



ESCUELA DE INGENIERÍA DE FUENLABRADA

GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

TRABAJO FIN DE GRADO

Detección de transmisores radiofrecuencia con drones
utilizando aprendizaje por refuerzo.

Autor: Cristian Sánchez Rodríguez

Tutor: Dr. Roberto Calvo Palomino

Curso académico 2022/2023

Agradecimientos

ToDo

Madrid, 30 de junio de 2023

Cristian Sánchez Rodríguez

Resumen

En la actualidad, la ciencia ha avanzado a pasos agigantados con respecto a las soluciones tecnológicas. Especialmente la robótica, también gracias a que abarca una inmensa variedad de campos donde se pueden desarrollar soluciones eficientes y robustas.

Además, ha surgido un nuevo paradigma con el uso de drones, o sistemas aéreos provistos de sensores y actuadores, que amplian el espectro de uso para herramientas tecnológicas, permitiendo abordar los problemas desde nuevas perspectivas. En este proyecto, el foco de estudio se centra en los *Unmanned Air Vehicles* (UAV), ya que se busca automatizar todo el proceso de manejo del mismo.

De este modo, surge la idea de realizar este Trabajo de Fin de Grado (TFG), juntando lo mejor de ambos mundos, soluciones autónomas con dispositivos aéreos tremadamente adaptables a las circunstancias del problema.

Concretamente, el objetivo de este proyecto ha sido demostrar que, empleando aprendizaje por refuerzo (Q-Learning), se puede lograr rastrear y navegar hacia una señal Radio Frecuencia (RF) de forma más efectiva que con el resto de aproximaciones planteadas.

Acrónimos

RAE Real Academia Española

TFG Trabajo de Fin de Grado

UAV *Unmanned Air Vehicles*

UAS *Unmanned Aerial Systems*

GCS *Ground Control Station*

SUAV *Small Unmanned Air Vehicle*

LOS *Line Of Sight*

IA Inteligencia Artificial

RF Radio Frecuencia

ROS *Robot Operating System*

ADC *Analog to Digital Converter*

RSSI *Received Signal Strength Indicator*

SNR *Signal to Noise Ratio*

PLE *Path-Loss Exponent*

Índice general

1. Introducción	1
1.1. Robots	2
1.1.1. Drones	3
1.2. Inteligencia artificial	5
1.2.1. Aprendizaje por refuerzo	6
1.3. Señales	7
1.4. Síntesis	7
2. Objetivos	8
2.1. Descripción del problema	8
2.2. Metodología	9
2.3. Plan de trabajo	9
3. Plataformas de desarrollo y herramientas utilizadas	11
3.1. Lenguajes de programación	11
3.1.1. Python	11
3.1.2. C++	12
3.2. <i>Robot Operating System</i> (ROS)	12
3.2.1. Gazebo 11	13
3.2.2. Rviz	13
3.3. Plataformas de programación	14
3.3.1. Visual Studio Code	14
3.3.2. Github	14
3.4. Módulos	15
3.4.1. OpenCV	15
3.4.2. Matplotlib	15
3.4.3. PX4 autopilot	16
3.5. Iris	16

4. Diseño	17
4.1. Preparación del entorno	17
4.1.1. JdeRobot - drones	17
4.1.2. Teleoperador	18
4.2. Señales	22
4.2.1. Aproximación de Friis	23
4.2.2. Módulo python de Friis	24
4.2.3. Aplicación de Friis	27
4.3. Integración conjunta	29
4.3.1. Primeros pasos	29
4.3.2. Algoritmos	30
4.3.3. Métricas empleadas	36
4.3.4. Experimentos y resultados	38
4.3.5. Líneas a futuro - Experimentos con obstáculos	44
5. Conclusiones	46
5.1. Objetivos cumplidos	46
5.2. Balance global y competencias adquiridas	46
5.3. Líneas futuras	47
6. Anexo	48
Bibliografía	50

Índice de figuras

1.1.	Robótica industrial VS robótica móvil	2
1.2.	Definición de robot	2
1.3.	Historia de los drones	3
1.4.	UAS	4
1.5.	Clasificación de aprendizaje máquina	5
1.6.	Aprendizaje por refuerzo	6
1.7.	Modelo de Friis	7
3.1.	Esquema ROS Noetic	12
3.2.	Ejemplo de uso de Gazebo	13
3.3.	Ejemplo de uso de Rviz	13
3.4.	VS Code logo	14
3.5.	Ejemplo de uso OpenCV (detección de bordes)	15
3.6.	Aplicación hecha en Matplotlib	15
3.7.	PX4 Autopilot logo	16
3.8.	Iris drone en Gazebo 11	16
4.1.	3DR Iris simulado	18
4.2.	Primera versión del teleoperador	19
4.3.	Versión final del teleoperador	20
4.4.	Tabla ejemplos exponente n	23
4.5.	Representación del algoritmo	26
4.6.	Primera versión de la interfaz	27
4.7.	Versión final de la interfaz	28
4.8.	Sistemas de referencia	31
4.9.	Representación algoritmo manual	32
4.10.	Representación algoritmo manual optimizado	33
4.11.	Esquema episodio fase de entrenamiento	35
4.12.	Mapa de puntos 30x30 con la señal en la esquina	36

4.13. Trayectorias seguidas en mapa 12x12 con señal en el centro	36
4.14. Gráfico de entrenamiento	37
4.15. Gráficos comparativos	38
4.16. Características de la señal por defecto	38
4.17. Mapa de puntos (12x12), señal centrada	39
4.18. Comparativas (12x12), señal centrada	39
4.19. Mapa de puntos (12x12), señal en la esquina	40
4.20. Comparativas (12x12), señal en la esquina	40
4.21. Mapa de puntos (30x30), señal centrada	41
4.22. Comparativas (30x30), señal centrada	41
4.23. Mapa de puntos (30x30), señal en la esquina	42
4.24. Comparativas (30x30), señal en la esquina	42
4.25. Mapa de puntos (30x30), señales diferentes	43
4.26. Comparativas (30x30), señales diferentes	43
4.27. Funcionamiento del algoritmo de puntos	44
4.28. Simulación de sensor para obstáculo	45

Listado de códigos

3.1. Hello world en Python	11
3.2. Hello world en C++	12
4.1. Main de center to center app	21
4.2. Ejemplo básico de uso del módulo Friis	25

Índice de cuadros

6.1. Anexo con las fuentes de donde se han obtenido las imágenes para este proyecto	49
---	----

Capítulo 1

Introducción

En la actualidad, la tecnología forma parte de nuestro día a día. Prácticamente, constituye un elemento imprescindible para llevar a cabo cualquier actividad, sea profesional o cotidiana. ¿Su función? solucionar problemas para hacernos la vida más sencilla.

Con esto en mente, se presenta la robótica, pero **¿qué es la robótica?**. Según la Real Academia Española (RAE), la robótica se define como “*técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales.*” (Real Academia Española, s.f., definición 2) [1]. Sin embargo, no es precisa, por ello una definición más concreta podría ser, ciencia que engloba diversas ramas tecnológicas, encargada del estudio y diseño de dispositivos mecánicos, provistos de sensores y actuadores, capaces de realizar tareas a través de la extracción y posterior procesamiento de la información, con el fin de generar respuestas adecuadas para resolver determinados problemas [2].

Dentro de la robótica, existen diversas maneras de clasificar, sin embargo, una de las más comunes esta relacionada con la movilidad del dispositivo, esto es, si el mecanismo se puede desplazar por su entorno o no, por tanto se distingue lo siguiente:

1. **Robótica industrial:** que involucra mecanismos fijos, capaces de realizar tareas de manera rápida, precisa y eficiente. Como es el caso de los brazos robóticos [3].
2. **Robótica móvil:** la cual abarca a los dispositivos móviles que se engloban en múltiples entornos y aplicaciones, como pueden ser, robótica aérea, terrestre y submarina [4].

Como tal, la robótica ayuda a resolver tareas repetitivas, peligrosas, delicadas y en ambientes problemáticos (conocidas como las 4 D's, *dull, dirty, dangerous and dear*) [5]. Sin embargo, uno de los problemas más complicados de abordar, es el **contexto**, es

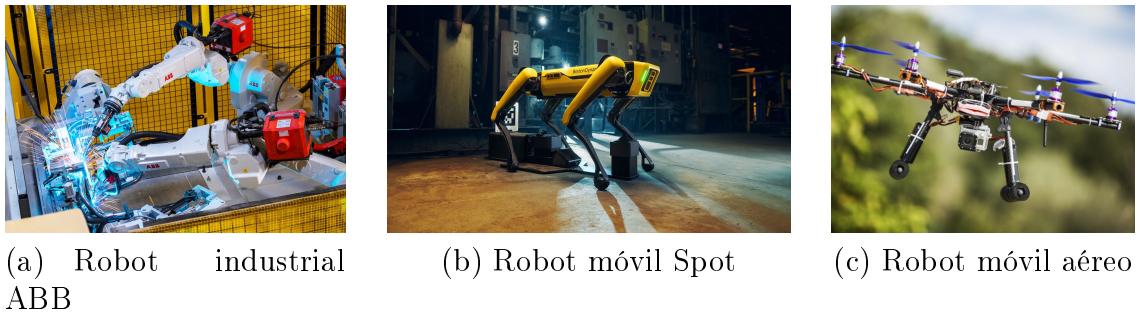


Figura 1.1: Robótica industrial VS robótica móvil

dicho, la capacidad de entender y adaptarse a las circunstancias del problema, como por ejemplo en el caso de la conducción autónoma, donde detectar un simple peatón, puede derivar en infinitos inconvenientes (condiciones de visibilidad, clima, atuendo, entre muchos otros). Es ahí, donde se presenta el segundo gran protagonista, la Inteligencia Artificial (IA) [6].

A continuación se definirán conceptos básicos como: ¿qué es exactamente un robot? ¿qué tipo de robot usaremos? ¿en qué consiste la IA y que emplearemos? ¿cómo funcionan las señales que rastrearemos? entre otras cuestiones.

1.1. Robots

Un robot es un dispositivo provisto con **sensores**, o elementos capaces de extraer información del entorno (por ejemplo una cámara), **actuadores**, o elementos que permiten al dispositivo realizar acciones (por ejemplo un motor), y una **unidad de procesamiento**, que se encarga de generar acciones a través de la información obtenida con los sensores, todo ello mediante algoritmos [7].

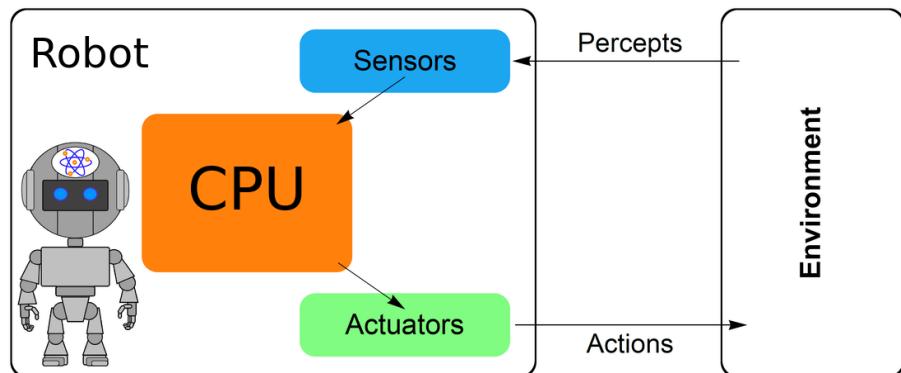


Figura 1.2: Definición de robot

Según el problema que se quiera resolver conviene usar unos u otros. En nuestro caso buscamos un robot con capacidad de navegar, preferiblemente grandes distancias y que pueda tomar medidas de la intensidad de una señal de forma autónoma.

Existen múltiples robots capaces de satisfacer estas condiciones, vease, los AGV/AMR o plataformas robóticas terrestres ampliamente empleadas logística [8]; robots bipedos, los cuales emulan el movimiento humanoide, lo que aumenta su adaptabilidad a cualquier entorno real (ya que el mundo esta diseñado para la biomecánica humana), sin embargo, es bastante complejo replicar la marcha bípeda [9]; y por último, drones, las cuales detallaremos a continuación. (REFERENCIAR!!!)

1.1.1. Drones

Los *Unmanned Aerial Systems* (UAS) tienen origen en la primera guerra mundial, con el biplano llamado **Kettering bug**. Se trataba de un torpedo que era lanzado desde una carretilla, capaz de volar de forma no tripulada, hasta que se liberaba de sus alas y caía sobre el objetivo [10]. Más tarde, entre la primera y segunda guerra mundial (1935), se diseño el **Queen Bee**, de donde surgió el termino “drone”, como abeja macho en busca de la reina, que se trataba de un avión no tripulado, con el fin de servir de objetivo para realizar prácticas de artillería aérea [11]. Sin embargo, no fue hasta **Operation Aphrodite**, en la segunda guerra mundial, donde realmente se vió el primer dron radio tripulado, con el fin de poder volar en entornos “sucios” o dirty, dado el nuevo paradigma de las bombas atómicas [12].



(a) Kettering Bug



(b) Queen Bee



(c) Aphrodite Operation

Figura 1.3: Historia de los drones

Existen múltiples avances y ejemplos posteriores, pero en la actualidad podemos definir un **UAS** teniendo en cuenta lo siguiente:

1. **Ground Control Station (GCS)**: es la estación de tierra o el elemento encargado de controlar la nave.
2. **Comunicación**: conecta y gestiona la transmisión de datos entre el UAV y la GCS, mediante **data links**, o canales de transmisión.
3. **UAV**: hace referencia directamente a la aeronave.

[13] [14] [15]

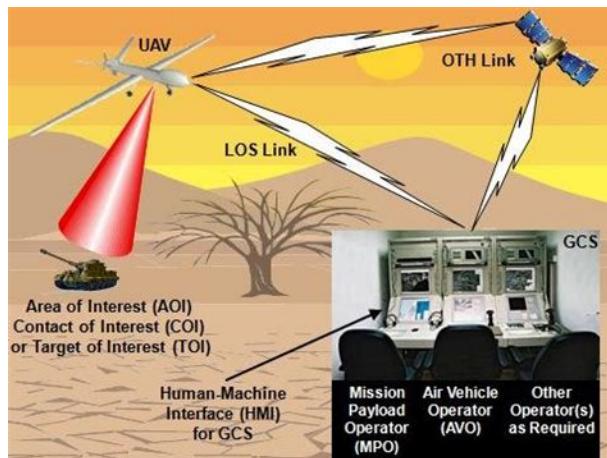


Figura 1.4: UAS

También cabe destacar que hay variedad de drones, según su peso y capacidad de carga de pago, o elementos que sea capaz de cargar, lo cual influye en la **legislación** detrás de su uso (de forma general, cuanto mayor sea el peso, más legislación debe cumplir y mayores restricciones de uso tiene) [16]. Por ello, el dispositivo seleccionado es de la categoría más inferior o también denominados *Small Unmanned Air Vehicle* (SUAV).

Tal y como fue mencionado, la gran ventaja del uso de vehículos aéreos es poder evitar las irregularidades del terreno, sin embargo, hay ligados al uso de estos dispositivos ciertos problemas, como son el clima, la carga de pago que afecta a la autonomía (peso de las baterías), los interiores (afectan a la señal GPS), entre otros. Esto, se debe tener en cuenta de cara a la resolución del problema.

1.2. Inteligencia artificial

El segundo pilar mencionado en este TFG, es el de la IA, pero, ¿de dónde surge esto?. Quizás, la primera pregunta que se buscó responder fue la siguiente **¿Puede una máquina pensar?**, formulada en “*Computing Machinery and Intelligence*” (Alan Turing, 1950), de donde surgió el famoso test de Turing, entre otras ideas [17]. La búsqueda de la IA enfrentó desafíos iniciales debido a la incapacidad de las primeras computadoras para almacenar datos y su elevado precio. Sin embargo, en 1956, se presentó el primer programa de IA llamado **Logic Theorist** en el **Dartmouth Summer Research Project on Artificial Intelligence** [18]. Con el tiempo, la IA progresó con mejores algoritmos y mejoras en la capacidad de las computadoras. A pesar de esto, lograr los objetivos finales de la IA, como comprender el lenguaje humano y el pensamiento abstracto, sigue siendo un desafío a día de hoy [19].

En general, la IA es capaz de abordar los siguientes problemas:

1. **Predicción:** que busca adelantar una respuesta precisa, con ciertos datos de entrada. Por ejemplo, predecir el precio de una vivienda en función de sus metros cuadrados.
2. **Clasificación:** que engloba datos en grupos según sus características. Como puede ser, detectar rostros de personas en imágenes.
3. **Aprendizaje:** Que busca resolver un problema a base de prueba y error, mediante un sistema de recompensas. Como por ejemplo, el juego del ajedrez, donde se busque ganar en el menor número de movimientos posibles.

[20]

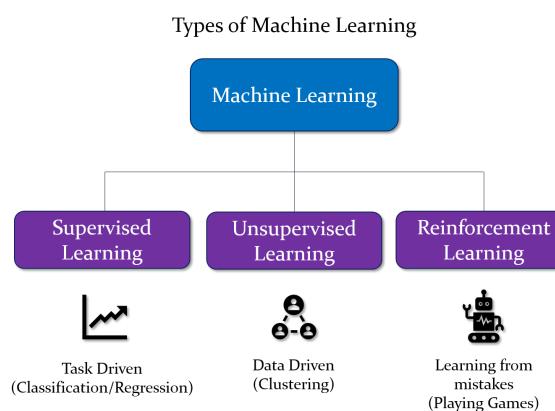


Figura 1.5: Clasificación de aprendizaje máquina

1.2.1. Aprendizaje por refuerzo

Dada la naturaleza de nuestro problema, el enfoque se basará en el aprendizaje por refuerzo, donde se planteará un sistema de **recompensas y penalizaciones**, que enseñará al robot a tomar las mejores acciones posibles [21].

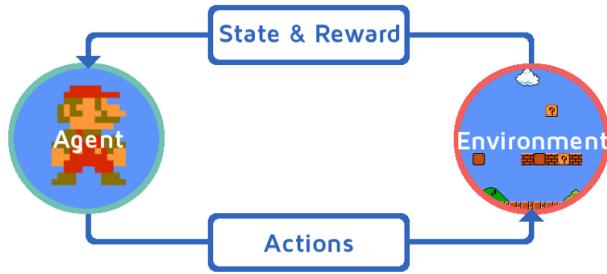


Figura 1.6: Aprendizaje por refuerzo

Cabe destacar que, este enfoque está directamente extraido de la **psicología** y el estudio del comportamiento, donde en función de recompensas y castigos se induce al aprendizaje en distintas tareas, como por ejemplo, enseñar a jugar al ping pong a dos palomas [22] [23].

Concretamente, se hará uso de la metodología **Q-Learning**, que busca generar una tabla numérica donde cada fila se interprete como un estado del robot, que puede ser su posición; y cada columna sea una determinada acción, como puede ser moverse hacia algún lugar. De este modo, y una vez construida la tabla (que se rellena con números en función de las recompensas y penalizaciones), el robot solo tiene que identificar en qué estado se encuentra y elegir la columna con mayor valor numérico, lo que se traducirá en la mejor acción para dicho estado [24].

1.3. Señales

Es el último elemento presente en el problema, que es la fuente de datos que se empleará para construir la inteligencia del robot. En este caso, hablamos de las señales, concretamente de señales **RF**, como son por ejemplo Wi-Fi, radio FM, 4G, 5G, entre otros tipos de señales.

En particular, se usará la aproximación de **Friis**, que ofrece un método para calcular la potencia de la señal de un receptor, en función de la potencia del transmisor, las ganancias del transmisor y el receptor, la longitud de onda (que determina el tipo de señal que es), la distancia entre ambos, el medio en el que se encuentren y un factor de pérdidas asociado al sistema empleado [25] [26] [27].

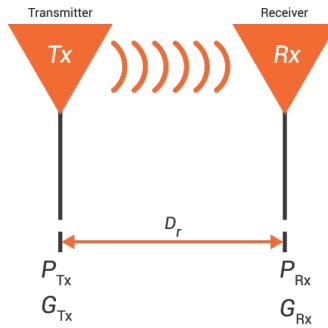


Figura 1.7: Modelo de Friis

1.4. Síntesis

Como conclusión de este capítulo, queda unir las piezas que conforman este puzzle, pero no sin antes mencionar algunos proyectos relacionados, donde se destaca el uso conjunto de drones y robótica para la creación de un módulo ROS, en el que se engloban las comunicaciones para programar un sigue-persona [28], junto con el uso de estas aeronaves en síntesis con la IA, para lograr la navegación en interiores de la misma [29]

En nuestro caso, se usará el mencionado dispositivo del tipo SUAV, provisto de un receptor RF como sensor, y los motores propios como actuadores. Se le agregarán algoritmos sistemáticos para compararlos con soluciones Q-Learning, en la tarea de resolver la detección y posterior navegación al origen de una señal RF, la cual será simulada mediante la aproximación de Friis.

Capítulo 2

Objetivos

Una vez presentado el contexto y el punto de partida del problema, el siguiente paso es definir el fin y los pasos a seguir para su resolución.

2.1. Descripción del problema

Tal y como se comentó previamente, los drones son una herramienta tremadamente versátil, ya que permiten solventar los inconvenientes orográficos de forma sencilla, y pueden ser provistos de múltiples sensores, lo que incrementa su adaptabilidad para solucionar un gran abanico de retos ingenieriles.

Dicho esto, el foco de este TFG consistirá en adaptar el funcionamiento de un dron, mediante los medios que nos proporciona la robótica, para detectar una señal RF estática y navegar hasta ella. Todo ello, con el uso exclusivo de un sensor capaz de detectar la intensidad de la señal, lo que permite al dispositivo navegar tanto en exteriores como en interiores. Además, se realizará una comparativa a cerca del rendimiento de diversos algoritmos, donde se incluirán algoritmos de IA, con el fin de determinar el método más óptimo para resolver el problema. Esto puede ser especialmente útil en labores de rastreo e identificación de objetivos.

Para ello, se establecen los siguientes objetivos:

1. Uso y comprensión de la herramienta de simulación Gazebo 11, para estudiar el comportamiento de un cuadracóptero. Todo ello a través de ROS Noetic, mediante el paquete MavROS, que permite la comunicación con el dron mediante ROS.
2. Desarrollo de una aplicación OpenCV, enfocada a teleoperar el dispositivo, empleando RVIZ para la visualización y el tratamiento del código.

3. Estudio de la propagación de señales y los diferentes parámetros involucrados, con el fin de desarrollar una aplicación responsiva (usando el módulo matplotlib), que simule el comportamiento de una señal, modificando sus características en tiempo real. Todo ello, a través del desarrollo de un módulo personalizado de Python, capaz de generar mapas de señales que siguen la aproximación de Friis.
4. Implementación conjunta de los puntos tratados previamente, con el fin de crear escenarios personalizados, sobre los que probar diversos algoritmos.
5. Desarrollo y testeo de diferentes algoritmos, con el fin de realizar comparativas y concluir que aproximación resuelve mejor.

2.2. Metodología

Este trabajo, comenzó oficialmente en Septiembre de 2022, aunque se pusieran en común las ideas a principios del verano, y se concluye a finales de Septiembre de 2023.

La metodología para llevarlo a cabo fue la siguiente:

1. Reunión de control semanal o cada dos semanas vía Teams con el tutor, donde se realizaba una valoración del estado del proyecto y se establecían los futuros puntos a seguir.
2. Uso de la metodología Kanban, para la organización de los objetivos a corto plazo.
3. Uso y desarrollo guiado del código y otros recursos, a través de un repositorio común, así como publicaciones ocasionales del estado del proyecto en un blog, todo ello mediante Github.

2.3. Plan de trabajo

Para concluir este capítulo, los pasos seguidos han sido:

1. Etapa inicial, donde tras establecer los objetivos del proyecto, se empezó por investigar el estado del arte del uso de drones para aplicaciones robóticas. Posteriormente, se realizó una instalación del modelo del dron, para que funcionará mediante ROS y sus herramientas.

2. Primeros pasos en el TFG. Una vez establecidos los objetivos y teniendo el dron funcional en ROS, lo primero fue realizar una serie de aplicaciones sencillas, que permitieran teleoperar al dron. El punto era realizar movimientos que posteriormente se transladarían a los algoritmos finales del proyecto, como por ejemplo, desplazarse entre centros de celdas. Para ello:
 - Se estudió la comunicación MavROS, y se desarrolló un controlador compatible para enviar ordenes al dron.
 - Se empleó openCV para el desarrollo de una interfaz gráfica intuitiva con la que controlar el dispositivo.
 - Se agregó una cámara al modelo SDF para previsualizar desde la aplicación.
 - Se desarrolló una serie de funciones para observar el funcionamiento del programa, mediante la herramienta RVIZ.
3. El siguiente punto consistió en el estudio de las señales RF a través de la ecuación de Friis, el desarrollo de un módulo python que aplicase la teoría, y la creación de otra aplicación de previsualización mediante matplotlib.
4. El último punto relacionado con el desarrollo, fué la síntesis de los dos pasos previos, donde a través de ROS y sus herramientas, se desarrollaron diferentes algoritmos que se apoyaban en el módulo previo, para poder resolver el problema de detectar y navegar hasta una señal mediante un dron. Todo ello realizando diversas comparativas para poder extraer conclusiones.
5. Finalmente, se realizó la redacción de la memoria.

Capítulo 3

Plataformas de desarrollo y herramientas utilizadas

Para hacer realidad todo lo mencionado en capítulos anteriores, se usaron diversos recursos ingenieriles que hicieron posible el proyecto. A continuación, se detallará cada uno de ellos.

3.1. Lenguajes de programación

3.1.1. Python

A día de hoy, considerado el lenguaje de programación más popular [30], se ideó en 1991 por Guido van Rossum y se desarrolló en la Python Software Foundation [31]. Es interpretado, es decir, usa un programa que traduce las líneas de código para la máquina en tiempo de ejecución (lo cual lo hace más intuitivo pero menos eficiente). Además, permite la programación orientada a objetos en alto nivel, lo que ofrece gran dinamismo a la hora de usarlo [32] [33].

Debido a su amplia popularidad, podemos acceder a una gran variedad de módulos y utilidades desarrollados por la comunidad, los cuales se integran perfectamente en la resolución de nuestro problema. Entraremos en más detalles en apartados posteriores.

```
#! /usr/bin/env python

if __name__ == "__main__":
    print("Hello world!")
```

Código 3.1: *Hello world* en Python

3.1.2. C++

Seguido muy de cerca en fama, se encuentra el lenguaje de programación creado por Bjarne Stroustrup, en los laboratorios Bell en 1971. En este caso es compilado, lo que implica la traducción y enlazado previo a la ejecución. De corte más eficiente que Python, también permite la programación orientada a objetos. Se sitúa a medio camino entre un lenguaje de alto nivel y uno de bajo nivel [34].

```
#include <iostream>

int main(int argc, char ** argv) {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Código 3.2: *Hello world* en C++

3.2. ROS

Si se habla de robótica, se habla de ROS, ya que es el medio predilecto para el desarrollo de soluciones de este ámbito, pero, ¿qué es exactamente ROS?.

Se trata de un *middleware*, es decir, una infraestructura software situada entre el sistema operativo y el desarrollador, que incluye una serie de módulos y funcionalidades enfocadas al desarrollo de aplicaciones robóticas [35] [36]. La idea detrás, busca estandarizar soluciones que no dependan de los drivers de cada sensor y actuador presentes. De forma general, se trata de una arquitectura basada en nodos que se comunican entre sí, transmitiendo una serie de mensajes propios, a través de canales compartidos llamados *topics*.

Entre las herramientas usadas en este proyecto, se encuentran las siguientes.

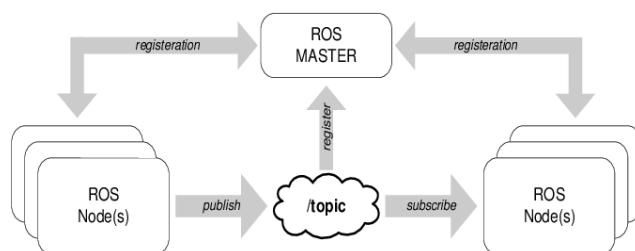


Figura 3.1: Esquema ROS Noetic

3.2.1. Gazebo 11

Se trata del simulador sobre el cual se desarrolla el proyecto. Concretamente consta de un conjunto de módulos optimizados para desarrollar aplicaciones robóticas, a través del previamente mencionado ROS [37].

Esta herramienta, nos permite visualizar en directo, el comportamiento del dron frente a los diversos escenarios que se planteen.

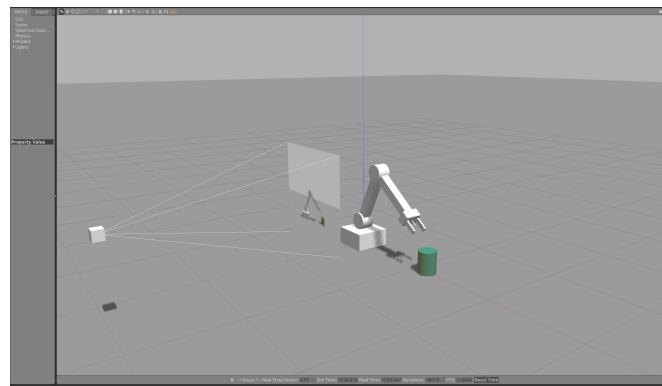


Figura 3.2: Ejemplo de uso de Gazebo

3.2.2. Rviz

Por el otro lado, se encuentra *rviz*, que es un visualizador 3D diseñado para la depuración de aplicaciones ROS [38].

En nuestro caso, nos permitirá ver como se dispersa la señal RF, que trayectoria y orientación sigue el dron, que efecto tiene sobre la señal la presencia de obstáculos, y otras tantas opciones.

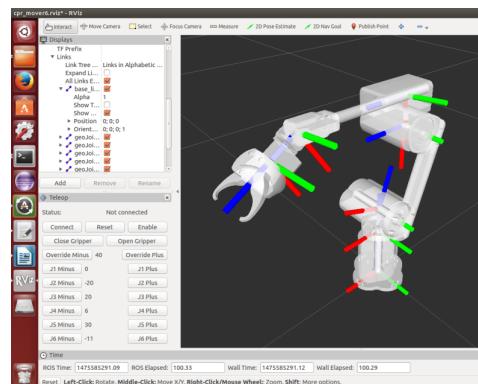


Figura 3.3: Ejemplo de uso de Rviz

3.3. Plataformas de programación

3.3.1. Visual Studio Code

Entre las plataformas usadas para programar, *Visual Studio Code*, o mejor conocido como *VS code*, es un editor de código ligero, funcional tanto en Linux, Windows y macOS [39].

Su principal ventaja, es que es altamente personalizable a el tipo de desarrollo software que se deseé realizar. Todo ello a través de las múltiples extensiones que ofrece, así como la conexión directa y fluida con plataformas como Github, que se detallarán a continuación.

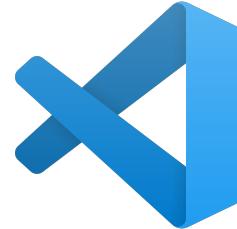


Figura 3.4: VS Code logo

3.3.2. Github

Github nace de la herramienta **git**, creada por Linus Torvalds (desarrollador de Linux), que es un sistema de control de versiones, que funciona a grandes rasgos a través de repositorios (o lugares donde se almacenan los sistemas de versiones de forma local), y commits (que permiten actualizar la versión del código almacenado del repositorio) [40].

Sabiendo esto entonces, ¿qué es github?. Consiste en trasladar la idea de, en vez de tener repositorios locales, que estén distribuidos en una plataforma online, donde además se permita el desarrollo conjunto de aplicaciones de manera distribuida.

Por ello, el papel que toma en este proyecto es de vital importancia, ya que asegura un seguimiento y una seguridad, de cara a tener copias de seguridad, donde todo el que desee puede acceder a ver en qué punto se encuentra el TFG pueda hacerlo.

3.4. Módulos

3.4.1. OpenCV

Es una biblioteca software de python, enfocada a visión artificial, que dispone de métodos para desarrollar interfaces gráficas de manera intuitiva [41].

Inicialmente se usó para esto último, por su sencillez y porque se implementó una cámara al dron, de cara a una posible extensión de su funcionalidad en algún punto del proyecto.



Figura 3.5: Ejemplo de uso OpenCV (detección de bordes)

3.4.2. Matplotlib

Presentada por John Hunter en 2002, se usó como biblioteca python alternativa a la anterior mencionada. La gran diferencia radica en que está diseñada para trabajar con estructuras numéricas del tipo array (muy compatible con Numpy¹), lo que permite ofrecer una gran visualización y una interfaz responsiva [42].

En el caso concreto del proyecto, se usó para desarrollar interfaz gráfica, que simula y muestra el comportamiento de una señal RF, permitiendo modificar los parámetros de la ecuación en tiempo real.

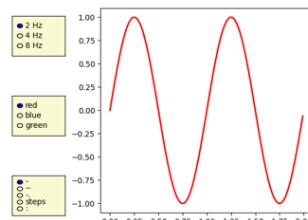


Figura 3.6: Aplicación hecha en Matplotlib

¹<https://numpy.org/about/>

3.4.3. PX4 autopilot

Es la capa de software que permite hacer funcionar las aeronaves con sus componentes, esto es a través del controlador de vuelo, que a efectos prácticos se trata del cerebro del sistema, es decir, interconecta los sensores y actuadores, permitiendo comandar diversas acciones [43] [44].

Este sistema, usa un conocido protocolo de comunicaciones llamado **MAVLink**, que se encarga de gestionar la comunicación entre el controlador de vuelo y la GCS. En nuestro caso, y como queremos desarrollar aplicaciones mediante ROS, debemos añadir una capa más, que se encarga de traducir los mensajes ROS a mensajes compatibles con el protocolo MAVLink, y de esto se encarga **MAVRos** [45] [46].



Figura 3.7: PX4 Autopilot logo

3.5. Iris

Es el nombre de la aeronave usada para solucionar los problemas planteados. En síntesis, es un dron cuadracóptero provisto de una cámara y un sensor de RF simulado. Dicha aeronave es cortesía de **JdeRobot**, que es un conjunto de herramientas pensadas para desarrollar aplicaciones robóticas [47].

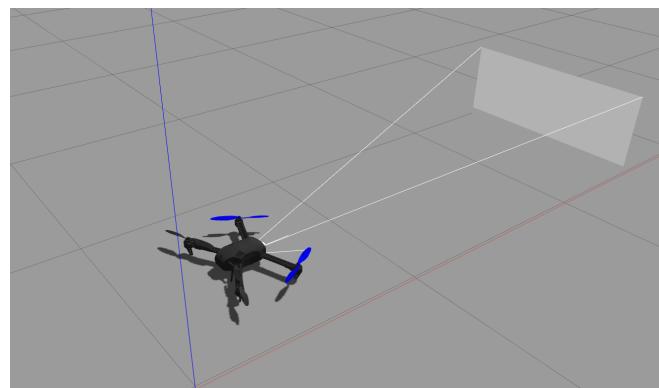


Figura 3.8: Iris drone en Gazebo 11

Capítulo 4

Diseño

Tras haber puesto en contexto todo lo anterior, en este capítulo se expondrá, de forma detallada, el proceso seguido para conseguir que un dron detecte y navegue hacia una señal RF.

Además, se mostrará el desarrollo de una aplicación responsive, que simula el comportamiento de una señal (en un espacio libre de obstáculos), basada en la aproximación de Friis.

Por último, se buscará determinar cuál de los métodos empleados es mejor y por qué, a través de diversas métricas comparativas que se expondrán en detalle posteriormente.

4.1. Preparación del entorno

Lo primero que se debía conseguir, era un entorno de simulación compatible con ROS, así como un sistema de control de versiones, que nos permitiera mantener la trazabilidad y los backups a mano. Por ello, se estableció un repositorio común en **GitHub** y se empleó el paquete de herramientas dispuesto por **JdeRobot**.

4.1.1. JdeRobot - drones

Gracias a esta plataforma, se pudieron obtener los modelos y los módulos necesarios para simular en Gazebo 11 el desempeño de un cuadracóptero, provisto de un sistema autopilot PX4.

El modelo usado es el **3DR Iris simulado**, con un plugin de una cámara frontal. Este dispositivo utiliza MAVROS para realizar la comunicación, lo que nos permite enviar y recibir mensajes ROS compatibles con el protocolo de comunicaciones típico de estas aeronaves, MAVLink.

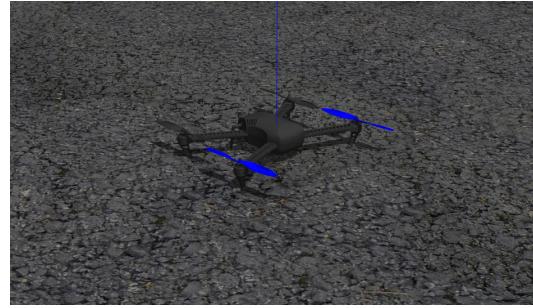


Figura 4.1: 3DR Iris simulado

4.1.2. Teleoperador

Ya propiamente hablando de resolver el problema planteado, la primera aproximación propuesta fue hacer una interfaz gráfica simple, que permitiera enviar órdenes a la aeronave.

Para ello, inicialmente se debía conseguir enviar mensajes de forma programática. Por tanto, se diseñó un **script controlador** encargado de la comunicación directa con el **controlador de la aeronave**, para enviar y recibir diversos datos vía MAVROS. De igual modo, se debían satisfacer una serie de requisitos que aseguraran el correcto funcionamiento del sistema:

1. La comunicación debe darse a **más de 2Hz**, para evitar cambios indeseados en el funcionamiento interno del controlador PX4 (de la aeronave).
2. Antes de realizar cualquier comunicación, **debe asegurarse que el estado es “connected”**, lo que significa que el dron esta armado y en modo *OFFBOARD* (nuestra aeronave posee 7 modos distintos, *HOLD*, que mantiene la posición, *RETURN*, que vuelve al punto de despegue, *MISSION*, que permite cargar rutas programadas con anterioridad, *TAKEOFF*, habilita el despegue, *LAND*, habilita el aterrizaje, *FOLLOW ME*, que permite seguir objetivos, y *OFFBOARD*, que permite comandar al dron sin necesidad de GPS, lo que es útil de cara al desarrollo de aplicaciones robóticas) [48].

3. Una vez esta conectado, se deben **enviar datos** (velocidades en nuestro caso) al controlador PX4, con el fin de **evitar el cierre de la conexión**. Estos datos carecen de utilidad más que la de asegurar dicha conectividad.
4. Por último, y antes de enviar cualquier posición, velocidad o comando (distintos modos de actuación), se debe comprobar siempre que el **modo activo** es *OFFBOARD* y que el dron esta **armado** (listo para volar). En caso contrario, se debe solicitar al controlador, mediante servicios, dichas especificaciones.

Por tanto, con todo esto funcionando de forma correcta, la manera de generar comportamientos en el dron en sí, es mediante *topics*. Concretamente, los que genera MAVROS automáticamente cuando se lanza todo el sistema. Tal y como se comentó en apartados previos, estos *topics* sólo admiten mensajes ROS, lo que encapsula el mensaje real transmitido al controlador PX4, que solo es compatible con MAVLINK.

En nuestro caso, enviaremos posiciones (PoseStamped), velocidades (Twist) y comandos (sevicios formados por mensajes personalizados, creados por MAVROS, con formato ROS). Esto, nos permitirá conectar el resto de aplicaciones con el script controlador, mediante *topics* comunes, de forma que se encargue de enviar la acción final al dron, mientras el resto de scripts se encarguen de resolver otras tareas.

De este modo, inicia el desarrollo propiamente dicho del **teleoperador**. El teleoperador, se diseñó con el fin de generar comportamientos que se usarían en las fases finales del TFG. Sin embargo, en la primera versión, tan solo se buscó construir una interfaz gráfica sencilla, capaz de enviar órdenes usando ROS que, en última instancia, llegasen al dron y produjesen diversos comportamientos, tales como **moverse y girar**, a través de sliders.

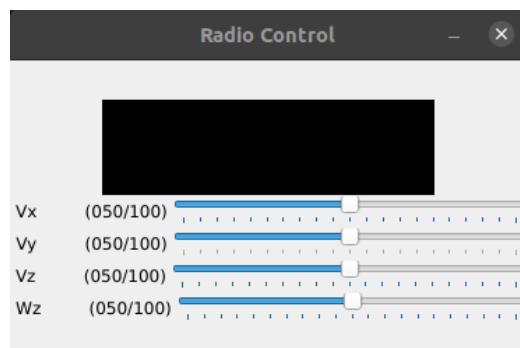


Figura 4.2: Primera versión del teleoperador

En el siguiente paso, se buscó programar **comportamientos predefinidos**, es decir, acciones predeterminadas tales como desplazarse distancias concretas en ciertas direcciones, o girar un número específico de grados en un sentido u otro. Para ello, se diseñó una ampliación sobre la interfaz anterior, en la que se añadió un **botón por cada acción** concreta desarrollada, además de una **imagen de la cámara frontal** en directo. Es importante destacar la acción de moverse de distancias específicas, ya que se empleará en los algoritmos posteriores.

Finalmente, y continuando con lo anterior, se intentó afinar el comportamiento de las acciones predefinidas, para hacer que el dron se desplazase de celda en celda, concretamente de **centro en centro**. La idea es que, el dron solo tomará medidas de la señal en posiciones concretas y no en movimiento. Además, se agregaron **marcadores en rviz**, para determinar las celdas visitadas (con colores aleatorios), junto con otro marcador que muestra la trayectoria que sigue la aeronave.

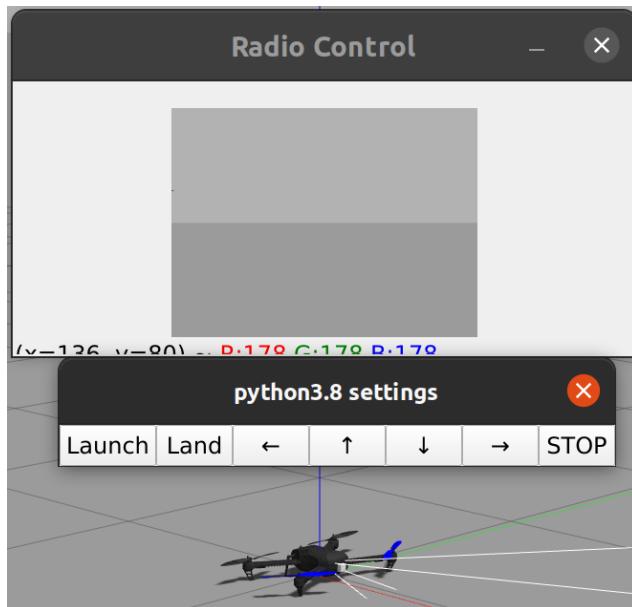


Figura 4.3: Versión final del teleoperador

A continuación, se muestra el “*main*” de la aplicación mencionada:

```

if __name__ == '__main__':
    try:
        rospy.init_node(NODENAME, anonymous=True)

        # Msgs
        ## Subscribers
        image_sub = rospy.Subscriber(IMAGE_TOPIC, Image, callback = image_cb)
        current_pos_sub = rospy.Subscriber(LOCAL_POSE_TOPIC, PoseStamped,
                                           callback = current_pos_cb)

        ## Publishers
        pos_pub = rospy.Publisher(RADIO_CONTROL_POS_TOPIC, PoseStamped,
                                  queue_size=10)
        cmd_pub = rospy.Publisher(RADIO_CONTROL_CMD_TOPIC, Px4Cmd,
                                  queue_size=10)

        # -- OPENCV --
        cv2.namedWindow(WINDOWNAME)

        # Buttons
        cv2.createButton('Launch', launch_button, None, cv2.QT_PUSH_BUTTON, 1)
        cv2.createButton('Land', land_button, None, cv2.QT_PUSH_BUTTON, 1)
        cv2.createButton('←', left_button, None, cv2.QT_PUSH_BUTTON, 1)
        cv2.createButton('↑', front_button, None, cv2.QT_PUSH_BUTTON, 1)
        cv2.createButton('↓', back_button, None, cv2.QT_PUSH_BUTTON, 1)
        cv2.createButton('→', right_button, None, cv2.QT_PUSH_BUTTON, 1)
        cv2.createButton('STOP', stop_button, None, cv2.QT_PUSH_BUTTON, 1)

        cv2.waitKey(0)
        cv2.destroyAllWindows()
    except rospy.ROSInterruptException:
        pass

```

Código 4.1: Main de center to center app

Donde, tras inicializar el nodo ROS, se definen por un lado los **suscriptores**, encargados de recibir los datos de la cámara y la posición del dron (usando MAVROS), los **publicadores**, cuya función es enviar posiciones y/o comandos al script controlador, y por último la **interfaz gráfica** diseñada con **OpenCV**, donde se define la ventana y los botones con las diversas acciones predefinidas ¹.

¹Código completo en https://github.com/RoboticsLabURJC/2022-tfg-cristian-sanchez/blob/main/src/teleop/scripts/c2c_control.py

4.2. Señales

Continuando con el proyecto, entramos en el segundo gran bloque, las **señales RF**. Este apartado fue especialmente relevante, ya que nos permitió desarrollar una aplicación reactiva, con la meta de generar entornos sobre los que probar soluciones robóticas.

Pero, antes de entrar en detalle en estos aspectos, conviene familiarizarse con algunos **conceptos básicos** sobre el **procesamiento de señales**:

1. *Señal*: se trata de una función que describe un fenómeno físico, y que se emplea para la transmisión de información.
2. *Dominio temporal*: establece el eje de abcisas con el tiempo.
3. *Dominio de la señal*: determina si la señal se expresa en tiempo o en frecuencia (a través de transformadas).
4. *Analog to Digital Converter (ADC)*: elemento electrónico que permite la conversión de señales analógicas a señales digitales.
5. *Received Signal Strength Indicator (RSSI)*: permite establecer el nivel de potencia de una señal, con respecto a 1 mW de potencia. Se expresa en dBm.
6. *Signal to Noise Ratio (SNR)*: métrica que permite medir la potencia de la señal con respecto al ruido ambiente.
7. *Frecuencia*: parámetro de la función que define a la señal, el cual determina el número de veces que se repite en un segundo. Se mide en hercios (Hz).
8. *TX*: Se refiere a la transmisión de la señal.
9. *RX*: Hace referencia a la recepción de la señal.

[49]

Definidos estos términos, nuestro objetivo se basó en obtener un modelo capaz de calcular la **propagación de la señal**, es decir, un sistema capaz de calcular como se comporta la potencia de una señal, según las características de su entorno.

4.2.1. Aproximación de Friis

Por ello, se optó por emplear la aproximación de Friis, que nos proporciona una sencilla fórmula sobre la cual modelizar nuestro problema:

$$P_r = P_t \cdot \left(\frac{G_t G_r \lambda^2}{(4\pi)^2 d^n L} \right) \quad (4.1)$$

Donde cada término significa lo siguiente:

1. P_t y P_r : aluden a la potencia del transmisor y la potencia del receptor respectivamente.
2. *Ganancias* (G_t , G_r): representan un valor incremental aplicado a la potencia de emisión y de recepción, respectivamente.
3. *Valor lambda* (λ): hace referencia a la longitud de onda. Está directamente relacionado con la frecuencia.
4. *Distancia* (d): desde el origen de la señal a un punto en el espacio.
5. L : representa todas aquellas pérdidas no asociadas a la propagación de la señal.
6. *Path-Loss Exponent* (PLE) (n): permite ajustar el modelo a diversos entornos. Es un valor constante extraido de forma empírica. A continuación se muestra una tabla con varios ejemplos.

[25]

Environment	Path Loss Exponent (n)
Free space	2
Urban area cellular radio	2.7 to 3.5
Shadowed urban cellular radio	3 to 5
Inside a building - line-of-sight	1.6 to 1.8
Obstructed in building	4 to 6
Obstructed in factory	2 to 3

Figura 4.4: Tabla ejemplos exponente n

Además, empleando este método, podemos modelizar las **pérdidas de una señal** estimadas durante su propagación, a través de la siguiente ecuación:

$$P_L(dB) = -10 \log_{10} \frac{\lambda^2}{(4\pi d)^2} \quad (4.2)$$

Sin embargo, para nuestro caso no es relevante (aunque está incluido dentro del módulo).

4.2.2. Módulo python de Friis

Una vez realizadas las pruebas iniciales, se buscó compactar todo, de forma que fuera accesible para cada aplicación que lo necesitase.

Por ello, surgió la idea de crear un **módulo python** que se encargará de modelizar las ecuaciones previamente mencionadas. Para ello, se diseño una clase cuyo constructor se encargara de recibir, por parámetros, las variables implicadas en las ecuaciones de Friis, además de las dimensiones del mapa y su resolución (la cual afecta al tamaño de celda).

Básicamente, el **proceso** seguido para usar este módulo es el siguiente, primero se crea un objeto de la clase Friis donde se especifican las características de la señal y las variables relacionadas con el mapa, lo que genera internamente un array 2D vacío, que será rellenado en función del modelo seleccionado. Posteriormente, se selecciona el modelo deseado (propagación o pérdidas, este último no está testeado), pasando las coordenadas del origen de la señal por parámetros. Esto, retornará el mapa lleno con los valores asociados a las ecuaciones del modelo de Friis seleccionado.

A continuación se mostrará un ejemplo sencillo de uso, donde se obtendrá un mapa de propagación de señal, en forma de Numpy array 2D:

```

#! /usr/bin/env python
import friss as fr

if __name__ == '__main__':
    friis_object = fr.Friiss(power_tras=10.0,
                            gain_tras=1.5,
                            gain_recv=2.0,
                            freq=fr.FREQ_WIFI,
                            losses_factor=1.0,
                            losses_path=2.0,
                            world_sz=(10,10),
                            resolution=1.0)

    signal_map = friis_object.model_power_signal(origin=(5,3))

```

Código 4.2: Ejemplo básico de uso del módulo Friis

Yendo al detalle, se puede ver que se genera un mapa 10x10 con resolución 1 (es decir, cada celdilla es de 1x1 unidades), de una señal WIFI, donde el transmisor emite a 10 W, con una ganancia de 1.5, el receptor posee una ganancia de 2, un factor de pérdidas (L) de 1, es decir, sin pérdidas, y por último, el exponente n (PLE) con un valor de 2, que representa el espacio vacío. Luego se genera el mapa de propagación de la señal, indicando que la fuente se encuentra en las coordenadas (5,3).

No obstante, este módulo posee **otras funcionalidades** útiles para trabajar con él, como son:

1. ***reset_world***: modifica el mapa que hubiera, estableciendo todos sus valores a cero.
2. ***get_world_sz***: retorna las dimensiones del mapa.
3. ***set_values***: modifica las características de la señal simulada.

Hay que tener en cuenta que, aunque se modifiquen los parámetros, se debe modelar de nuevo el mapa para que surtan efectos los cambios.

Por último, se buscó **extender la funcionalidad del módulo**, de manera que fuera posible incluir obstáculos en el mapa. La idea era crear un método capaz de posicionar **obstáculos** en el mapa de la señal, y simular su efecto en la señal.

Para ello, se buscó trazar proyecciones desde la señal hasta los bordes del obstáculo, de forma que se generase un polígono con los bordes del mapa. Posteriormente, se revisó que puntos se encontraban dentro del polígono (que no fueran propiamente parte del obstáculo), a través de un algoritmo basado en el número de intersecciones del punto con el perímetro del polígono, trazando una línea horizontal y hacia la derecha [50]. Por último, para cada punto dentro del polígono se le aplicó un factor de pérdida al valor almacenado de la señal, de manera que se simulase el efecto del obstáculo.

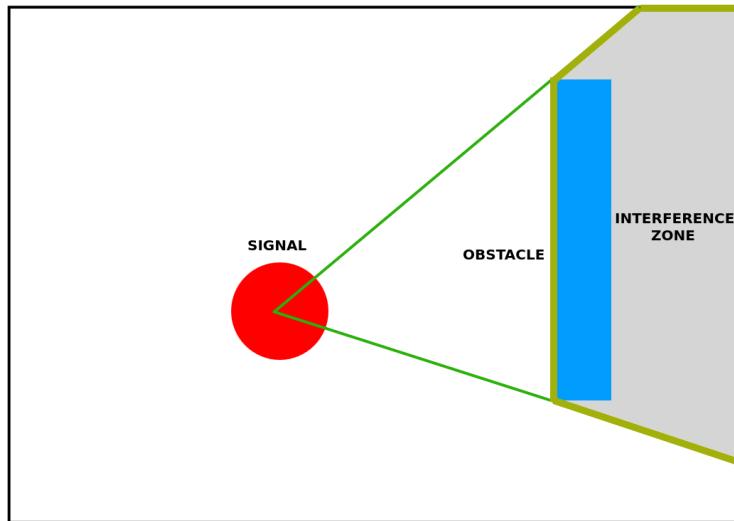


Figura 4.5: Representación del algoritmo

La idea fue abrir puertas a nuevos entornos más complejos y realistas, sobre los que poder probar diversas soluciones, en nuestro caso, para testear el efecto de un muro en el camino del dron a la señal.

4.2.3. Aplicación de Friis

La última sección relacionada con las señales se centró en integrar el módulo previo, en una **interfaz gráfica intuitiva para el usuario**. La idea era estudiar, en tiempo real, como evolucionaba la señal cuando alguno de sus parámetros, en la ecuación de Friis, era modificado.

Para llevarlo a cabo, se empleó la librería **matplotlib**, debido a la enorme cantidad de herramientas de las que dispone, así como de su sencillez a la hora de crear nuevas aplicaciones. Funciona de manera que se generan eventos que son gestionados en “*callbacks*”, es decir, se generan bucles asociados a dichos eventos, que reaccionan a cambios en la interfaz generada.

En nuestro caso, la **estructura base** consta de una figura, sobre la que se agregan todos los elementos, entre los que encontramos los mapas de calor o “*heatmap*” en forma de plots, las barras de acción o “*sliders*”, botones, entre otros elementos que se explicarán más adelante.

Inicialmente, se optó por representar un mapa de calor con una barra de color asociada a los distintos valores de la señal. Además, se integraron sliders correspondientes a cada valor presente en la ecuación de Friis, tal y como se muestra a continuación:

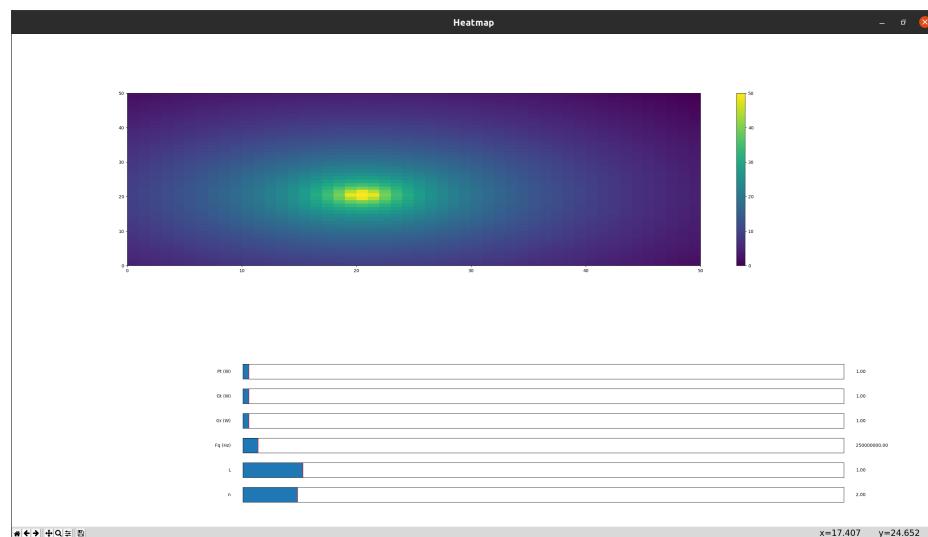


Figura 4.6: Primera versión de la interfaz

El problema fue, que al actualizar los valores, también lo hacía la representación, por lo que no se apreciaba el efecto de los cambios en el plot.

Por ello, se agregaron dos mapas de calor, uno con el máximo y el mínimo fijados a mano (donde sí se aprecian los cambios), y el anterior mencionado. Para elegir cual usar, se agregó una casilla marcable. Además, se incluyeron dos variables relevantes a la hora de modelar, el **tamaño del mapa** y la **resolución**, manejadas a través de “*sliders*”, los cuales a su vez se activan al pulsar un botón de **SET**, que recarga la interfaz. Además, se ajustó los saltos de valores para que fueran coherentes en el resto de barras de acción, tal y como se puede apreciar a continuación:

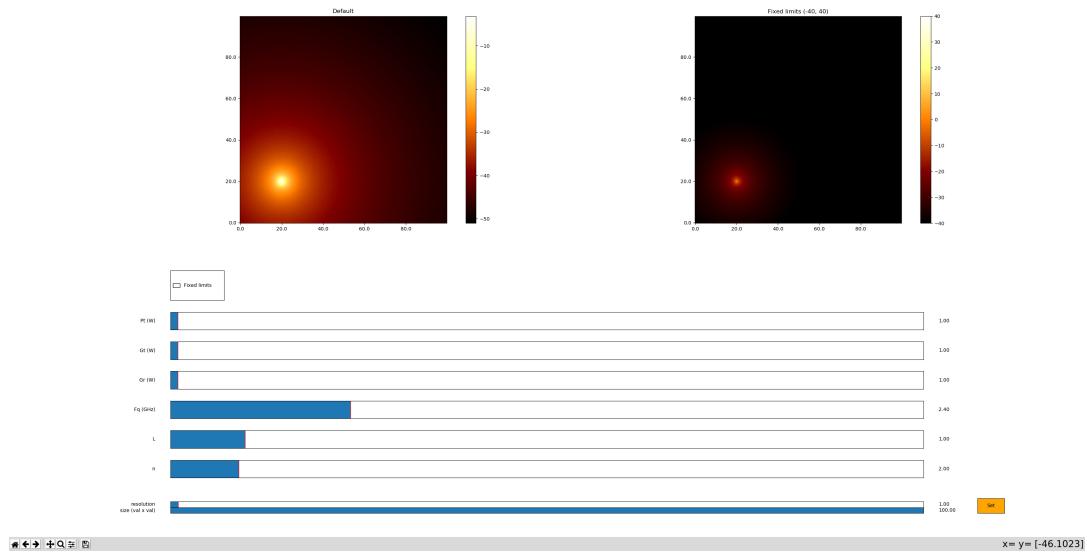


Figura 4.7: Versión final de la interfaz

Cabe destacar que, por motivos de desarrollo, no se agregó la parte de generación dinámica de obstáculos a la aplicación, ya que esta se desarrolló como un extra al final del TFG.

4.3. Integración conjunta

La integración conjunta engloba la parte final del proyecto, es la fase donde se juntaron las secciones anteriores, con el fin de generar el entorno deseado para resolver el problema.

El objetivo, a parte de llevar a cabo la tarea encomendada, era comprobar y comparar los distintos algoritmos entre sí, a través de diversas métricas de rendimiento.

4.3.1. Primeros pasos

Inicialmente, se debía construir todo el entorno en base a lo anterior.

Por ello, se diseñó una **aplicación servidor de datos**, que funciona como intermediaria con el módulo de Friis. Siendo concisos, dicha aplicación contiene **dos servidores** basados en **acciones ROS**, que son especialmente útiles en este caso, dada su naturaleza asíncrona. Dichos servidores gestionan las peticiones para el dron y para rviz. A continuación detallamos cada caso:

1. *Caso dron*: a groso modo, el dron envía su posición en coordenadas transformadas al sistema de referencia del “*heatmap*”, y recibe el valor de la señal de dichas coordenadas. En un caso real, el dron tan solo accedería al valor de la señal a través de un sensor que se lo permitiera. A posteriori, se agregó la funcionalidad de enviar, en dicha petición, si se deseaba un mapa con obstáculos o no.
2. *Caso rviz*: recibe una petición donde se agregan todas las características de la señal para generar el “*heatmap*” deseado, vease el origen y sus componentes. Esto, genera como respuesta un array de floats que contiene la información del mapa de calor, en un formato adecuado para su representación, es decir, para generar el mapa de forma gráfica, se emplea la biblioteca **grid_map**, que a través de un topic de ROS, permite enviar los datos a un plugin de rviz, el cual genera la representación visual buscada². También se agregó la funcionalidad de los obstáculos para experimentación futura.

²Toda la funcionalidad englobada en el directorio **heatmap_util** del proyecto

4.3.2. Algoritmos

En esta sección, se resume el núcleo del proyecto. Es el lugar donde se mostrarán todas las soluciones implementadas para comandar al dron hacia la resolución del problema y se explicará, desde la estructura general de la aplicación, hasta la lógica empleada detrás de cada algoritmo.

Por ello, lo primero consistió en definir una **clase “Drone”**, cuyo constructor se encargara de conectar los topics al controlador PX4 para comandar órdenes a la aeronave. Además, establece la comunicación con el servidor de datos (tanto para la potencia como para rviz) y se definen los diferentes atributos pertenecientes a la clase, que en este caso aluden a parámetros necesarios para los algoritmos y la extracción de resultados.

En general, la clase sigue una estructura basada en lo siguiente:

1. *Métodos para comandar al dron*: o conjunto de funciones encargadas del movimiento del dispositivo (como despegar, aterrizar, desplazarse, entre otros). Mucha de esa funcionalidad fue adaptada del teleoperador realizado al inicio del TFG.
2. *Métodos de tolerancia*: encargados de establecer un margen aceptable entre la posición del dron y el objetivo deseado. Estos métodos sirven para controlar con precisión problemas que surgen de la deriva y de condiciones externas, como puede ser el viento.
3. *Métodos de conversión*: que en este caso nos permiten transformar las coordenadas entre los sistemas de referencia, tal y como se puede apreciar a continuación.
4. *Algoritmos*: o las soluciones propiamente dichas, que en sí contienen el conjunto de métodos que cada cual necesita para llevarse a cabo. Podemos distinguir entre **manual, manual optimizado y Q-Learning**.

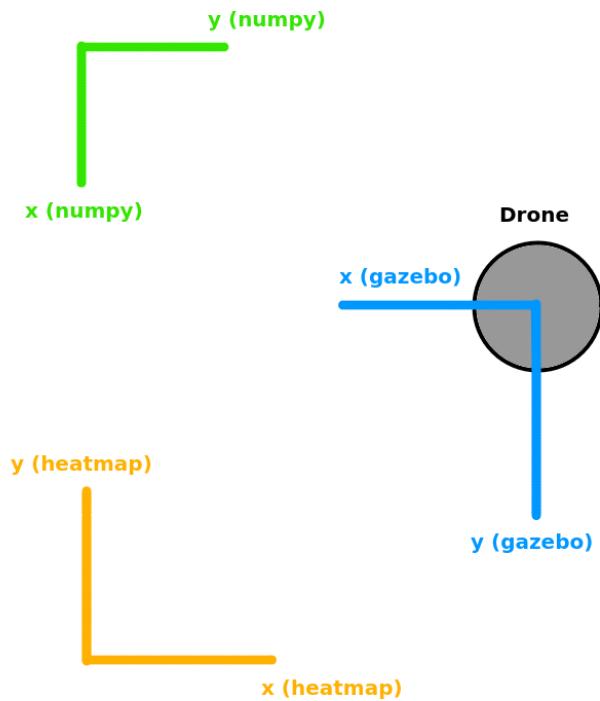


Figura 4.8: Sistemas de referencia

En cuanto al desarrollo propiamente dicho de los algoritmos, se deben cumplir una serie de premisas de cara a la simulación.

Primero que, **todos los movimientos realizados por el dron deben estar contenidos en el mapa de calor generado**; además, **la medida de la señal** sólo podrá tomarse cuando el dron esté en el **centro de la celda**; los movimientos del dron deberán ser **de centro en centro** aunque esto abarque más celdas de distancia (problema resuelto y adaptado del teleoperador); y cada celda mide 1x1 metros.

Algoritmo manual

Es básicamente la primera aproximación, consiste en **visitar todos los vecinos más cercanos** y realizar el desplazamiento hacia las coordenadas del **vecino con mayor señal** medida.

La **condición de parada** se basa en analizar si, las coordenadas objetivo de la iteración anterior son las mismas que las coordenadas objetivo de la iteración actual, además de que se cumpla que todos los vecinos colindantes tengan menor valor de señal mencionado.

En cuanto a los métodos que usa, se encuentra el de verificar movimientos válidos y comprobar si ha llegado al final, mediante la verificación de que todos los vecinos adyacentes, tienen potencias de señal inferior.

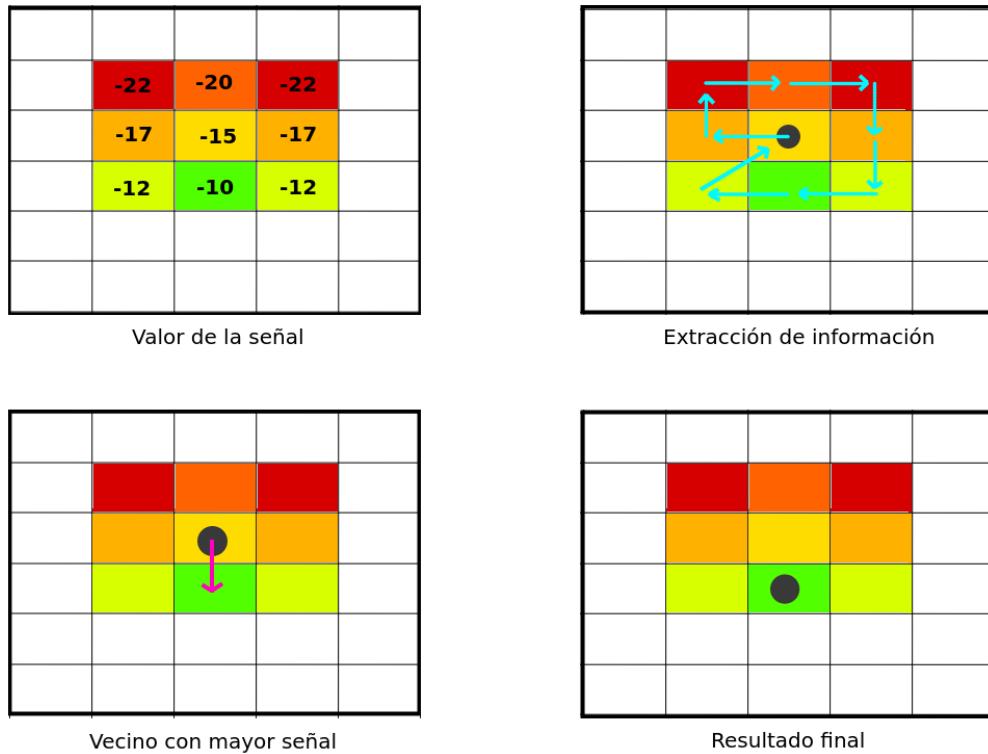


Figura 4.9: Representación algoritmo manual

Algoritmo manual (optimizado)

Tomando como referencia el algoritmo anterior, se buscó agregar ciertas mejoras y eficiencia. El principio es el mismo, obtener la información de los vecinos y navegar hacia el mejor candidato.

La diferencia radica en **no revisitar vecinos** cuya información se conozca. Para ello se implementa un array que almacena hasta 18 coordenadas de vecinos visitados, de modo que solo se navega hacia coordenadas no contenidas en el mismo, y que por supuesto cumplan las condiciones del problema (no salir del mapa de calor, mover de centro a centro, entre otras).

La condición de parada es idéntica a la anterior, y los métodos usados también.

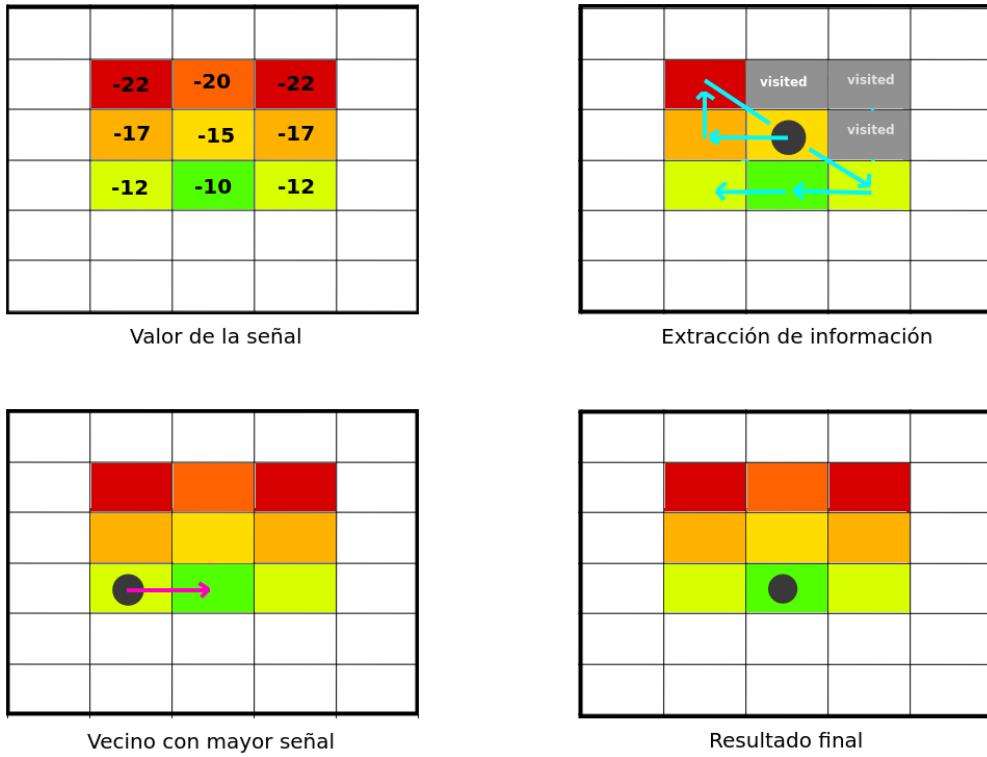


Figura 4.10: Representación algoritmo manual optimizado

Algoritmo Q-Learning

El último algoritmo planteado, se basó en técnicas de **aprendizaje por refuerzo**. Concretamente empleando Q-Learning, que tal y como comentamos al principio de la memoria, consiste en la obtención de una tabla Q, de estados y acciones, donde se asignan valores numéricos cada acción según su estado, de modo que la acción más favorable acaba teniendo mayor valor que el resto.

En nuestro caso, los **estados** son las **coordenadas del dron** en términos del mapa de calor, y las **acciones** son los **movimientos cardinales y diagonales**, de una o más celdas de distancia.

Como todo algoritmo de esta naturaleza, posee dos fases bien diferenciadas, la **fase de entrenamiento**, cuyo objetivo es llenar de forma eficaz la tabla Q, y la **fase de inferencia**, donde se prueban los resultados obtenidos del entrenamiento.

Dentro del entrenamiento, distinguimos los **episodios**, que en nuestro caso son las llegadas a la fuente, o las salidas del mapa de calor (adicionalmente se probó añadir otra condición que fuera basada en el número de malas acciones consecutivas, pero

para nuestra solución se decidió obviar); y las **iteraciones**, que se definen como el desempeño de una acción literalmente.

Además, para llenar el contenido de la tabla, se definieron las pertinentes **recompensas y penalizaciones** basadas en la diferencia entre la medidas, antes y después de realizar una acción (agregando un pequeño multiplicador a las recompensas negativas), excepto si se sale del mapa, en cuyo caso se establece una recompensa fija negativa, calculada en proporción al resto de recompensas. Posteriormente se asignan valores en la tabla Q, usando la ecuación de Bellman:

$$Q(s, a) = (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a')) \quad (4.3)$$

Cabe destacar que, durante el entrenamiento, se especifican una serie de parámetros que fueron ajustados a través de la experimentación, entre los que se encuentran: el **número de episodios totales**, que repercute directamente en la *fase de exploración* (detallado a continuación); el **parámetro** α , o la tasa de aprendizaje, que afecta a la convergencia de las soluciones durante el aprendizaje; el **parámetro** γ , o factor de descuento, que atañe a la importancia de las acciones futuras con respecto a las inmediatas; y por último los valores de **epsilon** (ϵ), que determinan si la acción tomada será aleatoria o extraída de la tabla, esto está directamente asociado a la *fase de exploración*, donde se prioriza la aleatoriedad con el fin de enriquecer con información la tabla Q.

En nuestro caso, esta fase ocupa un **20 % del número de episodios**, de forma lineal, es decir, que cada vez la prioridad se va decantando más del lado de la tabla y no de la aleatoriedad (durante el entrenamiento siempre se mantiene cierta posibilidad de tomar una acción arbitraria, para seguir actualizando los datos).

Para poder entrenar de forma eficiente, se establecieron distintos puntos de entrenamiento repartidos de forma uniforme por el mapa, hablaremos en detalle de esto, en la sección de métricas empleadas.

Los métodos usados para Q-Learning, se basan en funcionalidades necesarias para desempeñar todo lo anterior, véase la generación de estados y acciones para la tabla, la extracción de índices dentro de la misma, el tratamiento del parámetro ϵ , la obtención de coordenadas válidas, entre otros³.

³Todos los métodos están explicados dentro del código <https://github.com/RoboticsLabURJC/2022-tfg-cristian-sanchez/blob/main/src/teleop/scripts/algorithms.py>

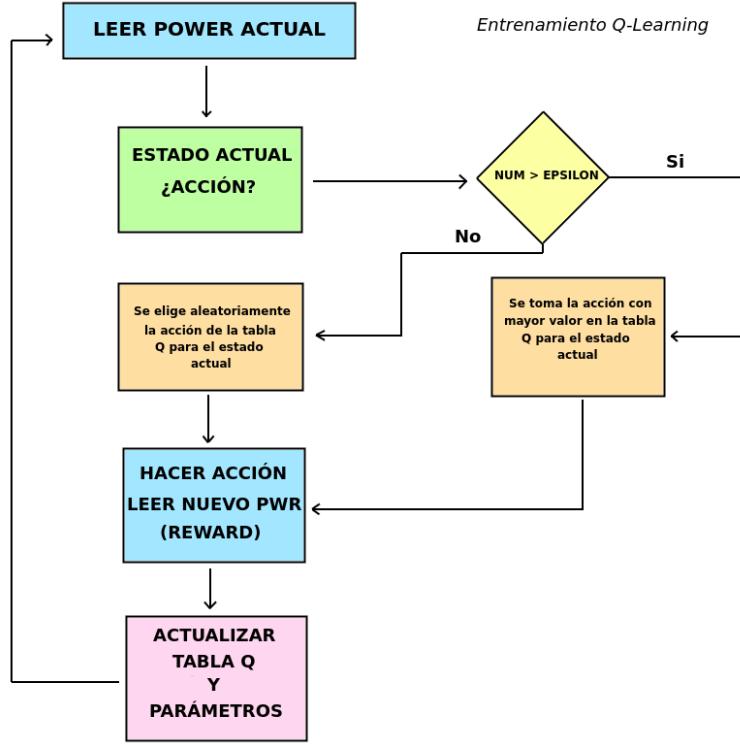


Figura 4.11: Esquema episodio fase de entrenamiento

Cabe resaltar que, si la acción tomada lleva al dron hacia una condición de final, este acaba el episodio, viaja hacia una nueva posición de entrenamiento y actualiza ciertos parámetros, como por ejemplo el parámetro ϵ . La condición de final se aplica siempre tras actualizar los valores.

Por último, en la *fase de inferencia*, el dron analiza su estado (o sus coordenadas dentro del mapa de calor), y observa la mejor acción disponible dentro de la tabla Q ya rellena. Esto lo realiza hasta que detecta la condición de parada, que se cumple cuando la medida anterior de señal es mayor que la actual y todos los vecinos adyacentes a la medida anterior poseen señal inferior. Para hacer un correcto análisis, se parte siempre de coordenadas distintas a las que se usaron para entrenar y llenar la tabla Q.

4.3.3. Métricas empleadas

He decidido comentar las métricas empleadas en una sección individual, debido a la importancia que poseen de cara al desarrollo y los resultados del proyecto.

En primer lugar se encuentra el **mapa de puntos**. Aquí se muestran las posiciones en el mapa de calor donde el dron entrenará, hará la inferencia (o desde donde partirá en los algoritmos manuales), además de la posición de la señal y propios los límites del mapa.

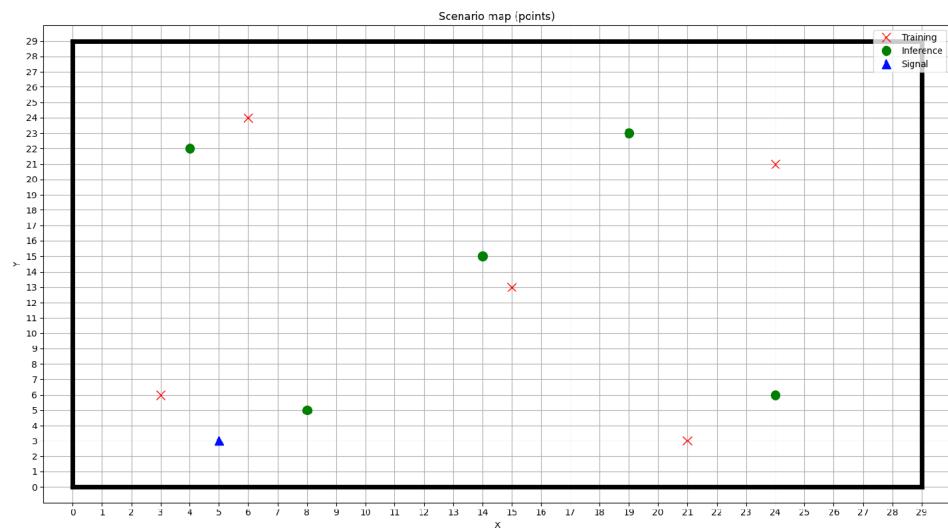


Figura 4.12: Mapa de puntos 30x30 con la señal en la esquina

El siguiente gráfico representa el **caminio seguido** por el dron al aplicar cada algoritmo para unas mismas coordenadas.

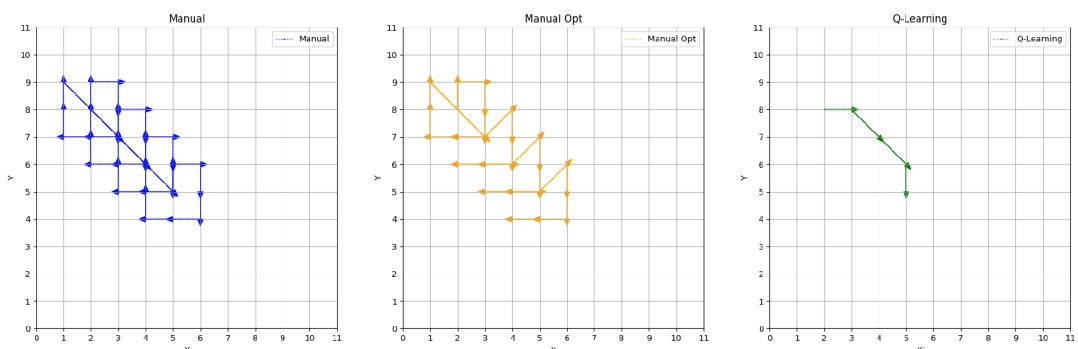


Figura 4.13: Trayectorias seguidas en mapa 12x12 con señal en el centro

A continuación se presenta uno de los gráficos más relevantes, en este caso, un gráfico triple que nos permite **conocer en detalle como ha ido el entrenamiento**. En concreto, representa tres métricas: el valor de **epsilon (ϵ)**, en el que se distingue la fase de exploración; la **recompensa acumulada**, que nos permite analizar la convergencia del entrenamiento; y el **número de iteraciones**, donde se observa que conforme el algoritmo aprende, el número se reduce. Todo ello con respecto a cada episodio.

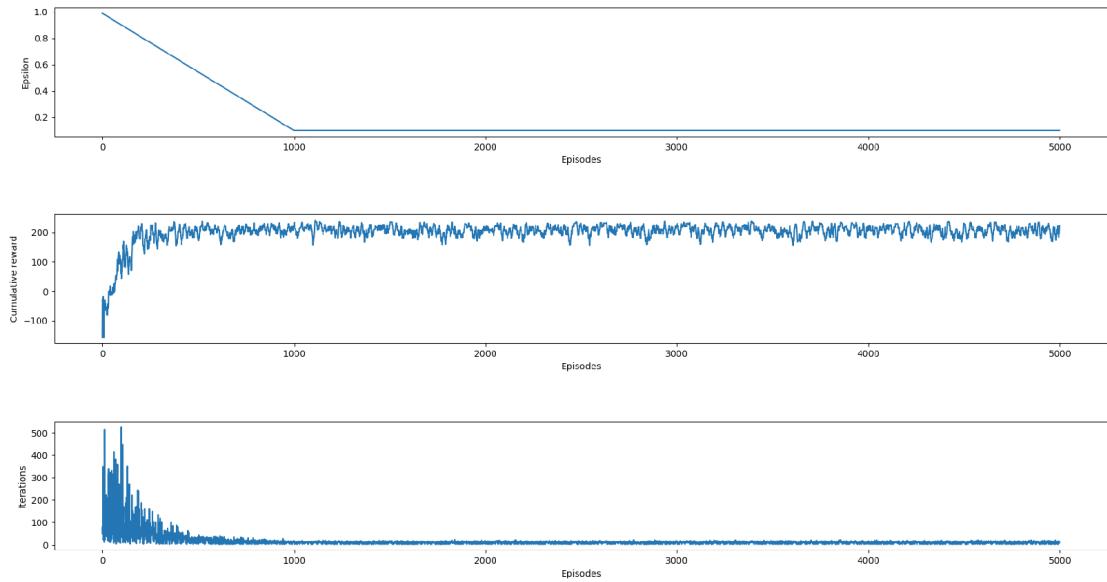


Figura 4.14: Gráfico de entrenamiento

Por último, se muestran los gráficos comparativos que nos dan un aproximado del **rendimiento** de cada algoritmo. En este caso, también se analizan tres cosas: el **tiempo medio** en segundos que tarda el dron desde que despegue hasta que vuelve a su posición de despegue; el **número medio de iteraciones** empleadas para alcanzar la señal; y el **número medio de movimientos** hacia coordenadas donde la señal es menor y no mayor⁴.

⁴Los datos arrojados han sido guardados en formato *csv*.

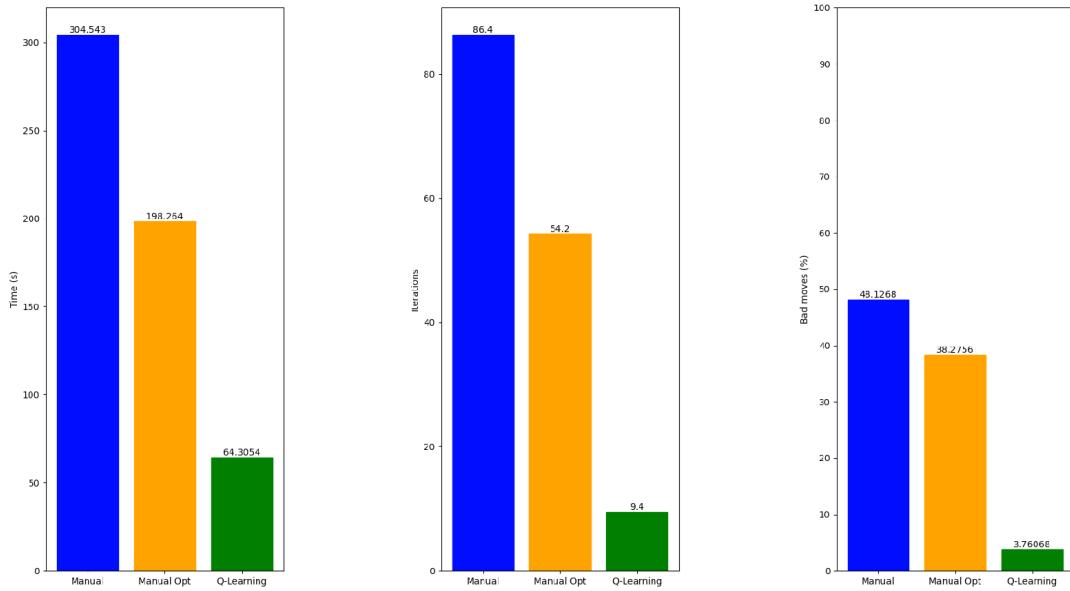


Figura 4.15: Gráficos comparativos

4.3.4. Experimentos y resultados

Una vez sabemos que métricas se van a usar, queda ver que resultados arroja la experimentación. A excepción del último caso, las características de la señal siempre son los valores por defecto a excepción del tamaño del mapa, que se va modificando conforme el experimento. Además cabe destacar que la señal se toma como estática con respecto al dron, en posiciones diversas, también según el experimento a realizar.

```
def __init__(self, power_tras=1.0, gain_tras=1.0, gain_recv=1.0,
            freq=FREQ_WIFI, losses_factor=2.4, losses_path=2.0,
            world_sz=(50, 50), resolution=1.0):
```

Figura 4.16: Características de la señal por defecto

Primero se probó sobre un escenario de tamaño **12x12 metros**, donde se distinguen dos posiciones de señal, una centrada y otra cerca de una esquina.

Para la **señal cerca del centro**, se dispuso en las coordenadas (5, 5) del “*heatmap*”, siguiendo el siguiente mapa de puntos para los puntos de entrenamiento e inferencia:

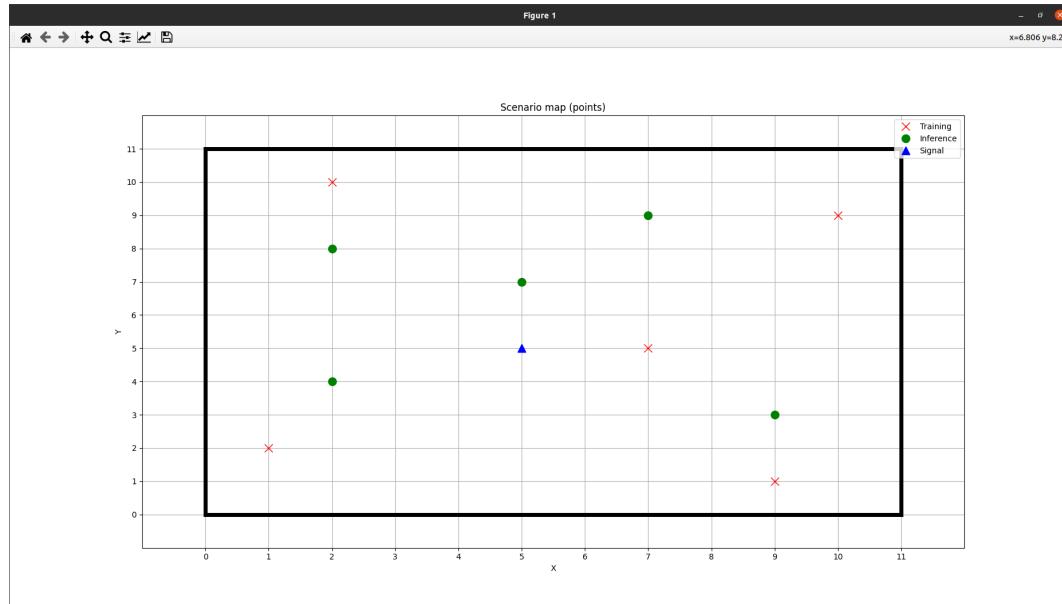


Figura 4.17: Mapa de puntos (12x12), señal centrada

Los resultados obtenidos arrojan que el algoritmo más eficiente es el de Q-Learning, ya que tarda menos tiempo, realiza menos iteraciones hasta llegar a la meta y tiene un porcentaje inferior de malas acciones, tal y como podemos ver a continuación:

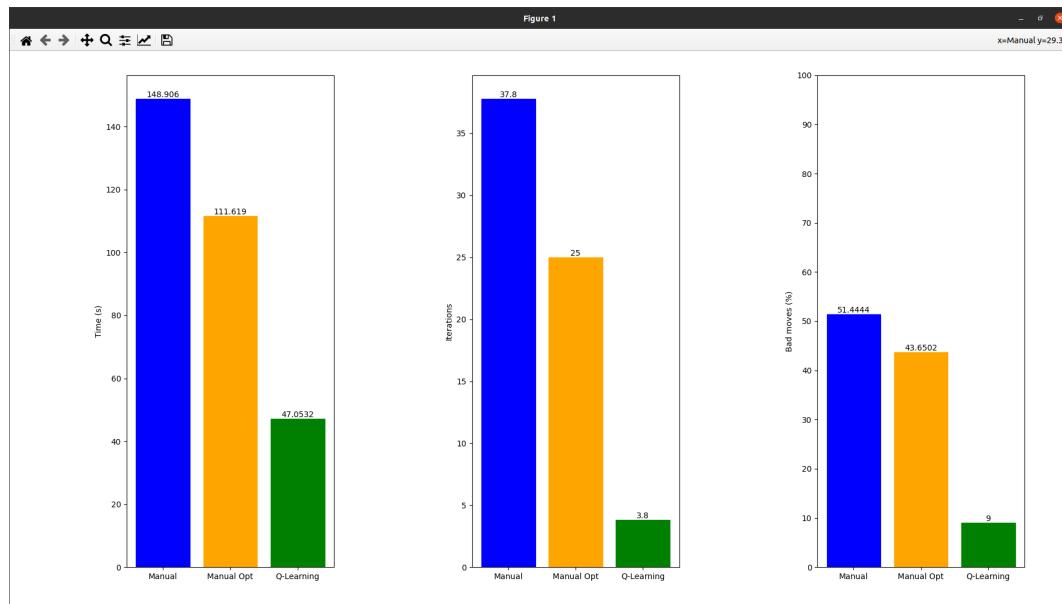


Figura 4.18: Comparativas (12x12), señal centrada

En el caso de la **señal cerca de la esquina**, la señal se estableció en (3, 1) referente a las coordenadas del “*heatmap*”, siendo su mapa de puntos el siguiente:

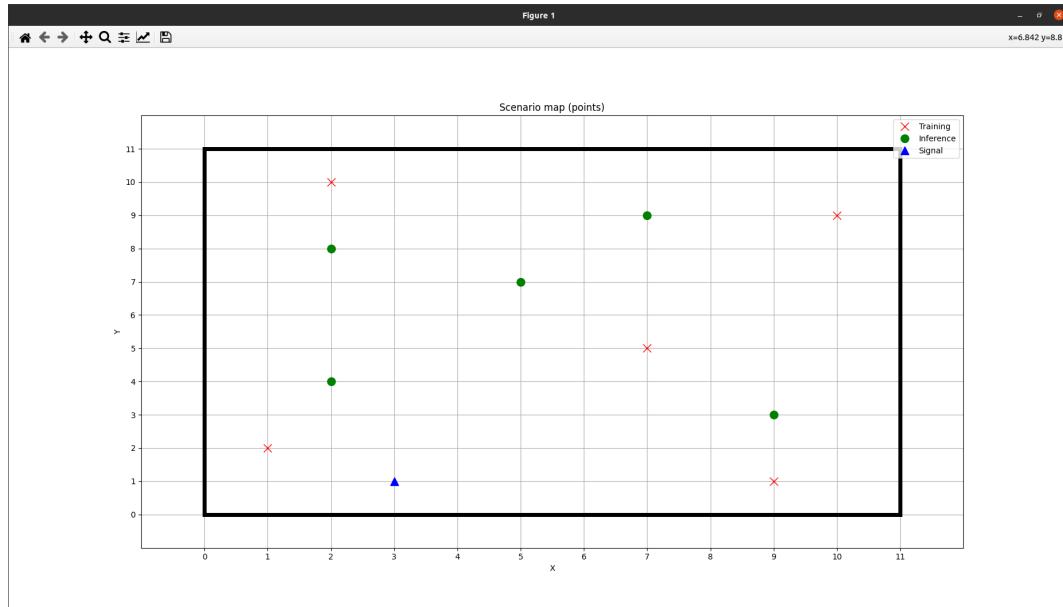


Figura 4.19: Mapa de puntos (12x12), señal en la esquina

En este caso, se obtiene la misma conclusión que en el escenario anterior, tal y como se puede apreciar a continuación:

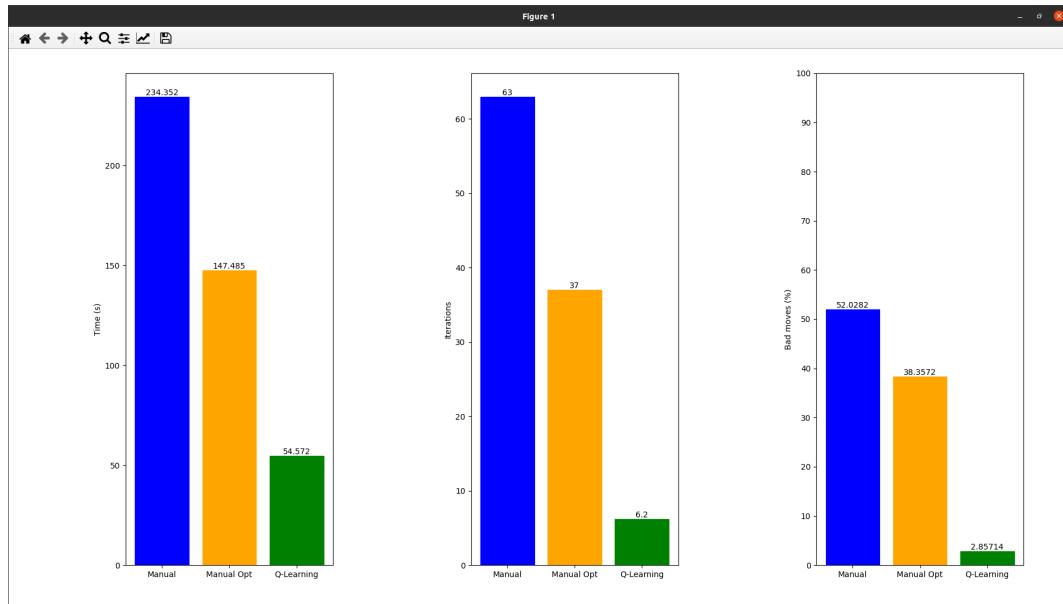


Figura 4.20: Comparativas (12x12), señal en la esquina

En segundo lugar, se incrementó el tamaño del mapa hasta **30x30 metros**.

Nuevamente, para la **señal cerca del centro** en (12, 12):

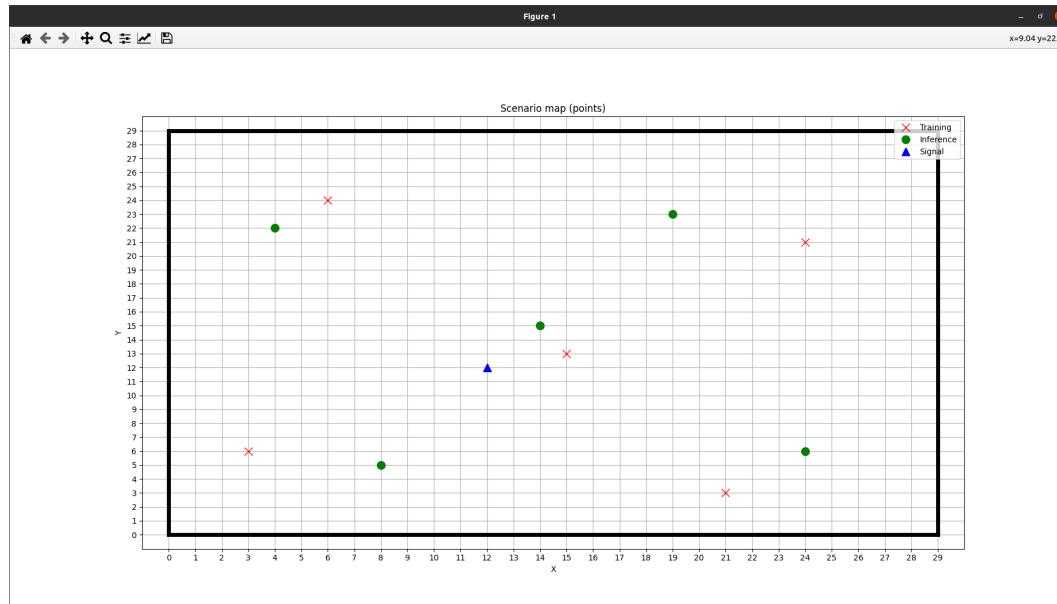


Figura 4.21: Mapa de puntos (30x30), señal centrada

En cuanto a los resultados, concluimos que Q-Learning vuelve a ser la mejor opción, que aunque se vea un incremento temporal y de iteraciones por el aumento de mapa, sigue superando al resto:

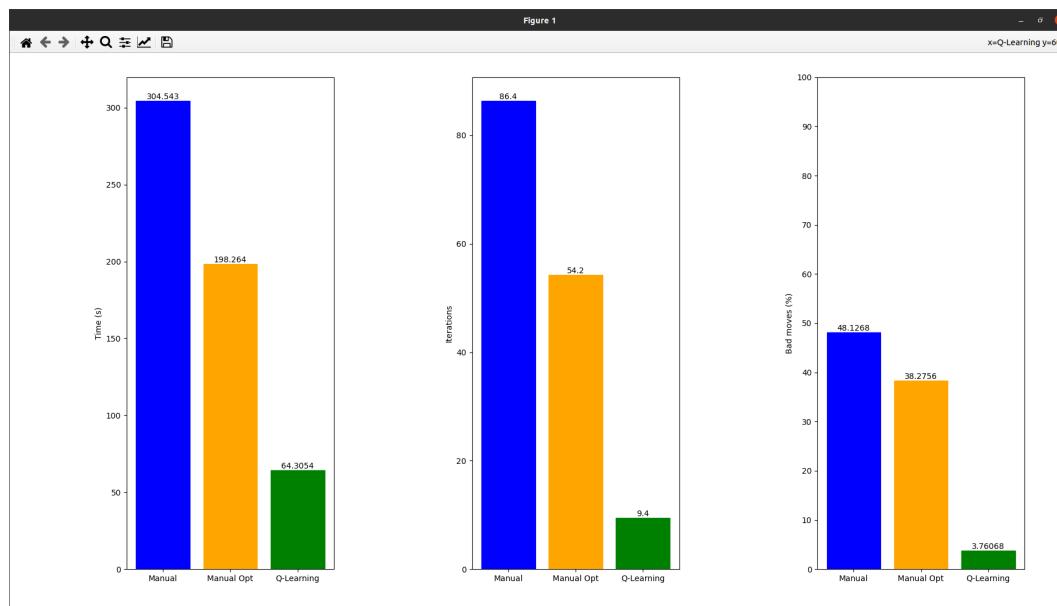


Figura 4.22: Comparativas (30x30), señal centrada

Continuando con la **señal cerca de la esquina**, en este caso se encuentra en las coordenadas (5, 3), y su mapa de puntos es:

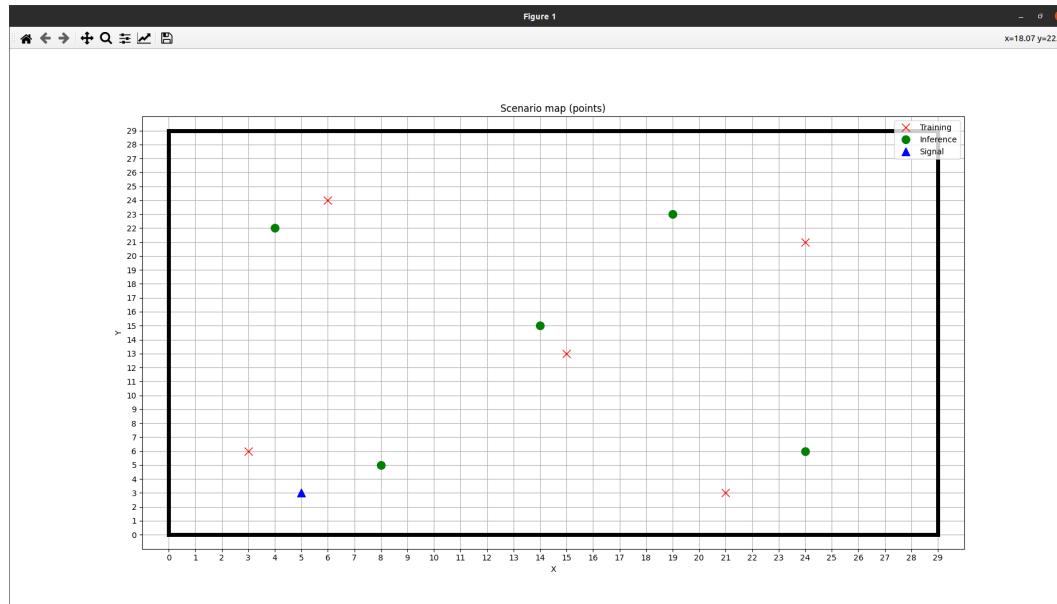


Figura 4.23: Mapa de puntos (30x30), señal en la esquina

De igual modo vemos que el resultado de aprendizaje por refuerzo es claramente superior:

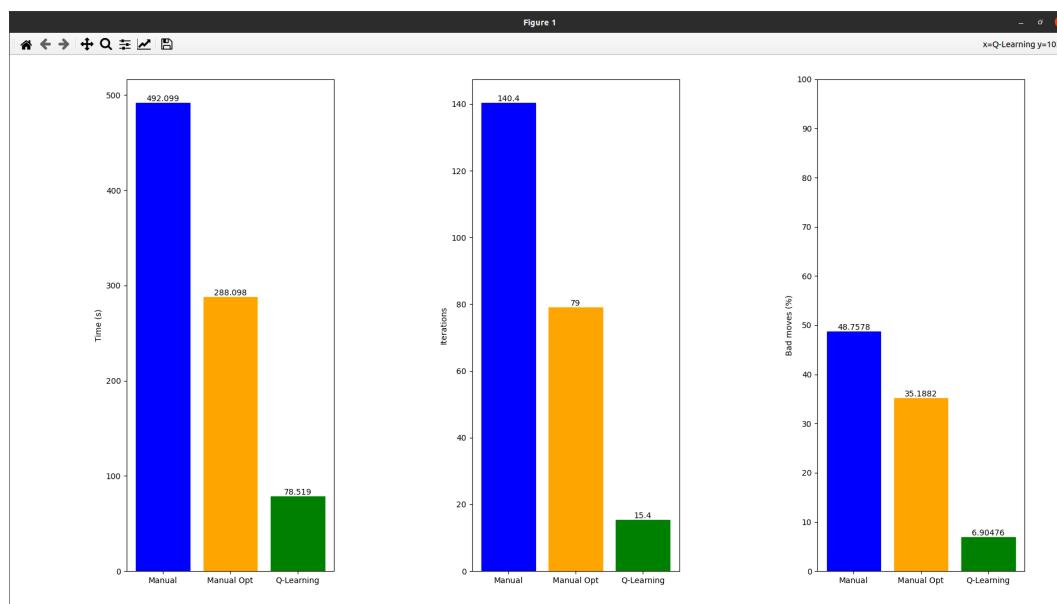


Figura 4.24: Comparativas (30x30), señal en la esquina

Por último, se planteó un problema sólo para Q-Learning, en el cual se **entrenaba al modelo con una señal**, y se realizaba inferencia con otra señal con **características distintas**, aumentando la potencia del transmisor al doble y cambiando la frecuencia para simular una señal 5G, situada en (5, 3):

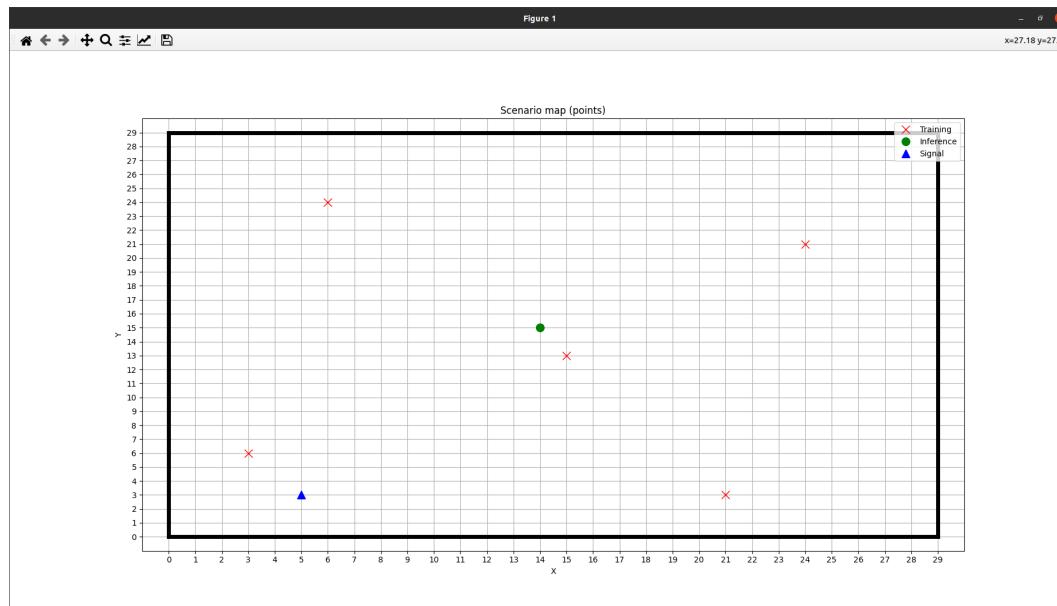


Figura 4.25: Mapa de puntos (30x30), señales diferentes

Para este caso, el problema se resolvía de igual forma para ambos casos, con una cierta variación en el tiempo, derivada probablemente de la propia simulación:

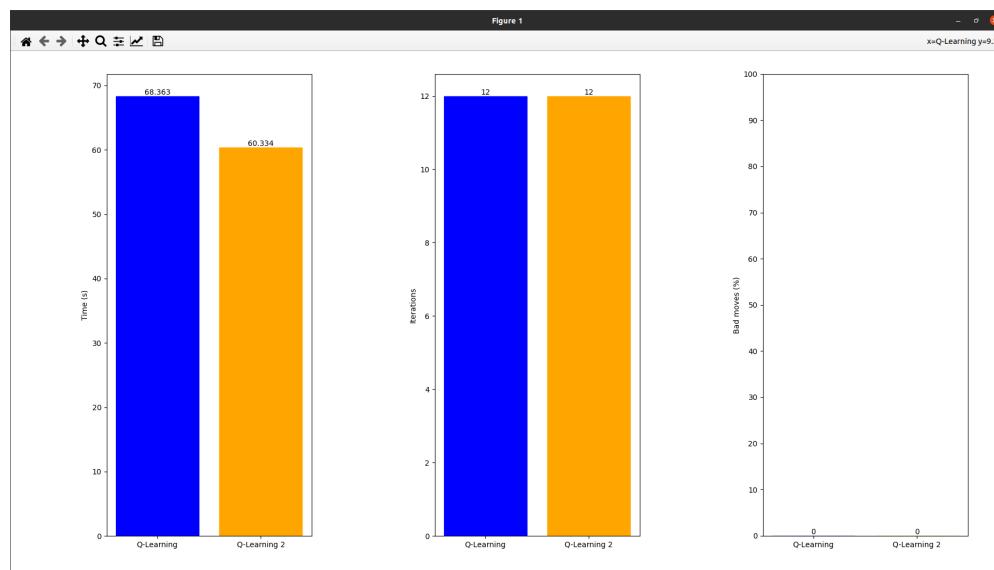


Figura 4.26: Comparativas (30x30), señales diferentes

4.3.5. Líneas a futuro - Experimentos con obstáculos

El escenario planteado durante el TFG, al fin y al cabo, es la aproximación más simple, es decir, en un **caso real** no es esperable un entorno vacío de perturbaciones y obstáculos. Por ello, el siguiente paso lógico es **implementar muros que distorsionen la señal** y ver como se comporta con Q-Learning.

Para ello, la idea es conseguir modificar el módulo de Friis, para que pueda agregar de forma dinámica obstáculos al mapa, empleando un algoritmo que identifica si los puntos se encuentran dentro del polígono generado por las proyecciones de la señal, con los vértices del muro [50]. Posteriormente se degrada la señal usando un factor de pérdidas.

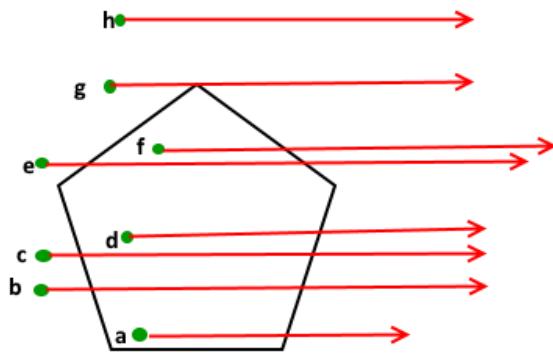


Figura 4.27: Funcionamiento del algoritmo de puntos

Para las primeras pruebas, se establecieron a mano los valores degradados de la señal, sobre un mapa con obstáculos.

Además, la idea inicial era realizar el entrenamiento **sin obstáculos**, y ajustar los algoritmos para que fueran capaces de sortear los mismos.

De este modo, y como primera aproximación a este proceso, se distinguieron **dos casos**:

1. *El dron vuela por encima de la altura del obstáculo:* En cuyo caso se ve que es capaz de navegar satisfactoriamente hacia la señal, con las soluciones empleadas previamente.
2. *El dron vuela a la misma altura que el obstáculo:* Donde se observa que la solución de Q-Learning queda insuficiente para resolver el problema, ya que el dron colisiona directamente con el obstáculo. Por ello, se plantea la idea de agregar un sistema de detección (o sensor) que permita al dron sortear los muros. Para el caso simulado simplemente se comprueba si la siguiente posición corresponde a un obstáculo en el mapa de calor y se actúa en consecuencia. Sin embargo, seguimos trabajando en este apartado.

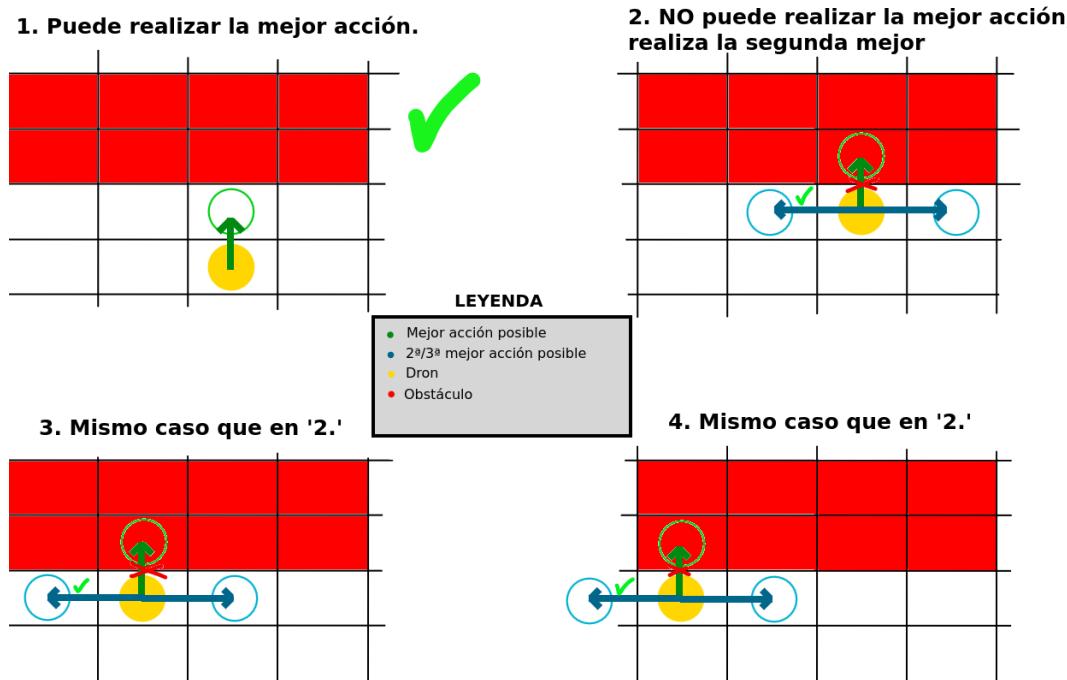


Figura 4.28: Simulación de sensor para obstáculo

Capítulo 5

Conclusiones

En esta última sección, se comentarán las ideas extraídas tras el desarrollo del proyecto, así como el conocimiento obtenido y la posible continuación del mismo.

5.1. Objetivos cumplidos

Entre los objetivos planteados y resueltos encontramos los siguientes:

1. Desarrollo de una solución ROS para una aeronave no tripulada o dron.
2. Creación de una aplicación reactiva de simulación de señales, siguiendo el modelo de Friis, y empleando Matplotlib.
3. Extracción del mejor algoritmo (entre los probados), a través de métricas comparativas y experimentación, haciendo uso de Gazebo 11, rviz y técnicas de aprendizaje por refuerzo (Q-Learning).

5.2. Balance global y competencias adquiridas

En cuanto a los conocimientos adquiridos, podemos distinguir:

1. Desarrollo de aplicaciones usando OpenCV y Matplotlib.
2. Uso de plugins y trabajo con el modelo SDF del Iris Drone.
3. Creación de entornos personalizados empleando Gazebo 11.
4. Uso de marcadores y del módulo grid_map para rviz, así como su aplicación conjunta a través ROS en C++ y Python.
5. Empleo de PX4 y MAVROS para el control de la aeronave.
6. Desarrollo de soluciones basadas en Q-Learning a través de Python.
7. Adquisición de conocimientos relacionados al estudio de señales y su comportamiento.

5.3. Líneas futuras

Finalmente, y tal y como se comentó en la sección anterior, la manera de continuar este proyecto es haciéndolo más afín a entornos realistas, lo que implica agregar elementos como obstáculos y perturbaciones, así como añadir modos de funcionamiento para el dron que le permita mejorar su adaptabilidad.

Además, el mejor algoritmo según los resultados obtenidos alude a la solución por Q-Learning, sin embargo, se podría considerar emplear otras variantes (como empleando aprendizaje profundo) y comprobar si arrojan mejores resultados.

Capítulo 6

Anexo

A continuación se muestran las referencias a las figuras de este trabajo junto con la fuente de la que han sido obtenidas:

Referencia imágenes	Fuente de la que se ha obtenido
??	<p>1.https://revistaderobots.com/robots-y-robotica/que-es-la-robotica/?cn-reloaded=1</p> <p>2.https://www.elindependiente.com/vida-sana/2018/01/22/los-robots-que-nos-cuidaran-en-2050/</p> <p>3.https://www.iguanarobot.com/wp-content/uploads/2021/03/429190-1.jpg</p> <p>4.https://www.robotexplorador.com/</p> <p>5.https://www.edsrobotics.com/blog/robots-autonomos-que-son/</p>
??	<p>1.https://www.hogarmania.com/hogar/economia/como-elegir-mejor-robot-aspirador.html</p> <p>2.http://automata.cps.unizar.es/robotica/Morfologia.pdf</p> <p>3.https://www.aarp.org/espanol/salud/enfermedades-y-tratamientos/info-12-2013/cirugia-robotica-beneficios-riesgos.html</p> <p>4.https://www.nobbot.com/mars-home-planet-reto-mundial-colonizar-marte/</p>
??	https://elpais.com/eps/2023-05-27/robots-que-sienten-lo-que-tocan.html
??	http://www.technovelgy.com/ct/Science-Fiction-News.asp?NewsNum=455
??	https://www.medicalexpo.es/prod/hocoma/product-68750-438408.html
??	https://exoesqueleto/pediatrico/puede/comercializar

??	1. https://altertecnia.com/exoesqueletos-mejorar-productividad/ 2. https://www.eafit.edu.co/innovacion/spinoff/natural-vitro/PublishingImages/banner%20-exoesqueleto.jpg 3. https://www.marsibionics.com/atlas-pediatric-exo-pacientes/ 4. https://exoesqueleto-militar
??	1. https://shop.bihar.coop/es/inicio/34-exoesqueleto-flexible.html 2. https://exoesqueleto/rigido/movilidad
??	https://person-pose-keypoints
??	https://stm32cubeide.html
??	1. https://journals.sagepub.com/doi/full/10.1177/1687814017735791 2. https://cjme.springeropen.com/articles/10.1186/s10033-020-00465-z
??	https://www.atriainnovation.com/que-son-las-redes-neuronales-y-sus-funciones/
??	https://esquema/arbol/decision
??	https://movenet/pose/estimation
??	https://www.analyticsvidhya.com/blog/2022/03/pose-detection-in-image-usi
??	https://developers.google.com/mediapipe/solutions/examples

Cuadro 6.1: Anexo con las fuentes de donde se han obtenido las imágenes para este proyecto

Bibliografía

- [1] Real Academia Española. *robótico, ca.* <https://dle.rae.es/rob%C3%B3tico#WYTncqf>, 2023.
- [2] Revista de Robots. *Robótica. Qué es la robótica y para qué sirve.* <https://revistaderobots.com/robots-y-robotica/que-es-la-robotica/?cn-reloaded=1>, 2023.
- [3] geeksforgeeks. *Industrial Robots.* <https://www.geeksforgeeks.org/industrial-robots/>, 2022.
- [4] geeksforgeeks. *Mobile Robots.* <https://www.geeksforgeeks.org/mobile-robots/>, 2022.
- [5] Bernard Marr. *The 4 Ds Of Robotisation: Dull, Dirty, Dangerous And Dear.* <https://bernardmarr.com/the-4-ds-of-robotisation-dull-dirty-dangerous-and-dear/>, 2021.
- [6] Daniel Dworakowski and Goldie Nejat. Robots understanding contextual information in human-centered environments using weakly supervised mask data distillation, 2020.
- [7] Jiefei Wang and Damith Herath. *What Makes Robots? Sensors, Actuators, and Algorithms*, pages 177–203. Springer Nature Singapore, Singapore, 2022.
- [8] National Museum of the United States Air Force. *Kettering Aerial Torpedo “Bug”.* <https://www.nationalmuseum.af.mil/Visit/Museum-Exhibits/Fact-Sheets/Display/Article/198095/kettering-aerial-torpedo-bug/>, 2023.
- [9] Andrea Maiorino and Giovanni Gerardo Muscolo. Biped robots with compliant joints for walking and running performance growing. *Frontiers in Mechanical Engineering*, 6, 2020.

- [10] National Museum of the United States Air Force. *Kettering Aerial Torpedo “Bug”*. <https://www.nationalmuseum.af.mil/Visit/Museum-Exhibits/Fact-Sheets/Display/Article/198095/kettering-aerial-torpedo-bug/>, 2023.
- [11] de Havilland Aircraft Museum. *de Havilland DH82B Queen Bee*. <https://www.dehavillandmuseum.co.uk/aircraft/de-havilland-dh82b-queen-bee/>, 2023.
- [12] Mason B. Webb. *Operation Aphrodite*. <https://warfarehistorynetwork.com/article/operation-aphrodite/>, 2014.
- [13] srmconsulting. ¿*UAV, UAS, RPA, dron... como llamarlos?* <https://srmconsulting.es/blog/uav-uas-rpa-dron-como-llamarlos.html>, 2021.
- [14] Brett Daniel. *Ground Control Stations: The Lifeblood of Remotely Piloted Aircraft*. <https://www.trentonsystems.com/blog/ground-control-stations>, 2020.
- [15] R. K. Nichols; H.C. Mumm; W.D. Lonstein; C. M. Carter; and J.P. Hood. *Chapter 13: Data Links Functions, Attributes and Latency*. New Prairie Press, 2018.
- [16] SAFEDroneFlying. *drone regulations*. <https://www.safedroneflying.aero/en/drone-guide/drone-regulations>, 2023.
- [17] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.
- [18] Leo Gugerty. Newell and simon’s logic theorist: Historical background and impact on cognitive modeling. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 50:880–884, 10 2006.
- [19] Rockwell Anyoha. *The History of Artificial Intelligence*. <https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/>, 2017.
- [20] datacamp. *Classification in Machine Learning: An Introduction*. <https://www.datacamp.com/blog/classification-machine-learning>, 2022.
- [21] Prateek Bajaj. *Reinforcement learning*. <https://www.geeksforgeeks.org/what-is-reinforcement-learning/>, 2023.
- [22] Prateek Bajaj. *Reinforcement and Punishment*. <https://pressbooks.online.ucf.edu/lumenpsychology/chapter/operant-conditioning/>, 2023.

- [23] B.F. Skinner. *Operant Conditioning*. <https://pressbooks-dev.oer.hawaii.edu/psychology/chapter/operant-conditioning/>, 2023.
- [24] OpenStax and Lumen Learning. *Reinforcement Learning Explained Visually (Part 4): Q Learning, step-by-step*. <https://towardsdatascience.com/reinforcement-learning-explained-visually-part-4-q-learning-step-by-step-b652023>.
- [25] Mathuranathan. *Friis Free Space Propagation Model*. <https://www.gaussianwaves.com/2013/09/friss-free-space-propagation-model/>, 2013.
- [26] ECIT Engineering. *Friis Free Space Propagation Model Example*. https://www.youtube.com/watch?v=E_oGNF3S-0, 2022.
- [27] antenna theory. *The Friis Equation*. <https://www.antenna-theory.com/basics/friis.php>, 2015.
- [28] Pedro Arias Pérez. *Infraestructura de programación de robots aéreos y aplicaciones visuales con aprendizaje profundo*. https://gsyc.urjc.es/jmplaza/students/tfm-drones-followperson-pedro_arias-2022.pdf, 2022.
- [29] Md Moin Uddin Chowdhury, Fatih Erden, and Ismail Guvenc. Rss-based q-learning for indoor uav navigation, 2019.
- [30] Tiobe. *TIOBE Index for August 2023*. <https://www.tiobe.com/tiobe-index/>, 2023.
- [31] SohomPramanick. *History of Python*. <https://www.geeksforgeeks.org/history-of-python/>, 2022.
- [32] Python Source Fundation. *What is Python? Executive Summary*. <https://www.python.org/doc/essays/blurb/>, 2023.
- [33] Bryce Limón. *Compiled vs interpreted language: Basics for beginning devs*. <https://www.educative.io/blog/compiled-vs-interpreted-language>, 2022.
- [34] SohomPramanick. *History of C++*. <https://www.geeksforgeeks.org/history-of-c/>, 2022.
- [35] IBM. *What is middleware?* <https://www.ibm.com/topics/middleware>, 2023.
- [36] ROS. *ROS - Robot Operating System*. <https://www.ros.org/>, 2023.

- [37] Gazebo. *About Gazebo*. <https://gazebosim.org/about>, 2023.
- [38] ros visualization. *RVIZ*. <https://github.com/ros-visualization/rviz#readme>, 2023.
- [39] Visual Studio Code. *Getting Started*. <https://code.visualstudio.com/docs>, 2023.
- [40] HTG Stuff. *What Is GitHub, and What Is It Used For?* <https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/>, 2016.
- [41] OpenCV. *About*. <https://opencv.org/about/>, 2023.
- [42] KattamuriMeghna. *Python / Introduction to Matplotlib*. <https://www.geeksforgeeks.org/python-introduction-matplotlib/>, 2023.
- [43] Peter Karanja. *How Drone Controllers Work (Explained for Beginners)*. <https://www.droneblog.com/drone-controller/>, 2023.
- [44] PX4 User Guide. *PX4 Autopilot User Guide (main)*. <https://docs.px4.io/main/en/>, 2023.
- [45] PX4 User Guide. *MAVLink Messaging*. <https://docs.px4.io/main/en/middleware/mavlink.html>, 2023.
- [46] PX4 User Guide. *ROS 1 with MAVROS*. <https://docs.px4.io/main/en/ros/ros1.html>, 2023.
- [47] JdeRobot. *JdeRobot*. <https://jderobot.github.io/>, 2023.
- [48] PX4 Autopilot. *Autonomous and Manual Modes*. https://docs.px4.io/main/en/getting_started/flight_modes.html, 2023.
- [49] UPM. *MOOC Software-Defined Radio 101 with RTL-SDR*. https://www.youtube.com/playlist?list=PL8bSwVy8_IcPCsBE71CYBLbQSS8ckWm6x, 2018.
- [50] geeksforgeeks. *How to check if a given point lies inside or outside a polygon?* <https://www.geeksforgeeks.org/how-to-check-if-a-given-point-lies-inside-a-polygon/>, 2023.