UNIVERSIDAD SANTO TOMÁS

Implementación de un Enjambre de Drones Crazyflie 2.1 Coordinado Mediante Q-learning y ROS

Realizado por

David Santiago González Álvarez

Grupo de Investigación GED (Grupo de Estudio y Desarrollo en Robótica)
Facultad de Ingeniería Electrónica
División de Ingenierías

2024

Implementación de un Enjambre de Drones Crazyflie 2.1 Coordinado Mediante Q-learning y ROS

Realizado por

David Santiago González Álvarez

Proyecto de grado enviado en cumplimiento del requisito parcial
para optar por el grado de Ingeniería Electrónica

Dirigido por

David Alejandro Martínez Vásquez Ph.D

Codirigido por

Sindy Paola Amaya M.Sc

Armando Mateus Rojas M.Sc

Grupo de Investigación GED (Grupo de Estudio y Desarrollo en Robótica)
Facultad de Ingeniería Electrónica
División de Ingenierías

2024

Autoridades de la Universidad

RECTOR GENERAL
R.P. FRAY ÁLVARO JOSÉ ARANGO RESTREPO, O.P.

VICERRECTOR ADMINISTRATIVO Y FINANCIERO GENERAL
R.P. FRAY HERNÁN YESID RIVERA ROBERTO, O.P.

VICERRECTOR ACADÉMICO GENERAL
R.P. FRAY MAURICIO ANTONIO CORTÉS GALLEGO , O.P.

SECRETARIA GENERAL
Dra. LUCERO GALVIS CANO.

DECANO DIVISIÓN DE INGENIERÍAS
R.P. FRAY JAVIER ANTONIO HINCAPIÉ ARDILA, O.P.

SECRETARIA DE DIVISIÓN
E.C. LUZ PATRICIA ROCHA CAICEDO.

DECANO FACULTAD DE INGENIERÍA ELECTRÓNICA
Ing. CARLOS ANDRÉS TORRES PINZÓN.

Advertencia

La Universidad Santo Tomás no se hace responsable de las opiniones y conceptos expresados en el trabajo de grado, solo velará por qué no se publique nada contrario al dogma ni a la moral católica y porque el trabajo no tenga ataques personales y únicamente se vea el anhelo de buscar la verdad científica.
Capítulo III - Art. 46 del Reglamento de la Universidad Santo Tomás.

Agradecimientos

Quisiera agradecer a mis familiares, por todo el apoyo y cariño que me han dado. A mis amigos por haberme animado y haber compartido muchos momentos especiales. A la universidad y a mis tutores por el conocimiento y procesos de aprendizaje que me permitieron transitar por diferentes campos de la Ing. Electrónica.

David Santiago González Álvarez

Resumen

La implementación de redes de cooperación con vehículos aéreos no tripulados, organizados mediante inteligencia artificial, tiene una aplicabilidad interesante y pertinente en la sociedad, ya que permite la solución de problemas de movilidad y organización en espacios extensos.

El presente proyecto genera una investigación teórica y aplicada, organizando una red de cooperación de drones en un escenario a pequeña escala, donde se puede implementar el dron Crazyflie 2.1 y su sistema de posicionamiento Lighthouse, para validar la pertinencia del uso de estos sistemas. Puntualmente se desarrolla un proceso de control por toma de decisiones, de cada partícula o agente que compone un enjambre de drones. El mismo se logra organizado en torno a un líder y los seguidores o compañeros. El algoritmo de aprendizaje aplicado es Q-learning, y se desarrolla todo el marco de ejecución en torno a que cada agente llegue al objetivo y evite chocar con sus compañeros.

Se encuentra que el proceso de aprendizaje mediante Q-learning puede darse con una discretización del espacio de estados muy bajo en comparación al respectivo espacio de estados del proceso de aplicación, lo que se traduce en velocidad en el aprendizaje y precisión al actuar en la implementación del enjambre.

Además de lo anteriormente expuesto se encuentra que el aprendizaje por fases permite un llenado acertado de la tabla de políticas Q que usa el agente para la toma de decisiones, junto con la correcta definición de todos los posibles estados, 256 para un ambiente sobre dos ejes de posición.

Palabras clave: Q-learning, Reinforcement learning, Particle Swarm Optimization y Enjambre de drones.

Abstract

The implementation of cooperation networks based on unmanned aereal vehicles, using artificial intelligence, have a social interesting applicability, because it allows to solve problems of mobility and organization on large areas of land.

The present project generates a theoretical and applied research by the organize a mini dron cooperation network based on Crazyflie 2.1 and his positional system calls Lighthouse, to validate the use of this systems before practical implementations. Specifically developing a decision control process for all the agents involved on the network. The learning algorithm is Q-learning, used to prevent crashes and organize the swarm.

In the learning process is possible to minimize the estates of spaces and maximize it when require the performance of the swarm, it able to do a process of learning fast and a precise performance on implementation.

Besides, develop a certain phases of training and test of the algorithm to generate a right Q policy table, that the agents use to take decisions, together with a 250 possible state definition, implemented for a two dimensional environment.

Keywords: Q-learning, Reinforcement learning, Particle Swarm Optimization and Drones swarm.

Índice de figuras

1. Proceso de decisiones de Markov, tomado de [25] 11
2. Pseudocódigo para la implementación de Q-learning, tomado de [27] 14
3. Aplicación Cfclient, usada para el ajuste de los drones. Imágen tomada en el momento en que se instaló en el dispositivo de desarrollo. 19
4. Estimador de posición para cada dron. Tomado de la librería del fabricante [33] . 20
5. Diferentes funciones para controlar el enjambre. Tomado de la librería del fabricante [33] 21
6. Protocolo para definir el enjambre y llamado de una función en paralelo. Tomado de la librería del fabricante [33] 21

7. Librerías implementadas en la asignación de ruta para un dron individual 27
8. Variables implementadas en la asignación de ruta para un dron individual 27
9. Programa principal, asignación de ruta para un dron individual 28
10. Hilo para asignación de coordenadas, asignación de ruta para un dron individual 28
11. Obtención de coordenadas en el loop principal 29
12. Método para impresión de coordenadas 29
13. Conformación de la tabla Q 31
14. Variables para implementar Q-learning 31
15. Generación de políticas respecto a los episodios 32
16. Variables para la revisión del aprendizaje 33
17. Programa principal, ejecución de los episodios para el entrenamiento. 34
18. Organización de los procesos para llevar a cabo Q-learning y ecuación en Python. 34
19. Función para la selección de la acción. 35
20. Función para ejecutar el movimiento del agente. 35
21. Definición del estado respecto al objetivo. 36
22. Definición del estado respecto al compañero más cercano. 36
23. Función Get_dist_angle 37
24. Función Identificar_comp 37
25. Función de recompensa 37
26. Ingreso al nodo, subscripción y emisión 38
27. Tipos de mensajes comunes 39
28. Comunicación con el usuario. 40
29. Verificación de los drones en la red 41
30. Selección de coordenadas efectuada por el líder. 42
31. Comunicación de la coordenada objetivo de los seguidores por parte del líder. . . 42

32. Recepción de datos del `topic` . . . 43
33. Recepción de datos del `topic` por parte del seguidor . . . 44
34. Librerías del script para conexión y asignación de coordenadas a los drones en vuelo. . . . 45
35. Creación del servidor y conexión a la red. . . . 46
36. Recepción de los datos del cliente, paralelo al programa principal. . . . 46
37. Variables globales para coordinar los drones mediante el servidor. . . . 47
38. Funciones complementarias para la asignación de coordenadas del enjambre . . 47
39. Programa principal del servidor, donde se asignan coordenadas al enjambre y se realiza la comunicación con el cliente. . . . 48
40. Variables globales y librerías implementadas en el cliente. . . . 49
41. Inicialización del puerto y conexión con el servidor. . . . 49
42. Recepción de los datos que envía el servidor. . . . 50
43. Programa principal del cliente. . . . 50
44. Revisión del proceso de obtención de las imágenes de las rutas de los drones. . . 52

45. Secuencia para estimación de movimientos . . . 53
46. Dron en vuelo . . . 54
47. Vuelo programado, perspectiva en 3D . . . 54
48. Movimiento en los ejes (x) y (y) . . . 55
49. Movimiento en los ejes (x) y (z) . . . 55
50. Resultados de la primera fase de entrenamiento de Q-learning, imágenes tomadas de los comentarios en consola producidos por el código desarrollado . . . 57
51. Episodio 979 a 999, parte final de la segunda fase . . . 58
52. Episodio 1479 a 1499 . . . 58
53. Resultados gráficos del entrenamiento del algoritmo. . . . 59
54. Red generada en ROS para cuatro drones, tomada del desarrollo presentado, obtenida mediante el comando en terminal: (rosrun rqt_graph rqt_graph). . . . 61
55. Mensajes publicados a la red de ROS . . . 61
56. Comportamiento del conjunto de drones en el ambiente virtual . . . 62
57. Cuadrantes de coordenadas . . . 63
58. Primer escenario real . . . 64
59. Escenario 2 real . . . 65
60. Escenario 3 real . . . 66
61. Zona de inestabilidad . . . 66
62. Escenario 4 real . . . 68
63. Escenario 5 real . . . 70
64. Espacio escenario 6 . . . 71
65. Sexto escenario real, primera ruta. . . . 72
66. Sexto escenario real, segunda ruta. . . . 72
67. Sexto escenario real, tercera ruta. . . . 73

Índice de cuadros

1. Tasas de ajuste para Q-learning de referencia 13
2. Bits para cada estado, autoría propia 31
3. Revisión de tres estados diferentes entre los 256 posibles, en la tabla Q generada, luego del entrenamiento y prueba. 60
4. Coordenadas iniciales, primer escenario real 63
5. Coordenadas iniciales, segundo escenario real. 64
6. Coordenadas iniciales, tercer escenario real. 65
7. Coordenadas iniciales, cuarto escenario real. 67
8. Coordenadas objetivo, cuarto escenario real. 68
9. Coordenadas objetivo, quinto escenario real. 69
10. Coordenadas objetivo, sexto escenario real. 72

Índice general

Índice de figuras VIII

Índice de cuadros X

1. Introducción **1**
1.1. Planteamiento del problema . . . 2
1.2. Pregunta problema . . . 2
1.3. Objetivos . . . 3
1.3.1. Objetivo general . . . 3
1.3.2. Objetivos específicos . . . 3
1.4. Alcance . . . 3
1.5. Justificación . . . 4
1.6. Impacto social . . . 5

2. Estado del arte **6**
2.1. Q-learning . . . 6
2.2. Comunicación entre los agentes con ROS . . . 7
2.3. Obtención de coordenadas del dron . . . 8
2.4. Control asociado al posicionamiento . . . 8

3. Marco Teórico **10**
3.1. Conceptos previos . . . 10
3.2. Q-learning . . . 11
3.2.1. Aprendizaje por refuerzo . . . 11
3.2.2. Políticas para llevar a cabo episodios . . . 12
3.2.3. Entornos dinámicos y parámetros de aprendizaje implementando Q-learning 12
3.2.4. Entornos dinámicos y recompensas en Q-learning . . . 13
3.2.5. Pasos para ejecutar Q-learning . . . 13
3.2.6. Fórmula y parametrización . . . 15
3.3. ROS . . . 15
3.3.1. Nodo en ROS . . . 15
3.3.2. Topic en ROS . . . 17
3.3.3. Implementación de Catkin como espacio de trabajo para el desarrollo . . . 17
3.4. Crazyflie 2.1 y Cflib . . . 17

3.4.1. Conexión del dron . . . 18
3.4.2. Asignación de movimientos . . . 18
3.4.3. Recepción de coordenadas . . . 18
3.4.4. Swarm . . . 19
3.5. Comunicación por sockets mediante Python . . . 21
3.5.1. Servidor . . . 22
3.5.2. Cliente . . . 22
3.5.3. Funciones en común . . . 22

4. Metodología **24**

5. Desarrollo Conceptual **26**
5.1. Asignación de comandos y recepción de coordenadas del dron . . . 26
5.2. Implementación de Q-learning, entrenamiento y convergencia . . . 30
5.2.1. Conformación de la tabla Q . . . 30
5.2.2. Aprendizaje y generación de políticas . . . 31
5.2.3. Código de Q-learning . . . 34
5.3. Roles del enjambre e implementación en código . . . 38
5.3.1. Roles del líder . . . 40
5.3.2. Roles de los seguidores . . . 43
5.4. Implementación del conjunto de drones con el modelo obtenido . . . 44
5.4.1. Comunicación y sus funciones de los dispositivos de comunicación entre ROS y el enjambre . . . 44
5.4.1.1. Servidor . . . 45
5.4.1.2. Cliente . . . 48
5.4.1.3. Consideraciones respecto a los nodos . . . 51
5.4.2. Obtención de las rutas reales . . . 51

6. Resultados y Discusión **53**
6.1. Asignación de movimientos y estimación de vuelo . . . 53
6.2. Convergencia del modelo entrenado . . . 56
6.2.1. Primera fase . . . 56
6.2.2. Segunda fase . . . 57
6.2.3. Tercera fase . . . 58
6.2.4. Tabla Q . . . 59
6.3. Interacción de diferentes agentes virtuales con el modelo . . . 60
6.4. Vuelo en conjunto y revisión de las rutas individuales . . . 62
6.4.1. Escenario 1 . . . 63
6.4.2. Escenarios 2 y 3 . . . 64
6.4.2.1. Escenario 2 . . . 64
6.4.2.2. Escenario 3 . . . 65
6.4.2.3. Zona de inestabilidad . . . 66
6.4.3. Escenario 4 . . . 67
6.4.4. Escenario 5 . . . 69

6.4.5. Escenario 6 71

7. Conclusiones y Trabajos Futuros **74**
7.1. El sistema de posicionamiento para uso académico 74
7.2. El desempeño de Q-learning 75
7.3. ROS y el modelo de comunicación 75
7.4. Validación experimental 76

Bibliografía **77**

Capítulo 1

Introducción

El desarrollo de la robótica en el siglo XXI ha permitido la aplicación de diferentes tipos de tecnologías para la solución de tareas peligrosas o repetitivas para los seres humanos. En torno a esto, la implementación de vehículos autónomos, específicamente aéreos, permite la implementación de vehículos de tamaño reducido, al no necesitar adecuarse para transportar a una persona. Además, los mismos pueden moverse en diferentes posiciones a velocidades óptimas y se revisa la característica de que los riesgos de implementación en espacios controlados no suponen pérdidas humanas.

Aunado a esto, gracias a todas estas capacidades, los drones pueden ser implementados en redes de cooperación, tales como sistemas de entrega de paquetería o irrigación de cultivos. El proyecto de grado desarrollado genera una solución a la organización e implementación de una red de cooperación de drones tipo enjambre, mediante el algoritmo de aprendizaje por refuerzo Q-learning.

Este proyecto se desarrolla en los espacios de la Universidad Santo Tomás, destinados al grupo de investigación GED, cuyas siglas significan: grupo de estudio y desarrollo, que tiene como énfasis la robótica y los campos de estudio relacionados. La inteligencia artificial, el control de sistemas electromecánicos, las interfaces entre humano y máquina, prototipado y desarrollo de soluciones electrónicas son ejemplos de lo que se estudia en el grupo y algunos hacen parte del desarrollo de la presente investigación.

1.1. Planteamiento del problema

La investigación asociada a los sistemas multiagente demanda herramientas relacionadas con la implementación en sistemas reales, ya que, debido a los costos de infraestructura, estos han sido generalmente elaborados en entornos de simulación, haciendo que la validación experimental sea escasa.

Una alternativa es la implementación de sistemas a escala, a un costo reducido, de fácil acceso, comprensión y sin requerimiento de permisos estatales [1], además de servir como plataformas experimentales para las posteriores aplicaciones reales como las revisadas en [2], donde se plantea un enjambre capaz de evadir obstáculos, o como en [3], en donde se propone un sistema descentralizado capaz de organizarse en posiciones geométricas variadas.

Al igual que los desarrollos citados anteriormente, se han planteado proyectos de grado ambiciosos donde el alcance no permite llegar a la implementación física a pesar de ser un objetivo tácito, como sucede en [4], donde se realiza un ejercicio investigativo fructífero y pertinente, pero se evidencia una necesidad de dar solución a más retos que proponen estos desarrollos, por lo interesante y apasionante que llega a tornarse este campo de la robótica.

Aunado a esto, se tiene un gran interés por el uso de inteligencia artificial aplicado a sistemas con recursos limitados como memoria o velocidad de procesamiento [5], en los cuales algoritmos tales como Q-learning permiten la sintonización efectiva del control por toma de decisiones con eficiencia aceptable.

De acuerdo con las razones descritas anteriormente, surge la necesidad de implementar algoritmos de inteligencia artificial en dispositivos reales, de tal manera que sea posible evaluar su rendimiento y operación fuera de un ambiente de simulación.

1.2. Pregunta problema

En este sentido, se plantea una pregunta problema que sintetiza todas las necesidades encontradas. El desarrollo propuesto en este proyecto de grado pretende responder a ¿Cómo desarrollar un sistema multiagente e implementarlo físicamente mediante un enjambre de drones?

1.3. Objetivos

1.3.1. Objetivo general

Desarrollar un sistema multiagente PSO, coordinando cada agente con Q-learning y utilizando ROS como medio de comunicación para su implementación física y posterior validación experimental.

1.3.2. Objetivos específicos

- Integrar un sistema de posicionamiento que estime las coordenadas de cada dron.
- Sintonizar el algoritmo Q-learning para el movimiento de los drones.
- Diseñar un modelo jerárquico líder-seguidor que permita a los agentes comunicarse dentro de su dinámica de movimiento.
- Validar el sistema de interacción entre los agentes mediante ROS y el hardware especializado.

1.4. Alcance

La aplicación concisa del proyecto de grado es implementar un sistema de agentes controlados individualmente usando el algoritmo Q-learning siguiendo un esquema líder-seguidor o compañero. En la Universidad se dispone de un conjunto de cuatro drones Crazyflie 2.1 y un sistema de posicionamiento Lighthouse de Bitcraze. Debido a que Bitcraze ha desarrollado sus librerías para Python, es necesario usar este lenguaje para el manejo de los drones. Además, la implementación de nodos en ROS también usa este lenguaje de programación, por lo tanto, este será el único utilizado en el desarrollo. Es importante recalcar que el sistema puede escalar a un número mucho mayor de agentes físicos por la naturaleza de la disposición jerárquica y de comunicación planeada, sin embargo, debido a las limitaciones espaciales del sistema de posicionamiento, es posible implementar satisfactoriamente el comportamiento con la cantidad de drones mencionada, lo dicho se toma referente a [6].

1.5. Justificación

Las áreas de robótica e inteligencia artificial han cobrado gran interés para la ingeniería en los últimos años, ya que permiten aplicar diseños de programación, estructuras físicas basadas en mecánica, e inclusive discusiones filosóficas respecto al uso y simplificación del trabajo humano en torno a la dignificación de este, como se revisa en [7]. Se hace necesario implementar de forma física los desarrollos en inteligencia artificial concretamente, ya que es un campo de desarrollo novedoso, que requiere validación experimental y para lo cual existe un gran interés por parte de los desarrolladores. Parte importante de este ejercicio gira en torno al control por toma de decisiones de múltiples agentes, cuyas interacciones comúnmente se basan en comportamientos de enjambre, los cuales pueden ser implementados en diferentes aplicaciones, se destaca su uso en la agricultura de presición, revisado en [8] donde se usa un sistema multiagente de drones para el sondeo de terrenos. Por otro lado también en la organización de agentes que permite el posicionamiento más estable para el envío de información entre redes de sensores móviles, evitando redundancia y optimizando el uso del espacio [9].

En este documento, se presenta un desarrollo enfocado en cumplir este propósito, generando una ruta que inicia con la elección de una plataforma UAV de código abierto tal como Crazyflie 2.1 que permita ser controlada de forma remota. También se plantea revisar la mejor disposición geométrica del sistema de posicionamiento de tal manera que se eviten problemas de implementación en términos de colisión de agentes (drones), se identifica Lighthouse de Bitcraze como el sistema de identificación de coordenadas adecuado y se revisa [3] para tener un marco de referencia. El desarrollo producto de la investigación se plantea como base para la implementación de diferentes algoritmos de inteligencia artificial sobre sistemas multiagentes, que el lector interesado pueda utilizar como base para sus aplicaciones, esto especialmente en el apartado de hardware. En este caso se plantea un comportamiento de rebaño o seguidor-líder basado en el modelo de aprendizaje Q-learning implementado en lenguaje Python.

En [7] se identifica a la robótica interactiva como una expresión social necesaria, ya que incentiva el desarrollo de nuevas herramientas aplicables en la sociedad. El desarrollo generado en este trabajo es el punto de partida para la implementación a gran escala de sistemas multiagente que solucionen problemas de aplicaciones en la agricultura [8], la industria de manufactura o sistemas de búsqueda y rescate, entre otros. En ese sentido, se espera que los ejercicios investigativos futuros, especialmente los desarrollados dentro de la facultad de Ingeniería Electrónica de la Universidad Santo Tomás, puedan tomar el presente trabajo como referencia. No obstante, también se revisan diferentes aplicaciones en seguridad [10], o en la agricultura [11], donde

los enjambres de drones pueden ser una solución efectiva, y para ello se requiere el desarrollo e investigación para su ejecución.

1.6. Impacto social

Dentro de los objetivos para el desarrollo sostenible (ODS) creados por la ONU y revisados en [12], se identifica la necesidad de avanzar en la innovación e infraestructura de ciudades y comunidades inteligentes, lo cual abre posibilidades para la implementación de sistemas multiagentes a pequeña escala que funcionan como punto de partida para los desarrollos con sistemas UAV.

La población objetivo del ejercicio investigativo presentado son los desarrolladores que requieran validar sus sistemas multiagentes sobre plataformas móviles. Se presenta una ruta de desarrollo completa que puede permitir a la población objetivo partir desde un hardware concreto y fácil de apropiar, referente a la interconectividad entre los agentes y los dispositivos de programación, para que puedan proceder a la parte de implementación de hardware de forma rápida y puedan enfocarse en sus propios desarrollos, en [6] se revisa el cómo los sistemas de Crazyflie permiten entornos de realidad mixta (referente a simulación e implementación física) para la experimentación.

Concretamente, se enfatiza la utilidad para los grupos de robótica en Latinoamérica y países hispanohablantes en general como lo refiere [13], ya que las herramientas facilitadas son presentadas en este documento con el propósito de ser breves, concisas y pertinentes en un lenguaje natural y entendible. Con respecto a la aplicabilidad de la robótica para la enseñanza, [14] presenta el valor que tiene el aprendizaje mediante herramientas robóticas en instituciones educativas de Bogotá donde se pueden aplicar proyectos como el presentado en este documento. por otro lado, se reconoce la necesidad de implementar enjambres de robots en la facultad de ingeniería electrónica de la Universidad Santo Tomás con el fin de actualizar e indagar sobre nuevas tecnologías que entusiasmen a los estudiantes a continuar la investigación sobre inteligencia artificial.

Capítulo 2

Estado del arte

El desarrollo de tecnologías en torno a sistemas multiagentes ha tendido a integrar diferentes tipos de UAV (vehículos aéreos no tripulados) de formas variadas. Con el fin de tener una ruta de trabajo se plantea la búsqueda de documentación, partiendo desde el modelo de interacción del enjambre, pasando por los aspectos de comunicación, hasta el apartado técnico en torno a la elección de un tipo de UAV con el cual sea posible implementar el proyecto. Por lo tanto, los temas a tratar relevantes y sus fuentes asociadas se revisan a continuación:

2.1. Q-learning

El desarrollo de algoritmos para el aprendizaje por refuerzo, puntualmente aplicado a dispositivos limitados en términos de procesamiento y almacenamiento de datos, es tratado desde el modelo Q-learning. En [15] se revisa la convergencia de los modelos de dicho algoritmo, presentando el problema, el método y la notación matemática sobre la cual se sientan las bases del desarrollo propuesto. De forma resumida, se trata de un algoritmo para el cual se revisan acciones en estados particulares y se evalúan los resultados en torno a la satisfacción del proceso, estimando el valor del estado tomado. El documento aporta contextualización sobre Q-learning al trabajo propuesto.

Q-learning ha sido extensamente implementado, y se han desarrollado diferentes aplicaciones en sistemas multiagentes. En [16], se presenta un compendio de los más relevantes en la actualidad, en donde se revisa el tipo de interacción adecuada para el propósito de la investigación desarrollada en este proyecto y se identifica PSO (Particle Swarm Optimization) como tal. Dicha interacción permite crear un sistema de envío y recepción de información en el que interviene

cada agente (partícula), permitiendo que la ejecución de una tarea común sea dividida en tareas más sencillas, PSO es el esquema funcional elegido en el desarrollo propuesto.

En [17], se revisa un sistema de enjambre con un comportamiento de rebaño. La base de esta investigación gira en torno a PSO, asignando roles a los agentes del enjambre que se desarrolla en un sistema de drones donde el control revisa el posicionamiento y sobre éste se desarrolla una función de recompensa con el fin de que cada individuo sea capaz de acercarse a un líder, evitando colisiones con él y los diferentes agentes seguidores. En el documento se encuentran las fórmulas para calcular el valor de recompensa en torno a la distancia. El aporte de esta referencia es sobre la estructuración del enjambre en torno a Q-learning.

2.2. Comunicación entre los agentes con ROS

Para la implementación del enjambre propuesto es necesario permitir un medio común para la recepción y emisión de la posición de los agentes. En el campo de los sistemas operativos, ROS es una herramienta adecuada. En [18], se encuentran los pasos a seguir para poder implementar ROS sobre Ubuntu, y crear programas de enlace con base a la asignación de nodos, los cuales permiten mediante un canal centralizado, conectar los agentes y compartir paquetes de datos. Esta referencia sienta las bases del trabajo con nodos a implementar en el desarrollo del proyecto consignado en este documento.

El orden de los nodos en ROS para un sistema descentralizado con comportamiento de rebaño se encuentra consignado en [19], en donde se plantea una red de seis agentes simulados en la cual resalta en su estructuración, ya que, para todos los agentes el primer nodo es un control de posición que evita cambios bruscos e inestabilidades en el movimiento individual, luego hay un nodo publicador de la posición y una suscripción a todos los demás, finalmente, todas las posiciones son compartidas a un tercer suscriptor que se encarga de mostrar en pantalla la posición de cada uno de los drones. Aunado a esto, todo el desarrollo es posible implementando el dron Crazyflie de Bitcraze en simulación, usando las herramientas de software que el fabricante ofrece en la red. De este documento se apropia la asignación de las funciones a los nodos para la investigación propuesta.

2.3. Obtención de coordenadas del dron

Dando continuidad a la ruta de revisión bibliográfica, se halla que en [20] el fabricante del dron mencionado desarrolla la librería cflib para el lenguaje de programación Python, en la cual se encuentran comandos de desplazamiento y recepción de las coordenadas, además de los drivers para la comunicación con el dron. Con el fin de localizar a los drones en un espacio de coordenadas, Bitcraze desarrolló la infraestructura de Lighthouse Position System, la cual permite estimar las coordenadas mediante mapeo en diferentes direcciones del espacio mediante haces de luz infrarroja, esta referencia brinda las funciones dadas por Bitcraze que se implementarán. En [21] se encuentra información detallada sobre la correcta implementación de este sistema, entregando medidas espaciales y rangos de error en el comportamiento del dron que es de interés para calcular los rangos de distancia prudente entre cada agente. Esta referencia permite un acercamiento al comportamiento del sistema que se implementará en este proyecto de grado para identificar las coordenadas de los agentes.

2.4. Control asociado al posicionamiento

En [22], se aborda el problema de control de posición en coordenadas tridimensionales planteando la implementación de un PID, presentando el modelo del dron desde la perspectiva de vuelo, que se compone de los sensores de giroscopio y acelerómetro, además de integrar la revisión externa, es decir, desde el mencionado Lighthouse Position System, desde esta fuente se revisa la precisión con la que el dron a implementar cuenta a la hora de la asignación de movimientos con el fin de justificar su uso. En [4], se expone la necesidad de integrar un control de estabilización y de posición/velocidad, además de brindar una explicación en torno a la teoría de sistemas sobre el filtro de Kalman extendido que integra el Firmware del dron Crazyflie y su filtro complementario, se apropia en la investigación propuesta para entender el control de posición que tiene el dron seleccionado. Por otro lado, en torno a la organización de difrentes agentes, [23] presenta una forma de ajuste de estados y optimización de los recursos entorno a la teoría de juegos, implementada en sistemas multiagente. Específicamente en torno a el equilibrio de Nash que se basa a grandes rasgos en que la estrategia de cada agente permite que el conjunto se vea beneficiado, logrando un máximo aprovachamiento de los recursos. Aunado a esto se revisa que en [9] se dispone de procesos de regresión gaussiana para la organización espacial de agentes en un ambiente donde es requerido el aprovechamiento del espacio y el control del flujo de información entre ellos. Resulta interesante debido a que muestra que los

ajentes son capaces de comprender el comportamiento de su entorno, teniendo la capacidad de exploración gracias a la consideración del conjunto de posiciones posibles para avanzar.

Capítulo 3

Marco Teórico

3.1. Conceptos previos

Para la ejecución de un proceso en el cual se requiere inteligencia artificial, el objeto que realiza la toma de decisiones es denominado agente. Dicho agente toma decisiones sobre su entorno, el cual es discretizado, teniendo como resultado una clasificación de los diferentes estados posibles donde el agente puede estar (espacio de estados).

El objetivo de aplicar inteligencia artificial para búsqueda y toma de decisiones es permitir que el agente transite los estados posibles, con las acciones acertadas, que le permitan llegar al objetivo. Para ello, la elección del algoritmo de búsqueda debe girar en torno al tipo de objetivos y escenarios, como se revisa en [24].

El tipo de búsqueda para este desarrollo es informada (se conocen la posición y distancia de los objetos del entorno), para un agente simple y en un entorno dinámico. Por la cantidad de posibles estados se considera la implementación de un algoritmo de exploración en entornos dinámicos (se descartan algoritmos de árboles de búsqueda por la complejidad del espacio de estados), para cada agente particular (se descartan algoritmos multiagentes), que permita ser entrenado y además tenga un gasto computacional bajo, ya que se requerirán varios agentes ejecutándose al tiempo (se descartan las redes neuronales).

En este punto, al descartar, la mejor herramienta para lo que se requiere implementar es el aprendizaje por refuerzo, ya que permite la mejor toma de decisiones en el estado en que se encuentra el agente, independientemente de que su entorno cambie. Q-learning es un algoritmo práctico el cual se ha implementado en entornos dinámicos, bajo condiciones especiales y

aunado a todo esto, su complejidad computacional es baja. Este algoritmo se revisa a continuación.

3.2. Q-learning

3.2.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo se trata de permitir al agente tomar decisiones secuenciales, donde los resultados en parte son por explotación o con el fin de explorar. El marco matemático definido para el proceso descrito anteriormente se conoce como MDP (proceso de decisiones de Markov). [25] Refiere que en este proceso se consideran pasos de la máquina (n o t), en cada uno de estos pasos el agente revisa su estado (s) en el espacio, basado en esto se produce una reacción que se conoce como acción (a) , y a cada nuevo par estado-acción es asignada una recompensa(r).

FIGURA 1: Proceso de decisiones de Markov, tomado de [25]

Con la finalidad de maximizar la recompensa acumulada, el rendimiento descontado esperado del agente viene representado por la ecuación:

$$y = (\sum_{k=0}^{\text{ínf}} (\gamma^k) R_t + k + 1. \tag{3.1}$$

Donde, (Rt) representa la recompensa, (k) es el paso de ejecución y, la tasa de descuento (γ). A grandes rasgos (γ) representa el peso o importancia de las recompensas actuales con respecto al tiempo, entre mayor sea su valor, más peso tendrán las recompensas futuras en un rango entre cero y uno, es decir, que el mismo es un factor clave para la convergencia del modelo con respecto al tiempo. En el posterior análisis de Q-learning se complementará su utilidad.

3.2.2. Políticas para llevar a cabo episodios

De [26] se resalta el manejo de escenarios o episodios, en dichos episodios se toma una cantidad de pasos posibles que dará el agente tanto en su dinámica de entrenamiento como de implementación. Una política (p) en términos de MDP, según la revisión de [26] , es el marco de referencia o normativas sobre el cual el agente aprende cómo ejecutar la función esperada en un episodio. El ideal es que se encuentre una política lo más cercana al valor óptimo denotado como (*), que es el valor de política para un episodio perfectamente ejecutado (se llega al objetivo en el menor tiempo posible):

$$q^*(s,a) = max_p q_p(s,a). \tag{3.2}$$

Este marco de referencia depende del aprendizaje efectuado en el entrenamiento del agente, por lo tanto, en el mismo es necesario desechar los valores no optimizados conseguidos para cada conjunto de estado-acción.

En Q-learning se tiene la tabla Q, la cual representa los diferentes pares estado-acción. Los valores de dicha tabla son las políticas que el agente sigue en la ecuación del algoritmo.

3.2.3. Entornos dinámicos y parámetros de aprendizaje implementando Q-learning

Como se revisa en [27], Q-learning generalmente se implementa en escenarios estáticos, por lo que es necesario considerar el tratamiento de los parámetros que modifican las políticas, con el fin de que pueda haber un comportamiento adaptativo a entornos dinámicos.
En este punto se consideran las tasas de aprendizaje (α), de descuento (γ) y de exploración (ϵ), el cambio de estos valores permite el correcto llenado de la tabla Q. Las tasas de aprendizaje y descuento se refieren a la importancia que tienen las políticas aprendidas con relación a las recompensas adquiridas en el tiempo, y la tasa de exploración, la frecuencia con la que el agente toma acciones aleatorias, contra un proceso de explotación.
En [27] se recomiendan valores altos de estos parámetros en el entrenamiento previo a la implementación en un entorno estático. Es importante tener en cuenta que los valores son tratados como porcentajes, y que además no existe una fórmula concreta para el cálculo de estos, por lo que cualquier proceso de optimización puede ser implementado y debe ajustarse a las necesidades de ejecución.

Hyperparameter	Default Value
Learning Rate	0.90
Gamma	0.81
Epsilon	0.70

CUADRO 1: Tasas de ajuste para Q-learning de referencia

En la parte de discusión sobre implementación en entornos dinámicos presentada en [27] se toma que, para los mismos, es necesario mantener valores de aprendizaje y descuento altos debido a que el agente requiere actualizar la tabla en torno al cambio del ambiente. Además, siendo necesario optimizar los parámetros para ajustarse a la ejecución en entornos dinámicos, eventualmente partiendo del ajuste en la implementación en un entorno estático. Esta podría considerarse la primera fase de implementación de Q-learning, donde el agente explora y almacena en su tabla Q todos los valores posibles (se espera que el agente falle mientras es entrenado).

En [28] se remarca que entre mayor experiencia (paso de episodios) se debe producir el decaimiento de las tasas referidas (exploración, descuento y aprendizaje) con el fin de afianzar las políticas adquiridas, en un proceso denominado convergencia del modelo. Esto vendría refiriéndose a una fase próxima a la definida anteriormente, previa a la convergencia del modelo y posterior validación.

3.2.4. Entornos dinámicos y recompensas en Q-learning

Las recompensas en Q learning son la forma en que el programador entrena el algoritmo. Con base a los valores obtenidos se actualizan las políticas en la tabla Q. Las recompensas son asignadas a la elección que tiene el algoritmo, si es acertada se dará un valor positivo, si no lo es, como se revisa en [27], puede ser negativo o cero.

Para entornos dinámicos es elemental considerar una definición correcta de los estados del agente, mediante una discretización propicia de los sensores que componen el espectro de entradas, y la consideración de una recompensa asociada al estado y al movimiento entre ellos, como lo denota [29].

3.2.5. Pasos para ejecutar Q-learning

El algoritmo de Q-learning tiene pasos específicos que permiten el correcto llenado de la tabla. De [27] se adopta un pseudocódigo pertinente consignado en la figura 2.

Line No.	Activity
1	For 1 to n agents, do
2	Define a new agent 'a'
3	Set parameters for a: learning rate, gamma, and epsilon
4	Initialize Q Table of n states x actions
5	For episode 1 to n do
6	While t < max steps do
7	Reset the environment
8	If random(0,1) < epsilon then
9	Action= sample action from train env
10	Else
11	Action = argmax(Q[state,:])
12	End if
13	Take the chosen action and collect the reward
14	Calculate the new Q-value
15	If a new state is a goal state, then
16	Increase the number of successful episodes
17	Break
18	If a new state is a hole state, then
19	Break
20	End while
21	Check if action is optimal
22	Update success ratio and optimal actions
23	End for
24	End for

FIGURA 2: Pseudocódigo para la implementación de Q-learning, tomado de [27]

El pseudocódigo revisado es útil para cualquier tipo de implementación si se obvia la línea siete del código respecto a reiniciar el ambiente. Sea entrenamiento o pruebas de ejecución, en un principio se definen los parámetros de aprendizaje, luego en un bucle que se termina con la cantidad de episodios: primero, entre la línea ocho y once, se tiene la elección de acciones, con enfoque a explotación contra exploración en dependencia de épsilon. Seguido se realiza la elección de la recompensa obtenida y se llena la tabla con la ecuación de Q-learning. Finalmente, si el agente llega a la meta, se termina el episodio.

3.2.6. Fórmula y parametrización

Para finalizar, se presenta la ecuación de Q-learning [27] que permite la definición de las políticas mediante los anteriores ítems mencionados:

$$Q(s_t, a_t) < -Q(s_t, a_t) + \alpha[R_(t+1) + \gamma * max_a Q(s_(t+1), a) - Q(s_t, a_t)] \tag{3.3}$$

La ecuación permite actualizar cada política en dependencia al par (estado, acción), y considera el estado actual más el factor de aprendizaje, multiplicado por: la recompensa actual y el factor de descuento, que a su vez se multiplica por la mejor opción para ese estado, menos el valor del estado actual. Es importante recalcar que el valor de Q(estado, acción) es puntualmente un número, no una matriz ni un vector.

3.3. ROS

En [30] se evidencia el uso de ROS como un puente de comunicación entre diferentes tecnologías, dispositivos e interfaces para el desarrollo de la robótica. Es posible conectar software de simulación, añadir a la Red los códigos asociados a los agentes involucrados y ejecutar sus instrucciones desde Python, como se revisa en el apartado de implementación de los drones. Particularmente es requerida su utilización para la organización de los agentes mediante un ordenador y la comunicación entre ellos.

En [31] se especifica que ROS está desarrollado para ser ejecutado desde un inicio sobre la distribución Ubuntu de Linux, siendo este sistema operativo el adecuado para el desarrollo. Es importante considerar diferentes conceptos y códigos que serán mostrados a continuación:

3.3.1. Nodo en ROS

En [32] se define que los nodos son ejecutables que pueden ser comunicados con otros procesos mediante topics (que es el nombre dado por los desarrolladores). A la hora de ejecutar los nodos, estos pueden ser implementados con las librerías referentes al lenguaje: `rospy` o `roscpp`. Los nodos pueden ser accedidos mediante la terminal de Ubuntu, con el comando `rosnode`, seguido de la instrucción a requerir como: `rosnode list` que permite ver qué nodos están siendo ejecutados y `rosnode kill node` , para poder finalizar los nodos que se quedan en bucles por fallos de implementación de los scripts.

Es importante tener en cuenta que un nodo tiene un nombre definido y un ID que va a ser utilizado para la comunicación mediante los topics.

3.3.2. Topic en ROS

Los topics son los puentes de comunicación entre nodos, permiten al nodo subscribirse a la información que se encuentra en el topic o ser emisor sobre el mismo, siempre y cuando se maneje el mismo tipo de mensaje entre los nodos y el topic. Los topics pueden ser transmitidos mediante TCP/IP y UDP, puntualmente se implementa una conexión persistente de TCP/IP conocida como TCPROS, esto significa que mediante protocolos de comunicación es posible conectar diferentes dispositivos, tanto como subprocesos en una sola máquina, según [31]. Con la finalidad de obtener información se implementa en la terminal de Ubuntu el comando `rostopic` seguido de la instrucción a utilizar, tal como `rostopic echo\NombreDelTopic` para conocer qué mensajes fueron enviados al topic.

3.3.3. Implementación de Catkin como espacio de trabajo para el desarrollo

El sistema de compilación oficial de ROS es Catkin [32], el mismo permite la implementación de comandos para la manipulación de la información implementada en los proyectos de ROS, siendo parte crucial el manejo como entorno de trabajo.

En el folder catkin `workspace` es donde serán desarrollados todos los proyectos a implementar sobre ROS, es decir, allí serán consignados los `paquetes`. Cuando se desea crear un paquete se implementa la instrucción en la términal: `catkin_create_pkg <package_name> [depend1] [depend2] [depend3]` donde las dependencias son las librerías tales como `rospy` o `stdlib`. Para poder actualizar el sistema de archivos luego de haber modificado los scripts necesarios, se implementa el comando `catkin_make`. Con el fin de correr los nodos es necesario inicializar el nodo maestro `roscore` en una terminal, de forma siguiente, sobre otra terminal se corre el paquete que se desee implementar mediante el comando `roslaunch [package] [filename.launch]`. En el archivo con extensión `.launch` se consignan todos los scripts que serán ejecutados al mismo tiempo.

3.4. Crazyflie 2.1 y Cflib

La librería cflib [33] fue creada en 2006, con el fin de poder conectar el dron con un dispositivo remoto, el cual tuviera la posibilidad de ejecutar scripts en Python. La misma no solo incluye el cómo ejecutar la comunicación con el dron para la asignación de coordenadas, sino que permite

recibir la posición actual, la posición asignada y datos de telemetría entre otros. A continuación, se revisan los elementos necesarios para la implementación:

3.4.1. Conexión del dron

En este apartado se revisan las librerías, métodos y variables que permiten este fin. Inicialmente la librería `cflib.utils` contiene herramientas generales, donde `_Dirección_` es el canal y frecuencia de radio tomado del aplicativo Cfclient. El comando `uri = cflib.utils.urifrom_env(default = _Dirección_)` permite estimar la dirección de radio del dron para identificarlo respecto a los demás. En la librería Crtp, `cflib.crtp.init_drivers()` se inicializan los drivers como usb, radio, Tcp entre otros.

Cuando se realiza el llamado: `from cflib.crazyflie import Crazyflie`: se importa de la librería un objeto que representa al crazyflie, `cf = Crazyflie(rw_cache='./cache')`: permite configurar el objeto y finalmente, se llama el recurso: `from cflib.crazyflie.syncCrazyflie import SyncCrazyflie` para importar la función de sincronización con el dron. Se configura: `SyncCrazyflie(uri, cf=cf)SyncCrazyflie(uri, cf=cf).`

Tanto la asignación de movimientos, como la obtención de coordenadas se realiza dentro del bloque generado como: `with SyncCrazyflie(uri, cf=cf) as scf:`

3.4.2. Asignación de movimientos

La librería llamada `position_hl_comander.py` incluye las funciones necesarias para controlar el dron, es necesario importarlo en el programa principal mediante: `from cflib.positioning.position_hl_commander import PositionHlCommander`.

Luego de ello se debe inicializar con el objeto `scf` mencionado anteriormente un bloque de asignación de movimientos con `with PositionHlCommander(scf) as mc:`, donde puntualmente la función `mc.set_default_velocity(_vel_)`, y `mc.go_to(_x_,_y_,_z_)` son las indicadas para la asignación de velocidad y coordenadas.

3.4.3. Recepción de coordenadas

Existen dos formas de reconocer las coordenadas actuales, por medio de la librería mencionada `position_hl_comander.py` mediante el método `data=mc.get_position()` se puede

obtener la coordenada asignada previamente. Para conocer la coordenada que el sistema de posicionamiento recibe actualmente es necesario implementar la librería `logging.py`, e importar el objeto mediante la línea `from cflib.crazyflie.syncLogger import SyncLogger`. Para este fin es necesario tomar una porción de código de igual manera que con la asignación de movimiento, empezando con `with SyncLogger(scf, lg_stab) as logger:`, y recibiendo los datos de posición mediante `datos = logger`

3.4.4. Swarm

Cuando se requiere trabajar con enjambres de drones es crucial estimar las coordenadas correctamente, asignar las direcciones de radio y el firmware adecuado para cada dron. Todo esto puede realizarse mediante la aplicación Cfclient.

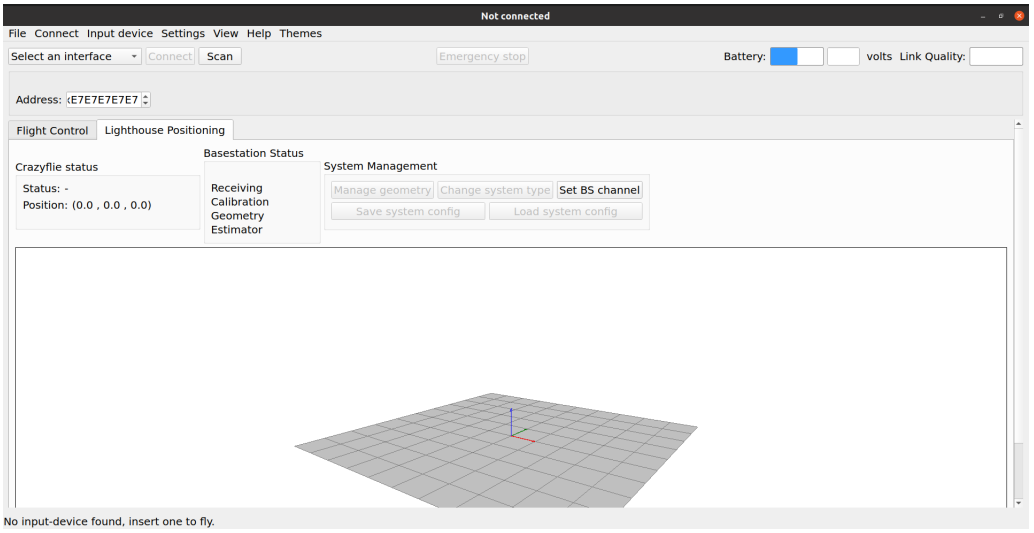


FIGURA 3: Aplicación Cfclient, usada para el ajuste de los drones. Imágen tomada en el momento en que se instaló en el dispositivo de desarrollo.

En el menú, en la opción (connect), es posible configurar la frecuencia de radio de cada dron. En la opción (Configure 2.x) se puede configurar la dirección de radio, para los enjambres se recomienda usar la misma frecuencia para todos y cambiar el valor del canal, o dirección (por default viene 0xE7E7E7E7E7). Se recomienda un ancho de banda entre 20 y 100 MHz [34]. En la opción (Bootloader) es posible cambiar el firmware, desde el inicio del proyecto se revisa que la versión 2022.03 es la más estable, con otras versiones los drones se reinician con movimientos bruscos o de plano no se reconoce el hardware. Finalmente la estimación de la geometría puede realizarse mediante el entorno gráfico, en el menú de (System Management), en la opción (Manage geometry) se puede estimar la geometría del sistema de posicionamiento mediante una

serie de pasos en los cuales se cambia la posición del dron. Cuando se termina la estimación es posible guardar la configuración en (Save System Config), lo que producirá un archivo `.yml` que puede ser cargado a los demás drones, es importante hacerlo con todos los que se vayan a usar.

En la parte de código, tal y como se implementa la librería `Psition_hl_commander`, se debe usar la librería `High_level_commander` que es la precisada para el uso de enjambres, mediante la misma es posible coordinar el enjambre. En principio hay funciones de la librería y desarrolladas por la comunidad que permiten controlar los enjambres.

```
def wait_for_position_estimator(scf):
    print('Waiting for estimator to find position...')

    log_config = LogConfig(name='Kalman Variance', period_in_ms=500)
    log_config.add_variable('kalman.varPX', 'float')
    log_config.add_variable('kalman.varPY', 'float')
    log_config.add_variable('kalman.varPZ', 'float')

    var_y_history = [1000] * 10
    var_x_history = [1000] * 10
    var_z_history = [1000] * 10

    threshold = 0.001

    with SyncLogger(scf, log_config) as logger:
        for log_entry in logger:
            data = log_entry[1]

            var_x_history.append(data['kalman.varPX'])
            var_x_history.pop(0)
            var_y_history.append(data['kalman.varPY'])
            var_y_history.pop(0)
            var_z_history.append(data['kalman.varPZ'])
            var_z_history.pop(0)

            min_x = min(var_x_history)
            max_x = max(var_x_history)
            min_y = min(var_y_history)
            max_y = max(var_y_history)
            min_z = min(var_z_history)
            max_z = max(var_z_history)

            # print("{} {} {}".
            #       format(max_x - min_x, max_y - min_y, max_z - min_z))

            if (max_x - min_x) < threshold and (
                    max_y - min_y) < threshold and (
                    max_z - min_z) < threshold:
                break
```

FIGURA 4: Estimador de posición para cada dron. Tomado de la librería del fabricante [33]

Como se puede ver en la Figura 4, el tipo de control desarrollado para los drones consta de un filtro de Kalman, que es ajustado para el posicionamiento. Se trata de un filtro complementario al control de estabilización que puede ser Mellinger o el clásico PID, se recomienda desactivar la función que viene por defecto para el uso de Mellinger [34].

```python
def reset_estimator(scf):
    cf = scf.cf
    cf.param.set_value('kalman.resetEstimation', '1')
    time.sleep(0.1)
    cf.param.set_value('kalman.resetEstimation', '0')
    wait_for_position_estimator(scf)

def activate_high_level_commander(scf):
    scf.cf.param.set_value('commander.enHighLevel', '1')

def activate_mellinger_controller(scf, use_mellinger):
    controller = 1
    if use_mellinger:
        controller = 2
    scf.cf.param.set_value('stabilizer.controller', controller)

def take_off(scf):
    activate_mellinger_controller(scf, False)
    commander= scf.cf.high_level_commander
    commander.takeoff(1.0, duration_s = 3)
    time.sleep(3)

def land(scf):
    commander= scf.cf.high_level_commander
    commander.land(0.0, 2.0)
    time.sleep(2)
```

FIGURA 5: Diferentes funciones para controlar el enjambre. Tomado de la librería del fabricante [33]

Las funciones mencionadas permiten controlar cada uno de los drones mediante hilos que son llamados con la función `swarm.parallel_safe(nombre_de_función)` donde el objeto `swarm` se instancia mediante un protocolo desarrollado por el fabricante.

```python
if __name__ == "__main__":
    cflib.crtp.init_drivers()
    factory = CachedCfFactory(rw_cache='./cache')

    with Swarm(uris, factory=factory) as swarm:
        # Conexión CF y despegue
        print('Connected to  Crazyflies')
        swarm.parallel_safe(activate_high_level_commander)
```

FIGURA 6: Protocolo para definir el enjambre y llamado de una función en paralelo. Tomado de la librería del fabricante [33]

3.5. Comunicación por sockets mediante Python

En [35] se encuentra la información asociada a cómo generar una comunicación mediante TCP/IP. Es posible implementar la librería `socket` en Python para el manejo de puertos, con ella asignar el objeto socket a una instancia y poder generar tanto la implementación de un servidor como de un cliente. Es necesario que ambos dispositivos se encuentren conectados a la misma red para hacerlo de una forma breve, puntualmente al mismo router, ya sea, mediante Wifi como Ethernet. Es crucial conocer la dirección IP de cada uno de los computadores, y

además del router o dispositivo de enlace, para ello es posible escribir en la terminal de linux: `hostname -i`, para conocer la dirección IP del dispositivo y `curl ifconfig.me` para conocer la dirección IP pública, es decir, del router. Luego de conocer las direcciones IP asociadas, el paso siguiente es desarrollar los scripts para servidor y cliente:

3.5.1. Servidor

En principio se genera la instancia del socket mediante el método socket asociado a la clase socket: `instancia = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`, luego se llama el método `instancia.bind(ip_server, port)`, donde el primer parámetro contiene una dirección ip para el servicio (se revisa el uso de la dirección 0.0.0.0) y el segundo, el puerto, que se trata de un número decimal de cuatro cifras. Es importante recalcar que la dirección IP mencionada no es la misma del equipo ni del router.

Para establecer la comunicación se implementa el método asociado a la instancia: `instancia.listen(numero_conexiones)`, y para identificar al cliente se implementa la función asociada a la instancia: `conexión, dirección = instancia.accept()`.

3.5.2. Cliente

La conformación del cliente es similar a la del servidor, en tanto se instancia de la misma manera el objeto socket, sin embargo, para realizar la comunicación, se implementa el código: `instancia.connect((IP_servidor, puerto_servidor))`. Los parámetros asignados a la función deben corresponderse con la información del servidor.

3.5.3. Funciones en común

La primera función crucial para la comunicación implementada tanto en el cliente como en el servidor es `datos = conexión.recv(tamaño_buffer)`, junto con la función `datos.decode(utf-8)` que permiten recibir y decodificar los datos.

Para el envío de datos se implementa `conexión.send(mensaje.encode('utf-8'))`, siendo `mensaje` una cadena de caracteres, en el caso puntual de utilización en el presente proyecto. Sin embargo puede enviarse cualquier tipo de variable o archivo por medio de esta comunicación.

Es importante notar que la instancia `conexión` en el servidor cumple la misma función que la instancia del socket en el cliente.

Capítulo 4

Metodología

La primera fase se desarrolla para identificar el correcto funcionamiento del dron Crazyflie 2.1 apropiando el software pertinente para poder asignar movimientos mediante un computador [20], puntualmente se desarrolla un conjunto de acciones en torno a la revisión de la estabilidad del dron, a la precisión del seguimiento de coordenadas y además la ejecución de un script que pueda servir como base para el desarrollo del enjambre.

Para la segunda fase se apropia el algoritmo Q-learning y se precisa sintonizar, [25] brinda las bases para la comprensión del aprendizaje por refuerzo del cual se basa Q-learning, en [26] se revisa la estructura de Q-learning, la estructuración específicamente en el apartado de creación de políticas y reglas que debe seguir el agente, de [27] se revisa el comportamiento de Q-learning en entornos dinámicos, es importante, ya que la función de recompensa planteada debe ser dinámica debido a que el objetivo es móvil (dron líder), En [28] se consignan instrucciones y consideracoiones para aplicar el algoritmo Q-learning sobre Python, cumple con la función de guía para la implementación del lenguaje de programación mencionado y la estructuración del algoritmo.

En la tercera fase, que es para la asignación de roles y el código a implementar para cada agente, es necesario considerar la cantidad de agentes disponibles y su disposición. Se revisa la estructuración del enjambre desde una visión general y también desde cada rol de los agentes, con el fin de encontrar la mejor forma de organizar los nodos y tópicos en ROS.

En la cuarta fase se implementa un esquema de transmisión de datos relacionados con las ubicaciones de los agentes a los cuales se les asigna un nodo en ROS. Para tener un acercamiento a la implementación de nodos incluyendo un dron Crazyflie se toma [30]. Además, en [31] se presenta la base para el desarrollo sobre ROS, las consideraciones e instrucciones para poder

llevar a cabo el programa, en [32] se revisa la utilización de Catkin, herramienta para la creación de un entorno de trabajo estructurado y que brinda facilidades en el uso del sistema de archivos requerido para el proyecto a implementar, al final de esta fase se implementa la comunicación sobre los agentes y se valida el comportamiento de enjambre sobre las plataformas físicas dispuestas, es importante recalcar que se parte del hecho de revisar que no existan colisiones y que los agentes sigan al líder, bajo estos criterios se ajustan los procesos desarrollados en las anteriores fases.

Capítulo 5

Desarrollo Conceptual

5.1. Asignación de comandos y recepción de coordenadas del dron

En este apartado se revisa el cómo se realizó la estimación de las coordenadas mediante el sistema de posicionamiento, para revisar la precisión del movimiento del dron, junto con la asignación de coordenadas en un vuelo de prueba. Puntualmente la forma en que se produjo el script.

En torno a la librería `Cflib` se desarrolla todo el código que permite la asignación de movimientos al dron y estimar su posición real mediante el sistema. Se decide implementar sobre dos hilos el desarrollo, uno para cada fin mencionado anteriormente. Para ello se realiza la importación de la librería `threads`, y para muestrear la posición respecto al tiempo es necesario implementar la librería `time`. La librería `matplotlib` se usará para revisar el comportamiento en el tiempo, además de todas las mencionadas anteriormente que hacen parte de la librería `Cflib`.

```python
# Librerías
import time
from threading  import Thread
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import logging
import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncLogger import SyncLogger
from cflib.utils import uri_helper
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.positioning.position_hl_commander import PositionHlCommander
```

FIGURA 7: Librerías implementadas en la asignación de ruta para un dron individual

```python
# Constantes
max_xy = 0.7 # valor máximo medido (70cm)
sequence = [(0.0, 0.0, 1), # coord. a seguir
            (max_xy , 0.0, 2),
            (max_xy , max_xy , 1.5),
            (-max_xy , max_xy , 1),
            (-max_xy , -max_xy , 1.5),
            (max_xy , -max_xy , 1),
            (0.0, 0.0, 0.5)]

#tm = 1/100, 100 muestras por segundo
stop = 80000
endTime = time.time() + stop
i=0
# Array donde se alacenan los datos
x = np.zeros(stop)
y = np.zeros(stop)
z = np.zeros(stop)
# Almaceno la dirección de radio e inicializo loggin
uri = uri_helper.uri_from_env(default='radio://0/80/2M/E7E7E7E7E7')
logging.basicConfig(level=logging.ERROR)
```

FIGURA 8: Variables implementadas en la asignación de ruta para un dron individual

Como se puede revisar en la figura 8, se implementa una ruta de vuelo preestablecida, consignada en la variable `sequence`, se revisa el paso del dron sobre estos puntos asignados. Posteriormente, se asigna un tiempo sobre estimado para el paso del vuelo en la variable `endtime`, que es quien asigna la terminación del hilo de lectura de coordenadas. En las matrices `x, y, z`, se consignarán todas las posiciones que el dron tuvo en el tiempo. Finalmente, se asigna el

canal de radio suministrado por la aplicación `Cfclient` a la hora de realizar la conexión de prueba.

```
81  # Main
82  if __name__ == '__main__':
83      stop_condition=0
84      cflib.crtp.init_drivers()# inicializo los drivers
85      #Configuro la recepción de las coordenadas:
86      lg_stab = LogConfig(name='Stabilizer', period_in_ms=10)
87      lg_stab.add_variable('stateEstimate.x', 'float')
88      lg_stab.add_variable('stateEstimate.y', 'float')
89      lg_stab.add_variable('stateEstimate.z', 'float')
90      # Se crea el objeto cf
91      cf = Crazyflie(rw_cache='./cache')
92
93      with SyncCrazyflie(uri, cf=cf) as scf:
94          lead.start() #inicio el hilo de asignación de coordenadas
95          look_oordenates() # corro la revisión de coordenadas
96
97      print_coord()
```

FIGURA 9: Programa principal, asignación de ruta para un dron individual

En el programa principal, se realiza la inicialización tanto de la asignación como de la recepción de las coordenadas, seguidamente se llama al hilo de asignación en la línea 94 y luego la recepción de coordenadas, para finalmente imprimir mediante matplotlib dichas coordenadas en el tiempo, estas funciones se revisan en las posteriores figuras 10, 11 y 12:

```
63  # Hilo: asignar coordenadas
64  def asign_oordenates():
65      global stop_condition
66      with PositionHlCommander(scf) as mc:
67          mc.set_default_velocity(0.25) #configuro la velocidad
68          for position in sequence:
69              mc.go_to(position[0],position[1],position[2]) #me muevo a la corrdenada objetivo
70              time.sleep(0.1)
71              #x = mc.get_position()
72              #print(x)
73          time.sleep(0.05)
74          mc.land() # aterrizo al final
75          stop_condition = 1 # asigno stop condition
76  lead = Thread(target = asign_oordenates, name="01")
```

FIGURA 10: Hilo para asignación de coordenadas, asignación de ruta para un dron individual

El código de la figura 10 permite la asignación de las coordenadas mediante la herramienta `PositionHlComander`, mediante la misma se configura la velocidad a la que se espera que el dron reaccione. Seguidamente, se asignan las coordenadas establecidas en la variable `position`, que se encarga de tomar las coordenadas asignadas en `secuence`. Es importante

notar que el ordenador ejecuta el método `mc.goto(x,y,z)` hasta que el dron se encuentra en la coordenada asignada. Cuando terminan los intentos el dron aterriza mediante `mc.land`, y se activa la condición de stop que sirve para detener la lectura ejecutada en el hilo principal.

```
49  def look_oordenates():
50      with SyncLogger(scf, lg_stab) as logger:
51          for log_entry in logger:
52              data = log_entry[1]
53              x[i] = data['stateEstimate.x']
54              y[i] = data['stateEstimate.y']
55              z[i] = data['stateEstimate.z']
56              i=i+1
57              #print('%d %f %f %f'% (i, data['stateEstimate.x'], data['stateEstimate.y'], data['stateEstimate.z']))
58              time.sleep(0.01)
59              if stop_condition==1: # finalizo cuando el otro hilo lo indique
60                  break
```

FIGURA 11: Obtención de coordenadas en el loop principal

El hilo principal ejecuta la función `Look_oordenates`, la cual mediante la herramienta `SyncLogger` permite estimar las coordenadas que entrega el sistema de localización tal y como se mencionó en el marco teórico. Es importante tener en cuenta que `Log_entry` se refiere a la posición de cada dron, en este caso solo se lee las coordenadas de uno. `Log_entri` es una variable de tipo librería, la cual asocia el valor de la coordenada con su nombre como se revisa entre las líneas 53 y 55 del códgio de la figura 12. Dicho nombre se compone de `'stateEstimate.x'` donde (x) es cualquier coordenada. Como se observa, se tienen los array `x, y, z` y se va guardando su respectivo valor cada 0.05 segundos. Se detiene la ejecución cuando el hilo de asignación de coordenadas asigna a `stop_condition` el valor de uno.

```
41  # Funciones
42  def print_coord():
43      fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))
44      ax.plot(x, y, z)
45      ax.axes.set_xlim(left=-1.5, right=1.5)
46      ax.axes.set_ylim(bottom=-1.5, top=1.5)
47      plt.show()
```

FIGURA 12: Método para impresión de coordenadas

Finalmente mediante la librería matplotlib se grafican las coordenadas obtenidas anteriormente (figura 12), se crea una imagen de tipo proyección 3d, se imprimen los ejes mencionados y se limita el rango de impresión de la proyección 3d.

5.2. Implementación de Q-learning, entrenamiento y convergencia

En este apartado se revisa el script de entrenamiento del algoritmo Q-learning, denotando la forma en que se conforma la tabla de políticas Q, el cómo se realiza el entrenamiento en torno a las fases que le componen y además la forma en que son asignadas las recompensas.

5.2.1. Conformación de la tabla Q

Partiendo de que las políticas (lo que debe hacer el agente), son generadas por el proceso de aprendizaje, y que el mismo se da mediante la interacción con el ambiente, específicamente en el reconocimiento de qué se debe hacer (acciones), en cada situación (estados), se construye la tabla de políticas Q, considerando las acciones y los estados posibles.

Las acciones y estados posibles de un dron son innumerables ya que este agente permite movimientos en todos los ángulos en tercera dimensión. Sin embargo, es importante recalcar que para el correcto entrenamiento del algoritmo es necesario discretizar y tomar una cantidad de acciones y estados acorde a la carga computacional, ya sea para la implementación de un solo dron en un MCU o de todo el enjambre en un PC central.

Con base en lo expuesto, se empiezan a revisar los estados posibles del agente. El tipo de búsqueda realizada es informada, eso quiere decir que es posible conocer la posición del objetivo y compañeros por parte de cada agente, además de la distancia respecto a ellos. En torno a esto se plantea la discretización tanto de las distancias como del posicionamiento, con el objetivo de obtener la cantidad de estados y su organización. Se plantean entonces cuatro estados de posicionamiento respecto al objetivo, es decir, el objetivo se encuentra a la derecha (315 a 360, o 0 a 45 grados), izquierda (135 a 225 grados), adelante (45 a 135 grados) y atrás (225 a 315). La discretización de estos estados se puede dar en bits, para estos cuatro estados es posible implementar dos bits. Luego se tiene la discretización de la distancia, se plantean cuatro distancias posibles: 0 a 20 cm, 20 a 50 cm, 50 a 70 cm y más de 70 cm, estos cuatro estados también pueden ser representados con 2 bits. En este punto se tiene que la definición de los estados respecto al objetivo son 16 posibilidades (4 bits). Para evitar las colisiones solo se considera al compañero más cercano, ya que es quien puede producir un choque. Si consideramos los mismos estados respecto al compañero más cercano, entonces se tendrán otros 16 estados. Para finalizar, entonces se tiene que hay 256 estados uniendo las posibilidades de detección de objetivo y compañero (8 bits). En el cuadro 2, es posible revisar el orden de los bits de izquierda a derecha y lo que significa su valor.

Dir. Objetivo	Dist. Objetivo	Dir. Compañero	Dist. Compañero
Adelante (0b00)	0.0 - 0.2 (0b00)	Adelante (0b00)	0.0 - 0.2 (0b00)
Atrás (0b01)	0.2 - 0.5 (0b01)	Atrás (0b01)	0.2 - 0.5 (0b01)
Derecha (0b10)	0.5 - 0.7 (0b10)	Derecha (0b10)	0.5 - 0.7 (0b10)
Izquierda (0b11)	0.7<(0b11)	Izquierda (0b11)	0.7<(0b11)

CUADRO 2: Bits para cada estado, autoría propia

Luego de haber definido los estados, se asignan las acciones a las direcciones posibles, con el fin de tener congruencia, eso quiere decir, que las acciones que tendrá el agente serán adelante, atrás, izquierda y derecha. En este punto ya se tiene una tabla Q conformada.

```
73  """ Configuración de la tabla Q ---------------------------------- """
74  Estados = np.zeros(256, dtype=int)
75  for Estado_ in range(0,len(Estados)):
76      Estados[Estado_]=Estado_
77
78  Acciones = ['Adelante',
79              'Atrás',
80              'Derecha',
81              'Izquierda']
82
83  Q = np.array(np.zeros(( len(Estados), len(Acciones) )))
```

FIGURA 13: Conformación de la tabla Q

5.2.2. Aprendizaje y generación de políticas

El proceso de aprendizaje depende de los factores enunciados en el marco teórico, implementados en el código como A (Alfa: tasa de aprendizaje), R (Gamma: tasa de descuento) y E (Épsilon: tasa de exploración). Además de estos, se revisa que son requeridas variables auxiliares para los estados, las acciones, la recompensa y los pasos:

```
61  """ Variables auxiliares Q_learning ---
62  A = 1.0 # Alfa      (Aprendizaje)
63  R = 0.3 # Gamma     (Descuento)
64  E = 1.0 # Epsilon   (Exploración)
65  Rec = 0 # Recompensa
66  Pasos = 0
67  Estado = 0
68  Estado_id = Estado
69  Accion_seleccionada = '              '
70  Accion_id = Accion_seleccionada
71  Objetivo_alcanzado = False
```

FIGURA 14: Variables para implementar Q-learning

Mediante la variación de estos tres parámetros referidos se generan las políticas, es decir una tabla Q que funciona para todos los casos, cuyos valores pretenden ser inamovibles. El procedimiento planteado para llegar a este fin consta de tres fases que se desarrollan con el paso de los episodios:

En la primera fase se realiza el llenado de la tabla Q con acciones aleatorias (E: 100 % a 53 %) y el aprendizaje en torno solo a las recompensas (A: 100 % y R: 33 % a 97 %) .

En la segunda fase se prioriza el asentamiento de las políticas, disminuyendo la aleatoriedad (E:50 % a 6 %), y disminuyendo la importancia del aprendizaje y el descuento (A: 100 % a 10 % y R: 97 % a 7 %).

Finalmente en la tercera fase se revisa mediante diferentes casos, los choques ocurridos, y la cantidad de veces que el agente llegó al objetivo, es decir la convergencia del modelo.

```
110  def Generar_politicas():
111      global E, A, R
112      # fase 1 (Llenado por pruebas)
113      if Episodio < Num_Episodios/3:
114          E -= (1.4/Num_Episodios)
115          A = 1
116          R += (2/Num_Episodios)
117      # fase 2 (Construcción de las políticas)
118      if Episodio >= Num_Episodios/3 and Episodio < Num_Episodios*2/3:
119          R -= (2.7/Num_Episodios)
120          A -= (2.7/Num_Episodios)
121          E -= (1.4/Num_Episodios)
122      # fase 3 (pruebas entrenado)
123      else: None
```

FIGURA 15: Generación de políticas respecto a los episodios

En este punto, es pertinente revisar qué cantidad de episodios y pasos son necesarios para que sea efectivo el aprendizaje, planteado mediante las fases descritas. En principio se plantea que el agente va a dar un paso cada 5cm. Teniendo en cuenta que el espacio a implementar el enjambre son dos metros cuadrados, por lo tanto, para que el agente recorra todo el espacio, se tiene una cantidad de pasos dada por:

$$Pasos = (2^2)/0{,}05 = 80. \tag{5.1}$$

Considerando que la toma de decisiones puede ser equivocada en el entrenamiento, entonces se plantea un máximo de 160 pasos (dos veces recorrer todo el ambiente) por cada episodio. Si bien es una medida para permitir al agente tomar una cantidad de decisiones suficientes, es

importante señalar que no se debe extender la cantidad de pasos a infinito o a valores muy altos ya que el entrenamiento será ineficiente, esto debido a que el agente puede alejarse demasiado del espacio de estados en el entrenamiento, y no ser entrenado satisfactoriamente.

Para la designación del número de episodios se considera que debe producirse una cantidad de estos, que permita que el agente pueda estar en todos los estados posibles, recalcando que son 255 estados. La forma en que se halló la cantidad de episodios, se realizó mediante inspección del llenado de una lista de estados y la depuración (eliminación de estados repetidos), con el fin de conocer cuantos estados se habían visitado, luego de este proceso se encuentra que 1500 son los episodios pertinentes y que cada fase constará de 500.

```
36  """ Objetivo y compañeros ------------------------------------
37  Compañeros = np.array([ [1.0, 1.0], #Aquí va a estar el lider
38                          [0.5, 0.5],
39                          [0.5, 0.0],
40                          [0.2, 0.2],
41                          [0.1, 0.5]])
42
43  Objetivo = [-0.2, 0.2] # Asigno la dirección Objetivo
44
45  """ Variables para identificación de aprendizaje ------------
46  Num_Episodios = 1500
47  Max_pasos= 160
48  P = 0.05                # distancia del paso
49  x = np.zeros(Max_pasos) # coordenada x histórica
50  y = np.zeros(Max_pasos) # coordenada x histórica
51  Veces = 0               # veces que llegó
52  Veces_entrenado = 0
53  Choques = 0
54  Habilitador_conteo_choque = False
55  Estados_visitados = []
56  Estados_depurados = []  # almacena los estados sin repetir
57  Distancia_obj = 100
58  Distancia_obj_ant = 101
59  Distancia_comp = 100
60  Distancia_comp_ant = 101
```

FIGURA 16: Variables para la revisión del aprendizaje

Aunado a esto también se implementan variables para la dinámica de interacción entre los compañeros, el objetivo y el ambiente. Y finalmente se revisa las veces que el agente ha llegado y ha chocado.

```
283  """ Main ------------------------------------
284  if __name__ == '__main__':
285      for Episodio in range(Num_Episodios):
286          # Inicializo las variables
287          Objetivo_alcanzado = False
288          Pasos=0
289          Habilitador_conteo_choque = False
290          Escenario_random()
291          Generar_politicas()
292
293          while not Objetivo_alcanzado:
294              Qlearning()
295              x[Pasos] = coordendas_actuales[0]
296              y[Pasos] = coordendas_actuales[1]
297              Pasos = Pasos+1
298              Final()
```

FIGURA 17: Programa principal, ejecución de los episodios para el entrenamiento.

El aprendizaje entonces se ejecuta en el programa principal, allí, se tiene que por cada episodio la posición inicial del objetivo, compañeros y agente va a cambiar. Se van a modificar los parámetros de aprendizaje, y mientras no se alcance el objetivo o se cumpla el máximo de pasos, se correrá el episodio. Un episodio consta de la implementación del código en Q-learning (donde también se ejecutan los movimientos) y el almacenamiento de las coordenadas respecto al tiempo por cada paso, para su posterior revisión gráfica mediante `matplotlib`.

5.2.3. Código de Q-learning

Todo lo planteado anteriormente se aterriza en el código desarrollado para el aprendizaje de máquina, el mismo consta de diferentes funciones que permiten ejecutar el algoritmo Q-learning.

```
253  def Qlearning():
254      global Q
255      Sel_accion()
256      Movimiento()
257      Estado_nuevo()
258      Recompensa()
259      Q[Estado_id, Accion_id] = (1-A)*Q[Estado_id, Accion_id] + A*(Rec + R*np.max(Q[Estados[Estado]]))
```

FIGURA 18: Organización de los procesos para llevar a cabo Q-learning y ecuación en Python.

En la función `Q-learning` se realiza todo el proceso mencionado. A continuación, serán descritas las funciones que se implementan:

```
157  def Sel_accion():
158      global Accion_seleccionada, Estado_id, Accion_id
159
160      if np.random.uniform(0, 1) < E:
161          # Explorar: seleccionar una acción aleatoria
162              Accion_seleccionada = np.random.choice(Acciones)
163      else:
164          # Explotar: seleccionar la acción con el valor de Q máximo para el estado actual
165              Estado_id = Estados[Estado]
166              Accion_id = np.argmax(Q[Estado_id])
167              Accion_seleccionada = Acciones[Accion_id]
168
169      Estado_id = Estados[Estado]
170      Accion_id = Acciones.index(Accion_seleccionada)
```

FIGURA 19: Función para la selección de la acción.

En la función `Sel_accion` se realiza la toma de decisiones en torno al factor de exploración E, Aquí se decide si se toma una acción al azar o si se realiza la toma de la mejor decisión posible para el estado actual.

```
172  def Movimiento():
173      global coordendas_actuales
174      # Acciones para llegar  la dirección neceraria
175      if Accion_seleccionada == 'Adelante':
176          coordendas_actuales[0] =  round(coordendas_actuales[0]+P,2)
177
178      elif Accion_seleccionada == 'Atrás':
179          coordendas_actuales[0] = round(coordendas_actuales[0]-P,2)
180
181      elif Accion_seleccionada == 'Derecha':
182          coordendas_actuales[1] = round(coordendas_actuales[1]+P,2)
183
184      elif Accion_seleccionada == 'Izquierda':
185          coordendas_actuales[1] = round(coordendas_actuales[1]-P,2)
```

FIGURA 20: Función para ejecutar el movimiento del agente.

En la función `Movimiento` se ejecutan las modificaciones al valor de las coordenadas actuales, mediante el valor del paso anteriormente especificado.

```
187  def Estado_nuevo():
188      """ …
195      global Estado, Distancia_obj, Distancia_comp, Estados_visitados
196      global Distancia_obj_ant, Distancia_comp_ant
197      Estado=0; ang = 0
198      me = coordendas_actuales
199      ob = Objetivo
200
201      #-- Estado respecto al Objetivo
202      Distancia_obj_ant = Distancia_obj
203      Distancia_obj, ang = Get_dist_angle(me[0], me[1], ob[0], ob[1])
204
205      # Defino el estado de posición
206      if ang>=45 and ang < 135:                          Estado = 0 # adelante
207      if ang>=135 and ang < 225:                         Estado = 0B11000000 # izquierda
208      if ang >= 225 and ang < 315:                       Estado = 0B01000000 # atrás
209      if (ang>=315 and ang<=360) or (ang>=0 and ang<=45): Estado = 0B10000000 # derecha
210
211      # Defino el estado de distancia
212      if Distancia_obj >= 0 and Distancia_obj < 0.2:    Estado |= 0B00000000
213      if Distancia_obj >= 0.2 and Distancia_obj < 0.5:  Estado |= 0B00010000
214      if Distancia_obj >= 0.5 and Distancia_obj < 0.7:  Estado |= 0B00100000
215      if Distancia_obj >= 0.7:                           Estado |= 0B00110000
```

FIGURA 21: Definición del estado respecto al objetivo.

```
172  def Movimiento():
173      global coordendas_actuales
174      # Acciones para llegar  la dirección neceraria
175      if Accion_seleccionada == 'Adelante':
176          coordendas_actuales[0] =  round(coordendas_actuales[0]+P,2)
177
178      elif Accion_seleccionada == 'Atrás':
179          coordendas_actuales[0] = round(coordendas_actuales[0]-P,2)
180
181      elif Accion_seleccionada == 'Derecha':
182          coordendas_actuales[1] = round(coordendas_actuales[1]+P,2)
183
184      elif Accion_seleccionada == 'Izquierda':
185          coordendas_actuales[1] = round(coordendas_actuales[1]-P,2)
```

FIGURA 22: Definición del estado respecto al compañero más cercano.

Como se puede ver, en `Estado_nuevo` se asignan los índices del estado nuevo mediante números binarios. Además, para poder lleva a cabo el proceso de recompensa es necesario guardar las distancias respecto al objetivo y compañero más cercano, tanto actuales como anteriores. La función en la figura 23 `Get_dist_angle` entrega la distancia y ángulo de dos puntos, mediante:

$$\Theta = tan^{-}1(Y_distancia/X_distancia). \tag{5.2}$$

$$Distancia = \sqrt{Y_distancia^2 - X_distancia^2}. \tag{5.3}$$

```python
def Get_dist_angle(xo, yo, xc, yc):
    angle = 0
    x_dist = xc-xo
    y_dist = yc-yo
    T_dist = ((x_dist**2)+(y_dist**2))**(1/2)
    angle = np.arctan2(y_dist,x_dist)
    angle *= 180/np.pi
    if angle < 0:
        angle += 360
    return round(T_dist,2), angle
```

FIGURA 23: Función Get_dist_angle

```python
def Identificar_comp():
    Own = coordendas_actuales
    id=[];  dist_id = []
    dist = 0; ang = 0
    for comp in range(0, len(Compañeros)):
        if comp != ID:
            dist, ang  = Get_dist_angle(Own[0],Own[1],Compañeros[comp][0],Compañeros[comp][1])
            dist_id.append(dist)
    min_dist = min(dist_id)
    id.append(dist_id.index(min_dist))
    return Compañeros[id[0]]
```

FIGURA 24: Función Identificar_comp

La función `Identificar_comp` en la figura 24 devuelve el compañero más cercano calculando la distancia con todos y devolviendo las coordenadas del mismo. en esta función (`Estado_nuevo`) es donde se realiza el llenado de la lista de estados visitados para su posterior depuración e identificación de la cantidad de estados visitados.

```python
def Recompensa():
    global Rec, Habilitador_conteo_choque, Choques
    Rec = 0;

    # Asigno la recompensa respecto al objetivo
    if Distancia_obj_ant < Distancia_obj :    Rec -= 0.4
    else:                                     Rec += 0.2

    # Asigno la recompensa respecto al compañero más cercano
    if Distancia_comp<0.2 and Distancia_comp<Distancia_comp_ant:    Rec += -0.8
    if Distancia_comp<0.3 and Distancia_comp>Distancia_comp_ant:    Rec += 0.2

    # Reviso si hay choques en el entrenado
    if Distancia_comp<0.05 and Habilitador_conteo_choque==False and Episodio>=(Num_Episodios*2/3):
        Choques+=1
        Habilitador_conteo_choque = True
```

FIGURA 25: Función de recompensa

Finalmente, en la función `Recompensa`, se asigna una recompensa que depende de la dinámica del agente, específicamente en torno a las distancias entre objetivo y los demás agentes. La asignación de recompensas en torno a la distancia permitió generar una tabla Q, donde cada par estado acción fue debidamente llenado mediante la fase uno (de exploración).

Es importante notar que los valores de la recompensa fueron asignados por porcentajes de importancia que fueron variados a la hora de ejecutar el aprendizaje, el método de inspección de resultados fue aplicado para consignar estos valores, teniendo en cuenta que los choques siempre iban a ser el estado por evitar.

5.3. Roles del enjambre e implementación en código

En esta sección se describe la forma en que se codificó la red base para el enjambre de drones, enunciando las características globales y presentando todas las herramientas necesarias para ejecutar virtualmente el enjambre.

El tipo de comunicación y el cómo se organizan los roles de los agentes se basa en la comunicación serial, donde se tiene un maestro y varios esclavos identificables. Con base en esto se desarrollan las funciones de comunicación y se asignan roles específicos a cada tipo de agente. En un principio se decide manejar cada agente como un nodo subscriptor y publicador que se comunica mediante un `topic` común en ros, para inicializar se implementa la siguiente función:

```
def Init_Ros():
    pub = rospy.Publisher('red', String, queue_size=1) # topic coordenadas
    rospy.init_node(f'd{ID}', anonymous=False) # nombre del nodo
    Escuchar() # escucho en ambos nodos
    return pub
```

FIGURA 26: Ingreso al nodo, subscripción y emisión

Como se puede denotar el nombre del nodo, dentro de ese `topic` común, viene dado por su ID. El método desarrollado funciona para el líder y los seguidores, al igual que el presentado a continuación en la figura 27.

```python
def Comunicar_coordenadas(pub):
    global coordendas_actuales
    x = coordendas_actuales[0]
    y = coordendas_actuales[1]
    z = coordendas_actuales[2]
    Str = f"coord,{ID},{x},{y},{z}"
    pub.publish(Str)

def Comunicar_red(pub):
    global mensaje_ext
    pub.publish(f"red,{ID},{mensaje_ext}")
```

FIGURA 27: Tipos de mensajes comunes

La estructura de los mensajes que se envían al `topic` en común se desarrolla en un arreglo tipo `string` con el orden: `tipo de mensaje, ID del comunicador, mensaje a enviar`. En el método `Comunicar_coordenadas` el mensaje a enviar son las coordenadas actuales. El envío siempre será realizado mediante el método `pub.publish(datos)`. En el apartado de `Comunicar_red` el mensaje enviado se compone de una palabra, dichas palabras indicarán estados, tales como disponibilidad (envío de un "Hola"), verificación de un proceso y además el aviso de que se cumplió una tarea por parte de cualquiera de los agentes.

Para la recepción también se implementan métodos similares entre el usado por el líder y los seguidores, El método de recepción es una función `callback` o (similar a un llamado de interrupción), en ella se realiza la lectura del `topic` periódicamente. Sin embargo su ejecución entre el líder y los seguidores difiere como se verá más adelante.

5.3.1. Roles del líder

En principio el líder se encarga de comunicarse con el usuario para saber las coordenadas objetivo del enjambre, para ello se implementa la función a continuación:

```
#----------------------- Comunicación con el usuario ---------------#
def Com_usr_init():

    global objetivos
    global vuelos
    com_aux = 0

    vuelo_aux = 0
    coordenadas = np.array([0.0, 0.0, 0.0])
    nombres = np.array(['x','y','z'])

    print("Por favor digita cuantos vuelos deseas hacer entre 1 y 5:")
    while 1:
        usr_data = input(f"\r\n Cantidad de vuelos:")
        num_aux = abs(int(usr_data))
        if num_aux <= 5 and num_aux > 0:
            vuelos = num_aux
            break
```

(A) Obtención de los intentos vuelos

```
    print("Por favor digita las coordenadas objetivo para el dron")
    print("debe ser un número entre -1.5 y 1.5 para 'x' y 'y', para")
    print("'z' debe ser un número entre 0.5 y 2.5, presiona enter al")
    print("escribirlo")

    for vuelo_aux in range (0, vuelos):
        for com_aux in range(0,3):
            while 1:
                usr_data = input(f"{nombres[com_aux]}:")
                num_aux = float(usr_data)
                if abs(num_aux) <= 1.5 and nombres[com_aux] != 'z':
                    coordenadas[com_aux]= num_aux
                    break
                if abs(num_aux) <= 2.5 and abs(num_aux) >= 0.5 and nombres[com_aux] == 'z':
                    coordenadas[com_aux]= num_aux
                    break
                else:
                    print("error")

        print(f"\r\nlas coordenadas ingresadas para el vuelo {vuelo_aux} son {coordenadas}\r\n")
        objetivos[vuelo_aux] = coordenadas
```

(B) Obtención de las coordenadas objetivo.

FIGURA 28: Comunicación con el usuario.

El código presentado cumple la función de recibir la cantidad de vuelos (entre uno y 5) y las coordenadas objetivo de cada vuelo, es algo extenso porque se implementaron funciones para

evitar que el usuario ingrese coordenadas que no puede cubrir el sistema y además evitar que se realicen más vuelos de los permitidos.

Estos objetivos son almacenados en la variable `coordenadas`, los datos almacenados se implementarán para asignar las posiciones a cada dron seguidor en una función que debe calcular el valor más cercano. Primeramente, es necesario realizar una verificación de los nodos que representan los drones se encuentran disponibles:

```
def Init_com(pub):
    global mensaje_ext
    # Inicio el sistema avisando a los demás drones que inicien operaciones
    #   1) Aviso inicio a los demás drones
    mensaje_ext = "Hola" #formato: "id,mensaje
    Comunicar_red(pub)

    #   2) Espero que los demás drones respondan
    for aux_red in range(1, num_drones):
        while 1:
            if mensajes_drones[aux_red]=="Hola":
                break
```

FIGURA 29: Verificación de los drones en la red

Tal y como indican los comentarios de la función, primero se comunica un (Hola) a todos los drones y se espera la respuesta de cada uno de ellos, hasta que no se comuniquen todos, no se procede.

A continuación, se realiza el cálculo de las coordenadas objetivo que serán asignadas a cada seguidor, independientemente que el dron principal llegue al objetivo, los demás deben llegar a su coordenada establecida en esta parte.

```python
def Sel_coord(pub):
    Compañeros_objetivos[0] = objetivos[episodio]
    Pos_aux = Compañeros_objetivos[0]
    Pos_ex = np.array([0.2, -0.2])
    #1 Designo los espacios posibles
    Esp_pos = []
    #Guardo los valores posibles alrededor del objeetivo
    Esp_pos.append( [Pos_aux[0]+Pos_ex[0], Pos_aux[1]+Pos_ex[0], Pos_aux[2]+Pos_ex[0]] )
    Esp_pos.append( [Pos_aux[0]+Pos_ex[0], Pos_aux[1]+Pos_ex[0], Pos_aux[2]+Pos_ex[1]] )
    Esp_pos.append( [Pos_aux[0]+Pos_ex[0], Pos_aux[1]+Pos_ex[1], Pos_aux[2]+Pos_ex[0]] )
    Esp_pos.append( [Pos_aux[0]+Pos_ex[0], Pos_aux[1]+Pos_ex[1], Pos_aux[2]+Pos_ex[1]] )
    Esp_pos.append( [Pos_aux[0]+Pos_ex[1], Pos_aux[1]+Pos_ex[0], Pos_aux[2]+Pos_ex[0]] )
    Esp_pos.append( [Pos_aux[0]+Pos_ex[1], Pos_aux[1]+Pos_ex[0], Pos_aux[2]+Pos_ex[1]] )
    Esp_pos.append( [Pos_aux[0]+Pos_ex[1], Pos_aux[1]+Pos_ex[1], Pos_aux[2]+Pos_ex[0]] )
    Esp_pos.append( [Pos_aux[0]+Pos_ex[1], Pos_aux[1]+Pos_ex[1], Pos_aux[2]+Pos_ex[1]] )
    #2 calculo la distancia con respecto al compañero
    i=1
    for compañero in range(1,len(Compañeros)):
        distances = []
        for pos in Esp_pos:
            distx = (Compañeros[compañero][0]-pos[0])**2
            disty = (Compañeros[compañero][1]-pos[1])**2
            distz = (Compañeros[compañero][2]-pos[2])**2
            dist = (distx+disty+distz)**(1/2)
            distances.append(dist)

        Comp_ind = round(distances.index(min(distances)),2)
        Compañeros_objetivos[compañero] =  Esp_pos[Comp_ind]
        Comunicar_objetivo(pub,i)
        time.sleep(0.1)
        del Esp_pos[Comp_ind]
        i=i+1
```

FIGURA 30: Selección de coordenadas efectuada por el líder.

Se designan coordenadas a una distancia de 20cm dependiendo de la posición en la que se encuentren los seguidores. es importante considerar que la forma en que se asignan debe ser excluyente, es decir, las coordenadas objetivo serán asignadas a solo un dron para evitar colisiones, en un principio se comunica a cada dron su coordenada objetivo mediante el método `Comunicar_objetivo`:

```python
def Comunicar_objetivo(pub, d):
    global Compañeros_objetivos
    Co=Compañeros_objetivos
    pub.publish(f"obj,{d}, {Co[d][0]},{Co[d][1]},{Co[d][2]}")
```

FIGURA 31: Comunicación de la coordenada objetivo de los seguidores por parte del líder.

Es importante resaltar que el mensaje tiene la etiqueta (obj), que es para la comunicación de objetivos. Luego se comunica el ID del dron seleccionado y finalmente la coordenada objetivo, en el orden para los mensajes descrito anteriormente.

Cada que todos los drones cumplen con llegar al objetivo comunican (Terminé) a la red con el indicador (red). En la función de callback se reciben estos datos, y se procede a reiniciar el proceso para un nuevo objetivo. Estos datos, las coordenadas y los mensajes de propósito general son recibidos en la función mencionada:

```
def callback(data):
    global mensajes_drones
    total = data.data.split(',')
    if total[0]=="coord":
        id_aux = int(total[1])
        Compañeros[id_aux][0] = float(total[2])
        Compañeros[id_aux][1]  = float(total[3])
        Compañeros[id_aux][2]  = float(total[4])

    if total[0]=="red":
        id_aux = int(total[1])
        mensajes_drones[id_aux] = total[2]
```

FIGURA 32: Recepción de datos del `topic`

La recepción de los datos se realiza en una función callback, al igual que en el nodo de líder, allí se atiende a los datos enviados al `topic` común. Si se recibe un dato con el indice (Coord), se procede a almacenar la coordenada enviada por el nodo que representa al dron y se identifica el ID del mismo con el fin de guardarlo correctamente en el llamado `Compañeros`. Por otra parte, cuando se recibe comunicación, con el indicativo (red), se realiza el mismo procedimiento pero se guardan los mensajes de los drones, en dependencia de su ID en la variable `mensajes_drones`.

Es importante notar que el nodo líder debe considerar los estados que comunican los seguidores, en torno a esto identifica si el enjambre ya cumplió su cometido y procede a la asignación de nuevos objetivos.

5.3.2. Roles de los seguidores

Las funciones del seguidor, además de las comunes mencionadas anteriormente, giran en torno a llegar a la coordenada asignada por el líder, y comunicar sus estados.

```
def callback(data):
    global mensajes_drones, Init

    total = data.data.split(',')
    if total[0]=="coord":
        id_aux = int(total[1])
        Compañeros[id_aux][0] = float(total[2])
        Compañeros[id_aux][1]  = float(total[3])
        Compañeros[id_aux][2]  = float(total[4])

    if total[0]=="red":
        id_aux = int(total[1])
        mensajes_drones[id_aux] = total[2]

    if total[0]=="obj":
        id_aux = int(total[1])
        if(id_aux == ID):
            objetivos[0] = float(total[2])
            objetivos[1] = float(total[3])
            objetivos[2] = float(total[4])
            print(f'Mi objetivo es:{objetivos}')
            Init = True
```

FIGURA 33: Recepción de datos del `topic` por parte del seguidor

El método integra el reconocimiento de los mensajes con identificación (coord) y (red) al igual que el maestro. Sin embargo, difiere en que tiene un apartado para la recepción de las coordenadas objetivo enviadas por el líder bajo la identificación (obj).

En el proceso de comunicación, el seguidor cumple con responder al líder si este necesita identificar a los que se encuentran en la red, además de comentar si llega al objetivo o si cumple sus pasos máximos mediante el identificador (red), y la palabra (Terminé) anteriormente mencionada.

5.4. Implementación del conjunto de drones con el modelo obtenido

5.4.1. Comunicación y sus funciones de los dispositivos de comunicación entre ROS y el enjambre

La implementación de ROS y el sistema de posicionamiento en un solo ordenador no es posible, ya que el último accede a recursos compartidos (OpenCv), que generan problemas con ROS y se evidencian a la hora de controlar el vuelo de los drones [34]. Por esta razón se hace necesario implementar dos ordenadores, concretamente se desarrolla el código de cliente (encargado

de la generación de coordenadas implementando ros) y servidor (encargado de la asignación de coordenadas a los drones en vuelo). A continuación, se explica la funcionalidad de ambos scripts, junto con porciones de código detalladas.

5.4.1.1. Servidor

En este apartado se genera un script que será lanzado mediante un ordenador, el mismo tiene los drivers y el hardware asociado a los drones. En él se inicializa la comunicación como servidor, y su tarea se basa en posicionar los drones mencionados, señalar al cliente que está dispuesto a recibir las coordenadas objetivo y finalmente coordinar el enjambre de drones. La coordinación se ejecuta mediante los comandos revisados en el subcapítulo 3.4.4, consignado en el marco teórico como: Swarm.

Las librerías a implementar son útiles para generar la conexión con el servidor, realizar la lectura en paralelo de los datos recibidos del cliente, decodificarla y controlar el vuelo de los drones.

```
1  import socket
2  import time
3  import json
4  import numpy as np
5  from threading import Thread
6
7  import cflib.crtp
8  from cflib.crazyflie.log import LogConfig
9  from cflib.crazyflie.swarm import CachedCfFactory
10 from cflib.crazyflie.swarm import Swarm
11 from cflib.crazyflie.syncLogger import SyncLogger
12 from cflib.crazyflie import syncCrazyflie
```

FIGURA 34: Librerías del script para conexión y asignación de coordenadas a los drones en vuelo.

En el apartado de conexión (figura 35), se realiza la instancia del puerto asociado al servidor. Es importante recalcar que el código de la línea 32 detiene la ejecución del programa, para poder esperar la comunicación con el cliente. Cuando se realiza la conexión se instancia el objeto `conexion`, que permite la comunicación con el cliente enlazado.

```
# Conectar a la red
mi_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_ip = "0.0.0.0"
port = 5555
mi_socket.bind((server_ip, port))
mi_socket.listen(1)
print("Esperando")
conexion, addr = mi_socket.accept()
print("Nueva conexión establecida")
print(addr)
```

FIGURA 35: Creación del servidor y conexión a la red.

Como la coordinación de dos equipos puede presentar problemas en torno a la sincronización, se decide crear un hilo de recepción, el cual va a estar pendiente de los datos que entran, en periodos de tiempo determinados. En dicho hilo se ejecuta un procedimiento de arreglo de las coordenadas que llegan, para poder cargarlas a los drones con el formato adecuado (se pasa de un formato utf-8 a vectores de datos tipo float).

```
def Escuchar_red():
    global PC1_msg, Compañeros
    while True:
        datos = conexion.recv(1024)
        PC1_msg = datos.decode('utf-8')
        if len(PC1_msg) > 30: # recibo coordenadas
            None
            PC1_msg = PC1_msg.replace("\n ", ",")
            PC1_msg = PC1_msg.replace("   ", " ")
            PC1_msg = PC1_msg.replace("  ", ",")
            PC1_msg = PC1_msg.replace(" -", ",-")
            PC1_msg = PC1_msg.replace(" ]", "]")
            PC1_msg = PC1_msg.replace("[ ", "[")
            PC1_msg = PC1_msg.replace(" ", ", ")
            PC1_msg = PC1_msg.replace(".,", ".0,")
            PC1_msg = PC1_msg.replace("][", "],[")
            PC1_msg = PC1_msg.replace(",]", "]")
            Compañeros = json.loads(PC1_msg)
        time.sleep(0.1)
listen_PC1 = Thread(target=Escuchar_red, name="01")
```

FIGURA 36: Recepción de los datos del cliente, paralelo al programa principal.

Con todo lo descrito anteriormente, la parte de conectividad está cubierta, ahora, para poder operar con los drones es necesario tener las direcciones de radio de cada uno de ellos, el tiempo de envío de las coordenadas a los drones y las posiciones iniciales de estos, como se observa en la figura 38.

```
14  # Variables gloobales
15  PC1_msg = "                    "
16  time_tt = 0.5
17  Compañeros = np.array([ [0.0, 0.0, 0.7],
18                          [0.0, -0.4, 0.7],
19                          [0.0, 0.4, 0.7]])
20  uris = []
21  uris.append('radio://0/90/2M/E7E7E7E7E3')# Derecha
22  uris.append('radio://0/90/2M/E7E7E7E7E1')# Izquierda
23  uris.append('radio://0/90/2M/E7E7E7E7E4')# Centro
```

FIGURA 37: Variables globales para coordinar los drones mediante el servidor.

Además de las funciones que se implementan mediante hilos generados por la librería `Swarm`, se crearon dos funciones adicionales. La primera función es `go()`, que asigna al dron la coordenada objetivo que debe seguir en base a unos argumentos destinados a cada uno de ellos, además de que permite también indicar el tiempo de respuesta que se requiere del mismo. Nótese que la variable `relative` se asigna como un valor falso, de esta manera es posible indicar que la coordenada asignada es respecto al sistema en común y no a una coordenada relativa respecto a su posición de despegue.

La segunda función `asignar_coordenadas()`, se implementa para cargar las coordenadas recibidas en el hilo de comunicación, a los argumentos que serán pasados a la función `go()`.

```
127 def go (scf: syncCrazyflie, c):
128     activate_mellinger_controller(scf, False)
129     commander = scf.cf.high_level_commander
130     commander.go_to(c[0], c[1], c[2], 0, duration_s =time_tt, relative=False)
131     time.sleep(time_tt)
132
133 def asignar_coordenadas():
134     args = {uris[0]: [Compañeros[0]],
135                 uris[1]: [Compañeros[1]],
136                 uris[2]: [Compañeros[2]]}
137     return(args)
```

FIGURA 38: Funciones complementarias para la asignación de coordenadas del enjambre

Finalmente, en el programa principal se inica el hilo de recepción mencionado, los drivers para controlar los drones y se instancia el objeto de tipo `Swarm`. con este objeto se llaman las funciones de la librería, en principio se inicializa el driver de asignación de coordenadas `active_high_level_commander`. Posteriormente se reinician los estimadores de posición para poder despegar desde cualquier lugar (si no se hace los drones tratan de dirigirse al centro en el despegue y chocan), posteriormente se realiza el despegue y se dirigen los drones a las

coordenadas de inicio. Cuando llegan al lugar, se comunica al cliente que el sistema está en disposición de empezar a moverse, para recibir las coordenadas que asigne.

Entre las líneas 156 y 164 se realiza el proceso de asignación de coordenadas, enviando la palabra (Start) al cliente para que calcule la coordenada siguiente, y (Stop) cuando se ejecute el movimiento de un paso por cada dron. Al finalizar se cierra la comunicación.

```
139 if __name__ == "__main__":
140     listen_PC1.start()
141     cflib.crtp.init_drivers()
142     factory = CachedCfFactory(rw_cache='./cache')
143     with Swarm(uris, factory=factory) as swarm:
144         # Conexión CF y despegue:
145         print('Connected to  Crazyflies')
146         swarm.parallel_safe(activate_high_level_commander)
147         swarm.reset_estimators()
148         swarm.parallel_safe(take_off)
149         time.sleep(0.5)
150         swarm.parallel_safe(go, args_dict=asignar_coordenadas())
151         time.sleep(0.5)
152         # Envío mensaje de confirmación de posicionamiento:
153         mensaje = "Ok"
154         conexion.send(mensaje.encode('utf-8'))
155 
156         while PC1_msg != "Fin":
157             #-----------------------------------------------------start
158             mensaje = "Start", conexion.send(mensaje.encode('utf-8'))
159             time.sleep(0.05)
160             #-----------------------------------------------------stop
161             mensaje = "Stop", conexion.send(mensaje.encode('utf-8'))
162             print(Compañeros[0], Compañeros[1], Compañeros[2])
163             swarm.parallel_safe(go, args_dict= asignar_coordenadas())
164             time.sleep(time_tt/2)
165     """ Final """
166     listen_PC1.join
167     conexion.close
```

FIGURA 39: Programa principal del servidor, donde se asignan coordenadas al enjambre y se realiza la comunicación con el cliente.

5.4.1.2. Cliente

La coordinación de los drones mediante ROS en este apartado es igual que en el desarrollo revisado en el subcapítulo 5.3, llamado roles del enjambre e implementación en código. El cambio realizado trata de eliminar un dron (antes se usaban 4 para realizar las estimaciones en el entorno virtual) y además el rol del pintor se elimina. En este desarrollo se crea un nuevo script

nombrado `Sockets_ros.py`, en el cual se realiza la recepción de la comunicación por parte del servidor, y se efectúa la comunicación mediante su propio nodo hacia el `topic` de ros.

Las funciones de comunicación entre los nodos son las mismas implementadas para el dron cero, con la particularidad de que el id de este nodo es `ID=9`.

```
1   #!/usr/bin/env python3
2
3   from threading import Thread
4   import time
5   import socket
6   import numpy as np
7   import rospy
8   from std_msgs.msg import String
9
10  Compañeros = np.array([ [0.0, 0.0, 0.7],
11                          [0.0, -0.4, 0.7],
12                          [0.0, 0.4, 0.7]])
13
14  mensajes_drones = np.array(["        ", "        ", "        "])
15  mensaje_ext = "        "
16  PC2_msg = "             "
```

FIGURA 40: Variables globales y librerías implementadas en el cliente.

En la figura 41 se puede revisar la forma en la que se crea el cliente y cómo se realiza la conexión. La dirección IP consignada debe ser la del servidor, y el puerto de enlace debe ser el mismo (5555). Cuando se ejecuta esta porción de código, el servidor reconoce la conexión.

```
18  """ Inicio el puerto """
19  host_servidor = '192.168.0.115'
20  puerto_servidor = 5555
21  cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22  cliente.connect((host_servidor, puerto_servidor))
```

FIGURA 41: Inicialización del puerto y conexión con el servidor.

Además de lo mencionado, se adiciona la función de lectura periódica del buffer asociado a la red TCP/IP generada por el servidor, similar a como se utilizó en el script del servidor mencionado.

```
24  """ Funciones de comunicación PC """
25  def Receive_PC2():
26      global PC2_msg
27      while True :
28          datos = cliente.recv(1024)
29          PC2_msg = datos.decode('utf8')
30          time.sleep(0.05)
31  Listen_PC2 = Thread(target = Receive_PC2, name="01")
```

FIGURA 42: Recepción de los datos que envía el servidor.

A grandes rasgos, el programa tiene el objetivo de inicializar la escucha periódica con el servidor, esperar a que los drones se posicionen e indicar a cada uno de los nodos de ROS cuando pueden estimar el nuevo paso a dar, para posteriormente recibirlos y enviarlos a la red con el servidor.

```
61  if __name__ == '__main__':
62      pub = Init_Ros()
63      Listen_PC2.start()
64      while PC2_msg != "Ok": None     # Pregunto si los drones se posicionaron
65
66      # Comunico a la red que están listos y espero que todos se unan
67      mensaje_ext = "ok"
68      Comunicar_red(pub)
69      while mensajes_drones[0] != 'Hola': None # Espero que el dron0 salude
70
71      # Empiezo a solicitar coordenadas y enviar al PC2 la info
72      while mensajes_drones[0] != "Fin" :
73          # 1) Pregunto la indicación del central de drones
74          while PC2_msg != "Start": None
75          # 2) Envío a los drones que pueden comunicarse
76          mensaje_ext = "Start"
77          Comunicar_red(pub)
78          time.sleep(0.1)
79          # 3) Envío indicación de paro a los drones
80          mensaje_ext = "Stop"
81          Comunicar_red(pub)
82          time.sleep(0.1)
83          # 4) Envío las coordenadas al PC2
84          mensaje = f"{Compañeros}"
85          cliente.send(mensaje.encode('utf-8'))
86
87      # Cierra la conexión
88      Listen_PC2.join()
89      cliente.close()
```

FIGURA 43: Programa principal del cliente.

5.4.1.3. Consideraciones respecto a los nodos

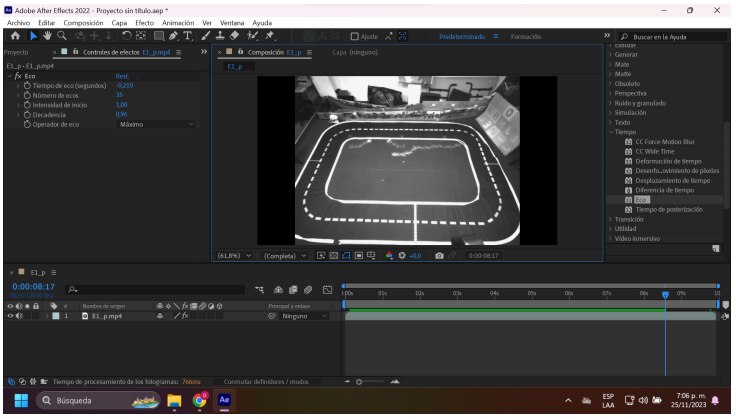
Es importante recalcar que en cada nodo ya no se produce el cálculo de la dirección de los pasos por intervalos de tiempo. Ahora cada nodo encargado de calcular la ruta espera a que el cliente avise que puede calcular, mediante la instrucción (Start) enviada a la red de comunicación, con el indicador (red).

Por otro lado, por la naturaleza del hardware, se encuentra que la velocidad de asignación de coordenadas para cada dron permite un comportamiento estable si se produce cada 0.5 segundos. Como se mejoró la resolución de la red en el subcapítulo 6.3, es necesario que se realice la ejecución, pero que cada diez pasos se comunique al dron en vuelo la coordenada objetivo. Se realiza lo anteriormente mencionado con el fin que el sistema no demore demasiado en reaccionar, pasando de 800 pasos, cada uno de 5mm a 80 pasos de 5cm, permitiendo recorrer el mapa en un tiempo cercano a 40 segundos. Se realiza de esta manera para poder conservar los ángulos generados en mejor resolución, ya que si se ejecuta el código de entrenamiento inicial, con la resolución para el cálculo de trayectorias (5cmm por paso en apenas cuatro direcciones), el sistema pierde precisión, la cual es muy importante por las constantes oscilaciones del dron en vuelo.

5.4.2. Obtención de las rutas reales

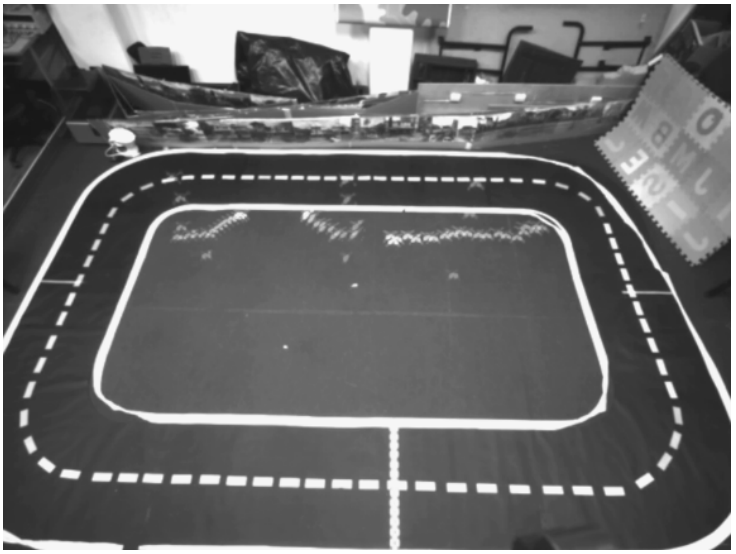
La forma en que se desarrolla la validación del sistema y se obtienen las características específicas del mismo se realizó mediante el diseño de cinco escenarios. El primer escenario consta de revisar cómo funciona el sistema al agruparse, el segundo y el tercero se implementan para revisar el paso de los drones por todo el espacio de coordenadas. El cuarto escenario se realiza para evidenciar cómo el sistema puede organizarse y agruparse en una zona reducida, permitiendo evidenciar la forma en que se evitan los choques mediante el algoritmo. Finalmente, el quinto cumple con revisar cómo se comporta el enjambre al seleccionar el mayor número de rutas estimado que debe cubrir.

Para el presente documento se revisó la manera de constatar el movimiento de los drones y se encontró interesante realizar las grabaciones de los episodios en ejecución, para posteriormente implementar el software Adobe After Effects, y mediante el efecto de eco ver cómo se producían las rutas, viendo que el dron se mueve desde la estela menos luminosa al punto con más luz.



(A) Entorno de trabajo en After Effects. Software utilizado para la obtención de las rutas

(B) Inicio del proceso de identificación de las rutas mediante ecos



(C) Pasos intermedios, donde se revisan que los pasos más antiguos tienen menos contraste que los nuevos

FIGURA 44: Revisión del proceso de obtención de las imágenes de las rutas de los drones.

Capítulo 6

Resultados y Discusión

6.1. Asignación de movimientos y estimación de vuelo

En este apartado se revisa el cómo el sistema de posicionamiento siguió el vuelo real que fue asignado al dron mediante el código dispuesto para un vuelo de prueba.

```
max_xy = 0.7 # valor máximo medido (70cm)
sequence = [(0.0, 0.0, 1), # coord. a seguir
            (max_xy , 0.0, 2),
            (max_xy , max_xy , 1.5),
            (-max_xy , max_xy , 1),
            (-max_xy , -max_xy , 1.5),
            (max_xy , -max_xy , 1),
            (0.0, 0.0, 0.5)]
```

FIGURA 45: Secuencia para estimación de movimientos

Se elige esta secuencia con la finalidad de cubrir el espacio sobre el cual se dará el vuelo, en la variable `sequence` se encuentran las coordenadas en el orden (x, y, z). en principio, mediante la revisión del comportamiento se estima que el radio frontera máximo, sobre el cual el dron puede volar es un metro, sin embargo, un máximo seguro es 70cm.

FIGURA 46: Dron en vuelo

Las coordenadas objetivo son comunicadas desde el ordenador hacia el dron, mediante el dispositivo de radio conectado a este, las mismas son estimadas mediante el sistema Lighthouse.

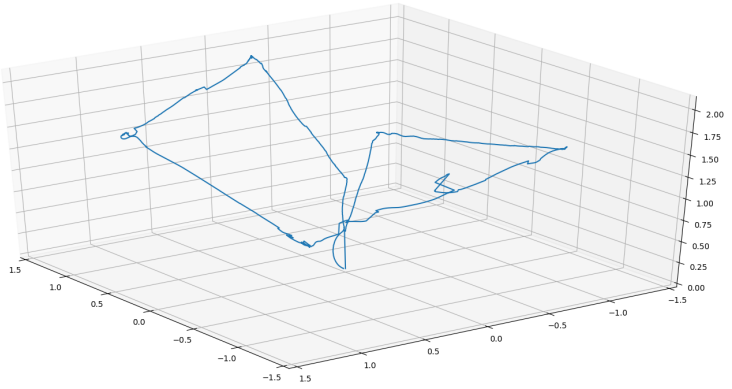


FIGURA 47: Vuelo programado, perspectiva en 3D

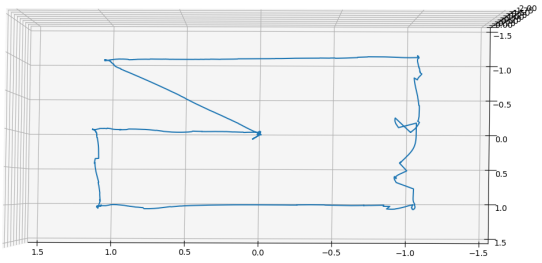


FIGURA 48: Movimiento en los ejes (x) y (y)

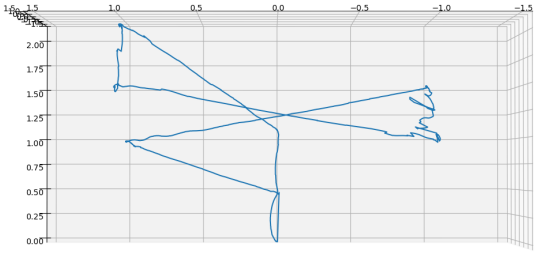


FIGURA 49: Movimiento en los ejes (x) y (z)

Como se puede observar en la figura 45 a 47, el paso entre la coordenada (siguiendo el orden [x,y,z]) [-1,1,1] a [-1,-1,1.5] presenta bastantes perturbaciones, esto es traducido como movimientos erráticos del dron por salir de la zona de detección, por todo lo demás, el movimiento es cercano a líneas rectas entre las coordenadas objetivo.

6.2. Convergencia del modelo entrenado

En este apartado es posible ver los resultados del código implementado para el entrenamiento y prueba de Q-learning. Los resultados son los mensajes en consola, los mismos evidancian las fases del entrenamiento y sus resultados, mediante ellos es posible ver la evolución del agente de prueba en los escenarios propuestos.

6.2.1. Primera fase

En la primera fase se revisa que, entre el episodio 0 y el 20 (figura 50A), el agente solo llega una vez a la coordenada objetivo asignada al azar, esto es debido a que todas sus acciones o la mayoría son aleatorias, es importante notar el decaimiento de la tasa de exploración (E) y el aumento de la tasa de descuento (R). Los efectos de la primera fase de entrenamiento se pueden revisar en la figura 50B, Es importante recalcar que el parámetro (Llegó) indica las veces que ha llegado el agente a su objetivo en el transcurso de todo el entrenamiento, con este calculamos la efectividad del agente por cada fase. El ítem (Pasos) indica la cantidad de pasos que dió el agente el episodio actual, 160 es el máximo de pasos a dar, generalmente cuando el agente toma este número de pasos es debido a que no llegó al objetivo.

```
Episodio:0, Llegó:0, E:1.0, A:1, R:0.3, Pasos:160
Episodio:1, Llegó:0, E:1.0, A:1, R:0.3, Pasos:160
Episodio:2, Llegó:0, E:1.0, A:1, R:0.3, Pasos:160
Episodio:3, Llegó:0, E:1.0, A:1, R:0.31, Pasos:160
Episodio:4, Llegó:0, E:1.0, A:1, R:0.31, Pasos:160
Episodio:5, Llegó:0, E:0.99, A:1, R:0.31, Pasos:160
Episodio:6, Llegó:0, E:0.99, A:1, R:0.31, Pasos:160
Episodio:7, Llegó:0, E:0.99, A:1, R:0.31, Pasos:160
Episodio:8, Llegó:0, E:0.99, A:1, R:0.31, Pasos:160
Episodio:9, Llegó:0, E:0.99, A:1, R:0.31, Pasos:160
Episodio:10, Llegó:0, E:0.99, A:1, R:0.31, Pasos:160
Episodio:11, Llegó:0, E:0.99, A:1, R:0.32, Pasos:160
Episodio:12, Llegó:0, E:0.99, A:1, R:0.32, Pasos:160
Episodio:13, Llegó:0, E:0.99, A:1, R:0.32, Pasos:160
Episodio:14, Llegó:0, E:0.99, A:1, R:0.32, Pasos:160
Episodio:15, Llegó:0, E:0.99, A:1, R:0.32, Pasos:160
Episodio:16, Llegó:0, E:0.98, A:1, R:0.32, Pasos:160
Episodio:17, Llegó:0, E:0.98, A:1, R:0.32, Pasos:160
Episodio:18, Llegó:0, E:0.98, A:1, R:0.33, Pasos:160
Episodio:19, Llegó:0, E:0.98, A:1, R:0.33, Pasos:160
Episodio:20, Llegó:1, E:0.98, A:1, R:0.33, Pasos:113
```

(A) Episodio 0 a 20.

```
Episodio:484, Llegó:188, E:0.55, A:1, R:0.95, Pasos:131
Episodio:485, Llegó:189, E:0.55, A:1, R:0.95, Pasos:150
Episodio:486, Llegó:189, E:0.55, A:1, R:0.95, Pasos:160
Episodio:487, Llegó:189, E:0.54, A:1, R:0.95, Pasos:160
Episodio:488, Llegó:190, E:0.54, A:1, R:0.95, Pasos:28
Episodio:489, Llegó:191, E:0.54, A:1, R:0.95, Pasos:32
Episodio:490, Llegó:192, E:0.54, A:1, R:0.95, Pasos:63
Episodio:491, Llegó:193, E:0.54, A:1, R:0.96, Pasos:125
Episodio:492, Llegó:194, E:0.54, A:1, R:0.96, Pasos:60
Episodio:493, Llegó:194, E:0.54, A:1, R:0.96, Pasos:160
Episodio:494, Llegó:194, E:0.54, A:1, R:0.96, Pasos:160
Episodio:495, Llegó:194, E:0.54, A:1, R:0.96, Pasos:160
Episodio:496, Llegó:195, E:0.54, A:1, R:0.96, Pasos:152
Episodio:497, Llegó:195, E:0.54, A:1, R:0.96, Pasos:160
Episodio:498, Llegó:196, E:0.53, A:1, R:0.97, Pasos:68
Episodio:499, Llegó:197, E:0.53, A:1, R:0.97, Pasos:108
Episodio:500, Llegó:198, E:0.53, A:1.0, R:0.96, Pasos:135
```

(B) Episodio 484 a 500.

FIGURA 50: Resultados de la primera fase de entrenamiento de Q-learning, imágenes tomadas de los comentarios en consola producidos por el código desarrollado

En esta fase, de quinientos episodios, solo en 198 la elección de la ruta fue exitosa, es decir, una efectividad del 39 %, tomada de los episodios revisados en la figura 50, calculada como el total de episodios, sobre los episodios efectivos, expresada en porcentaje.

6.2.2. Segunda fase

En la fase dos, se puede evidenciar en la figura 51 el decaimiento de todas las tasas, lo que representa que el aprendizaje pasó de ser central (en la primera fase), a perder importancia con el paso de los episodios. Por otro lado, de 500 episodios transcurridos, el agente llegó 474 veces (restando al total de veces que llegó hasta el episodio 999, las veces que llegó en la fase uno), eso quiere decir, una efectividad del 94.8 %.

```
Episodio:979, Llegó:654, E:0.09, A:0.14, R:0.1, Pasos:160
Episodio:980, Llegó:655, E:0.08, A:0.13, R:0.1, Pasos:33
Episodio:981, Llegó:656, E:0.08, A:0.13, R:0.1, Pasos:6
Episodio:982, Llegó:657, E:0.08, A:0.13, R:0.1, Pasos:59
Episodio:983, Llegó:658, E:0.08, A:0.13, R:0.1, Pasos:22
Episodio:984, Llegó:659, E:0.08, A:0.13, R:0.09, Pasos:57
Episodio:985, Llegó:660, E:0.08, A:0.13, R:0.09, Pasos:30
Episodio:986, Llegó:661, E:0.08, A:0.12, R:0.09, Pasos:34
Episodio:987, Llegó:662, E:0.08, A:0.12, R:0.09, Pasos:29
Episodio:988, Llegó:663, E:0.08, A:0.12, R:0.09, Pasos:37
Episodio:989, Llegó:664, E:0.08, A:0.12, R:0.08, Pasos:10
Episodio:990, Llegó:664, E:0.08, A:0.12, R:0.08, Pasos:160
Episodio:991, Llegó:665, E:0.07, A:0.11, R:0.08, Pasos:65
Episodio:992, Llegó:666, E:0.07, A:0.11, R:0.08, Pasos:21
Episodio:993, Llegó:667, E:0.07, A:0.11, R:0.08, Pasos:10
Episodio:994, Llegó:667, E:0.07, A:0.11, R:0.08, Pasos:160
Episodio:995, Llegó:668, E:0.07, A:0.11, R:0.07, Pasos:49
Episodio:996, Llegó:669, E:0.07, A:0.11, R:0.07, Pasos:63
Episodio:997, Llegó:670, E:0.07, A:0.1, R:0.07, Pasos:73
Episodio:998, Llegó:671, E:0.07, A:0.1, R:0.07, Pasos:21
Episodio:999, Llegó:672, E:0.07, A:0.1, R:0.07, Pasos:26
```

FIGURA 51: Episodio 979 a 999, parte final de la segunda fase

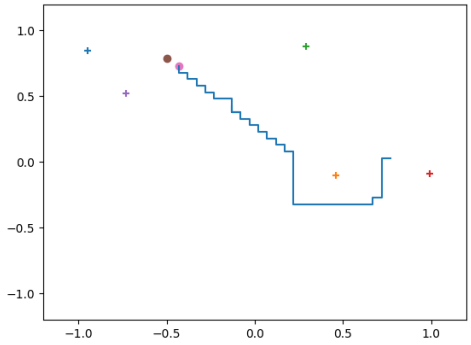
6.2.3. Tercera fase

Para finalizar, en la fase tres, donde los parámetros de aprendizaje no son modificados y tienen poca importancia, revisada en la figura 52, se tiene un resultado en el Episodio final (1499), de 482 veces que el agente llega al objetivo, de 500 totales, con una efectividad del 96.4 % y además apenas 15 choques en esta misma cantidad de episodios, es decir, una probabilidad de choque del 3 %. Es importante tener en cuenta que el ítem (Choques) indica cuantas veces se han producido choques solo en la tercera fase, en este caso, la probabilidad obtenida es el resultado de dividir el total de pasos de la fase (500) por la cantidad de choques ocurridos (15).

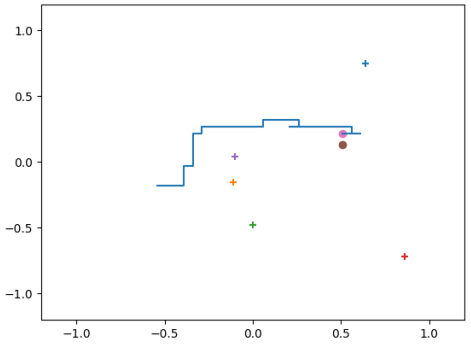
```
Episodio:1479, Llegó:1135, Entrenado:462, Pasos:160, Choques:14
Episodio:1480, Llegó:1136, Entrenado:463, Pasos:25, Choques:14
Episodio:1481, Llegó:1137, Entrenado:464, Pasos:24, Choques:14
Episodio:1482, Llegó:1138, Entrenado:465, Pasos:25, Choques:14
Episodio:1483, Llegó:1139, Entrenado:466, Pasos:7, Choques:14
Episodio:1484, Llegó:1140, Entrenado:467, Pasos:45, Choques:14
Episodio:1485, Llegó:1141, Entrenado:468, Pasos:23, Choques:14
Episodio:1486, Llegó:1142, Entrenado:469, Pasos:51, Choques:14
Episodio:1487, Llegó:1143, Entrenado:470, Pasos:45, Choques:14
Episodio:1488, Llegó:1144, Entrenado:471, Pasos:32, Choques:14
Episodio:1489, Llegó:1145, Entrenado:472, Pasos:67, Choques:14
Episodio:1490, Llegó:1146, Entrenado:473, Pasos:8, Choques:14
Episodio:1491, Llegó:1147, Entrenado:474, Pasos:57, Choques:14
Episodio:1492, Llegó:1148, Entrenado:475, Pasos:18, Choques:14
Episodio:1493, Llegó:1149, Entrenado:476, Pasos:37, Choques:14
Episodio:1494, Llegó:1150, Entrenado:477, Pasos:39, Choques:14
Episodio:1495, Llegó:1151, Entrenado:478, Pasos:19, Choques:14
Episodio:1496, Llegó:1152, Entrenado:479, Pasos:9, Choques:15
Episodio:1497, Llegó:1153, Entrenado:480, Pasos:21, Choques:15
Episodio:1498, Llegó:1154, Entrenado:481, Pasos:9, Choques:15
Episodio:1499, Llegó:1155, Entrenado:482, Pasos:7, Choques:15
Saving Qlearning table
Estados visitados: 256
```

FIGURA 52: Episodio 1479 a 1499

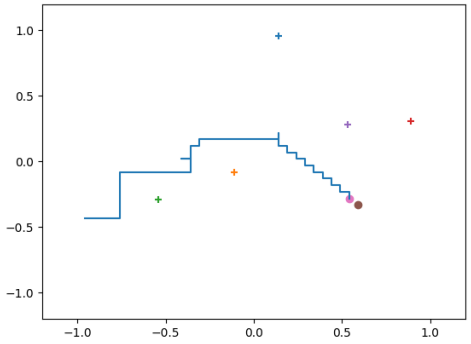
Además de todo lo mencionado, es importante recalcar que en todo el proceso, el agente estuvo en 256 estados, que son todos los posibles. En la figura 53 se pueden revisar diferentes gráficas del comportamiento del agente en esta fase. Como se puede ver, el agente (burbuja lila) se dirige hacia su objetivo (burbuja café), y trata de evitar a sus compañeros (cruces), además se evidencia en la figura 53 la ruta del agente (línea azul).



(A) Primer escenario virtual individual.



(B) Segundo escenario virtual.



(C) Tercer escenario virtual.

FIGURA 53: Resultados gráficos del entrenamiento del algoritmo.

6.2.4. Tabla Q

Finalmente se obtiene una tabla Q que puede ser implementada para cada dron en el enjambre de drones, pudiendo así lograr el comportamiento de partículas (PSO) que se requiere en este proyecto. Por lo extensa que es (256x4 datos), no se muestra en el documento.

Recordando que los estados vienen definidos como se puede revisar en el cuadro 3 consignado en el desarrollo conceptual, en la parte de la conformación de la tabla Q (Subcapítulo 5.2.1), es posible interpretar la información de cada caso de la siguiente manera:

Índice	Estado	Adelante	Atrás	Derecha	Izquierda
40	00101000	-0,2248	-0,1871	-0,1955	-0,1851
181	10110101	0,2141	-0,2536	-0,1394	0,1026
25	00011001	-0,0191	-0,0362	0,2128	-0,2844

CUADRO 3: Revisión de tres estados diferentes entre los 256 posibles, en la tabla Q generada, luego del entrenamiento y prueba.

El estado número 40, cuyo número binario es 00101000, significa que el agente se encuentra en un estado de: (00) objetivo adelante, (10) la distancia con el objetivo es entre 0.5 y 0.7 metros, (10) compañero más cercano está a la derecha, (00) a menos de 0.2 metros. Para este estado se revisa que la mejor opción es dirigirse hacia la izquierda (mayor valor). todos los estados son negativos ya que esta posición representa un alto riesgo de choque y por lo tanto es un estado que se quiere evitar.

El estado número 181, cuyo número binario es 10110101, representa que el agente (10) tiene a su objetivo a la derecha, (11) su distancia con el objetivo es mayor a 0.7, (01) su compañero más cercano está hacia atrás y (01) se encuentra entre 0.2 y 0.5 metros de distancia. La mejor opción que puede tomar según la tabla es dirigirse hacia adelante, es adecuado ya que se aleja del compañero más cercano.

El estado número 251, cuyo valor binario es 00011001, representa que el agente se encuentra en un estado de: (00) el objetivo está en frente, (01) se encuentra entre 0.2 y 0.5 metros, (10) su compañero más cercano está a su derecha y (01) se encuentra entre 0.2 y 0.5 metros. La mejor opción posible es moverse a la derecha, si bien se aproxima a su compañero, no se encuentra aún en riesgo de choque, este estado hace parte de una serie de estados de transición donde la opción real no es muy clara y el algoritmo puede llegar a confundirse.

6.3. Interacción de diferentes agentes virtuales con el modelo

Para este apartado se desarrolló un código que permite la interacción de los agentes en el entorno virtual desarrollado para el llenado de la tabla Q. A grandes rasgos, se modificaron la cantidad de pasos (de 160 a 800) y el valor del paso(de 50cm a 5cm) con el fin de producir un comportamiento más preciso, este cambio es pertinente ya que el algoritmo funcionaba correctamente, pero a veces se podían producir choques porque la distancia de los pasos dados no era precisa.

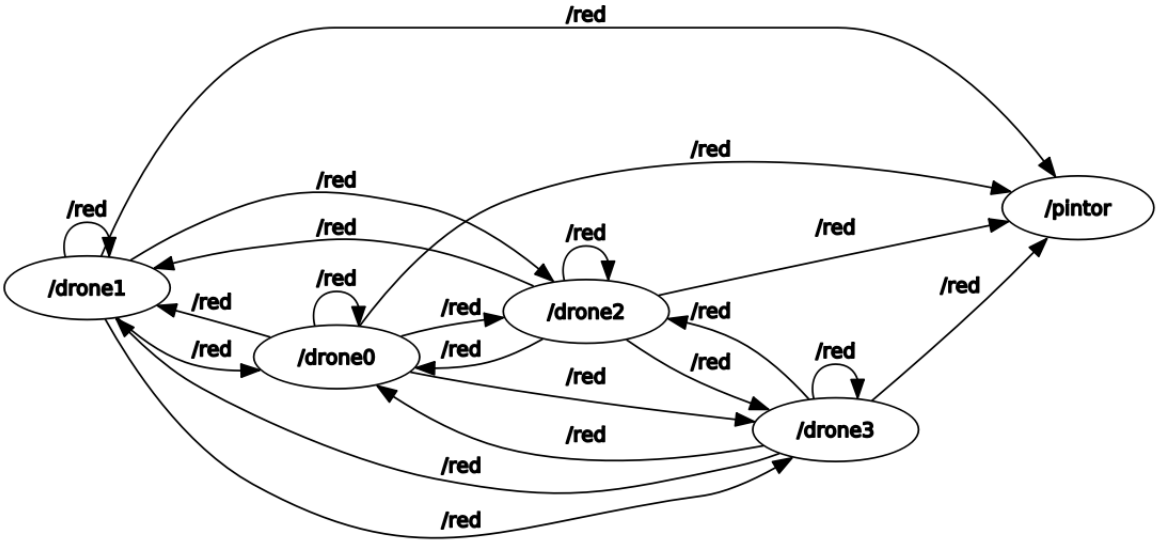


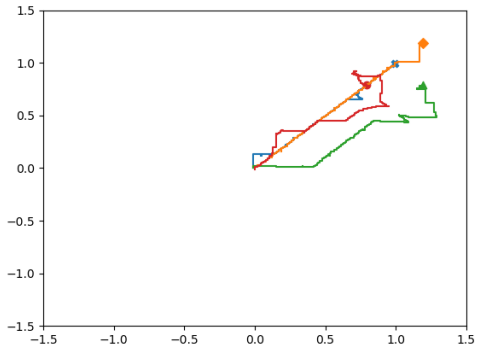
FIGURA 54: Red generada en ROS para cuatro drones, tomada del desarrollo presentado, obtenida mediante el comando en terminal: (rosrun rqt_graph rqt_graph).

La red generada en la figura 54 tiene como nodo central el dispuesto en dron0.py, este es el líder, el mismo se comunica con todos los demás nodos de drones para coordinarlos. Los nodos de los drones compañeros, se encargan de contestar al nodo principal y recibir instrucciones, tal y como se planteó en el desarrollo conceptual asociado a este apartado. Finalmente, todos los nodos envían sus coordenadas al nodo pintor.py, que no tiene respuesta alguna en la red.

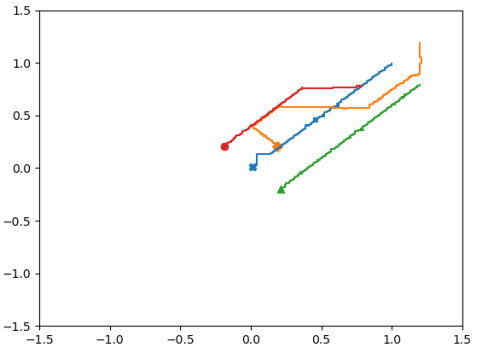
```
dsg@dsg-IdeaPad-3-14ADA6:~$ rostopic echo /red
WARNING: topic [/red] does not appear to be published yet
data: "red,0,Hola"
---
data: "red,1,Hola"
---
data: "red,3,Hola"
---
data: "red,2,Hola"
---
data: "obj,1, 1.2,1.2,1.0"
---
data: "coord,1,-0.01,0.0,0.1"
---
data: "coord,1,-0.01,0.01,0.1"
---
data: "coord,1,-0.01,0.02,0.1"
---
data: "coord,1,0.0,0.02,0.1"
---
data: "coord,1,0.01,0.02,0.1"
---
data: "coord,1,0.02,0.02,0.1"
---
```

FIGURA 55: Mensajes publicados a la red de ROS

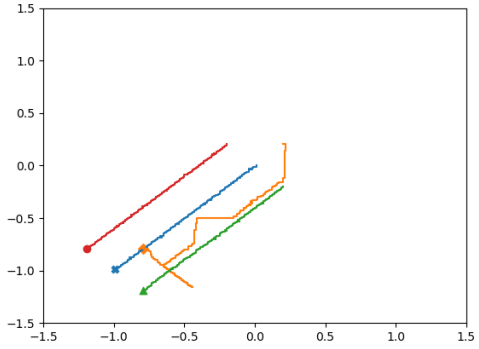
En la figura 55, se puede evidenciar el cómo se envían los mensajes a la red. En principio se produce una fase de saludo, que es para verificar la disponibilidad de los drones (todos envían un mensaje: red, ID, Hola), luego todos empiezan a comunicar sus coordenadas (coord,ID,x,y,z), para que, en breves lapsos de tiempo, el líder asigne las coordenadas objetivo a cada uno de los compañeros (obj, ID destino, x, y, z).



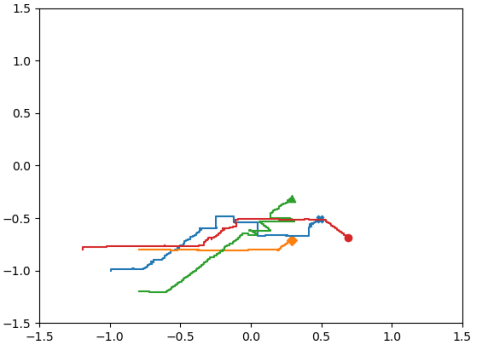
(A) Primer escenario virtual del enjambre.



(B) Segundo escenario virtual.



(C) Tercer escenario virtual.



(D) Cuarto escenario virtual.

FIGURA 56: Comportamiento del conjunto de drones en el ambiente virtual

Como se puede revisar en la figura 56, en los vuelos ejecutados las trayectorias se cruzan, pero el indicador de choque consignado en el `pintor.py` nunca imprime la indicación de choque en la terminal, sin embargo, en la implementación real es posible ver este efecto.

6.4. Vuelo en conjunto y revisión de las rutas individuales

En este subcapítulo se presentan los diferentes escenarios dispuestos para la revisión del comportamiento del sistema y sus características particulares. Es posible encontrar los videos de

los escenarios mediante el siguiente enlace: `Pruebas_de_vuelo`.

A continuación, se podrá ver mediante imágenes los ecos de las rutas que tomaron los drones mediante el método explicado en el desarrollo conceptual. El posicionamiento de los drones gira en torno a cuatro cuadrantes:

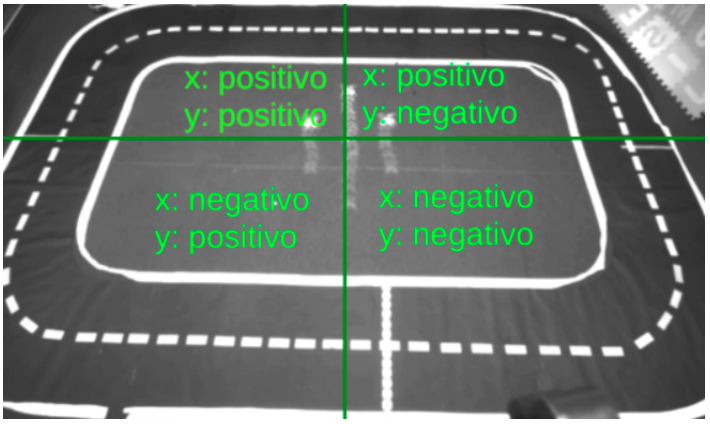


FIGURA 57: Cuadrantes de coordenadas

6.4.1. Escenario 1

El escenario 1 está dispuesto para revisar el movimiento del enjambre cuando se requiere agrupar drones dispersos en diferentes posiciones, en este caso se tiene que la disposición inicial de los drones es:

Dron	**Coordenada inicial (x,y) [metros]**
ID=0	(0.0 , -1.0)
ID=1	(0.0, 1.0)
ID=2	(0.0, 0.0)

CUADRO 4: Coordenadas iniciales, primer escenario real

FIGURA 58: Primer escenario real

El objetivo de este escenario es que el enjambre pueda trasladarse a la coordenada (x=5.0, y=5.0). Nótese que la dirección del dron viene dada por la intensidad de luz, la misma disminuye con el número de pasos dado.

6.4.2. Escenarios 2 y 3

Los escenarios mencionados permiten ver cómo se comporta el sistema cuando se parte de un punto alejado del origen, se recorre todo el mapa, y se solicita volver al origen.

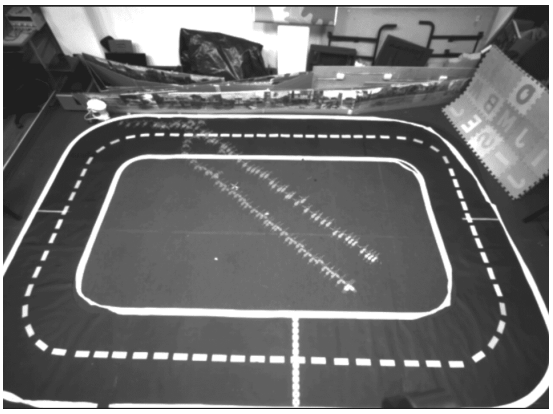
6.4.2.1. Escenario 2

Las coordenadas iniciales del segundo escenario son:

Dron	**Coordenada inicial (x,y) [metros]**
ID=0	(0.65 , 0.5)
ID=1	(0.65 , 0.7)
ID=2	(0.65 , 1.3)

CUADRO 5: Coordenadas iniciales, segundo escenario real.

El objetivo del vuelo es dirigirse a las coordenadas (x=-0.7, -0.7) en la primera ruta, y en la segunda dirigirse al centro del sistema de coordenadas.



(A) Segundo escenario real, primera ruta.

(B) Segundo escenario real, segunda ruta.

FIGURA 59: Escenario 2 real

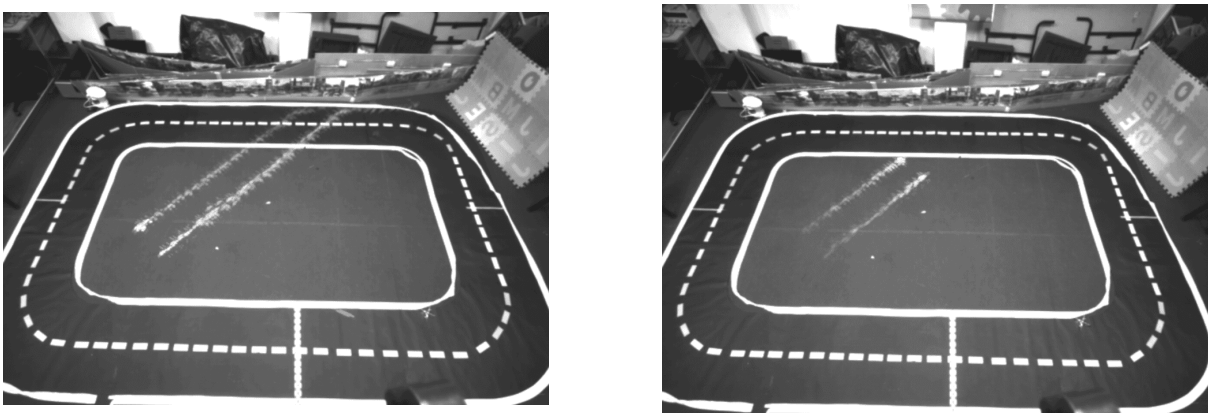
Como se puede observar, uno de los drones se desestabilizó y cayó en el paso entre el cuadrante positivo al negativo (dron con ID=1), sin embargo, dos de ellos pudieron continuar con el vuelo. Por otro lado, es interesante notar que las trayectorias de los drones se cruzan, pero no existen choques.

6.4.2.2. Escenario 3

Este escenario es una réplica del anterior, solo que se invierten las coordenadas tanto objetivo (x=-0.7, 0.7) como las iniciales.

Dron	**Coordenada inicial (x,y) [metros]**
ID=0	(0.65 , -1.3)
ID=1	(0.65 , -0.7)
ID=2	(0.65 , -0.5)

CUADRO 6: Coordenadas iniciales, tercer escenario real.



(A) Tercer escenario real, primera ruta.

(B) Tercer escenario real, segunda ruta.

FIGURA 60: Escenario 3 real

En este caso, también existe la pérdida de uno de los drones (nuevamente el dron con ID=1), cerca de la coordenada en el eje (y) donde previamente se había perdido. Además de lo dicho se revisa que el dron con ID=0 también se inestabiliza al pasar por esa zona. En este caso las rutas no se cruzan, dando un resultado similar a las revisadas en el subcapítulo 6.3, donde se realizan los ejemplos del enjambre virtual.

6.4.2.3. Zona de inestabilidad

Cada uno de los escenarios anteriores fue ejecutado dos veces y se revisa que hay una zona en el espacio de coordenadas donde los drones pierden su estabilidad, por ende fue necesario identificar las causas y la forma en que es posible eliminar factores de inestabilidad. Para esta fase de la implementación se identifica, mediante la repetición de vuelos de prueba, que hay tres factores clave en la inestabilidad del enjambre en torno al sistema de posicionamiento.

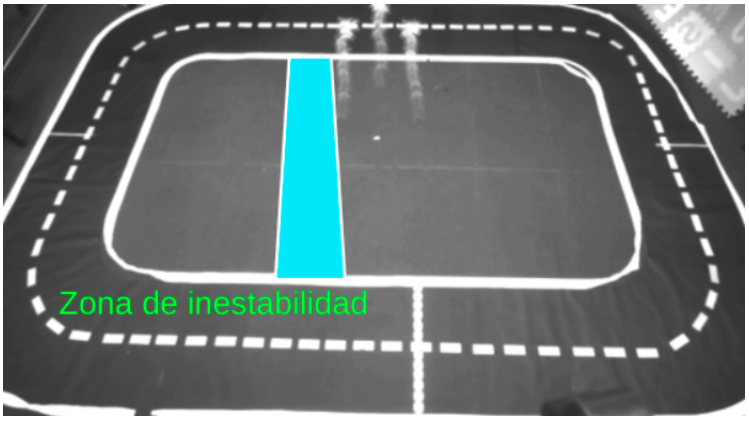


FIGURA 61: Zona de inestabilidad

El primer causante son otras fuentes de luz infrarroja. Como se pudo observar en la figura 61, la zona de inestabilidad fue identificada como un espacio que es afectado en su mayoría en torno a la coordenada (y) respecto a la porción de la coordenada (x), lo que deja entrever la dirección de la fuente del problema. En este caso se identifica que el causante fue uno de los robots que estaba siendo utilizado en el laboratorio de robótica de la universidad, específicamente el robot Pepper, que utiliza luz infrarroja para poder sensar la profundidad del espacio. El evitar fuentes de luz infrarroja, incluye los rayos de sol, ya que los mismos contienen un espectro amplio de frecuencias, donde están incluidos los rayos infrarrojos.

El segundo causante de inestabilidad es la interferencia entre los drones y el sistema lighthouse. Es necesario que para que el dron se pueda estabilizar de manera correcta, no haya un obstáculo entre las fuentes de luz de lighthouse y los sensores ópticos que tiene el dron, si el dron es capaz de captar solo una fuente de luz, o ninguna, al asignarle una coordenada objetivo, su comportamiento va a ser errático y va a chocar. Esta interferencia puede ser debida inclusive a la interacción de los drones en vuelo, identificando que cuando un dron pasa sobre otro, el que está abajo se desestabiliza y choca.

El tercer causante es la luz artificial emitida por algunos sistemas de iluminación. En el salón de robótica se llevaron a cabo pruebas con las luces apagadas y encendidas, y se evidenció que el sistema es más estable cuando la luz está apagada, cuando no, la variación del movimiento de los drones es mayor, lo que produce mayor probabilidad de choques.

Para el último escenario (consignado más adelante como escenario 6), se tomaron todas las precauciones mencionadas, junto con las revisadas a continuación, en el escenario 4.

6.4.3. Escenario 4

El escenario a continuación se implementa para la revisión del comportamiento del enjambre en espacios reducidos, permitiendo poner a prueba la capacidad del algoritmo para asignar coordenadas que eviten los choques.

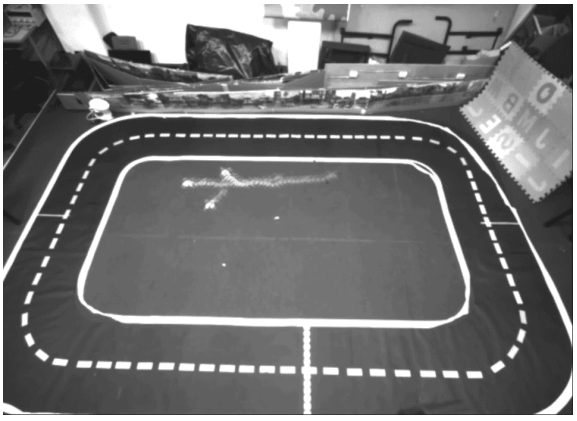
Dron	**Coordenada inicial (x,y) [metros]**
ID=0	(0.0 , 0.0)
ID=1	(0.0 , 0.4))
ID=2	(0.0 , -0.4)

CUADRO 7: Coordenadas iniciales, cuarto escenario real.

Ruta	Objetivo (x,y) [metros]
0	(0.0 , 0.5)
1	(0.5 , 0.5
2	(0.5 , -0.5)

CUADRO 8: Coordenadas objetivo, cuarto escenario real.

Como se puede observar en la figura 62, las rutas llegan a cruzarse, pero no se producen choques, sin embargo, por la naturaleza del sistema y la variación que a veces presentan los drones, si llegan a acercarse bastante. Otro punto importante a analizar es que los drones se organizan y tratan de encontrar un espacio entre ellos para evitar las colisiones, inclusive alejándose mutuamente.



(A) Cuarto escenario real, primera ruta.

(B) Cuarto escenario real, segunda ruta.

(C) Cuarto escenario real, tercera ruta.

FIGURA 62: Escenario 4 real

En este escenario se identifica que es necesario aumentar la distancia entre los drones con el fin de evitar los choques, ya que si bien el resultado del escenario fue satisfactorio, se corrió

el riesgo de producir choques. En los siguientes escenarios, específicamente el escenario 6 se corrigió este inconveniente.

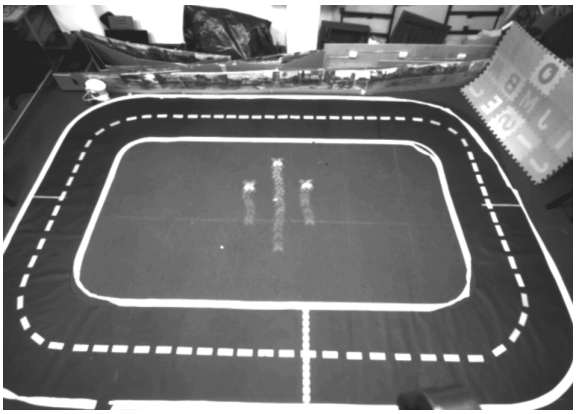
6.4.4. Escenario 5

En este escenario se pretende revisar si el sistema es capaz de cumplir con ejecutar cinco rutas por vuelo, y además verificar que la asignación de coordenadas por parte del líder es correcta. En este sentido se plantean vuelos que parten del origen, van a la coordenada objetivo y retornan al mismo. Puntualmente se hace para visualizar si existen cruces entre las rutas, en un escenario con rutas sencillas. Las coordenadas de partida son las mismas que en el escenario cuatro.

Ruta	**Objetivo (x,y) [metros]**
0	(-0.5 , 0.0)
1	(0.0 , 0.0)
2	(0.5 , 0.0)
3	(0.0 , 0.0)
4	(0.0 , -0.5)

CUADRO 9: Coordenadas objetivo, quinto escenario real.

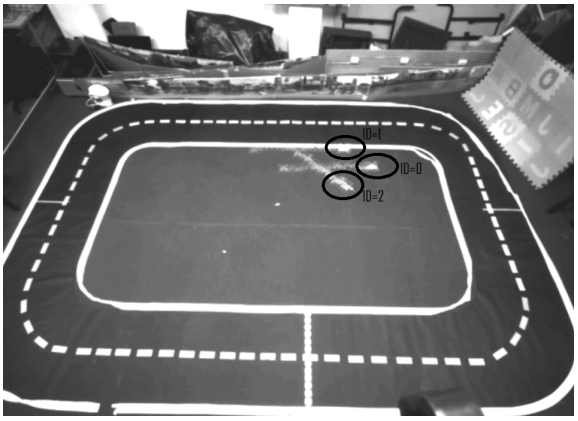
(A) Quinto escenario real, primera ruta.



(B) Quinto escenario real, segunda ruta.

(C) Quinto escenario real, tercera ruta.

(D) Cuarto escenario real, cuarta ruta.



(E) Cuarto escenario real, quinta ruta.

FIGURA 63: Escenario 5 real

Como se puede observar en la figura 63, desde la primera ruta, hasta la cuarta los drones trazaron líneas rectas. En la quinta ruta existió un cruce de rutas, sin embargo, sucedió porque tanto el dron con ID=2 siguió al dron con ID=1 mientras se posicionaba el dron con ID=0, de otra manera se hubiese producido un choque.

Otro punto importante a resaltar es que el dron líder, que siempre ha sido el dron con ID=0, siempre llega a la coordenada especificada, y los demás solo se acercan, siendo el primero quien cubrió más pasos en este escenario que los demás, esto no pasa en todos los escenarios, pero esto indica que siempre el deber de evadir a los compañeros lo tiene el dron que esté más lejos de su coordenada objetivo.

6.4.5. Escenario 6

El presente escenario se construye con la finalidad de encontrar solución a los problemas hallados en los anteriores escenarios. En principio se consideró trasladarse a otro espacio para evitar todas las posibles interferencias que pueden ser causadas por los equipos que se implementan en el laboratorio de robótica. Aunado a esto se buscó un espacio donde pudiera ser eliminada toda fuente de luz externa, para así eliminar los problemas de inestabilidad causados por estos dos factores, la disposición del escenario se encuentra en la figura a continuación.

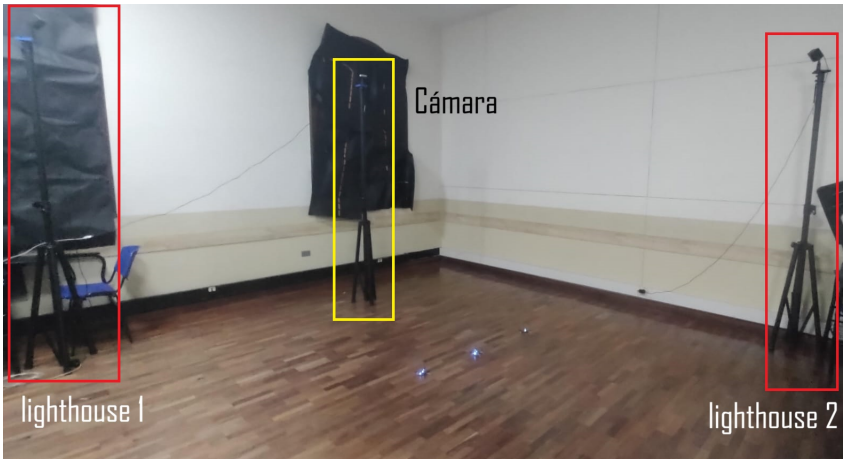


FIGURA 64: Espacio escenario 6

Se plantearon rutas para medir el comportamiento del enjambre en el desplazamiento diagonal y transversal, como se había hecho en los escenarios anteriores. Los drones fueron inicializados en las mismas coordenadas que las vistas en el escenario 5 y 4, y los objetivos que siguió el enjambre fueron los revisados en el cuadro 10.

Ruta	Objetivo (x,y) [metros]
0	(0.6 , 0.6)
1	(-0.6 , 0.6)
2	(0.6 , -0.6)

CUADRO 10: Coordenadas objetivo, sexto escenario real.

En el cumplimiento de la primera ruta se revisa que ahora la distancia en vuelo entre los drones es mucho mayor a la hora de reunirse, es importante recalcar que al aumentar dicha distancia, los drones tienden a trazar rutas mucho más abiertas que en los escenarios anteriores. Lo mencionado era otro punto clave al mejorar el desempeño del enjambre, se puede revisar en la primera ruta:

FIGURA 65: Sexto escenario real, primera ruta.

La expansión de la distancia entre los drones puede ser configurada cambiando el valor del estado de cercanía al dron en el código de Q-learning (en la función de definición de estados), antes estaba en 0.2 metros, ahora se configuró a 0.3. El cambio mencionado debe hacerse en cada uno de los nodos que representan a los drones, que componen la red desarrollada en ROS, para la coordinación del enjambre.

FIGURA 66: Sexto escenario real, segunda ruta.

Para la segunda ruta, se evidencia nuevamente que las rutas de los drones se cruzan, sin embargo, no se producen choques ya que los drones pasan por las mismas coordenadas en tiempos distintos.

FIGURA 67: Sexto escenario real, tercera ruta.

La expansión de la distancia entre los drones no es completamente fructífera ya que debido a la restricción impuesta, los drones tienden a tomar rutas muy alejadas del enjambre, lo que puede ocasionar que los mismos salgan del rango del espacio de coordenadas generado por el sistema de posicionamiento, perdiendo estabilidad. Un ejemplo de estas desviaciones innecesarias se revisa en la figura anterior.

Capítulo 7

Conclusiones y Trabajos Futuros

7.1. El sistema de posicionamiento para uso académico

En conclusión al trabajo realizado se estima que el sistema de posicionamiento y la implementación del hardware desarrollado por Bitcraze (Crazyflie, lighthouse position y sus complementos), tiene ventajas tales como la posibilidad de evaluar el rendimiento de los algoritmos desarrollados para el control de varios agentes.

Contrario a lo anteriormente expuesto, dicho hardware aún se encuentra en desarrollo y por lo mismo puede presentar inconvenientes. Entre ellos se tiene la imposibilidad de mezclar ROS con la comunicación entre el ordenador y los diferentes drones, la repetitiva reconfiguración del sistema de coordenadas por inexactitud de la estimación, y una necesidad de espacios aislados. La experiencia en el uso del hardware permite concluir, que el mismo se inestabiliza por interferencia de otros dispositivos, de la luz solar, e inclusive de los mismos drones entre sí. Esto no quiere decir que sea un mal sistema, solo que tiene características que lo hacen algo complejo de implementar y que requiere atención a los diferentes factores mencionados.

Como trabajo futuro se plantea la elaboración de un sistema de posicionamiento desarrollado específicamente para varios agentes móviles, que no sea sentralizado, sino que dependa de la perspectiva de cada agente, ya sea en la implementación de odometría o fotogrametría.

7.2. El desempeño de Q-learning

Se concluye que el algoritmo tiene ventajas excepcionales a la hora de funcionar en escenarios donde no es posible generar un set de instrucciones, sino que el comportamiento del agente depende de cada paso que da. Entre estas ventajas se considera que el algoritmo en su fase de entrenamiento y prueba es sencillo de implementar, tanto por la estructura para la codificación, como la posibilidad de configurar las tasas de aprendizaje y hacer pruebas en cortos periodos de tiempo, una ventaja clave respecto a las redes neuronales. Aunado a esto, se tiene que el entrenamiento de un solo modelo genera una tabla de políticas (tabla Q), que puede ser replicada para cada uno de los agentes a implementar, lo que lo hace ideal para sistemas descentralizados. Junto con lo anterior, también es posible ejecutar en un dispositivo el algoritmo para varios agentes, sin exigir un alto grado de procesamiento, y simular virtualmente un sistema compuesto de varios agentes como se revisó en el subcapítulo 6.3.

Una particularidad interesante que se revisa en este desarrollo, es que el entrenamiento se puede dar con un ambiente mucho menos complejo respecto al real. Como se pudo ver en el entrenamiento inicial, la resolución de los pasos era inferior respecto al utilizado para las pruebas del enjambre con ROS y en los vuelos realizados. Esto fue posible gracias a que la definición de los estados fue acertada (considerando todos los estados posibles respecto a posición y distancia), lo que permitió generar un entrenamiento en torno a la dinámica del agente (específicamente la asignación de recompensas, que solo se basaba en las diferencias entre el estado actual y el anterior), que era independiente a la distancia entre los pasos.

Como trabajo futuro se identifica que el algoritmo desarrollado puede adecuarse para la solución de problemas de movilidad en cualquier tipo de transporte, por ejemplo la movilidad de automóviles en ciudades con un gran flujo vehicular y restricciones. Haciendo especial énfasis en el evitar colisionar con elementos en el entorno que no se hayan agregado al sistema.

7.3. ROS y el modelo de comunicación

El uso de ROS en este trabajo de grado fue crucial ya que permitió generar toda una red de comunicación entre agentes virtuales, y revisar el comportamiento del enjambre antes de implementarlo físicamente. Una ventaja que resalta es que posibilita organizar la comunicación y los roles de los dispositivos de una manera sencilla y ordenada, donde podemos inclusive generar los diagramas de comunicación, revisados en 6.3.

Como trabajo futuro se considera que la estructura realizada puede ser implementada en diferentes ordenadores integrados en drones para labores de agricultura, teniendo en cuenta que solo deben cambiarse los puentes de comunicación entre los dispositivos, respecto al trabajo realizado.

7.4. Validación experimental

En torno a la implementación del enjambre, se considera válido el desarrollo en tanto el sistema de posicionamiento se encuentre configurado correctamente y no tenga interferencias. Por lo demás, tanto el aprendizaje de máquina efectuado, como la jerarquía desarrollada, permiten implementar un enjambre de drones funcional.

En los escenarios dispuestos se pudo revisar que la asignación de objetivos por parte del líder permite evitar que las rutas de los drones se crucen (escenario 5). Aunado a esto, cuando alguno de los drones precisa acercarse a otro, el mismo tiende a tomar una ruta que le permita rodearlo y además, cuando los drones se acercan y no han llegado a su objetivo, ambos tienden a separarse (revisado en el escenario 4). Junto a lo dicho, también se revisó que el enjambre puede ser inicializado en diferentes puntos del sistema de coordenadas y reunirse (especialmente revisado en el escenario 1). Además, a pesar de que alguno de los drones deje de funcionar o se pierda, los demás van a continuar con su recorrido (en referencia al escenario 2 y 3).

Una consideración negativa respecto al desarrollo es que por la naturaleza del sistema de posicionamiento, la asignación de coordenadas debe hacerse con un tiempo mínimo de 500ms, lo que hace que, si uno de los drones debe dar 80 pasos o más, el enjambre se demore un tiempo considerable en organizarse.

Se considera como trabajo futuro la implementación del sistema desarrollado en un espacio abierto, con aeronaves que permitan estimar las coordenadas de forma descentralizada, además donde pueda ser de ayuda específicamente para el sector agrícola, ya sea para aspersión o para control de plagas. Considerando todos los obstáculos físicos que el entorno puede tener. Esto requeriría el uso de todos los desarrollos futuros comentados en los anteriores subcapítulos.

Bibliografía

[1] A. Al-Kaff, D. Martin, F. Garcia, A. de la Escalera y J. M. Armingol, «Survey of computer vision algorithms and applications for unmanned aerial vehicles», *Expert Systems with Applications*, vol. 92, págs. 447-463, 2018.

[2] A. Brandstätter, S. A. Smolka, S. D. Stoller, A. Tiwari y R. Grosu, «Multi-Agent Spatial Predictive Control with Application to Drone Flocking (Extended Version)», *arXiv preprint arXiv:2203.16960*, 2022.

[3] Y. Wang, Z. Cheng y M. Xiao, «UAVs' formation keeping control based on multi–agent system consensus», *IEEE Access*, vol. 8, págs. 49 000-49 012, 2020.

[4] R. Zahıínos Marıín, «Sistema de coordinación y control de múltiples vehıículos aéreos no tripulados en testbed de interiores», 2020.

[5] P. Radanliev y D. De Roure, «Review of algorithms for artificial intelligence on low memory devices», *IEEE Access*, vol. 9, págs. 109 986-109 993, 2021.

[6] W. Hoenig, C. Milanes, L. Scaria, T. Phan, M. Bolas y N. Ayanian, «Mixed reality for robotics», en *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2015, págs. 5382-5387.

[7] M. Toboso-Martıín y M. A. Grau Ruiz, «Vivir con robots. Reflexiones éticas, jurıídicas, sociales y culturales», 2021.

[8] C. Ju y H. I. Son, «Multiple UAV systems for agricultural applications: Control, implementation, and evaluation», *Electronics*, vol. 7, n.º 9, pág. 162, 2018.

[9] D. A. Martínez, E. Mojica-Nava, K. Watson y T. Usländer, «Multiagent Self-Redundancy Identification and Tuned Greedy-Exploration», *IEEE Transactions on Cybernetics*, vol. 52, n.º 7, págs. 5744-5755, 2022. DOI: 10.1109/TCYB.2020.3035783.

[10] D. J. Silva Amador, «Análisis de la utilización de Drones como técnica de fumigación de cultivos de banano en el corregimiento de Orihueca, Zona Bananera, Colombia», 2021.

[11] M. G. Balestreri y F. Falck, «De ficción a realidad: drones y seguridad ciudadana en América Latina», *Ciencia y poder aéreo*, vol. 10, n.º 1, págs. 71-84, 2015.

[12] M. Rieckmann, *Educación para los Objetivos de Desarrollo Sostenible: objetivos de aprendizaje*. UNESCO Publishing, 2017.

[13] L. F. C. Chica, B. C. L. Zambrano y D. D. C. Rivadeneira, «Desafiíos, tendencias, retos y oportunidades de la educación en Latinoamérica», *Domino de las Ciencias*, vol. 9, n.º 3, págs. 231-238, 2023.

[14] D. J. Molano Garcıía, «La robótica educativa: una interdisciplina didáctica integradora para la enseñanza»,

[15] P. Watkins Christopher J. C. H. Dayan, «Q-learning», *Springer*, vol. 8, 1992. DOI: 10.1007/BF00992698.

[16] B. Jang, M. Kim, G. Harerimana y J. W. Kim, «Q-Learning Algorithms: A Comprehensive Classification and Applications», *IEEE Access*, vol. 7, págs. 133 653-133 667, 2019. DOI: 10.1109/ACCESS.2019.2941229.

[17] N. D. Gómez Garzón y N. H. Peña Castro, «Generación de Movimientos Coordinados de Enjambre en Múltiples Drones a través de Algoritmos de Aprendizaje Profundo»,

[18] L. Joseph, *Robot operating system (ros) for absolute beginners*. Springer, 2018.

[19] C. Escribano Garcıía-Machın, «Leader-Follower Decentraliced Control of a Nanoquadrotor Swarm», 2019.

[20] Bitcraze, *Crazyflie python library*, 2006. dirección: https://github.com/bitcraze/crazyflie-lib-python.

[21] A. Taffanel, B. Rousselot, J. Danielsson et al., «Lighthouse Positioning System: Dataset, Accuracy, and Precision for UAV Research», *arXiv preprint arXiv:2104.11523*, 2021.

[22] B. Tanyeri, Z. U. Bayrak y U. U. UÇAR, «The Experimental Study of Attitude Stabilization Control for Programmable Nano Quadcopter», *Journal of Aviation*, vol. 6, n.º 1, págs. 1-11, 2022.

[23] D. Martínez y E. Mojica-Nava, «Distortion based potential game for distributed coverage control», *Information Sciences*, vol. 600, págs. 209-225, 2022, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2022.03.090. dirección: https://www.sciencedirect.com/science/article/pii/S0020025522003176.

[24] S. J. Russell, *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[25] M. Rothmann y M. Porrmann, «A Survey of Domain-Specific Architectures for Reinforcement Learning», *IEEE Access*, vol. 10, págs. 13 753-13 767, 2022. DOI: 10.1109/ACCESS.2022.3146518.

[26] F. Brunner, «Mastering the game of Go with deep neural networks and tree search (Silver et al., 2016)», 2019.

[27] A. Gupta, P. P. Roy y V. Dutt, «Evaluation of Instance-Based Learning and Q-Learning Algorithms in Dynamic Environments», *IEEE Access*, vol. 9, págs. 138 775-138 790, 2021. DOI: 10.1109/ACCESS.2021.3117855.

[28] N. Habib, *Hands-On Q-Learning with Python: Practical Q-learning with OpenAI Gym, Keras, and TensorFlow*. Packt Publishing Ltd, 2019.

[29] G. Cardona, C. Bravo, W. Quesada et al., «Autonomous navigation for exploration of unknown environments and collision avoidance in mobile robots using reinforcement learning», en *2019 SoutheastCon*, IEEE, 2019, págs. 1-7.

[30] Á. Ramıírez-Linarez, M. Torres-Rivera et al., «Sistema de visión artificial y vuelo autónomo para un cuadricóptero en ros 2», *Pädi Boletín Científico de Ciencias Básicas e Ingenierías del ICBI*, vol. 10, n.º Especial6, págs. 33-41, 2022.

[31] A. Martinez y E. Fernndez, *Learning ROS for robotics programming*. Packt Publishing, 2013.

[32] F. Duan, W. Li e Y. Tan, *Intelligent Robot: Implementation and Applications*. Springer Nature, 2023.

[33] K. Richardsson, *cf_lib*, 2015. dirección: https://github.com/bitcraze/crazyflie-lib-python.

[34] K. Richardsson y K. says: *Improved Lighthouse Geometry Estimation*. dirección: https://www.bitcraze.io/2022/01/improved-lighthouse-geometry-estimation/.

[35] J. Hunt, «Sockets in python», en *Advanced Guide to Python 3 Programming*, Springer, 2023, págs. 557-569.