



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

*Desarrollo y validación de un sistema de simulación y de
enjambres de unidades de superficie autónomas aplicados a
operaciones militares*

Grado en Ingeniería Mecánica

ALUMNO: Manuel Ángel Varela García

DIRECTORES: José Antonio González Prieto
Ana Ogando Martínez

CURSO ACADÉMICO: 2024-2025

Universida_{de}Vigo



Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

*Desarrollo y validación de un sistema de simulación y de
enjambres de unidades de superficie autónomas aplicados a
operaciones militares*

Grado en Ingeniería Mecánica

Intensificación en Tecnología Naval

Cuerpo General

UniversidadeVigo

RESUMEN

En el contexto actual de la guerra entre Rusia y Ucrania, el uso de enjambres de vehículos no tripulados (USVs) se ha convertido en una herramienta de gran importancia de la defensa de unidades propias y aliadas. Este trabajo de fin de grado propone desarrollar un sistema de enjambre de USVs orientado a la defensa, controlado mediante un interfaz, diseñado y programado, para que al usuario le sea fácil dar órdenes y controlar al enjambre. La interfaz deberá ser capaz de gestionar varios USVs simultáneamente permitiendo al usuario dar órdenes específicas y monitorear las acciones que se llevan a cabo. Se investigarán y aplicarán algoritmos de inteligencia artificial y técnicas de control para optimizar el comportamiento del enjambre. Estos algoritmos permitirán que los USVs actúen de manera autónoma, mejorando la capacidad de respuesta y la efectividad del sistema en escenarios de defensa.

PALABRAS CLAVE

USV, Enjambre, Planificación, Rutas, Defensa

AGRADECIMIENTOS

La realización de este Trabajo de Fin de Grado ha sido un camino lleno de aprendizaje, esfuerzo y superación. A lo largo de este proceso, he contado con el apoyo de muchas personas que, de una u otra forma, han contribuido a que este proyecto sea una realidad.

En primer lugar, quiero expresar mi más sincero agradecimiento a mis directores del Trabajo de Fin de Grado, José Antonio y Ana, cuya orientación, paciencia y conocimientos han sido fundamentales para el desarrollo de este trabajo. Su apoyo constante y su forma de transmitirme sus conocimientos han enriquecido cada etapa del proyecto. José Antonio, Ana, muchas gracias.

A mis padres, por ser mi pilar inquebrantable, por su amor incondicional y por enseñarme, con su ejemplo el verdadero significado del esfuerzo, la perseverancia y la humildad. Gracias por inculcarme el valor de las cosas, no solo por lo que son, sino por lo que representan. Y, sobre todo, gracias por confiar en mí, incluso en aquellos momentos en los que hasta yo mismo dudaba. Gracias.

A mi novia, por ser mi compañera incondicional en este camino, por su paciencia infinita y su apoyo en los momentos de mayor desafío. Gracias por creer en mí, Rocío, por motivarme a seguir adelante y por recordarme, con tu amor y confianza, que siempre hay un motivo para dar lo mejor de uno mismo. Tu presencia ha sido mi refugio y tu aliento, mi impulso para superar cada obstáculo. Muchas gracias.

A mis amigos, quienes han estado a mi lado en los momentos de mayor presión, brindándome ánimo y distrayéndome cuando más lo necesitaba. Gracias por comprenderme, apoyarme y hacer que este camino haya sido más llevadero.

Quiero dedicar un agradecimiento especial a mi abuelo, quien, aunque ya no está, sigue presente en cada paso que doy. Su sabiduría, su bondad y su manera de ver la vida me han inspirado siempre. Su recuerdo ha sido un motor de motivación en los momentos difíciles, y este logro, de alguna manera, también es suyo. Ojalá pudiera compartir con él este momento, pero sé que estaría orgulloso.

Finalmente, agradezco a todas las personas que, de una forma u otra, han influido en mi formación y en la culminación de este proyecto. Sin su apoyo, este trabajo no habría sido posible.

A todos, gracias de corazón.

CONTENIDO

Contenido	1
Índice de Figuras	3
1 Introducción y objetivos	5
1.1 Contexto y motivación	5
1.2 Objetivos y alcance	6
1.3 Estructura de la memoria	6
2 Estado del arte	8
2.1 Contexto histórico	8
2.2 Drones en el ámbito militar	9
2.3 Escenarios de combate naval con drones y su clasificación	16
3 Desarrollo del TFG	18
3.1 Definición de escenarios	18
3.2 Generación de rutas geométricas	19
3.2.1 Ruta lineal	19
3.2.2 Ruta semicircular	21
3.2.3 Ruta sinusoidal	24
3.2.4 Ruta dientes de sierra	26
3.2.5 Ruta espiral	29
3.3 Navegando entre obstáculos	32
3.4 Rutas óptimas. El algoritmo A*	42
3.4.1 Ruta circular con obstáculos	43
3.4.2 Ruta sinusoidal con obstáculos	44
3.5 Trayectorias de referencia para atacantes	45
3.6 Planificación de rutas defensivas	47
3.6.1 Estrategia 0	49
3.6.2 Estrategia 1	49
3.6.3 Estrategia 2	51
3.6.4 Estrategia 3	53
3.6.5 Estrategia 4	56
3.6.6 Estrategia 5	57
4 Resultados	59
4.1 Escenarios de simulación	59
4.2 Resultados de las simulaciones sin obstáculos	60

4.3 Resultados de las simulaciones con obstáculos	65
5 Conclusiones y líneas futuras	68
6 Bibliografía.....	70
Anexo I: Implicaciones Sociales, y/o Económicas, y/o Ambientales	72
Anexo II: Reflexiones Éticas y Sociales	73
Anexo III: Código desarrollado.....	74
Planificación de rutas geométricas (<i>test_path_planning.py</i>)	74
Mapa Tangencial de Visibilidad (<i>test_tangent_visibility_map.py</i>).....	77
Navegación entre obstáculos. (<i>test_obstacle_avodiance.py</i>).....	84
Simulación de escenarios (<i>test_scenarios.py</i>).....	87
Estrategias defensivas. (<i>test_simulations.py</i>).....	98

ÍNDICE DE FIGURAS

Figura 2-1.- Dron “SEA HUNTER”	9
Figura 2-2.- Dron “SEA HAWK”	9
Figura 2-3.- Dron “OVERLORD”	10
Figura 2-4.- Dron “SEAGULL”	10
Figura 2-5.- Dron “JARI-USV”	10
Figura 2-6.- Dron “ZHUHAIYUN”	11
Figura 2-7.- Dron de Estados Unidos T38 Devil Ray	11
Figura 2-8.- Dron “Orca XLUUV”	12
Figura 2-9.- Dron “MUSKIE M-18”	12
Figura 2-10.- Dron Al Seer Hydronalix	12
Figura 2-11.- Dron “UK MAST-13”	13
Figura 2-12.- Dron “MAGURA V5”	13
Figura 2-13.- Dron “SEA BABY”	13
Figura 2-14.- Dron “SEAD 23”	14
Figura 2-15.- Dron de salvamento “EMILY”	14
Figura 2-16.- Dron de estudio oceánico “SAILDRONE”	15
Figura 2-17.- Dron salvavidas “OCEANALPHA DOLPHIN 1”	15
Figura 2-18.- Dron de salvamento submarino “QYSEA FIFISH RESCUE”	16
Figura 3-1.- Resultado obtenido con la función <i>computeLinePath</i> para genera una ruta lineal de 30 puntos entre el origen en (100.0, 10.0) y el destino en (-100.0, -5.0).....	21
Figura 3-2.- Resultado obtenido con la función <i>computeCirclePath</i> para genera una ruta circular de 30 puntos entre el origen y el destino	24
Figura 3-3.- Resultado obtenido con la función <i>computeSinePath</i> para generar una ruta sinusoidal de 30 puntos entre el origen y el destino.....	26
Figura 3-4.- Resultado obtenido con la función <i>computeSawPath</i> para generar una ruta de diente de sierra de 30 puntos entre el origen y el destino	29
Figura 3-5.- Resultado obtenido con la función <i>computeSpiraPath</i> para generar una ruta en forma de espiral de 60 puntos entre el origen y el destino	32
Figura 3-6.- Representación de la ruta de línea de visión directa	34
Figura 3-7.- Representación gráfica de tangentes externas entre dos circunferencias	36
Figura 3-8.- Representación gráfica de tangentes internas entre dos circunferencias.....	38
Figura 3-9.- Representación gráfica de tangentes desde un punto externo a una circunferencia.....	39
Figura 3-10.- Representación gráfica de un mapa de visibilidad de tangentes entre varias circunferencias	41

Figura 3-11.- Resultado obtenido con la aplicación del algoritmo “ <i>A-star</i> ” para obtener una ruta circular que evada los obstáculos.....	44
Figura 3-12.- Resultado obtenido con la aplicación del algoritmo “ <i>A-star</i> ” para obtener una ruta sinusoidal que evada los obstáculos.....	45
Figura 3-13.- Ruta en diente de sierra a la que no se le aplica el algoritmo A*.....	46
Figura 3-14.- Representación gráfica de la ruta en diente de sierra a la que se aplica el algoritmo A*.....	47
Figura 3-15.- Representación gráfica del escenario de la Estrategia 0.....	49
Figura 3-16.- Representación gráfica del escenario de la Estrategia 1.....	51
Figura 3-17.- Representación gráfica del escenario de la estrategia nº2.....	53
Figura 3-18.- Representación gráfica del escenario de la estrategia nº3.....	55
Figura 3-19.- Representación gráfica del escenario de la estrategia nº4.....	57
Figura 3-20.- Representación gráfica del escenario de la estrategia nº5.....	58
Figura 4-1.- Representación gráfica de la relación entre el número de simulaciones y las estrategias ganadas por los atacantes y defensores.	61
Figura 4-2.- Representación gráfica de la relación entre la variación del tamaño del enjambre y las estrategias utilizadas	63
Figura 4-3.- Representación gráfica entre costes totales y tiempos de simulación por estrategia ...	64
Figura 4-4.- Representación gráfica de la relación entre el número de simulaciones y las estrategias ganadas por los atacantes y defensores.	65
Figura 4-5.- Representación gráfica de la relación entre la variación del tamaño del enjambre y las estrategias utilizadas	66
Figura 4-6.- Representación gráfica entre costes totales y tiempos de simulación por estrategia ...	67

1 INTRODUCCIÓN Y OBJETIVOS

1.1 Contexto y motivación

En los últimos años, los drones han adquirido un papel fundamental en el ámbito de la vigilancia y la defensa, transformando por completo la manera en que se llevan a cabo las operaciones militares y de seguridad. Su creciente presencia en conflictos recientes ha demostrado su eficacia en múltiples tareas, desde el reconocimiento y la recolección de inteligencia hasta la neutralización de objetivos estratégicos.

Gracias a su capacidad para operar en entornos hostiles sin poner en riesgo la vida de los operadores, los drones han redefinido el concepto de guerra moderna, permitiendo una respuesta rápida y precisa ante amenazas emergentes.

La versatilidad, rapidez de despliegue y bajos costos operativos de estos sistemas los han convertido en una alternativa altamente eficiente frente a tecnologías más costosas, como los misiles defensivos o los sistemas tripulados de combate. Su capacidad para operar en entornos urbanos, montañosos e incluso en alta mar les otorga una ventaja táctica significativa, permitiendo a los ejércitos y fuerzas de seguridad adaptarse a diversos escenarios sin comprometer grandes cantidades de recursos. Además, los avances en inteligencia artificial han permitido mejorar la autonomía de estos dispositivos, reduciendo la necesidad de intervención humana y optimizando su desempeño en situaciones de alto riesgo.

Uno de los principales desafíos en la integración de drones en misiones defensivas es la gestión de enjambres, donde múltiples unidades deben operar de manera coordinada para ejecutar tareas colaborativas. La optimización del control, la comunicación entre estos dispositivos y la toma de decisiones en tiempo real entre otros dispositivos es crucial para maximizar su efectividad en escenarios complejos. Esta estrategia no solo permite una mayor cobertura y precisión en misiones de reconocimiento o ataque, sino que también dificulta la detección y neutralización de los drones por parte del enemigo. Sin embargo, el desarrollo de tecnologías de guerra electrónica y sistemas anti-drones plantea un nuevo reto en la implementación de estas tácticas, obligando a los desarrolladores a buscar métodos de encriptación de señales y contramedidas avanzadas para mantener la operatividad de estos sistemas[1].

A medida que la tecnología avanza, es previsible que el uso de drones en defensa y seguridad continúe en expansión, consolidándose como una herramienta esencial en los conflictos del futuro. Su integración con tecnologías emergentes, como la inteligencia artificial, el aprendizaje automático y los sistemas de comunicación satelital, permitirá desarrollar drones más autónomos, inteligentes y resistentes a las contramedidas enemigas. En este sentido, la evolución de los drones no solo impactará en el ámbito militar, sino que también tendrá implicaciones en la seguridad nacional, la protección de infraestructuras críticas y la respuesta ante desastres naturales o ataques terroristas.

En definitiva, los drones han pasado de ser una innovación tecnológica, a convertirse en un pilar fundamental de la estrategia militar y de seguridad en el siglo XXI. Su continua evolución y adaptación a los nuevos desafíos del panorama global garantizarán su protagonismo en el desarrollo de tácticas y sistemas defensivos en el futuro[2].

Tal es la importancia de estos drones para los países occidentales que en España ha causado interés en la Armada y ésta ha firmado un contrato para adquirir un dron de superficie marina, el vehículo no tripulado de superficie Sead 23 con la empresa viguesa de Seadrone por 1,1 millones de euros. La Jefatura de Apoyo Logístico ha cerrado con la empresa española Seadrone, del grupo Zelenda, la compra de un vehículo no tripulado de superficie Sead 23 por un importe de 1,1 millones de euros [3].

1.2 Objetivos y alcance

El objetivo de este proyecto es desarrollar una aplicación cuyo objetivo es analizar y validar el uso de enjambres de Vehículos de Superficie No Tripulados (USV) en la protección e interceptación de amenazas en entornos marítimos. Los USV han surgido como una tecnología clave para vigilancia y defensa en áreas estratégicas como puertos y rutas marítimas. Ante el aumento de amenazas como embarcaciones no autorizadas o piratería, los enjambres de USV ofrecen una solución innovadora, ya que pueden operar de manera autónoma y coordinada, proporcionando una respuesta rápida ante intrusiones.

El proyecto busca desarrollar un software con tres objetivos principales. Primero, la simulación: crear entornos donde los enjambres de USV se desplieguen y simulen escenarios reales de protección, permitiendo de esta manera evaluar su desempeño en distintas situaciones operativas. Segundo, el análisis: implementar sistemas de control para que un operador humano gestione los USV en diversas configuraciones, evaluando variables como detección de amenazas, navegación y tácticas de interceptación. Tercero, la validación: probar y validar el uso de enjambres mediante simulaciones avanzadas y análisis comparativos con sistemas tradicionales de seguridad marítima.

Adicionalmente, se propone integrar mejoras como la optimización de estrategias para aumentar la eficiencia del enjambre y la incorporación de inteligencia artificial para gestionar el comportamiento autónomo de los USV.

El desarrollo de la aplicación incluirá la implementación de algoritmos de navegación autónoma, coordinación de enjambres y detección de amenazas a través de sensores virtuales. Además, se establecerá un modelo de simulación que permita recrear entornos marítimos con distintos niveles de complejidad, considerando factores como condiciones climáticas, tráfico naval y patrones de comportamiento de amenazas potenciales.

Finalmente, el proyecto establecerá un marco de referencia para futuras aplicaciones de enjambres de USV en tareas defensivas, de vigilancia, de búsqueda y apoyo, abriendo la posibilidad a nuevas investigaciones y desarrollos en el ámbito de la seguridad marítima.

1.3 Estructura de la memoria

La presente memoria del proyecto se organiza conforme a la estructura previamente expuesta en el índice de contenidos. A continuación, se detalla la distribución de los capítulos, proporcionando una visión general de cada una de las secciones que conforman este documento:

1. **Introducción y objetivos.** En este primer capítulo se contextualiza el proyecto, exponiendo la motivación que ha llevado a su desarrollo y la relevancia de este dentro del ámbito de estudio. Se describen las problemáticas existentes y cómo este trabajo pretende abordarlas. Además, se definen los objetivos generales y específicos que se pretenden alcanzar, estableciendo las bases metodológicas sobre las que se sustenta la investigación y el desarrollo del proyecto.

2. **Estado del arte.** En este apartado se realiza un análisis exhaustivo del estado del arte relacionado con la temática del proyecto. Se revisan las investigaciones y desarrollos previos en el campo de estudio, estableciendo una comparación entre las diferentes estrategias existentes y su aplicabilidad a la problemática abordada en este trabajo. Se incluyen referencias a estudio, tecnologías y metodologías que han servido de base para el desarrollo del proyecto, proporcionando un marco teórico sólido.
3. **Desarrollo.** Este capítulo constituye el núcleo central del trabajo, donde se expone detalladamente el proceso de diseño, implementación y experimentación llevado a cabo. Se describen las metodologías adoptadas, los algoritmos utilizados y las herramientas empleadas. Asimismo, se presenta la estructura del sistema, detallando su arquitectura y funcionamiento. En este apartado, se incluyen gráficas y fragmentos de código que permiten ilustrar de manera clara los conceptos clave del proyecto.

Resultados. En esta sección se presentan los resultados obtenidos tras la ejecución de las pruebas y experimentos realizados. Se analizan los datos recopilados, comparando las diferentes estrategias o enfoques implementados y evaluando su efectividad en base a métricas previamente definidas. Se incluyen representaciones gráficas que facilitan la interpretación de los resultados, así como un análisis crítico sobre su significado y sus implicaciones dentro del contexto del estudio.

4. **Conclusiones y líneas futuras:** El documento concluye con un resumen de los logros alcanzados y un análisis crítico sobre el trabajo realizado. Se reflexiona acerca de las limitaciones encontradas, las dificultades superadas y las posibles mejoras que podrían implementarse en futuras investigaciones. Además, se plantean líneas futuras de trabajo, sugiriendo posibles extensiones y aplicaciones del proyecto, con el objetivo de proporcionar continuidad a la investigación realizada.

2 ESTADO DEL ARTE

2.1 Contexto histórico

El uso de vehículos no tripulados, tanto aéreos como marítimos, ha evolucionado de manera significativa a lo largo de la historia, desempeñando un papel clave en la vigilancia, la defensa y otras aplicaciones estratégicas. En la actualidad los drones son generalmente asociados con las aeronaves no tripuladas, su concepto tiene raíces profundas en el desarrollo de tecnología militar y civil.

El uso de estos vehículos no tripulados en el ámbito militar tiene antecedentes desde la Primera Guerra Mundial. Durante este periodo se experimentó con los primeros aviones no tripulados, como el “Kettering Bug”, desarrollado en 1918 por Estados Unidos (EE. UU.), un torpedo aéreo guiado diseñado para misiones de ataque. Sin embargo, su aplicación fue limitada debido a la tecnología rudimentaria de la época.

En la Segunda Guerra Mundial, el concepto de drones y vehículos no tripulados tomó mayor importancia. Un claro ejemplo de ello fueron los minisubmarinos japoneses tipo Ko-Hyoteki, utilizados en el ataque a Pearl Harbor el 7 de diciembre de 1941 [4]. Estos submarinos, operados por tripulaciones de dos personas, tenían la misión de infiltrarse en la bahía y lanzar torpedos contra los buques estadounidenses. Este hecho marcó un precedente en el uso de vehículos compactos con capacidades furtivas. Por otro lado, los alemanes también experimentaron con bombas planeadoras teledirigidas, como la Fieseler Fi 103 (V-1), una de las primeras armas no tripuladas de ataque a larga distancia, precursor de los misiles de crucero modernos [5].

Durante la Guerra Fría, el desarrollo de drones y vehículos no tripulados experimentó un crecimiento acelerado, impulsado por la necesidad del espionaje y reconocimiento sin riesgo para los pilotos. EE. UU. y la Unión Soviética desarrollaron aviones de reconocimiento no tripulados, como el Ryan Model 147 Lightning Bug, utilizado por EE. UU. en misiones de espionaje sobre Vietnam del Norte y China. En el ámbito marítimo se exploraron tecnologías para la detección y eliminación de minas submarinas, dando paso a la evolución de los drones submarinos autónomos que hoy son fundamentales en la guerra naval moderna.

En las últimas décadas, el uso de drones ha trascendido el ámbito militar para aplicaciones civiles, científicas y comerciales. Sin embargo, también han sido adoptados por grupos terroristas y de crimen organizado, como, por ejemplo, el uso de drones semi-sumergibles y submarinos por parte de los cárteles de la droga.

El avance de la inteligencia artificial y la robótica ha permitido la creación de drones submarinos más sofisticados, que son usados en operaciones tales como la exploración oceánica, arqueológica submarina, y la detección de desastres ambientales.

El uso de drones y vehículos no tripulados ha evolucionado desde sus primeras apariciones en conflictos bélicos hasta convertirse en una herramienta muy versátil en diversos campos, su desarrollo ha sido impulsado tanto por necesidades estratégicas como por avances tecnológicos. Con la creciente integración de la inteligencia artificial, estos sistemas continúan redefiniendo la vigilancia, la seguridad y la logística en el siglo XXI.

2.2 Drones en el ámbito militar

Los drones navales de superficie están transformando el ámbito militar y de seguridad marítima. Su capacidad para operar de manera autónoma o semiautónoma en entornos hostiles los ha convertido en una herramienta clave para misiones de vigilancia, patrullaje, guerra antisubmarina y salvamento marítimo. A diferencia de los buques tripulados, los USVs pueden desplegarse con menores costos operativos y sin exponer vidas humanas, lo que los hace especialmente útiles en conflictos y operaciones de alto riesgo.

En la actualidad, diversas potencias militares han desarrollado USVs con capacidades avanzadas, cubriendo un amplio rango de misiones:

Drones navales de superficie militares:

1. SEA HUNTER (EE. UU.) [6]. Diseñado por DARPA, este USV autónomo de Clase III puede operar durante largos periodos en mar abierto para realizar misiones de guerra antisubmarina y reconocimiento. Este dron es el que se muestra en la Figura 2-1.



Figura 2-1.- Dron “SEA HUNTER”

2. SEA HAWK (EE. UU.) [7]. Diseñado por DARPA, es el mismo modelo que el dron USV SEA HUNTER. Se observa su imagen en la Figura 2-2.



Figura 2-2.- Dron “SEA HAWK”

3. OVERLORD (EE. UU.) [8]. Diseñado para realización de operaciones de vigilancia y seguridad marítima integrados con IA. Este tipo de dron puede observarse en la Figura 2-3.



Figura 2-3.- Dron “OVERLORD”

4. SEAGULL (Israel) [9]. Un USV desarrollado por Elbit Systems, especializado en la detección y neutralización de minas y submarinos enemigos. El dron se observa en la Figura 2-4.



Figura 2-4.- Dron “SEAGULL”

5. JARI-USV (China) [10]. Un dron naval multifunción con capacidades ofensivas y defensivas, equipado con misiles, radares y sistemas de guerra electrónica. El dron se observa a continuación en la Figura 2-5.



Figura 2-5.- Dron “JARI-USV”

6. ZHUHAIYUN (China) [11]. USV con capacidad de ataque, vigilancia y guerra antisubmarina, equipado con misiles y sistemas de guerra electrónica. El dron puede observarse a continuación, en la Figura 2-6.



Figura 2-6.- Dron “ZHUHAIYUN”

7. T38 DEVIL RAY (EE. UU.) [12]: Fabricado por MARTAC, este USV de alta velocidad es utilizado para operaciones de reconocimiento, patrullaje y misiones tácticas. El dron puede observarse a continuación, en la Figura 2-7.



Figura 2-7.- Dron de Estados Unidos T38 Devil Ray

8. ORCA XLUUV (EE. UU.) [13]. Un híbrido entre USV y UUV (Unmanned Underwater Vehicle), diseñado para misiones prolongadas de inteligencia, vigilancia y guerra antisubmarina. El dron puede observarse a continuación, en la Figura 2-8.



Figura 2-8.- Dron “Orca XLUV”

9. MUSKIE M18 (EE. UU.) [14]. Este tipo de dron está diseñado por la US. Navy para llevar a cabo actividades de amenaza asimétrica. Tiene 5.5 m de eslora, alcanza una velocidad de hasta 92 km/h, una autonomía de hasta 926 km y con capacidad de transportar 450 kg de explosivo. El dron puede observarse a continuación, en la Figura 2-9.



Figura 2-9.- Dron “MUSKIE M-18”

10. AL SEER HYDRONALIX (EAU). Un USV de Emiratos Árabes Unidos diseñado para misiones de patrullaje y vigilancia en el Golfo Pérsico. El dron se puede observar en la Figura 2-10.



Figura 2-10.- Dron Al Seer Hydronalix

11. UK MAST-13 (Reino Unido) [15]. Un dron de superficie desarrollado por la Royal Navy para probar tecnologías de guerra autónoma. El dron puede observarse a continuación, en la Figura 2-11.



Figura 2-11.- Dron “UK MAST-13”

12. MAGURA V5 (Ucrania) [16]. Dron de superficie desarrollado por Ucrania con el objetivo de atacar la flota rusa. Tiene un alcance de 800 km, una velocidad de 90 km/h y contiene una carga explosiva de hasta 200 kg. A continuación, se observa el dron en la Figura 2-12.



Figura 2-12.- Dron “MAGURA V5”

13. SEA BABY (Ucrania) [17]. USV desarrollado por Ucrania con las mismas características que el dron de superficie MAGURA V5. A continuación, se observa el dron en la Figura 2-13.



Figura 2-13.- Dron “SEA BABY”

14. SEAD 23 (España). Vehículo de superficie no tripulado para realizar operaciones de vigilancia. Este vehículo mide 7 m de eslora, una manga de 2,3 m y un calado de 1,2 m. Cuenta con un sistema de propulsión *waterjet* y un motor de 240 CV diésel que permite alcanzar los 33 nudos de velocidad (61,12 km/h). A continuación, se puede observar el dron en la Figura 2-14.



Figura 2-14.- Dron “SEAD 23”

Drones navales de superficie para salvamento y rescate marítimo:

1. EMILY (Emergency Integrated Lifesaving Lanyard) [18]. Un dron diseñado para rescates en alta mar, capaz de navegar en condiciones adversas y llevar salvavidas a personas en peligro. A continuación, se puede observar el dron en la Figura 2-15.



Figura 2-15.- Dron de salvamento “EMILY”

2. SAILDRONE (EE. UU.) [19]. Aunque su función principal es la recopilación de datos oceánicos, este USV también se ha utilizado en tareas de búsqueda y rescate. A continuación, se puede observar al dron en la Figura 2-16.



Figura 2-16.- Dron de estudio oceánico “SAILDRONE”

3. OCEANALPHA Dolphin 1 [20]. Un dron salvavidas controlado a distancia, usado para operaciones de rescate en zonas de difícil acceso. A continuación, se observa el dron en la Figura 2-17.



Figura 2-17.- Dron salvavidas “OCEANALPHA DOLPHIN 1”

4. QYSEA FIFISH RESCUE [21]. Un sistema híbrido entre USV y ROV (*Remotely Operated Vehicle*), utilizado para misiones de salvamento submarino. El dron puede observarse en la Figura 2-18.



Figura 2-18.- Dron de salvamento submarino “QYSEA FIFISH RESCUE”

A medida que la tecnología de autonomía y la inteligencia artificial avanzan, el uso de drones de superficie en operaciones militares y de seguridad marítima seguirá creciendo. La integración de USVs en enjambres colaborativos, capaces de trabajar en conjunto con buques tripulados y otros vehículos no tripulados, es uno de los principales desafíos actuales. Además, su bajo costo en comparación con sistemas tripulados, como buques de guerra y misiles defensivos, los convierte en una alternativa altamente eficiente.

En el futuro, se espera que los USVs desempeñen un papel aún más relevante en la protección de aguas territoriales, misiones de búsqueda y rescate, e incluso en la disuasión de amenazas asimétricas. Su flexibilidad y capacidad para operar en entornos hostiles sin comprometer vidas humanas los consolidan como una herramienta indispensable en la estrategia naval del siglo XXI.

2.3 Escenarios de combate naval con drones y su clasificación

El uso de sistemas no tripulados en el ámbito naval ha cobrado una gran relevancia en los últimos años, impulsado tanto por la guerra híbrida entre Rusia y Ucrania, como por los avances en inteligencia artificial (IA), autonomía y guerra electrónica[22].

Estos vehículos (drones navales), han evolucionado significativamente, permitiendo a las fuerzas navales ampliar sus capacidades operativas y estratégicas sin exponer a las tripulaciones a entornos de alto riesgo.

Los drones navales se emplean en misiones de vigilancia, reconocimiento, combate y guerra electrónica, ofreciendo ventajas significativas en términos de eficiencia, versatilidad y reducción de costos. Gracias a su capacidad de operar en escenarios de alta peligrosidad, estos sistemas han transformado la forma en que se llevan a cabo las operaciones marítimas, proporcionando una mayor cobertura, recolección de datos en tiempo real y ejecución de misiones sin intervención humana directa.

El desarrollo de estos sistemas ha sido impulsado por la necesidad de contar con plataformas más autónomas, sigilosas y eficientes, capaces de operar en entornos complejos y dinámicos, como aguas territoriales hostiles o zonas de guerra naval, ya que una ventaja significativa que ofrecen estos tipos de drones es brindar a los radares enemigos una RCS (*Radar Cross Section*) mínima.

En este sentido, los drones navales han demostrado ser herramientas clave para la protección de flotas y bases navales, la detección y neutralización de amenazas submarinas, la guerra electrónica y el sabotaje de sistemas enemigos o la ejecución de ataques de precisión en operaciones ofensivas navales.

El avance tecnológico en los campos de la comunicación, sensores y algoritmos de navegación ha permitido la integración de drones navales en operaciones conjuntas con buques, submarinos y aeronaves tripuladas, maximizando su efectividad en combate y reduciendo la exposición al peligro de las fuerzas humanas. No obstante, su implementación también conlleva desafíos estratégicos y éticos, especialmente en lo que respecta a su autonomía en la toma de decisiones y la posibilidad de ciberataques o interferencias electrónicas [23].

Dado su creciente protagonismo en la guerra naval moderna, el estudio de los drones en combate naval resulta fundamental para comprender sus aplicaciones, limitaciones y el impacto que tendrán en los conflictos marítimos en el futuro[24].

Hoy en día en el ámbito naval se usan tres tipos de drones: los drones aéreos, los drones de superficie (que son los que atañen a este trabajo) y los drones submarinos[25].

En primer lugar, están los drones aéreos (UAVs – *Unmanned Aerial Vehicles*) son utilizados principalmente para llevar a cabo misiones de vigilancia y ataque en entornos marítimos. Sus principales tipos incluyen:

- Drones de reconocimiento: están equipados con sensores avanzados y cámaras de largo alcance para la detección de amenazas en el mar.
- Drones de combate: capaces de portar misiles o actuar como municiones merodeadoras.
- Drones de guerra electrónica: están diseñados para interferir con radares, sistemas de comunicación y navegación enemigos.

En segundo lugar, los drones marítimos (USVs) son aquellos que operan en la superficie del mar y pueden cumplir distintas funciones, como, por ejemplo:

- Drones de vigilancia: son los encargados de realizar misiones de patrulla/vigilancia y detección de submarinos.
- Drones de combate: están diseñados para portar los misiles, torpedos o carga explosiva para proporcionar apoyo ofensivo.
- Drones logísticos: utilizados para el transporte de suministros en misiones donde el uso de embarcaciones tripuladas supone alto riesgo para la vida.

En tercer y último lugar, los drones Submarinos (UUVs – *Unmanned Underwater Vehicles*) son aquellos que operan bajo el agua y permiten llevar a cabo misiones especializadas como, por ejemplo:

- Drones de reconocimiento submarino: para la detección de minas y exploración de zonas estratégicas.
- Drones de guerra antisubmarina (ASW): equipados con sensores acústicos y torpedos.
- Drones de sigilo: desplegados para misiones de inteligencia artificial.

3 DESARROLLO DEL TFG

3.1 Definición de escenarios

En el ámbito de la simulación de escenarios navales, el objetivo principal de este trabajo es desarrollar utilidades para la creación de una herramienta que facilite la investigación y el análisis de algoritmos avanzados para la gestión de enjambres defensivos. Para ello, es fundamental definir las condiciones bajo las cuales se llevarán a cabo las simulaciones, estableciendo un marco de referencia que permita evaluar distintas estrategias de defensa en entornos marítimos.

Dado que la simulación de escenarios reales de combate implica una gran complejidad, se introducen una serie de simplificaciones para facilitar el desarrollo inicial del sistema. No obstante, el diseño debe considerar la futura incorporación de nuevas restricciones y condiciones que permitirán modelar escenarios más realistas y complejos.

El escenario de simulación está compuesto por los siguientes elementos fundamentales:

1. Características generales del escenario:

Se definen aspectos como:

- Dimensiones del área de simulación.
- Ubicación del punto de destino que debe ser defendido de los atacantes, o “*target_point*”.
- Tipo de modelado empleado para establecer el movimiento de los elementos (atacantes y defensores) que interactúan en el escenario, el cual puede abordarse desde una perspectiva geométrica, cinemática o dinámica.

2. Obstáculos:

- Los obstáculos presentes en el escenario se modelan empleando circunferencias de diferentes radios.
- Deben ser evitados por los USVs (vehículos de superficie no tripulados) para prevenir colisiones que resultarían en su destrucción.

3. Atacantes:

- Son USVs cuya misión es alcanzar el “*target_point*”.
- Se asume que siguen una trayectoria geométrica predefinida que incorpora estrategias básicas para evitar los obstáculos, que se suponen previamente conocidos por los atacantes.
- Los atacantes no disponen de capacidad para detectar a los defensores.

- Su principal característica diferenciadora es su velocidad máxima, que suele ser superior a la de los defensores.

4. Defensores:

- Son USVs con velocidades y capacidades de maniobra inferiores a las de los atacantes.
- Su misión es interceptar a los atacantes antes de que estos alcancen el “*target_point*”, formando enjambres de defensa coordinados.
- Deben evitar los obstáculos fijos mientras se desplazan hacia su objetivo.
- Cada defensor tiene conocimiento de:
 - La ubicación y tamaño de los obstáculos.
 - La ubicación de su atacante asignado.
 - La posición de los demás defensores.
 - La ubicación del “*target_point*” a proteger.

A medida que se refine la simulación, se podrán introducir nuevas restricciones y mejoras, como:

- **Detección activa en los atacantes:** Incorporación de sensores o modelos de percepción que permitan a los atacantes identificar y reaccionar dinámicamente ante los defensores.
- **Modelado más realista del entorno:** Introducción de condiciones climáticas, corrientes marinas y otros factores externos que influyan en la navegación.
- **Estrategias avanzadas de enjambre:** Desarrollo de algoritmos de cooperación más sofisticados para optimizar la defensa del “*target_point*”.

Este enfoque progresivo permitirá que la simulación evolucione hacia escenarios más representativos de operaciones reales, proporcionando un entorno robusto para la experimentación y validación de estrategias de defensa con USVs.

A continuación, se exponen las herramientas que has sido desarrolladas para ser integradas en una aplicación de simulador de escenarios.

3.2 Generación de rutas geométricas

Tal y como se comentó anteriormente, se asume que los atacantes van a seguir trayectorias geométricas predefinidas que deben ser alteradas para evitar los obstáculos presentes en el escenario. Llegados a este punto se plantean distintos escenarios en los que los obstáculos cambian de lugar, se combinan y las trayectorias que deben seguir los USVs deben variar para conseguir llegar al objetivo. Dados dos puntos (origen y destino), una ruta se representa por medio de una serie de puntos que representen la ruta a seguir en función de algún patrón geométrico. Para este trabajo se han diseñado distintos patrones geométricos tales como el patrón lineal, el patrón de semicircunferencia, el patrón para una ruta en forma de seno o el patrón para que la ruta tenga la forma de dientes de sierra o el patrón para la que la ruta tenga forma de espiral.

A continuación, se muestra una a una las distintas rutas propuestas, con sus detalles y resultados gráficos obtenidos.

3.2.1 Ruta lineal

En este apartado se describe la generación de una trayectoria en línea recta, utilizada para conectar dos puntos en un espacio bidimensional de la forma más eficiente posible. Este enfoque es especialmente útil en aplicaciones donde se requiere minimizar la distancia recorrida, como en la navegación de drones o cualquier sistema autónomo de desplazamiento.

A diferencia de otras trayectorias más complejas, la trayectoria en línea recta garantiza el menor consumo de energía y el tiempo de viaje más corto, evitando desviaciones innecesarias.

El sistema desarrollado consta de una función principal:

- “*computeLinePath*”: se encarga de calcular los puntos que conforman la trayectoria rectilínea entre un punto de origen y un punto de destino.

A continuación, se presenta la definición de la función empleada para calcular la ruta lineal:

```
def computeLinePath(origin_point, target_point, parameters)
```

El cálculo de la trayectoria en línea recta sigue un procedimiento estructurado y eficiente, basado en la interpolación lineal de puntos entre el origen y el destino. Los pasos principales del algoritmo son los siguientes:

1. El primer paso define los puntos inicial y final. Para general la trayectoria, primero se identifican las coordenadas del punto de origen y del punto de destino, que se representan como:

$$\begin{aligned}(x_f, y_f)(Origen) \\ (x_t, y_t)(Destino)\end{aligned}$$

Estos valores son utilizados como referencia para la generación de los puntos intermedios de la trayectoria.

2. En el segundo paso, se determina el número de puntos intermedios. El usuario define un parámetro de entrada llamado “*num_points*”, que determina cuántos puntos conformarán la trayectoria rectilínea. Cuanto mayor sea este número, más precisa y suave será la trayectoria, aunque con un mayor coste computacional.

El valor de “*num_points*” se utiliza para dividir la trayectoria en segmentos iguales entre el punto de origen y el punto de destino.

3. En el tercer paso, se realizan los cálculos de las coordenadas de los puntos intermedios. Los puntos intermedios se generan utilizando la interpolación lineal en cada coordenada. Esto significa que se calcula un conjunto de valores intermedios que se distribuyen equidistantemente entre el origen y el destino.

Para calcular las coordenadas en el eje X, se utiliza la siguiente ecuación:

$$x_i = x_0 + i \cdot \frac{x_t - x_0}{N - 1}$$

Donde:

- x_i es la coordenada X del punto intermedio i .
- N es el número total de puntos de la trayectoria.
- i varía desde 1 hasta $N-1$, excluyendo el punto de origen, pero incluyendo el destino.

Este cálculo asegura que los puntos estén uniformemente distribuidos a lo largo del eje horizontal.

De forma similar, las coordenadas en el eje Y se calculan con la ecuación:

$$y_i = y_0 + i \cdot \frac{y_t - y_0}{N - 1}$$

Esto permite que los puntos intermedios mantengan la dirección de la línea recta, ajustándose de manera proporcional en el eje vertical.

4. En el cuarto paso, se lleva a cabo el almacenamiento de los puntos de la trayectoria. Una vez calculadas las coordenadas X e Y de cada punto intermedio, estas se combinan en una lista de tuplas (x, y), que representa la trayectoria completa desde el origen hasta el destino.

Este formato es ideal para su visualización y procesamiento en simulaciones de navegación.

5. La estrategia de trayectoria lineal tiene varias ventajas clave:
 - Eficiencia óptima: permite el desplazamiento más corto entre dos puntos.
 - Simplicidad computacional: su cálculo es rápido y directo.
 - Fácil implementación: puede adaptarse fácilmente a diferentes escenarios y condiciones.

Sin embargo, este método no es adecuado para entornos con obstáculos, ya que la trayectoria no considera colisiones. En estos casos, es necesario combinar la estrategia lineal con algoritmos de evitación de obstáculos.

Finalmente, el modelo de trayectoria rectilínea es una solución eficiente y directa para conectar dos puntos minimizando la distancia recorrida. Su uso es ideal en aplicaciones donde el tiempo de desplazamiento es crítico, como la navegación de drones o vehículos autónomos en espacios sin restricciones.

Este método puede complementarse con otras estrategias avanzadas, como trayectorias semicirculares o sinusoidales, para adaptarse a escenarios con restricciones de movilidad.

Como se puede observar en la Figura 3-1, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta lineal de manera precisa mediante cálculos geométricos.

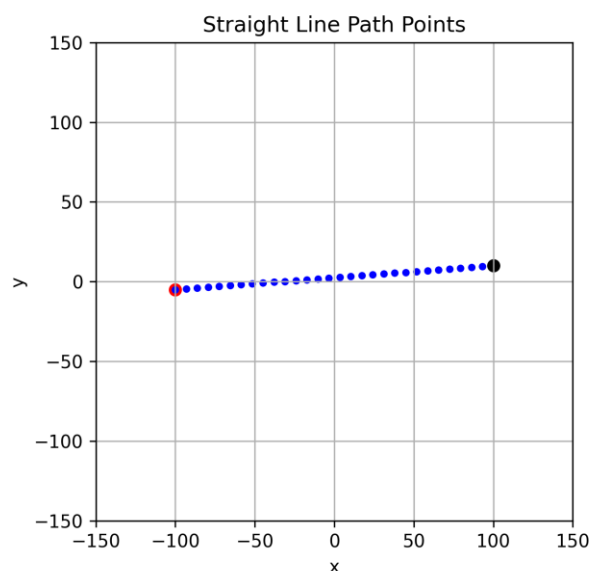


Figura 3-1.- Resultado obtenido con la función *computeLinePath* para genera una ruta lineal de 30 puntos entre el origen en (100.0, 10.0) y el destino en (-100.0, -5.0)

El código completo se encuentra en el [Anexo III](#), donde se detallan los cálculos y la implementación en Python.

3.2.2 Ruta semicircular

En este apartado se describe la generación de una trayectoria semicircular, la cual se emplea para que un objeto en movimiento siga una curva suave desde un punto de origen hasta un punto de destino.

Esta estrategia es útil en aplicaciones donde es necesario realizar maniobras sin cambios bruscos de dirección, como la planificación de rutas en sistemas de navegación autónoma.

A diferencia de las trayectorias rectilíneas o sinusoidales, la ruta semicircular permite una transición fluida entre puntos, lo que puede ser beneficioso en entornos donde se requiere evitar movimientos abruptos.

El sistema desarrollado consta de una función principal:

“*computeCirclePath*”: se encarga de calcular los puntos que conforman la trayectoria semicircular. La función “*computeCirclePath*” genera una serie de puntos intermedios que forman un semicírculo entre el punto de origen y el punto de destino. El procedimiento sigue los siguientes pasos:

1. El primer paso en la generación de la trayectoria es determinar el punto central del semicírculo. Para ello, se calcula el punto medio entre el punto de origen y el punto de destino, dado por:

$$x_c = \frac{x_0 + x_t}{2}, \quad y_c = \frac{y_0 + y_t}{2}$$

Donde:

- (x_c, y_c) representa el centro del semicírculo.
- (x_0, y_0) es la coordenada del origen.
- (x_t, y_t) es la coordenada del destino.

Este centro sirve como pivote para la generación de la curva semicircular.

2. El segundo paso es calcular el radio del semicírculo. El radio del semicírculo se define como la distancia desde el centro hasta el punto de destino. Matemáticamente, se calcula mediante la fórmula:

$$r = \sqrt{(x_t - x_c)^2 + (y_t - y_c)^2}$$

Esto garantiza que todos los puntos de la trayectoria se ubiquen a una distancia constante del centro del semicírculo.

3. El tercer paso consiste en generar los ángulos de la trayectoria. Para definir la forma del semicírculo, se genera una serie de ángulos equidistantes en el rango de 0 a π (180 grados). Este rango de ángulos permite recorrer medio círculo desde el punto de origen hasta el destino.

Los ángulos se calculan con la ecuación:

$$\theta_i = \frac{i \cdot \pi}{N}$$

Donde:

- θ_i representa el ángulo del punto i .
 - N es el número total de puntos que conforman la trayectoria.
 - i varía desde 0 hasta N , generando una distribución uniforme de puntos a lo largo del semicírculo.
4. En cuarto lugar, se realiza el cálculo de las coordenadas de los puntos de la trayectoria. A partir del radio y los ángulos generados en el paso anterior, se calculan las coordenadas de los puntos utilizando las ecuaciones de un círculo en coordenadas polares:

$$x_i = x_c + r \cdot \cos(\theta_i + \beta)$$

$$y_i = y_c + r \cdot \sin(\theta_i + \beta)$$

Donde:

- (x_i, y_i) representa la posición del punto i en la trayectoria.
- r es el radio del semicírculo.
- β es el ángulo de ajuste, determinado por la orientación del segmento entre el punto de origen y el punto de destino.

El signo del coseno y el seno en las ecuaciones puede modificarse en función de la dirección de la trayectoria, permitiendo ajustar si la curva debe orientarse hacia la izquierda o hacia la derecha.

En algunos casos, puede ser necesario cambiar la orientación del semicírculo. Para ello, se introduce un parámetro llamado "*path_rotation*", el cual permite modificar el signo de las ecuaciones anteriores para invertir la dirección de la curva.

Si "*path_rotation*" = 0 (es igual a cero), es decir, no hay rotación de la curva, la curva se genera en un sentido y , si "*path_rotation*" = 1 (es igual a 1), por lo tanto, hay rotación de la curva, esta se genera en el sentido opuesto. Esto brinda flexibilidad en la generación de rutas.

5. Por último, los puntos calculados en los pasos anteriores se almacenan en una lista de coordenadas (x, y) . Esta lista representa la trayectoria semicircular completa y puede utilizarse posteriormente para su visualización en simulaciones o sistemas de navegación.

El modelo de trayectoria semicircular permite generar un desplazamiento suave y eficiente, ideal para situarse donde se requiera una curva de maniobra sin giros bruscos. A diferencia de una trayectoria en línea recta, este enfoque ofrece una transición más fluida entre puntos, lo que es útil en movimientos autónomos de drones.

Este método puede complementarse con técnicas de ajuste dinámico, permitiendo modificar el radio del semicírculo en función de factores externos, como la velocidad del objeto o a la presencia de obstáculos.

Como puede observarse en la Figura 3-2, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta semicircular de manera precisa mediante cálculos geométricos.

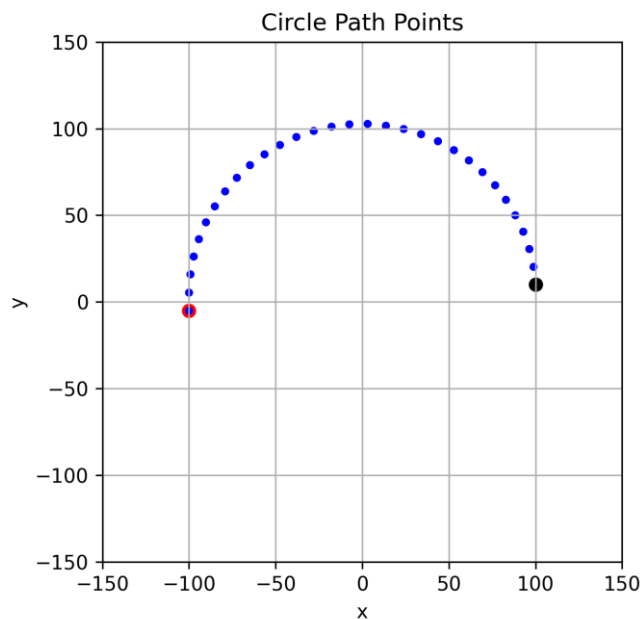


Figura 3-2.- Resultado obtenido con la función *computeCirclePath* para genera una ruta circular de 30 puntos entre el origen y el destino

. El código completo se encuentra en el [Anexo III](#), donde se detallan los cálculos y la implementación en Python.

3.2.3 Ruta sinusoidal

En este apartado se describe la generación de una trayectoria en forma de onda sinusoidal, un patrón de movimiento caracterizado por oscilaciones periódicas a lo largo del trayecto entre un punto de origen y un destino. Este tipo de ruta es útil en aplicaciones de navegación autónoma, ya que permite evitar movimientos predecibles y hacer que un objeto en movimiento cubra más área sin seguir una trayectoria rectilínea.

A diferencia de otros enfoques, como la trayectoria en espiral o en diente de sierra, la ruta sinusoidal proporciona un desplazamiento fluido y uniforme, lo que la hace ideal para simulaciones de movimientos naturales o evasivos.

El sistema desarrollado consta de una función principal:

- “*computeSinePath*”: se encarga de calcular los puntos que conforman la trayectoria sinusoidal.

La función “*computeSinePath*” se encarga de definir una serie de puntos intermedios entre el punto de origen y el punto de destino, garantizando que la trayectoria siga una curva ondulada en lugar de una lista recta.

Los elementos clave que intervienen en la generación de esta ruta son:

- Número de puntos (“*num_points*”): determina la cantidad de divisiones a lo largo de la trayectoria.
- Número de oscilaciones (“*signal_periods*”): define cuántas veces la trayectoria completa un ciclo de onda sinusoidal entre el origen y el destino.
- Amplitud de la onda (“*signal_module*”): controla la altura máxima de las oscilaciones con respecto a la trayectoria base.

A continuación, se describen los pasos fundamentales que se llevan a cabo para definir esta trayectoria, asegurando que la espiral tenga una forma coherente y útil dentro del entorno de simulación:

1. El primer paso en la generación de la trayectoria es definir una serie de puntos equidistantes en el eje X, lo que asegura que el movimiento sea uniforme en la dirección horizontal. Esto se logra mediante la ecuación:

$$x_F = x_0 + i \cdot \frac{(x_t - x_0)}{N - 1}$$

Donde:

- i es la coordenada x del punto i
- (x_0, x_t) Representan el origen y destino, respectivamente.
- N es el número total de puntos.

Este paso es esencial para garantizar que el objeto se mueva de forma continua y proporcional desde su punto de partida hasta su destino.

2. El segundo paso consiste en realizar la generación de la línea base en el eje Y. Una vez obtenidos los valores de x, se genera una línea base en el eje Y mediante interpolación lineal, asegurando una transición suave entre el punto de origen y el de destino.

$$y_f = y_0 + i \cdot \frac{(y_t - y_0)}{N - 1}$$

Esta línea servirá como referencia para la aplicación de la oscilación sinusoidal, de manera que la trayectoria mantenga una progresión natural sin desviarse de su curso general.

3. La característica distintiva de esta trayectoria es la oscilación sinusoidal, que se superpone sobre la línea base. Para lograrlo, se usa la función matemática del seno, definida como:

$$y_{sin} = s \cdot \sin\left(2\pi P \frac{x_f - x_0}{x_t - x_0}\right)$$

Donde:

- S es la amplitud de la onda (controlada por “*signal_module*”).
- P es el número de ciclos de oscilación dentro de la trayectoria (controlado por “*signal_periods*”).
- x_f representa el punto inicial de la trayectoria.
- x_t representa el punto de destino.
- x_0 representa el punto de origen.

La ecuación garantiza que la onda sinusoidal oscile periódicamente entre valores positivos y negativos, generando así el movimiento ondulado.

La coordenada final de cada punto en la trayectoria se obtiene sumando la línea base y la oscilación sinusoidal:

$$y_f = y_{base} + y_{sin}$$

4. Finalmente se generan y almacenan los puntos de la trayectoria. Los puntos calculados mediante los pasos anteriores se almacenan en una lista de coordenadas (x, y). Esta lista representa la trayectoria completa en forma de onda sinusoidal y puede ser utilizada posteriormente para su visualización o análisis.

El resultado es una ruta fluida y continua, con osciladores suaves que permiten simular movimientos más naturales en comparación con una trayectoria lineal.

El modelo de trayectoria sinusoidal permite generar un movimiento fluido y predecible que puede ser utilizado en diversas aplicaciones, desde movimientos evasivos hasta simulación de trayectorias naturales. A diferencia de trayectorias más abruptas, como el diente de sierra, este enfoque proporciona una transición más suave entre puntos, lo que puede ser beneficioso en sistemas donde se requiera optimizar la aceleración y la estabilidad del movimiento.

Como se puede observar en la Figura 3-3, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta sinusoidal de manera precisa mediante cálculos geométricos:

El código completo se encuentra en el [Anexo III](#), donde se detallan los cálculos y la implementación en Python.

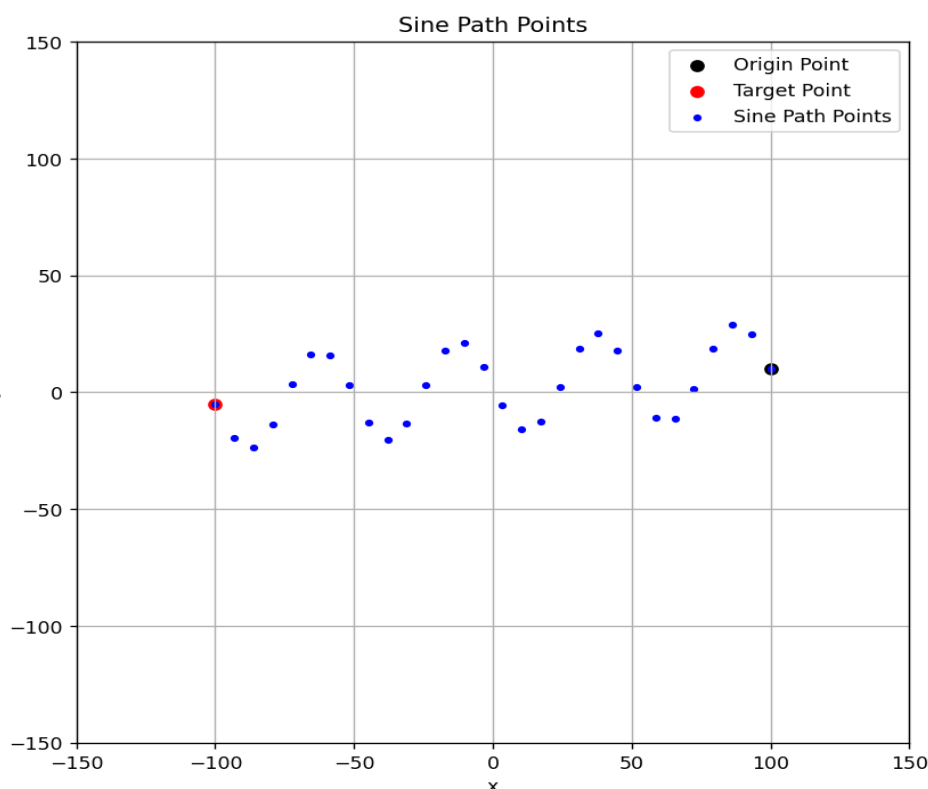


Figura 3-3.- Resultado obtenido con la función *computeSinePath* para generar una ruta sinusoidal de 30 puntos entre el origen y el destino

3.2.4 Ruta dientes de sierra

En este apartado se describe la generación de una trayectoria en forma de diente de sierra, un patrón de movimiento caracterizado por oscilaciones repetitivas a lo largo de la trayectoria entre el punto de origen y el destino. Este tipo de ruta es ampliamente utilizado en aplicaciones de navegación autónoma, particularmente en estrategias de búsqueda y exploración, ya que permite recorrer una mayor superficie en comparación con una trayectoria lineal.

A diferencia de otros métodos como la trayectoria en espiral, en la cual el radio disminuye progresivamente, la ruta en diente de sierra alterna constantemente su desplazamiento vertical, manteniendo un patrón periódico que cubre más área a lo largo de su recorrido.

El sistema desarrollado consta de una función principal:

- “*computeSawPath*”: se encarga de calcular los puntos que conforman la trayectoria en diente de sierra.

La función “*computeSawPath*” se encarga de calcular una serie de puntos de referencia que definen la trayectoria de la ruta. Para ello, el sistema emplea una combinación de interpolación lineal y una onda triangular, logrando así el patrón de diente de sierra.

La trayectoria se genera a partir de dos puntos clave:

- Punto de origen (“*origin_point*”): coordenadas desde donde comienza la trayectoria.
- Punto de destino (“*target_point*”): coordenadas donde finaliza la trayectoria.

Adicionalmente, se configuran una serie de parámetros que permiten ajustar la forma y la amplitud del movimiento oscilante:

- “*num_points*”: número total de puntos en la trayectoria.
- “*signal_periods*”: número de ciclos completos de oscilación dentro de la trayectoria.
- “*signal_module*”: amplitud de la oscilación vertical, determinando la intensidad del patrón de diente de sierra.

A continuación, se describen los pasos fundamentales que se llevan a cabo para definir esta trayectoria, asegurando que la espiral tenga una forma coherente y útil dentro del entorno de simulación:

1. El primer paso en la generación de la trayectoria es distribuir uniformemente los puntos de la trayectoria a lo largo del eje X, asegurando que estén espaciados de manera equidistantes desde el punto de origen hasta el punto de destino. Esto se consigue mediante la siguiente fórmula:

$$x_f = x_0 + i \cdot \frac{(x_t - x_0)}{N - 1}$$

Donde:

- x_f es la coordenada x del punto i.
- x_0 es la coordenada x del punto de origen.
- x_t es la coordenada x del punto de destino.
- N es el número total de puntos de la trayectoria.

Esto permite definir un movimiento constante en la dirección horizontal.

2. El segundo paso consiste en llevar a cabo la generación de la línea base en el eje Y. Una vez determinados los valores del eje x, se genera una línea base en el eje Y, que representa una interpolación lineal entre el punto de origen y el de destino. Matemáticamente esta interpolación se expresa como:

$$y_f = y_{0+i} \cdot \frac{(y_t - y_0)}{N - 1}$$

Donde:

- y_f es la coordenada y del punto i.
- y_0 e y_t son las coordenadas y del origen y del destino, respectivamente.

Esta base servirá como referencia para la aplicación de la oscilación en diente de sierra.

3. El tercer paso consiste en la generación del patrón de onda triangular. Para lograr la oscilación característica de la trayectoria en diente de sierra, se introduce una onda triangular que modifica la coordenada y de cada punto. Esta onda es generada de la siguiente manera:

- Se define una fase de oscilación, que representa el avance de la onda dentro de la trayectoria:

$$fase = \left(\frac{i + \left(\frac{N-1}{4 \cdot P} \right)}{N-1} \right) \cdot P$$

Donde:

- P es el número total de periodos de oscilación de la trayectoria.
 - N es el número de puntos.
 - i representa el índice del punto actual.
- A partir de esta fase, se genera una onda triangular simétrica, utilizando la función:

$$y_{triangular} = S \cdot (1 - 4 \cdot |(Fase \bmod 1) - 0.5|)$$

Donde:

- S es la amplitud de la oscilación (controlada por “*signal_module*”).
 - La operación mod 1 garantiza que la función triangular se repita a lo largo de toda la trayectoria.
- Finalmente, la coordenada y final del punto se obtiene como la suma de la línea base y la oscilación triangular:

$$y_i = y_{base} + y_{triangular}$$

4. Finalmente, el cuarto paso consiste en la generación y almacenamiento de los puntos de la trayectoria. Los puntos calculados mediante los pasos anteriores se almacenan en una lista de coordenadas (x, y). Esta lista representa la trayectoria completa en diente de sierra y puede ser utilizada posteriormente para su representación gráfica.

El resultado es una ruta con oscilaciones constantes, permitiendo una exploración más eficiente del área recorrida en comparación con un movimiento rectilíneo.

El modelo de trayectoria en diente de sierra proporciona una manera efectiva de recorrer un área con un movimiento oscilante periódico, lo cual es útil en estrategias de búsqueda y exploración. A diferencia de una trayectoria recta, este método permite cubrir más superficie en menos tiempo.

Este enfoque puede mejorarse mediante ajustes dinámicos en la amplitud de la onda o mediante la combinación con otras estrategias. Además, su implementación puede ser optimizada para escenarios donde el terreno presenta obstáculos o restricciones de movimiento.

Como se puede observar en la Figura 3-4, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta dientes de sierra de manera precisa mediante cálculos geométricos:

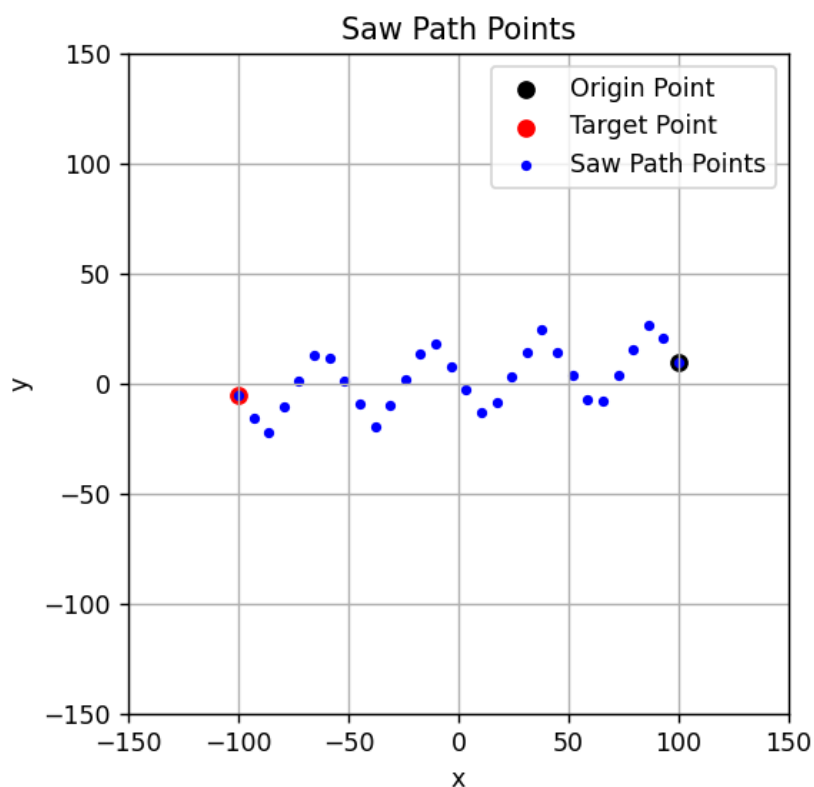


Figura 3-4.- Resultado obtenido con la función *computeSawPath* para generar una ruta de diente de sierra de 30 puntos entre el origen y el destino

Como se observa, la función “*computeSawPath (origin_point, target_point, parameters)*” permite calcular trayectorias con un patrón oscilatorio en diente de sierra, asegurando una interpolación precisa y adaptable a diversas aplicaciones. El código completo se encuentra en el [Anexo III](#), donde se detallan los cálculos y la implementación en Python.

3.2.5 Ruta espiral

En esta sección se describe la implementación de un sistema que permite la generación y visualización de una trayectoria en forma de espiral. Este tipo de ruta es útil en aplicaciones de navegación autónoma, donde es necesario que un agente (como un dron o un robot) se desplace desde un punto de inicio hasta un objetivo siguiendo una trayectoria controlada y progresiva y precisa, que se adapte a diferentes escenarios.

A diferencia de una trayectoria en línea recta, la espiral permite explorar progresivamente el entorno antes de alcanzar el destino, lo que puede ser útil en estrategias de búsqueda, aproximaciones controladas o maniobras evasivas.

El sistema desarrollado consta de dos funciones principales:

- “*computeSpiralPath*”: se encarga de calcular los puntos que conforman la trayectoria en espiral.
- “*plotSpiralPath*”: dibuja la trayectoria generada sobre un gráfico, permitiendo su visualización.

El cálculo de la ruta en espiral se realiza en la función “*computeSpiralPath*”, la cual toma como entrada dos puntos clave:

- Punto de origen (“*origin_point*”): coordenadas del punto de partida.
- Punto de destino (“*target_point*”): coordenadas del punto donde finaliza trayectoria en espiral.

Además, la función recibe un conjunto de parámetros de configuración, que incluyen:

- “*num_points*”: número total de puntos que conformarán la espiral
- “*path_rotation*”: dirección de la espiral (horaria o antihoraria)
- “*num_rotations*”: número total de giros completos que realizará la espiral antes de alcanzar el objetivo.

El cálculo de los puntos de la espiral es un proceso clave en la generación de una trayectoria que parte de un punto de origen y se dirige progresivamente hacia un destino siguiendo un patrón curvado. A continuación, se describen los pasos fundamentales que se llevan a cabo para definir esta trayectoria, asegurando que la espiral tenga una forma coherente y útil dentro del entorno de simulación:

1. El primer paso consiste en determinar qué tan lejos se encuentra el punto de destino respecto al punto de origen. Para esto, se utiliza la distancia euclidiana, que es la manera más sencilla de calcular la separación entre dos puntos en un espacio bidimensional. Matemáticamente hablando se expresa mediante la siguiente fórmula:

$$distancia = \sqrt{(x_f - x_t)^2 + (y_f - y_t)^2};$$

Donde:

- D es la distancia entre puntos.
- (x_f, y_f) representan las coordenadas del punto de origen.
- (x_t, y_t) representa las coordenadas del punto de destino.

Esta distancia servirá como radio inicial de la espiral, que irá reduciéndose a medida que se avanza en la trayectoria.

2. A continuación, como segundo paso, se realizan los cálculos de los Ángulos de Rotación. La espiral se genera siguiendo una serie de ángulos de rotación que determinan su curvatura. Dependiendo de la configuración, la espiral puede desarrollarse en sentido horario o antihorario. Para ello, se genera una secuencia de valores angulares entre 0 y un número determinado de vueltas completas.
 - Si la espiral realiza una trayectoria en sentido antihorario, los ángulos aumentan progresivamente: $\theta = [0, 2\pi, 4\pi, \dots, 2\pi \cdot n]$.
 - Si la espiral realiza una trayectoria en sentido horario, los ángulos disminuyen progresivamente: $\theta = [0, -2\pi, -4\pi, \dots, -2\pi \cdot n]$.

Donde n representa el número de vueltas completas que da la espiral. Esto permite definir el patrón de giro que la trayectoria seguirá al moverse hacia el destino.

3. Siguiendo con el tercer paso, se lleva a cabo la definición de los radios decrecientes de la espiral. A diferencia de una circunferencia, donde el radio es constante, en una espiral este radio disminuye progresivamente a medida que se acerca al punto de destino. Este comportamiento se logra generando una secuencia de valores de radio que van desde el radio inicial (la distancia calculada en el primer paso) hasta cero en el destino.

El radio en cada punto de la espiral se define de la siguiente manera:

$$r_i = r_{inicial} \cdot \left(1 - \frac{i}{N}\right)$$

Donde:

- r_i es el radio en el punto i .
- $r_{inicial}$ es el radio máximo calculado previamente.
- i representa el índice del punto dentro de la trayectoria.
- N es el número total de puntos de la espiral.

Este mecanismo garantiza que la espiral disminuya su radio progresivamente, haciendo que se “cierre” y termine en el punto de destino.

4. El cuarto paso consiste en el cálculo de las coordenadas de los puntos. Una vez definidos los ángulos y los radios, se procede a calcular las coordenadas de cada punto de la trayectoria. Para ello, se emplean las funciones trigonométricas seno y coseno, que permiten determinar la ubicación de un punto en un sistema de coordenadas polares.

Las ecuaciones para obtener las coordenadas x e y de cada punto en la espiral son las que a continuación se relacionan:

$$x_f = x_t + r_i \cdot \cos(\theta_i)$$

$$y_f = y_t + r_i \cdot \sin(\theta_i)$$

Donde:

- x_f, y_f son las coordenadas del punto inicial.
- x_t, y_t son las coordenadas del punto de destino.
- r_i es el radio en el punto i .
- θ_i es el ángulo correspondiente a ese punto.

Cada punto calculado se almacena en una lista para su posterior uso en la visualización y representación gráfica.

5. Finalmente, como quinto y último paso, se lleva a cabo la generación y almacenamiento de los puntos de la espiral. Todos los puntos generados se almacenan en una lista de coordenadas (x, y) , asegurando que la espiral siga la forma deseada. Este conjunto de puntos es lo que posteriormente se utiliza para representar gráficamente la trayectoria en el sistema de simulación.

El resultado final es una trayectoria en espiral que parte del punto de origen y se reduce progresivamente hasta alcanzar el punto de destino, ajustándose a los parámetros configurados y proporcionando un movimiento suave y progresivo.

El proceso de generación de la espiral permite definir una trayectoria controlada y progresiva en un entorno bidimensional. Esta metodología es fundamental en múltiples aplicaciones dentro de la navegación autónoma, desde estrategias de exploración hasta mecanismos de acercamiento controlado a un objetivo.

Como se puede observar en la Figura 3-5, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta espiral de manera precisa mediante cálculos geométricos.

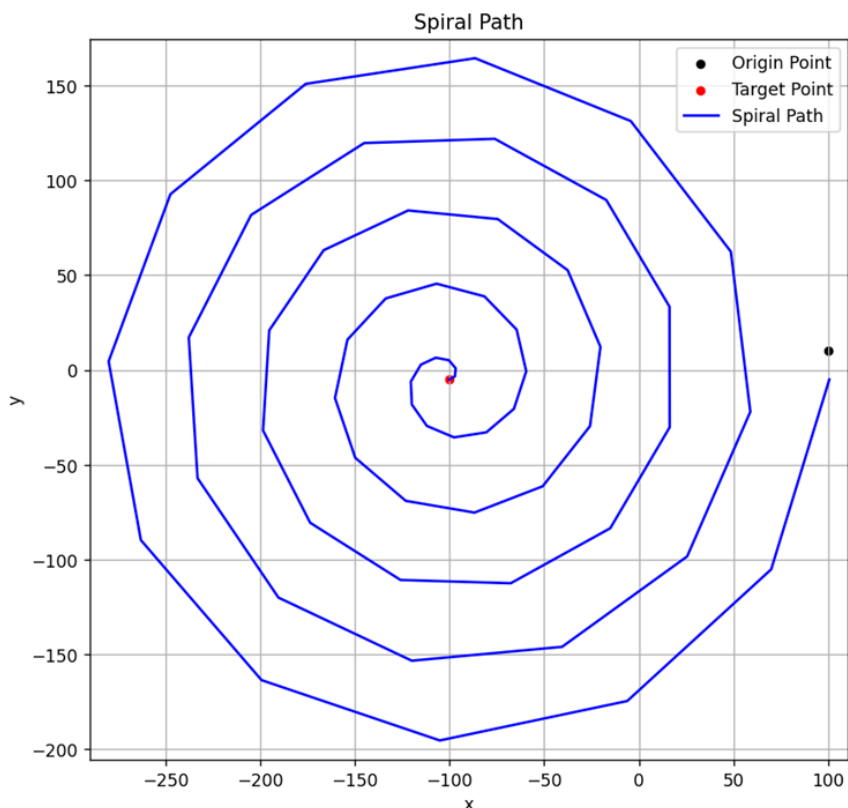


Figura 3-5.- Resultado obtenido con la función *computeSpiraPath* para generar una ruta en forma de espiral de 60 puntos entre el origen y el destino

El código completo se encuentra en el [Anexo III](#), donde se detallan los cálculos y la implementación en Python.

3.3 Navegando entre obstáculos.

La siguiente fase del problema consiste en el diseño e implementación de un sistema que permita al dron, utilizando en simulación de los escenarios, desplazarse de manera autónoma a través de un entorno con obstáculos. Para lograr este objetivo, es esencial que el dron no solo pueda reconocer los obstáculos presentes en su entorno, sino que también sea capaz de generar rutas óptimas que le permitan alcanzar de manera segura y eficiente.

Este proceso implica la implementación de algoritmos de navegación y planificación de trayectorias que optimicen el desplazamiento del dron, garantizando su capacidad de adaptación a entornos dinámicos y a posibles variaciones en la distribución de los obstáculos. Además, podrían considerarse el uso de sensores y sistemas de percepción que permitan al dron detectar los elementos de su entorno en tiempo real, favoreciendo una navegación segura y eficiente. El procesamiento de estos datos mediante algoritmos de visión artificial y técnicas de aprendizaje automático permite al sistema identificar obstáculos estáticos y dinámicos, anticipando posibles colisiones y ajustando la trayectoria en consecuencia.

La planificación de la trayectoria es otro aspecto clave en este proceso. Se requieren algoritmos de navegación avanzada, como el algoritmo A* (“A-star”) o algoritmos basados en campos potenciales, que permitan generar caminos eficientes desde el punto de origen hasta el destino. Estos algoritmos deben ser capaces de calcular rutas evitando colisiones, minimizando la distancia recorrida y optimizando el consumo de energía del dron. En escenarios dinámicos, donde los obstáculos pueden moverse o aparecer

repentinamente, es crucial integrar métodos de planificación en tiempo real que permitan ajustes constantes en la trayectoria.

Además de la detección y la planificación de la ruta, la estabilidad y el control del dron durante el vuelo son factores determinantes en la navegación autónoma. Para garantizar un desplazamiento preciso, es necesario implementar sistemas de control robustos, como estrategias basadas en redes neuronales. Estos sistemas aseguran que el dron pueda responder de manera adecuada a perturbaciones externas, como ráfagas de viento o cambios en el terreno.

3.3.1 Ruta de línea de visión directa

En el marco de este proyecto, uno de los aspectos clave en la simulación de enfrentamientos entre drones defensores y atacantes es determinar si un dron puede identificar a otro, es decir, si existe una línea de visión libre de obstáculos entre dos puntos del escenario. Este tipo de análisis es esencial para decidir, por ejemplo, si un dron puede interceptar a un atacante directamente o si debe cambiar su trayectoria para evitar un obstáculo.

Con este propósito, se pretende implementar la función *“computeLineOfSight”*, que permite verificar si el segmento que une dos puntos del plano (por ejemplo, la posición actual de un dron y su objetivo) está libre de colisiones con un obstáculo representado como un círculo. Esta función emplea herramientas de geometría computacional y es especialmente útil para escenarios donde se deben tener en cuenta restricciones físicas del entorno.

La función *“computeLineOfSight”* toma tres elementos como entrada:

- *“pointA”* y *“pointB”*: son las coordenadas de los puntos que se desea conectar con una línea recta. Estos puntos suelen representar la posición de un dron y el objetivo al que se dirige (como otro dron o un punto fijo del escenario).
- *“Circle”*: representa un obstáculo, definido por las coordenadas de su centro y su radio. Este modelo permite simular, por ejemplo, una zona de exclusión o un obstáculo físico en el entorno, como una estructura o una montaña.

A continuación, el algoritmo realiza los siguientes pasos:

1. Representación matemática del problema: Se modela el segmento de línea que conecta los puntos A y B como una función matemática. A la vez, se representa el obstáculo como una circunferencia en el plano.
2. Cálculo de intersecciones: el algoritmo calcula si el segmento de línea intersecta la circunferencia. Para ello, resuelve una ecuación cuadrática que representa la distancia del segmento al centro del círculo.
 - Si no existen soluciones reales (es decir, si el discriminante de la ecuación es negativo), significa que el segmento no toca el círculo, por lo que la línea de visión está completamente libre.
 - Si hay soluciones reales, se calcula en qué puntos el segmento intersecta la circunferencia. Se verifica que estos puntos estén dentro del segmento que une *“pointA”* con *“pointB”*, ya que no interesa si la línea prolongada interseca fuera de ese rango.
3. Distancia desde el obstáculo al segmento: En caso de intersección, se calcula la distancia perpendicular desde el centro del círculo hasta la línea recta. Esta información es útil para medir el grado de “obstrucción” de la línea de visión.
4. Selección de los puntos de intersección: Si se detecta que el obstáculo interfiere con la línea de visión, se identifican los dos puntos donde el segmento corta la circunferencia. Estos

puntos son devueltos por la función ordenados según su proximidad al punto A, permitiendo conocer dónde empieza y termina la interferencia visual.

En función de los resultados obtenidos del análisis, la función devuelve:

- “None”: si la línea entre “pointA” y “pointB” está completamente libre de obstáculos (línea de visión clara).
- Una tupla (d, “ipointA”, “ipointB”) si hay una colisión, donde:
 - d es la distancia entre el centro del círculo y la línea recta.
 - “ipointA” es el punto de intersección más cercano a “pointA”.
 - “ipointB” es el punto de intersección más cercano a “pointB”.

La detección de líneas de visión tiene múltiples aplicaciones prácticas dentro del proyecto:

- Estrategias defensivas: permite a los drones defensores detectar si pueden avanzar en línea recta hacia un atacante sin colisionar con un obstáculo.
- Planificación de rutas: ayuda al sistema a decidir si debe buscar una ruta alternativa para evitar una colisión.
- Evaluación de visibilidad: facilita el análisis acerca de qué zonas del mapa están cubiertas o bloqueadas por obstáculos, lo que puede ser clave para posicionar unidades estratégicamente.

En resumen, la función “computeLineOfSight” es una herramienta esencial para la simulación realista de escenarios con obstáculos. Gracias a su implementación, los drones virtuales pueden tomar decisiones basadas en la presencia o ausencia de visibilidad directa, lo que incrementa el realismo y la utilidad de los resultados obtenidos. Su correcto funcionamiento permite detectar colisiones potenciales y garantizar una navegación segura y eficiente, algo especialmente relevante en entornos con múltiples agentes móviles y obstáculos distribuidos aleatoriamente.

Como se puede observar en la Figura 3-6, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta de línea de visión directa de manera mediante cálculos geométricos.

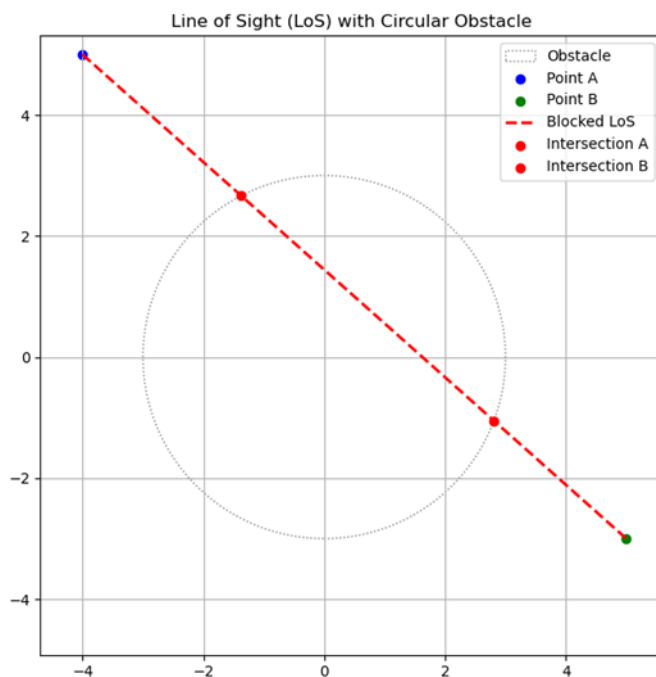


Figura 3-6.- Representación de la ruta de línea de visión directa

3.3.2 Cálculo de tangentes externas entre dos circunferencias

En este apartado se describe la implementación de una función matemática cuya finalidad es calcular los puntos de tangencia entre dos circunferencias, cuando dichas tangentes son externas. Este tipo de cálculo resulta fundamental en el contexto de planificación de trayectorias para drones, ya que permite definir rutas seguras que eviten colisiones con zonas restringidas o peligrosas representadas mediante circunferencias.

La función *“computeCircleToCircleExternalTangets”* calcula los cuatro puntos que definen las dos tangentes externas entre dos circunferencias dadas. Como entrada, esta función requiere las siguientes características de las circunferencias:

- *“circleA”*: coordenadas del centro ($c1x$, $c1y$) y radio $r1$ de la primera circunferencia.
- *“circleB”*: coordenadas del centro ($c2x$, $c2y$) y radio $r2$ de la segunda circunferencia.

El resultado de esta función es una tupla de cuatro puntos:

- Dos puntos sobre la circunferencia A, que representan los lugares donde las tangentes tocan esta circunferencia.
- Dos puntos sobre la circunferencia B, que representan los puntos de tangencia correspondientes.

Estos puntos permiten trazar las dos líneas tangentes externas que pasan tangencialmente por los extremos de ambas circunferencias sin atravesarlas.

A continuación, se explica el funcionamiento paso a paso:

1. Cálculo de la distancia entre los centros: Se calcula la distancia entre los centros de ambas circunferencias mediante el teorema de Pitágoras. Esta distancia es fundamental para determinar si las tangentes externas existen.
2. Comprobación de la existencia de tangentes: Si una de las circunferencias está completamente dentro de la otra (es decir, si la distancia entre los centros es menor que la diferencia de radios), entonces no existen tangentes externas, ya que cualquier línea que conecte las circunferencias pasará necesariamente por su interior. En ese caso, la función devuelve *“None”* (la función determina que no hay una solución geométrica válida, no se han podido calcular las tangentes).
3. Se calculan los ángulos para las rectas tangentes:
 - *“Alpha”* (α): es el ángulo que indica la dirección desde el centro de la primera circunferencia hasta el centro de la segunda.
 - *“Theta”* (θ): es el ángulo de desviación necesario para llegar a los puntos de tangencia desde el centro de cada circunferencia.
4. Determinación de los puntos de tangencia: Con los ángulos anteriores se calculan las coordenadas exactas de los puntos de tangencia sobre cada circunferencia utilizando funciones trigonométricas (coseno y seno).

Se generan así dos tangentes externas:

- La primera tangente se calcula sumando *“Theta”* al ángulo *“Alpha”*.
- La segunda tangente se calcula restando *“Theta”* a *“Alpha”*.

Este cálculo resulta muy útil en situaciones en las que los drones deben evitar determinadas zonas circulares en su trayectoria. Por ejemplo:

- Eludir obstáculos: si una zona peligrosa se representa como una circunferencia, las tangentes externas permiten diseñar trayectorias que mantengan al dron fuera de esa zona.
- Planificación cooperativa: permite a varios drones calcular rutas independientes que pasen alrededor de áreas compartidas sin colisionar entre sí.
- Optimización de rutas: las tangentes externas suelen ser trayectorias más cortas que otras rutas para evitar obstáculos.

La función *“computeCircleToCircleExternalTargets”* representa una herramienta clave para la navegación segura en entornos donde los drones deben evitar áreas específicas. Gracias a su uso, es posible construir rutas precisas y eficientes que optimicen la movilidad y reducen el riesgo de colisión. Su integración en el sistema de planificación de trayectorias contribuye directamente a la robustez y realismo de las simulaciones desarrolladas en este proyecto.

Como se puede observar en la Figura 3-7, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta de tangentes externas entre dos circunferencias.

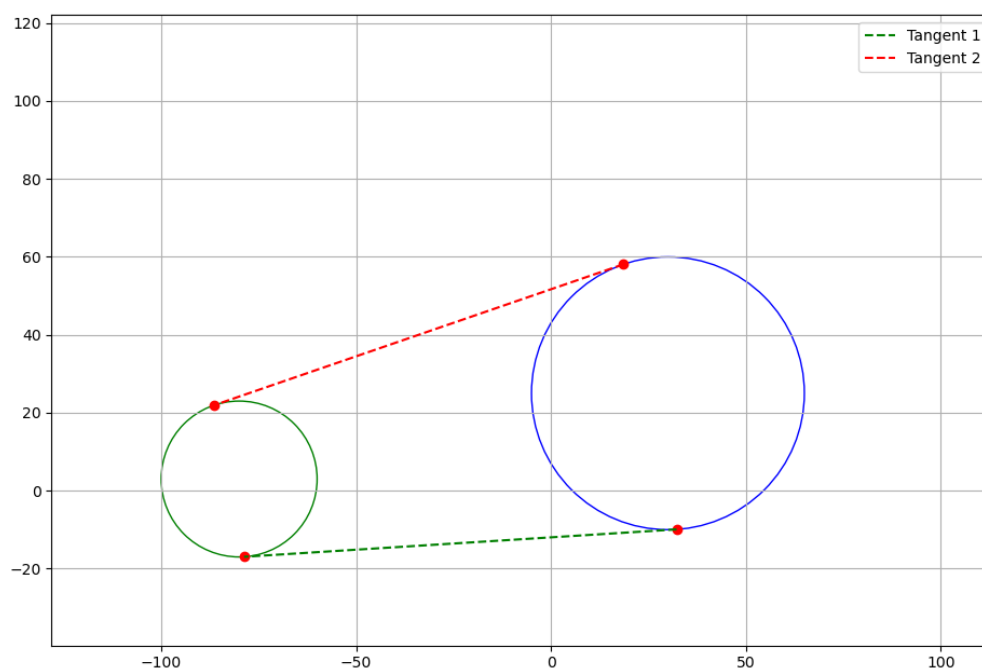


Figura 3-7.- Representación gráfica de tangentes externas entre dos circunferencias

3.3.3 Cálculo de tangentes internas entre dos circunferencias

En determinados escenarios de navegación y planificación de rutas, es necesario determinar si es posible trazar una trayectoria que toque dos obstáculos circulares sin atravesarlos, pero pasando entre ellos. Estas trayectorias se conocen como tangentes internas y son útiles, por ejemplo, cuando un dron necesita pasar por un corredor estrecho entre dos zonas restringidas.

La función *“computeCircleToCircleInternalTargets”* está diseñada para calcular las tangentes internas comunes a dos circunferencias, es decir, aquellas líneas que:

- Rozan cada circunferencia en un solo punto.
- Se sitúan entre ambas circunferencias, tocándolas desde dentro y sin cruzarlas.

Esta función requiere como entrada dos circunferencias (*“circleA”* y *“circleB”*), cada una definida mediante tres valores:

- La coordenada x del centro.
- La coordenada y del centro.
- El radio de la circunferencia.

Estos se representan mediante tuplas del tipo (x, y, r) .

Antes de comenzar con la de los cálculos, la función verifica si es matemáticamente posible que existan tangentes internas entre las dos circunferencias. Para que existan:

- La distancia entre los centros de las circunferencias debe ser mayor que la suma de sus radios.
- Si no se cumple esta condición, significa que las circunferencias se solapan o una está contenida dentro de la otra, lo que hace imposible que exista una tangente interna. En este caso, la función imprime un mensaje de advertencia y devuelve el valor especial *“None”*, que en programación representa que no se ha encontrado una solución válida.

Cuando las condiciones son adecuadas, la función procede a calcular las tangentes internas siguiendo los siguientes pasos:

- Cálculo de la dirección entre los centros: Se determina el ángulo de la línea que une los centros de ambas circunferencias.
- Cálculo del ángulo de inclinación: A partir de la distancia entre los centros y la suma de los radios, se calcula el ángulo necesario para definir la orientación de las tangentes internas.
- Obtención de los puntos de tangencia:
 - Para cada circunferencia, se calcula el punto exacto en el borde donde la línea tangente intersecta.
 - Se obtienen así cuatro puntos: dos en la primera circunferencia (A) y dos en la segunda (B), que definen las dos tangentes posibles.

La función devuelve un resultado con cuatro coordenadas, que representan los dos pares de puntos donde las tangentes internas tocan las circunferencias. En términos prácticos, esto permite dibujar dos líneas tangentes internas que conectan *“circleA”* y *“circleB”* sin penetrar en su interior.

Este tipo de cálculo resulta especialmente útil en la planificación de trayectorias seguras para vehículos no tripulados, como los drones, ya que permite determinar caminos que pasan entre obstáculos sin colisionar con ellos. Estas tangentes internas pueden servir como rutas alternativas cuando no hay una línea de visión directa disponible o cuando se busca una estrategia de evasión eficiente.

Como se puede observar en la Figura 3-8, la función que se ha definido con los parámetros indicados anteriormente permite generar una figura de tangentes internas entre dos circunferencias.

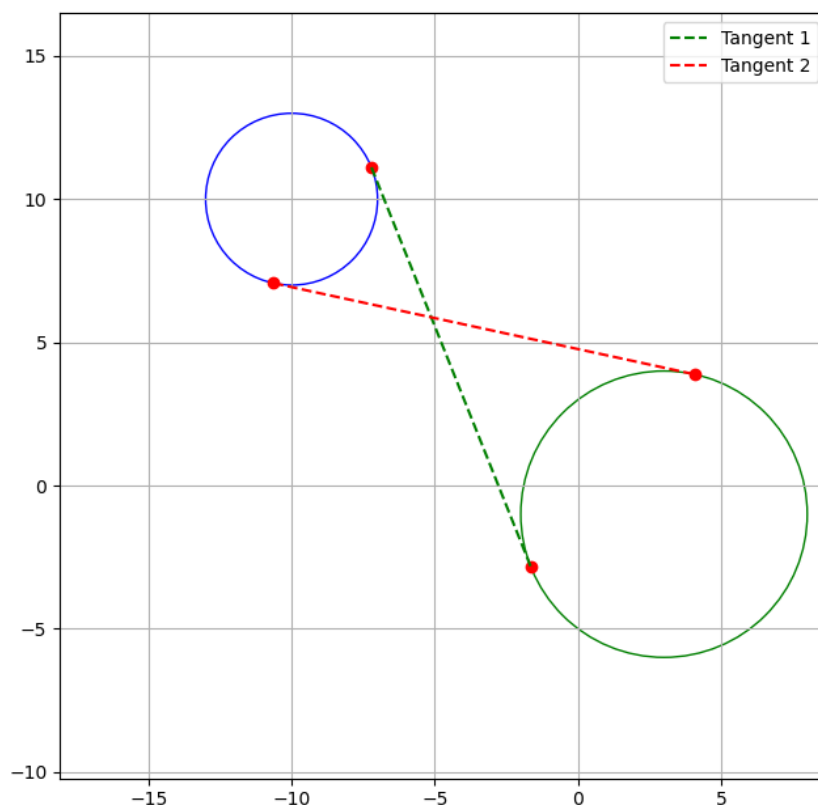


Figura 3-8.- Representación gráfica de tangentes internas entre dos circunferencias

3.3.4 Cálculo de tangentes desde un punto externo a una circunferencia

En el contexto del presente trabajo, centrado en la planificación de trayectorias seguras para drones autónomos, resulta fundamental determinar qué caminos pueden trazarse desde un punto determinado hasta los límites de una zona restringida o de peligro representada mediante una circunferencia. Una herramienta clave para este tipo de análisis es el cálculo de tangentes desde un punto exterior a una circunferencia.

La función *“computePointToCircleTangents”* tiene como finalidad determinar los puntos de tangencia desde un punto externo hasta una circunferencia. Es decir, identifica los dos únicos puntos del borde de la circunferencia en los que se pueden trazar líneas rectas desde el punto dado que toquen la circunferencia sin penetrarla. Estas líneas, denominadas tangentes, son especialmente útiles en entornos donde se requiere evitar zonas peligrosas sin alejarse excesivamente del objetivo.

La función requiere dos entradas:

- Un punto externo, definido por sus coordenadas (px, py) .
- Una circunferencia, definida por su centro (cx, cy) y su radio r .

Estas entradas permiten situar tanto el punto como la circunferencia en el espacio.

Antes de realizar ningún tipo de cálculo, la función evalúa si el punto se encuentra dentro de la circunferencia. Esto se hace midiendo la distancia entre el punto y el centro del círculo:

- Si esta distancia es menor que el radio, el punto está dentro del círculo, y por tanto no existen tangentes posibles desde ese punto, ya que toda línea que se trace desde el interior cortará el círculo en lugar de tocarlo tangencialmente. En este caso, la función devuelve el valor *“None”*, lo que indica que no hay solución válida.

En caso de que el punto se encuentre fuera de la circunferencia, la función procede al cálculo de los dos puntos de tangencia:

- Cálculo de la dirección: se determina el ángulo que forma la línea entre el punto externo y el centro de la circunferencia.
- Ángulo de tangencia: se calcula el ángulo que forma cada tangente con respecto a la línea central. Este valor depende de la proporción entre el radio del círculo y la distancia al punto externo.
- Determinación de los puntos de tangencia: se aplican funciones trigonométricas para encontrar las coordenadas exactas de los dos puntos sobre la circunferencia en los que las tangentes tocan la misma. Estos dos puntos son simétricos con respecto a la línea central que une el punto externo con el centro del círculo.

La función devuelve una tupla que contiene las coordenadas de los dos puntos de tangencia. Estos valores pueden emplearse para:

- Dibujar visualmente las líneas tangentes.
- Determinar rutas posibles que bordean la circunferencia.
- Evaluar alternativas en estrategias de evasión o planificación de trayectorias seguras.

Esta función contribuye directamente al diseño de sistemas de navegación inteligente. Concretamente, permite a un dron evaluar posibles rutas que eviten colisiones con obstáculos representados mediante circunferencias. Estas tangentes se convierten en alternativas óptimas de paso cuando un obstáculo bloquea una trayectoria directa hacia el destino. Asimismo, permiten implementar decisiones más sofisticadas en entornos dinámicos o densos, donde los márgenes de maniobra son limitados.

Como se puede observar en la Figura 3-9, la función que se ha definido con los parámetros indicados anteriormente permite generar una figura de tangentes desde un punto externo a una circunferencia.

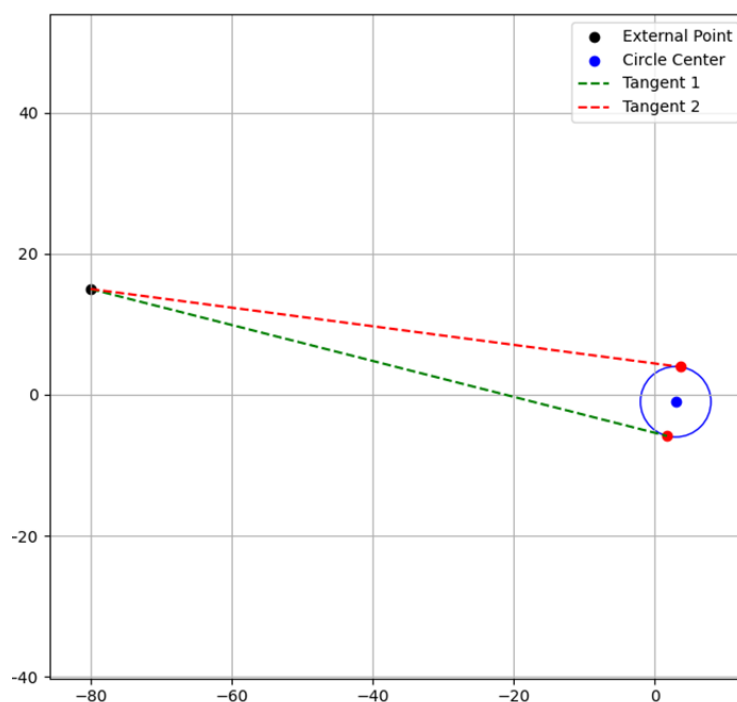


Figura 3-9.- Representación gráfica de tangentes desde un punto externo a una circunferencia

3.3.5 Mapa de visibilidad de tangentes

Uno de los retos principales en la navegación autónoma de drones en entornos con obstáculos es identificar rutas seguras que eviten colisiones con dichos obstáculos. En este contexto, los obstáculos se representan mediante circunferencias, y el objetivo es encontrar trayectorias tangenciales que conecten los bordes de los obstáculos sin interferencias. Para abordar esta tarea, se ha implementado una función denominada “*generateTangentVisibilityMap*”, que calcula el conjunto de líneas tangentes (visibilidad tangencial) que pueden trazarse entre pares de obstáculos, siempre que no exista obstrucción visual por parte de otros elementos del entorno.

Esta función tiene como finalidad construir un mapa de visibilidad tangencial, es decir, un conjunto de trayectorias que conectan los bordes de dos circunferencias diferentes mediante líneas tangentes. Se consideran tanto tangentes internas como externas entre los pares de circunferencias, y únicamente se conservan aquellas cuya línea de visión no está bloqueada por ninguna otra circunferencia del entorno. Este mapa resulta esencial para la planificación de trayectorias eficientes en entornos congestionados, ya que proporciona rutas alternativas que bordean los obstáculos con seguridad.

Esta función recibe como parámetros de entrada los siguientes:

- “*circles*”. una lista de circunferencias representadas como tuplas de la forma (cx, cy, r) , donde cx y cy son las coordenadas del centro, y r es el radio.
- “*risk_distance*”. una distancia de seguridad adicional que se suma al radio de cada circunferencia para garantizar una separación segura entre el dron y los obstáculos.

El proceso que sigue la función se explica a continuación en los siguientes pasos:

- Inicialización: se crean dos listas vacías para almacenar las tangentes internas y externas que se consideran válidas.
- Evaluación de pares de circunferencias: la función analiza todas las combinaciones posibles de pares de circunferencias, evitando duplicidades y autoevaluaciones.
- Se realiza el cálculo de tangentes internas de la siguiente manera:
 - Para cada par de circunferencias, se calculan las tangentes internas ajustando el radio de cada una de ellas con el parámetro “*risk_distance*”.
 - Si estas tangentes existen (es decir, si las circunferencias no se solapan), se comprueba si el trayecto que une cada par de puntos tangentes está libre de obstrucciones, es decir, si no pasa por el interior de ninguna otra circunferencia del entorno.
 - Si la línea de visión está libre, esa tangente se añade a la lista de tangentes internas válidas.
- Se realiza el cálculo de tangentes externas de la siguiente manera:
 - Se repite el mismo procedimiento que se realiza para calcular las tangentes internas.
 - Si detecta alguna tangente con obstrucción (bloqueo visual por otro obstáculo), se descarta.
- Al finalizar, la función devuelve dos listas: una con las tangentes internas válidas y otra con las externas. Cada tangente se representa como un par de puntos: el punto de inicio en una circunferencia y el punto de destino en otra.

Este mapa de visibilidad tangencial tiene múltiples aplicaciones dentro del proyecto:

- Evitar zonas de riesgo: permite al sistema de navegación calcular rutas que rodeen los obstáculos en lugar de internar atravesarlos directamente.

- Mejora de la eficiencia: ofrece caminos alternativos más cortos y directos que no comprometen la seguridad del dron.
- Flexibilidad en la planificación: al disponer de un conjunto amplio de opciones de navegación entre obstáculos, el sistema puede adaptarse mejor a cambios dinámicos en el entorno.

La función “*generateTangentVisibilityMap*” es un componente clave en la construcción de algoritmos de navegación seguros y adaptativos. Su capacidad para identificar rutas tangenciales válidas entre obstáculos mejora significativamente la capacidad del dron para maniobrar de forma segura y eficiente en entornos complejos, cumpliendo así uno de los principales objetivos de este Trabajo de Fin de Grado.

Como se puede observar en la Figura 3-10, la función que se ha definido con los parámetros indicados anteriormente permite generar la figura de un mapa de visibilidad de tangentes.

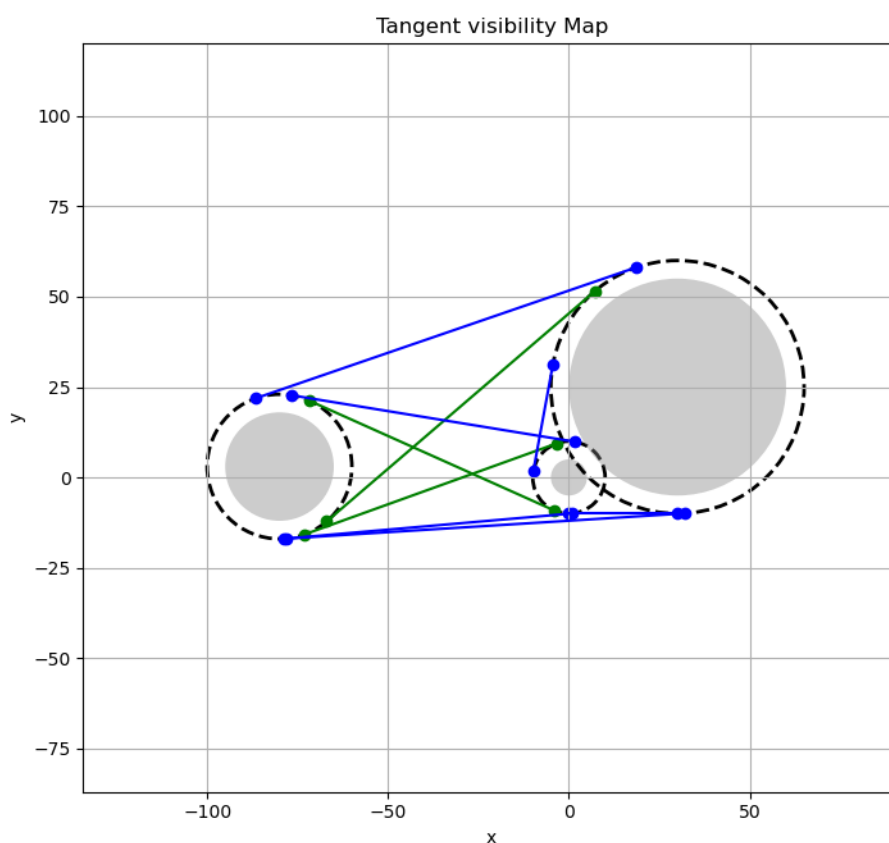


Figura 3-10.- Representación gráfica de un mapa de visibilidad de tangentes entre varias circunferencias

Por último, la integración de la comunicación entre el dron y la estación de control terrestre puede proporcionar información adicional sobre el estado del sistema y permitir intervenciones en caso de emergencia. Esta comunicación puede realizarse mediante protocolos como MAVLink, que permiten la transmisión de datos de telemetría y comandos de navegación.

En conclusión, la navegación autónoma de un dron en entornos con obstáculos es un problema multidimensional que involucra la combinación de técnicas de percepción, planificación de trayectorias, control de vuelo y comunicación en tiempo real. La correcta integración de estos componentes permitirá al dron desplazarse de manera eficiente y segura garantizando el cumplimiento de los objetivos establecidos en la simulación de los escenarios.

3.4 Rutas óptimas. El algoritmo A*

La búsqueda de caminos es un problema que usualmente se resuelve por medio de algoritmos de búsqueda en grafos como *width-first*, *depth-first*, *Dijkstra* y *A**.

En este tipo de problemas se conoce un estado inicial, un estado final (meta) y un conjunto de reglas mediante las cuales se pueden realizar “movidas” para ir avanzando en un grafo compuesto por nodos. Eventualmente, avanzando por el grafo, se llega al nodo que corresponde a la meta habiendo finalizado la búsqueda y habiendo obtenido el conjunto de movidas que nos lleva de un punto a otro.

El algoritmo A* es uno de los preferidos para este tipo de aplicaciones. Contar con información sobre un espacio de estado evita a los algoritmos emprender búsqueda a ciegas, permitiendo así encontrar soluciones con más eficiencia.

A* es un algoritmo de búsqueda inteligente o respaldado por información que busca el camino más corto desde un estado inicial al estado objetivo a través de un espacio de problema usando una heurística óptima.

Pertenece al método de búsqueda preferente por lo mejor (*Best First Search*). Es muy popular en la planificación de rutas y ha sido exitosamente utilizado en áreas como inteligencia artificial y robótica.

A* es un algoritmo genérico de búsqueda que utiliza información heurística para determinar cuál es el mejor camino hacia el destino. La información que utiliza es el costo estimado desde el nodo explorado hacia el nodo destino. Esta información es proporcionada por la función llamada función heurística $h'(n)$.

A* es un algoritmo que basa su comportamiento en la evaluación de una función expresada del siguiente modo:

$$f(n) = g(n) + h(n)$$

Función de evaluación:

- $g(n)$: costo del mejor camino desde el estado inicial al estado n .
- $h(n)$: estimación (heurística) del coste desde n hasta un estado final óptimo.
- $f(n)$: costo estimado de la mejor solución que pasa por el estado n .

Este tipo de algoritmo es óptimo y completo si:

- Todo nodo tiene un n° finito de sucesores.
- El costo de cada arco u operador es mayor de cero.
- La función $h(n)$ es una heurística admisible.

Es óptimo porque encuentra la mejor solución si existen varias. Es complejo porque garantiza encontrar una solución, si existe. Para un correcto funcionamiento del algoritmo debe cumplirse que $h(n)$ sea una heurística admisible que nunca sobreestima el coste real desde n hasta un estado meta óptimo.

El funcionamiento de este algoritmo consiste en incorporar la longitud del camino desde la raíz hasta el estado actual en la función de evaluación h , considera no sólo lo bueno que es un estado si no que tiene en cuenta cómo es el camino usado para alcanzarlo [26].

Para resolver el problema que se presenta en esta fase es necesario aplicar un algoritmo que es capaz de evaluar la situación dada, calcular todos los caminos posibles entre dos puntos dados (A y B), evaluar dichos caminos y, de entre todos, escoger los óptimos para el desplazamiento de los drones con los que se pretende llevar a cabo este ejercicio.

Una de las situaciones en las que se aplica dicho algoritmo es, para el caso de los drones atacantes, cuando la ruta que se le ordena seguir desde el punto de origen del atacante al punto de destino pasa a través de un obstáculo.

Es entonces cuando se aplica el algoritmo A^* , para encontrar la ruta óptima que bordeé al obstáculo y así generar una nueva ruta capaz alcanzar el punto objetivo sin generar un impacto con el obstáculo.

En el caso de los drones defensores este algoritmo se aplica desde el principio para que el dron defensor alcance de la manera óptima al dron atacante o en su caso el punto de destino que la estrategia defina como objetivo para cada defensor, el cual se propone como un intento de reducir tanto el tiempo de reacción como las posibilidades de que el dron atacante cumpla con su objetivo.

A la hora de realizar la implementación de este algoritmo en código, es necesario indicar que existen dos formas:

- Se genera un grafo total del problema que se desea solucionar y se envía este grafo a la función del algoritmo A^* .
- En la llamada a la función del algoritmo A^* se indica cual es la función de devuelve los nodos vecinos para un nodo dado, además de como calcular el valor de la función heurística $h(n)$. En este trabajo se emplea una librería existente que emplea este método, lo cual permite trabajar en entornos muy complejos en donde el grafo a generar podría tener un tamaño excesivo.

A continuación, se presentan algunos ejemplos de la integración de las rutas geométricas, vistas en el apartado anterior, con el uso del algoritmo A^* para modificar esas rutas iniciales en escenarios con obstáculos de forma que los eviten, lo cual permitirá generar las trayectorias deseadas para los drones atacantes o modificar las rutas dinámicas que deben emplear los defensores.

3.4.1 Ruta circular con obstáculos

En este escenario, se parte de una configuración inicial en la que la trayectoria base del dron sigue la ruta circular mencionada en los apartados anteriores. Sin embargo, para simular un entorno más complejo y realista, se han incorporado una serie de obstáculos distribuidos estratégicamente en el área de navegación. Estos obstáculos representan elementos que el dron debe detectar y evitar durante su desplazamiento, lo que introduce la necesidad de implementar el algoritmo A^* (“A-star”), que garantizará que el dron sea capaz de sortear los obstáculos de manera eficiente y alcanzar su punto de destino.

Este proceso garantiza que la trayectoria generada sea óptima y libre de colisiones, ajustando dinámicamente la ruta en función de la disposición de los obstáculos.

La aplicación de A^* en este contexto no solo mejora la eficiencia de sistema de navegación autónoma, sino que también sienta las bases para la implementación de estrategias avanzadas de planificación en entornos dinámicos.

Para este escenario se importan las funciones “*computeCirclePath*” y “*plotCirclePath*” para poder calcular la trayectoria circular inicial entre un punto de origen y otro punto de destino, y poder visualizarla en un gráfico (la ruta pasa a través de los obstáculos). Ahora se calcula la ruta con evitación de obstáculos generando una nueva ruta optimizada que los evita utilizando un *Grid Map* (mapa de rejilla). La función que se utiliza para ello es la siguiente:

“*computePathWithObstacleAvoidanceGridMap*”, la cual tiene en cuenta los siguientes parámetros:

- Punto de origen y de destino (“*origin_point*, *target_point*”).
- Ruta original (“*path_points*”).
- Lista de obstáculos (“*obstacles*”).
- Parámetros del entorno (“*parameters*”).

Como resultado se obtiene una nueva lista de puntos de la nueva trayectoria optimizada evitando los obstáculos (*“new_path_points”*), así los puntos intermedios de impacto de la trayectoria original utilizados en la corrección que evita los obstáculos (*“bridge_points”*).

A continuación, se actualiza el gráfico para mostrar la nueva trayectoria que evita los obstáculos la cual se encarga de llevar a cabo las siguientes acciones:

- Dibujar la nueva ruta generada (*“new_path_points”*).
- Representar los obstáculos en el gráfico (*“obstacles”*).
- Incluir los puntos intermedios de ajuste (*“bridge_points”*).

Como se puede observar en la Figura 3-11, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta circular que evita los obstáculos:

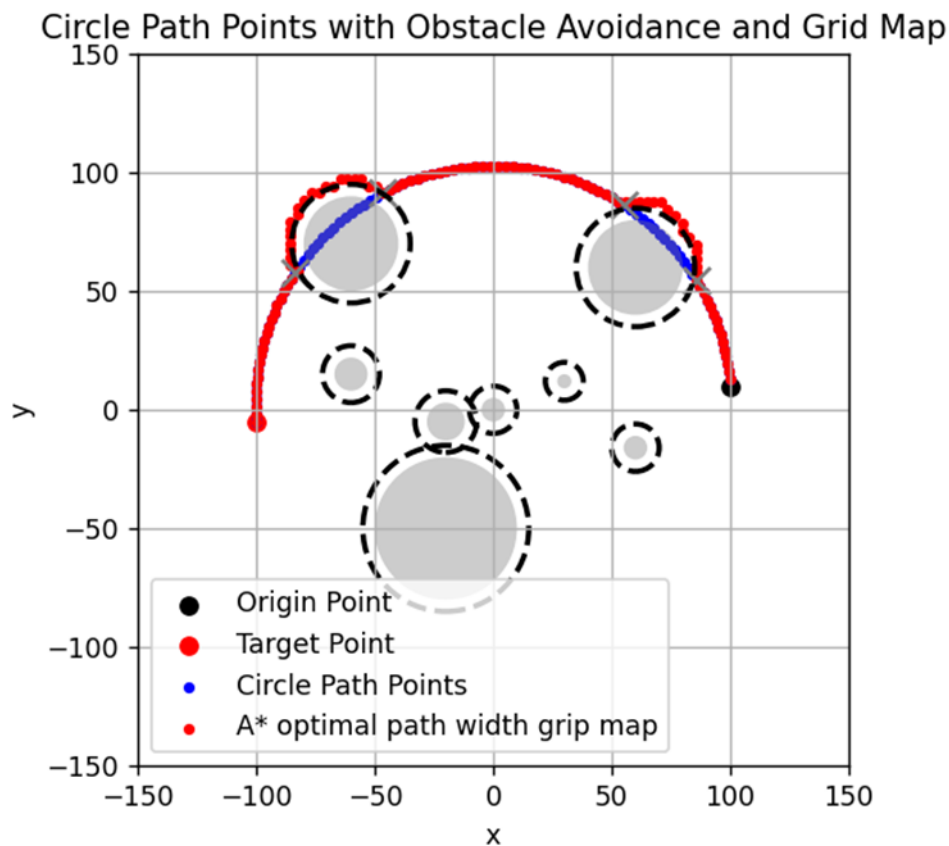


Figura 3-11.- Resultado obtenido con la aplicación del algoritmo “A-star” para obtener una ruta circular que evada los obstáculos

El código completo se encuentra en el [Anexo III](#), donde se detallan los cálculos y la implementación en *Python*.

3.4.2 Ruta sinusoidal con obstáculos

Este código tiene el mismo propósito que el anterior, pero en lugar de generar una ruta circular, se genera una trayectoria en forma de onda sinusoidal (tipo seno o sinusoidal). Este tipo de trayectorias se usan comúnmente en navegación autónoma, simulaciones de movimiento y planificación de rutas en terrenos irregulares.

Se siguen los mismos pasos que para el caso anterior, pero ahora las funciones son *“computeSinePath”*, con la que se calcula la trayectoria en forma de onda sinusoidal desde el punto de

origen hasta el punto de destino sin tener en cuenta los obstáculos, y la función “*plotSinePath*”, la cual dibuja la trayectoria generada por la función anterior.

Para realizar la visualización se usa la función “*plotSinePath*”, la cual dibuja la trayectoria desde el punto inicial hasta el punto de destino sin tener en cuenta los obstáculos, es decir, atravesando obstáculos si se diese el caso.

Para evitar esta última situación, los obstáculos, se aplica el mismo procedimiento que en el punto anterior para obtener la ruta modificada y dibujar el resultado comparándolo con la ruta geométrica original.

Como se puede observar en la Figura 3-12, la función que se ha definido con los parámetros indicados anteriormente permite generar una ruta sinusoidal que evita los obstáculos presentes en el escenario:

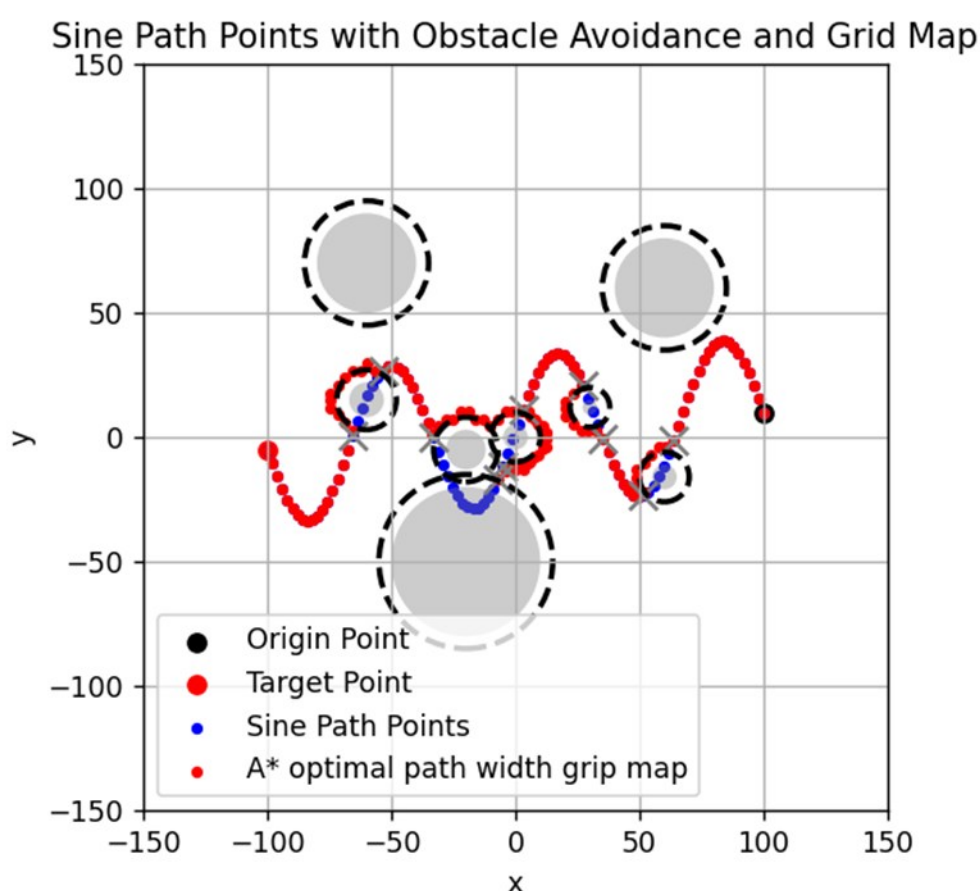


Figura 3-12.- Resultado obtenido con la aplicación del algoritmo “*A-star*” para obtener una ruta sinusoidal que evada los obstáculos

El código completo se encuentra en el [Anexo III](#), donde se detallan los cálculos y la implementación en Python.

3.5 Trayectorias de referencia para atacantes

En este estudio, los drones atacantes siguen una serie de rutas predefinidas, diseñadas para garantizar un desplazamiento eficiente y adaptativo dentro del entorno simulado. Para optimizar su trayectoria y asegurar que puedan evadir obstáculos de manera efectiva, se ha implementado el algoritmo A* (“A-star”), mencionado previamente en este trabajo.

El proceso de planificación de rutas se lleva a cabo en varias fases:

- Generación de la trayectoria predefinida: Se establecen puntos de referencia iniciales y finales para cada dron atacante, determinando una ruta geométrica base en ausencia de obstáculos.
- Aplicación del algoritmo A* ("A-star"): A partir de la ruta predefinida, el algoritmo A* recalcula la trayectoria para evitar obstáculos en el camino, ajustando dinámicamente la ruta en función de las condiciones del entorno.
- Ejecución y seguimiento de la trayectoria optimizada: una vez determinada la ruta óptima, el dron atacante sigue el camino generado, adaptándose en tiempo real a posibles variaciones en la disposición de los obstáculos.

El uso de A* en este contexto no solo mejora la capacidad de los drones atacantes para sortear obstáculos, sino que también optimiza el tiempo de llegada a su destino, garantizando una navegación más eficiente. Asimismo, la implementación de este algoritmo permite evaluar distintos escenarios en los que los obstáculos pueden ubicarse de manera aleatoria o en configuraciones estratégicas, proporcionando información clave sobre el rendimiento del sistema en condiciones dinámicas y adversas.

A continuación, se exponen dos imágenes con el objetivo de comparar la ruta en la que se no se aplica el algoritmo A* y la ruta en la que sí se aplica el mismo:

- Ruta en la que no se aplica el algoritmo A*:

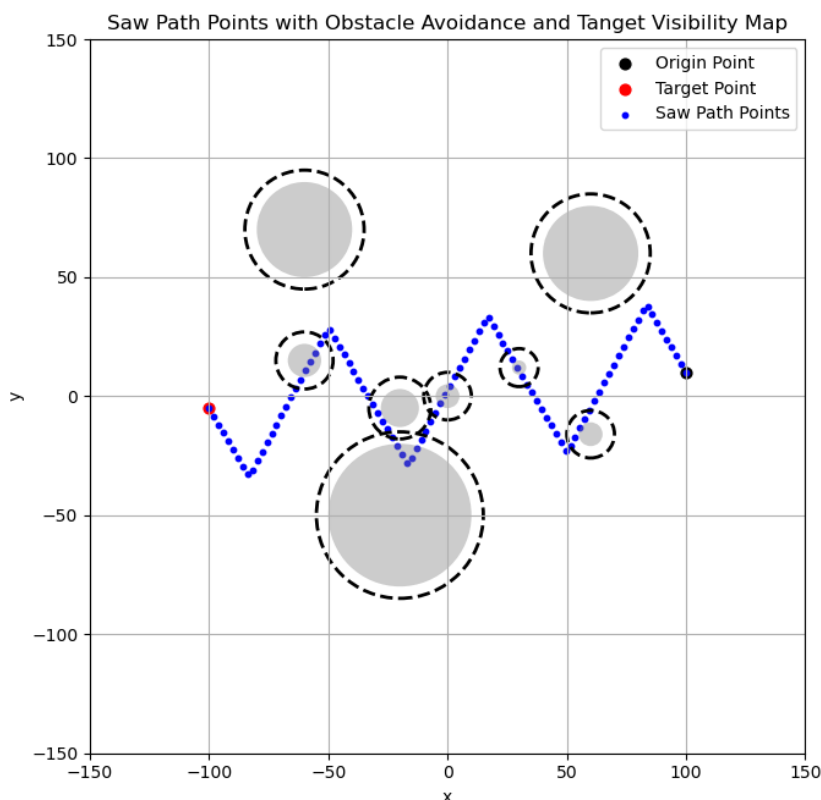


Figura 3-13.- Ruta en diente de sierra a la que no se le aplica el algoritmo A*

Como se puede observar en la Figura 3-13, el dron que quiere ir desde el punto de origen al punto de destino debe llevar a cabo una ruta la cual atraviesa seis obstáculos, impidiendo a la hora de la simulación que esta misma se lleve a cabo.

A continuación, se muestra el resultado de aplicar el algoritmo A* a esta misma ruta para que el dron pueda evitar los obstáculos que sean necesarios y cumplir con el objetivo:

- Ruta en la que se aplica algoritmo A*:

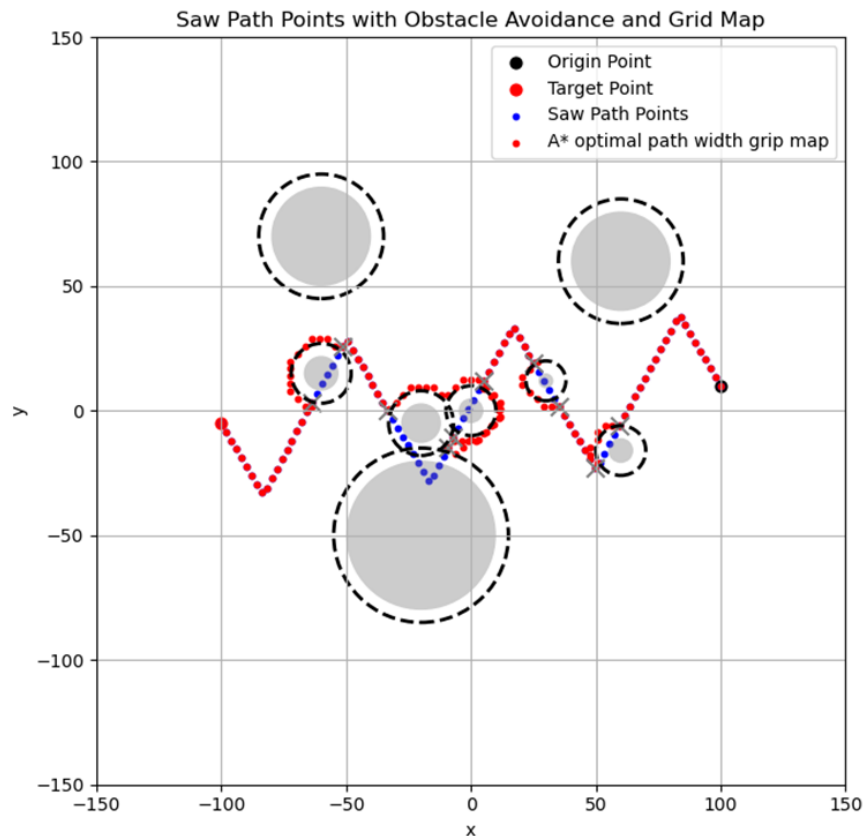


Figura 3-14.- Representación gráfica de la ruta en diente de sierra a la que se aplica el algoritmo A*

Como se observa en la Figura 3-14, al aplicar el algoritmo A* la ruta evade los obstáculos haciendo que el dron tome el camino óptimo a la hora de sortear los obstáculos

3.6 Planificación de rutas defensivas.

En el contexto del proyecto actual, la tarea que debe ejecutar cada dron defensor no solo viene determinada en función de cuál es su asignación a un atacante específico, sino que también se determina en cada instante en base a establecer a qué punto del escenario debe dirigirse. Esta toma de decisiones puede emplear en múltiples factores, tales como:

- **La posición y velocidad del atacante asignado**, para prever su trayectoria y definir el mejor punto de intercepción.
- **La ubicación de otros drones defensores**, con el fin de evitar redundancias y mejorar la distribución del equipo.
- **La presencia de obstáculos en el entorno**, garantizando que las trayectorias sean viables y seguras.

A partir de esta información, cada defensor debe establecer cuales su punto de destino, de forma que la ruta a seguir será establecida por el algoritmo A* que se ha visto anteriormente, para determinar en cada instante cual es la ruta óptima planificada para un defensor hasta su punto de destino.

Es importante destacar que se considera que los drones atacantes poseen una capacidad de maniobra superior a la de los drones defensores, especialmente en términos de velocidad máxima. Esta asimetría en las capacidades impone restricciones adicionales al diseño de las estrategias defensivas, lo que obliga

a optimizar la asignación de recursos y la cooperación entre los defensores para maximizar su desempeño.

En este apartado se presentan y analizan las diversas estrategias de defensa diseñadas para contrarrestar posibles amenazas dentro del entorno simulado. La correcta planificación y evaluación de estas estrategias resultan fundamentales para comprender el comportamiento y la efectividad de los drones defensores frente a los drones atacantes, cuyo objetivo es alcanzar un punto de alto valor estratégico. Cada una de las estrategias formuladas emplea un conjunto de parámetros predefinidos, los cuales se determinan a partir de la información disponible en cada instante de la simulación. Estos parámetros se complementan en cada instante con la posición y velocidad de los drones atacantes, la ubicación inicial y dinámica de los drones defensores, así como la disposición de obstáculos en el entorno. Con base a estos datos, se generan trayectorias y patrones de movimiento que los defensores deben seguir a lo largo del experimento. Dichos parámetros han sido diseñados para garantizar la flexibilidad del modelo ante diferentes escenarios y permitir un análisis posterior basado en métricas cuantificables.

En todas las estrategias evaluadas se asume que los drones atacantes siguen rutas previamente calculadas que consideran la presencia de obstáculos, con el propósito de alcanzar su destino de manera eficiente. Por otro lado, los drones defensores se posicionan inicialmente en ubicaciones aleatorias dentro del área de simulación y despliegan tácticas de intercepción y contención para evitar que los atacantes logren su objetivo. A cada instante, se establece una asignación de objetivos, donde cada dron defensor tiene un atacante específico asignado. Si varios defensores comparten el mismo objetivo, forman un enjambre que se debe coordinar de forma adecuada para incrementar la efectividad de la defensa.

Las estrategias de defensa implementadas en este estudio han sido desarrolladas en función de distintos enfoques, cada uno con ventajas y limitaciones específicas. A continuación, se presentan los principales modelos analizados establecidas en función del punto de destino de cada dron defensor:

- **Estrategias de intercepción directa:**

En este enfoque, los drones defensores se dirigen inmediatamente hacia los atacantes desde su posición inicial, intentando priorizar la minimización del tiempo de respuesta y la distancia recorrida por el dron atacante en el escenario.

- **Estrategias basadas en barreras defensivas:**

En lugar de perseguir directamente a los atacantes, los defensores establecen una línea de contención alrededor de una zona crítica, actuando como una barrera protectora. Esta táctica busca dificultar el avance enemigo intentando incrementar así la probabilidad de neutralización.

- **Estrategias dinámicas adaptativas:**

En este caso, los drones defensores ajustan su posición y trayectoria en función del movimiento de los atacantes. De esta forma se trata de estrategias adaptativas que combinando diferentes ideas que generen estructuras de defensa cuyas posiciones cambian en función de la posición del atacante de forma dinámica.

Cada una de estas estrategias será evaluada mediante simulaciones computacionales para medir su efectividad en distintos escenarios. Los principales factores considerados en el análisis incluyen:

- Velocidad de los drones atacantes y defensores.
- Densidad y distribución de obstáculos en el entorno.
- Número de unidades participantes en cada bando.
- Nivel de cooperación entre los drones defensores.
- Adaptabilidad de las estrategias a cambios inesperados en el comportamiento de los atacantes.

El análisis de los resultados obtenidos permitirá identificar los enfoques con mayor índice de éxito en la contención de amenazas. Además, con base en los hallazgos, se pueden proponer mejoras en el diseño de estrategias defensivas utilizando técnicas avanzadas de inteligencia artificial y optimización de trayectorias.

3.6.1 Estrategia 0

El objetivo principal de esta estrategia es crear la solución más simple posible al problema planteado que sirva de referencia para crear nuevas estrategias. Se define básicamente de forma que para cada defensor el punto de destino establecido es la posición actual del atacante asignado, es decir, los defensores persiguen directamente a los atacantes asignados en su enjambre. La Figura 3-15 representa el esquema básico de actuación para cada uno de los defensores. Esta estrategia es sencilla de implementar, pero debido a las diferentes capacidades de los defensores respecto a los atacantes, es muy sencillo que estos últimos logren escapar al no existir ningún tipo de coordinación entre defensores.

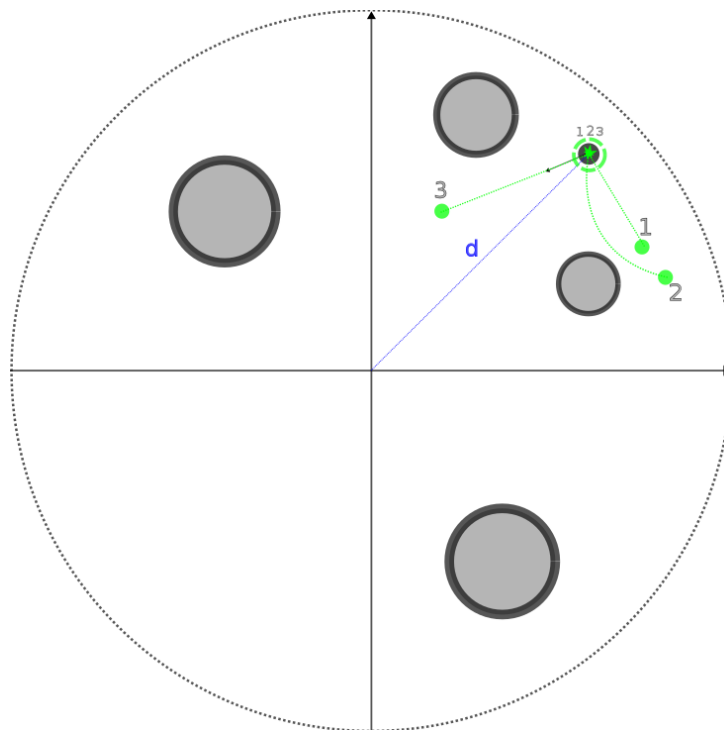


Figura 3-15.- Representación gráfica del escenario de la Estrategia 0

En el siguiente enlace y código QR puede accederse a un video que muestra la operativa de esta estrategia: https://drive.google.com/file/d/13XobdJ3L6_bPSGwuZUFLk4YDhX9voEdP/view?usp=drive_link



3.6.2 Estrategia 1

En este apartado se describe la implementación de la Estrategia 1, cuyo objetivo principal es optimizar la posición de los drones defensores en relación con la ubicación del atacante con respecto a

su punto de destino. A diferencia del enfoque más básico de la estrategia anterior, donde los defensores persiguen directamente al atacante, esta estrategia establece un radio defensivo, en el cual los drones se distribuyen estratégicamente alrededor del punto de destino que debe ser defendido de los atacantes, tratando de crear una barrera defensiva.

Para definir la configuración del escenario, se han utilizado los siguientes parámetros

- “*defensive_radius*”: es el radio de seguridad en el que deben posicionarse los defensores alrededor del punto de destino.
- “*collison_distance*”: es la distancia a la que se considera que se ha producido un impacto.

Para alcanzar sus objetivos, el sistema calcula dinámicamente:

- La distancia y ángulo del atacante con respecto al punto de destino.
- La distancia de los defensores al atacante.

A partir de esta información el algoritmo implementa la siguiente lógica:

- Si la distancia de un defensor al atacante está por debajo de un cierto valor umbral ($30 \times$ “*collison_distance*”), el defensor se dirige directamente al atacante.
- Si la distancia es superior a ese valor umbral, el defensor se dirige a un punto situado en la circunferencia que define el radio defensivo de valor constante. Cada dron defensivo de un mismo enjambre asignado a un atacante tendrá una posición ordenada en dicha circunferencia en función de la posición del dron en el enjambre, que es asignada de forma fija al inicio de la simulación.
- Los puntos de destino asignados sobre esta circunferencia para un mismo enjambre, son separados entre sí un cierto valor angular, que es mayor cuanto más lejos se encuentre el atacante del destino. De esta forma, cuando el atacante está más lejos se cubre una mayor área, pero con menor precisión, y a medida que el atacante se acerca a la zona del radio defensivo, esta situación se invierte.
- El cálculo del valor del ángulo asignado a cada uno de los defensores en función de su posición en el enjambre es el siguiente:

```
if position == 0:
    delta_yaw = 0
elif position % 2 == 0:
    delta_yaw = (position-1)*(5*(1.0-np.exp(-5*attacker_position_distance)))*np.pi/180.0
else:
    delta_yaw = -position*(5*(1.0-np.exp(-5*attacker_position_distance)))*np.pi/180.0
```

La representación gráfica de dicha estrategia se puede observar en la Figura 3-16, en donde a cada uno de los 3 defensores se le asigna un punto de destino sobre la circunferencia asociada al radio defensivo. La posición del primer punto asignado al primer defensor del enjambre se corresponde con el cruce entre la circunferencia defensiva y el vector que une el atacante con su punto de destino.

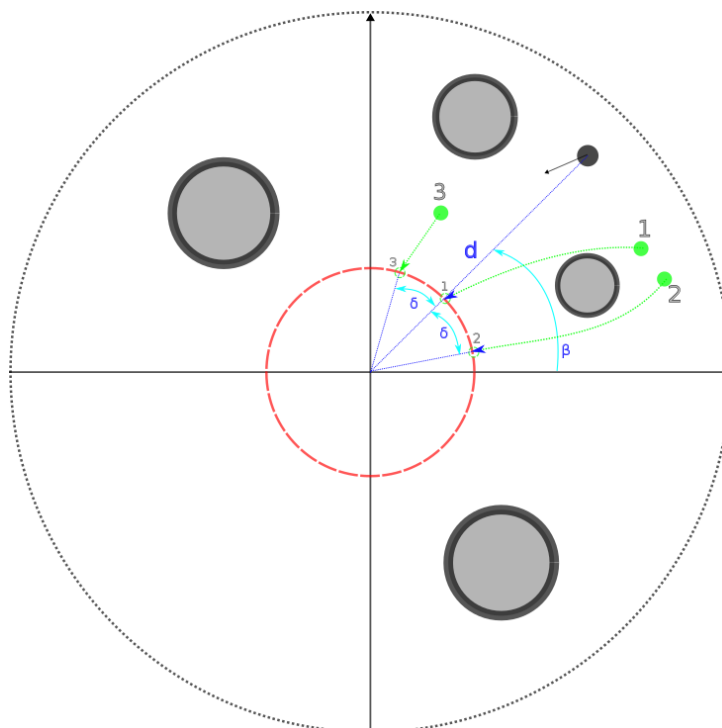


Figura 3-16.- Representación gráfica del escenario de la Estrategia 1

Para concluir con esta estrategia cabe resaltar que implementa un modelo defensivo optimizado con respecto a un área defensiva de tamaño fijo situada sobre la circunferencia en donde se posicionan, en el que los drones defensores no persiguen al atacante directamente, sino que se posicionan estratégicamente dentro de un radio defensivo para mejorar la interceptación. Este enfoque permite disponer de una mayor cobertura del espacio y una mejor capacidad de respuesta ante estos ataques enemigos, evitando colisiones y permitiendo una defensa más estructurada, aunque también se debe tener en cuenta el tiempo necesario, por parte de los defensores, para poder crear la barrera a tiempo, antes de que el atacante llegue a su entorno. De esta forma, un atacante que se moviese de forma muy oscilante podría burlar esta barrera si los defensores no están lo suficientemente cerca. Una posible forma de modificar este comportamiento podría ser emplear un radio defensivo de valor adaptativo, que cambie su valor en función de la distancia del atacante a su punto objetivo.

En el siguiente enlace y código QR puede accederse a un video que muestra la operativa de esta estrategia:

https://drive.google.com/file/d/1pGaw73dpBhBJuExhEu25S1oYn-uuS2pH/view?usp=drive_link



3.6.3 Estrategia 2

En este apartado se describe la implementación de la Estrategia 2, el cual propone un modelo de defensa basado en un ajuste dinámico del posicionamiento de los drones defensores. Su principal

objetivo es optimizar la respuesta defensiva al modificar la distancia a la que se colocan los drones con respecto al atacante, en función de su proximidad.

A diferencia de la Estrategia 1, donde los defensores se mantenían en una posición fija dentro de un radio determinado, en esta nueva estrategia el radio defensivo varía dinámicamente, lo que podría permitir que la defensa sea más flexible y eficiente en la intercepción del atacante.

Para definir la configuración del escenario, se han utilizado los siguientes parámetros:

- “*defensive_radius*”: es el radio de seguridad en el que deben posicionarse los defensores alrededor del punto de destino.
- “*collision_distance*”: la distancia de colisión que evita el choque entre los drones.

Para determinar la organización de la defensa, el sistema realiza los siguientes cálculos:

- La distancia entre el atacante y el centro del escenario
- La distancia entre cada defensor y el atacante, lo que permitirá definir si es necesario reaccionar de manera inmediata o mantener una barrera estratégica.

A partir de esta información, el algoritmo sigue la siguiente lógica:

- Si la distancia entre un defensor y el atacante es menor a 30 veces la distancia de colisión, el defensor abandonará la formación y se dirigirá directamente hacia el atacante. Esta acción garantiza una respuesta rápida ante situaciones de riesgo.
- Si la distancia es mayor, el defensor tiene como objetivo un punto en el círculo defensivo asociado a su atacante de igual forma a la estrategia anterior, pero teniendo en cuenta que en este caso el radio se define dinámicamente en función de la distancia del atacante.

En esta estrategia, el valor del radio defensivo ajustado se calcula con la siguiente ecuación, en donde el valor mínimo permitido se iguala al parámetro que establece el valor del radio defensivo en la situación estática asociada a la estrategia anterior:

```
defensive_radius = max(base_defensive_radius, 0.6*attacker_position_distance)
```

Esta expresión garantiza que el radio defensivo aumente o disminuya dependiendo de la distancia del atacante al centro del escenario, permitiendo una formación más dispersa cuando el atacante está lejos y una formación más compacta a medida que la amenaza se acerca.

A diferencia de la Estrategia 1, donde los defensores ocupaban posiciones fijas dentro del radio defensivo, en esta estrategia, su ubicación se ajusta de manera dinámica en función de la posición del atacante. Cada defensor es asignado a un punto de la circunferencia defensiva, con un ajuste angular (“*delta_yaw*”) basado en su posición dentro del enjambre.

- Si la posición del defensor es par, su orientación se desplaza en sentido positivo respecto al atacante (ángulo se incrementa en sentido antihorario).
- Si la posición del defensor es impar, la orientación se ajusta en sentido negativo (ángulo disminuye en sentido horario).
- El ajuste angular decrece exponencialmente con la distancia del atacante, permitiendo una mayor separación entre defensores cuando la amenaza está lejos y una formación más cerrada cuando se acerca.

La representación gráfica de esta estrategia se puede observar en la Figura 3-17, en la cual hay tres defensores a los cuales se les asigna un atacante que está a una distancia ($d1$) y otros tres defensores que están asignados a otro atacante que se encuentra a otra distancia distinta ($d2$). Cada grupo de 3 defensores conforman su propio radio de defensa, los cuales varían con la distancia de cada atacante al punto de destino.

La Estrategia 2 intenta crear una defensa más flexible y dinámica que las estrategias anteriores al ajustar el radio defensivo en función de la distancia del atacante.

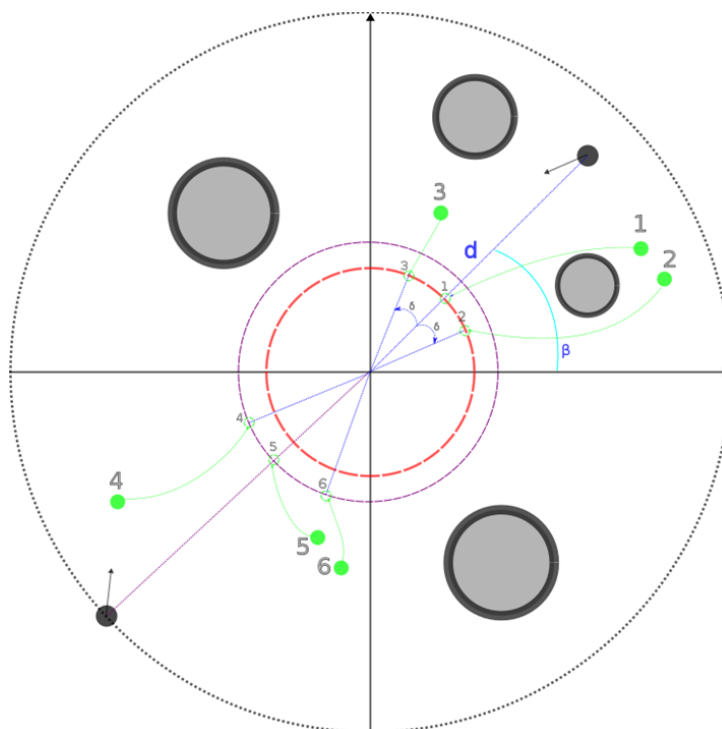


Figura 3-17.- Representación gráfica del escenario de la estrategia n°2

- Si el atacante está lejos, la barrera se expande, cubriendo un área más amplia con menor densidad de defensores.
- Si el atacante se acerca, la barrera se contrae, aumentando la concentración de defensores y mejorando la capacidad de interceptación.
- Si la distancia entre un defensor y un atacante es crítica, se abandona la formación y se opta por una interceptación directa.

Este enfoque intenta equilibrar la estabilidad de la defensa con la capacidad de adaptación, proporcionando una solución eficiente para escenarios en los que las amenazas se desplazan dinámicamente.

En el siguiente enlace y código QR puede accederse a un video que muestra la operativa de esta estrategia:

https://drive.google.com/file/d/1GY-W9G-CaU71XYsT-euRVNTsgAHdXEso/view?usp=drive_link



3.6.4 Estrategia 3

En este apartado se describe la implementación de la estrategia 3, en la cual se propone una táctica defensiva basada en la disposición relativa de los defensores respecto del atacante, formando una

estructura de arco circular que se mantiene constantemente interceptando la ruta del atacante en todo momento. Este enfoque pretende mejorar la cobertura defensiva sin depender de la distancia del atacante al centro del escenario, asegurando una presencia uniforme de los defensores en su entorno inmediato.

A diferencia de la Estrategia 2, donde los defensores modificaban el radio defensivo de manera dinámica, en esta estrategia, el objetivo principal es crear un perímetro estable y constante de vigilancia y bloqueo alrededor del atacante, evitando que este pueda moverse libremente sin encontrar resistencia. Para ello, los defensores se distribuyen equidistantemente en torno a un radio variable de intercepción (*“attacker_radius”*), garantizando una cobertura simétrica y evitando la superposición de trayectorias.

La Estrategia 3 busca implementar un esquema defensivo en el que los drones defensores se posicionan en una circunferencia alrededor del atacante, manteniendo una distancia fija con respecto a este.

Para definir la estructura de la defensa, se utilizan los siguientes parámetros clave:

- *“attacker_radius”*: es la distancia fija a la que se posicionan los defensores alrededor del atacante.
- *“collision_distance”*: define la distancia mínima en la que se considera que ha ocurrido una interceptación con el atacante.

Este enfoque permite que los defensores formen un anillo protector en torno al atacante, asegurando que esté constantemente rodeado por unidades defensivas.

El sistema está diseñado con la idea de que analice en tiempo real los siguientes datos:

- La distancia del atacante con respecto al centro del escenario, lo que permite ajustar el tamaño del círculo defensivo según la posición del atacante.
- La distancia entre cada defensor y el atacante, lo que determina si se debe actuar de forma inmediata o mantener la formación circular.
- El ángulo de posicionamiento de cada defensor, permitiendo que se distribuyan equitativamente alrededor del atacante.

A partir de estos cálculos, se podrían establecer dos posibles líneas de acción:

1. Si un defensor se encuentra demasiado cerca del atacante, se considera que el enfrenamiento es inminente, por lo que el defensor se dirige directamente a interceptarlo.
2. Si el defensor aún está a una distancia segura, se mantiene en la formación circular alrededor del atacante, con un ajuste angular específico para garantizar una distribución equilibrada.

Para lograr esta distribución, los defensores reciben un ajuste angular (*“delta_yaw”*), el cual depende de su posición en la formación:

- Si el defensor tiene un número de posición par (2, 4, 6, ...), su ángulo se incrementa ligeramente en sentido antihorario.
- Si el defensor tiene un número de posición impar (1, 3, 5, ...), su ángulo se reduce ligeramente en sentido horario.

Este ajuste angular asegura que los defensores se repartan uniformemente alrededor del atacante, evitando acumulaciones en una misma zona y garantizando una cobertura equitativa.

Una vez definidos los parámetros de la estrategia, se calcula la posición exacta de cada defensor en relación con el atacante:

- El defensor se ubica en un punto de la circunferencia defensiva, manteniendo una distancia constante con respecto al atacante.

- El ángulo de su posición se ajusta en función de su número dentro del enjambre, permitiendo una distribución uniforme.
- Cada defensor se reubica constantemente para asegurarse de que la formación se mantiene en torno al atacante sin desorganizarse.

Este posicionamiento permite que la estrategia mantenga al atacante permanentemente rodeado, reduciendo sus opciones de escape y facilitando su eventual intercepción, tal y como se puede apreciar en la Figura 3-18.

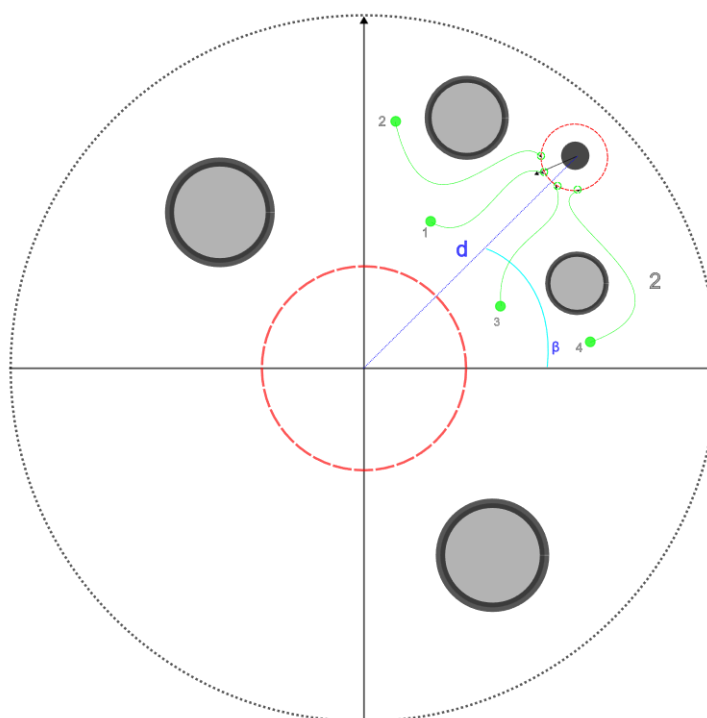


Figura 3-18.- Representación gráfica del escenario de la estrategia n°3

En conclusión, la Estrategia 3 ofrece una defensa estructurada y estable, en la que los defensores no persiguen directamente al atacante, sino que lo rodean de manera continua, formando una barrera circular a su alrededor.

- Si el atacante se encuentra dentro de un rango crítico, los defensores avanzan hacia él para interceptarlo.
- Si el atacante aún tiene margen de movimiento, los defensores se mantienen en la circunferencia defensiva, adaptando su posición para optimizar la cobertura.
- La formación circular evita que el atacante pueda moverse libremente, obligándolo a enfrentarse a los defensores desde todos los ángulos.

Este enfoque es particularmente útil en escenarios donde el objetivo es contener la amenaza dentro de un área específica, asegurando que el atacante no pueda evadir la defensa con facilidad.

En el siguiente enlace y código QR puede accederse a un video que muestra la operativa de esta estrategia:

https://drive.google.com/file/d/1dDYdRNWh-mqK60DwqY4Wa0J534n14vSB/view?usp=drive_link



3.6.5 Estrategia 4

En este apartado se describe la implementación de la estrategia 4, un modelo mixto que combina una barrera estática con un grupo de defensores que avanzan de forma progresiva hacia el atacante. Este enfoque busca un equilibrio entre estabilidad y agresividad, permitiendo que una parte de los defensores mantenga una estructura fija en el radio defensivo, mientras que otros defensores se acercan gradualmente al atacante para interceptarlo antes de que alcance su objetivo.

A diferencia de estrategias anteriores, esta estrategia es el resultado de unir las bases de las estrategias en donde se introduce una distribución alterna de roles entre los drones, lo que intenta permitir disponer de una defensa más adaptable y eficaz.

La representación gráfica de esta estrategia se puede observar en la Figura 3-19, en la cual hay entre 6 defensores para un atacante. En la imagen se puede ver cómo los drones defensivos adoptan una formación en forma de barrera con radio defensivo fijo mientras otros se adelantan para interceptar al atacante de manera más agresiva.

- Si el atacante está lejos, la estructura defensiva se mantiene amplia y estable.
- Si el atacante se acerca, los defensores móviles reducen progresivamente la distancia, aumentando la presión sobre la amenaza.
- Si la distancia con el atacante es crítica, todos los defensores pueden abandonar la formación y dirigirse directamente a la interceptación.

Este modelo busca generar una mayor flexibilidad y eficacia, asegurando que la defensa no solo contenga la amenaza, sino que también busque neutralizarla antes de que pueda comprometer su objetivo.

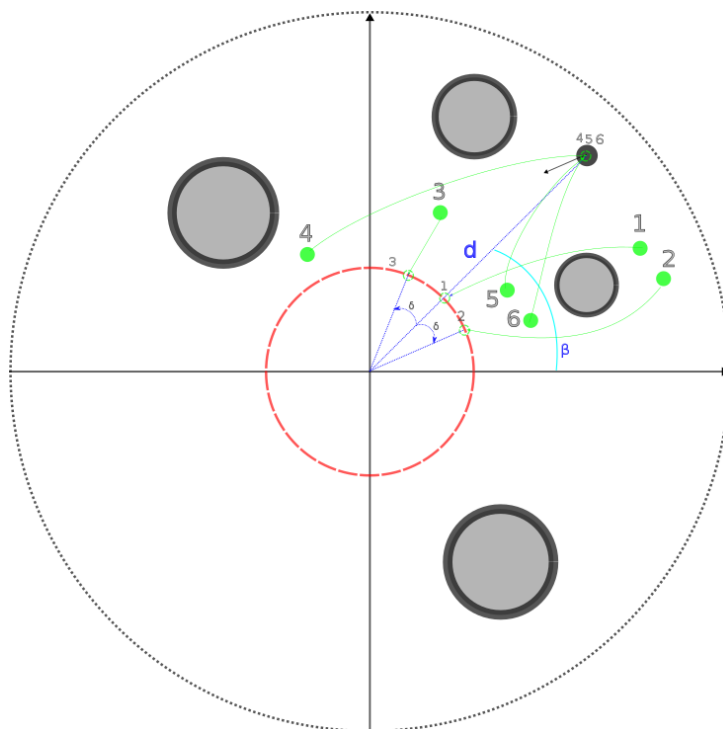


Figura 3-19.- Representación gráfica del escenario de la estrategia n°4

En el siguiente enlace y código QR puede accederse a un video que muestra la operativa de esta estrategia:

https://drive.google.com/file/d/1N8mj9n4NMH72pBKCZLNt1xRyXsDKHUUf/view?usp=drive_link



3.6.6 Estrategia 5

En este apartado se describe la implementación de la estrategia 5, la cual, de igual forma que en la estrategia anterior, se crea mediante la combinación de dos de las estrategias ya propuestas: la barrera defensiva de radio fijo y la barrera defensiva de radio variable.

La representación gráfica de esta estrategia se puede observar en la Figura 3-20, en la cual hay entre 6 defensores para un atacante. En la imagen se puede ver cómo los drones defensivos adoptan dos formaciones en forma de barrera, una en la circunferencia del radio defensivo y otra en una posición más cercana al atacante cuyo radio cambia de forma dinámica en función de la distancia del atacante al origen. Este modelo busca aprovechar las propiedades de cada una de las estrategias en las que se basa,

pero tiene el problema de depender de un número mínimo de drones defensivos para ser eficaz, mayor que en otros casos.

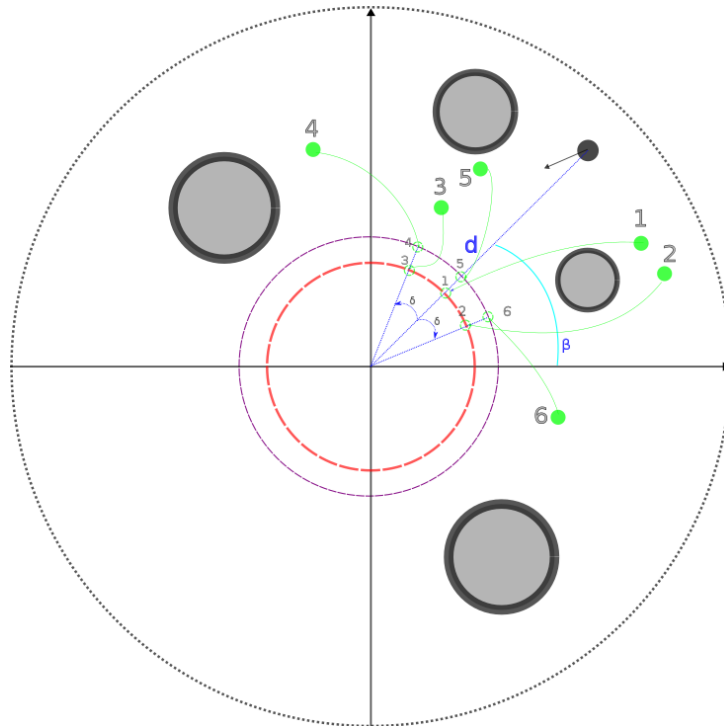


Figura 3-20.- Representación gráfica del escenario de la estrategia n°5

En el siguiente enlace y código QR puede accederse a un video que muestra la operativa de esta estrategia:

https://drive.google.com/file/d/1aJz7Wj1LFKn386jZBcNnd7ZCbAmWkF8b/view?usp=drive_link



4 RESULTADOS

4.1 Escenarios de simulación

En este apartado se presentan los resultados obtenidos a partir de la simulación de diversos escenarios diseñados para evaluar la eficacia de distintas estrategias de defensa. Con este propósito, se han generado múltiples entornos con condicionantes iniciales aleatorias, lo que ha permitido que analizar el comportamiento, tanto de los atacantes como de los defensores, bajo diferentes configuraciones operativas.

De esta forma, una de las cuestiones que se plantean es tratar de identificar la cantidad mínima de drones defensores necesaria para garantizar un cierto porcentaje de éxito en la tarea de la interceptación de los atacantes en todo tipo de situaciones y bajo qué condiciones se consigue este resultado. Este análisis es fundamental para establecer criterios cuantificables que optimicen la distribución de recursos defensivos en entornos marinos dinámicos.

Para llevar a cabo esta evaluación, en este trabajo se han desarrollado distintos escenarios de simulación en los que los atacantes siguen trayectorias predefinidas mientras un conjunto de defensores implementa diversas estrategias de interceptación con el objetivo de neutralizar su avance. La variabilidad de las condiciones iniciales ha permitido examinar la influencia de factores como la posición de inicio de los drones, la velocidad máxima permitida a cada dron o la presencia de obstáculos en la efectividad de la interceptación.

El objetivo principal es generar estrategias que garanticen que los defensores logren interceptar al atacante en las simulaciones, evaluando as su vez la robustez de los algoritmos de planificación de rutas utilizados. Para ello, se han implementado y comparado diferentes enfoques de navegación, incluyendo métodos basados en inteligencia artificial, planificación de trayectorias y optimización de estrategias cooperativas.

Los resultados obtenidos permitirán validar la eficacia de los algoritmos propuestos y establecer criterios de optimización que contribuyan a mejorar el rendimiento de los defensores en futuros escenarios. En las siguientes secciones se presentará un análisis detallado de los datos obtenidos, incluyendo el porcentaje de éxito en la interceptación, la influencia de los parámetros iniciales en el desempeño de los defensores y una comparación entre las estrategias implementadas para determinar su efectividad en distintos contextos operativos.

Con el objetivo de evaluar los resultados se propone la creación de una función de coste que evalúe cada posible escenario. Si el valor de la función de coste es positivo, significa que los defensores no están cumpliendo su misión de interceptación y los atacantes van ganando. Si el coste disminuye e

incluso se hace negativo implica una buena estrategia defensiva. Esta función de coste empleada para la evaluación se define de la siguiente forma:

1. En cada instante se calcula el coste de cada atacante en base a la distancia a la que se encuentra de su punto de destino (*attacker_distance_to_target*) de la siguiente forma:

```
step_attackers_cost += (1.0/num_attackers)*np.exp(-k*attacker_distance_to_target)
```

2. En cada instante se calcula el coste de cada defensor en base a la distancia a la que se encuentra de su atacante asignado (*defender_distance_to_target*) de la siguiente forma:

```
step_defenders_cost -= (1/num_defenders)*np.exp(-5*k*defender_distance_to_target)
```

3. Si un atacante alcanza su punto de destino, el episodio finaliza con victoria para los atacantes y la función de coste se incrementa con 10 puntos.
4. Cada vez que un atacante es interceptado por un defensor, la función de coste se reduce un valor:

```
step_defenders_cost -= 5.0/num_defenders
```

Otro factor a tener en cuenta en el análisis será el tiempo en que finaliza cada episodio, de forma que escenarios con tiempo reducido y en donde ganen los atacantes serán los que indiquen un peor rendimiento de la estrategia defensiva. Por otra parte, escenarios con tiempo reducido en donde ganen los defensores serán los que consigan una mejor puntuación final y presenten las mejores condiciones para establecer una estrategia defensiva.

4.2 Resultados de las simulaciones sin obstáculos

En este apartado se presentan los resultados obtenidos a partir de las simulaciones realizadas en distintos escenarios en donde no hay presentes obstáculos. Las condiciones generales para configurar los escenarios son las siguientes:

- El número de *drones* atacantes en cada escenario puede ser de entre 4 y 6.
- El número de *drones* defensores asignados a cada atacante puede ser entre 4 y 11.
- La distancia de colisión es de 5.0 m.
- El radio del escenario es de 2.000 m.
- El radio defensivo de radio fijo es de 300.0 m.
- El radio defensivo alrededor de un atacante es de 300.0 m.
- La velocidad de los atacantes puede ser entre 30.0 y 60.0 m/s.
- La velocidad de los defensores puede ser entre 10.0 y 30.0 m/s.
- La simulación se ejecuta con un tiempo de paso de 0,5 s.

Estas configuraciones han sido seleccionadas para simular escenarios de diferente complejidad, en los que se evalúa la capacidad de los defensores para interceptar a los atacantes bajo diversas circunstancias. Para obtener datos estadísticamente significativos, se han llevado a cabo 1000 simulaciones en las que se han registrado múltiples variables clave, tales como el tiempo de interceptación, la tasa de éxito en la defensa, el número de defensores activos al finalizar la simulación y la distancia media de interceptación. El tiempo empleado para ejecutar estas simulaciones ha sido de 3 horas y 20 minutos en un PC con Windows 10 y un sistema con dos procesadores iCore5.

Los resultados obtenidos proporcionan una visión sobre el desempeño de los defensores y su capacidad de respuesta ante los atacantes en distintos escenarios. A lo largo de las siguientes secciones,

se presentará un análisis cuantitativo de los datos recopilados, destacando patrones, tendencias y factores determinantes en la efectividad de la estrategia defensiva.

Este análisis permitirá identificar qué configuraciones resultan más eficaces para garantizar la intercepción del atacante, estableciendo criterios para mejorar la planificación y optimización de las estrategias defensivas en futuras aplicaciones.

En primer lugar; tal y como se puede observar en la Figura 4-1, se realiza una comparación entre las estrategias disponibles en función de si la victoria final ha sido para los atacantes o para los defensores.

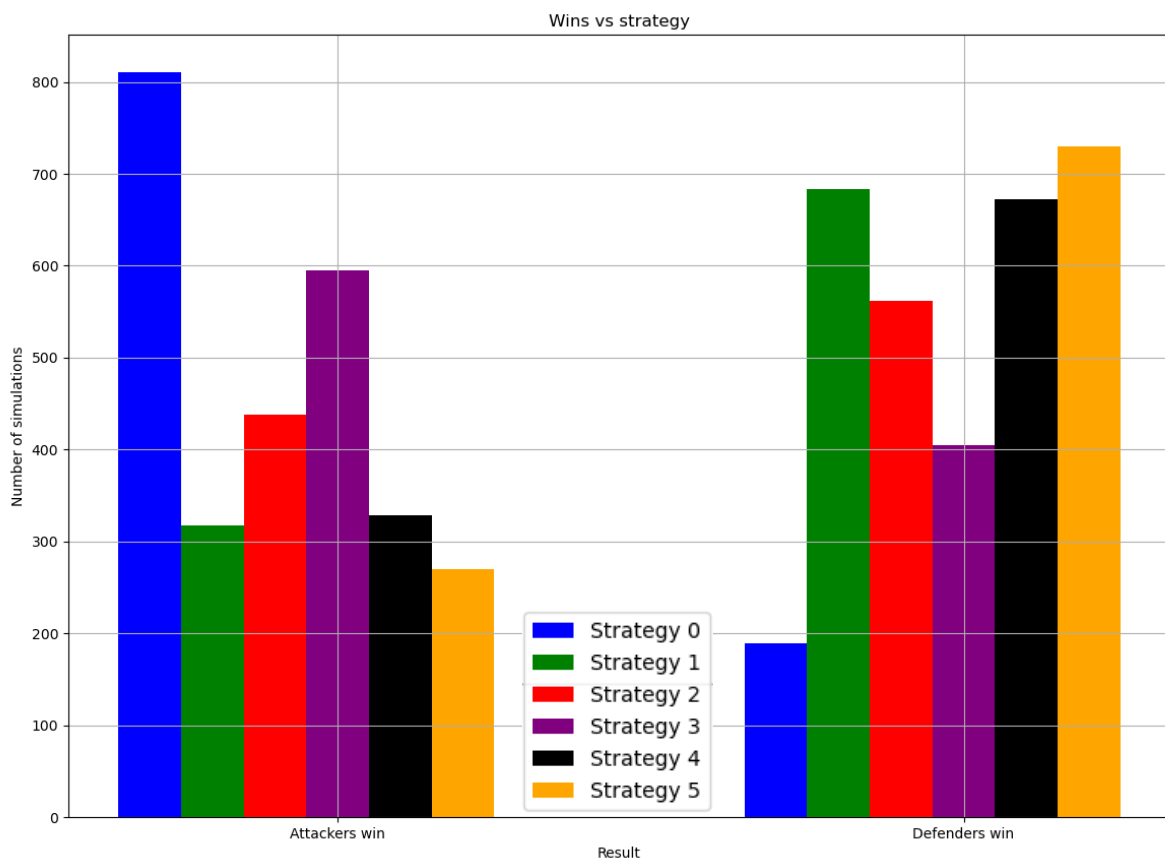


Figura 4-1.- Representación gráfica de la relación entre el número de simulaciones y las estrategias ganadas por los atacantes y defensores.

A partir de estos resultados se observa que:

- Estrategia 5 (color naranja): Se posiciona como la estrategia más óptima, indicando que los defensores logran un mayor número de intercepciones con esta configuración. Se recuerda que esta estrategia combinaba las estrategias 1 y 2, generando una doble barrera defensiva con un radio fijo para todos los atacantes y otro adaptativo para cada atacante.
- Estrategia 1 (color verde): Se presenta como una de las estrategias más eficaces, y aunque, como se ha visto, su combinación con la estrategia 2 produce mejores resultados, el proceso de creación de una barrera defensiva de radio fijo parece ser una buena estrategia de base con la que poder obtener unos mínimos resultados satisfactorios.
- Estrategia 4 (color negro): También muestra un buen desempeño, obteniendo un rendimiento muy cercano al de la estrategia anterior. En este caso, se dispone de una barrera defensiva de radio fijo mientras que otros defensores se dirigen directamente al atacante.

Por otro lado, las estrategias que han demostrado ser menos efectivas para los defensores son:

- Estrategia 0 (color azul): En este caso los defensores persiguen directamente al atacante, pero debido a sus menores prestaciones los atacantes logran escapar con demasiada frecuencia.
- Estrategia 3 (color lila): Similar a la anterior, el hecho de desplazarse hacia puntos que rodean al atacante implica que para funcionar bien deben estar inicialmente a una distancia que les permita alcanzar esas posiciones, aunque debido a su velocidad inferior, la probabilidad de que esto ocurra no es muy favorable.

Por último, cabe destacar que la Estrategia 2 (color rojo), consigue peores resultados empleando un radio defensivo dinámico que los obtenidos con un radio defensivo fijo Estrategia 1 (color verde), pero que al ser combinada con esta se alcanzan los mejores resultados en la Estrategia 5 (color naranja) lo que parece dar a entender que funciona complementando los errores que pueden producirse de la Estrategia 1.

En conclusión, este análisis permite obtener una imagen general de la relación entre las estrategias utilizadas por los defensores, identificando cuáles son las configuraciones más eficientes en términos de victorias obtenidas. Esto proporciona información clave para futuras optimizaciones en la planificación de estrategias defensivas y ofensivas en entornos de simulación similares.

A continuación, se lleva a cabo un análisis comparativo para determinar qué estrategia resulta más óptima considerando como variable de comparación el tamaño del enjambre asignado a cada atacante presente en la simulación. Esta evaluación permite identificar cómo la cantidad de drones involucrados en la simulación afecta el rendimiento de cada estrategia y en qué configuración resulta más efectiva cada estrategia en distintos escenarios.

Tal y como se observa en la Figura 4-2, se obtienen unos resultados en donde pueden destacarse los siguientes comentarios:

- Como era de esperar, a medida que el número de drones defensores aumenta, se observa una clara reducción en las victorias de los atacantes y un incremento en las victorias de los defensores. Esto indica que, en enjambres pequeños, los atacantes tienen una mayor facilidad para evadir la interceptación, mientras que, en enjambres más grandes, los defensores logran organizarse mejor y aumentar su efectividad.
- La Estrategia 2 (color rojo), es la que presenta una menor variación de resultados en función del tamaño del enjambre de captura, y la Estrategia 1 (color verde) la que, ligeramente, presenta una mayor variación de resultados.
- Por último, hay que destacar que la Estrategia 0 (color azul) es la única que para ningún tamaño del enjambre tiene unos resultados positivos para los defensores.

En definitiva, estos resultados ponen en evidencia que tanto la cantidad de defensores como la estrategia empleada juegan un papel determinante en el éxito de la defensa. Si bien en enjambres pequeños los atacantes tienen mayor ventaja, conforme se incrementa el número de defensores y se aplican estrategias más avanzadas, la capacidad de interceptación mejora significativamente si se emplean procesos de generación de barreras que esperan a los atacantes combinados entre sí. Este análisis ofrece información valiosa para la optimización de estrategias defensivas en futuros escenarios.

Por último, se realiza el análisis de costes y tiempos de simulación. Este análisis es fundamental para evaluar la eficacia de las diferentes estrategias implementadas en el sistema defensivo. En este apartado, se presentan los resultados obtenidos al comparar el coste total y el tiempo de simulación en función de la estrategia utilizada, diferenciando entre escenarios en los que los atacantes logran alcanzar su objetivo y aquellos en los que los defensores consiguen interceptarlos con éxito.

El objetivo principal de este estudio es determinar cómo cada estrategia influye en la eficiencia operativa del sistema de defensa, considerando no solo la tasa de éxito, sino también el impacto en los recursos utilizados y el tiempo requerido para completar la misión.

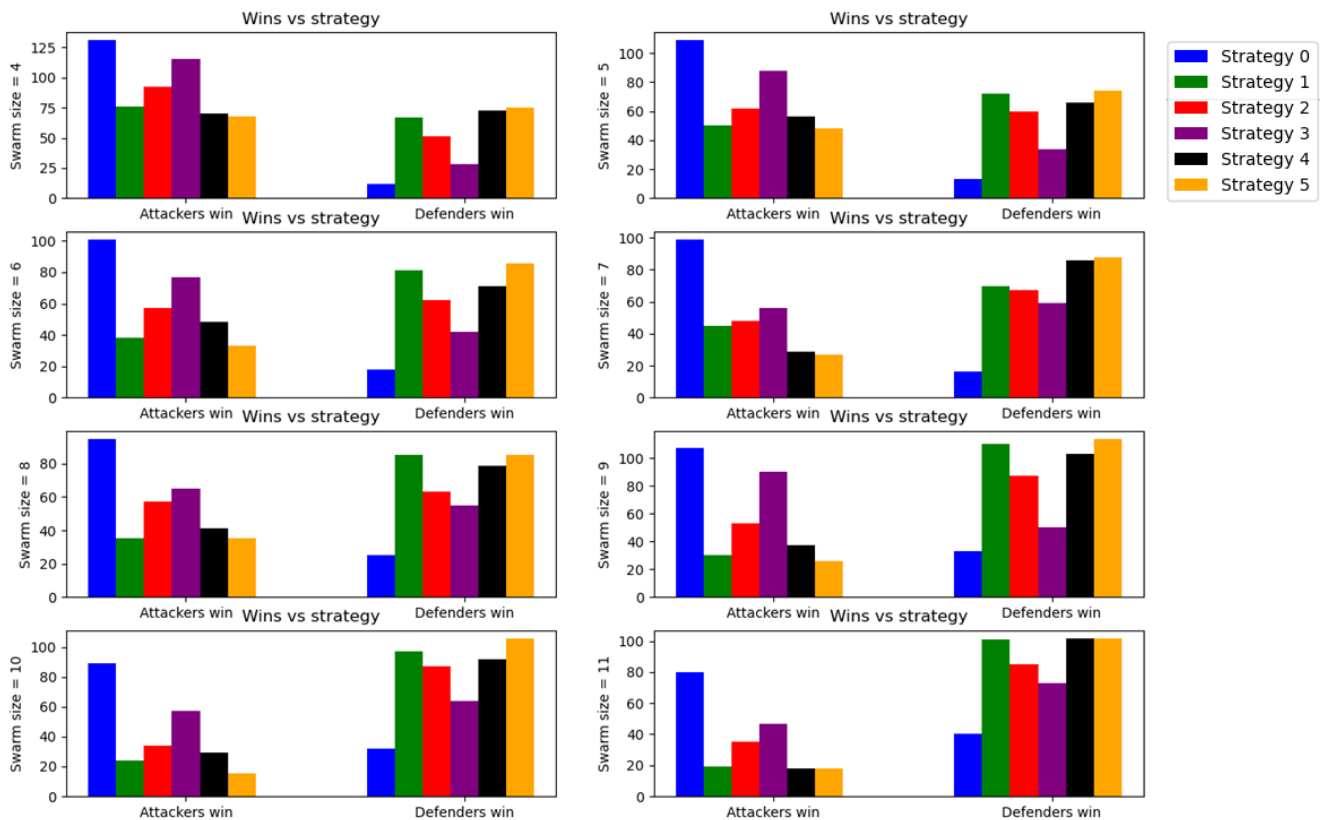


Figura 4-2.- Representación gráfica de la relación entre la variación del tamaño del enjambre y las estrategias utilizadas

Los gráficos presentados en la Figura 4-3 reflejan la distribución de los valores de coste total y tiempo de simulación para cada estrategia, proporcionando una visión clara sobre su desempeño. A través de este análisis, se pretende identificar qué estrategias ofrecen un equilibrio óptimo entre efectividad, eficiencia en el uso de recursos y tiempo de respuesta, con el fin de optimizar la planificación defensiva en futuras simulaciones.

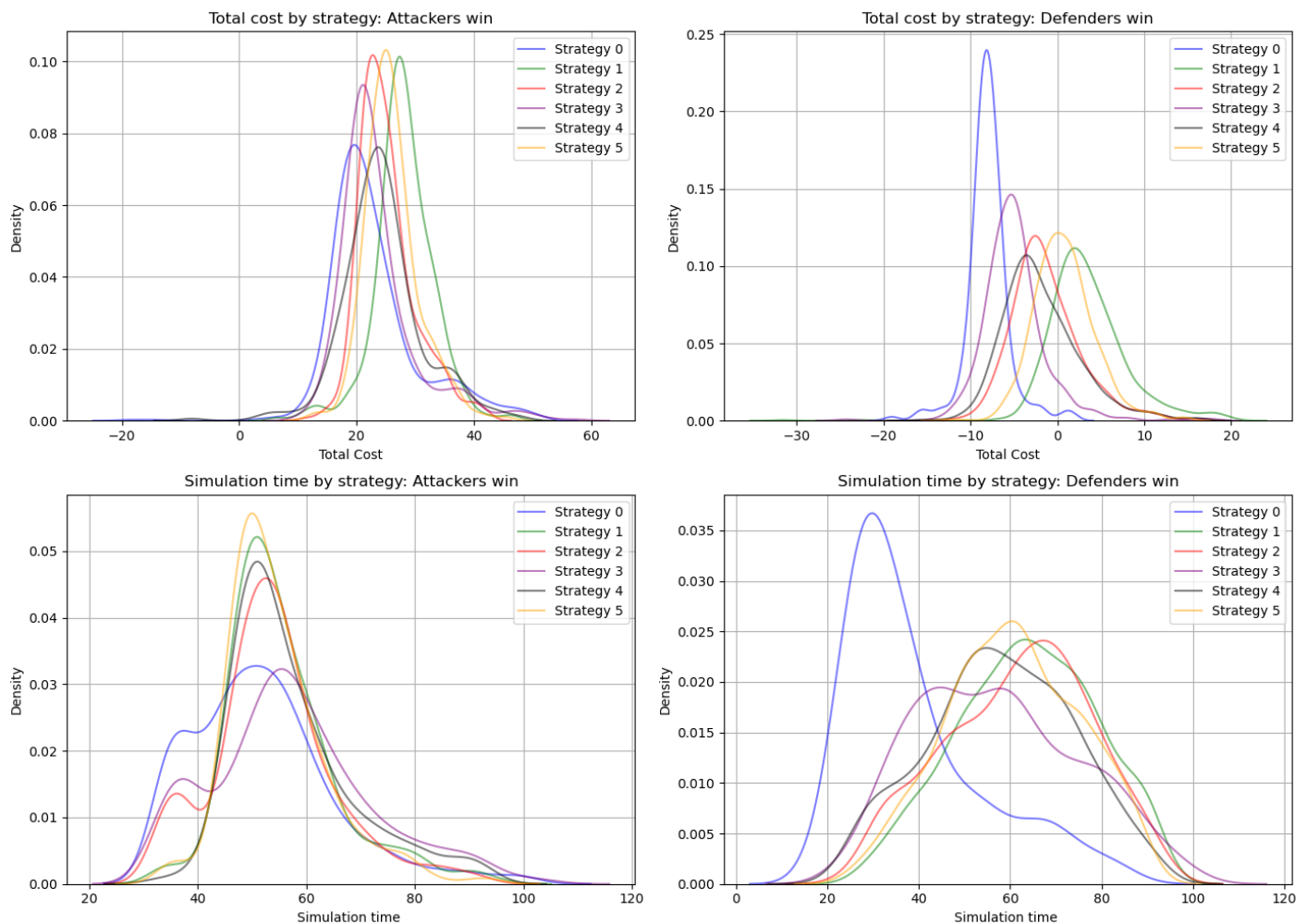


Figura 4-3.- Representación gráfica entre costes totales y tiempos de simulación por estrategia

Se pueden realizar las siguientes observaciones en base a estos resultados:

1. Cuando ganan los atacantes (gráficas superior e inferior izquierda), todas las estrategias generan una distribución similar entre sí, aunque aquellas que contienen el seguimiento directo del atacante o su entorno tienden a tener una distribución ligeramente más dispersa.
2. Cuando ganan los defensores (gráficas superior e inferior derecha) se puede destacar, en primer lugar, que la Estrategia 0 (color azul) genera unas simulaciones con menor tiempo y coste. Esto puede deberse a que el proceso aleatorio que genera las posiciones iniciales de los defensores permita, en cierto tipo de escenarios, que se produzca mejores condiciones para que realicen la interceptación por encontrarse muy cerca de los atacantes. En segundo lugar, puede descartarse que las que en los resultados totales de victorias obtuvieron mejores resultados (verde y naranja), tienden a generar mayores costes y tiempo de simulación, lo cual plantea el equilibrio existente entre la necesidad de obtener una victoria y la necesidad de reducir el tiempo en que los atacantes permanecen en el escenario. En ocasiones, intentar reducir demasiado rápido el tiempo de respuesta puede llevar a cometer más errores de interceptación debido a las condiciones inferiores de los defensores, que no pueden seguir a los atacantes. Visto de otra forma, la generación de barreras que permiten mejorar el número de interceptaciones lleva un tiempo que lleva a interceptar a los atacantes en posiciones más próximas al origen.

4.3 Resultados de las simulaciones con obstáculos

En este apartado se presentan los resultados obtenidos a partir de las simulaciones realizadas en distintos escenarios en donde hay presentes obstáculos. Las condiciones generales para configurar los escenarios son las mismas que en el apartado anterior teniendo en cuenta que:

- El número de obstáculos puede ser de entre 3 y 19.
- El número radio de cada obstáculo puede ser entre 50,0 y 200,0 m.

En la Figura 4-4 se presentan los resultados obtenidos, en donde puede observarse que se obtiene un resultado muy similar al apartado anterior en donde no había obstáculos. La Estrategia 3 (color lila) es la que muestra una mayor variación en sus resultados, presentado una cierta mejora en comparación con el caso sin obstáculos, algo que también se observa en la Estrategia 1 (color azul). Esto puede deberse a que la presencia de obstáculos hace que las trayectorias de los atacantes sean más largas al tener que esquivar los obstáculos, lo cual permite a los defensores lograr alcanzar su objetivo de interceptación en mejores condiciones, a que estos en todo momento optimizan sus trayectorias en base al punto de destino deseado y la continua actualización de la ruta basada en el algoritmo A*.

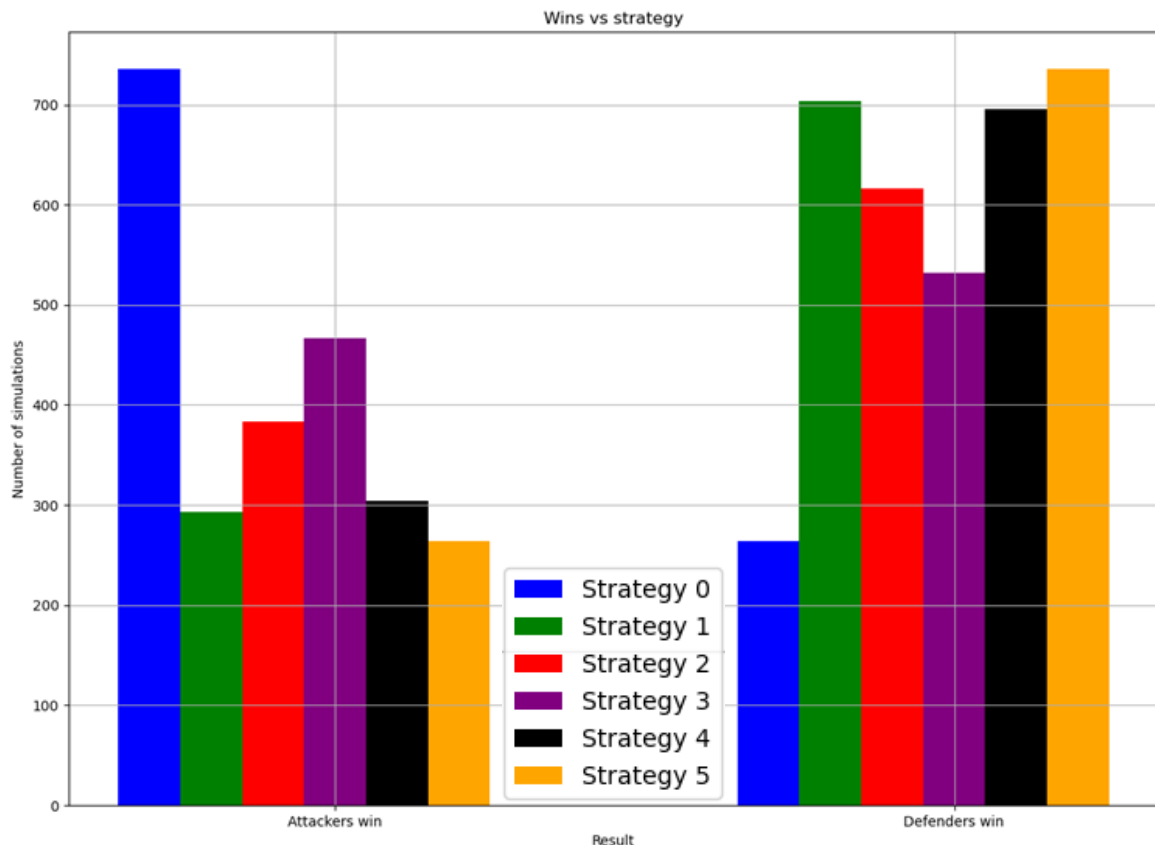


Figura 4-4.- Representación gráfica de la relación entre el número de simulaciones y las estrategias ganadas por los atacantes y defensores.

En la Figura 4-5 se presentan los resultados de victorias en función del tamaño del enjambre, en donde nuevamente se obtienen unas gráficas similares a las obtenidas en el apartado anterior, en las que en líneas generales destaca una mejoría en las estrategias que tiene un patrón de seguimiento de la posición del atacante, aunque sigan siendo peores que aquellos que combinan estrategias de barrera con otras opciones.

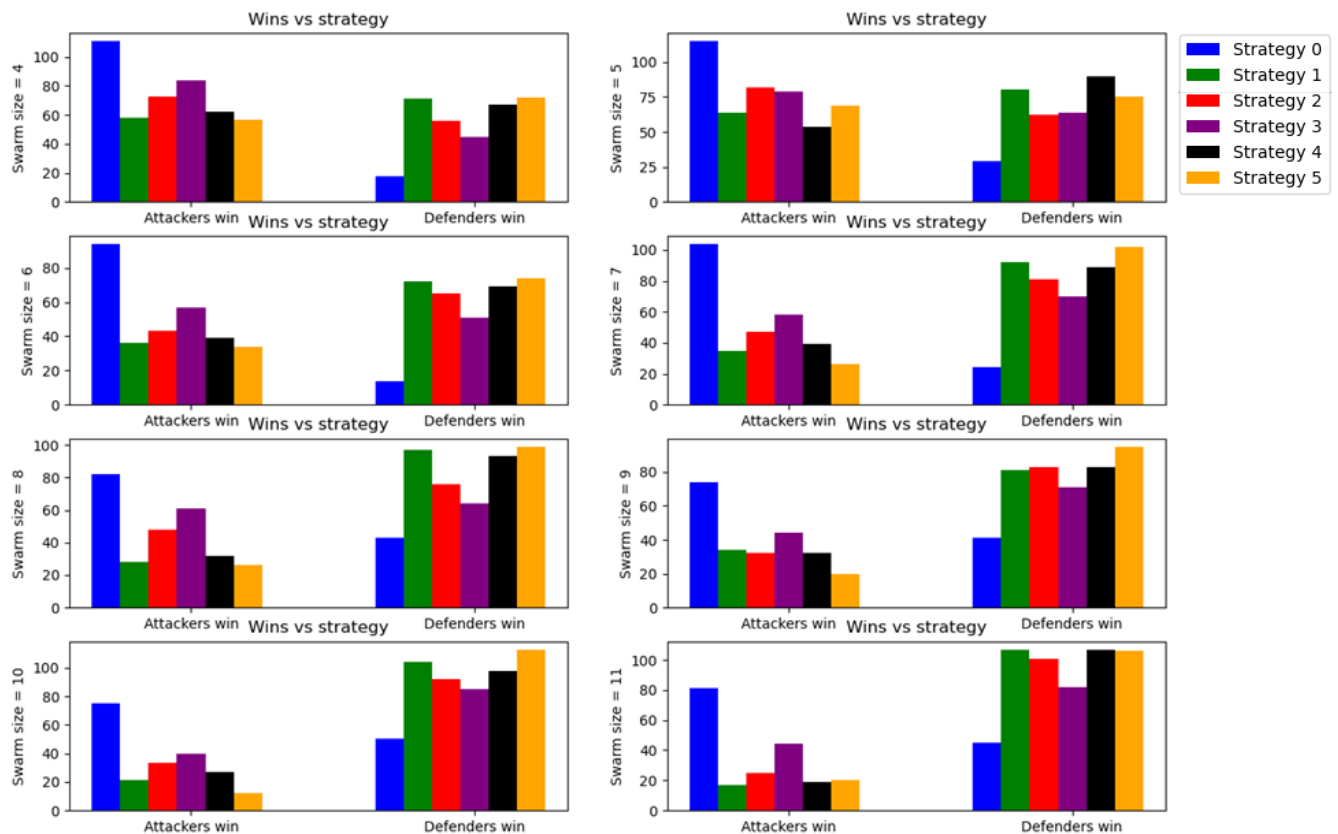


Figura 4-5.- Representación gráfica de la relación entre la variación del tamaño del enjambre y las estrategias utilizadas

Por último, en la Figura 4-6 se muestra la distribución del coste y el tiempo de simulación en donde se puede volver a observar que se repiten las conclusiones obtenidas hasta ahora, en donde los resultados no parecen cambiar mucho con respecto a las simulaciones sin obstáculos.

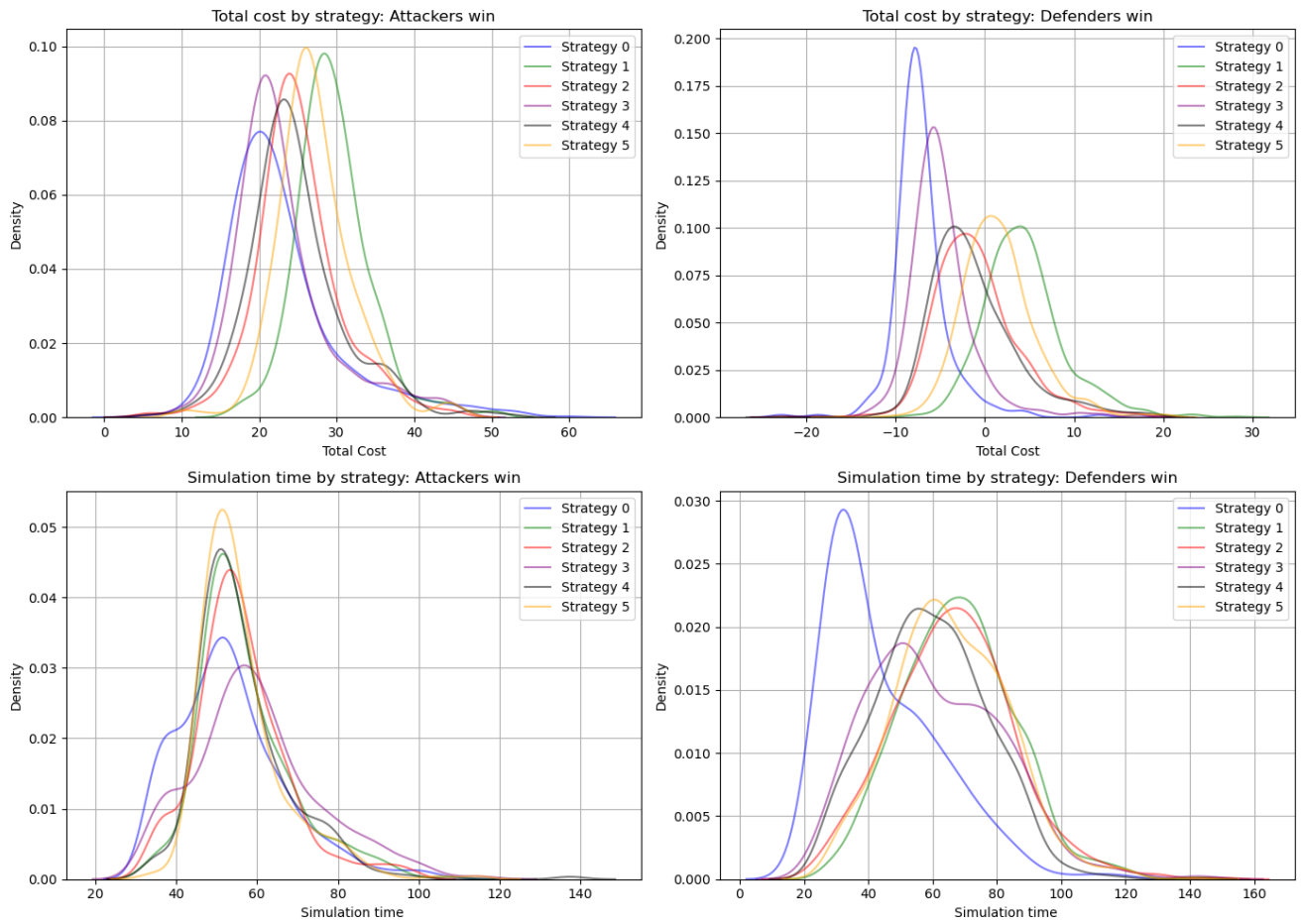


Figura 4-6.- Representación gráfica entre costes totales y tiempos de simulación por estrategia

5 CONCLUSIONES Y LÍNEAS FUTURAS

A lo largo de este trabajo, se ha desarrollado un sistema de simulación que permite analizar la efectividad de distintas estrategias defensivas en un entorno dinámico donde un enjambre de atacantes intenta alcanzar un objetivo, mientras que un conjunto de drones defensores trata de interceptarlos antes de que lo consigan.

Mediante la implementación y evaluación de múltiples escenarios con condiciones iniciales aleatorias, se han obtenido resultados significativos que permiten extraer diversas conclusiones sobre el rendimiento y la eficiencia de cada estrategia.

En primer lugar, se ha evidenciado que el número de defensores es un factor determinante en la capacidad de interceptación. Estrategias que optimizan la distribución espacial de los defensores tienden a requerir un menor número de unidades para lograr un porcentaje de éxito elevado, en comparación con aquellas que siguen enfoques más reactivos. En particular, las estrategias 1, 4 y 5 han demostrado ser las más eficientes en términos de interceptación, mientras que la Estrategia 0, basada en la persecución directa del atacante, ha resultado ser la menos efectiva en la mayoría de los escenarios evaluados.

Además, el análisis del coste total y del tiempo de simulación ha permitido comprender la eficiencia operativa de cada estrategia. Se ha observado que aquellas estrategias más estructuradas y con una distribución previa de los defensores tienden a generar menores costes acumulados, mientras que estrategias que dependen de respuestas inmediatas frente al atacante presentan mayor variabilidad y un incremento en el consumo de recursos.

También se ha identificado que la velocidad relativa entre atacantes y defensores juega un papel fundamental en la efectividad de las estrategias. En los escenarios en los que los atacantes cuentan con una ventaja significativa en velocidad, las estrategias que construyen barreras defensivas en torno al objetivo han sido más exitosas que aquellas centradas en el seguimiento directo del enemigo.

Por otro lado, el estudio ha permitido examinar la influencia del tamaño del enjambre en los resultados. Se ha comprobado que existe un umbral a partir del cual aumentar el número de defensores no incrementa significativamente la tasa de éxito, lo que indica que una distribución eficiente de los recursos es más importante que el simple aumento de unidades disponibles.

En líneas generales, se puede afirmar que los objetivos planteados al inicio del trabajo se han alcanzado satisfactoriamente. No obstante, es importante señalar que, con un mayor margen de tiempo, habría sido posible profundizar en enfoques más avanzados de toma de decisiones, como el desarrollo de un sistema de entrenamiento automático para la generación de estrategias defensivas. Técnicas basadas en sistemas difusos (*“fuzzy logic”*) o en algoritmos evolutivos podrían haber permitido el diseño y la optimización de estrategias adaptativas, mejorando así la capacidad de respuesta de los defensores frente a amenazas emergentes en tiempo real.

En conclusión, este trabajo ha proporcionado una visión detallada sobre la optimización de estrategias defensivas en entornos simulados, estableciendo criterios cuantificables para la toma de decisiones y sentando las bases para futuras investigaciones en planificación de trayectorias, coordinación de enjambres y aplicación de inteligencia artificial en escenarios de defensa y seguridad.

6 BIBLIOGRAFÍA

- [1] J. Díaz Camacho, «Estrategias de ataque y defensa de enjambre de drones heterogéneos», 2018.
- [2] D. M. Roldan y D. M. Rueda, «Drones en la vanguardia: Tecnologías avanzadas redefiniendo la guerra moderna».
- [3] «La Armada compra un vehículo de superficie no tripulado a la empresa española Seadrone». [En línea]. Disponible en: <https://www.infodefensa.com/texto-diario/mostrar/5057254/armada-compra-vehiculo-superficie-no-tripulado-empresa-espanola-seadrone>
- [4] «Ataque secreto de un submarino enano japonés a Pearl Harbor». [En línea]. Disponible en: <https://pearl-harbor.info/secret-japanese-midget-submarine-attack-on-pearl-harbor/>
- [5] «Fieseler Fi 103 Reichenberg». [En línea]. Disponible en: <https://aeromodelismo192.wordpress.com/2015/04/15/fieseler-fi-103-reichenberg/>
- [6] «Dron SEA HUNTER». [En línea]. Disponible en: <https://www.elradar.es/sea-hunter-buque-antisubmarino-autonomo-la-us-navy/>
- [7] «Dron SEA HAWK». [En línea]. Disponible en: <https://news.usni.org/2021/04/08/navy-takes-delivery-of-sea-hawk-unmanned-vessel>
- [8] «Dron de superficie estadounidense tipo “Overlord”». [En línea]. Disponible en: [overlord usv - Buscar con Google](#)
- [9] «Dron SeaGull». [En línea]. Disponible en: <https://www.defensa.com/industria/elbit-systems-realiza-israel-mision-antisubmarina-usv-seagull>
- [10] «Dron chino JARI USV». [En línea]. Disponible en: [JARI USV – Estadísticas MW](#)
- [11] «Dron de superficie chino ZHUHAIYUN USV». [En línea]. Disponible en: [Zhuhaiyun usv - Buscar con Google](#)
- [12] «Dron Devil Ray». [En línea]. Disponible en: <https://martacsystems.com/products/t38/>
- [13] «Dron submarino Orca». [En línea]. Disponible en: <https://galaxiamilitar.es/la-marina-de-ee-uu-prueba-el-submarino-no-tripulado-orca-xl-que-coloca-minas-y-pueden-disparar-un-torpedo/>
- [14] «Muskie M18». [En línea]. Disponible en: [MARTAC Muskie M18 y Devil Ray USV - Grupo BlueZone](#)
- [15] «Dron MAST-13». [En línea]. Disponible en: https://militaryleak.com/2019/12/18/maritime-autonomy-surface-testbed-mast-13/#google_vignette
- [16] «MAGURA V5». [En línea]. Disponible en: [MAGURA V5 - Buscar con Google](#)
- [17] «Dron de superficie ucraniano SEA BABY». [En línea]. Disponible en: [sea baby drone ukraine - Buscar con Google](#)
- [18] «Dron Emily». [En línea]. Disponible en: <https://www.roboticstomorrow.com/story/2022/10/what-does-emily-mean-for-the-future-of-robot-assisted-rescue/19633/>
- [19] «Dron de superficie estadounidense SAILDRONE EXPLORER». [En línea]. Disponible en: [saildrone explorer us navy - Buscar con Google](#)

- [20] «DRON OCEAN ALPHA». [En línea]. Disponible en: https://oceanalpha.com/news_list/oceanalpha-launches-the-dolphin-1-unmanned-surface-rescue-vehicle-usrv/
- [21] «DRON QYSEA». [En línea]. Disponible en: https://www.cascoantiguo.com/en/tek/underwater-drones/fifish-v6-qysea?srsId=AfmBOopitORAi59vYS4Js8yaoY82be_Yfx8P3TbJcsIEwShjLSorRbjk
- [22] E. Beltran y W. Bolívar, «El uso de los drones armados y su impacto en la guerra contemporanea estadounidense», *Universidad Militar Nueva Granada, Bogotá DC, Colombia. Recuperado de: https://repository.unimilitar.edu.co/bitstream/handle/10654/17595/BeltranPinedaEduardo%20y%20BolivarWilliam2018.pdf*, 2017.
- [23] «Drones y vehículos no tripulados en guerra». [En línea]. Disponible en: <https://revistamarina.cl/es/articulo/drones-y-vehiculos-no-tripulados-en-guerra?>
- [24] E. Rivera López, «Los drones, la moralidad profunda y las convenciones de la guerra», *Isonomía*, n.º 46, pp. 11-28, 2017.
- [25] M. G. Fernández, «Vehículos no tripulados para utilización naval», *ARMADA ESPAÑOLA*, vol. 2013, n.º 15, p. 69, 2013.
- [26] «A-star». [En línea]. Disponible en: [https://nlaredo.tecnm.mx/takeyas/apuntes/Inteligencia%20Artificial/Apuntes/tareas_alumnos/A-Star/A-Star\(2005-II-B\).pdf](https://nlaredo.tecnm.mx/takeyas/apuntes/Inteligencia%20Artificial/Apuntes/tareas_alumnos/A-Star/A-Star(2005-II-B).pdf)

ANEXO I: IMPLICACIONES SOCIALES, Y/O ECONÓMICAS, Y/O AMBIENTALES

El desarrollo de este Trabajo de Fin de Grado, centrado en el diseño y simulación de estrategias defensivas con drones autónomos, plantea una serie de implicaciones que trascienden el ámbito puramente técnico y que merecen ser consideradas desde una perspectiva más amplia, tanto social como económica y ambiental.

Desde un punto de vista social, la creciente presencia de sistemas autónomos como los drones en escenarios de vigilancia, defensa o control de infraestructuras críticas plantea importantes cuestiones relacionadas con la ética, la privacidad y la seguridad ciudadana. Si bien estos sistemas pueden ofrecer respuestas más rápidas y eficaces ante situaciones de emergencia o amenazas, también deben ser regulados adecuadamente para garantizar que su uso no vulnere derechos fundamentales, como el derecho a la intimidad o a la libertad de movimiento. La automatización de decisiones estratégicas, como la interceptación de objetos o entidades móviles, exige una reflexión profunda sobre el papel del ser humano en el proceso de supervisión y control de estos dispositivos.

En términos económicos, el uso de enjambres de drones como mecanismo de defensa representa una alternativa potencialmente más eficiente y rentable frente a sistemas tradicionales, como radares, interceptores tripulados o misiles defensivos. La capacidad de simular diferentes estrategias permite optimizar la cantidad de recursos necesarios, minimizando costes sin comprometer la eficacia operativa. Además, la modularidad, la posibilidad de reutilización y los avances tecnológicos en componentes de bajo coste hacen que este tipo de soluciones puedan adaptarse a diferentes escalas, desde aplicaciones militares hasta usos en seguridad civil o industrial.

En lo que respecta a las implicaciones ambientales, cabe destacar que los drones, en comparación con otros sistemas más pesados o dependientes de combustibles fósiles, generan un menor impacto ecológico, especialmente cuando están alimentados por fuentes eléctricas. Sin embargo, su proliferación también plantea retos en cuanto a la gestión de residuos electrónicos, la producción de baterías y el uso intensivo de materiales como el litio o el cobalto. Asimismo, la realización de simulaciones como las implementadas en este trabajo contribuye a reducir la necesidad de ensayos físicos, minimizando el consumo de energía y recursos asociados a pruebas en entornos reales.

En resumen, este proyecto no solo ofrece una aportación técnica en el campo de la robótica autónoma y la planificación de estrategias defensivas, sino que también invita a considerar el impacto que este tipo de desarrollos puede tener en la sociedad, la economía y el medioambiente. La tecnología debe ir siempre acompañada de una reflexión ética y responsable que garantice su integración sostenible en el entorno actual y futuro.

ANEXO II: REFLEXIONES ÉTICAS Y SOCIALES

El desarrollo de sistemas autónomos para la defensa y seguridad, como los drones utilizados en este trabajo, plantea una serie de implicaciones éticas y sociales que deben ser consideradas con responsabilidad. Si bien la presente investigación se centra en la optimización de estrategias de intercepción dentro de un entorno simulado, sus aplicaciones potenciales en el mundo real requieren una evaluación profunda de sus impactos tanto en el ámbito militar como en la sociedad en general.

Uno de los principales desafíos éticos asociados al uso de drones en operaciones defensivas es la automatización de la toma de decisiones en escenarios críticos.

A medida que la inteligencia artificial y los algoritmos de planificación de trayectorias avanzan, surge la preocupación sobre el grado de autonomía que estos sistemas deben tener en la selección de objetivos y la ejecución de acciones.

La posibilidad de que drones defensivos intercepten a atacantes sin intervención humana directa plantea cuestiones sobre la responsabilidad moral y legal de sus acciones, especialmente en situaciones donde pueden ocurrir errores o daños colaterales.

Otro aspecto ético importante es el potencial uso de estas tecnologías en contextos bélicos o de vigilancia masiva. Aunque los sistemas de defensa autónomos pueden desempeñar un papel clave en la protección de infraestructuras críticas y en la reducción de riesgos para el personal humano, su uso indebido podría derivar en escenarios donde se vulneren los derechos humanos o la privacidad de las personas. Es fundamental que el desarrollo y la implementación de estas tecnologías se realicen bajo regulaciones claras y principios éticos sólidos que garanticen su uso responsable.

Desde una perspectiva social, el uso de drones en tareas de defensa puede traer beneficios significativos en términos de seguridad y eficiencia operativa. Su capacidad para actuar de manera rápida y coordinada en entornos dinámicos permite mejorar la protección de instalaciones estratégicas y reducir el riesgo de pérdidas humanas en situaciones de emergencia. No obstante, la creciente dependencia de sistemas autónomos también puede generar desafíos en términos de empleo y reconfiguración del trabajo humano, especialmente en sectores donde la automatización reemplaza funciones tradicionalmente desempeñadas por personas.

Además, la percepción pública sobre los drones en escenarios de defensa y seguridad puede variar según su implementación y la transparencia en su uso. La falta de información clara sobre los alcances y limitaciones de estos sistemas puede generar desconfianza y preocupaciones sobre su impacto en la vida cotidiana. Por ello, es fundamental fomentar el debate social y la educación en torno a estas tecnologías, asegurando que la sociedad esté informada sobre sus beneficios y riesgos potenciales.

Para garantizar un desarrollo ético y socialmente responsable de este tipo de tecnologías, es imprescindible adoptar un enfoque basado en la ética del diseño y la toma de decisiones informadas. Esto implica establecer marcos normativos adecuados, promover la investigación en inteligencia artificial explicable y transparente, y fomentar la colaboración entre expertos en tecnología, ética y legislación.

En última instancia, la integración de drones en operaciones de defensa debe priorizar la seguridad, la equidad y el respeto por los derechos fundamentales. Solo mediante un enfoque consciente y responsable será posible aprovechar los beneficios de estas innovaciones sin comprometer los valores esenciales de la sociedad.

ANEXO III: CÓDIGO DESARROLLADO

En este anexo se presentan los códigos elaborados para la realización de las simulaciones y la posterior obtención de resultados. El código realizado por el alumno aparecerá sombreado en color gris:

Planificación de rutas geométricas (*test_path_planning.py*)

```
def computeSawPath(origin_point, target_point, parameters):
    try:
        x_f = origin_point[0]
        y_f = origin_point[1]
        x_t = target_point[0]
        y_t = target_point[1]
        num_path_points = parameters['num_points']
        path_rotation = parameters['path_rotation']
        signal_periods = parameters['signal_periods']
        signal_module = parameters['signal_module']

        # Generate x-coordinates with equal spacing
        x_coords = np.linspace(x_f, x_t, num_path_points)

        # Compute baseline y-coordinates (linear interpolation)
        baseline = np.linspace(y_f, y_t, num_path_points)

        # Generate the triangular wave, ensuring no initial or final bias
        path_points = [] # Start with the exact origin point
        for i in range(num_path_points):
            x = x_coords[i] # Linearly spaced x-coordinates

            # Phase progression from 0 to signal_periods
            phase = ((i+((num_path_points-1)/(4*signal_periods)))) / (num_path_points - 1)) * signal_periods

            # Generate a symmetric triangular wave
            y_triangle = signal_module * (1 - 4 * abs((phase % 1) - 0.5))

            # Final y-coordinate is baseline + triangular wave
            y = baseline[i] + y_triangle
            path_points.append((x,y))
        return path_points
    except:
        print('computePathSaw error:', str(traceback.format_exc()))
        return None

def plotSinePath(axis, origin_point, target_point, path_parameters):
    # Draw the origin and target point
    axis.scatter(origin_point[0], origin_point[1], color='black', label="Origin Point", s=40)
    axis.scatter(target_point[0], target_point[1], color='red', label="Target Point", s=40)
```

```
# Compute path points
path_points = computeSinePath(origin_point, target_point, path_parameters)
if path_points is None:
    print("Not found path points!!")
    return False
else:
    path_points_x, path_points_y = zip(*path_points)
    axis.scatter(path_points_x, path_points_y, label="Sine Path Points", color="blue", s=10)

# Configure plot
scenario_size = path_parameters['scenario_size']
axis.set_xlim(-int(scenario_size/2.0),int(scenario_size/2.0))
axis.set_ylim(-int(scenario_size/2.0),int(scenario_size/2.0))
axis.set_aspect('equal')
axis.set_title('Sine Path Points')
axis.set_xlabel('x')
axis.set_ylabel('y')
axis.grid()
return True

def computeSinePath(origin_point, target_point, parameters):
    try:
        x_f = origin_point[0]
        y_f = origin_point[1]
        x_t = target_point[0]
        y_t = target_point[1]
        num_path_points = parameters['num_points']
        path_rotation = parameters['path_rotation']
        signal_periods = parameters['signal_periods']
        signal_module = parameters['signal_module']

        # Generate x-coordinates linearly between origin and destination
        x_coords = np.linspace(x_f, x_t, num_path_points)

        # Generate y-coordinates as a sinusoidal wave
        # Linearly interpolate the baseline between y_f and y_t
        baseline = np.linspace(y_f, y_t, num_path_points)

        # Add the sinusoidal variation
        y_coords = baseline + signal_module * np.sin(2.0 * np.pi * signal_periods * (x_coords - x_f) / (x_t -
x_f))

        # Combine x and y into a list of tuples
        path_points = list(zip(x_coords, y_coords))
        return path_points
    except:
        print('computePathSine error:', str(traceback.format_exc()))
        return None
    return True
```

```
def computeCirclePath(origin_point, target_point, parameters):
    try:
        x_f = origin_point[0]
        y_f = origin_point[1]
        x_t = target_point[0]
        y_t = target_point[1]
        num_path_points = parameters['num_points']
        path_rotation = parameters['path_rotation']

        # Calculate the midpoint (center of the semicircle)
        center_x = (x_f + x_t) / 2.0
        center_y = (y_f + y_t) / 2.0

        # Calculate the radius of the semicircle
        radius = computeDistance((center_x, center_y), target_point) #np.sqrt((x_t - center_x)**2.0 + (y_t -
center_y)**2.0)
        beta = computeAngle(target_point, origin_point) #np.arctan2((y_f - y_t), (x_f - x_t))

        # Generate angles for the semicircle (from 0 to  $\pi$ )
        delta_angle = np.pi/num_path_points
        angles = np.linspace(delta_angle, np.pi, num_path_points) + beta

        # Create the semicircle points
        if path_rotation == 0:
            x_coords = center_x + radius * np.cos(angles) # X-coordinates
            y_coords = center_y + radius * np.sin(angles) # Y-coordinates
        else:
            x_coords = center_x - radius * np.cos(angles) # X-coordinates
            y_coords = center_y - radius * np.sin(angles) # Y-coordinates

        # Combine x and y into a list of tuples
        path_points = list(zip(x_coords, y_coords))
        return path_points
    except:
        print('computePathCircle error:', str(traceback.format_exc()))
        return None
    return True

def computeLinePath(origin_point, target_point, parameters):
    try:
        x_f = origin_point[0]
        y_f = origin_point[1]
        x_t = target_point[0]
        y_t = target_point[1]
        num_path_points = parameters['num_points']
        x_coords = [x_f + i * (x_t - x_f) / (num_path_points - 1) for i in range(1, num_path_points)]
        y_coords = [y_f + i * (y_t - y_f) / (num_path_points - 1) for i in range(1, num_path_points)]
        path_points = list(zip(x_coords, y_coords))
        return path_points
```

```
except:
    print('computePathLine error:' , str(traceback.format_exc()))
    return None
```

Mapa Tangencial de Visibilidad (*test_tangent_visibility_map.py*)

```
def plotTangentVisibilityMap(circles):
    """Plots the circles and the tangents that belongs to the visibility map."""
    risk_distance = 5.0
    fig, axis = plt.subplots()
    axis.set_aspect('equal')
    plotObstacles(axis, circles, risk_distance)
    internal_tangents, external_tangents = generateTangentVisibilityMap(circles, risk_distance)
    for internal_tanget in internal_tangents:
        (tAx, tAy), (tBx, tBy) = internal_tanget
        axis.plot([tAx, tBx], [tAy, tBy], 'g-')
        axis.scatter ([tAx, tBx], [tAy, tBy], color = 'green')

    for external_tanget in external_tangents:
        (tAx, tAy), (tBx, tBy) = external_tanget
        axis.plot([tAx, tBx], [tAy, tBy], 'b-')
        axis.scatter ([tAx, tBx], [tAy, tBy], color = 'blue')

    axis.set_aspect('equal')
    axis.set_title('Tangent visibility Map')
    axis.set_xlabel('x')
    axis.set_ylabel('y')
    plt.xlim(-150, 150)
    plt.ylim(-150, 150)
    plt.grid(True)

def generateTangentVisibilityMap(circles, risk_distance):
    internal_tangents = []
    external_tangents = []

    # Compute and plot tangents
    for i, circleA in enumerate(circles):
        for j, circleB in enumerate(circles):
            if i >= j:
                continue # Avoid duplicate pairs

            # Compute internal tangents
            circle_internal_tangents = computeCircleToCircleInternalTangets((circleA[0],circleA[1],circleA[2] +
risk_distance),
                                                                              (circleB[0],circleB[1],circleB[2] +
risk_distance))

            if not circle_internal_tangents is None:
                # Read circle internal tangents
                (t1Ax, t1Ay), (t1Bx, t1By), (t2Ax, t2Ay), (t2Bx, t2By) = circle_internal_tangents
```

```
# Check line of sight taking account other circles
free_line_of_sight_1 = True
free_line_of_sight_2 = True
for k, circleC in enumerate(circles):
    if k == i or k == j:
        continue
    if free_line_of_sight_1:
        line_of_sight_result_1 = computeLineOfSight(pointA=(t1Ax, t1Ay), pointB=(t1Bx, t1By),
circle=(circleC[0],circleC[1], circleC[2] + risk_distance))
        if not line_of_sight_result_1 is None:
            free_line_of_sight_1 = False
    if free_line_of_sight_2:
        line_of_sight_result_2 = computeLineOfSight(pointA=(t2Ax, t2Ay), pointB=(t2Bx, t2By),
circle=(circleC[0],circleC[1], circleC[2] + risk_distance))
        if not line_of_sight_result_2 is None:
            free_line_of_sight_2 = False
    if not free_line_of_sight_1 and not free_line_of_sight_2:
        break
    if free_line_of_sight_1:
        internal_tangets.append(((t1Ax, t1Ay), (t1Bx, t1By)))
    if free_line_of_sight_2:
        internal_tangets.append(((t2Ax, t2Ay), (t2Bx, t2By)))

# Compute external tangents
circle_external_tangents = computeCircleToCircleExternalTangets((circleA[0],circleA[1],circleA[2]+
risk_distance),
                                                                    (circleB[0],circleB[1],circleB[2]+
risk_distance))
    if not circle_external_tangents is None:
        (t1Ax, t1Ay), (t1Bx, t1By), (t2Ax, t2Ay), (t2Bx, t2By) = circle_external_tangents
        free_line_of_sight_1 = True
        free_line_of_sight_2 = True
        for k, circleC in enumerate(circles):
            if k == i or k == j:
                continue
            if free_line_of_sight_1:
                line_of_sight_result_1 = computeLineOfSight(pointA=(t1Ax, t1Ay), pointB=(t1Bx,
t1By), circle=(circleC[0],circleC[1], circleC[2] + risk_distance))
                if not line_of_sight_result_1 is None:
                    free_line_of_sight_1 = False
            if free_line_of_sight_2:
                line_of_sight_result_2 = computeLineOfSight(pointA=(t2Ax, t2Ay), pointB=(t2Bx, t2By),
circle=(circleC[0],circleC[1], circleC[2] + risk_distance))
                if not line_of_sight_result_2 is None:
                    free_line_of_sight_2 = False
            if not free_line_of_sight_1 and not free_line_of_sight_2:
                break
        if free_line_of_sight_1:
            external_tangets.append(((t1Ax, t1Ay), (t1Bx, t1By)))
```

```
        if free_line_of_sight_2:
            external_tangets.append(((t2Ax, t2Ay), (t2Bx, t2By)))
    return internal_tangets, external_tangets

def plotLineOfSight(pointA, pointB, circle):
    fig, ax = plt.subplots()

    # Unpack circle data
    (cx, cy, r) = circle

    # Draw the circle
    circle_patch = plt.Circle((cx, cy), r, color='gray', fill=False, linestyle='dotted', label="Obstacle")
    ax.add_patch(circle_patch)

    # Compute line of sight
    result = computeLineOfSight(pointA, pointB, circle)

    # Plot start & end points
    ax.scatter(*pointA, color='blue', label="Point A")
    ax.scatter(*pointB, color='green', label="Point B")

    # Plot the line of sight
    if not result is None:
        d, ipointA, ipointB = result
        plt.plot([pointA[0], pointB[0]], [pointA[1], pointB[1]], 'r--', linewidth=2, label="Blocked LoS")

        # Mark intersection points
        ax.scatter(*ipointA, color='red', label="Intersection A")
        ax.scatter(*ipointB, color='red', label="Intersection B")
    else:
        plt.plot([pointA[0], pointB[0]], [pointA[1], pointB[1]], 'g-', linewidth=2, label="Clear LoS")

    # Configure plot
    ax.set_xlim(cx - r - 2, cx + r + 2)
    ax.set_ylim(cy - r - 2, cy + r + 2)
    ax.set_aspect('equal')
    ax.legend()
    plt.grid()
    plt.title("Line of Sight (LoS) with Circular Obstacle")

def computeLineOfSight(pointA, pointB, circle):
    (cx, cy, r) = circle
    (x1, y1), (x2, y2) = pointA, pointB

    # Vector representation
    dx, dy = x2 - x1, y2 - y1
    A = dx**2 + dy**2
    B = 2 * ((x1 - cx) * dx + (y1 - cy) * dy)
    C = (x1 - cx)**2 + (y1 - cy)**2 - r**2
```

```
# Compute discriminant
discriminant = B**2 - 4 * A * C

if discriminant < 0:
    return None # No intersection (clear LoS)

# Compute intersection points
sqrt_disc = np.sqrt(discriminant)
t1 = (-B + sqrt_disc) / (2 * A)
t2 = (-B - sqrt_disc) / (2 * A)

intersection_points = []
for t in (t1, t2):
    if 0 <= t <= 1: # Only consider points along the segment
        ix, iy = x1 + t * dx, y1 + t * dy
        intersection_points.append((ix, iy))

if len(intersection_points) < 2:
    return None # Clear LoS

# Compute perpendicular distance from circle center to line
num = abs(dy * cx - dx * cy + x2 * y1 - y2 * x1)
den = np.sqrt(dx**2 + dy**2)
d = num / den # Distance from center to the line

# Find closest intersection points
ipointA, ipointB = sorted(intersection_points, key=lambda p: np.linalg.norm(np.array(p) - np.array(pointA)))

return d, ipointA, ipointB

def plotCircleToCircleExternalTangets(circleA, circleB):
    fig, ax= plt.subplots()

    c1x, c1y, r1 = circleA
    circle_patch1 = plt.Circle((c1x, c1y), r1, color = 'b', fill = False)
    ax.add_patch(circle_patch1)

    c2x, c2y, r2 = circleB
    circle_patch2 = plt.Circle((c2x, c2y), r2, color = 'g', fill = False)
    ax.add_patch(circle_patch2)

    #Compute and draw tangents points
    tangent_points = computeCircleToCircleExternalTangets(circleA, circleB)
    if tangent_points is None:
        print('A circle is inside the other circle')
    else:
        (tAx, tAy), (tBx, tBy), (t2Ax, t2Ay), (t2Bx, t2By) = tangent_points
        ax.plot([tAx, tBx], [tAy, tBy], 'g--', label="Tangent 1") # Plot line of tangent 1
```

```

ax.plot([t2Ax, t2Bx], [t2Ay, t2By], 'r--', label="Tangent 2") # Plot line of tangent 2
ax.scatter ([tAx, tBx, t2Ax, t2Bx], [tAy, tBy, t2Ay, t2By], color = 'red')

#configuration plot:
ax.set_xlim(-150, 150)
ax.set_ylim(-150, 150)
ax.set_aspect ('equal')
ax.legend()
plt.grid()

def computeCircleToCircleExternalTangets(circleA, circleB):
    c1x, c1y, r1 = circleA
    c2x, c2y, r2 = circleB
    dx, dy = c2x - c1x, c2y - c1y
    d = np.hypot(dx, dy)

    if d < abs(r1 - r2):
        print("One circle is inside the other, no external tangents exist.")
        return None

    theta = np.arccos((r1 - r2) / d)
    alpha = np.arctan2(dy, dx)

    theta = np.arccos((r1 - r2) / d)
    alpha = np.arctan2(dy, dx)

    # First external tangent
    tAx = c1x + r1 * np.cos(alpha + theta)
    tAy = c1y + r1 * np.sin(alpha + theta)
    tBx = c2x + r2 * np.cos(alpha + theta)
    tBy = c2y + r2 * np.sin(alpha + theta)

    # Second external tangent
    t2Ax = c1x + r1 * np.cos(alpha - theta)
    t2Ay = c1y + r1 * np.sin(alpha - theta)
    t2Bx = c2x + r2 * np.cos(alpha - theta)
    t2By = c2y + r2 * np.sin(alpha - theta)

    return (tAx, tAy), (tBx, tBy), (t2Ax, t2Ay), (t2Bx, t2By)

def plotCircleToCircleInternalTangets(circleA, circleB):
    fig, ax= plt.subplots()

    c1x, c1y, r1 = circleA
    circle_patch1 = plt.Circle ((c1x, c1y), r1, color = 'b', fill = False)
    ax.add_patch(circle_patch1)

    c2x, c2y, r2 = circleB

```

```
circle_patch2 = plt.Circle((c2x, c2y), r2, color = 'g', fill = False)
ax.add_patch(circle_patch2)

#Compute and draw tangents points
tangent_points = computeCircleToCircleInternalTangets(circleA, circleB)
if tangent_points is None:
    print ('A circle is inside the other circle')
else:
    (tAx, tAy), (tBx, tBy), (t2Ax, t2Ay), (t2Bx, t2By) = tangent_points
    ax.plot([tAx, tBx], [tAy, tBy], 'g--', label="Tangent 1") # Plot line of tangent 1
    ax.plot([t2Ax, t2Bx], [t2Ay, t2By], 'r--', label="Tangent 2") # Plot line of tangent 2
    ax.scatter ([tAx, tBx, t2Ax, t2Bx], [tAy, tBy, t2Ay, t2By], color = 'red')

#configuration plot:
ax.set_xlim(-100, 100)
ax.set_ylim(-100, 100)
ax.set_aspect ('equal')
ax.legend()
plt.grid ()

def computeCircleToCircleInternalTangets(circleA, circleB):
    c1x, c1y, r1 = circleA
    c2x, c2y, r2= circleB
    dx, dy = (c2x - c1x), (c2y - c1y)
    d = np.hypot(dx,dy)

    if d <= abs(r1+r2):
        print ('A circle is inside the other circle')
        return None

    # Ahora hay que definir los ángulos:
    theta = np.arctan2(dy, dx)
    alpha = np.arccos(np.clip((r1 + r2) / d, -1, 1)) # Clip to avoid math error

    # Compute first internal tangent
    tAx = c1x + r1 * np.cos(theta + alpha)
    tAy = c1y + r1 * np.sin(theta + alpha)
    tBx = c2x - r2 * np.cos(theta + alpha)
    tBy = c2y - r2 * np.sin(theta + alpha)

    # Compute second internal tangent
    t2Ax = c1x + r1 * np.cos(theta - alpha)
    t2Ay = c1y + r1 * np.sin(theta - alpha)
    t2Bx = c2x - r2 * np.cos(theta - alpha)
    t2By = c2y - r2 * np.sin(theta - alpha)

    return (tAx, tAy), (tBx, tBy), (t2Ax, t2Ay), (t2Bx, t2By)
```

```
#####
def plotPointToCircleTangets(point, circle):
    """Plots the circle, point, and tangent lines."""
    # Create figure and axis
    fig, ax = plt.subplots()

    # Draw the circle
    cx, cy, r = circle
    circle_patch = plt.Circle((cx, cy), r, color='b', fill=False)
    ax.add_patch(circle_patch)

    # Plot point
    px, py = point
    ax.scatter(px, py, color='black', label="External Point")
    ax.scatter(cx, cy, color='blue', label="Circle Center")

    # Compute and draw tangent points
    tangent_points = computePointToCircleTangets(point, circle)
    if tangent_points is None:
        print("Point is inside the circle; no tangents exist.")
    else:
        (t1x, t1y), (t2x, t2y) = tangent_points
        px, py = point
        ax.plot([px, t1x], [py, t1y], 'g--', label="Tangent 1") # Plot line of tangent 1
        ax.plot([px, t2x], [py, t2y], 'r--', label="Tangent 2") # Plot line of tangent 2
        ax.scatter([t1x, t2x], [t1y, t2y], color='red') # Plot tangent points

    # Configure plot
    ax.set_xlim(-15,15)
    ax.set_ylim(-15,15)
    ax.set_aspect('equal')
    ax.legend()
    plt.grid()

def computePointToCircleTangets(point, circle):
    px, py = point
    cx, cy, r = circle
    dx, dy = px - cx, py - cy
    d = np.hypot(dx, dy) # Distance from point to circle center

    if d < r:
        return None # Point is inside the circle; no tangents exist.

    # Compute the angle to the external point
    theta = np.arctan2(dy, dx)
    alpha = np.arccos(r / d) # Angle between center-line and tangent line

    # Compute the tangent points
    t1x = cx + r * np.cos(theta + alpha)
```

```
t1y = cy + r * np.sin(theta + alpha)
t2x = cx + r * np.cos(theta - alpha)
t2y = cy + r * np.sin(theta - alpha)

return (t1x, t1y), (t2x, t2y)
```

Navegación entre obstáculos. (*test_obstacle_avodiance.py*)

```
def computePathWithObstacleAvoidanceTangentVisibilityMap(origin_point, target_point, path_points, obstacles,
parameters):
    try:
        new_path_points = []
        bridge_points = []
        risk_distance = parameters['risk_distance']
        min_path_size = parameters['min_path_size']
        bridge_point_A = None
        bridge_point_B = None
        previous_point = None
        path_counter = 0

        # Compute the Tangent Visibility Graph
        tangent_visibility_map = computeTangentVisibilityGraph(origin_point, target_point, obstacles, parameters)
        print('tangent_visibility_map=',tangent_visibility_map)

        if False:
            for path_point in path_points:
                collision_obstacles = computePointCollisionWithObstacles(path_point, obstacles, risk_distance)
                if collision_obstacles is None:
                    if bridge_point_A is None and bridge_point_B is None: # State 1
                        new_path_points.append(path_point) # Add point to path
                    elif not bridge_point_A is None and bridge_point_B is None: # State 2
                        if path_counter > min_path_size:
                            # Set bridge point B
                            bridge_point_B = path_point

                            # ????????????????????????????????????????????????????????????? #
                            # calculate bridges points with A* and add to new path points
                            a_star_path_points, a_star_closed_list = AStarSearch(start=bridge_point_A,
goal=bridge_point_B, obstacles=obstacles,
neighbors_function=getObstacleAv
oidanceTangentVisibilityMapNeighbors,
distance_function=computeDistance,
parameters=parameters)
                            for a_star_path_point in a_star_path_points:
                                #print('a_star_path_point=',a_star_path_point)
                                new_path_points.append(a_star_path_point) # Add point to path
                                # ????????????????????????????????????????????????????????????? #
```

```

        # Add new bridge points
        bridge_points.append((bridge_point_A, bridge_point_B))

    # Return to state 1
    bridge_point_A = None
    bridge_point_B = None
    else:
        path_counter += 1
    else: # Collision with obstacle !!!
        if bridge_point_A is None and bridge_point_B is None: # State 1
            if not previous_point is None:
                # Set bridge point A
                bridge_point_A = previous_point
                path_counter = 0
                path_counter += 1
            # Memorize point
            previous_point = path_point

    return new_path_points, bridge_points
except:
    print('computePathWithObstacleAvoidanceTangentVisibilityMap error:', str(traceback.format_exc()))
    return None

def computeTangentVisibilityGraph(origin_point, target_point, obstacles, parameters):
    tangent_visibility_map = {}
    risk_distance = parameters['risk_distance']
    internal_tangets, external_tangets = generateTangentVisibilityMap(obstacles, risk_distance)
    print('internal_tangets=', internal_tangets)

    return tangent_visibility_map

def getObstacleAvoidanceTangentVisibilityMapNeighbors(node, obstacles, parameters):
    risk_distance = parameters['risk_distance']
    scenario_radius = parameters['scenario_radius']
    a_plus_grid_size = parameters['a_plus_grid_size']
    neighbors = []
    return neighbors

#####
def computePathWithObstacleAvoidanceGridMap(origin_point, target_point, path_points, obstacles, parameters):
    try:

        new_path_points = []
        bridge_points = []
        risk_distance = parameters['risk_distance']
        min_path_size = parameters['min_path_size']
        bridge_point_A = None
        bridge_point_B = None
        previous_point = None

```

```
for path_point in path_points:
    collision_obstacles = computePointCollisionWithObstacles(path_point, obstacles, risk_distance)
    if collision_obstacles is None: # Point is free of collision !!!
        if bridge_point_A is None and bridge_point_B is None: # State 1
            new_path_points.append(path_point) # Add point to path

        elif not bridge_point_A is None and bridge_point_B is None: # State 2
            # Set bridge point B
            bridge_point_B = path_point

            # ?????????????????????????????????????????????????????????????? #
            # calculate bridges points with A* and add to new path points a_star_path_points,
a_star_closed_list = AStarSearch(start=bridge_point_A, goal=bridge_point_B, obstacles=obstacles,
                                neighbors_function=getObstacleAvoidanceGridMapNeighbors,
distance_function=computeDistance, parameters=parameters)
            if not a_star_path_points is None:
                for a_star_path_point in a_star_path_points:
                    #print('a_star_path_point=',a_star_path_point)
                    new_path_points.append(a_star_path_point) # Add point to path
                # ?????????????????????????????????????????????????????????????? #

            # Add new bridge points
            bridge_points.append((bridge_point_A, bridge_point_B))

            # Return to state 1
            bridge_point_A = None
            bridge_point_B = None

        else: # Collision with obstacle !!!
            if bridge_point_A is None and bridge_point_B is None: # State 1
                if not previous_point is None:
                    # Set bridge point A
                    bridge_point_A = previous_point

            # Memorize point
            previous_point = path_point
    return new_path_points, bridge_points
except:
    print('computePathWithObstacleAvoidanceGridMap error:' , str(traceback.format_exc()))
    return None

def getObstacleAvoidanceGridMapNeighbors(node, obstacles, parameters):
    risk_distance = parameters['risk_distance']
    scenario_radius = parameters['scenario_radius']
    a_plus_grid_size = parameters['a_plus_grid_size']
    neighbors_points = [
        (node[0] + a_plus_grid_size, node[1]),
        (node[0] + a_plus_grid_size, node[1] + a_plus_grid_size),
        (node[0], node[1] + a_plus_grid_size),
        (node[0] - a_plus_grid_size, node[1] + a_plus_grid_size),
```

```
(node[0] - a_plus_grid_size, node[1]),
(node[0] - a_plus_grid_size, node[1] - a_plus_grid_size),
(node[0], node[1] - a_plus_grid_size),
(node[0] + a_plus_grid_size, node[1] - a_plus_grid_size)
]
neighbors = []
for neighbor_point in neighbors_points:
    if abs(neighbor_point[0]) > scenario_radius: # / 2.0:
        continue
    if abs(neighbor_point[1]) > scenario_radius: # / 2.0:
        continue
    collision_obstacles = computePointCollisionWithObstacles(neighbor_point, obstacles, risk_distance)
    if collision_obstacles is None:
        neighbors.append(neighbor_point)
return neighbors

def plotObstacleAvoidance(axis, origin_point, target_point, obstacles, parameters, path_points, bridge_points):
    if len(path_points) > 1:
        path_points_x, path_points_y = zip(*path_points)
        axis.scatter(path_points_x, path_points_y, color="red", s=10, label='A* optimal path width grip map')
        for bridge_pointA, bridge_pointB in bridge_points:
            axis.scatter(bridge_pointA[0], bridge_pointA[1], color="gray", s=100, marker='x')
            axis.scatter(bridge_pointB[0], bridge_pointB[1], color="gray", s=100, marker='x')
        risk_distance = parameters['risk_distance']
        plotObstacles(axis, obstacles, risk_distance)
    return True
```

Simulación de escenarios (*test_scenarios.py*)

```
def plotSimulation(simulation_data, plot_name):
    # simulation_data : List of tuples
    #(step, simulation_time, simulation_state, num_active_defenders, num_active_attackers, total_cost,
    attackers_cost, defenders_cost, step_attackers_cost, step_defenders_cost)
    step, simulation_time, simulation_state, num_active_defenders, num_active_attackers, total_cost,
    attackers_cost, defenders_cost, step_attackers_cost, step_defenders_cost = zip(*simulation_data)

    # Determine winner based on final state
    final_state = simulation_state[-1]
    if final_state == 1:
        winner_label = "Attacker wins"
    elif final_state == 2:
        winner_label = "Defenders win"
    else:
        winner_label = "No winner"

    # Create figure and axis
    fig, axes = plt.subplots(2, 1, figsize = (10, 10), sharex = True)

    # First subplot: Cost Values
    axes[0].plot(simulation_time, total_cost, color='black', label= 'Total_cost', linewidth = 2)
```

```
axes[0].plot(simulation_time, attackers_cost, color='red', label = 'Attackers_cost', linestyle = 'dashed')
axes[0].plot(simulation_time, defenders_cost, color='blue', label = 'Defenders_cost', linestyle = 'dotted')

axes[0].set_title('Simulation Cost Over Time')
axes[0].set_ylabel('Cost')
axes[0].grid(True)
axes[0].legend(['Total Cost', 'Attackers cost', 'Defenders cost'])
axes[0].set_xlim(0, simulation_time[-1])
axes[0].set_ylim(min(defenders_cost), max(attackers_cost))

# Add winner text in the upper middle of the Cost Graph
axes[0].text(simulation_time[len(simulation_time)//3], max(total_cost) * 0.9, winner_label, fontsize = 10,
fontweight = 'bold', color = 'black', ha = 'center', bbox = dict(facecolor = 'white', alpha = 0.6))

# Second subplot: Active Units over time
axes[1].plot(simulation_time, num_active_defenders, color = 'blue', linestyle = 'solid', marker='o', label
='Active Defenders', linewidth = 2)
axes[1].plot(simulation_time, num_active_attackers, color = 'red', linestyle='solid', marker='o', label = 'Active
Attackers', linewidth = 2)

axes[1].set_title('Active Units Over Time')
axes[1].set_xlabel('Time (s)')
axes[1].set_ylabel('Number of Active Units')
axes[1].grid(True)
axes[1].legend(['Active defenders', 'Active attackers'])
axes[1].set_xlim(0, simulation_time[-1])
axes[1].set_ylim(0, max(num_active_defenders) + 2)

"""plt.xlim(0, simulation_time[-1])
plt.ylim(min(defenders_cost), max(attackers_cost))
plt.grid(True)"""

plt.tight_layout()

# Save figure to png
plt.savefig(plot_name, dpi=100)
plt.show()

def createDefenderTargetPoint(defender, attacker, obstacles, defender_parameters, attacker_parameters, strategy):
    target_point = None
    if strategy == 0:
        target_point = (attacker['x'], attacker['y'])
    elif strategy == 1:
        collision_distance = defender_parameters['collision_distance']
        defensive_radius = defender_parameters['defensive_radius']
        attacker_point = (attacker['x'], attacker['y'])
        defender_point = (defender['x'], defender['y'])
```

```

defender_attacker_distance = computeDistance(defender_point, attacker_point)
if defender_attacker_distance < 20*collision_distance:
    target_point = (attacker['x'], attacker['y'])
else:
    position = defender['position']
    attacker_position_angle = computeAngle((0.0,0.0), attacker_point)
    attacker_position_distance = computeDistance((0.0,0.0), attacker_point)
    #delta_yaw = position*(10*(1.0-np.exp(-0.1*distance))+2)*np.pi/180.0
    if position == 0:
        delta_yaw = 0
    elif position % 2 == 0:
        delta_yaw = (position-1)*(2*(1.0-np.exp(-5*attacker_position_distance)))*np.pi/180.0
    else:
        delta_yaw = -position*(2*(1.0-np.exp(-5*attacker_position_distance)))*np.pi/180.0
    attacker_position_angle += delta_yaw
    #print('position=', position, ' delta_yaw=',delta_yaw*180.0/np.pi)
    target_point = (defensive_radius*np.cos(attacker_position_angle),
defensive_radius*np.sin(attacker_position_angle))
    #print('id=', defender['id'], ' position=',defender['position'])

elif strategy == 2:
    collision_distance = defender_parameters['collision_distance']
    base_defensive_radius = defender_parameters['defensive_radius']

    attacker_point = (attacker['x'], attacker['y'])
    defender_point = (defender['x'], defender['y'])
    defender_attacker_distance = computeDistance(defender_point, attacker_point)

    # Dynamic Defensive Radius Adjustment
    attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
    defensive_radius = max(base_defensive_radius, 0.6*attacker_position_distance)
    attacker['defensive_zone_radius'] = defensive_radius
    if defender_attacker_distance < 20*collision_distance:
        target_point = (attacker['x'], attacker['y'])
    else:
        position = defender ['position']
        attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)

        #compute delta_yaw based on position
        if position == 0:
            delta_yaw = 0
        elif position % 2 == 0:
            delta_yaw = (position -1)*(2*(1.0 - np.exp(-5*attacker_position_distance)))*np.pi/180.0
        else:
            delta_yaw = -position*(2*(1.0 - np.exp(-5*attacker_position_distance)))* np.pi/180.0

        attacker_position_angle += delta_yaw

    #Compute target point based on adjusted defensive_radius

```

```
target_point = (defensive_radius*np.cos(attacker_position_angle),
defensive_radius*np.sin(attacker_position_angle))

elif strategy == 3:
    attacker_radius = attacker_parameters['attacker_radius'] #fixed radius
    attacker_point = (attacker['x'], attacker['y'])
    attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
    defensive_radius = max(attacker_radius, 0.3*attacker_position_distance)
    position = defender ['position']
    defender_point = (defender['x'], defender['y'])
    defender_attacker_distance = computeDistance(defender_point, attacker_point)
    collision_distance = defender_parameters['collision_distance']

    if defender_attacker_distance < 20*collision_distance:
        target_point = (attacker['x'], attacker['y'])
        attacker['defensive_radius'] = collision_distance
    else:

        # compute angle for attacker placement
        attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)

        # Evenly distribute defenders around attacker_point
        if position == 0:
            delta_yaw = 0
        elif position % 2 == 0:
            delta_yaw = (position -1)*(2*np.pi/180.0) #Fixed small angle
        else:
            delta_yaw = -position*(2*np.pi/180.0) #Fixed small angle
        attacker_position_angle += delta_yaw

        #compute target point at a fixed distance from the attacker
        target_point = (attacker_point[0] + defensive_radius*np.cos(attacker_position_angle + np.pi),
attacker_point[1] + defensive_radius*np.sin(attacker_position_angle + np.pi))
        #Save attacker actual defensive radius to make its plot
        attacker['defensive_radius'] = defensive_radius

elif strategy == 4:
    collision_distance = defender_parameters['collision_distance']
    base_defensive_radius = defender_parameters['defensive_radius']
    num_defenders_foreach_attacker = defender_parameters['num_defenders_foreach_attacker']
    attacker_point = (attacker['x'], attacker['y'])
    defender_point = (defender['x'], defender['y'])
    defender_attacker_distance = computeDistance (defender_point, attacker_point)
    position = defender['position']
    attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)
    attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
    if defender_attacker_distance < 20*collision_distance:
        target_point = (attacker['x'], attacker['y'])
    else:
        if position > int(num_defenders_foreach_attacker/2): #position % 2 == 0:
```

```

        target_point = (attacker['x'], attacker['y'])
    else:
        defensive_radius = base_defensive_radius
        if position == 0:
            delta_yaw = 0
        elif position % 2 == 0:
            delta_yaw = (position-1)*(2*np.pi/180.0) #Fixed small angle
        else:
            delta_yaw = -position*(2*np.pi/180.0) #Fixed small angle
        attacker_position_angle += delta_yaw
        target_point = (defensive_radius*np.cos(attacker_position_angle),
defensive_radius*np.sin(attacker_position_angle))

    elif strategy == 5:
        collision_distance = defender_parameters['collision_distance']
        base_defensive_radius = defender_parameters['defensive_radius']
        num_defenders_foreach_attacker = defender_parameters['num_defenders_foreach_attacker']
        attacker_point = (attacker['x'], attacker['y'])
        defender_point = (defender['x'], defender['y'])
        defender_attacker_distance = computeDistance (defender_point, attacker_point)
        position = defender['position']
        attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)
        attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
        if defender_attacker_distance < 20*collision_distance:
            target_point = (attacker['x'], attacker['y'])
        else:
            if position < int(num_defenders_foreach_attacker/2):
                defensive_radius = base_defensive_radius
            else:
                defensive_radius = max(1.3*base_defensive_radius, 0.6*attacker_position_distance)

        # Compute new target point based on the computed defensive_radius
        if position == 0 or position == int(num_defenders_foreach_attacker/2):
            delta_yaw = 0
        elif position % 2 == 0:
            delta_yaw = (position-1)*(2*np.pi/180.0) #Fixed small angle
        else:
            delta_yaw = -position*(2*np.pi/180.0) #Fixed small angle
        attacker_position_angle += delta_yaw
        target_point=(defensive_radius*np.cos(attacker_position_angle), defensive_radius*np.sin(attacker_position_angle))

    if target_point is None:
        #print('defensive_radius=', defensive_radius , 'attacker_position_angle=',attacker_position_angle, '
attacker=', attacker['x'] )
        target_point = (attacker['x'], attacker['y'])
    return target_point

def moveDefender(defender, attacker, obstacles, time_step, defenders_strategy, parameters, attacker_parameters):
# Geometric movement: velocity and yaw

```

```
velocity = defender['velocity']
path_point_index = defender['path_point_index']
#path_points = defender['path_points']
#collision_distance = parameters['collision_distance']
defender_point = (defender['x'], defender['y'])
target_point = createDefenderTargetPoint(defender, attacker, obstacles, parameters, attacker_parameters,
strategy = defenders_strategy)
distance = computeDistance(defender_point, target_point)
run_distance = velocity*time_step
#print('run_distance=', run_distance, ' distance=', distance)
if distance <= run_distance:
    defender['x'] = target_point[0]
    defender['y'] = target_point[1]
    defender['path_point_index'] = len(defender['path_points']) - 1
else:
    free_path_points, a_star_closed_list = AStarSearch(start=defender_point, goal=target_point,
obstacles=obstacles,
                                                    neighbors_function=getObstacleAvoidanceGridMapNeighbors,
                                                    distance_function=computeDistance,
parameters=parameters)
    if not free_path_points is None:
        defender['path_points'] = free_path_points
    if not defender['path_points'] is None:
        move_point_data = moveGeometric (object_position = (defender['x'], defender['y']) , path_points =
defender['path_points'],
                                                    velocity=velocity, time_step=time_step,
current_index=path_point_index)
        defender['x'] = move_point_data[0]
        defender['y'] = move_point_data[1]
        defender['yaw'] = move_point_data[2]
        defender['path_point_index'] = move_point_data[3]
    return

def changeMapOfDefendersAttacker (defenders, attackers, attacker_id):
    defenders_attackers_map = []
    free_attackers_id = [attacker_id for attacker_id, attacker in attackers.items() if attacker['state'] == 1]
    if len(free_attackers_id) > 0:
        for defender_id, defender in defenders.items():
            if defender['attacker_id'] == attacker_id:
                selected_attacker_id = np.random.choice(free_attackers_id)
                #print ('defender_id=', defender_id , ' changes to ', selected_attacker_id)
                defender['attacker_id'] = selected_attacker_id
                defenders_attackers_map.append((defender_id,selected_attacker_id))
    return defenders_attackers_map

def createSimulation (num_attackers, attacker_parameters, num_defenders_foreach_attacker, defender_parameters,
num_obstacles, obstacle_parameters):
    attackers = {}
    defenders = {}
    obstacles = list ()
```

```

max_attackers_velocity = attacker_parameters['max_attackers_velocity']
max_defenders_velocity = defender_parameters['max_defenders_velocity']
obstacle_min_radius = obstacle_parameters['min_radius']
obstacle_max_radius = obstacle_parameters['max_radius']
scenario_radius = obstacle_parameters['scenario_radius']
defensive_radius = obstacle_parameters['defensive_radius']
risk_distance = attacker_parameters['risk_distance']

if num_obstacles > 0:
    for obstacle_id in range(1, num_obstacles+1):
        distance = np.random.uniform(2*defensive_radius, scenario_radius)
        angle = np.random.uniform(0, 2*np.pi)
        radius = np.random.uniform(obstacle_min_radius, obstacle_max_radius)
        obstacles.append((distance*np.cos(angle), distance*np.sin(angle), radius))

if num_attackers > 0:
    defender_id = 1
    for attacker_id in range(1, num_attackers+1):
        #print('attacker_id=', attacker_id)
        attacker_angle = np.random.uniform(0, 2*np.pi)
        attacker_point_x = scenario_radius*np.cos(attacker_angle)
        attacker_point_y = scenario_radius*np.sin(attacker_angle)
        attacker_point = (attacker_point_x, attacker_point_y)
        attacker_parameters['velocity'] = np.random.uniform(0.50, 1.0)*max_attackers_velocity
        attackers[attacker_id] = createAttacker(attacker_id, attacker_point, obstacles, parameters =
attacker_parameters)

    num_defenders = 0
    if num_defenders_foreach_attacker > 0:
        while num_defenders < num_defenders_foreach_attacker:
            defender_angle = attacker_angle + np.random.uniform(-90*np.pi/180.0, 90*np.pi/180.0)
            defender_distance = np.random.uniform(defensive_radius, 0.3*scenario_radius)
            defender_point_x = defender_distance*np.cos(defender_angle)
            defender_point_y = defender_distance*np.sin(defender_angle)
            defender_point = (defender_point_x, defender_point_y)
            collision = computePointCollisionWithObstacles(defender_point, obstacles, risk_distance)
            #print('collision=', collision)
            if collision is None:
                #print('defender_id=', defender_id)
                defender_parameters['velocity'] = np.random.uniform(0.50, 1.0)*max_defenders_velocity
                defenders[defender_id] = createDefender(defender_id, defender_point, obstacles, attacker_id,
attacker_point, parameters=defender_parameters)
                defenders[defender_id] ['position'] = num_defenders
                defender_id += 1
                num_defenders += 1
    return attackers, defenders, obstacles

def checkDefenderDefendersCollision (defender, defenders, collision_distance):
    defender_point = (defender['x'], defender['y'])
    defender_id = defender['id']

```

```
collision_result = False
for col_defender_id, col_defender in defenders.items():
    #print('defender_id=', defender_id)
    if col_defender['id'] == defender_id:
        continue
    if col_defender['state'] == 1: # Active
        col_defender_id = col_defender['id']
        col_defender_point = (col_defender['x'], col_defender['y'])
        distance = computeDistance(defender_point, col_defender_point)
        if distance <= collision_distance:
            collision_result = True
            defender['state'] = 0
            col_defender['state'] = 0
            break
return collision_result

def checkDefenderAttackersCollision (defender, attackers, collision_distance):
    defender_point = (defender['x'], defender['y'])
    defender_id = defender['id']
    collision_result = False
    for attacker_id, attacker in attackers.items():
        #print('attacker_id=', attacker_id)
        if attacker['state'] == 1: # Active
            attacker_id = attacker['id']
            attacker_point = (attacker['x'], attacker['y'])
            distance = computeDistance(defender_point, attacker_point)
            if distance <= collision_distance:
                collision_result = True
                defender['state'] = 0
                attackers[attacker_id]['state'] = 0
                break
    return collision_result

def runSimulation (attackers, attacker_parameters, defenders, defender_parameters, obstacles, parameters):
    show_animation = parameters['show_animation']
    show_cost = parameters['show_cost']
    num_steps = parameters['num_steps']
    time_step = parameters['time_step']
    collision_distance = parameters['collision_distance']
    scenario_radius = parameters['scenario_radius']
    defenders_strategy = parameters['defenders_strategy']
    num_defenders = parameters['num_defenders']
    num_attackers = parameters['num_attackers']
    if show_animation:
        fig, axis, attackers_patches, defenders_patches = plotScenario(attackers, defenders, obstacles, parameters
= attacker_parameters)
        fig.canvas.draw()
        fig.canvas.flush_events()

    simulation_time = 0.0
```

```

simulation_state = 0 #0 running; 1:attacker wins; 2:defender wins
simulation_data = []
attackers_cost = 0.0
defenders_cost = 0.0
k = -np.log (0.1)/scenario_radius
for step in range(num_steps):
    step_attackers_cost = 0.0
    step_defenders_cost = 0.0
    num_active_defenders = 0
    defenders_attackers_map = None
    plot_defender_radius = True
    for defender_id, defender in defenders.items():
        if defender['state'] == 1: # Active
            num_active_defenders += 1
            defender_point = (defender['x'], defender['y'])
            defender_attacker_id = defender['attacker_id']
            defender_attacker = attackers[defender_attacker_id]
            defender_attacker_point = (defender_attacker['x'], defender_attacker['y'])
            defender_distance_to_target = computeDistance(defender_point, defender_attacker_point)
            step_defenders_cost -= (1/num_defenders) * np.exp(-5*k*defender_distance_to_target)
            moveDefender (defender, defender_attacker, obstacles, time_step, defenders_strategy, defender_parameters,
            attacker_parameters)

            collision_result = checkDefenderAttackersCollision (defender, attackers, collision_distance)
            if collision_result:
                defenders_attackers_map = changeMapOfDefendersAttacker (defenders, attackers, defender_attacker_id)
                step_defenders_cost -= 5.0/num_attackers
                num_active_defenders -= 1
            else:
                collision_result = False #checkDefenderDefendersCollision (defender, defenders, collision_distance)
                if collision_result:
                    step_defenders_cost += 5.0/num_defenders
                    num_active_defenders -= 2
                else:
                    if show_animation:
                        defender_patch = defenders_patches[defender_id]
                        defender_patch['circle'].center = (defender['x'], defender['y'])
                        radius = 15
                        vector_length = 2 * radius
                        end_x = defender['x'] + vector_length * np.cos(defender['yaw'])
                        end_y = defender['y'] + vector_length * np.sin(defender['yaw'])
                        defender_patch['yaw_line'].set_data([defender['x'],end_x], [defender['y'], end_y])
                        if not defender['path_points'] is None:
                            defender_patch['path_points']. set_offsets(defender['path_points'])

    if not defenders_attackers_map is None:
        #print ('defenders_attackers_map=', defenders_attackers_map)
        for (defender_id, attacker_id) in defenders_attackers_map:
            defenders[defender_id] ['attacker_id'] = attacker_id

```

```
num_active_attackers = 0
for attacker_id, attacker in attackers.items():
    #print('attacker_id=', attacker_id)
    if attacker['state'] == 1: # Active
        num_active_attackers += 1
        target_point = attacker['target_point']
        attacker_point = (attacker['x'], attacker['y'])
        attacker_distance_to_target = computeDistance(target_point, attacker_point)
        step_attackers_cost += (1.0/num_attackers)*neap(-k*attacker_distance_to_target)
        run_distance = attacker['velocity'] *time_step
        if attacker_distance_to_target <= collision_distance or run_distance >= attacker_distance_to_target:
            simulation_state = 1 # Attacker wins
            step_attackers_cost += 10.0 #/num_attackers
            attacker['state'] = 0
            num_active_attackers -= 1
        else:
            moveAttacker(attacker, time_step)
            if show_animation:
                attacker_patch = attackers_patches[attacker_id]
                attacker_patch['circle'].center = (attacker['x'], attacker['y'])
                radius = 15
                vector_length = 2 * radius
                end_x = attacker['x'] + vector_length * np.cos(attacker['yaw'])
                end_y = attacker['y'] + vector_length * np.sin(attacker['yaw'])
                attacker_patch['yaw_line'].set_data([attacker['x'],end_x], [attacker['y'], end_y])

                if 'defensive_patch' in attacker_patch.keys():
                    attacker_patch['defensive_patch'].set_center((attacker['x'], attacker['y']))
                    attacker_patch['defensive_patch'].set_radius(attacker['defensive_radius'])

                if 'defensive_zone' in attacker_patch.keys():
                    attacker_patch['defensive_zone'].set_radius(attacker['defensive_zone_radius'])

    if simulation_state == 0:
        if num_active_attackers == 0: # Defender wins
            step_defenders_cost -= 10.0#/num_defenders
            simulation_state = 2 # Defender wins

# print (f'Simulation ends at step {step}. Defender wins with total cost ={total_cost}!!!')
attackers_cost += step_attackers_cost
defenders_cost += step_defenders_cost
total_cost = attackers_cost + defenders_cost
if show_cost and step % 20 == 0:
    print ('.....')
    print ('step=', step, ' time=', simulation_time, ' total_cost=',total_cost , ' num_active_attackers=',
num_active_attackers, ' num_active_defenders=',num_active_defenders, ' simulation_state=',simulation_state)
    #print ('step_attackers_cost=',step_attackers_cost , ' step_defenders_cost=', step_defenders_cost)
    step_simulation_data = (step, simulation_time, simulation_state, num_active_defenders,
num_active_attackers, float(total_cost),float(attackers_cost), float(defenders_cost), float(step_attackers_cost),
float(step_defenders_cost))
```

```

simulation_data.append(step_simulation_data)
#print(step_simulation_data)
if show_animation:
    fig.canvas.draw()
    fig.canvas.flush_events()
    plt.pause(0.001)
    #plt. waitforbuttonpress()
if simulation_state > 0:
    break
simulation_time += time_step

print ( '%%%%%%%%%%' )
if simulation_state == 2:
    print (f'Defender wins. Time ={simulation_time}. Defenders:{num_defenders}. Attackers:{num_attackers}.
Cost ={total_cost}, strategy={defenders_strategy}')
elif simulation_state ==1:
    print (f'Attacker wins. Time ={simulation_time}. Defenders:{num_defenders}. Attackers:{num_attackers}.
Cost ={total_cost}, strategy={defenders_strategy}')
else:
    print (f'Nobody wins. Time ={simulation_time}. Defenders: {num_defenders}. Attackers: {num_attackers}.
Cost ={total_cost}, strategy={defenders_strategy}')
print ( '%%%%%%%%%%' )
if show_animation:
    plt.show()
return simulation_data

def moveGeometric(object_position, path_points, velocity, time_step, current_index=0):
    point_a = object_position
    num_path_points = len(path_points)
    travel_distance = velocity*time_step
    #print('.....moveGeometric.....')
    if current_index > num_path_points - 1:
        current_index = num_path_points - 1
        point_b = path_points[current_index]
        distance_a_b = computeDistance(point_a, point_b)
        distance_to_move = distance_a_b
    else:
        point_b = path_points[current_index]
        distance_a_b = computeDistance(point_a, point_b)
        #print ('travel_distance=',travel_distance , ' distance_a_b=',distance_a_b)
        if travel_distance < distance_a_b:
            distance_to_move = travel_distance
        else:
            distance_to_move = travel_distance - distance_a_b
            current_index += 1
            if current_index > num_path_points - 1:
                current_index = num_path_points - 1
            point_a = point_b
            point_b = path_points[current_index]

```

```
#print ('distance_to_move=', distance_to_move)
yawAB = computeAngle(point_a, point_b)
move_point_data = (point_a[0] + distance_to_move*np.cos(yawAB), point_a[1] + distance_to_move*np.sin(yawAB),
yawAB, current_index)
return move_point_data

def moveAttacker (attacker, time_step): # Geometric movement: velocity and yaw
    velocity = attacker['velocity']
    path_point_index = attacker['path_point_index']
    path_points = attacker['path_points']
    move_point_data = moveGeometric (object_position = (attacker['x'], attacker['y']) , path_points = path_points,
    velocity=velocity, time_step=time_step, current_index=path_point_index)
    attacker['x'] = move_point_data[0]
    attacker['y'] = move_point_data[1]
    attacker['yaw'] = move_point_data[2]
    attacker['path_point_index'] = move_point_data [3]
    return
```

Estrategias defensivas. (*test_simulations.py*)

```
def createDefenderTargetPoint(defender, attacker, obstacles, defender_parameters, attacker_parameters, strategy):
    target_point = None
    if strategy == 0:
        target_point = (attacker['x'], attacker['y'])
    elif strategy == 1:

        collision_distance = defender_parameters['collision_distance']
        defensive_radius = defender_parameters['defensive_radius']
        attacker_point = (attacker['x'], attacker['y'])
        defender_point = (defender['x'], defender['y'])
        defender_attacker_distance = computeDistance(defender_point, attacker_point)
        if defender_attacker_distance < 20*collision_distance:
            target_point = (attacker['x'], attacker['y'])
        else:
            position = defender['position']
            attacker_position_angle = computeAngle((0.0,0.0), attacker_point)
            attacker_position_distance = computeDistance((0.0,0.0), attacker_point)
            #delta_yaw = position*(10*(1.0-np.exp(-0.1*distance))+2)*np.pi/180.0
            if position == 0:
                delta_yaw = 0
            elif position % 2 == 0:
                delta_yaw = (position-1)*(2*(1.0-np.exp(-5*attacker_position_distance)))*np.pi/180.0
            else:
                delta_yaw = -position*(2*(1.0-np.exp(-5*attacker_position_distance)))*np.pi/180.0
            attacker_position_angle += delta_yaw
            #print('position=', position, ' delta_yaw=',delta_yaw*180.0/np.pi)
            target_point = (defensive_radius*np.cos(attacker_position_angle),
defensive_radius*np.sin(attacker_position_angle))
            #print('id=', defender['id'], ' position=',defender['position'])
```

```
elif strategy == 2:
    collision_distance = defender_parameters['collision_distance']
    base_defensive_radius = defender_parameters['defensive_radius']

    attacker_point = (attacker['x'], attacker['y'])
    defender_point = (defender['x'], defender['y'])
    defender_attacker_distance = computeDistance(defender_point, attacker_point)

    # Dynamic Defensive Radius Adjustment
    attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
    defensive_radius = max(base_defensive_radius, 0.6*attacker_position_distance)
    attacker['defensive_zone_radius'] = defensive_radius
    if defender_attacker_distance < 20*collision_distance:
        target_point = (attacker['x'], attacker['y'])
    else:
        position = defender ['position']
        attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)

        #compute delta_yaw based on position
        if position == 0:
            delta_yaw = 0
        elif position % 2 == 0:
            delta_yaw = (position -1)*(2*(1.0 - np.exp(-5*attacker_position_distance)))*np.pi/180.0
        else:
            delta_yaw = -position*(2*(1.0 - np.exp(-5*attacker_position_distance)))* np.pi/180.0

        attacker_position_angle += delta_yaw

    #Compute target point based on adjusted defensive_radius
    target_point = (defensive_radius*np.cos(attacker_position_angle),
defensive_radius*np.sin(attacker_position_angle))

elif strategy == 3:
    """Strategy 3: Defenders are positioned at a fixed radius around the attacker"""
    attacker_radius = attacker_parameters['attacker_radius'] #fixed radius
    attacker_point = (attacker['x'], attacker['y'])
    attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
    defensive_radius = max(attacker_radius, 0.3*attacker_position_distance)
    position = defender ['position']
    defender_point = (defender['x'], defender['y'])
    defender_attacker_distance = computeDistance(defender_point, attacker_point)
    collision_distance = defender_parameters['collision_distance']

    if defender_attacker_distance < 20*collision_distance:
        target_point = (attacker['x'], attacker['y'])
        attacker['defensive_radius'] = collision_distance
    else:
```

```
# compute angle for attacker placement
attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)

# Evenly distribute defenders around attacker_point
if position == 0:
    delta_yaw = 0
elif position % 2 == 0:
    delta_yaw = (position - 1) * (2 * np.pi / 180.0) #Fixed small angle
else:
    delta_yaw = -position * (2 * np.pi / 180.0) #Fixed small angle
attacker_position_angle += delta_yaw

#compute target point at a fixed distance from the attacker
target_point = (attacker_point[0] + defensive_radius * np.cos(attacker_position_angle + np.pi),
attacker_point[1] + defensive_radius * np.sin(attacker_position_angle + np.pi))
#Save attacker actual defensive radius to make its plot
attacker['defensive_radius'] = defensive_radius

elif strategy == 4:
    collision_distance = defender_parameters['collision_distance']
    base_defensive_radius = defender_parameters['defensive_radius']
    num_defenders_foreach_attacker = defender_parameters['num_defenders_foreach_attacker']
    attacker_point = (attacker['x'], attacker['y'])
    defender_point = (defender['x'], defender['y'])
    defender_attacker_distance = computeDistance (defender_point, attacker_point)
    position = defender['position']
    attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)
    attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
    if defender_attacker_distance < 20 * collision_distance:
        target_point = (attacker['x'], attacker['y'])
    else:
        if position > int(num_defenders_foreach_attacker / 2): #position % 2 == 0:
            target_point = (attacker['x'], attacker['y'])
        else:
            defensive_radius = base_defensive_radius
            if position == 0:
                delta_yaw = 0
            elif position % 2 == 0:
                delta_yaw = (position - 1) * (2 * np.pi / 180.0) #Fixed small angle
            else:
                delta_yaw = -position * (2 * np.pi / 180.0) #Fixed small angle
            attacker_position_angle += delta_yaw
            target_point = (defensive_radius * np.cos(attacker_position_angle),
defensive_radius * np.sin(attacker_position_angle))

elif strategy == 5:
    collision_distance = defender_parameters['collision_distance']
    base_defensive_radius = defender_parameters['defensive_radius']
    num_defenders_foreach_attacker = defender_parameters['num_defenders_foreach_attacker']
    attacker_point = (attacker['x'], attacker['y'])
```

```

defender_point = (defender['x'], defender['y'])
defender_attacker_distance = computeDistance (defender_point, attacker_point)
position = defender['position']
attacker_position_angle = computeAngle((0.0, 0.0), attacker_point)
attacker_position_distance = computeDistance((0.0, 0.0), attacker_point)
if defender_attacker_distance < 20*collision_distance:
    target_point = (attacker['x'], attacker['y'])
else:
    if position < int(num_defenders_foreach_attacker/2):
        defensive_radius = base_defensive_radius
    else:
        defensive_radius = max(1.3*base_defensive_radius, 0.6*attacker_position_distance)

    # Compute new target point based on the computed defensive_radius
    if position == 0 or position == int(num_defenders_foreach_attacker/2):
        delta_yaw = 0
    elif position % 2 == 0:
        delta_yaw = (position-1)*(2*np.pi/180.0) #Fixed small angle
    else:
        delta_yaw = -position*(2*np.pi/180.0) #Fixed small angle
    attacker_position_angle += delta_yaw
    target_point = (defensive_radius*np.cos(attacker_position_angle),
defensive_radius*np.sin(attacker_position_angle))

if target_point is None:
    #print('defensive_radius=', defensive_radius , 'attacker_position_angle=',attacker_position_angle, '
attacker=', attacker['x'] )
    target_point = (attacker['x'], attacker['y'])
return target_point

def plotSimulations (simulations_data, plot_name = None):
    #print(simulations_data)
    # simulations_data: List of tuples
    #(defenders_strategy,num_obstacles,num_attackers,num_defenders,simulation_steps,simulation_time,simulation_s
tate,total_cost,num_active_defenders,num_active_attackers)
    defenders_strategy,num_obstacles,num_attackers,num_defenders,simulation_steps,simulation_time,simulation_sta
te,total_cost,num_active_defenders,num_active_attackers = zip(*simulations_data)

    # Read data and create parameters
    strategies = list(set(defenders_strategy))
    num_simulations = len(defenders_strategy) / len(strategies)
    simulation_range = np.arange(1, num_simulations + 1)
    swarms_sizes = [int(num_def/num_att) for (num_att,num_def) in zip(num_attackers,num_defenders)]

    # Define strategy colors and labels
    strategy_colors = {0: 'blue', 1: 'green', 2: 'red', 3: 'purple', 4: 'black', 5: 'orange'}
    strategy_labels = {0: 'Strategy 0', 1: 'Strategy 1', 2: 'Strategy 2', 3: 'Strategy 3', 4: 'Strategy 4', 5:
'Strategy 5'}

```

```
# Iterate over the data and store the total cost per strategy
total_cost_per_strategy = {}
total_time_per_strategy = {}
win_counts = {}
win_counts_vs_swarm_size = {}
num_swarm_sizes = 0

for strategy, cost, sim_time, sim_state, swarm_size in zip(defenders_strategy, total_cost, simulation_time,
simulation_state, swarms_sizes):
    strategy = int(strategy)
    swarm_size = int(swarm_size)
    if not strategy in total_cost_per_strategy:
        total_cost_per_strategy[strategy] = {'attackers':[], 'defenders':[]}
    if sim_state == 1:
        total_cost_per_strategy[strategy]['attackers'].append(cost)
    elif sim_state == 2:
        total_cost_per_strategy[strategy]['defenders'].append(cost)

    if not strategy in total_time_per_strategy:
        total_time_per_strategy[strategy] = {'attackers':[], 'defenders':[]}
    if sim_state == 1:
        total_time_per_strategy[strategy]['attackers'].append(sim_time)
    elif sim_state == 2:
        total_time_per_strategy[strategy]['defenders'].append(sim_time)

    if not strategy in win_counts:
        win_counts[strategy] = {'attackers':0, 'defenders':0}
    if sim_state == 1:
        win_counts[strategy]['attackers'] += 1
    elif sim_state == 2:
        win_counts[strategy]['defenders'] += 1

    if not swarm_size in win_counts_vs_swarm_size:
        win_counts_vs_swarm_size[swarm_size] = {}
        num_swarm_sizes += 1
    if not strategy in win_counts_vs_swarm_size[swarm_size]:
        win_counts_vs_swarm_size[swarm_size][strategy] = {'attackers':0, 'defenders':0}
    if sim_state == 1:
        win_counts_vs_swarm_size[swarm_size][strategy]['attackers'] += 1
    elif sim_state == 2:
        win_counts_vs_swarm_size[swarm_size][strategy]['defenders'] += 1

# Contar victorias por atacante y defensor
attackers_wins = [win_counts[s]['attackers'] for s in strategies]
defenders_wins = [win_counts[s]['defenders'] for s in strategies]

# Crear gráfico de totales para ganadas por defensor o atacante
fig1, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(14,10))
x_labels = ['Attackers win', 'Defenders win']
x = range(len(x_labels))
width = 0.1 # Ancho de las barras
```

```

        for i, (att_wins, def_wins, color, label) in enumerate(zip(attackers_wins, defenders_wins,
strategy_colors.values(), strategy_labels.values())):
            ax1.bar(x[0] + i * width - 0.3, att_wins, width, label=label, color=color)
            ax1.bar(x[1] + i * width - 0.3, def_wins, width, color=color)
        ax1.set_xlabel('Result')
        ax1.set_ylabel('Number of simulations')
        ax1.set_title('Wins vs strategy')
        ax1.set_xticks(x)
        ax1.set_xticklabels(x_labels)
        ax1.grid()
        ax1.legend()
    if not plot_name is None:
        plt.savefig(plot_name + '_total_wins.png', dpi=100)

# Crear gráfico para ganadas por defensor o atacante en función del tamaño del swarm
fig3, ax3 = plt.subplots(nrows=int(num_swarm_sizes/2), ncols=2, figsize=(14,10))
x_labels = ['Attackers win', 'Defenders win']
x = range(len(x_labels))
width = 0.1 # Ancho de las barras
plot_row = 0
plot_col = 0
swarm_sizes = list(win_counts_vs_swarm_size.keys())
swarm_sizes.sort()
#print('swarm_sizes=', swarm_sizes)
for swarm_size in swarm_sizes:
    #print('swarm_size=', swarm_size, ' plot_row=', plot_row, ' num_swarm_sizes=', num_swarm_sizes)
    i = 0
    num_strategies_swarm_size = len(win_counts_vs_swarm_size[swarm_size].keys())
    if num_strategies_swarm_size > 0:
        for strategy in win_counts_vs_swarm_size[swarm_size].keys():
            color = strategy_colors[strategy]
            ax3[plot_row,plot_col].bar(x[0] + i * width - 0.3,
win_counts_vs_swarm_size[swarm_size][strategy]['attackers'], width, color=color)
            ax3[plot_row,plot_col].bar(x[1] + i * width - 0.3,
win_counts_vs_swarm_size[swarm_size][strategy]['defenders'], width, color=color)
            #ax3[plot_row,plot_col].set_xlabel('Result')
            ax3[plot_row,plot_col].set_ylabel(f'Swarm size = {swarm_size}')
            ax3[plot_row,plot_col].set_title('Wins vs strategy')
            ax3[plot_row,plot_col].set_xticks(x)
            ax3[plot_row,plot_col].set_xticklabels(x_labels)
            ax3[plot_row,plot_col].grid()
            #ax3[plot_row,plot_col].legend()
            i += 1
        plot_col += 1
        if plot_col > 1:
            plot_col = 0
            plot_row += 1
    #plt.tight_layout()
    if not plot_name is None:

```

```
plt.savefig(plot_name + '_total_wins_by_swarm_size.png', dpi=100)

# Crear gráficas de coste y tiempo de simulaciones
fig2, ax2 = plt.subplots(nrows=2, ncols=2, figsize=(14,10))

# Plot each strategy separately with total cost evolution over simulations
for strategy in strategies:
    sns.kdeplot(total_cost_per_strategy[strategy]['attackers'], color=strategy_colors[strategy],
label=strategy_labels[strategy], alpha=0.5, ax=ax2[0,0], common_norm=False)
    sns.kdeplot(total_cost_per_strategy[strategy]['defenders'], color=strategy_colors[strategy],
label=strategy_labels[strategy], alpha=0.5, ax=ax2[0,1], common_norm=False)
    sns.kdeplot(total_time_per_strategy[strategy]['attackers'], color=strategy_colors[strategy],
label=strategy_labels[strategy], alpha=0.5, ax=ax2[1,0], common_norm=False)
    sns.kdeplot(total_time_per_strategy[strategy]['defenders'], color=strategy_colors[strategy],
label=strategy_labels[strategy], alpha=0.5, ax=ax2[1,1], common_norm=False)

    #ax2[0].plot(simulation_range[:len(total_cost_per_strategy[strategy])], total_cost_per_strategy[strategy],
linestyle='-', color=strategy_colors[strategy], label=strategy_labels[strategy], linewidth=1)
    #ax2[1].plot(simulation_range[:len(total_time_per_strategy[strategy])], total_time_per_strategy[strategy],
linestyle='-', color=strategy_colors[strategy], label=strategy_labels[strategy], linewidth=1)

# Add labels and legends
ax2[0,0].set_xlabel('Total Cost')
ax2[0,0].set_ylabel('Density')
ax2[0,0].set_title(f'Total cost by strategy: {x_labels[0]}')
#ax2[0].set_xlim((1,num_simulations))
ax2[0,0].grid()
ax2[0,0].legend()
ax2[0,1].set_xlabel('Total Cost')
ax2[0,1].set_ylabel('Density')
ax2[0,1].set_title(f'Total cost by strategy: {x_labels[1]}')
#ax2[0].set_xlim((1,num_simulations))
ax2[0,1].grid()
ax2[0,1].legend()

ax2[1,0].set_xlabel('Simulation time ')
ax2[1,0].set_ylabel('Density')
ax2[1,0].set_title(f'Simulation time by strategy: {x_labels[0]}')
#ax2[1].set_xlim((1,num_simulations))
ax2[1,0].grid()
ax2[1,0].legend()
ax2[1,1].set_xlabel('Simulation time ')
ax2[1,1].set_ylabel('Density')
ax2[1,1].set_title(f'Simulation time by strategy: {x_labels[1]}')
#ax2[1].set_xlim((1,num_simulations))
ax2[1,1].grid()
ax2[1,1].legend()

plt.tight_layout()
```

```
if not plot_name is None:
    plt.savefig(plot_name + '_total_cost_and_time_by_strategy_and_win.png', dpi=100)

# Show plot
plt.show()
```