tiempo de simulación se han obtenido las trayectorias mostradas en la *Figura 52*.

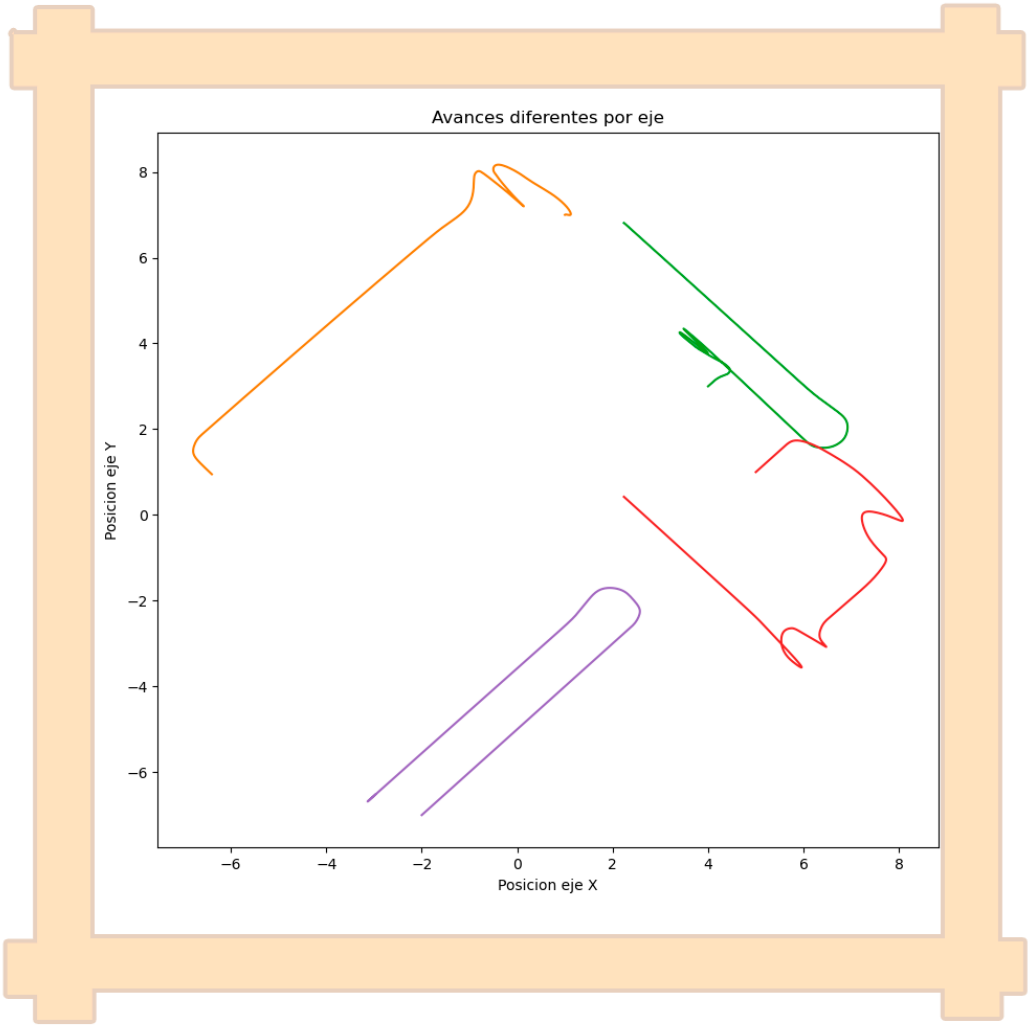


*Figura 52: Trayectorias avoid obstacle piscina cuadrada*

Aunque algunas trayectorias pasen por los mismos puntos, no lo hacen en el mismo instante temporal. Por lo que no hay ninguna colisión entre los robots o con el entorno, lo que muestra un buen funcionamiento al algoritmo. Por lo tanto, se puede suponer que nuestra modificación para la gestión de los dos ejes es correcta. Si se observa de nuevo la gráfica, también se puede ver como no solo no hay colisiones, sino que hay patrones muy marcados sobre la distancia que muestran las trayectorias en ciertos tramos, lo que supone esta distancia de seguridad que deja este nuevo algoritmo. Sin embargo, el radar tiene 8 estados sobre los que se realizan lecturas, esto permite que si un robot se sitúa en la posición correcta, se encuentre entre dos de esos estados siendo un punto ciego del radar. Pero al tener una buena adaptación, este algoritmo puede permitir la existencia de estos puntos ciegos y tomar acción con cierto retraso.

Para seguir poniendo aprueba el comportamiento del algoritmo ante ciertas situaciones se va a cambiar el mapa por el mapa octogonal. Con el fin de generar trayectorias más complejas y forzar la toma de decisiones y la capacidad de adaptación del algoritmo. Para el resto del entorno se mantendrán el resto de los parámetros y 4 robots generados en las mismas posiciones que en el caso anterior.

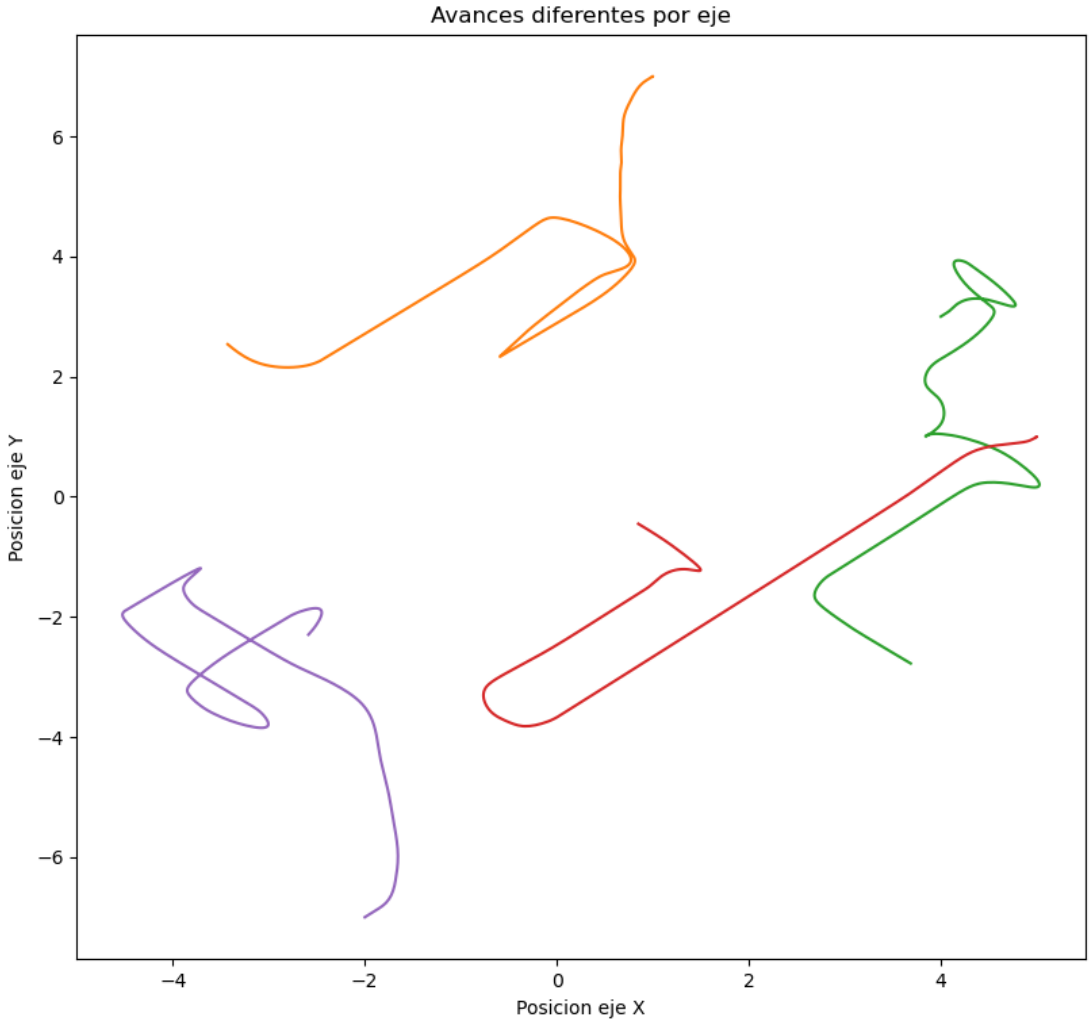


*Figura 53: Trayectorias avoid obstacle piscina octogonal*

En este caso no se ha representado el borde de la piscina porque la interacción con la misma es muy limitada debido a sus dimensiones (en el eje $x$ y en el $y$ tienen unas dimensiones de 24m en cada eje). Al igual que en el experimento anterior, se encuentra un funcionamiento sin colisiones y, como norma general, manteniendo distancias entre robots (a menos que se sitúen en estos puntos ciegos que se

menciona anteriormente). A nivel general se puede decir que el funcionamiento es también correcto y que la adaptación del algoritmo cumple los requisitos que se buscaban en un primer momento.

Finalmente, a modo de comparativa, al igual que se hizo con la primera versión, se va a estudiar este último escenario pero sustituyendo el sensor del radar por los de distancia. Es importante notar que los sensores de distancia están situados uno por eje, de forma que en el plano $x$ e $y$ existen la mitad de puntos de lectura, creándose más puntos ciegos de los que había antes. En la *Figura 54* se muestran las trayectorias.

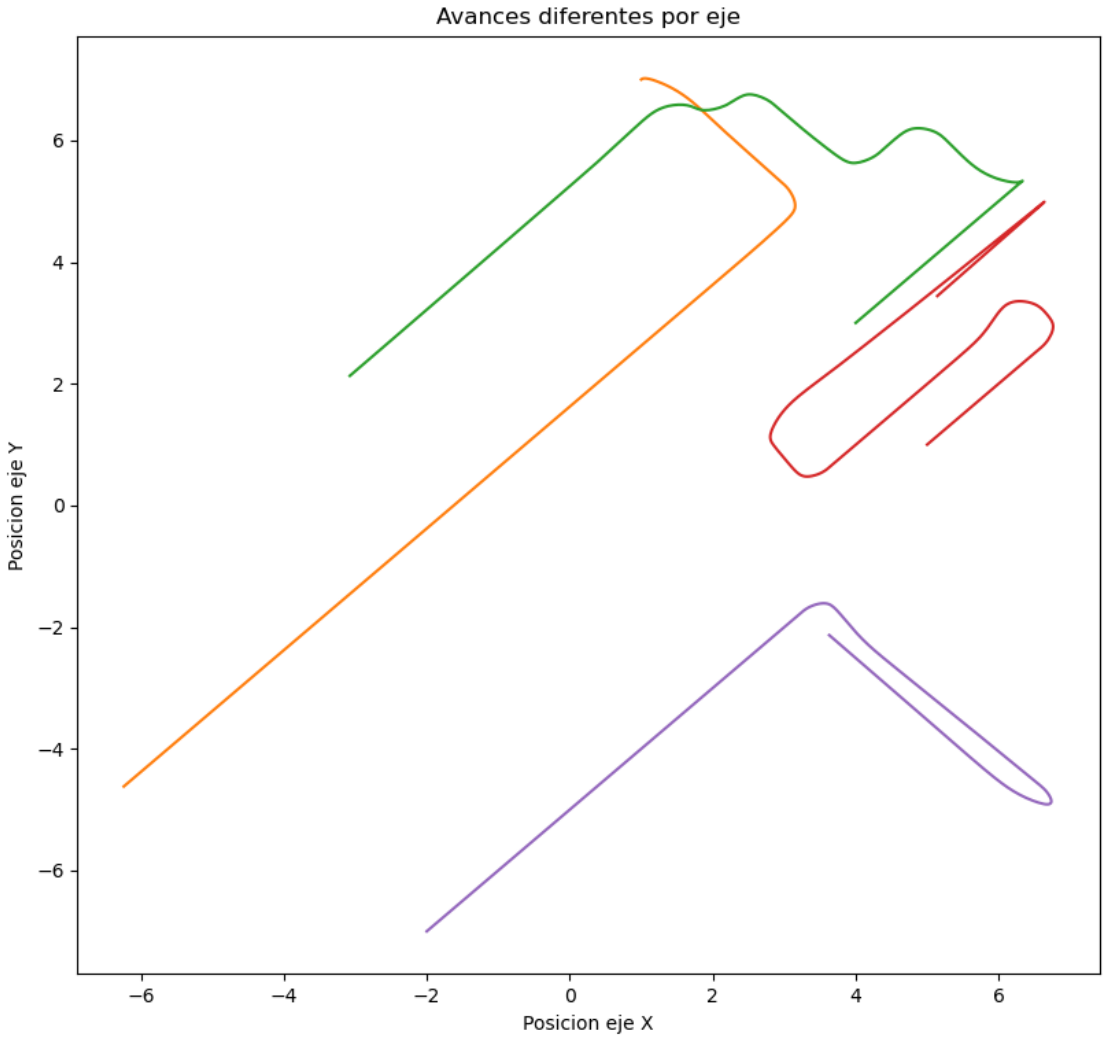


*Figura 54: Trayectorias avoid obstacle piscina octogonal con sensores de distancia*

Como se puede observar, el cambio de sensores no ha generado ninguna colisión, por lo que el funcionamiento a nivel general sigue siendo correcto. Pero sí que se aprecia un cambio de comportamiento bastante pronunciado, como los puntos ciegos han aumentado hay trayectorias que se sitúan más cerca unas de otras sin interferir entre ellas. A su vez también se ve como los comportamientos son más lineales, ya que al tener información concreta todo el rato de todos los ejes en cuanto algo entra en el camino se toma acción sobre ello.

En conclusión, sobre el experimento se puede ver que el funcionamiento del nuevo enfoque con control de velocidad funciona mucho mejor que en la anterior versión. Esto nos permite una gran adaptación a los diferentes mapas y número de robots del entorno. De forma que se ha llegado al objetivo final de tener un *avoid_obstacle* completamente funcional en ambos ejes y con múltiples robots en el entorno.

# 5. CONCLUSIONES Y LÍNEAS FUTURAS

## 5.1. CONCLUSIONES

En este TFG se ha realizado el diseño, implementación y evaluación de un simulador acuático para los robots y con el fin de realizar pruebas de funcionamiento sin arriesgar los robots reales. Para el desarrollo del simulador se ha utilizado el lenguaje de programación *Python* [2] apoyado con *OpenGL* [3], con el cual se ha conseguido implementar una visualización en 3d del entorno generado y sobre el cual los robots van a ser capaces de interaccionar gracias a los sensores que tienen. Para que los robots tengan un funcionamiento lo más parecido a la realidad se ha aislado completamente el controlador del simulador, de forma que este solo puede acceder a datos de entrada de los sensores y generar una reacción de salida por medio de los actuadores, tal y como ocurriría en un robot real. A su vez el simulador también permite configuraciones rápidas por vía de ejecución o por archivos de configuración. Esto otorga una gran versatilidad a la hora de realizar pruebas modificando el entorno, pudiendo también exportar los datos de estas pruebas y poder procesarlos adecuadamente, para su uso en aplicaciones IOT o el estudio de ciertos comportamientos.

En lo respectivo a la puesta a prueba del simulador, se puso como objetivo desarrollar un experimento donde el robot navegue esquivando obstáculos y en base a dicho propósito se generaron experimentos secundarios que permitían aproximar más la viabilidad de concluir exitosamente la función. El primer experimento se basa en un análisis de la movilidad del robot e interacción con el entorno, siendo el objetivo encontrar los controles que permitan mover sin rotar al robot en todos los ejes. Tras una evaluación de las fórmulas que relacionan las fuerzas de control con las fuerzas finales aplicadas a cada eje, se consiguió obtener este objetivo. Además, se pudo evaluar el peso que tenía la inercia dentro de las interacciones que se generaban con el entorno.

En el segundo experimento, se ha planteado un control de flotabilidad que permitía una estabilidad entorno al eje $z$ por parte del robot. De esta forma, en los posteriores experimentos se han podido desarrollar de forma específica los controles en los ejes $x$ e $y$. Para conseguir este objetivo, se ha relacionado la flotabilidad del robot con las fuerzas generadas por los motores, llegando a una función que genera un margen estable en cierto rango de alturas. Esta estabilidad implica un movimiento de oscilación entorno a una altura de equilibrio y pudiendo clasificar como mejor altura basándose en la amplitud de la oscilación y margen de seguridad con los extremos, el punto medio del margen.

Respecto al tercer experimento, se ha realizado una primera aproximación del experimento final. Para ello, se ha planteado una compensación de la inercia empleando una fase de calma, en la cual se aplica una fuerza contraria a la que tiene la inercia, decrementándola de forma constante para acabar con un movimiento estable. Tras el análisis de los resultados se vio que esta fase al reducir a la inercia de forma constante requiere de un tiempo de adaptación elevado, que de no ser respetado genera colisiones con el entorno. Es por ello que esta aproximación tuvo un resultado fallido.

Como solución al fallido experimento 3, se replanteo la necesidad de desarrollar un controlador que regulara la inercia de forma dinámica, o lo que es lo mismo un controlador de velocidad que permitiera ajustar la misma haciéndola estable a lo largo del proceso. Precisamente, en el experimento 4 se ha analizado el comportamiento de la inercia y se ha conseguido crear dicho controlador de velocidad, que tiene como parámetro la velocidad objetivo que se quiere mantener. Como consecuencia, se ha mejorado incluso el comportamiento esperado con la aplicación de este control de forma independiente a cada eje, permitiendo controlar la velocidad en específico de cada uno.

Finalmente, el experimento 5 se ha abordado nuevamente el objetivo final fallido anteriormente. En este caso se ha eliminado la fase de calma y se ha sustituido por el control de velocidad desarrollado en el experimento 4. Además este, se adaptó para que al detectar un obstáculo se aplicase una velocidad proporcional a como de cerca esta de este, en sentido contrario, y posteriormente regule su velocidad al objetivo en el sentido que lleve. En este caso, esta aproximación fue exitosa y se fue capaz de aplicar en ambos ejes con múltiples robots, generando en todos estos escenarios comportamientos sin ninguna colisión.

## 5.2. LINEAS FUTURAS

Tal y como se ha mencionado en repetidas ocasiones este simulador corresponde a lo que es la versión inicial de un proyecto mayor, es por ello que se espera un mayor desarrollo a corto plazo. Se planea una mejora tanto visual como de rendimiento, además de ampliar las posibilidades que se ofrecen a los usuarios del simulador a la hora de generar algoritmos para el funcionamiento del robot.

Uno de los principales campos sobre los que se espera una próxima investigación es sobre la implementación de algoritmos evolutivos con el uso de redes neuronales. *Python* [2] es uno de los lenguajes más potentes en lo referente al *machine learning*, contando con múltiples módulos muy especializados y usados en el campo, como puede ser *TensorFlow* [10], *Keras* [15] *o Pytorch* [16] entre otros. Esto facilita ampliamente el desarrollo de este tipo de algoritmos, por lo que es bastante interesante que uno de los próximos objetivos en el desarrollo del simulador sea la implementación de los mismos.

Por otro lado, a nivel eficiencia dentro del simulador, se puede observar cómo *Python*, en lo ajeno a la programación de controladores o desarrollo de redes neuronales, supone una limitación en la propia eficiencia. Esto implica que uno de los siguientes pasos a realizar sea convertir el simulador a otro lenguaje que tenga buena gestión de objetos y, a su vez, mejor velocidad de ejecución (como podría ser C++). Se generaría así un simulador híbrido que combine los mejores puntos de *Python*, solventando las carencias que puede tener en otros campos.

En lo respectivo al propio simulador, hay un campo que sería importante mejorar para aumentar también así su eficiencia. Dentro de los sensores de distancia que tienen los robots, a la hora de calcular las intersecciones con las superficies del entorno, esta versión calcula muchas más posibles intersecciones de las que realmente hay, realizando un gasto de recursos importante. Para solventar esto se debería agregar un filtro más preciso, calculando intersecciones solo cuando se sepa que van a ocurrir. Para lograr esto simplemente se debe crear un algoritmo que compruebe que el vector del sensor de distancia se encuentra en el rango generado por los vectores que unen el centro del robot con los vértices de la superficie evaluada. De esta forma, el número de superficies que se comprobarían sería mucho menor y se ganaría tiempo de ejecución.

Un campo interesante dentro del desarrollo del robot que sería bastante importante es la capacidad que debería tener el robot para mostrar información durante la ejecución que en la mayoría de los simuladores se suele recrear con un *LED*. En este caso podría implementarse algo similar que tuviera un controlador asociado y permitiera enviar una señal visual desde el controlador. Ello permitiría una comunicación directa, que no pase por el uso de un terminal o exportación de datos.

En lo referente al apartado visual, al usar *OpenGL* [3] se cuenta con un amplio número de proyectos que implementan simulaciones del agua y de fluidos, por lo que sería interesante la evaluación del cambio del plano que simula el agua dentro del entorno por un fluido con sus propias mecánicas [17]. Sin embargo, tal y como se ha mencionado, esta idea solo merece la pena si el margen eficiencia del simulador bueno ya que sino no sería aconsejable gastar recursos en el renderizado del mismo. Lo que si se podría hacer es tener la capacidad de seleccionarlo en el modo eficiencia, que de momento únicamente carga modelos 3d con menos polígonos.

Finalmente, uno de los campos con más margen de mejora en el simulador para ser lo más realista posible, es la capacidad de generar entornos más complejos que piscinas. Se podría partir de un objeto en 3d que represente el entorno, siendo los polígonos que lo forman las superficies sobre las que los sensores de distancia calculan las intersecciones con el medio. Este enfoque tiene dos problemas principales, el primero de ellos es que el mapeado del objeto va a suponer un algoritmo que analice la estructura propia del archivo y sea capaz de determinar qué puntos son los vértices de una superficie y el segundo es la diferencia que va a suponer a nivel superficies a evaluar, lo que va a requerir de una previa mejora de la eficiencia.

En conclusión, se puede ver como la cantidad de líneas que hay para abordar en el futuro es muy amplia y variada. Pero la principal sobre la que se deben basar todas las demás es una mejora del

rendimiento mediante una optimización de los sensores y del lenguaje en el que se basa el proyecto. Una vez desarrollado ese campo, las posibilidades son mucho más amplias pasando desde mejoras visuales del simulador, hasta la complejidad de algoritmos para desarrollar en los controladores e incluso mejorando la veracidad de los entornos en los que se puedan probar. Finalmente, se debe tener en cuenta que estas son las líneas futuras en lo respectivo al propio simulador. Sin embargo, al ser un proyecto conjunto habrá líneas de mejora independientes a nuestro desarrollo que mejoraran el propio simulador, como el desarrollo de nuevos modelos dinámicos, incorporación de fuerzas externas como corrientes y equipos especializados en el desarrollo de controladores que trabajaran en el propio simulador entre otros.

# 6. BIBLIOGRAFÍA


[1] «BlueROV2,» QSTAR Robótica Submarina, [En línea]. Available: https://bluerov.es/.

[2] «Python,» Pyhon, [En línea]. Available: https://www.python.org/about/.

[3] «OpenGL,» [En línea]. Available: https://www.opengl.org/.

[4] «PyGame,» [En línea]. Available: https://www.pygame.org/wiki/about.

[5] «Matlab,» MathWorks, [En línea]. Available: https://es.mathworks.com/products/matlab.html.

[6] S. E. M. Rozas, «Modelado y simulación de robots terrestres para la inspección del alcantarillado,» 2018. [En línea]. Available: https://biblus.us.es/bibing/proyectos/abreproy/71113/fichero/TFM-1113-MARTINEZ.pdf. [Último acceso: 2022].

[7] A. L. Christensen y M. Dorigo, «Evolving an integrated phototaxis and hole-avoidance behavior for a swarm-bot,» 2006. [En línea]. Available: https://www.researchgate.net/publication/229018915_Evolving_an_integrated_phototaxis_and_hole-avoidance_behavior_for_a_swarm-bot.

[8] E. C. a. Y. Bai, «PyBullet, a Python module for physics simulation for games, robotics and machine learning,» 2016--2022. [En línea]. Available: http://pybullet.org.

[9] R. S. Arranz y Á. Gutiérrez, «Evolution of situated and abstract communication in leader selection and borderline identification swarm robotics problems,» 11 8 2018. [En línea]. Available: http://www.robolabo.etsit.upm.es/publications/TFM/TFM_RafaelSendraArranz2.pdf. [Último acceso: 2022].

[10] «TensorFlow,» Google, [En línea]. Available: https://www.tensorflow.org/?hl=es-419.

[11] F. Pardo, «A Deep Reinforcement Learning Library for Fast Prototyping and Benchmarking,» 19 Mayo 2021. [En línea]. Available: https://arxiv.org/pdf/2011.07537.pdf.

[12] R. Mendonça, P. Santana, F. Marques, A. Lourenço, J. Silva y J. Barata, «Kelpie: A ROS-Based Multi-robot Simulator for Water Surface and Aerial Vehicles,» 16 Octubre 2013. [En línea]. Available: https://ieeexplore.ieee.org/abstract/document/6722374.

[13] «ROS,» [En línea]. Available: https://www.ros.org.

[14] G. Minghao, «Geometry3D,» [En línea]. Available: https://github.com/GouMinghao/Geometry3D.

[15] «Keras,» [En línea]. Available: https://keras.io/.

[16] «Pytorch,» [En línea]. Available: https://pytorch.org/.

[17] E. Wallace, «WebGL water,» 2016. [En línea]. Available: https://madebyevan.com/webgl-water/.

# ANEXO A: ASPECTOS ÉTICOS, ECONÓMICOS, SOCIALES Y AMBIENTALES

## A.1 INTRODUCCIÓN

A lo largo de este apéndice, se van a tratar los impactos potenciales que este TFG puede tener en términos éticos, económicos, sociales y ambientales. Sin embargo, al ser un proyecto que desarrolla un simulador que todavía no ha sido extendido al público, ni utilizado en entornos de desarrollo que tengan los robots reales, va a ser complicado realizar estas estimaciones. Por lo que el enfoque que se dará en los siguientes apartados será en función a los datos teóricos obtenido a lo largo del TFG y a la proyección que puede tener en un futuro.

## A.2 DESCRIPCIÓN DE IMPACTOS RELEVANTES RELACIONADOS CON EL PROYECTO

### A.2.1 IMPACTO SOCIAL

La robótica submarina, presenta una serie de usos que pueden ser realmente importantes para la sociedad. Uno de los más directos es la ayuda que puede dar a los submarinistas en tareas de búsqueda y rescate, permitiendo llegar de forma más rápida a lugares que son muy forzados para el ser humano. También por otro lado, puede tener aplicaciones inmediatas en tareas de transporte, desarrollando cada vez más el concepto de drones submarinos. A su vez, también puede ser interesante, los posibles usos de exploración que se les pueda dar, ya que hoy día el fondo marino sigue siendo un terreno desconocido en su mayoría y así muchos más ejemplos.

### A.2.2 IMPACTO AMBIENTAL

Aunque en la mayoría de casos el desarrollo industrial genera un impacto ambiental negativo, la robótica submarina puede tener grandes aplicaciones dentro de la limpieza de océanos, pudiendo crear algoritmos de control que hagan que los robots tengan como objetivo buscar basura y recogerla. Además de poder enfocarse con comportamientos colectivos, permitiendo limpiar regiones de forma efectiva, adaptándose a las mareas y evaluando su entorno para optimizar el proceso.

### A.2.3 IMPACTO ECONOMICO

Tal y como se menciona en la introducción Sec. 1.1, gran parte del auge que está teniendo el sector robótico submarino es debido al crecimiento de la economía azul, que muestra lo importante que pueden ser los océanos para la economía global y lo importante que es la limpieza de los mismos. Además, vcomo se ha mencionado en el Impacto social, estos robots van a poder automatizar muchos procesos forzados que hoy en día tienen que realizar humanos, sustituyendo estos trabajos, por unos de mantenimiento de estos robots en mejores condiciones y aumentando la productividad de la tarea iniciales gracias a los robots.

### A.2.3 IMPACTO ÉTICO

Dentro del impacto ético, en principio no hay uso directo del simulador en el ámbito militar, pero este desarrollo de la robótica submarina va a facilitar de forma más directa o indirecta ciertos avances en este tipo de ámbitos. Pero el simulador no tiene ningún tipo de implicación ética directa en ningún sentido.

## A.3 CONCLUSIONES

Como se ha podido valorar en la sección anterior, en normas generales, el impacto del simulador ayuda a desarrollar un campo que trae bastantes beneficios en todos los ámbitos. Aunque en el corto plazo realmente no tenga impactos tan directos como ya se ha explicado.

# ANEXO B: PRESUPUESTO ECONÓMICO

| COSTE DE MANO DE OBRA (coste directo) | Horas | Precio/hora | Total |
|---|---|---|---|
| | 500 | 15 € | **7.500 €** |

| COSTE DE RECURSOS MATERIALES (coste directo) | Precio de compra | Uso en meses | Amortización (en años) | Total |
|---|---|---|---|---|
| Ordenador personal (Software incluido) | 2.000,00 € | 6 | 5 | 200,00 € |
| Otro equipamiento | | | | |

| COSTE TOTAL DE RECURSOS MATERIALES | 200,00 € |
|---|---|

| | | | |
|---|---|---|---|
| **GASTOS GENERALES (costes indirectos)** | 15% | sobre CD | **1.155,00 €** |
| **BENEFICIO INDUSTRIAL** | 6% | sobre CD+CI | **531,30 €** |

**MATERIAL FUNGIBLE**

| | |
|---|---|
| Impresión | **0,00 €** |
| Encuadernación | **0,00 €** |

| | | |
|---|---|---|
| **SUBTOTAL PRESUPUESTO** | | **9.386,30 €** |
| **IVA APLICABLE** | 21% | **1.971,12 €** |
| **TOTAL PRESUPUESTO** | | **11.357,42 €** |

# ANEXO C: MANUAL DE USUARIO

A lo largo de este apéndice se van a cubrir una serie de procedimientos necesarios para que un usuario corriente comience a usar el simulador en su propia máquina y pueda empezar a desarrollar controladores dentro del mismo. El simulador está disponible para los tres principales sistemas operativos (*Windows*, *Linux* y *Mac*). Sin embargo, como la instalación parte de *Python*, se desarrollará de forma genérica pudiendo aplicarse en cualquiera de estos.

## C.1 INSTALACIÓN DEL SIMULADOR

El requisito básico que todo dispositivo debe tener es contar con una versión estable de *Python*. Durante el desarrollo se ha utilizado la versión *3.10.2*, pero no debería haber problemas de compatibilidad con versiones anteriores superiores a la *3.8* o versiones más actuales. Posteriormente, antes de poder ejecutar el simulador se deberán instalar una serie de módulos asociados, utilizando la instalación vía *pip*, se deben ejecutar los siguientes comandos:

```
pip install PyOpenGL
pip install pygame
pip install Geometry3D
```

*Figura 55: Comandos instalación requisitos*

De esta forma obtenemos la última versión de los paquetes necesarios para ejecutar el simulador. Una vez cubiertos todos los requisitos, ya solo queda descargar y ejecutar el simulador. Para descargarlo, se puede directamente clonar por medio del siguiente repositorio **NAUTILUS** y así finalmente ejecutarlo mediante el siguiente comando:

```
python main.py
```

*Figura 56: Comando ejecución NAUTILUS*

Terminando así la instalación correspondiente del simulador.

## C.2 FLAGS Y CONFIGURACIONES RÁPIDAS

A la hora de ejecutar el simulador podemos hacer diferentes ajustes rápido que permitirán un uso más cómodo y dinámico del mismo. Podemos modificar el entorno del experimento, entre otros parámetros, sin necesidad de modificar nada del código. Los *flags* que tiene asociados el programa principal son los siguientes:

- *-novisual*: Permite ejecutar el simulador sin la interfaz gráfica, eliminando así todo el proceso relacionado con OpenGL e incrementando de esta forma la velocidad de ejecución de los experimentos.
- *-c <controller>:* El siguiente *flag* establece el controlador que se va a utilizar en todos los robots que sean generados en el entorno.
- *-o*: En este caso el *flag* habilita la exportación de datos sobre el experimento que se va a ejecutar.
- *-m <arena>*: En este caso el *flag* permite seleccionar entre las diferentes arenas con las que cuenta el simulador.
- *-ef*: Activa el modo *eficiente*, el cual carga modelos 3d más simples, mejorando así ligeramente la eficiencia del simulador.
- *-cf <configuration_file>*: Este *flag* nos permite seleccionar un archivo de configuración perteneciente a la carpeta *configuration_files*

En la *Figura 57* podemos ver ejemplos de ejecución con flags.

```
python main.py -ef -m prismaArena -o
python main.py -m cylinderArena -c AvoidObstacle -cf avoid_obstacle.json -o
```

*Figura 57: Ejemplos de ejecución NAUTILUS*

Por otro lado, la otra forma de realizar configuraciones es mediante el uso de archivos de configuración alojados en la carpeta *configuration_files*. Dentro de esta, podemos especificar mediante archivos *JSON* cuantos robots queremos que se generen y que estado inicial van a tener. Para ello, los archivos van a estar formados por un objeto que tiene un atributo *robots*, de tipo lista y dentro de la misma habrá objetos con los parámetros que tiene el robot del simulador (posición, rotación …) que permiten definir el estado inicial, tal y como podemos ver en la *Figura 58.*

```
{
    "robots": [
        {
            "position": [
                0.0,
                7.0,
                -1.0
            ]
        }
    ]
}
```

*Figura 58: Ejemplo archivo de configuración*

## C.3 PROGRAMACIÓN DE CONTROLADORES

En lo referente a la programación de controladores, ya que corresponde al desarrollo de algoritmos, se va a tener un enfoque sobre las herramientas con las que se cuenta dentro de los mismos, para así tener un mejor enfoque a la hora de desarrollar.

Dentro de los controladores, debido a la estructura sensor-controlador-actuador, solo se va a poder acceder a los sensores para obtener datos y mostrar una respuesta mediante el uso de los actuadores. El robot tiene definidos en esta versión inicial tres sensores, podemos ver la salida de las lecturas que da cada uno a continuación:

- $self.sensors$["gps"] → $[x, y, z]$: Facilita la posición del robot dentro del entorno.
- $self.sensors$["distancia"] → $[d_x, d_{-x}, d_y, d_{-y}, d_z, d_{-z}]$: Devuelve las lecturas de cada sensor de distancia, estando cada sensor situado en un eje.
- $self.sensors$["radar"] → $[d_{+x}, d_{+x+y}, d_{+y}, d_{-x+y}, d_{-x}, d_{-x-y}, d_{-y}, d_{+x-y}]$: El sensor radar esta configurado para hacer 8 lecturas y estas van a ser realizadas aplicando un *Yaw* positivo, generando ese orden en las mismas.

Por otro lado, el robot de momento solo cuenta con un actuador, el control, que permite modificar la variable de control que tiene el robot, esta representa la fuerza de los motores:

- $self.actuators$["control"].$apply([F_1, F_2, F_3, F_4, F_5, F_6])$ : aplica la respuesta reflejada en la nueva fuerza que tienen los motores.

Siendo estas las herramientas que tiene el controlador y con las cuales se deben desarrollar los algoritmos deseados. Se pueden ver una serie de ejemplos correspondientes a los experimentos desarrollados en este TFG dentro de la carpeta *controllers*.

### C.3.1 EXPORT

Dentro de los controladores, hay una función secundaria llamada *export* que permite reflejar ciertos datos obtenidos dentro de los *output_files* devolviéndolos como un diccionario. Estos datos, pueden ser desde lecturas directas de los sensores hasta cálculos más complejos como puede ser la velocidad o el propio control que se aplica en los actuadores. Es importante tener en cuenta que para poder enlazar estos datos a la función, tendremos que almacenar previamente, estas lecturas y estos cálculos como parámetros del objeto, podemos ver un ejemplo en la *Figura 59.*

```python
class ControllerExample(Controller):
    def __init__(self):
        super().__init__()
        self.lastReadGPS = []
        self.lastControl = [0, 0, 0, 0, 0, 0]

    def step(self):
        self.lastReadGPS = self.controllerOwner.sensors['gps'].read()

        print("----------------------------STEP----------------------------")
        print(f"POSICION: {self.lastReadGPS}")
        self.lastControl[4:] = [(self.lastReadGPS[2] + 2) / 30] * 2
        print(f"CONTROL: {self.lastControl}")
        self.controllerOwner.actuators['control'].apply([
            50 * val for val in self.lastControl])

    def export(self):
        baseData = super().export()
        baseData["gps"] = self.lastReadGPS

        return baseData
```

*Figura 59: Ejemplo de controlador con export*

# ANEXO D: MANUAL DE DESARROLLADOR

En este anexo se va a realizar una pequeña introducción a la estructura y al enfoque de desarrollo del proyecto, para que en caso de querer contribuir al proyecto se pueda obtener una visión general más clara y rápida. Para no repetir excesivamente lo comentado en el Cap. 3, se va a realizar una visión más práctica y resumida del propio simulador.

## D.1 DESARROLLO DE NUEVO OBJETOS DINÁMICOS

En caso de querer crear nuevos objetos dinámicos que sean mostrados en el entorno, se deberá definir un modelo 3d asociado para su correcta visualización, evaluar el tamaño del mismo, asignar una dinámica (ya sea *static*, *modelo0* o una propia del objeto) y definir el estado que va a caracterizar a este objeto.

Por ejemplo, en el caso de querer desarrollar un objeto *Luz*, podríamos implementar un modelo 3d en forma de esfera, definirle un tamaño 0 para que no tuviera designado un volumen y una dinámica *static* para que no tuviera cambios de estado. Finalmente, quedaría definir el estado, que partiría de la posición en la que se encuentra y la rotación (debido a que todos los *Objects* deben tenerla, pero en este caso es irrelevante), seguido de una intensidad que podría definir como de cerca hay que estar de la misma para que se vea y ya quedaría implementado un nuevo *DynamicObject* dentro del simulador.

## D.2 DESARROLLO SENSOR-CONTROLADOR-ACTUADOR

El robot, tiene definidos dos diccionarios que establecen los sensores y los actuadores, además de dos métodos para agregar a cada uno de ellos nuevos elementos. Estos diccionarios van a asociar un *Id* a cada sensor/actuador el cual será utilizado para acceder a estos elementos de forma más cómoda desde el controlador. Por lo que cada vez que se desarrolle un nuevo sensor/actuador, deberá agregarse al robot en su constructor para así estar disponible en los controladores. Podemos ver un ejemplo de esto en la *Figura 60.*

```
class Robot(DynamicObject):
    robotID = 0

    def __init__(...):
        # ...
        self.sensors = {}
        self.actuators = {}
        # ...

        self.addSensor('gps', SensorGPS(self))
        self.addSensor('distancia', GrupoSensores(self,
            [SensorDistancia(self, vector, distanceThreshold, environment)
                for vector in distanceVectors]))
        self.addSensor('radar', SensorRadar(
            self, distanceThreshold, 8, 1, environment))
        self.addActuator('control', ActuatorControl(self))

        # ...
```

*Figura 60: Estructura constructor Robot*

### D.2.1 SENSORES

En el caso de querer desarrollar nuevos sensores, se tendrá que definir en un primer momento qué valor quieren medir. Se debe tener en cuenta, que estos valores tienen que poder calcularse mediante parámetros propios del robot o evaluaciones del entorno, que son los campos con los que se van a poder inicializar estos sensores.

Dentro de la clase *Sensor* de la que heredan todos los sensores del simulador, queda establecido que el funcionamiento debe ser entorno a un método *read*, que cuando es llamado debe devolver la lectura que se espera del entorno. Sin embargo, esta función no es la única, también cuenta con un método *render* que se encarga de mostrar visualmente información extra en el simulador de forma condicional, se puede alternar mediante el uso del método *toogleMoreInfo*. Para que esta estructura siga un comportamiento correcto, se debe agregar en el archivo *controls.py* una tecla que tenga asignada la acción de llamar a *toogleMoreInfo*. Finalmente, por muy definida que este la estructura de los sensores, al ser objetos de *Python* se debe tener en cuenta las funcionalidades que esto ofrece, como la capacidad de crear sensores con memoria, almacenando las lecturas previas como un parámetro del propio sensor por ejemplo.

### D.2.2 ACTUADORES

A diferencia del desarrollo de sensores, los actuadores deben ir enlazados a una variable del objeto al que pertenecen, además, habitualmente, de forma directa o indirecta realizará cambios en el estado o en cierto aspecto visual del mismo. Uno de los ejemplos más claros, es la implementación de un *LED* en los robots, para el cual deberá crearse una variable de control del *LED*. Por ejemplo, una variable que si es 0 el *LED* esta apagado y si es 1 está encendido (no se asigna como *bool* para dar la opción de poner más colores al *LED*). Se puede crear un actuador cuyo método *apply*, definido en la clase *Actuator* de la que hereda, modifique esta variable. Sin embargo, por mucho que el desarrollo del actuador esté terminado, la implementación compleja va enlazada a la reacción que debe tener la variable dentro del objeto. En este caso, tendría que agregarse al método *render* del robot un condicional que dependiera de esta nueva variable y que renderizara un *LED* apagado o un *LED* encendido según corresponda.

## D.3 DESARROLLO DE MAPAS

En esta versión inicial, los mapas están formados por *piscinas* y esta sección se destinará a la explicación de cómo crearlas. Sin embargo, si se busca crear entornos más complejos y realistas, el enfoque será más externo y centrado en programas de diseño 3d.

El concepto de *piscina*, sobre el cual se han creado todos los mapas que contiene el simulador, se basa principalmente en un enfoque 2d que defina el contorno de la misma, que luego será extruido y mapeado con el objeto *Wall*. De forma que la función generadora del mapa sitúa las paredes y devuelve una lista con todos los objetos *Wall* utilizados para que estos luego sean incorporados al entorno, tal y como podemos ver reflejado en el código de la piscina octogonal:

```
def cylinderArena():
    mapObjects = []

    for level in range(0, 1):
        for step in range(8):
            mapObjects.append(Wall(
                position=Point(
                    *[0.0, 12.0, -2.0 * level],
                    *[0, 0, 45 * step]
                ).coordinates,
                rotation=[90, 0, 45 * step]
            ))

    return mapObjects
```

*Figura 61: Función generadora de la piscina octogonal*

Como se puede observar, el funcionamiento suele ser un bucle asociado a las alturas de la piscina y otro a la forma como tal. Además, se puede utilizar la clase *Point*, que es la utilizada para la gestión de los vértices de los objetos como se vio en la Sec. 3.1.3, permitiendo aplicar rotaciones a los puntos y así facilitando la creación de ciertas formas. Por ejemplo, en caso de querer crear una piscina hexagonal, simplemente habría que cambiar la rotación acumulada en cada *step* de 45 a 60 y ajustar la distancia al centro para quedar cerrada.

# ANEXO E: CAPTURAS DEL SIMULADOR

Durante este apéndice se van a mostrar diferentes figuras en las que se pueden ver diferentes fotos del simulador realizando múltiples experimentos.

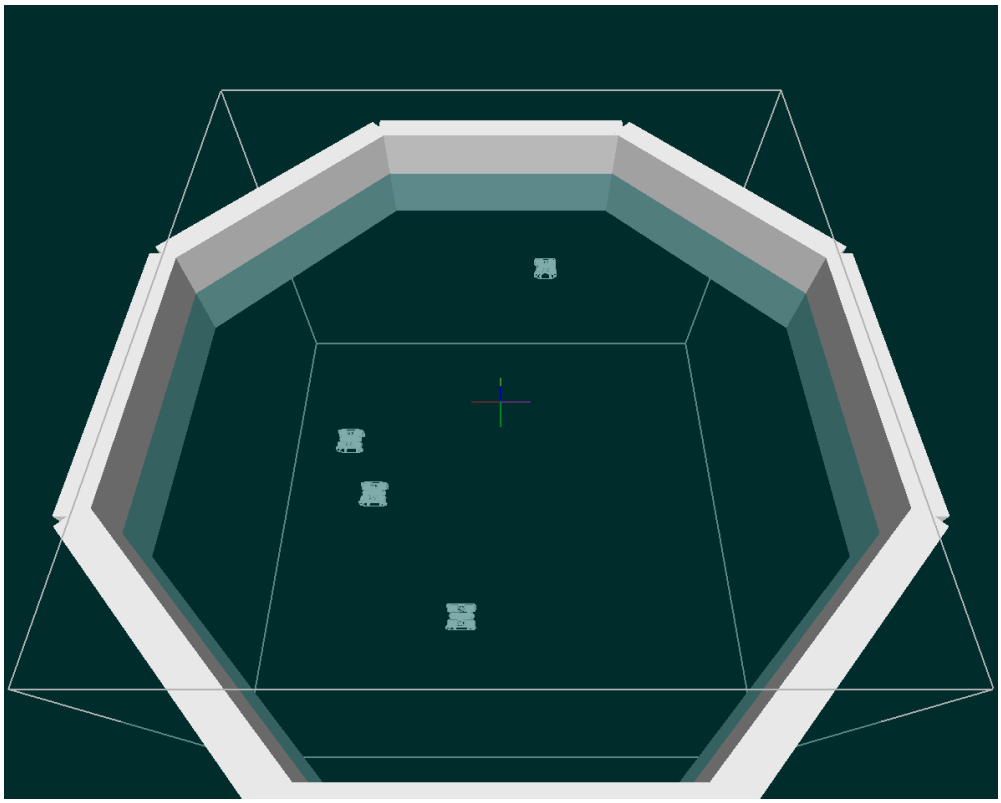

*Figura 62: Captura del simulador durante un experimento de avoid obstacle*

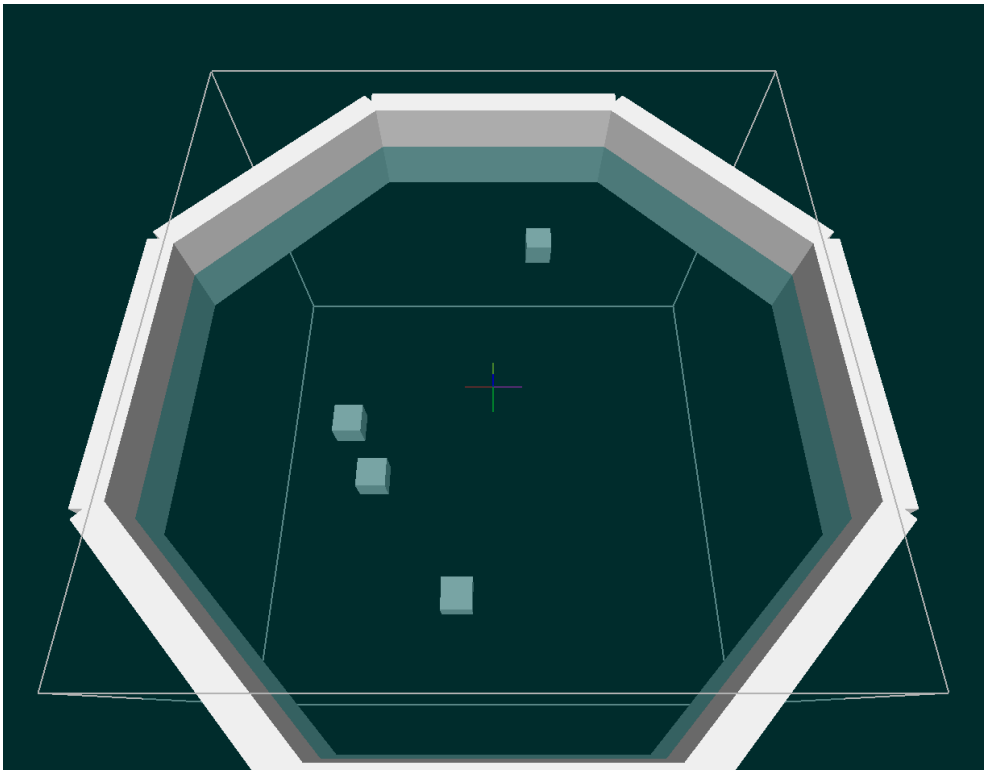

*Figura 63: Captura del simulador con el modo eficiencia activado*

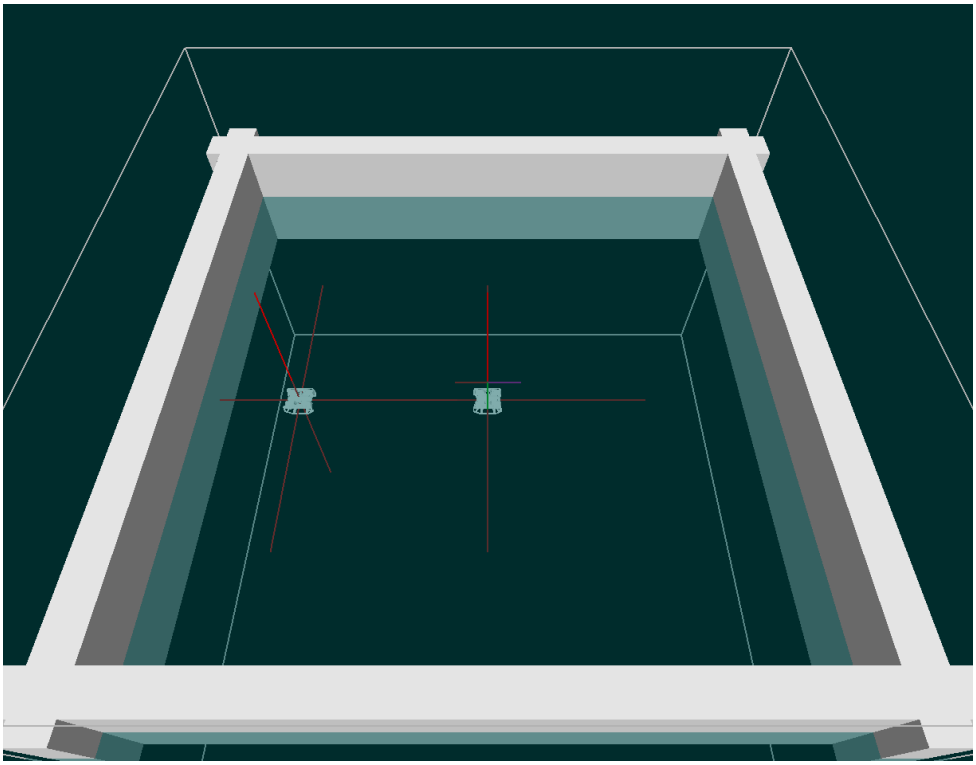

*Figura 64: Captura del simulador mostrando los sensores de distancia*