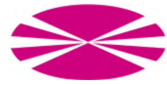
Escola Universitaria Politécnica



UNIVERSIDADE DA CORUÑA

Grado en Ingeniería Electrónica Industrial y Automática

TRABAJO DE FIN DE GRADO

TFG Nº: **770G01A185**

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

AUTOR: **DIEGO JAVIER GUTIÉRREZ BARRIO**

TUTOR: **ABRAHAM PRIETO GARCÍA**
PEDRO TRUEBA MARTÍNEZ

FECHA: **SEPTIEMBRE DE 2020**

Fdo.: EL AUTOR

Fdo.: EL TUTOR

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

ÍNDICE

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBREIRO, S/N

15405 - FERROL

FECHA: SEPTIEMBRE DE 2020

AUTOR: EL ALUMNO

Fdo.: DIEGO JAVIER GUTIÉRREZ BARRIO

I **ÍNDICE** . . . 3
Contenidos del TFG . . . 5
Listado de figuras . . . 9
Listado de tablas . . . 13
Listado de códigos de programación . . . 15

II **MEMORIA** . . . 17
Índice del documento Memoria . . . 19
1 OBJETO . . . 21
2 ALCANCE . . . 22
3 FUNDAMENTOS TECNOLÓGICOS . . . 23
3.1 Robobo . . . 23
3.1.1 Hardware . . . 23
3.1.2 Software . . . 25
3.2 Python . . . 26
3.3 OpenCV . . . 26
3.4 Pygame . . . 26
3.5 Pandas . . . 26
4 ANTECEDENTES . . . 27
4.1 Robótica . . . 27
4.1.1 Robótica Colectiva . . . 27
4.1.2 Arquitectura . . . 28
4.1.3 Reality Gap . . . 30
4.2 Métodos de Optimización . . . 30
4.2.1 Algoritmos Evolutivos . . . 31
4.2.2 Embodied Evolution . . . 34
4.3 Open-endedness . . . 36
4.3.1 Novelty Search . . . 36
4.3.2 Algoritmos QD . . . 37
5 NORMAS Y REFERENCIAS . . . 40
5.1 Disposiciones legales y normas aplicadas . . . 40
5.2 Software utilizado . . . 40
5.2.1 Software de Programación . . . 40
5.2.2 Software de cálculo . . . 40
5.2.3 Software de edición . . . 40
5.3 Referencias . . . 40
5.4 Otras referencias . . . 41
6 DEFINICIONES Y ABREVIATURAS . . . 42
7 REQUISITOS DE DISEÑO . . . 43

8 DESARROLLO DE LAS SOLUCIONES 44
8.1 Arquitectura 44
8.2 Implementación de herramientas de optimización 45
8.2.1 Benchmark Functions 45
8.2.2 Caracterización MAP-Elites 47
8.2.3 Caracterización Evolución Diferencial 51
8.3 Odometría visual 55
8.3.1 Sistema de visión 55
8.3.2 Detección de color 56
8.3.3 Estimación de la distancia 60
8.3.4 Análisis de errores en movimiento 62
8.4 Modelado 68
8.4.1 Modelado velocidad lineal 68
8.4.2 Modelado velocidad de rotación 72
8.5 Implementación del simulador 72
9 RESULTADOS FINALES 75
9.1 Configuración experimental 75
9.2 Navegación autónoma 75
9.3 Configuración algoritmos evolutivos 77
9.4 Resultados y análisis 77
9.4.1 Calidad de las soluciones 78
9.4.2 Diversidad de las soluciones 81
9.5 Transferencia al entorno real 84
10 CONCLUSIONES Y TRABAJO FUTURO 87
10.1 Conclusiones 87
10.2 Trabajo futuro 87
11 ORDEN DE PRIORIDAD ENTRE LOS DOCUMENTOS 89

III **ANEXOS** 91
Índice del documento Anexos 93
12 DOCUMENTACIÓN DE PARTIDA 95
12.1 Propuesta inicial de asignación de TFG 95
13 CÓDIGOS DE PROGRAMACIÓN 98
13.1 Algoritmos Evolutivos 98
13.2 Simulador 103

IV **PLANOS** 121

V **PLIEGO DE CONDICIONES** 125
Índice del documento Pliego de condiciones 127
14 PLIEGO DE CONDICIONES 129

VI **MEDICIONES** . . . 131
Índice del documento Mediciones . . . 133
15 Mediciones . . . 135

VII **PRESUPUESTO** . . . 137
Índice del documento Presupuesto . . . 139
16 PRESUPUESTO . . . 141

Listado de figuras

3.1 Robot Robobo [1]. 23
3.2 Componentes Robobo [1]. 24
3.3 Sensores infrarrojos [1]. 24
3.4 Interfaz de la aplicación. 25
4.1 Estructura red neuronal.[7] 29
4.2 Modelo neurona artificial.[8] 29
4.3 Estructura de los tipos Elman y Jordan.[6] 29
4.4 Algoritmo evolutivo típico 31
4.5 Algoritmo evolutivo diferencial[9]. 34
4.6 Clasificación Watson et al.[8]. 35
4.7 Clasificación Eiben et al..[4] 35
4.8 Espacio de búsqueda y espacio de características[6]. 38
8.1 Arquitectura del experimento 44
8.2 Arquitectura del sistema robótico 45
8.3 Representación de la función Ackley para dos dimensiones[10]. 46
8.4 Representación de la función Rastringin para dos dimensiones[11]. 47
8.5 Esquema de la representación en mapas de calor multidimensionales 49
8.6 Mapa de calor de la norma 49
8.7 Mapa de calor de la calidad 49
8.8 Mapa de calor del ratio de sustitución 50
8.9 Mapa de calor del porcentaje de ocupación 50
8.10 Soluciones obtenidas en una prueba para 4 dimensiones y 2 características. . . 51
8.11 Variación de la calidad según el número de dimensiones del problema. 52
8.12 Variación del tiempo según número de dimensiones del problema. 52
8.13 Variación de la calidad según el tamaño de población. 53
8.14 Variación del tiempo según el tamaño de población. 53
8.15 Variación de la calidad según CR. 54
8.16 Variación del tiempo según CR. 54
8.17 Variación del tiempo según F. 54
8.18 Variación del tiempo según F. 55
8.19 Información a extraer del sistema de visión. 56
8.20 Ilustración del espacio de color HSV[12]. 57
8.21 Histograma del parámetro Hue. 58

8.22 Histograma del parámetro Sat. 58
8.23 Histograma del parámetro Value. 58
8.24 Máscara e imagen resultante. 58
8.25 Histograma del parámetro Hue. 59
8.26 Histograma del parámetro Sat. 59
8.27 Histograma del parámetro Value. 59
8.28 Máscara e imagen resultante. 59
8.29 Posicionamiento del robot para obtener las imágenes. 60
8.30 Esquema geométrico de la estimación de la distancia. 60
8.31 Ajuste polinómico de segundo grado. 62
8.32 Ajuste polinómico de tercer grado. 62
8.33 Secuencia de paradas. 63
8.34 Discretización en una secuencia de movimiento. 64
8.35 Detección de la cinta con la zona difusa a su alrededor. 65
8.36 Detección de la cinta con la zona difusa a su alrededor. 65
8.37 Detección de la cinta con la zona difusa a su alrededor. 65
8.38 Comportamiento de los valores de pixel deseado. 66
8.39 Discretización nueva en una secuencia de movimiento. 66
8.40 Medición de la distancia durante la calibración del movimiento lineal 69
8.41 Calibración de la velocidad real para v = 10 69
8.42 Calibración de la velocidad real para v = 50 70
8.43 Distancia frente tiempo para v = 30 71
8.44 Distancia frente tiempo para v = 50 71
8.45 Geometría del movimiento de rotación del robot. 72
8.46 Geometría del movimiento de rotación del robot. 72
8.47 Representación del entorno en Pygame. 73
9.1 Disposición de los elementos en el entorno durante la prueba estática. 76
9.2 Disposición de los elementos en el entorno durante la prueba con los obstáculos en movimiento. 76
9.3 Izquierda: Calidad máxima para las tres configuraciones en cada iteración de MAP-Elites en el entorno vacío. Derecha: Calidad máxima para las tres configuraciones en cada iteración de Evolución Diferencial en el entorno vacío. 78
9.4 Izquierda: Calidad máxima para las tres configuraciones en cada ejecución de MAP-Elites en el entorno vacío. Derecha: Calidad máxima para las tres configuraciones en cada ejecución de Evolución Diferencial en el entorno vacío. 78
9.5 Izquierda: Calidad máxima para las tres configuraciones en cada iteración de MAP-Elites con obstáculos estáticos. Derecha: Calidad máxima para las tres configuraciones en cada iteración de Evolución Diferencial con obstáculos estáticos. 79

9.6 Izquierda: Calidad máxima para las tres configuraciones en cada ejecución de MAP-Elites con obstáculos estáticos. Derecha: Calidad máxima para las tres configuraciones en cada ejecución de Evolución Diferencial con obstáculos estáticos. . . . 79
9.7 Izquierda: Calidad máxima para las tres configuraciones en cada iteración de MAP-Elites con obstáculos dinámicos. Derecha: Calidad máxima para las tres configuraciones en cada iteración de Evolución Diferencial con obstáculos dinámicos. . . . 80
9.8 Izquierda: Calidad máxima para las tres configuraciones en cada ejecución de MAP-Elites con obstáculos dinámicos. Derecha: Calidad máxima para las tres configuraciones en cada ejecución de Evolución Diferencial con obstáculos dinámicos. . . . 80
9.9 Trayectorias producidas por MAP-Elites en el entorno vacío. . . . 81
9.10 Trayectorias producidas por Evolución Diferencial en el entorno vacío. . . . 81
9.11 Trayectorias producidas por MAP-Elites en el entorno con obstáculos. . . . 82
9.12 Trayectorias producidas por Evolución Diferencial en el entorno con obstáculos. 82
9.13 Trayectorias producidas por MAP-Elites en el entorno con obstáculos dinámicos. 82
9.14 Trayectorias producidas por Evolución Diferencial en el entorno con obstáculos dinámicos. . . . 83
9.15 Mapas de soluciones de una ejecución en un entorno vacío. . . . 83
9.16 Mapas de soluciones de una ejecución en un entorno con obstáculos. . . . 83
9.17 Mapas de soluciones de una ejecución en un entorno con obstáculos móviles. . 84
9.18 Trayectoria desarrollada por el controlador V2 en un entorno vacío. . . . 85
9.19 Trayectoria desarrollada por el controlador V2 en un entorno vacío. . . . 85
9.20 Trayectoria desarrollada por el controlador V3 en un entorno vacío. . . . 85
9.21 Trayectoria desarrollada por el controlador V3 en un entorno vacío. . . . 85
9.22 Trayectoria seguida por el controlador V2 en un entorno con obstáculos. . . . 86
9.23 Trayectoria seguida por el controlador V3 en un entorno con obstáculos. . . . 86

Listado de tablas

8.1 Parámetros utilizados durante las pruebas 48
8.2 Resultados obtenidos durante las pruebas. 48
8.3 Media y desviación de los resultados obtenidos durante las pruebas. 48
8.4 Resultados de las medias móviles. 49
8.5 Valor de los parámetros durante el barrido. 51
8.6 Límites HSV tipicos para diferentes colores. 57
8.7 Estimación de la distancia con la k_m 61
8.8 Estimación de la distancia tras los ajustes polinómicos. 61
8.9 Distancia de parada para un delay de 20 ms 63
8.10 Distancia de parada para un delay de 40 ms 63
8.11 Comparación de las discretizaciones en la prueba manual. 66
8.12 Distancias obtenidas a v = 10 67
8.13 Distancias obtenidas a v = 30 67
8.14 Distancias obtenidas a v = 40 67
8.15 Tiempos obtenidos de los diferentes eventos de una para a 70 cm de la cinta. . 68
8.16 Términos de la ecuación obtenidos a partir de la para a 70 cm. 68
8.17 Coficientes obtenidos tras el proceso de optimización de la función por partes. . 70
8.18 Error cuadrático medio obtenido en el proceso de optimización. 70
9.1 Configuración del simulador. 75
9.2 Configuración de los algoritmos evolutivos 77
16.1 Tabla de presupuesto. 141

Listado de códigos de programación

13.1 Implementación MAP-Elites con función Rastringin. 98
13.2 Implementación de Evolución Diferencial con función Rastringin. 101
13.3 Programa principial y visualizador del simulador. 103
13.4 Código de la red neuronal recurrente tipo Jordan. 108
13.5 Código del proceso evolutivo para MAP-Elites. 109
13.6 Código de los operadores evolutivos de MAP-Elites. 110
13.7 Código del proceso evolutivo para la Evolución Diferencial. 113
13.8 Código de los operadores evolutivos de Evolución Diferencial. 114
13.9 Código del entorno de simulación. 115
13.10 Código de la configuración del simulador. 118
13.11 Código de la prueba de los controladores en el robot real. 118

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

MEMORIA

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBREIRO, S/N

15405 - FERROL

FECHA: SEPTIEMBRE DE 2020

AUTOR: EL ALUMNO

Fdo.: DIEGO JAVIER GUTIÉRREZ BARRIO

Índice del documento MEMORIA

1 OBJETO **21**

2 ALCANCE **22**

3 FUNDAMENTOS TECNOLÓGICOS **23**

3.1 Robobo 23
3.1.1 Hardware 23
3.1.2 Software 25
3.2 Python 26
3.3 OpenCV 26
3.4 Pygame 26
3.5 Pandas 26

4 ANTECEDENTES **27**

4.1 Robótica 27
4.1.1 Robótica Colectiva 27
4.1.2 Arquitectura 28
4.1.2.1 Red neuronal artificial 28
4.1.2.2 Redes Neuronales Recurrentes 29
4.1.3 Reality Gap 30
4.2 Métodos de Optimización 30
4.2.1 Algoritmos Evolutivos 31
4.2.1.1 Evolución Diferencial 33
4.2.2 Embodied Evolution 34
4.3 Open-endedness 36
4.3.1 Novelty Search 36
4.3.2 Algoritmos QD 37
4.3.2.1 MAP Elites 37

5 NORMAS Y REFERENCIAS **40**

5.1 Disposiciones legales y normas aplicadas 40
5.2 Software utilizado 40
5.2.1 Software de Programación 40
5.2.2 Software de cálculo 40
5.2.3 Software de edición 40
5.3 Referencias 40
5.4 Otras referencias 41

6 DEFINICIONES Y ABREVIATURAS **42**

7 REQUISITOS DE DISEÑO **43**

8 DESARROLLO DE LAS SOLUCIONES **44**

8.1 Arquitectura 44
8.2 Implementación de herramientas de optimización 45
8.2.1 Benchmark Functions 45
8.2.2 Caracterización MAP-Elites 47
8.2.3 Caracterización Evolución Diferencial 51
8.3 Odometría visual 55
8.3.1 Sistema de visión 55
8.3.2 Detección de color 56
8.3.2.1 Modelo HSV 57
8.3.2.2 Análisis y procesamiento de imagen 58
8.3.3 Estimación de la distancia 60
8.3.4 Análisis de errores en movimiento 62
8.3.4.1 Análisis de la discretización 64
8.3.4.2 Análisis temporal 67
8.4 Modelado 68
8.4.1 Modelado velocidad lineal 68
8.4.2 Modelado velocidad de rotación 72
8.5 Implementación del simulador 72

9 RESULTADOS FINALES **75**

9.1 Configuración experimental 75
9.2 Navegación autónoma 75
9.3 Configuración algoritmos evolutivos 77
9.4 Resultados y análisis 77
9.4.1 Calidad de las soluciones 78
9.4.2 Diversidad de las soluciones 81
9.4.2.1 Comportamientos en entorno vacío 81
9.4.2.2 Comportamientos en entorno con obstáculos 81
9.4.2.3 Comportamientos en entorno con obstáculos en movimiento . . 82
9.4.2.4 Análisis de Mapas de Calor de MAP-Elites 83
9.5 Transferencia al entorno real 84

10 CONCLUSIONES Y TRABAJO FUTURO **87**

10.1 Conclusiones 87
10.2 Trabajo futuro 87

11 ORDEN DE PRIORIDAD ENTRE LOS DOCUMENTOS **89**

1 OBJETO

El objetivo principal de este trabajo es estudiar el uso de algoritmos evolutivos y modelos de simulación para la optimización del comportamiento de un robot real. Para ello se emplea un robot desarrollado en la propia UDC llamado Robobo. Este está basado en una plataforma móvil que transporta un smartphone, que proporciona tanto capacidades de sensorización como el procesado de la información.

El experimento debe ser desarrollado mediante un simulador para luego ser transferido al robot real, por lo que será necesario modelar el comportamiento del mismo. Con el fin de optimizar el comportamiento de los robots y resolver la tarea, se va a utilizar un algoritmo evolutivo. Además, deberá cumplir una serie de requisitos referentes a las capacidades de sensoriales del robot, y por lo tanto el correcto procesado de la información obtenida del smartphone.

2 ALCANCE

Para implementar el experimento en un entorno real, se deben abordar diversos aspectos, agrupados dentro de los campos de la inteligencia artificial y la robótica autónoma. El trabajo se divide en el estudio de las siguientes áreas:

- Estudio de los algoritmos evolutivos y sus diferentes variantes, en particular Differential Evolution y Quality Diversity Optimization.
- Implementación en el lenguaje de programación Python de los algoritmos evolutivos.
- Implementación de un simulador sencillo con la librería Pygame de Python.
- Estudio y modelado del comportamiento del robot basado en smartphone.
- Implementación del experimento y pruebas de optimización.
- Análisis de resultados y refinamiento de la implementación.
- Documentación

3 FUNDAMENTOS TECNOLÓGICOS

En este apartado se introduce la base tecnológica sobre la que se va a desarrollar este trabajo. En concreto se habla del robot Robobo, del lenguaje de programación Python, y de sus librerías asociadas

3.1. Robobo

Robobo es un robot educativo desarrollado en MINT, una spin-off de la UDC. El hardware de Robobo se compone de dos elementos: una base móvil con ruedas y un smarthphone conectado a esta.

Figura 3.1 – Robot Robobo [1].

3.1.1. Hardware

El hardware de la plataforma robótica está formado por cinco partes: la base, las ruedas, el pan, el tilt y la parte electrónica.

- **Base:** pieza que soporta los demás elementos y donde se alojan los diferentes componentes electrónicos del robot. En su parte trasera inferior, lleva un deslizador de teflón que a modo de tercer punto de apoyo, facilita su movimiento en diferentes tipos de superficie de interior.
- **Ruedas:** situadas en su parte frontal permiten el desplazamiento de la plataforma. Cada una contará con propio motor, siendo posible realizar giros a lo largo del movimiento del robot.

- **Pan:** integrada en la parte superior de la base, esta plataforma da soporte al tilt y permite girar el smartphone.

- **Tilt:** pieza que sujeta el smartphone y además del agarre que proporciona, posibilita su movimiento a lo largo del eje horizontal.

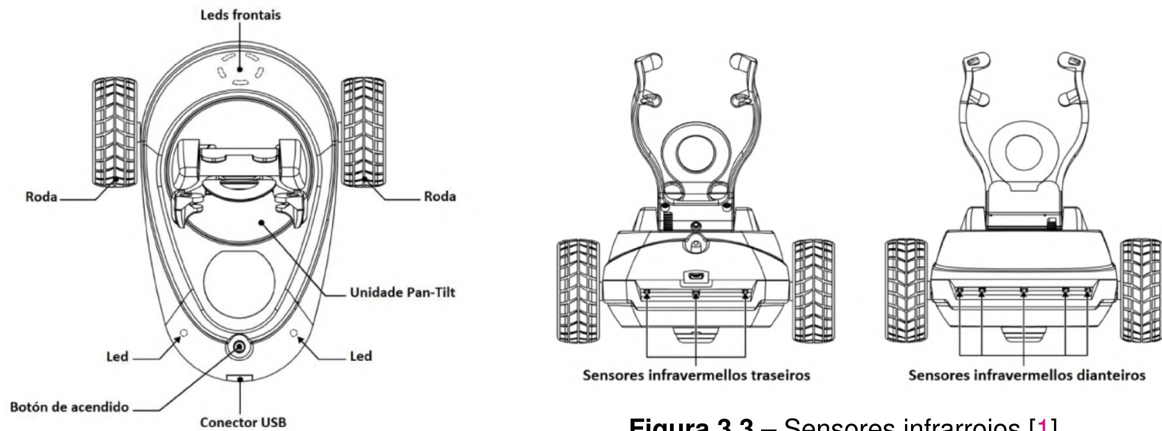


Figura 3.2 – Componentes Robobo [1].

Figura 3.3 – Sensores infrarrojos [1].

Dentro de la parte electrónica se pueden diferenciar los motores de cada rueda y los sensores infrarrojos y las luces LED.

- **Motor:** La plataforma robótica lleva incorporados cuatro motores de corriente continua del modelo TTF-N20VA-09220, equipados con caja reductora y encoders que permiten conocer la posición del motor. Este tipo de motores tienen como tensión nominal de trabajo 6V. No obstante, se puede utilizar en un rango de entre 3 y 9V. El movimiento en vacío comienza a partir de los 0.5V, pero no tiene potencia suficiente para mover el Robobo. Tanto los motores que producen el movimiento de las ruedas como el del pan tienen una reducción 150:1, ofreciendo un par nominal de 0.22 kgcm y una velocidad nominal de 81 rpm. En cuanto al tilt del smartphone su motor tiene una reducción de 1000:1 con un par nominal de 12 rpm. En cuanto al pan y el tilt, cuentan con una relación de salida de 68:10 y 54:25 respectiamente.

- **Sensores infrarrojos:** El robot dispone de ocho sensores infrarrojos, cinco localizados en la parte frontal y tres en la trasera. Estes dispositivos se emplean para la detección de objetos a distancias inferiores a 30 cm. Todos los sensores estarán orientados horizontalmente excepto uno de la parte trasera, que estará inclinado unos 15 grados hacia abajo, lo que hace posible detectar el suelo.

- **Luces LED:** La plataforma dispone de cinco LEDs en la parte frontal y dos en los lados. Permiten informar del estado del robot y a su vez la programación de estos para que se enciendan o cambien de color en determinadas circunstancias.

En cuanto a la batería el robot dispone de una del tipo LiPo de 5000mAH recargable a través de un puerto USB, ofreciendo una autonomía de entre 8 y 10 horas dependiendo de el uso.

Aparte de todos los sensores y actuadores incorporados dentro de la propia base, hay que tener en cuenta todo el potencial que conlleva el smartphone. Éste incorpora al conjunto: giroscopio, acelerómetro, sensor de luz, cámara, pantalla táctil, altavoz y micrófono. Todos estos elementos serán accesibles desde los diferentes módulos de programación, haciendo del smartphone el cerebro del robot, que se conectará al robot a través del bluetooth y al ordenador a través de una conexión WiFi.

3.1.2. Software

El robot posee una aplicación para Android. Esta sirve de conexión entre base, smartphone y ordenador. A través de ella se programa el robot e interactúa con los elementos que los componen. Además, el usuario puede interactuar con el robot que responderá según el programa implementado. En este trabajo, se utiliza una versión de la aplicación que posee la opción de streaming de vídeo. Esto permite transmitir la imagen obtenida por la cámara del móvil directamente hasta el PC para luego ser procesada. En la imagen siguiente se puede observar las distintas partes de la interfaz: la cara de Robobo, la imagen transmitida y la información relativa a los sensores del smartphone.

Figura 3.4 – Interfaz de la aplicación.

3.2. Python

Python [2] es un lenguaje de programación interpretado de alto nivel de uso extendido que destaca por su código legible, simple y su licencia abierta.

Python soporta múltiples paradigmas de programación, incluyendo la programación orientada a objetos, imperativa o funcional. Además, cuenta con un sistema tipo dinámico y de gestión de memoria automática. Uno de sus grandes atractivos es su librería de módulos, pertenecientes a múltiples disciplinas, que resultan versátiles y sencillos de implementar.

3.3. OpenCV

OpenCV [3] es una librería de código libre centrada en la visión artificial que, aunque fue desarrollada originalmente en C++, tiene numerosas interfaces para otros lenguajes de programación como es el caso de Python. La librería pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Es gratis para uso comercial y de investigación, debido a su publicación bajo la licencia BSD. Contiene más de 500 funciones enfocadas al procesamiento de imagen, reconocimiento de objetos y visión robótica. En este trabajo dicha librería proporciona los módulos pertenecientes a la detección de colores, captura, análisis y procesamiento de imagen.

3.4. Pygame

Pygame [4] es una librería del lenguaje Python que permite la creación de videojuegos o entornos de simulación en dos dimensiones, de una manera sencilla a través del manejo de sprites. Su uso es muy extendido debido a la facilidad de manejo, optimización de código y la amplia comunidad online.

Esta librería permite en el presente trabajo la programación del simulador con la que desarrollar el proceso de simulación, con las funciones de visualización de imágenes, con el fin de mostrar los elementos de la simulación, y las funciones de gestión de eventos.

3.5. Pandas

Pandas [5] es una librería software de Python extensión de Numpy para manipulación y análisis de datos. En concreto, ofrece tanto una estructura de datos como operaciones para procesarlos y representarlos gráficamente.

4 ANTECEDENTES

Como se ha descrito anteriormente, el objetivo de este TFG es desarrollar un experimento de robótica evolutiva. Por este motivo, se procede en este capítulo a revisar las diferentes tendencias en cuanto a los algoritmos evolutivos y robótica.

4.1. Robótica

La robótica es una rama de la ingeniería que se ocupa del diseño, fabricación y aplicación de los robots. Combina diversas disciplinas como la mecánica, la electrónica, la inteligencia artificial, la ingeniería de control y la física. Dentro de esta disciplina se engloba la robótica colectiva a la que pertenece el presente trabajo.

4.1.1. Robótica Colectiva

Esta tendencia consiste en el trabajo colectivo de robots que, formando un sistema multiagente, buscan resolver algún tipo de problema. Éste puede ser colectivo, es decir, que requiera más de un robot para ser resuelto, o simplemente verse beneficiado por una solución colectiva, aunque no se requiera de forma específica. Este enfoque se basa en una serie de mecanismos, como por ejemplo, objetivos colectivos, que actúan a modo de control del comportamiento de un grupo de individuos, y éstos interactúan entre sí utilizando la información de su entorno.

Aunque en última instancia la utilidad de este tipo de sistemas dependa de la tarea a realizar, existen numerosos casos en que es beneficioso utilizar este tipo de enfoque. Las principales ventajas de este tipo de sistemas serían:

- Resolución de tareas que sería imposibles para un solo robot.
- Resolución de tareas complejas mediante su división en tareas más sencillas.
- Sensorización y realización de acciones distribuidas.
- Aumento del rendimiento global.
- Mejorar la robustez y la tolerancia ante fallos.
- Versatilidad, flexibilidad y escalabilidad.
- Mayor adaptabilidad ante cambios en el entorno.

Por otra parte, también se deben tener en cuenta las posibles desventajas, ya que los problemas se vuelven más complejos y presentan diversas dificultades como las interferencias entre los agentes o la comunicación entre los mismos.

A su vez existen diversas formas de abordar la problemática de la robótica colectiva, debido en gran medida a la gran variedad de problemas existentes. Para solucionarlos se estudian desde diversos ámbitos, ya sea desde el comportamiento de animales a modelos computacionales. Entre los principales enfoques estarían:

- **Etiología:** Estudiando cómo los comportamientos de ciertos animales, así como cooperan y comunican entre ellos. Por ejemplo, la robótica de enjambre.
- **Enfoque de la sociedad humana:** Estudiando cómo funcionan las estructuras sociales humanas.
- **Modelos computacionales:** Aplicando multiprocesamiento y paralelismo.
- **Computación evolutiva:** Métodos de optimización y búsqueda basados en la evolución biológica.

4.1.2. Arquitectura

Un robot es un sistema que responde con una acción a unos determinados estímulos. Esto es posible gracias a una estructura que le permite la detectar estos estímulos, procesarlos y elegir la acción adecuada. Por lo tanto, podemos dividirlo en una serie de subsistemas:

- **Sensores**: Elementos que se encargan de recoger la información acerca del estado del robot y su entorno. Son claves para elegir la respuesta adecuada así como alertar de algún fallo en el sistema. Pueden ser de distintos tipos, los más comunes serían: radar, lidar, cámara, infrarrojos, acelerómetros...
- **Actuadores**: Elementos que llevan a cabo las acciones programas. Los tipos de actuadores más comunes son los motores eléctricos y los actuadores lineales.
- **Unidad de control**: Compuesto por los elementos computacionales y el software que regula el comportamiento. Por ejemplo, lógica difusa o redes neuronales, reguladores o sistema de reglas. La complejidad de los controladores puede variar dependiendo de la tarea y el sistema. Se suele utilizar técnicas de la teoría de control, motion-planning, lógica difusa o redes neuronales.

Dado que en el presente trabajo el controlador utilizado es una red neuronal, se explica en mayor profundidad en el siguiente apartado.

4.1.2.1. Red neuronal artificial

Una red neuronal es un modelo computacional inspirado en la estructura del sistema nervioso de los seres humanos. La arquitectura se forma conectando múltiples unidades llamadas neuronas artificiales. Estas formarán un sistema que se adaptará según un algoritmo de optimización. Este irá ajustando unos pesos que actuarán sobre las conexiones entre neuronas, el objetivo final es que alcance los requerimientos del problema sobre el que se aplica.

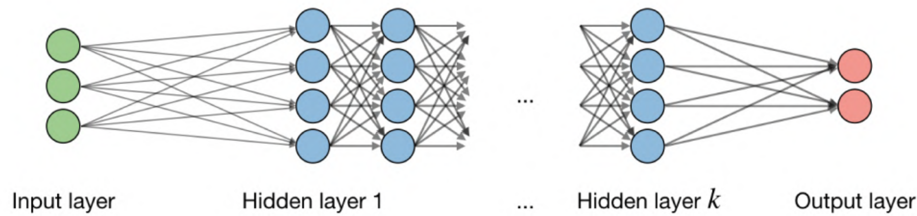


Figura 4.1 – Estructura red neuronal.[7]

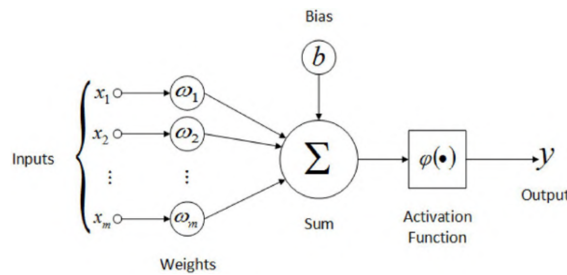


Figura 4.2 – Modelo neurona artificial.[8]

4.1.2.2. Redes Neuronales Recurrentes

Las redes neuronales recurrentes (RNNs) son una clase de redes neuronales que integran bucles de realimentación, permitiendo que a través de estos la información persista durante algunas iteraciones. Este tipo de redes tratan datos secuenciales de forma eficiente. Utilizar este tipo de redes conlleva una serie de ventajas como la posibilidad de procesar entradas de cualquier longitud, que el modelo no crezca según el tamaño de entrada o que los pesos son compartidos a través del tiempo. Como desventajas, este tipo de redes son computacionalmente más lentas y es difícil acceder a la información más antigua. Según el tipo de conexionado de la realimentación se pueden distinguir dos tipos de arquitecturas: Jordan y Elman.

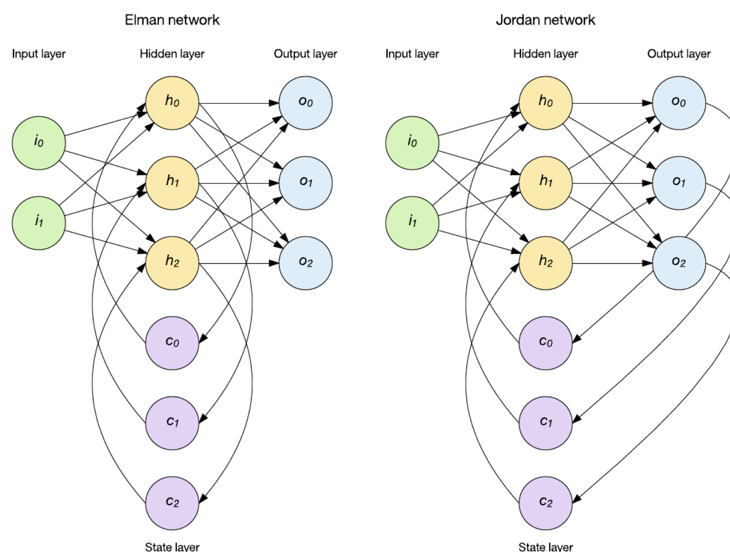


Figura 4.3 – Estructura de los tipos Elman y Jordan.[6]

4.1.3. Reality Gap

El ser humano es capaz de adquirir habilidades y destrezas a los largo de su vida, como consecuencia sus interacciones con el mundo. Del mismo modo los robots pueden utilizar procesos similares para aprender a realizar diferentes tareas. Esto no está exento de obstáculos. Por un lado, es necesario aportar en un corto período de tiempo toda la experiencia necesaria. Por otro, supone un coste computacional muy alto, haciendo el aprendizaje en un entorno real lento y costoso. Para ello se utilizan los simuladores, donde gracias a las técnicas de paralelización y modelización es posible entrenar controladores de una forma más eficiente.

El problema de estos simuladores, es que los controladores desarrollados fallan a la hora de generalizar el problema para un entorno real, por lo que cuando son transferidos no funcionan correctamente. Esto se puede deber a que los modelos físicos utilizados no representan fielmente las perturbaciones que pueden ocurrir en el mundo real. Además, algunos de los métodos de aprendizaje utilizados son capaces de explotar los fallos del simulador, dependiendo en gran medida de estos para obtener buenos resultados. Esta dificultad a la hora de transferir la experiencia obtenida durante la simulación al mundo real, es comúnmente llamada *reality gap* y es uno de los factores clave a la hora de realizar experimentos en robótica.

Para poder reducir el *reality gap*, es necesario modelizar lo mejor posible el comportamiento del robot y las características del entorno de funcionamiento real. A parte de esta tendencia a crear simuladores más precisos, existen otras alternativas. Por ejemplo, el *transferability approach* utiliza experiencia obtenida en el entorno real para predecir la precisión de los controladores generados durante el entrenamiento. De esta forma, se puede discriminar entre estos según sean o no transferibles al robot real.

4.2. Métodos de Optimización

Durante la realización del experimento, es necesario optimizar el controlador para que el robot pueda realizar la tarea a la que se enfrente eficientemente. Este proceso conlleva una serie de dificultades. En primer lugar, la información obtenida del entorno es limitada. En segundo lugar, el comportamiento del robot viene dado por una gran cantidad de parámetros y, en tercer lugar, no se trata de un problema lineal. Por estas razones se recurre a la utilización de algoritmos evolutivos, que son idóneos para este tipo de problemas no lineales con espacios de búsquedas muy extensos.

4.2.1. Algoritmos Evolutivos

Los algoritmos evolutivos (EAs) son métodos de optimización estocásticos que imitan la evolución biológica y están integrados dentro de la inteligencia artificial. Estos algoritmos han sido desarrollados como una alternativa a técnicas matemáticas clásicas a la hora de conseguir soluciones óptimas a problemas de optimización de gran escala. El flujograma que describe el funcionamiento de la gran mayoría de los algoritmos evolutivos es el siguiente:

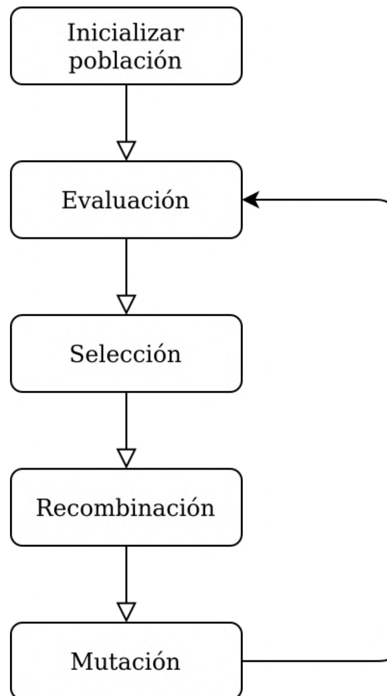


Figura 4.4 – Algoritmo evolutivo típico

Utilizar este tipo de estrategias frente a otro tipo de métodos presenta una serie de ventajas:

- **Simplicidad conceptual:** Aunque existan diferentes tipos de algoritmos, cada uno con sus características, en su gran mayoría suelen responder a un lógica y flujo común.
- **Aplicabilidad:** Este tipo de algoritmos pueden ser aplicados a casi cualquier problema que pueda formularse como un problema de optimización mediante una función objetivo. Lo que necesita es una estructura para representar la solución, un índice para evaluar a esta y un operador de variación para generar nuevas soluciones.
- **Potencial para formar métodos híbridos:** Los algoritmos evolutivos pueden verse beneficiados al incorporar otro tipo de métodos. Estos pueden pertenecer a otra rama diferente o simplemente ser otro evolutivo. Por ejemplo, se puede servir la poblaciónr resultante de un algoritmo y continuar el proceso de optimización. Además, la computación evolutiva puede ser usada para optimizar redes neuronales, como en la Neuroevolución, o sistemas borrosos.

- **Paralelismo:** Debido a que se trabaja con poblaciones, la evaluación de los individuos se puede realizar en paralelo, acortando el tiempo de computación necesario.
- **Robustez:** Los algoritmos evolutivos pueden ser usados para adaptar soluciones a cambios en el entorno, siendo la población de individuos evolucionados solo una base para una futura adaptación.

Dentro de los algoritmos evolutivos existe una variedad de diferentes posibilidades que están basadas en diferentes comportamientos evolutivos y el comportamiento social de las especies. El primer algoritmo de este tipo es el algoritmo genético, después con el fin de mejorar el tiempo de procesado, mejorar las soluciones o evitar el problema de quedarse estancado en un óptimo local, se desarrollan otro tipo de algoritmos evolutivos. A continuación, se explican y comparan una serie de técnicas de este tipo:

- **Algoritmo genético:** Inspirados en el proceso genético de los organismos, representan en forma de cadena ("cromosoma") la solución a un problema. Este a su vez contiene una serie de elementos llamados "genes", que portan el conjunto de valores de las variables a optimizar. La calidad de cada cromosoma es determinado mediante su evaluación con una función objetivo. Con el fin de simular la selección natural, los mejores cromosomas intercambian información para producir descendencia. Después, estas soluciones son utilizadas para evolucionar la población si proporcionan mejores soluciones que los peores miembros de la población. La mutación consiste en escoger aleatoriamente un cromosoma y modificar alguno de sus genes. Esto permite ir modificando progresivamente la población hacia un cromosoma óptimo.
- **Algoritmo memético:** Extensión de los algoritmos genéticos tradicionales que combinan elementos de otros métodos metaheurísticos, como por ejemplo la búsqueda local.
- **PSO (Particle swarm optimization):** Inspirado por el comportamiento social de bandadas de aves migratorias que tratan de alcanzar un destino desconocido. Cada solución es un pájaro en la bandada y es llamado "partícula". A diferencia de otros métodos no producen descendencia si no que lo que se evoluciona es su comportamiento. Lo que se hace es imitar la comunicación entre los pájaros. Cada uno acelera hacia el mejor con una velocidad según la posición actual. Después cada uno de ellos investiga el espacio de búsqueda desde su nueva posición.
- **Ant colony optimization:** Similar a PSO, evoluciona según el comportamiento de las hormigas cuando hacen el camino desde el nido hasta la comida evitando un obstáculo. Conforme las hormigas hacen su camino van dejando feromonas que durarán un tiempo determinado, las otras hormigas seguirán a estas y, el camino más corto será el que posea más feromonas.

- **Shuffled frog leaping algorithm:** Combina los beneficios de los algoritmos meméticos y los PSO. La población consiste en un conjunto de ranas (soluciones) que será repartidos en subconjuntos ("memeplexes"). Cada uno de estos subconjuntos realizará una búsqueda local. Los "memeplexes" contendrán ideas, que pueden ser influenciadas por las ideas de otras ranas, y evolucionar a través de un proceso de mimetización. Después de un número de pasos determinados las ideas serán mezcladas entre los "memeplexes". El proceso de búsqueda y mezcla continuará hasta alcanzar un criterio de convergencia.

4.2.1.1. Evolución Diferencial

Propuesto por Rairner Storn y Kenneth Price, la evolución diferencial surge con el objetivo de mejorar el rendimiento de los algoritmo evolutivos tradicionales [7]. Este algoritmo al igual que otros pertenecientes a esta categoría, mantiene una población de candidatos, estos, se cruzan y mutan para producir nuevos individuos que serán evaluados. Después, la elección entre los ya existentes y los nuevos se hará según como se hayan desempeñado en el problema.

Algorithm 1 Differential Evolution

```
1:  procedure DIFFERENTIAL EVOLUTION
2:      x ← random_population()          ▷ Inicialización a través de NP soluciones aleatorias
3:      for iter = 1 ← I do
4:          for ind = 1 ← NP do
5:              a, b, c ← random_selection(χ)      ▷ Selección aleatoria dentro de la población
6:              if r < CR then
7:                  V = a + F × (b − c)
8:                  if performance(V) < performance(x) then
9:                      x ← V
10:                 end if
11:             end if
12:         end for
13:     end for
14:     return x
15: end procedure
```

A diferencia del esquema básico de un algoritmo evolutivo, la idea fundamental de este método es la creación de vectores de prueba. Estos vectores serán generados haciendo la diferencia ponderada de dos miembros de la población y sumándosela a un tercero 4.1. Después se recombina con otro ya existente para formar un nuevo candidato. Si este tiene un mejor rendimiento que el original ocupa su lugar. En la figura 4.5 se muestra de forma esquemática este proceso. Como ventajas, este tipo de algoritmos son fáciles de implementar en cualquier lenguaje, tiene pocos parámetros de configuración y tiene buenas propiedades de convergencia.

$$v = x_1 + F(x_2 + x_3) \tag{4.1}$$

$$x_1, x_2, x_3, \in [0, NP - 1], \in N, F > 0 \tag{4.2}$$

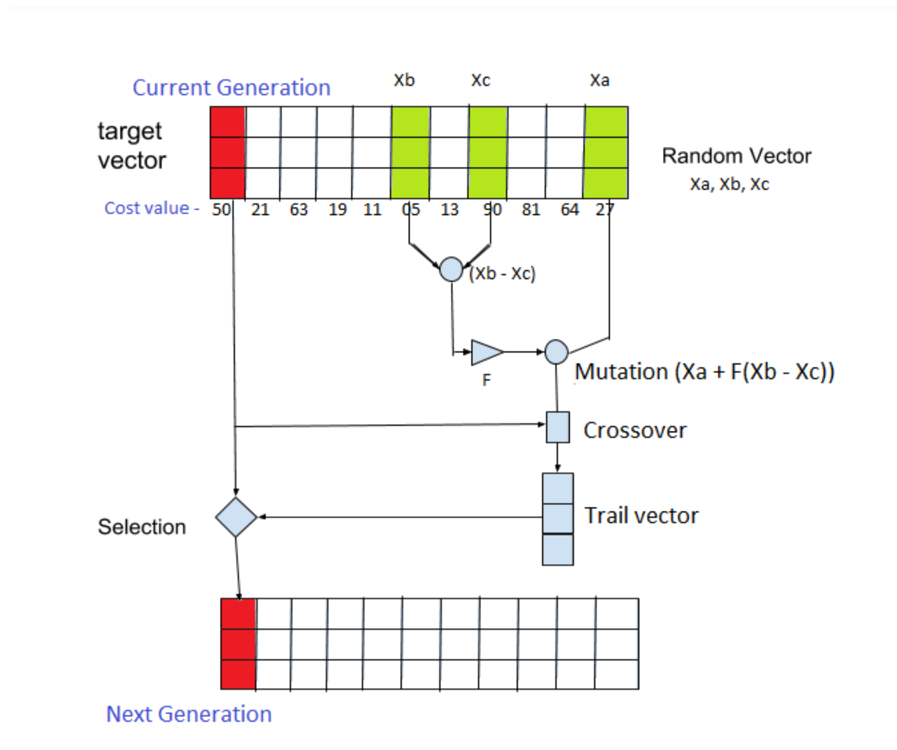


Figura 4.5 – Algoritmo evolutivo diferencial[9].

4.2.2. Embodied Evolution

Existen una gran variedad de formas de abordar la computación evolutiva aplicada a sistemas multi-robot. Esto hace que que sea un campo extenso y heterogéneo, donde se han realizado diversas propuestas con el objetivo de clarificar los diferentes enfoques existentes. Por ejemplo, se puede partir de la clasificación realizada por Watson et al., basada en dónde y cómo se produce la evolución de los individuos [8]. En esta clasificación se diferencia entre sistemas donde se evalúan los controladores en simulación, de aquellos donde la evaluación está incrustada en los robots reales. Dentro de este tipo, se diferencia entre las variantes en serie, donde los los controladores cambian de forma secuencial, y en paralelo, en donde cada robot evalúa un genotipo. Por último, distingue dentro de esta última, una variante centralizada y otra encapsulada, donde cada robot lleva un población entera, y otra distribuida, en donde cada robot lleva un código genético.

Otro enfoque es el propuesto por Eiben et al., que se centra en tres aspectos del proceso evolutivo: el temporal (cuándo se realiza, en tiempo de diseño o operación), espacial (dónde, dentro o fuera de los robots) y el modo (cómo, de forma distribuida o encapsulada) [4].

En la evolución *offline* el desarrollo de los controladores ocurre antes de que los robots empiecen su período de funcionamiento real. En la evolución *online* el desarrollo ocurre en el período de funcionamiento y es un proceso continuo durante este tiempo. Es decir, la diferencia radica en el momento que dejamos de utilizar los operadores evolutivos, después o en el momento de empezar el período de funcionamiento. En cuanto al aspecto espacial, se distingue *on-board* u *off-board*. En el primer caso, los operadores evolutivos actúan exclusivamente dentro del propio hardware del robot, en el caso contrario, estos actuarán desde un hardware externo al robot como una computadora.

Por último, se considera como se aplican los operadores evolutivos, si de una forma distribuida o encapsulada. En primer lugar, en el caso distribuido, cada robot contiene un genotipo y es controlado por el fenotipo correspondiente. Además, este podrá ,de forma autónoma, producir descendencia. En cambio, el modelo encapsulado conlleva que cada robot tendrá su propio algoritmo evolutivo con su población genotípica correspondiente. Estos algoritmos pueden ser iguales o diferentes respecto a los demás robots del sistema. Cada robot gestionará su población a través de un mecanismo de time-sharing.

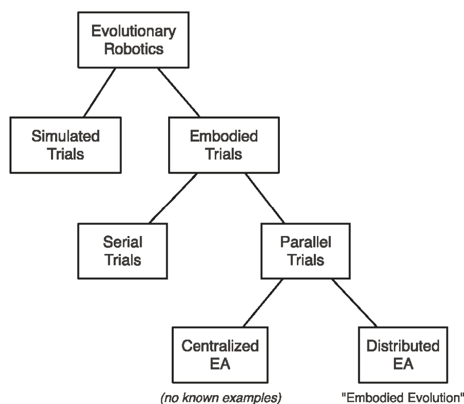


Figura 4.6 – Clasificación Watson et al.[8].

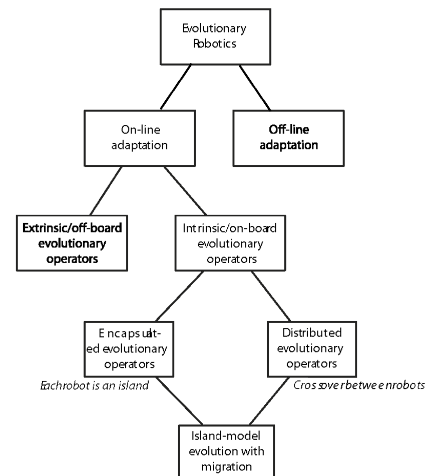


Figura 4.7 – Clasificación Eiben et al..[4]

En este trabajo se va a aplicar la definición propuesta por Bredeche et al., donde se considera Embodied Evolution como un paradigma donde el proceso evolutivo es implementado en un sistema multi-robot (dos robots o más) [1]. Estos sistemas deben poseer una serie de características:

- **Descentralización:** No hay una autoridad central que selecciona los padres o los individuos para ser reemplazados. En cambio, los robots evalúan su propia calidad, intercambia y seleccionan el material genético de forma autónoma con la información disponible.
- **On-line:** Los controladores se modifican sobre la marcha, mientras los robots van realizando sus acciones. La evolución ocurre durante el período de funcionamiento y dentro del entorno donde se realiza la tarea y continúa tras el desarrollo de los robots.

- **Paralelismo:** Ya sea un tarea colaborativa o no, la población será un sistema multi-robot, que se encuentra en el mismo entorno y interactúa para intercambiar material genético.

Este paradigma añade un nuevo operador evolutivo a los ya existentes (selección, variación y reemplazamiento), introduciendo así el apareamiento. Este nuevo operador es una acción donde dos o más robots deciden enviar o recibir material genético. Este material no tiene que ser utilizado para generar descendencia. Esto dependerá tanto del algoritmo como del comportamiento desarrollado por los robots durante el proceso.

4.3. Open-endedness

Los algoritmos evolutivos tradicionales están inspirados en la naturaleza, interpretando a esta como un proceso de simple optimización. Esto puede producir una serie de problemas ya que, cuanto más complejo sea el objetivo, más difícil es identificar los pasos para alcanzarlo, pudiendo estancarse en un mínimo local. La evolución natural en particular es un caso diferente, ya que es una búsqueda en la que simultáneamente se explota un nicho y quiere encontrar otros por el medio de la exploración de nuevas soluciones. Por lo tanto, tiene como objetivo tanto la calidad como la diversidad. Además, la evolución no es proceso finito, si no que está en constante funcionamiento, produciendo constantemente nuevas soluciones más complejas y diversas. Estas características fundamentales que posee la evolución natural, suponen una nueva base sobre la que fundar el alcance de los algoritmos evolutivos. Lehman et al. desarrollaron esta tendencia a través del Novelty Search, en el que introducen algoritmos con búsqueda de diversidad en vez de la calidad de los comportamientos [5]. Con el objetivo de juntar ambas tendencias se han desarrollado los llamados Quality Diversity algorithms (QD), que buscan dar un conjunto de comportamientos de altas prestaciones y a su vez que exista un alto grado de diversidad en estos.

4.3.1. Novelty Search

Este algoritmo busca premiar comportamientos nuevos en vez del rendimiento. Esto se hace a través del concepto de “novelty”, que medirá como de diferente es el comportamiento del individuo respecto individuos anteriores. Para ello será necesario caracterizar los comportamientos y tenerlos almacenados para su posterior comparación. Si se trabaja con un solo robot bastaría con sustituir la calidad por el concepto de “novelty”. Para medir cómo de diferente es el comportamiento de un individuo respecto a otros, se crea un archivo de comportamientos donde poder compararlos. Aquí los comportamientos ya caracterizados son comparados en el espacio novelty. Es decir, las zonas de este espacio que sean más densas y, por lo tanto contengan más comportamientos, tendrán menos diversidad y se las recompensará menos. Un buen indicador sería la dispersión de cada individuo teniendo en cuenta el archivo, que incluye los comportameintos vistos anteriormente, y la población actual a evaluar.

4.3.2. Algoritmos QD

Los algoritmos pertenecientes a esta categoría deben poseer las dos cualidades, es decir, presión para descubrir más nichos de comportamiento y una tendencia a mejorar el rendimiento de estos. Ha habido diferentes propuestas para juntar algoritmos como el novelty search con el concepto de función objetivo, normalmente a través de una algoritmo multi-objetivo. Sin embargo, esto introduce el concepto de calidad global, en vez de buscar la explotación dentro de un nicho. Dentro de este grupo de algoritmos que cumplen estos requisitos destacan dos: Novelty Search con Local Competition y MAP-Elites [3]. El primero como su nombre indica junta el Novelty Search con el concepto de competición local, que posibilita asignar alguna recompensa por rendimiento. Esta competición se realiza entre los individuos próximos de la población. Es decir, aquellos individuos con comportamientos similares. El segundo se explica con mayor profundidad en el siguiente apartado.

4.3.2.1. MAP Elites

Introducido por Mouret et al. surge como alternativa para evitar el problema del óptimo local que afecta a los algoritmos de optimización [6]. Esto se debe a que estos algoritmos se basan en realizar cambios aleatorios en buenas soluciones para llegar a otras mejores. Algunos tipos de problemas pueden presentar algún aspecto engañoso. Esto significa que llegar al óptimo global del problema no es un proceso trivial. Al contrario, para poder alcanzarlo se tiene que recorrer valles de puntos de bajo rendimiento y puede que el algoritmo se quede estancado en estos. Como solución a esto proponen mediante una serie de modificaciones premiar la diversidad, ya sea bien por medio de:

- Aumentar el ratio de mutación cuando se estanque el rendimiento.
- Incorporar el concepto de novelty.
- Cambiando la estructura de la población.

Estas propuestas tienen un efecto positivo a la hora de potenciar la diversidad, pero al final acaban por centrarse en una o dos soluciones concretas en vez de obtener un conjunto de ellas. Además, en el caso de tener un espacio de búsqueda muy grande no sería recomendable utilizar novelty, ya que para obtener una solución aceptable sería computacionalmente muy costoso. La exploración de espacios de búsqueda también sería un punto importante dentro la optimización, pese a esto, no se encuentran a menudo estas búsquedas, ya que normalmente se trata de espacio de muy alta dimensionalidad donde no es factible aplicar un algoritmo de reducción dimensional para poder ver propiedades interesantes para el usuario.

MAP-Elites surge con el objetivo de cumplir con estos requisitos. Este algoritmo busca en un espacio de alta dimensionalidad (espacio de búsqueda) para encontrar la mejor solución en cada punto de un espacio de baja dimensionalidad (espacio característico) donde estarán las características definidas por el usuario. A continuación, se describe el funcionamiento de dicho algoritmo.

En primer lugar, el usuario escoge una forma de medir la calidad que evaluará la solución *x*. Después, se debe definir el número de dimensiones que definirán el espacio de características. Cada dimensión será discretizada basándose en los criterios del usuario o en los recursos computacionales disponibles. Por ejemplo, esta granularidad podría ser fija y definida a mano, o variable según las capacidades computacionales lo permitan. Como resultado, se tendrá un mapa dividido en celdas donde se guardarán los individuos y se conocerá la calidad y el valor en cada una de las dimensiones.

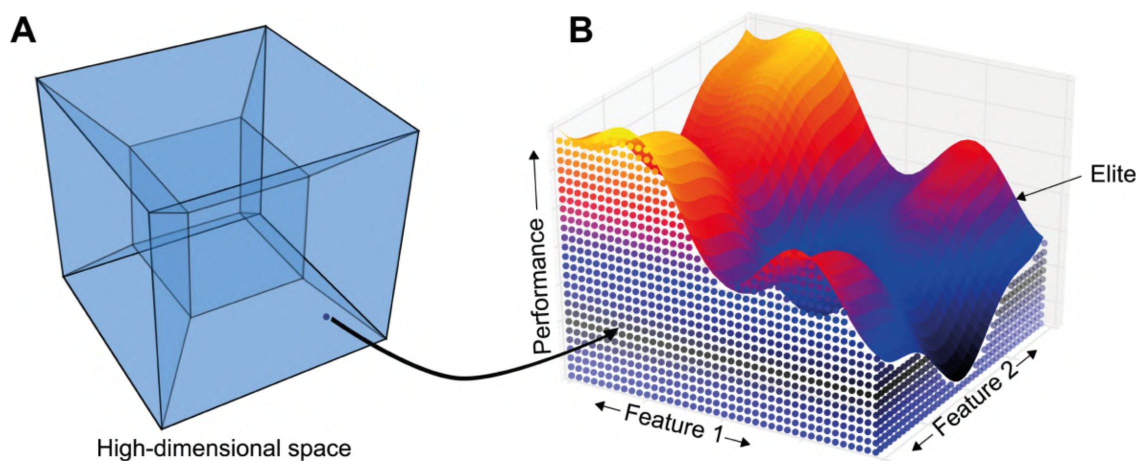


Figura 4.8 – Espacio de búsqueda y espacio de características[6].

Dada la discretización, el algoritmo buscará la mejor solución en cada celda del espacio de características. Este proceso será realizado en el espacio de búsqueda y empezará por generar una serie de individuos aleatorios. Una vez hecho esto serán mapeados dentro de la celda correspondiente según las características elegidas. A partir de aquí y en las siguientes generaciones de individuos, se aplicarán los distintos operadores ya sean de mutación, cruce o ambos. Una vez obtenidos los nuevos individuos se volverá a mapearlos, en el caso de pertenecer a una celda vacía será introducido automáticamente y en el caso de estar ocupada se comparará con este y se elegirá al que tenga mejor calidad. La condición de finalización puede elegirse entre varias alternativas, como llenar un determinado porcentaje de celdas, llegar a un rendimiento determinado o simplemente un límite temporal. Como resultado, se obtendrá un conjunto de soluciones diversas según las características definidas previamente.

Algorithm 2 MAP-Elites

1: **procedure** MAP-ELITES
2: $(\rho \leftarrow \emptyset, \chi \leftarrow \emptyset)$ ▷ Crea un mapa N-dimensional vacío, contiene calidad y soluciones.
3: **for** $iter = 1 \leftarrow$ *I* **do**
4: **if** $iter <$ *G* **then** ▷ Inicialización a través de G soluciones aleatorias
5: $x' \leftarrow random_solution()$
6: **else**
7: $x \leftarrow random_selection(\chi)$ ▷ Selección aleatoria de una elite *x* del mapa χ
8: $x' \leftarrow random_variation(x)$ ▷ Crea *x'* a partir de mutar/cruzar *x'*
9: **end if**
10: $b' \leftarrow feature_descriptor(x')$ ▷ Simula *x'* y guarda la solución y calidad
11: *p'* $\leftarrow performance(x')$
12: **if** $\rho(b') = \emptyset$ or $\rho(b') <$ *p'* **then** ▷ Si la celda está vacío o tiene una calidad menor
13: $\rho(b') \leftarrow$ *p'* ▷ se introduce la calidad y la solución en ella
14: $\chi(b') \leftarrow x'$
15: **end if**
16: **end for**
17: **return** map (ρ and χ)
18: **end procedure**

Utilizar este algoritmo implica una serie de beneficios que los hacen diferente de otro tipo de métodos:

- Crear diversidad según las características escogidas.
- “Iluminar” el rendimiento potencial del espacio de características, mostrando las relaciones entre las dimensiones y la calidad de las soluciones.
- Encontrar mejores soluciones que otros algoritmos en espacios de búsqueda más complejos, porque explora más del espacio de búsqueda.
- Devuelve un conjunto de soluciones diversas contenidas en un mapa que describe su posición en el espacio característico, esto puede ser utilizado para crear nuevos algoritmos o mejorar el rendimiento de alguno ya existente.

Cully et al. se inspiran en las capacidades de los animales para adaptarse tras sufrir una lesión y aplican este algoritmo a un robot hexápodo[2]. Éste antes de su funcionamiento real, utilizará el algoritmo para obtener un conjunto de soluciones de altas prestaciones. Utilizado a modo de intuición, este conjunto de soluciones servirá al agente para guiarse a través de un proceso de ensayo-error, siendo el objetivo final adaptarse a los posibles daños sufridos en su funcionamiento. En este ejemplo se pueden observar las características fundamentales que definen a los QD algorithm. En primer lugar, se produce un conjunto amplio de soluciones, y después el proceso no cesa con su obtención, si no que sirve como base para un proceso de adaptación.

5 NORMAS Y REFERENCIAS

5.1. Disposiciones legales y normas aplicadas

Al presente trabajo aplica el Reglamento del trabajo de fin de grado de la Escuela Universitaria Politécnica de la Universidad de A Coruña.

5.2. Software utilizado

5.2.1. Software de Programación

- VSCode

5.2.2. Software de cálculo

- Microsoft Excel: Realización de gráficas, métodos de optimización de modelos y análisis de datos.

5.2.3. Software de edición

- Overleaf
- Adobe Reader

5.3. Referencias

[1] N. Bredeche, E. Haasdijk, and A. Prieto, “Embodied evolution in collective robotics: A review,” *Frontiers Robotics AI*, vol. 5, no. FEB, 2018.

[2] A. Cully, J. Clune, D. Tarapore, and J. B. Mouret, “Robots that can adapt like animals,” *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

[3] A. Cully and Y. Demiris, “Quality and Diversity Optimization: A Unifying Modular Framework,” *IEEE Transactions on Evolutionary Computation*, vol. 22, no. 2, pp. 245–259, 2018.

[4] A. E. Eiben, E. Haasdijk, and N. Bredeche, “Embodied, On-line, On-board Evolution for Autonomous Robotics,” *Symbiotic Multi-Robot Organisms: Reliability, Adaptability, Evolution*, pp. 361–382, 2010. [Online]. Available: http://www.springer.com/engineering/mathematical/book/978-3-642-11691-9

[5] J. Lehman and K. O. Stanley, “Exploiting open-endedness to solve problems through the search for novelty,” *Artificial Life XI: Proceedings of the 11th International Conference on the Simulation and Synthesis of Living Systems, ALIFE 2008*, pp. 329–336, 2008.

[6] J.-B. Mouret and J. Clune, "Illuminating search spaces by mapping elites," pp. 1–15, 2015. [Online]. Available: http://arxiv.org/abs/1504.04909

[7] R. Storn and K. Price, "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997. [Online]. Available: https://doi.org/10.1023/A:1008202821328

[8] R. A. Watson, S. G. Ficici, and J. B. Pollack, "Embodied Evolution: Distributing an evolutionary algorithm in a population of robots," *Robotics and Autonomous Systems*, vol. 39, no. 1, pp. 1–18, 2002.

5.4. Otras referencias

[1] *Robobo*, MINT, [Fecha de la consulta]. Disponible en: `https://theroboboproject.com/`

[2] *Python*, Python Software Foundation, [Fecha de la consulta]. Disponible en: `https://www.python.org/`

[3] *OpenCV (Open Source Computer Vision)*, Itseez, Inc., [Fecha de la consulta]. Disponible en: `https://opencv.org/`

[4] *Pygame*, Pygame Community, [Fecha de la consulta]. Disponible en: `https://www.pygame.org/news`

[5] *Pandas*, Pandas Development Team, [Fecha de la consulta]. Disponible en: `https://pandas.pydata.org/`

[6] *Recurrent neural networks deep dive*, IBM, [Fecha de la consulta]. Disponible en: `https://developer.ibm.com/technologies/artificial-intelligence/articles/cc-cognitive-recurrent-neural-networks/`

[7] *Cheatsheet Deep Learning*, Standford, [Fecha de la consulta]. Disponible en: `https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning`

[8] *The Artificial Neural Networks Handbook: Part 4*, Medium, [Fecha de la consulta]. Disponible en: `https://medium.com/@jayeshbahire/the-artificial-neural-networks-handbook-part-4-d2087d1f583e`

[9] *Differential Evolution — Sounds cool right!*, Medium, [Fecha de la consulta]. Disponible en: `https://medium.com/@b516002/differential-evolution-sounds-cool-right-a5c245cbe6d9`

[10] *Ackley Function*, Virtual Library of Simulation Experiments, [Fecha de la consulta]. Disponible en: `https://www.sfu.ca/~ssurjano/ackley.html`

[11] *Rastrigin Function*, Virtual Library of Simulation Experiments, [Fecha de la consulta]. Disponible en: `https://www.sfu.ca/~ssurjano/rastr.html`

[12] *Convertir de HSV a ESPACIO de Color RGB*, MathWorks, [Fecha de la consulta]. Disponible en: https://la.mathworks.com/help/images/convert-from-hsv-to-rgb-color-space.html

6 DEFINICIONES Y ABREVIATURAS

Todas las definiciones y abreviaturas presentes a lo largo del TFG se recogen a continuación:

- **BSD**: Licencia de software libre otorgada principalmente para los sistemas Berkeley Software Distribution.
- **FOV**: Ángulo que se puede percibir del mundo virtual generado en el dispositivo de visualización asociado a la posición del punto de visión

7 REQUISITOS DE DISEÑO

A continuación se describen los requisitos para el desarrollo de las distintas partes que conforman el experimento que será realizado en este trabajo:

- Se deberá utilizar las capacidades sensoriales de alto nivel del Robobo, en este caso la cámara, y las capacidades de comunicación del mismo, en concreto WI-FI y Bluetooth.
- Se deberá utilizar el lenguaje de programación Python, tanto para la implementación de las herramientas de optimización y el simulador como la para los scripts pertinentes del robot Robobo. Ya que es uno de los lenguajes más utilizados tanto en robótica como inteligencia artificial y es de código abierto, facilitando la incorporación de nuevas funcionalidades en el futuro.
- El simulador debe implementarse respetando las características de los elementos del experimento en un entorno real. Por lo tanto, se debe modelar el comportamiento del robot, estudiando desde su movimiento hasta los diferentes tiempos de respuesta originados en la comunicación entre base, smartphone y PC. Así mismo debe modelarse sus parámetros para asemejarse en la medida de lo posible al lugar donde se realice el experimento.
- Todas las librerías y comportamientos desarrollados se deberán realizar en forma de script y deberán estar correctamente documentados para su fácil comprensión.

8 DESARROLLO DE LAS SOLUCIONES

En este capítulo se desarrolla la implementación de los diferentes algoritmos, herramientas y modelos para realización del experimento, así como la metodología utilizada durante el trabajo. Se comienza con la definición la arquitectura que dará forma a la solución implementada, después se desarrolla cada una las herramientas utilizadas.

En primer lugar, se procede a la implementación y estudio de los algoritmos de optimización aplicados a funciones sintéticas. Se continúa el modelado del sistema de sensorización, en particular la odometría del robot, que abarca desde el sistema de visión y la estimación de los posibles errores durante el funcionamiento del robot. Finalmente, se realiza el modelado de los actuadores del Robobo, para su posterior implementación en la solución final, en particular se trata el movimiento del robot.

8.1. Arquitectura

En esta sección se describe como son los componentes que forman el sistema y como se relacionan entre sí. Hay que diferenciar dos tipos de arquitecturas, una referente la configuración experimental y otra al sistema robótica que va a ser implementado.

Arquitectura experimental:

- **PC**: La computadora procesa la información, es decir, trata las imágenes, optimiza el controlador y lo ejecuta.
- **Robot**: Se encarga de realizar la tarea o resolver el problema, envía la información obtenida del entorno al PC.
- **Entorno**: Lugar donde se encuentra el robot y debe realizar la tarea.

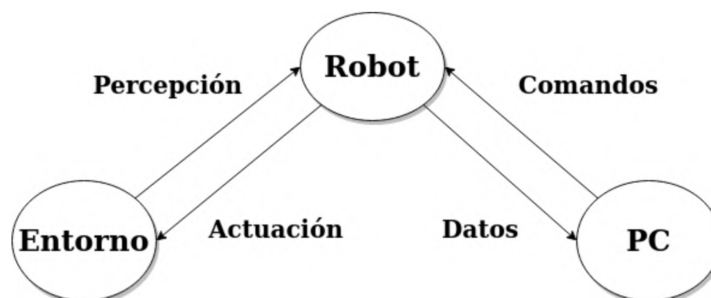


Figura 8.1 – Arquitectura del experimento

A partir de la estructura descrita en la sección 4.1.2 se define la arquitectura del sistema robótico a implementar en el presente trabajo.:

- **Sensores**: Encargados de captar la información proporcionada por el entorno.
- **Actuadores**: Es la parte que actúa sobre el entorno.
- **Controlador**: Con la información del entorno decide que acción tomar sobre el entorno.
- **Optimizador**: Encargado de optimizar los parámetros del controlador.

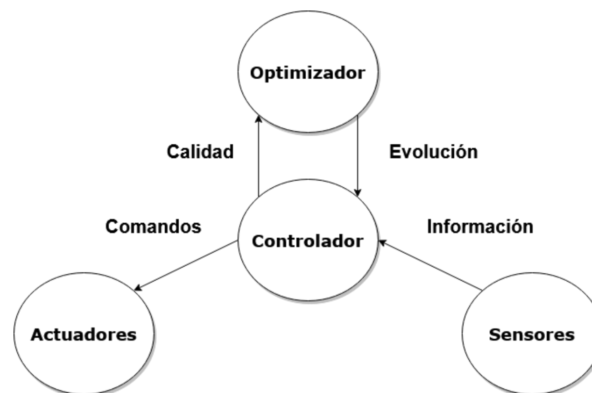


Figura 8.2 – Arquitectura del sistema robótico

8.2. Implementación de herramientas de optimización

En esta sección se va a realizar un análisis de los diferentes algoritmos evolutivos a implementar, en este caso evolución diferencial y MAP-Elites. Primero se exponen las funciones sintéticas, que serán el campo de aplicación. Sobre esto se procede al estudio de los parámetros que afectan a la respuesta.

8.2.1. Benchmark Functions

Para estudiar el funcionamiento de los algoritmos de optimización se utilizan una serie de funciones de testeo. También llamadas funciones sintéticas o artificial landscapes, muestran las distintas dificultades a las que se pueden enfrentar. Por lo tanto, son un aspecto importante a la hora de estudiar y comparar los algoritmos de optimización, permitiendo conocer que propiedades presentan y como de útiles sería su aplicación a un problema real. Existe una gran variedad de funciones de diferentes dificultades y para distintos tipos de optimización.

A continuación, se exponen dos de las funciones sintéticas habituales, así como su representación para dos dimensiones, aunque finalmente se vaya a utilizar solamente una para realizar las pruebas:

- **Ackley:** Función no convexa que se caracteriza por tener una región exterior más plana y una agujero en el centro. Supone un reto debido a sus múltiples mínimos locales de la zona plana.

$$f(x) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos cx_i\right) + a + exp(1) \tag{8.1}$$

  donde d es el número de dimensiones del problema, $x \in [-32{,}768, 32{,}768]$, para $i = 1, ..., d$, los términos a, b y c son constantes y tiene el mínimo global en $x = (0, ..., 0)$.

- **Rastrigin:** Función no convexa y multimodal, cuya dificultad radica en el hecho de que tenga una gran cantidad de mínimos locales y tenga un espacio de búsqueda tan grande.

$$f(x) = 10d + \sum_{i=1}^{d}[x_i^2 - 10\cos(2\pi x_i)] \tag{8.2}$$

  donde d es el número de dimensiones del problema, $x \in [-5{,}12, 5{,}12]$, para $i = 1, ..., d$, y tiene el mínimo global en $x = (0, ..., 0)$.

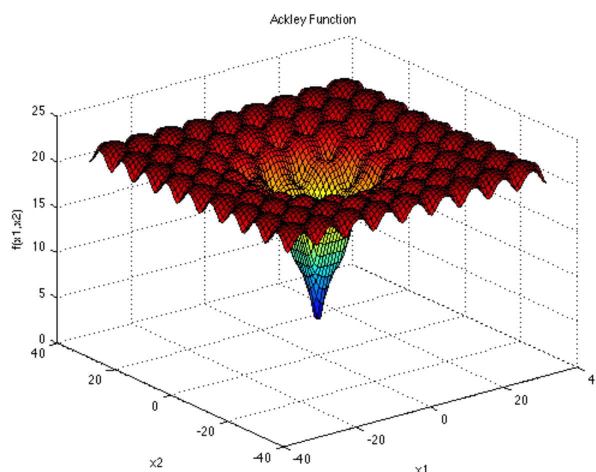


Figura 8.3 – Representación de la función Ackley para dos dimensiones[10].

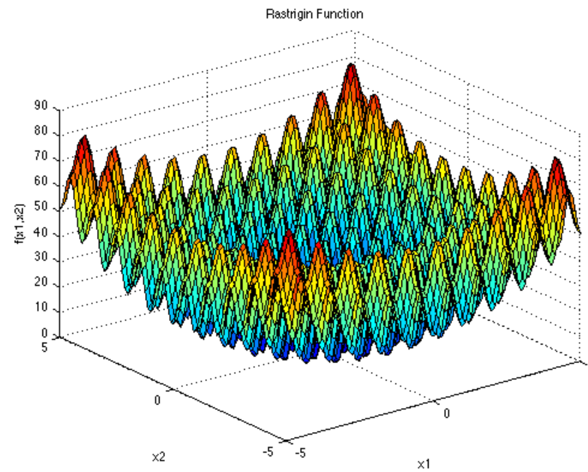


Figura 8.4 – Representación de la función Rastringin para dos dimensiones[11].

8.2.2. Caracterización MAP-Elites

Como se ha explicado con anterioridad, en este trabajo se van a implementar dos algoritmos de optimización: MAP-Elites y evolución diferencial. Respecto al primero de ellos, en esta sección se presentan los diferentes parámetros a configurar y que afectan al proceso de optimización y , por lo tanto, condicionan el resultado final. Por ello en esta prueba se va a aplicar la función sintética Rastrigin. En este caso, el espacio de búsqueda serán las distintas variables que definirán el problema. En cuanto al número de características se escoge un número determinado de estas variables , siendo su límite $n-1$, donde n son las dimensiones del problema.

A la hora de realizar el análisis se han escogido como parámetros relevantes: *población inicial*, *características*, *probabilidad de mutación* y *la amplitud (σ)*. Estos 4 parámetros serán barridos con el fin de observar como varía su respuesta y obtener una combinación ideal. Las características escogidas no deben llegar al número de variables del problema. En este caso, se ha fijado una función sintética de 7 dimensiones, por lo que el número de características llegará hasta 6. Como indicadores para medir la respuesta se escogen: la norma genotípica del mejor individuo (respecto al mínimo global), la calidad, el porcentaje de ocupación de las celdas y el ratio de sustitución (el número de iteraciones que le lleva realizar un reemplazamiento de las élites). El parámetro de la granularidad se debe escoger teniendo en cuenta que a altas dimensiones generaría una estructura de datos inviable computacionalmente. Por lo tanto, se fija un número de celdas totales a 2 dimensiones y se busca mantenerlas a lo largo del barrido. La condición de parada para cada prueba es de 30 segundos. Finalmente, para el operador de mutación se ha escogido uno de tipo gaussiano. Este tipo de operador aplica una perturbación al individuo mediante un valor obtenido de una distribución normal $N(\mu,\sigma)$, donde μ es la media, que será 0, y σ la desviación típica, que será un parámetro a variar.

Población inicial	10	25	99	671	9944
Granularidad	0.1	0.46	1.0089	1.6038	2.1845
Características	2	3	4	5	6
Mutación	0.1	0.2	0.4	0.8	
Sigma	0.05	0.1	0.2	0.4	

Tabla 8.1 – Parámetros utilizados durante las pruebas

Primero se realizan una serie de pruebas con los parámetros fijo y se observa en los resultados de la tabla 8.2 que no siempre se obtiene la misma calidad, por lo que presenta una componente aleatoria. Es necesario realizar un estudio de la variabilidad con el fin de obtener la menor variación posible en los resultados, mitigando la componente estocástica del algoritmo. Para ello se busca una media móvil que aporte unos resultados más estables. Como resultado se obtiene que es necesario realizar al menos 5 repeticiones para obtener unos valores más estables 8.4.

Norma	Sustituciones	% Ocupación	Calidad
3.2580	3.2162	92.4215	35,7062
2.9687	3.2206	92.4781	15,2473
3.0048	3.2117	92.4121	26,8501
3.2745	3.2222	92.0916	19,5957
4.3329	3.2091	92.2896	30,5949
3.7957	3.1894	92.0068	34,9852
3.9900	3.2300	91.9974	24,3482
4.0022	3.2247	92.0822	27,3872
3.2696	3.4477	90.5740	32,3596
4.2650	3.1920	92.1482	28,0432

Tabla 8.2 – Resultados obtenidos durante las pruebas.

	Norma	Sustituciones	Ocupación	Calidad
Media	3.6161	3.2364	92.0501	27.5118
Desviación	0.4913	0.0716	0.5201	6.1631

Tabla 8.3 – Media y desviación de los resultados obtenidos durante las pruebas.

Media móvil con 3	Media móvil con 4	Media móvil con 5
25.935	24,3498	25,5988
20.564	23,0720	25,4547
25.68	28,0065	27,2748
28.392	27,3810	27,3823
29.976	29,3289	29,9350
28.907	29,7701	29,4247
28.032	28,0346	27,9300
29.263	28,8255	
29.305		

Tabla 8.4 – Resultados de las medias móviles.

Una vez terminado el barrido, los datos son mostrados en una serie de gráficas. Los mapas de calor están estructurados de la siguiente manera. El punto origen se sitúa en la esquina superior izquierda debido a la librería utilizada para representar los datos. En cada gráfica se representan cuatro variables, dos para el eje exterior y otros dos para el interior. El color de la celda será el parámetros a estudiar. A continuación se muestra un esquema con el fin de aclarar la representación de las variables:

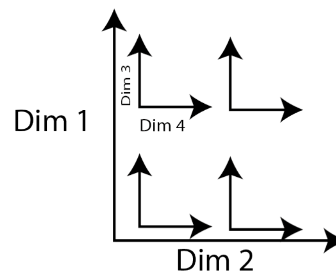


Figura 8.5 – Esquema de la representación en mapas de calor multidimensionales

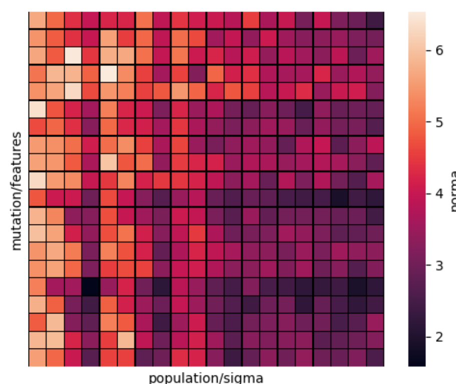


Figura 8.6 – Mapa de calor de la norma

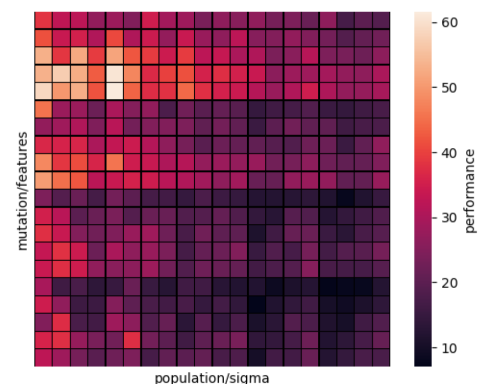


Figura 8.7 – Mapa de calor de la calidad

De los resultados obtenidos se pueden deducir una serie de comportamientos. En primer lugar, si se observa la gráfica correspondiente a la calidad, se comprueba como, conforme aumentan la probabilidad de mutación y la población, y a su vez disminuye las características, se obtiene un mejor resultado. En el caso particular de la población, existe un salto cualitativo importante para cada una de las variaciones del parámetro. Esto cesa al llegar a un número de individuos muy alto, momento en el que la influencia de este es menor. En cuanto al parámetro σ de la mutación, los mejores resultados se dan en los valores intermedios o más pequeños. Esto se debe a que para variaciones más pequeñas, una vez encontrado un buen nicho, este es explotado, mientras que con una variación muy grande se cambiaría de nicho con mayor facilidad sin llegar a profundizar en él lo suficiente. En la gráfica de la norma se observa un comportamiento similar, ya que ambos inidicadores están relacionados.

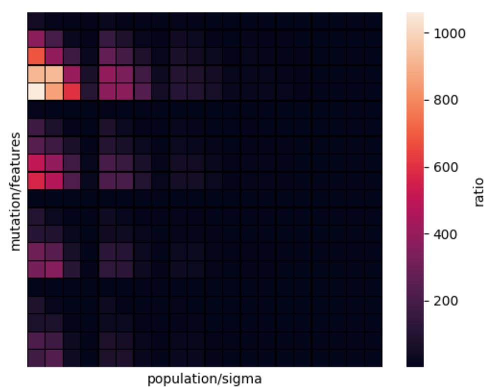


Figura 8.8 – Mapa de calor del ratio de sustitución

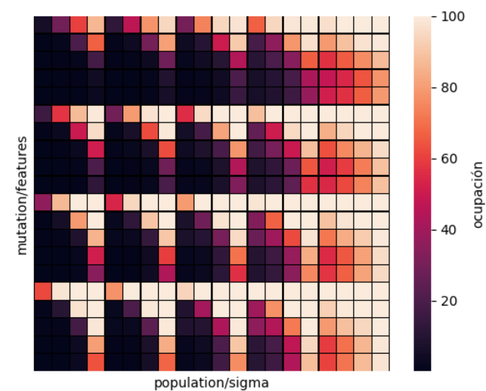


Figura 8.9 – Mapa de calor del porcentaje de ocupación

En cuanto a la gráfica de la ocupación se observa como, cuando la población inicial es pequeña, es necesario aumentar el σ para poder obtener una ocupación relevante que permita conocer el espacio de características. En cambio, conforme se aumenta la población es más sencillo ocupar el mapa y no haría falta tener un sigma tan alto. Además, se puede observar que, conforme aumenta el número de características, empeora la exploración. Por último, de la gráfica obtenida a partir del ratio de sustitución, se puede deducir que tiene un comportamiento similar a la ocupación, ya que conforme aumenta esta, las élites son sustituidas con más frecuencia.

A partir de los resultados se pretende deducir una configuración óptima del algoritmo. Tras estudiar la variación de los parámetros se deduce que es mejor tener una probabilidad de mutación alta (0.8), una sigma pequeña o media (0.05), una población inicial alta y un pequeño número de características. En la figura 8.10 se muestra un ejemplo del conjunto de soluciones obtenidas con esta configuración para un y problema de 4 dimensiones y 2 un espacio de características. En cuanto a las propiedades de este algoritmo, se puede deducir que tiene una alta capacidad de exploración, por lo que con la elección de estos parámetros se pretende encontrar equilibrio entre esta componente y la explotación de soluciones.

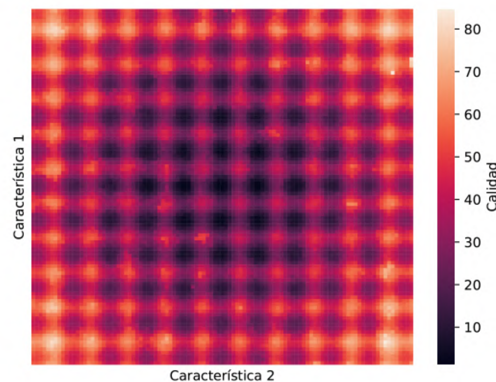


Figura 8.10 – Soluciones obtenidas en una prueba para 4 dimensiones y 2 características.

8.2.3. Caracterización Evolución Diferencial

De la misma forma que se ha realizado para el MAP-Elites, se implementa el algoritmo evolutivo diferencial para resolver el problema de optimización de la función Rastrigin. A la hora de analizar el comportamiento de este algoritmo, los parámetros que influyen sobre su comportamiento son: *F* (Ponderación de la diferencia), *CR* (Probabilidad de cruce) y *NP* (Población). En la literatura se puede encontrar una configuración inicial recomendada que es: *F = 0.8*, *CR = 0.9* y *NP* 10 veces la dimensión del problema [7]. Tanto el funcionamiento como el rendimiento de este algoritmo se ven afectados por la elección de estos parámetros, por lo que se hace un barrido de estos. Los valores seleccionados durante el proceso se pueden observar en la tabla 8.5. En este caso al realizar algunas pruebas con la configuración recomendada no se observa una variación importanteen los resultados, por lo tanto no se realiza una ponderación entre varios pruebas.

NP	10	19	43	125	483
F	0.2	0,4	0,6	0,8	1
CR	0,1	0,3	0,5	0,7	0,9

Tabla 8.5 – Valor de los parámetros durante el barrido.

Una vez realizado el barrido se grafican los resultados. En cada una de las gráficas se mostrará el parámetro a variar en el proceso, mostrando como varía las métricas correspondientes. En este caso se ha escogido la calidad y el tiempo de cálculo para observar como se comporta el algoritmo. La calidad viene dada por la función que define el problema, mientras que para medir el tiempo se establece un límite de tiempo máximo para la optimización de 30 ms, si lo hace antes se corta la ejecución del individuo y se pasa al siguiente.

Si se observan las gráficas correspondientes a la variación de la calidad y tiempo de convergencia según las dimensiones, se comprueba como conforme se aumenta la dimensionalidad del problema, empeora la calidad 8.11. De la misma forma, se puede observar como claramente tarda más en converger hacia la solución 8.12.

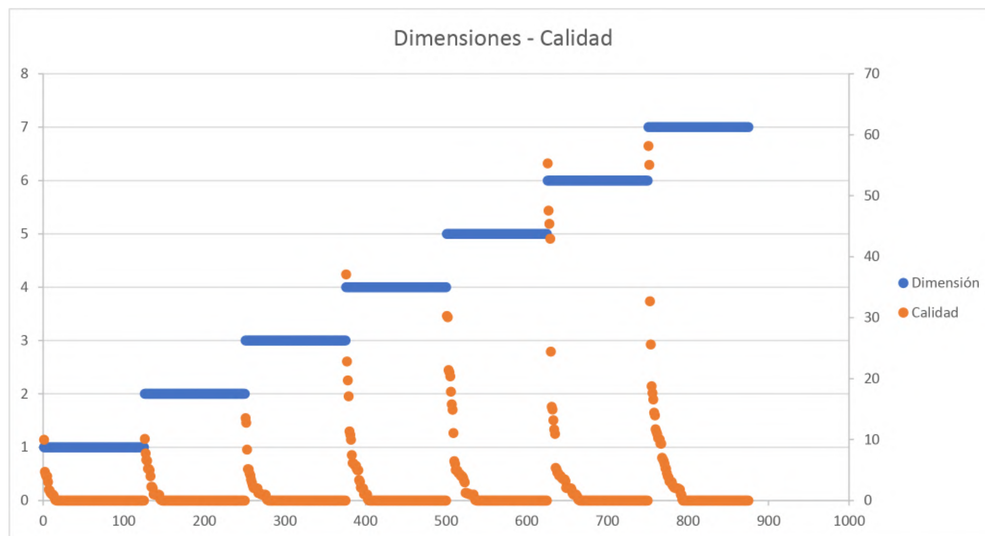


Figura 8.11 – Variación de la calidad según el número de dimensiones del problema.

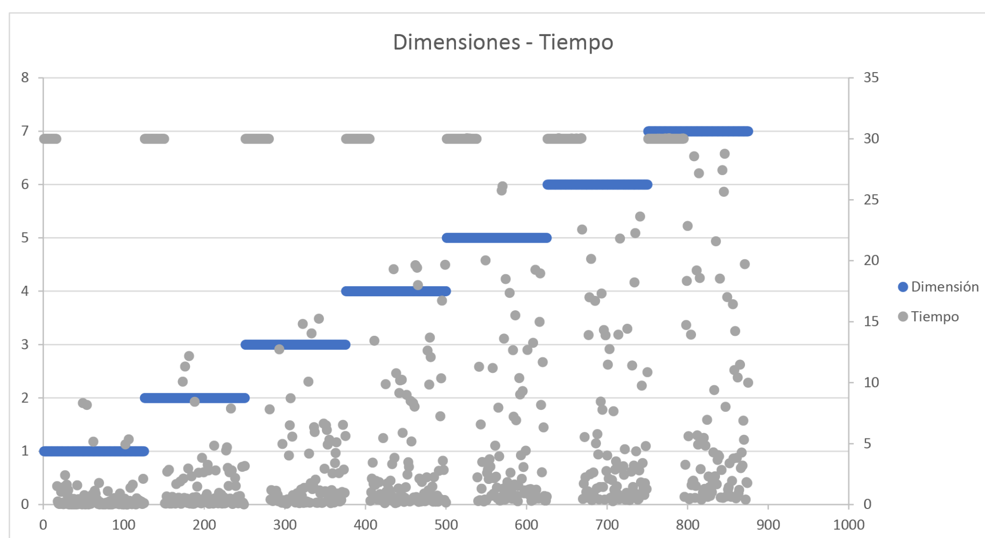


Figura 8.12 – Variación del tiempo según número de dimensiones del problema.

En cuanto a la población cuando ésta es muy baja, el algoritmo no es capaz de alcanzar la solución con frecuencia, conforme se aumenta el parámetro se observa como van mejorando los resultados obtenidos 8.13. Una vez pasado un tamaño determinado ya no se obtiene beneficio alguno, incluso mostrando peores resultados en el caso más alto. En cuanto al tiempo, en la gráfica se comprueba como en valores bajo tarda más en converger 8.14. En el caso de aumentar se observa una mejoría en este aspecto. Una vez alcanzado un tamaño determinado se observa como empieza a tardar más en converger y el efecto es contraproducente, no le lleva tanto tiempo como en el primer caso, pero se observa un aumento importante de valores en el rango medio del tiempo.

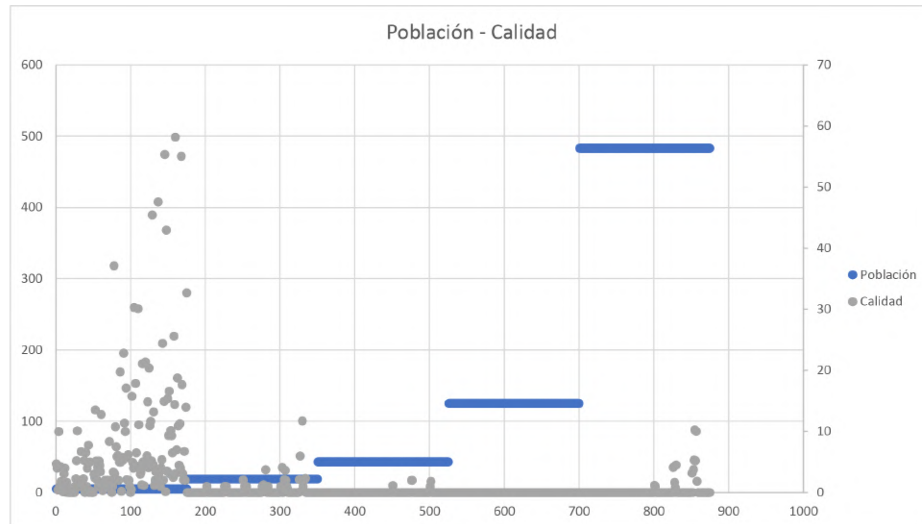


Figura 8.13 – Variación de la calidad según el tamaño de población.

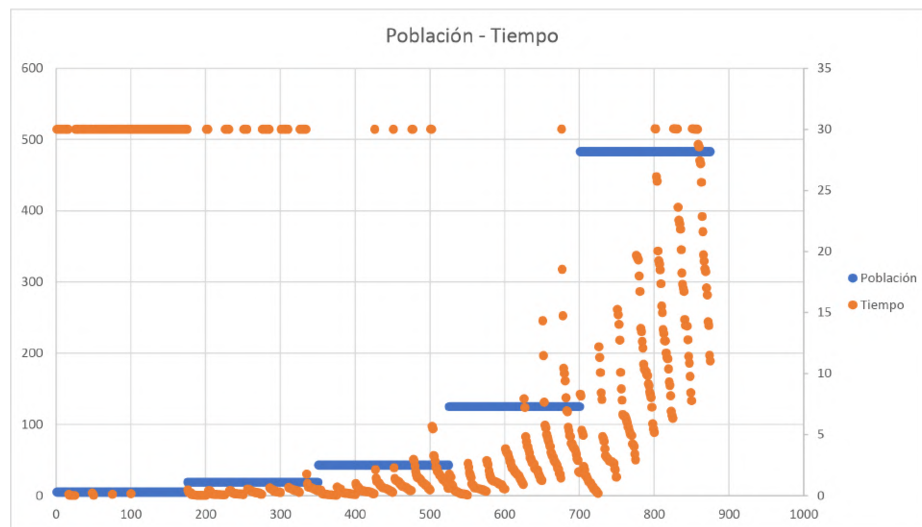


Figura 8.14 – Variación del tiempo según el tamaño de población.

A la hora de analizar los parámetros CR y F, se observa un comportamiento similar. En ambos, el tercer caso del barrido, 0.8 para F y 0.7 para CR, supone la mejor configuración para obtener los mejores valores de calidad, tal como se muestra en 8.15 y 8.17. Un aumento o reducción excesiva de estos supone empeorar los resultados. En cuanto al tiempo se observa en 8.16 y 8.18 una variación en el número de casos que alcanzan la solución antes del límite pero no suponen un cambio importante.

Tras realizar el análisis, se concluye que, efectivamente, la configuración recomendada como punto de partida por Storn and Price [7], se aproxima a lo visto en las pruebas, aunque es mejor un pequeño ajuste antes de aplicarlo al problema al que se aplique finalmente. Esto se debe a que ya no solo se debe conseguir que converga a la solución, si no optimizar los recursos computacionales disponibles. Además, hay que tener en cuenta para su aplicación que la calidad obtenida también depende de la función escogida para medirla. De esta forma, se debe tratar de incluir la mayor cantidad de información posible, para así facilitar la convergencia.

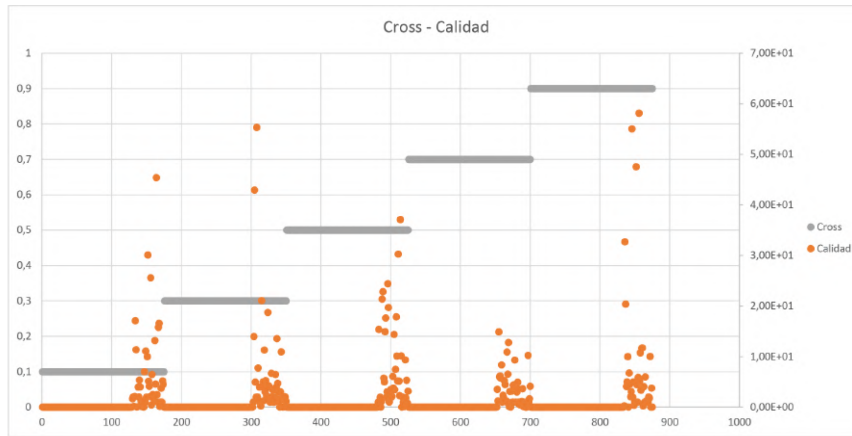


Figura 8.15 – Variación de la calidad según CR.

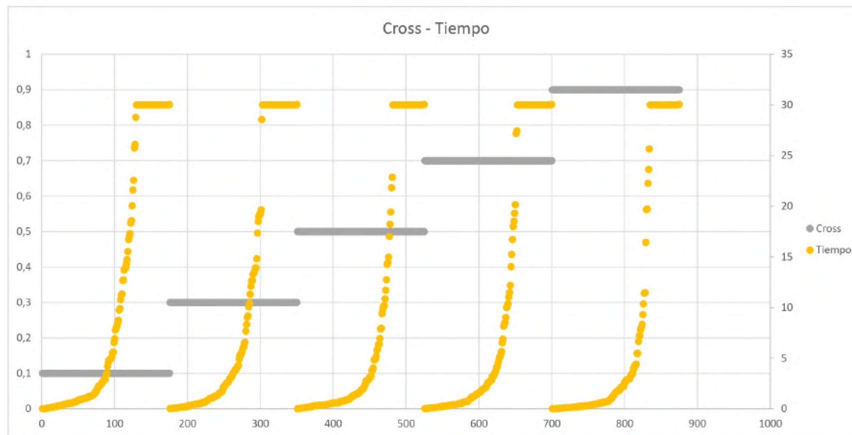


Figura 8.16 – Variación del tiempo según CR.

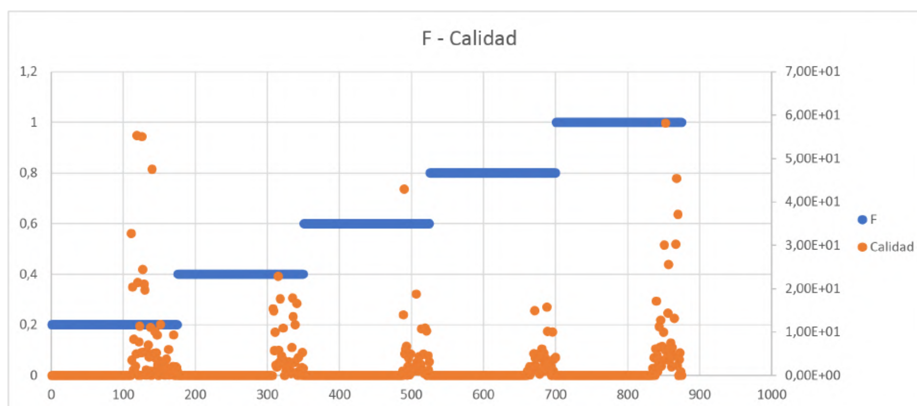


Figura 8.17 – Variación del tiempo según F.

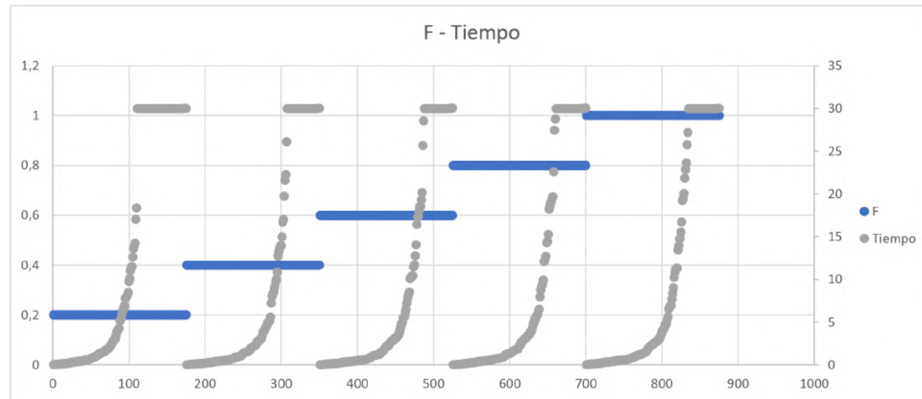


Figura 8.18 – Variación del tiempo según F.

8.3. Odometría visual

La odometría es el uso de la información extraída de los sensores de un robot para estimar su posición relativa a lo largo del tiempo. En este caso, es la estimación a partir del análisis de las imágenes tomadas por la cámara del smartphone. La precisión de este proceso depende de diversos factores como los datos obtenidos, la calibración de la instrumentación o el procesado que se realiza sobre la información. La idea fundamental es ir integrando la información que percibe el robot, esto se realiza de forma incremental a lo largo del tiempo, por lo que es inevitable que se vayan acumulando los errores producidos durante el movimiento. El hecho de utilizar odometría visual conlleva una serie de ventajas ya que, en el caso de utilizar los encoders, se haría bajo el supuesto de que las ruedas no patinan, además se puede aplicar a sistemas que no utilicen estos dispositivos.

8.3.1. Sistema de visión

Para poder actuar sobre el entorno que lo rodea, el robot debe ser capaz de extraer de él la información relevante. En el presente trabajo, se ha desarrollado un sistema de visión artificial, ya que la cámara es el sensor de que se puede extraer la mayor cantidad de información posible. Dentro de éste, será clave tanto la utilización de la cámara como el procesamiento de la imagen. Esto le permite desarrollar una serie de comportamientos como reconocer objetos o formas para su uso durante la realización de la tarea.

Para la obtención de la imagen se utilizará la cámara del smartphone, que se en encuentra su parte superior. Debido a que en el entorno del experimento los diferentes elementos que lo que componen se encuentran a baja altura, se ha decidido dar la vuelta al smartphone, por lo que en realidad estará situada casi al nivel de la plataforma robótica. Una vez obtenida la imagen, esta es transmitida al PC a través de la aplicación de streaming de vídeo de Robobo. Aquí es donde es procesada para extraer de ella la información necesaria. Los algoritmos para el tratamiento de la imagen se implementarán con la librería de visión artifical de OpenCV.

Figura 8.19 – Información a extraer del sistema de visión.

Para que el robot pueda guiarse en el entorno se utiliza un cinta de color azul de 4.8 cm. Esto servirá para conocer la distancia del agente a las paredes de la zona. Además, se va a incorporar una serie de objetos de distinto color de la franja, en este caso unos cilindros verdes, con el fin de que se interactúe con ellos de alguna forma durante el experimento. La información que se puede extraer de estos elementos se encontrará en dos franjas de píxeles: una vertical y otra horizontal. La primera, se utilizará para detectar el ancho de la cinta y la segunda para detectar los obstáculos. Finalmente, se mapeará la información en una rendija de cuatro celdas. Si en una se detecta un obstáculo se guarda un valor entre 0 y -1, según los píxeles que ocupe en la celda, y en el caso de la pared entre 0 y 1, según a la distancia a la que se encuentre.

Resumiendo, el proceso de sensorización constaría de los siguientes pasos:

1. **Obtención de la imagen**: Envío de la imagen desde el smartphone al PC.
2. **Obtención de las franjas**: Segmentación de la imagen según la cantidad de información que sea necesaria.
3. **Detección del color**: Aplicación de máscaras para aislar los colores que se vayan a utilizar para después extraer la información.
4. **Procesado**: Refinamiento del resultado obtenido en los pasos anteriores través de los filtros que sean necesarios.
5. **Extracción de características**: Operaciones necesarias sobre la imagen para obtener la información necesaria.

8.3.2. Detección de color

En este trabajo se va a utilizar la detección de color para para poder reconocer los diferentes elementos presentes en el entorno. A la hora de realizar este proceso hay que tener en cuenta una serie de aspectos del entorno donde se va a realizar el experimento, así como el modelo y algoritmos que se van a utilizar durante este proceso.

Para ello se utiliza la librería OpenCV, que tiene una serie de funciones de análisis y procesado de imagen. En este caso, se utilizará el modelo HSV para definir una máscara que permita aislar el color objetivo.

8.3.2.1. Modelo HSV

El modelo HSV (Hue, Saturation,Value) es una representación alternativa del modelo de colores RGB. A diferencia de este, cuyas coordenadas son euclidianas, el HSV sigue una representación similar a las coordenadas cilíndricas. Esta representación es más cercana a la forma en la que los humanos perciben los colores. Los tonos de cada Hue se encuentran en rebanadas radiales, alrededor de un eje central de colores neutros que variarán desde el negro, en la parte inferior, hasta el blanco en la parte superior. De esta forma, la principal ventaja de utilizar este modelo es que se puede obtener una mejor información sobre los colores a detectar, a diferencia del RGB, donde las componentes de sus colores se ven afectadas por la luz incidente. En la tabla 8.6 se muestran diferentes valores de los límites medios de algunos colores.

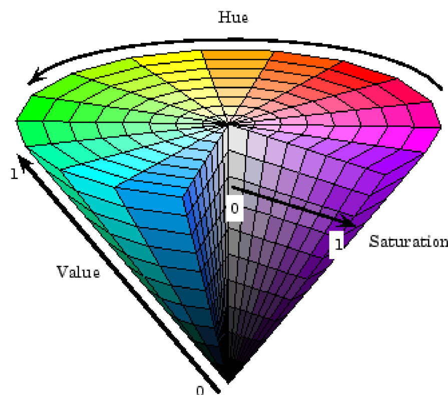


Figura 8.20 – Ilustración del espacio de color HSV[12].

Color	Límite Superior	Límite Inferior
Rojo	[20 255 255]	[0 100 100]
Azul	[110 255 255]	[90 100 100]
Verde	[95 255 255]	[75 100 100]
Amarillo	[40 255 255]	[20 100 100]

Tabla 8.6 – Límites HSV tipicos para diferentes colores.

- **Matiz (Hue):** Se representa como un valor en grados que irá entre 0 a 360. Cada valor corresponderá a un color puro. Por ejemplo, todos los rojos se encontrarán en un intervalo de matiz determinado.
- **Saturación (Saturation):** Es la cantidad de blanco del color. En el caso de ser 0 o próximo a este se observaría un color blanco mientras que si fuese máximo se tendría el tono de un color primario.

- **Valor (Value):** También llamado brillo, define la intensidad de color. Por lo tanto, si fuese cero sería negro, mientras que si se aumenta se observaría como irían apareciendo los diferentes colores.

8.3.2.2. Análisis y procesamiento de imagen

Una vez establecido el modelo a utilizar, es necesario estudiar los parámetros del entorno y los objetos a detectar para obtener una mejor respuesta a la hora de procesar la imagen. El objetivo final de este proceso es identificar de forma correcta el grosor de la banda y el ancho del cilindro. Para ello, es necesario encontrar un rango óptimo que permita la correcta detección del objeto e interferir lo menos posible en la forma original de éste.

En primer lugar, se analiza una imagen que tenga presente la cinta de color azul. De los histogramas obtenidos y teniendo en cuenta el modelo HSV, se puede extraer que están presentes tres fuentes de color principales: azul, blanco y marrón. Se observa en la figura 8.21 que los dos rangos con mayor densidad son los de 0-25 y 110-125. El primero pertenece al rojo, componente presente en el color marrón del suelo, y el segundo al azul de la banda. El otro tono predominante es el blanco, esto se puede comprobar en la gran cantidad de píxeles en el intervalo más bajo de la saturación 8.22.

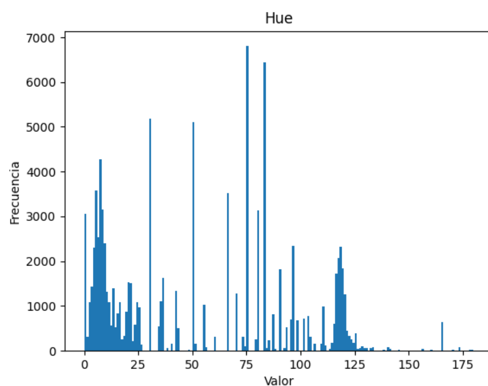


Figura 8.21 – Histograma del parámetro Hue.

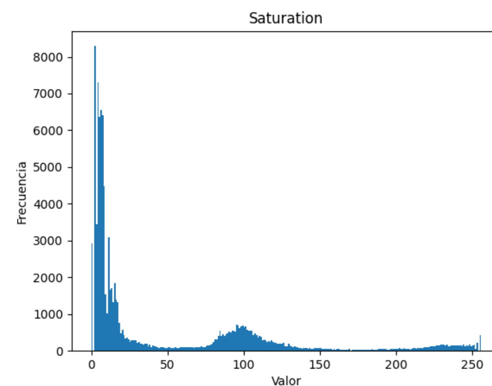


Figura 8.22 – Histograma del parámetro Sat.

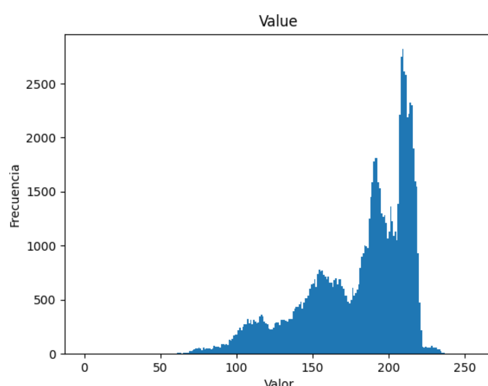


Figura 8.23 – Histograma del parámetro Value.

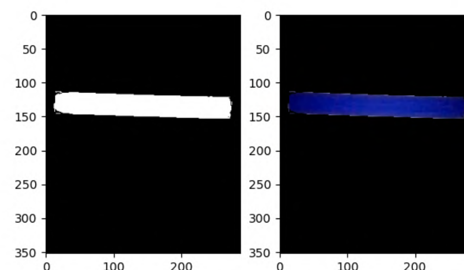


Figura 8.24 – Máscara e imagen resultante.

Una vez conocido los rangos, se establecen los limites inferior y superior que se le deben pasar a la función de OpenCV con la que se hace la máscara, el comando *inRange*. Tras un pequeño ajuste manual, los límites serán $(100, 60, 90)$ para el inferior y $(140, 255, 255)$ para el superior. Después se aplica a la imagen original con la función *Bitwise* para obtener el color aislado. En la figura 8.24 se muestra tanto el resultado de la obtención de la máscara como el resultado final.

Para la detección de un objeto se procede de forma análoga utilizando otro color como objetivo. El rango típico del verde estaría entre los valores de Hue de 75-100. De la misma forma que con la banda se observa que, a parte del color objetivo, los dos colores predominantes son el marrón, próximo al rojo, y el blanco o gris claro. Este último se puede observar en la gráfica de saturación 8.26 donde hay una cantidad importante de píxeles en el rango 0-25 y en el Value con la falta de píxeles en el rango más bajo 8.27. Por lo tanto, se limita tanto el rango de Hue como de Saturation. Finalmente, se los límites $(75, 70, 50)$ para el inferior y $(110, 255, 255)$ para el superior. El resultado se puede comprobar en la imagen 8.28.

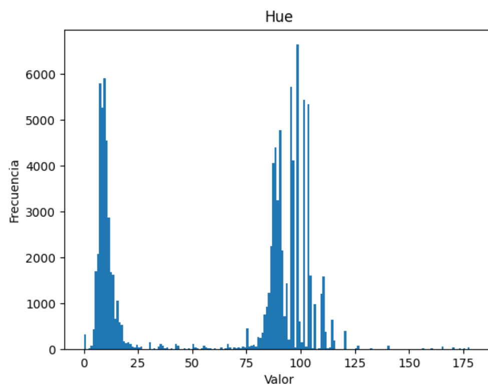


Figura 8.25 – Histograma del parámetro Hue.

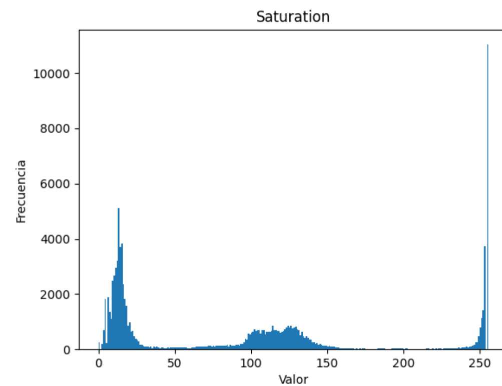


Figura 8.26 – Histograma del parámetro Sat.

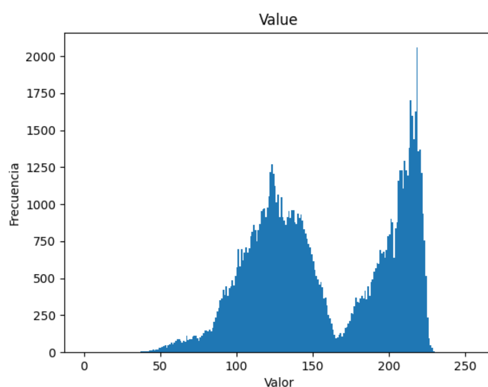


Figura 8.27 – Histograma del parámetro Value.

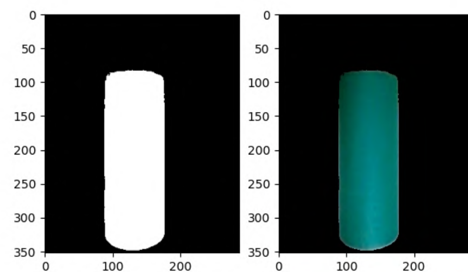


Figura 8.28 – Máscara e imagen resultante.

8.3.3. Estimación de la distancia

Tras aislar la cinta del entorno se procede a establecer una relación entre el grosor de esta y la distancia que la separa del robot. Para ello se prepara un entorno en el que se sacará mediante el comando *imwrite()* de OpenCV varias imágenes a distintas distancias y se mide el grosor de la banda. Aplicando óptica geométrica y semejanza de triángulos se puede conocer la relación entre el ancho de la banda en píxeles y la distancia real.

Figura 8.29 – Posicionamiento del robot para obtener las imágenes.

A continuación, se describe el problema de óptica al que se ha reducido el proceso y esquema para su visualización 8.30. Se tienen dos triángulos semejantes, pero de distinto tamaño. Siendo h la altura real, h' la altura imagen, d la distancia real y d' la distancia imagen, el objetivo es encontrar la razón de escala entre ambos triángulos. Como h' y d son conocidos se calcula k para cada una de las imágenes obtenidas. Por semejanza se obtiene que el producto entre la altura real y a distancia imagen es una constante, que a su vez también es igual a la altura imagen por la distancia real 8.3. Por lo tanto se deduce la distancia real ser á una constante entre la altura imagen 8.4.

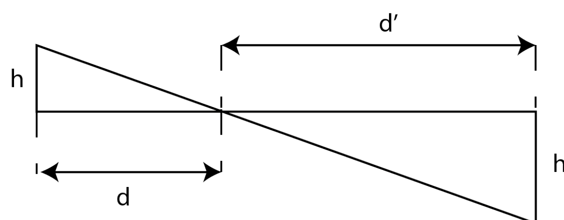


Figura 8.30 – Esquema geométrico de la estimación de la distancia.

$$\frac{k_m}{h'(pixeles) = d(cm)} \tag{8.3}$$

$$\frac{h}{d} = \frac{h'}{d'} \rightarrow h \times d' = h' \times d = k \tag{8.4}$$

Este cálculo se aplica a cada una de las imagenes, obteniendo la constante correspondiente a partir de la distancia a la que se ha sacado la foto y el grosor de la cinta en píxeles. Una vez calculadas todas , se obtiene la constante media y se prueba la estimación con esta *k* para cada uno de los puntos de medida. Debido a la necesidad obtener más información de las distancias más cercanas a la banda, se ha decidido tomar más datos de este rango. En la estimación obtenida 8.7 se observa que conforme aumenta la distancia también lo hace el error. Se compara los valores de la distancia real con la estimación en la gráfica 8.31 y se observa como se puede aproximar mediante un ajuste polinómico. En este caso, se ha decidido probar hasta un polinomio de grado tres. Una vez obtenido cada uno de los nuevos ajustes se vuelve a comparar con las medidas reales.

	Píxeles	Distancia (cm)	k	Estimación	Error
	124	10	1240	10.9953	-0.9953
	87	15	1305	15.6715	-0.6715
	60	23	1380	22.7238	0.2761
	22	63	1386	61.9740	1.0259
	13	103	1339	104.8791	-1.8791
	10	143	1430	136.3428	6.6571
	8	183	1464	170.4285	12.5714
Media			1363.429		2.4263
Desviación					4.8880

Tabla 8.7 – Estimación de la distancia con la k_m

Se puede observar en la tabla 8.8 que el error es menor en el ajuste de polinómico de grado 2 sobretodo a largas distancias. Todo lo contrario pasa en el rango intermedio, donde el error aumenta. Este rango de la estimación mejora con la introducción de un grado más en el ajuste polinómico aunque no se solventa del todo.

	Grado 2	Error	Grado 3	Error
	11.0937	-1.0937	10.3310	-0.3310
	15.4201	-0.4201	15.2124	-0.2124
	22.0263	0.9737	22.4842	0.5157
	60.5801	2.4198	61.8322	1.1677
	106.189	-3.1889	105.5503	-2.5503
	141.9356	1.0644	140.5196	2.4803
	182.8574	0.1425	183.4054	-0.4054
Media		-0.0146		0.0949
Desviación		1.6689		1.4413

Tabla 8.8 – Estimación de la distancia tras los ajustes polinómicos.

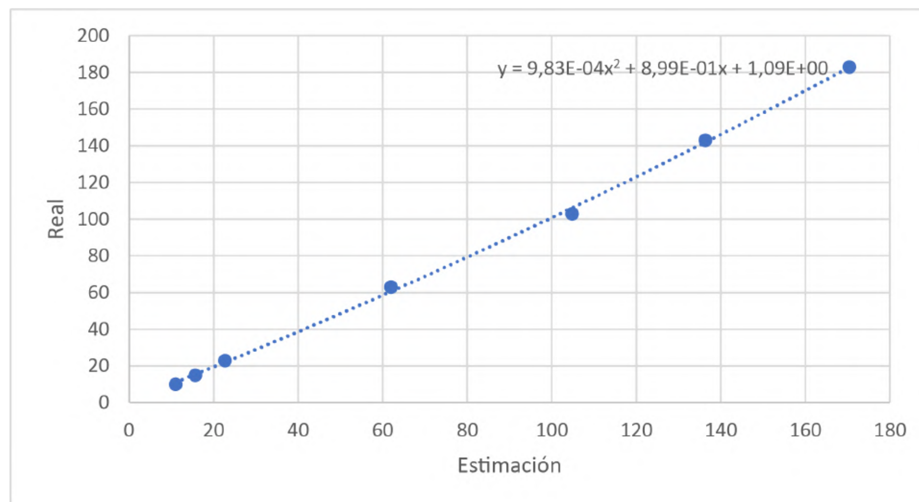


Figura 8.31 – Ajuste polinómico de segundo grado.

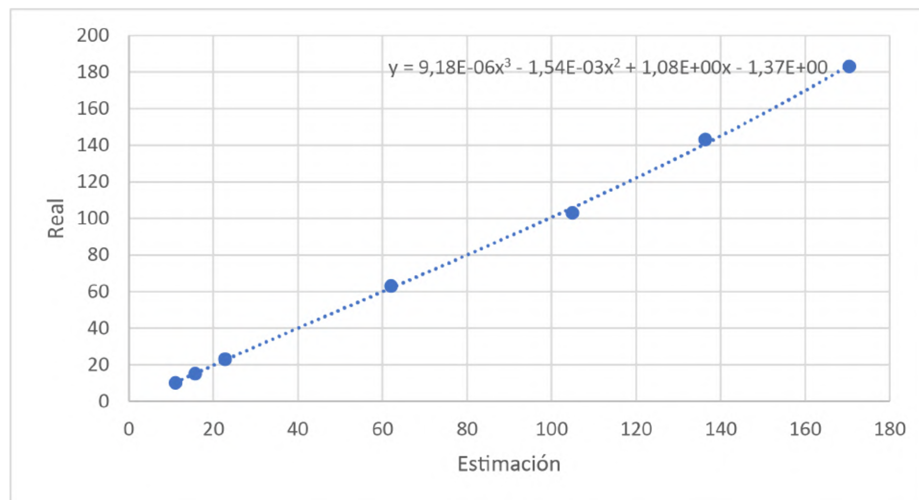


Figura 8.32 – Ajuste polinómico de tercer grado.

8.3.4. Análisis de errores en movimiento

Para conocer los posibles errores presentes durante la navegación se plantea una secuencia de movimientos que deberá realizar el robot. Éste empezará desde una distancia determinada y realizará una serie de paradas programadas previamente hasta llegar a la pared 8.33. Una vez terminado, se observan los valores de la distancia estimada, la real y el valor de los encoders de una de las ruedas. Para analizar el efecto que tiene la sensorización visual se varía la cantidad de frames que se procesan por segundo y también se varía la velocidad a la que realiza la secuencia. De esta forma, además se puede encontrar una velocidad límite y así establecer un rango de operación de la misma. En cuanto a la medida de las distancias de parada se utilizar un metro y, para evitar errores de tipo óptico se elige tomar la medida en el eje de la rueda. Para facilitar este proceso de medida se establece un tiempo de parada en cada punto de 2 segundos.

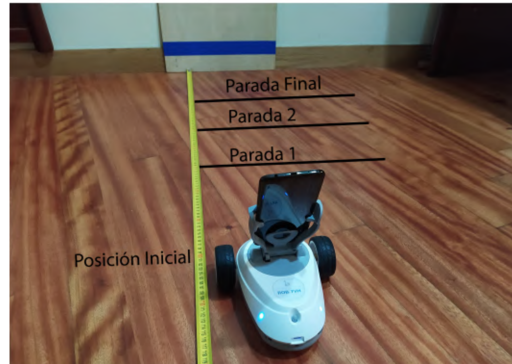


Figura 8.33 – Secuencia de paradas.

En los resultados obtenidos mostrados en las tablas 8.9 y 8.10 se pueden observar una serie de errores. En primer lugar, el robot no es capaz de parar con exactitud. Esto se puede deber a la existencia de un tiempo de reacción ante el cambio de distancia. Además, se puede comprobar como el error no es el mismo en cada parada lo que supone un error en la discretización. También se puede extraer de los datos una velocidad límite en la que directamente el robot no es capaz la secuencia completa, que sería a partir del 70 %. En este caso se muestran dos casos para una tasa de frames distintos, donde se observa que la variación de este no supone una mejora importante para evitar dichos errores. Para poder mitigar el efecto de estos errores se propone estudiar el efecto de la discretización en la medida en movimiento y en reposo y los tiempos involucrados durante el procesado de la información y el envío de los comandos.

Velocidad/ Parada	15	20	30	40	50	70	90
70	69	69	62	56	57	50	45
50	48	47	45	41	37		
30	26	26	23	18	18	14	10

Tabla 8.9 – Distancia de parada para un delay de 20 ms

Velocidad/ Parada	15	20	30	40	50	70	90
70	66	66	63	59	58	49	48
50	47	44	44	40	38		
30	25	25	22	19	18	10	5

Tabla 8.10 – Distancia de parada para un delay de 40 ms

8.3.4.1. Análisis de la discretización

Como se ha descrito anteriormente, el proceso de estimación de la posición del robot produce una serie de errores durante la navegación. Esto se puede producir por una serie de razones. En primer lugar, los cálculos se han realizado a través de medidas a unas distancias concretas y en estático. Además, la capacidad de discretización va a estar limitada por la resolución de la propia cámara de la aplicación. Por lo tanto, va a ser necesario estudiar como varía esta discretización a lo largo de los intervalos de medida. Después, se debe observar los errores existentes en una situación de movimiento. Esto se hace a través de dos pruebas. En la primera, se programa al robot para que avance de forma perpendicular a la banda a una velocidad constante y se toman datos de la estimación en tiempo real. La segunda, consiste en ir moviendo al robot a mano en los intervalos de medida para así conocer donde se producen los cambios en la estimación.

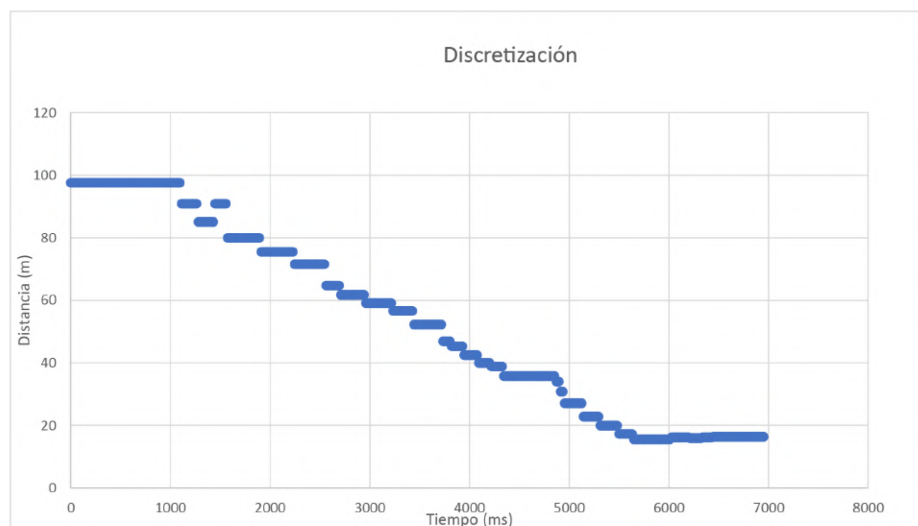


Figura 8.34 – Discretización en una secuencia de movimiento.

Como se puede observar en la gráfica 8.34 perteneciente a un movimiento continuo hasta una distancia de la pared, existe una discretización importante, lo que resultaría en un error considerable a la hora de reaccionar a cambios en la distancia. Esto se debe a que este método está limitado por la resolución de la cámara, produciéndose una zona difusa alrededor de la banda que afecta a cómo percibe la distancia 8.35. Por lo tanto, sería conveniente tener información con una resolución sub-pixel, pero sin suponer un elevado coste computacional. En este caso, se hará un filtro de precisión sub-pixel que se aprovechará de la zona difusa de las inmediaciones de los bordes, para estimar con mayor precisión su posición relativa.

Figura 8.35 – Detección de la cinta con la zona difusa a su alrededor.

El parámetro que separa esta región difusa será la saturación, ya que es donde el color pasa del tono puro a un tono grisáceo, por lo que se deberá modificar el rango de la máscara calculada previamente para detectarlo. El objetivo es dar un peso según el valor de saturación, siendo los píxeles de la cinta quienes obtendrán un valor máximo y los de los bordes un valor proporcional a la cantidad de saturación 8.38. Por lo que este aumentará conforme se vaya acercando a la zona de con mayor intensidad de la cinta. En las gráficas 8.36 y 8.37 se muestra el valor de la saturación antes y después de aplicar el filtro. Se puede observar como existe una zona donde existe una valor pico de y como conforme se va alejando de la cinta este valor desciendo.

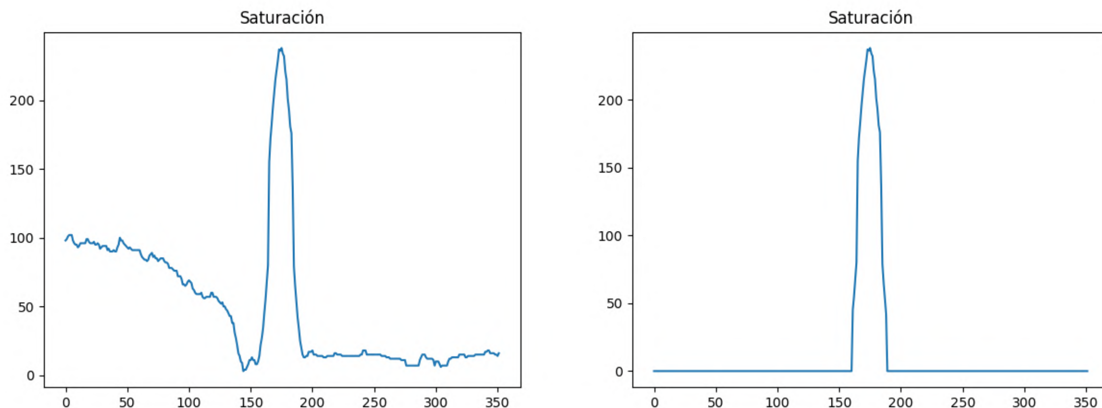


Figura 8.36 – Detección de la cinta con la zona difusa a su alrededor.

Figura 8.37 – Detección de la cinta con la zona difusa a su alrededor.

Este comportamiento se asemeja a una función sigmoide, que es un caso particular de una función logística, por lo que se aplica a modo de filtro. Es necesario normalizar los píxeles para que se adapten a la zona sensible y a su vez la pendiente de la función para obtener la respuesta deseada. Una vez obtenida la salida, se tendrán píxeles con valores de 0 a 1, que sumados proporcionan el ancho de la banda en píxeles con la precisión deseada.

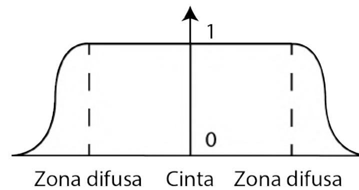


Figura 8.38 – Comportamiento de los valores de pixel deseado.

Se realiza una prueba idéntica a la primera, pero con esta discretización. Como se puede observar en la gráfica 8.39, este filtro proporciona una mejor respuesta a los cambios de distancia siendo más constantes durante todo el trayecto. Esto también se puede observar en las segunda prueba, donde los resultados muestran como conforme se va moviendo el robot a mano, el sistema de visión es capaz de mostrar una mejor discretización de la medida.

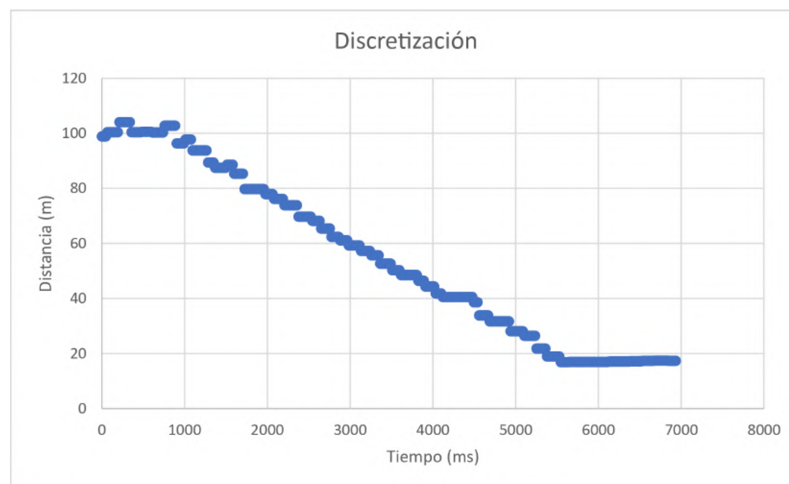


Figura 8.39 – Discretización nueva en una secuencia de movimiento.

Distancia Estimada (cm)	Precisión sub-pixel (cm)	Distancia Real (cm)
67.9852	71.1093	70
67.9852	69.0248	69
64.7496	68.0727	68
64.7496	66.6302	67
61.8128	65.0160	66
61.8128	64.0128	65
59.1339	64.0263	64
59.1339	62.4361	63
56.6795	61.7158	62
56.6795	60.6797	61
54.4217	59.7614	60

Tabla 8.11 – Comparación de las discretizaciones en la prueba manual.

8.3.4.2. Análisis temporal

Una vez que el sistema es capaz de producir una respuesta mucho más homogénea durante la navegación, se procede a analizar que tiempos influyen en la respuesta del robot. Para ello se utiliza la misma prueba utilizada al principio de este proceso con una variante, antes de frenar en cada parada el robot debe encender un LED. De esta forma, se puede estimar con mayor precisión los posibles retardos producidos. Para observar esto, se graba un vídeo y se analizan los frames. Además, se realizan unas marcas donde el robot debe detectar la parada según la discretización utilizada. A su vez, una vez finalizado el proceso se exportan los datos obtenidos durante el mismo para obtener más información. La información a analizar será: la distancia estimada, el estado del robot (movimiento o parada) y los tiempos de procesado y los comandos de movimiento y frenado.

Teorica	Discretización	LED
70	68	66
50	49	47
30	29	26

Tabla 8.12 – Distancias obtenidas a v = 10

Teorica	Discretizacion	LED
70	68	61
50	49	43
30	29	20

Tabla 8.13 – Distancias obtenidas a v = 30

Teorica	Discretizacion	LED
70	68	59
50	49	40
30	29	19

Tabla 8.14 – Distancias obtenidas a v = 40

De los tiempos del propio programa obtenidos a lo largo del proceso se obtiene que el existen dos tiempos que afectan al proceso interno: el tiempo de respuesta, que es el tiempo que tarda en encenderse el LED desde cruzar la marca, y tiempo de procesamiento. Ambos son constante en todo el proceso y los demás comandos se ejecutan de forma casi instantánea. El tiempo de obtención es siempre el mismo, pero el tiempo de procesamiento varía en función de la cantidad de información a procesar, por ejemplo aumentando la cantidad de colores a detectar, por lo que es factor a tener en cuenta. Además, de estos resultados se puede obtener que, efectivamente existe un retraso inherente a la aplicación de la cámara, ya que si se compara el momento exacto de la orden de parada y en el momento en el que se enciende el LED en el vídeo, no coinciden las discretizaciones. Para calcular este tiempo se escogen el frame donde el robot alcanza la marca y el frame donde enciende el LED, la diferencia de tiempos será el retardo de la cámara. Este proceso solamente se repite para distintas velocidades, ya que en unas primeras pruebas variando los frames no no se observó variaciones en la respuesta.

Una vez identificado los tiempos que influyen en el movimiento del robot, se estima el error cometido utilizando toda esta información. Se propone dos ecuaciones que, a partir de los tiempos, la velocidad de navegación y la aceleración de frenado, estima el tiempo y la distancia correspondiente al error en la parada. Los tiempo son conocidos y obteniendo el tiempo del frame donde alcanza la marca y donde queda estacionado, se obtiene el cociente entre la velocidad y la aceleración 8.5. A partir de los datos obtenidos en las pruebas por ejemplo para una parada a 70 cm de la cinta 8.15, se pueden obtener los siguientes resultados 8.16 para un tiempo de procesado de 0.005 s.

$$tiempo_{error} = t_{respuesta} + t_{procesado} + V/a \tag{8.5}$$

$$distancia_{error} = (t_{respuesta} + t_{procesado}) \times V + \frac{V}{2} \times \frac{V}{a} \tag{8.6}$$

Velocidad	Tiempo Marca (s)	Tiempo LED (s)	Tiempo Parada (s)
10	6.698	7.148	7.311
30	2.930	3.350	3.572
40	1.009	1.404	1.654

Tabla 8.15 – Tiempos obtenidos de los diferentes eventos de una para a 70 cm de la cinta.

Velocidad	Tiempo Respuesta (s)	Tiempo Error (s)	V/a
10	0.450	0.613	0.163
30	0.420	0.642	0.217
40	0.395	0.645	0.250

Tabla 8.16 – Términos de la ecuación obtenidos a partir de la para a 70 cm.

8.4. Modelado

En este apartado se modelará los actuadores de la unidad Robobo utilizados en el experimento. De esta forma, se podrá implementar en un simulador, haciendo que su comportamiento dentro de este sea lo más parecido al Robobo real, y permitiendo que se aplique la misma lógica a la hora de programarlo. Esto dos premisas son clave para después realizar una correcta transferencia de los comportamientos entre simulador y entorno real.

8.4.1. Modelado velocidad lineal

En primer lugar, se describe el proceso utilizado para obtener un modelo de la velocidad lineal del robot. Dado que el comando encargado del movimiento de las ruedas necesita un valor de velocidad para poder funcionar, el objetivo de éste apartado será obtener un expresión matemática que relacione este porcentaje con la velocidad real de navegación.

El procedimiento utilizado para obtener este modelo será programar el robot para que avance de forma lineal a una velocidad constante durante un tiempo determinado. Una vez recorrido este espacio se toma la medida de la distancia recorrida. El comando utilizado será *moveWheelByTime()*, al que se le pasa la velocidad de cada rueda y el tiempo que debe durar el movimiento. Esto se repite para distintas velocidades. El rango escogido es de 10 a 60, en pasos de 10, ya que este es el valor de la velocidad límite obtenida en el capítulo anterior. Para realizar la medición lo más precisa posible se coge la distancia en el eje de la rueda, de esta forma evitando algún error de tipo óptico. Este proceso se pueda observar en la figura 8.40. En cuanto a los tiempos se ha escogido valores más bajos y próximos para obtener más información de este intervalo.

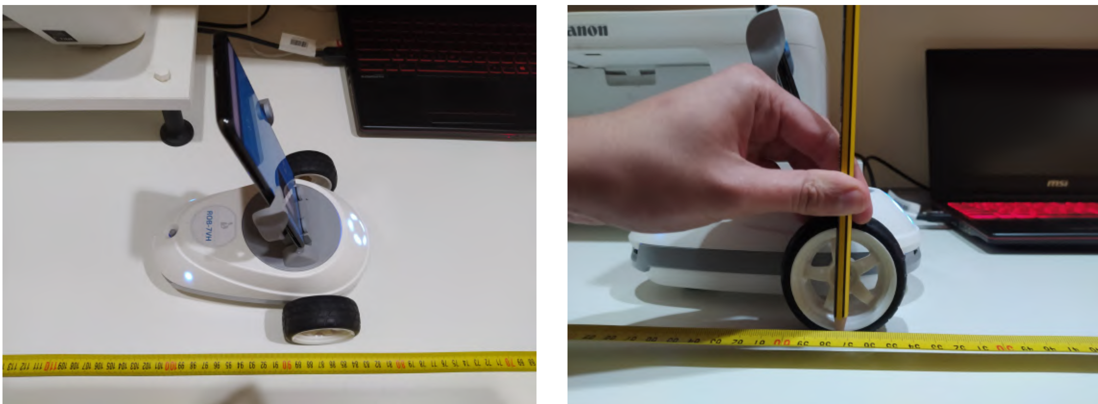


Figura 8.40 – Medición de la distancia durante la calibración del movimiento lineal

Los resultados obtenidos muestran como el robot se comporta de forma lineal, obteniendo valores para el coeficiente de determinación próximo a uno, tal como se puede observar en la gráfica 8.41. En este caso se muestran las medidas obtenidas para una velocidad de 10 8.41 y otra de 50 8.42. Hay que tener en cuenta que no se tiene información total del movimiento, ya que su comportamiento durante el arranque es incierto por lo que se debe tener en cuenta en el modelo.

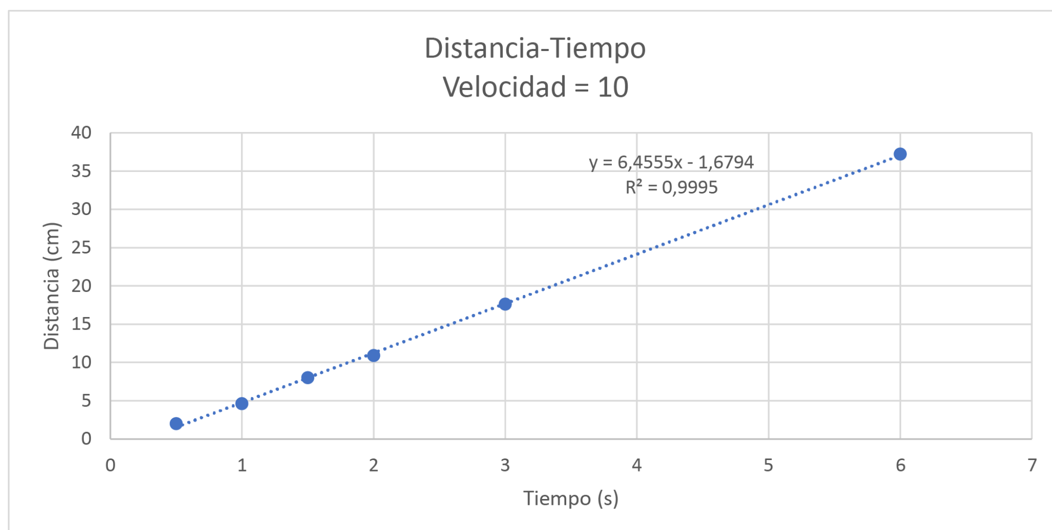


Figura 8.41 – Calibración de la velocidad real para v = 10

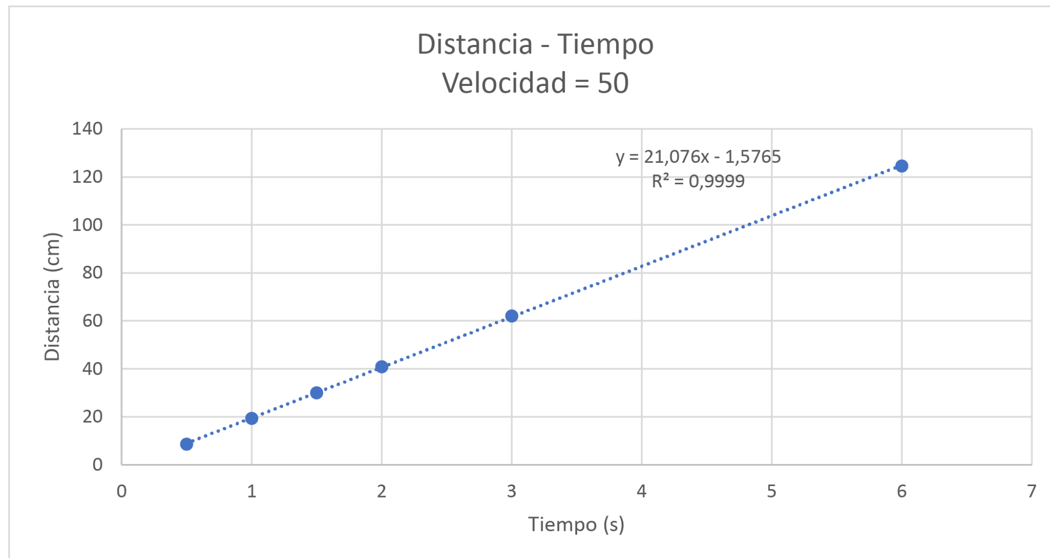


Figura 8.42 – Calibración de la velocidad real para v = 50

Conocida esta información se propone para modelar el movimiento del robot una función por partes. Dicha función estará dividida en tres intervalos separados por el tiempo 8.7. El término $P = V^2 \times C + V \times B + A$ será un polinomio de segundo grado, que depende del porcentaje de velocidad introducido. Con esto y los datos obtenidos de los movimientos a todas la velocidades se plantea un problema de optimización. El objetivo de dicho problema es minimizar el error cuadrático medio, calculado a partir de la diferencia entre la estimación y la medida real. Como restricción se establecen serán que los coeficientes del polinomio P tengan un valor de entre -100 y 100, el término Ap no sea mayor de 100 y $t0$ sea positivo. Para resolver el problema de optimización se utiliza en complemente Solver de Excel. En las tablas 8.17 y 8.18 se muestra los resultados de dicho proceso.

$$f(x) = \begin{cases} 0 & si\ t \leq t_0 \\ \frac{1}{2} \times A_p \times t^2 & si\ t_0 < t < \frac{P}{A_p} \\ \frac{1}{2} A_p (t0 + \frac{P}{A_p})^2 + P * (t - t_0 - \frac{P}{A_p}) & si\ t \geq \frac{P}{A_p} \end{cases} \tag{8.7}$$

A	B	C	Ap	t0
0,0000175	0,3797	2,12	103,56	0,1

Tabla 8.17 – Coficientes obtenidos tras el proceso de optimización de la función por partes.

ECM
0,99108142

Tabla 8.18 – Error cuadrático medio obtenido en el proceso de optimización.

En la figuras 8.43 y 8.44 se muestra la comparativa para V = 30 y V = 50 entre la estimación mediante la función y la distancia real. Se puede observar que para un comando de velocidad más bajo y en un tiempo pequeño se comente más error, pero produce una respuesta aceptable. En cambio, para una velocidad alta se observa que se ajusta mucho mejor al comportamiento real.

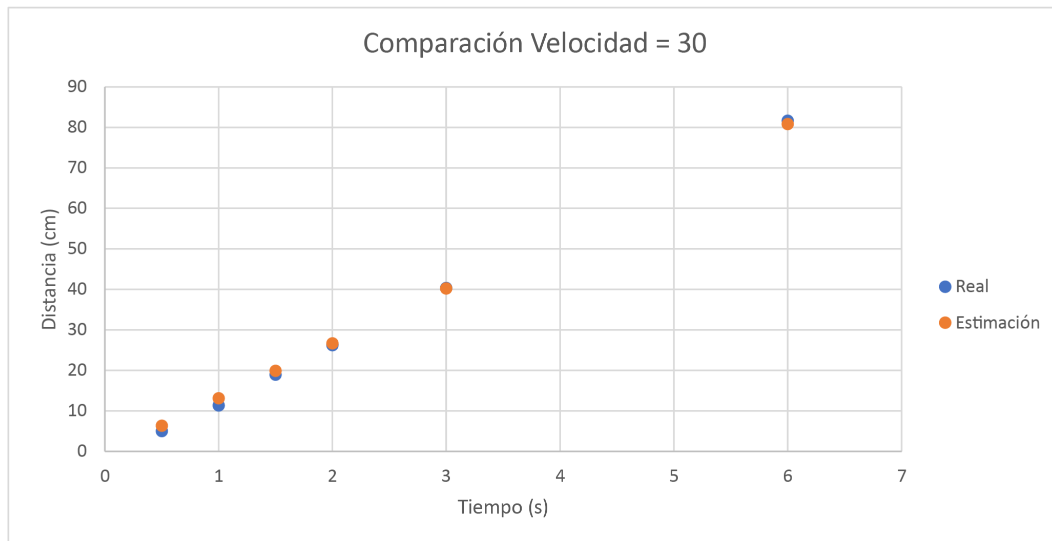


Figura 8.43 – Distancia frente tiempo para v = 30

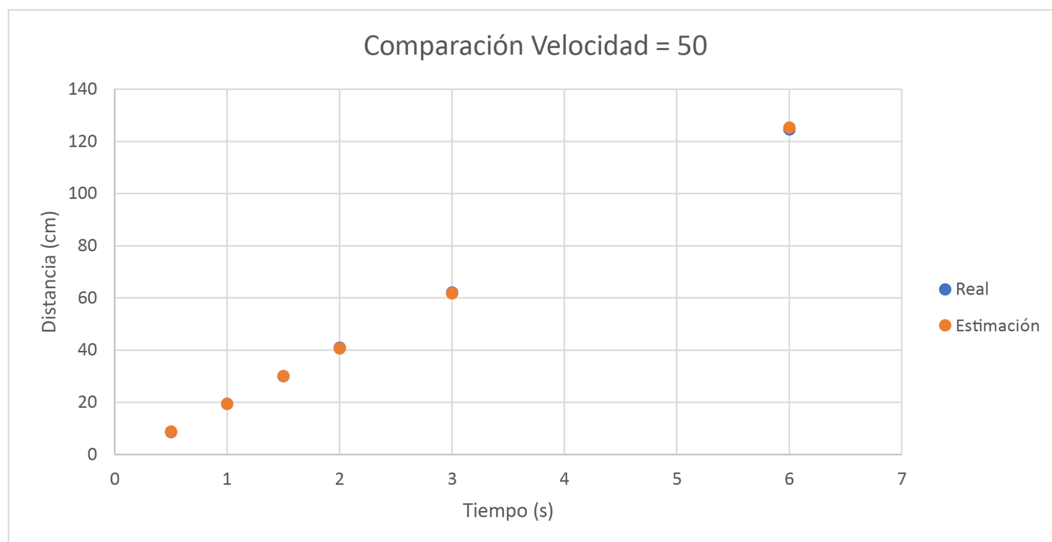


Figura 8.44 – Distancia frente tiempo para v = 50

8.4.2. Modelado velocidad de rotación

Para modelar el movimiento del robot con rotación se recurre a la información obtenida del modelo lineal. En primer lugar, se trabaja bajo el supuesto de que el arco descrito por una de las ruedas se puede aproximar a una distancia lineal d, esto será cierto para ángulos pequeños, aumentando el error estimado conforme aumente este. En las figuras 8.45 y 8.46 se puede observar la geometría del movimiento así como las aproximaciones realizadas. d_l y d_r son las distancias recorridas por las ruedas y d_c la resultante del eje central del robot.

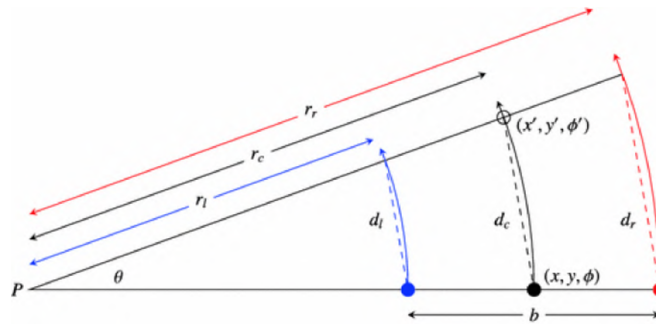


Figura 8.45 – Geometría del movimiento de rotación del robot.

Sabiendo que para cada distancia se tiene $d = r\theta$ se puede obtener las velocidades lineales, ya conocidas del apartado anterior, y angulares derivando ambas magnitudes d y θ, por lo tanto $v = rw$. Particularizando para el eje central del robot, se puede donde el radio será $\frac{r_r+r_l}{2}$ y por lo tanto la relación entre ambas velocidades será $v_c = w\frac{r_r+r_l}{2}$. Finalmente, se puede expresar la ecuación en función de las velocidades lineales en vez del radio $v_c = \frac{v_r+v_l}{2}$. Para calcular el ángulo de la nueva orientación del robot de la ecuación que relaciona las velocidades, en este caso en $\omega r_r - \omega r_l = v_r - v_l$ de donde se deduce $\omega = (v_r - v_l)/(r_r - r_l)$, siendo este el incremento que se debe sumar o restar al ángulo actual.

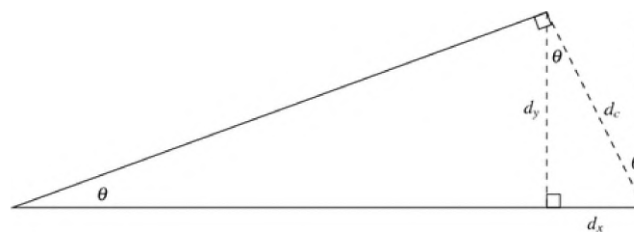


Figura 8.46 – Geometría del movimiento de rotación del robot.

8.5. Implementación del simulador

En esta sección se describe el simulador desarrollado en Pygame para realizar los experimentos y como se implementan los modelos realizados durante este trabajo. El programa consta de un entorno 2D delimitado por cuatro paredes infranqueables. Su dimensiones deben guardar la proporción correspondiente con el mundo real. Por lo tanto, se aplicará un factor de conversión para convertir la magnitudes necesarias, en este caso se ha elegido $0{,}3$ cm/pixel.

El robot en simulación debe responder a las características de su homólogo del mundo real, ya que es clave para poder transferir con el menor error posible el controlador al entorno real. De este modo, se debe aplicar la misma lógica del esquema propuesto al principio de este trabajo. En primer lugar, para la sensorización se utiliza una función proporcionado por el profesor que imita el proceso de obtención de la rendija.

En segundo lugar, los actuadores vienen definidos por el modelo definido en la sección 8.4. Se implementa una función para que el agente pase de una velocidad a otra a través de una aceleración positiva y otra negativa. La aceleración negativa se puede calcular a partir de la ecuación del procedimiento de la sección 8.3.4.2 y la positiva se obtiene del coeficiente A_p de la función 8.7.

Para el cálculo de la velocidad que toma el agente, se implementa la función por partes descrita en 8.7. El primer tramo será incorporado a través de un búfer que retrasará la velocidad hasta superar el tiempo t_0. El segundo se tiene en cuenta mediante la aceleración positiva explicada anteriormente. Por último, la velocidad de navegación será calculada a través del último tramo de la función.

Así mismo, se deben introducir los tiempos analizados en la sección 8.3.4.2 que influyen en la captura de la información y el funcionamiento de los actuadores. Ya que se trabaja con un paso de tiempo de 16 ms, debido a que el simulador funciona a 60 fps, y trabajar con una frecuencia menor no reporta algún tipo de mejoría en la odometría, se utiliza esta frecuencia de funcionamiento en la simulación. En cuanto a los tiempos de retardo, se añaden un búfer que almacena y retrasa la aplicación de los inputs.

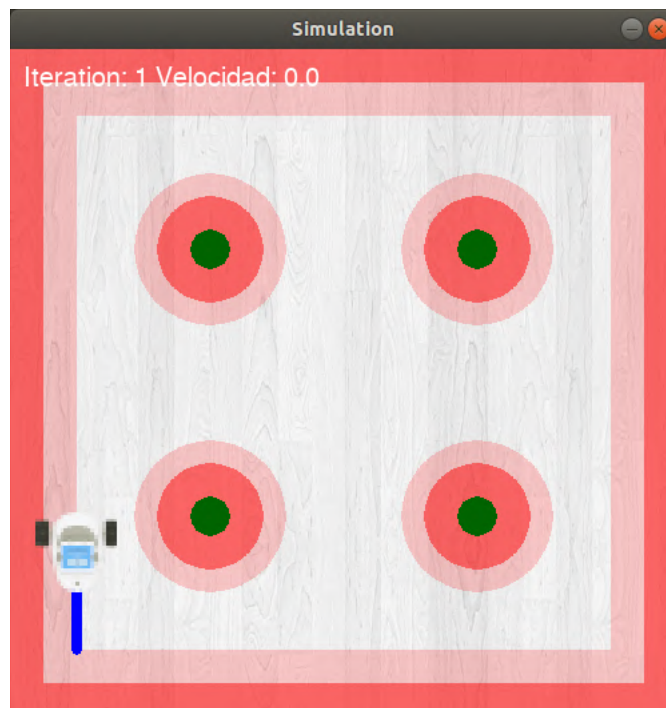


Figura 8.47 – Representación del entorno en Pygame.

En cuanto al apartado visual, el simulador contará con un sprite dedicado para el robot que se asemeja al Robobo real y los obstáculos serán representados con círculos verdes. Con el objetivo de analizar los comportamientos obtenidos en el experimentos se añade al robot un rastro que irá dejando a lo largo de la navegación, de este modo se podrá observar al final del mismo las trayectorias tomadas. Además, a partir de la odometría realizada en la sección 8.3.4, se establecen unas zonas de peligro de color rojo en las que el robot debería de corregir el rumbo para no chocar con los obstáculos o la pared. La tonalidad de la zonas variará según pertenezca a una velocidad media o alta. Finalmente, como información a mostrar en pantalla en el transcurso de la evolución, se utiliza el número de la iteración correspondiente y la velocidad de navegación.

9 RESULTADOS FINALES

En esta sección se expone el experimento finalmente desarrollado, que es fruto de todo el proceso detallado en los apartados anteriores, y el análisis de los resultados obtenidos a partir de este. En primer lugar, se hablará del proceso optimización en simulación. Por último, se explicará el proceso y resultados obtenidos tras transferir los resultados al entorno real.

9.1. Configuración experimental

Este apartado se dedica a la descripción de los parámetros que finalmente se han fijado en el simulador. Las pruebas son realizadas en un espacio de 150 x 150 cm , ocupando los *sprites* del robot y los obstáculos 14 y 9 cm respectivamente. Las colisiones entre *sprites* se contarán a través de una función interna de Pygame y, las que suceden entre la pared y el robot, a través del sensor de rendija. A este sensor se le ha fijado un ángulo de visión de 25° en cada lado, es decir un FOV de 50°. Para la aceleración negativa se ha escogido la media resultante de los datos obtenidos en la tabla 8.16. En cuanto a la velocidad se ha establecido que como máximo se le mandé el 60 % de la velocidad máxima. Esto se debe a que a un porcentaje mayor ya no funcionaría correctamente tal como se comprobó en el apartado 8.3.4. Finalmente, cada paso de tiempo es equivalente a 16 ms. A continuación, se muestra una tabla a modo de resumen de la configuración escogida:

Configuración simulador	
Tamaño	150 x 150
Paso	16 ms
Ángulo FOV	25°
Aceleración positiva	103.56
Aceleración negativa	0.213

Tabla 9.1 – Configuración del simulador.

9.2. Navegación autónoma

En este tipo de tareas el robot debe de navegar por el entorno sin colisionar lo más rápido posible. En este caso, se va a plantear tres versiones diferentes. En la primera se pondrá al agente a moverse en el entorno, donde los únicos obstáculos posibles serán las paredes del entorno. En la segunda, se introducen cuatro obstáculos fijos formando los vértices de un cuadrado. Finalmente, se añade a los obstáculos un movimiento circular. Esta disposición dota al entorno de cierta complejidad, por lo que la solución de la tarea admite distintos tipos de comportamientos. Como condición final de cada evaluación se ha escogido 700 pasos o que el agente detecte alguna colisión.

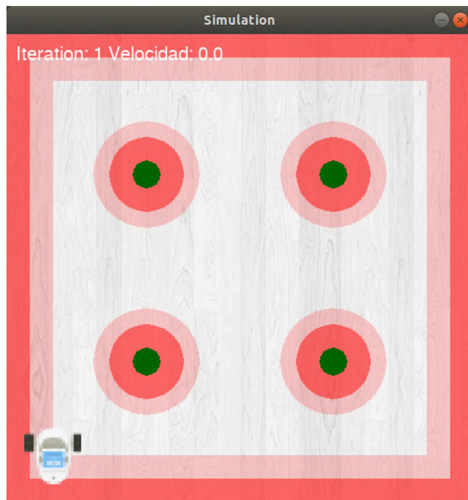


Figura 9.1 – Disposición de los elementos en el entorno durante la prueba estática.

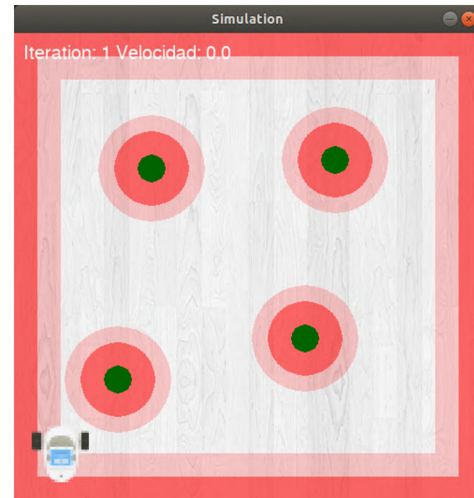


Figura 9.2 – Disposición de los elementos en el entorno durante la prueba con los obstáculos en movimiento.

El comportamiento que se espera que desarrolle el agente tiene las siguientes propiedades: el robot debe de avanzar hacia delante y en línea recta en la medida de lo posible, debe ir lo más rápido que pueda y debe evitar los obstáculos. Para poder mostrar estas propiedades se le asigna la siguiente función de calidad que debe maximizar:

$$f = \overline{v}(1 - \sqrt{\Delta v})F \tag{9.1}$$

$$F = \frac{steps_{colisión}}{steps_{max}} \tag{9.2}$$

El primer término $\overline{v}$ es la velocidad lineal media del agente durante la evaluación, con esto se incentiva que vaya lo más rápido posible. El segundo término, depende de la diferencia absoluta entre las velocidades lineales de ambas ruedas, con esto se motiva el avance en línea recta del robot. Por último, para evitar los obstáculos y por lo tanto el agente siga navegando durante más tiempo, se establece un factor F, resultado del cociente entre los pasos que lleve hasta la colisión y los que duraría como máximo la simulación. Por lo tanto, la función de calidad devolvería un valor máximo de 1 si ambas ruedas girasen a la misma velocidad, que esta fuese máxima y que no se produjese colisión alguna. Por otro lado, devolvería un cero en caso de que una de las velocidades fuese 0 y la otra máxima, o que se produzca una colisión en la primera iteración.

9.3. Configuración algoritmos evolutivos

En este experimento se prueban los dos algoritmos estudiados con anterioridad en este trabajo: Evolución Diferencial y MAP-Elites. Estos son utilizados para evolucionar la red neuronal que controlará el agente. Se utiliza una red neuronal recurrente con arquitectura Jordan, en este caso es simple por lo que solamente tendrá una capa. Los *inputs* serán la rendija y los *outputs* las velocidades de cada rueda. Los pesos de la red son codificados directamente en el cromosoma. En el caso de MAP-Elites, se elige como variables del espacio característico las posición final *(x,y)*. Por lo tanto esto resulta en un espacio característico bidimensonal, que será discretizado en 100 intervalos dando como resultado un total de 10000 celdas.

En primera instancia se hicieron unas pruebas preliminares para ajustar los parámetros tanto de la red como de los algoritmos evolutivos. Debido a que las pruebas mostraron que con la evolución diferencial no mostraba una buena convergencia, se decidió reducir el parámetro F. Tras fijar este parámetro y con el fin de reducir el tiempo de computación, se fija el tamaño de población a 100. En cuanto al MAP-Elites, se sigue la conclusiones obtenidas en la sección 8.2.2 y se comprueba que es capaz de alcanzar soluciones con calidad alta y de abarcar gran parte de las celdas. Dado que comprobar todas la configuraciones posibles para la red neuronal sería computacionalmente costoso en este simulador, se ha decidido probar con 4, 6 y 8 neuronas, siendo el primer caso el que mejor resultado ha dado.

Differential Evolution	
Población	100
Generaciones	100
Cruce	Binomial
Probabilidad de Cruce	0.9
F	0.1

MAP-Elites	
Población inicial	1000
Evaluaciones	10000
Nº características	2
Tamaño del archivo	100 x 100
Cruce	No
Mutación	Gausiana
Probabilidad de Mutación	0.8
σ	0.05

Tabla 9.2 – Configuración de los algoritmos evolutivos

9.4. Resultados y análisis

A continuación, se muestran y analizan los resultados obtenidos en el proceso de optimización en simulación. Para estas pruebas experimentales se definirán tres modelos distintos de comportamiento del robot respecto al caso real. El primero (V1), no tiene en cuenta los tiempos de funcionamiento del robot por lo que es un modelo completamente ideal. El segundo (V2), será el más próximo al caso real ya que si que los tiene en cuenta. Por último (V3), se define un modelo más conservador que el real, es decir, se ha programado duplicando los tiempos de funcionamiento. Cada caso será lanzado 10 veces, representado en una gráfica la calidad máxima promedio obtenida en cada iteración del proceso evolutivo. Además, se muestra un

diagrama de cajas con la calidad máxima obtenida tras cada ejecución.

9.4.1. Calidad de las soluciones

Está sección se centra en el análisis de la calidad del comportamiento según el agoriltmo y modelo utilizados. Como se puede observar en la gráfica 9.3 para el entorno vacío, los dos algoritmos son capaces de alcanzar valores altos de calidad en poco tiempo. Además, es capaz de producir buenos controladores para los tres tipos de modelo, aunque se observa que la calidad empeora conforme se haga más conservador. En cuanto a la distribución de las calidades más altas 9.8, se puede comprobar como siguiendo el comportamiento anterior, cuanto más conservador más varianza se observa. Además, ésta también es mayor para MAP-Elites.

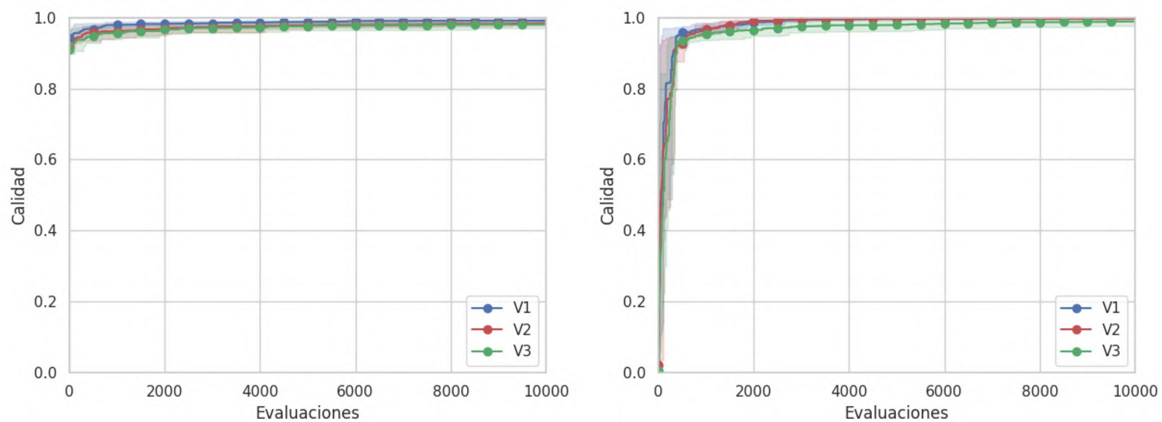


Figura 9.3 – Izquierda: Calidad máxima para las tres configuraciones en cada iteración de MAP-Elites en el entorno vacío. Derecha: Calidad máxima para las tres configuraciones en cada iteración de Evolución Diferencial en el entorno vacío.

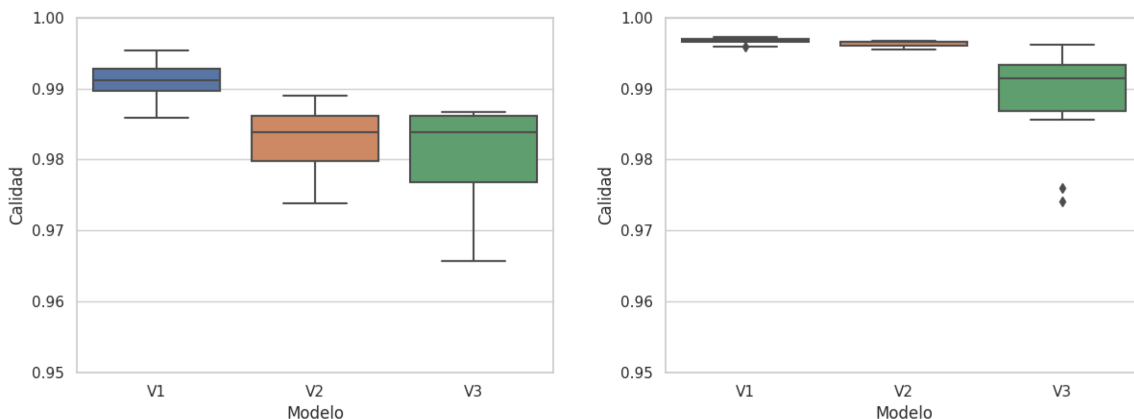


Figura 9.4 – Izquierda: Calidad máxima para las tres configuraciones en cada ejecución de MAP-Elites en el entorno vacío. Derecha: Calidad máxima para las tres configuraciones en cada ejecución de Evolución Diferencial en el entorno vacío.

En las pruebas realizadas en el entorno con obstáculos, se observa el aumento de dificultad de la tarea, ya que cualquiera de lo dos algoritmos tarda más en convergen hacía una buena solución 9.5. Si se compara entre ellos, MAP-Elites converge antes en cualquiera de los modelos. En concreto, es capaz de reducir la diferencia entre el modelo V1 con el V2, no siendo así con el V3. Por otro lado, en el algoritmo diferencial el modelo V2 deja su calidad casi al mismo nivel que el modelo V3. Si se compara la variabilidad durante el proceso, la evolución diferencial presenta valores mucho más alto, sobretodo para el caso V2. En cuanto a la distribución de los máximos 9.6, se comprueba como el algoritmo diferencial tiene más varianza excepto para el modelo V3.

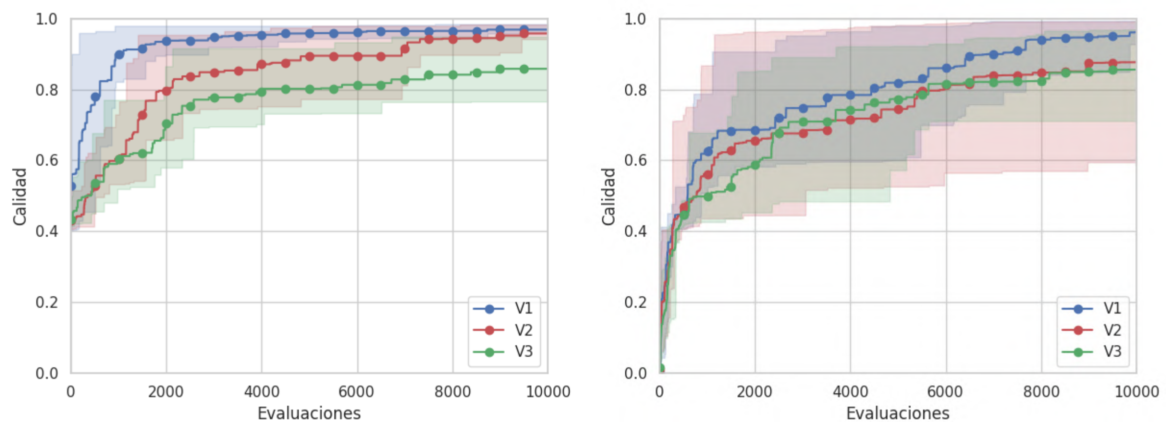


Figura 9.5 – Izquierda: Calidad máxima para las tres configuraciones en cada iteración de MAP-Elites con obstáculos estáticos. Derecha: Calidad máxima para las tres configuraciones en cada iteración de Evolución Diferencial con obstáculos estáticos.

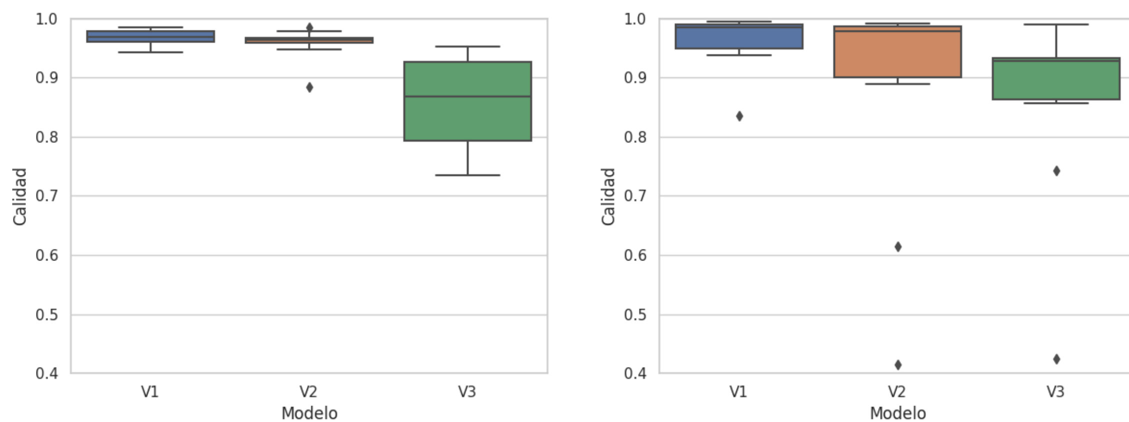


Figura 9.6 – Izquierda: Calidad máxima para las tres configuraciones en cada ejecución de MAP-Elites con obstáculos estáticos. Derecha: Calidad máxima para las tres configuraciones en cada ejecución de Evolución Diferencial con obstáculos estáticos.

Finalmente, en el caso del entorno con los obstáculos en movimiento se observa el mismo comportamiento que en los casos anteriores. Es el modelo V1 el que alcanza la mejor calidad en ambos algoritmos y desciende conforme se hace más conservador el modelo. Durante algunas iteraciones del MAP-Elites, la calidad del controlador V3 se acerca al V2 pero, finalmente, acaba por obtener una calidad menor 9.7. Atendiendo a la variabilidad, se puede observar como el que presenta más es el V2 seguido del conservador 9.8. En cualquiera de ambos algoritmos se puede ver una gran diferencia entre el V2 y el V3, pero en MAP-Elites, está diferencia decrece.

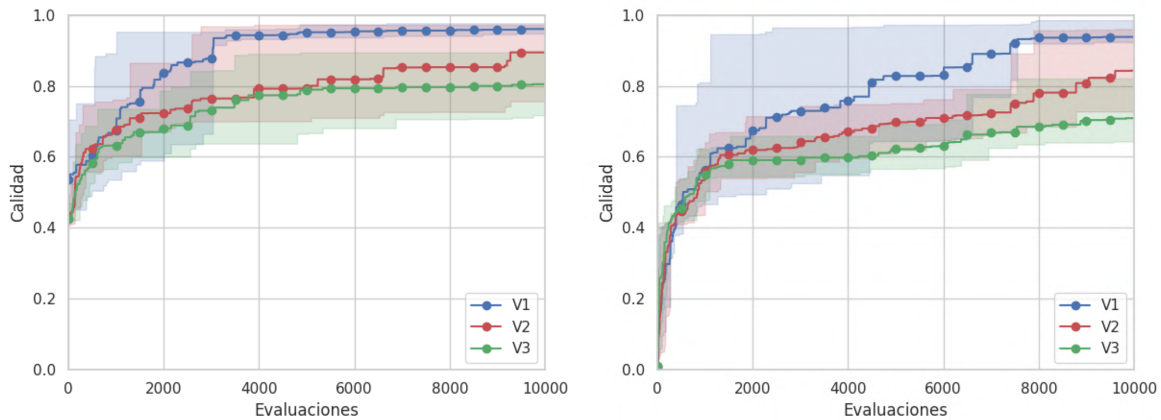


Figura 9.7 – Izquierda: Calidad máxima para las tres configuraciones en cada iteración de MAP-Elites con obstáculos dinámicos. Derecha: Calidad máxima para las tres configuraciones en cada iteración de Evolución Diferencial con obstáculos dinámicos.

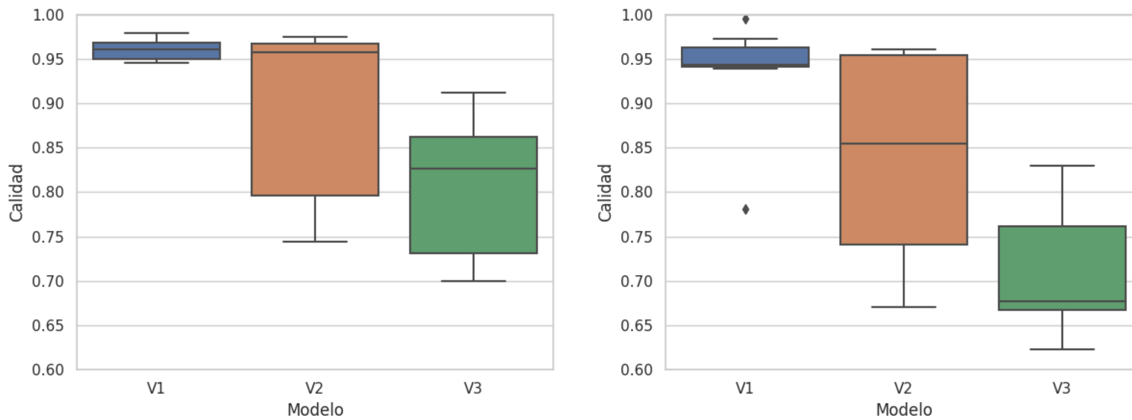


Figura 9.8 – Izquierda: Calidad máxima para las tres configuraciones en cada ejecución de MAP-Elites con obstáculos dinámicos. Derecha: Calidad máxima para las tres configuraciones en cada ejecución de Evolución Diferencial con obstáculos dinámicos.

9.4.2. Diversidad de las soluciones

Debido a que uno de los motivos del estudio y aplicación de MAP-Elites es su capacidad de generar conjuntos de soluciones diversas, es necesario realizar un análisis y comparativa de los comportamientos producidos por los algoritmos. Para ello se ha escogido un repertorio de soluciones producidas con el modelo real.

9.4.2.1. Comportamientos en entorno vacío

En primer lugar, se toma comportamientos de la tarea sin obstáculos. En las figuras 9.9 y 9.10, se puede observar como el algoritmo diferencial solo es capaz de producir un tipo de trayectorias casi indistinguibles. En cambio, MAP-Elites produce dos tipos de soluciones completamente diferentes, pero igualmente con valores de calidad alta.

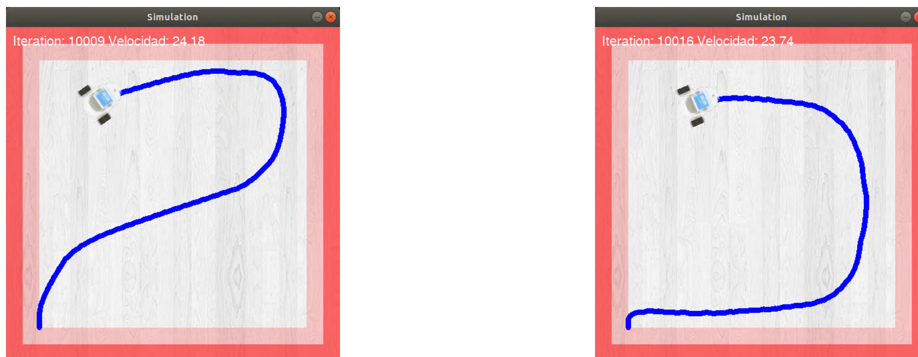


Figura 9.9 – Trayectorias producidas por MAP-Elites en el entorno vacío.

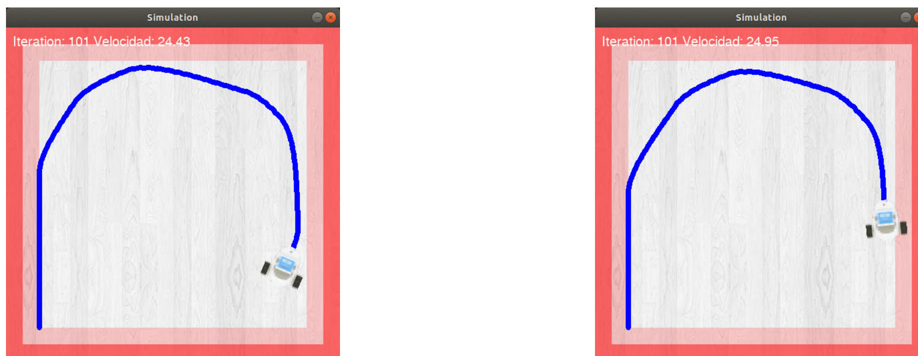


Figura 9.10 – Trayectorias producidas por Evolución Diferencial en el entorno vacío.

9.4.2.2. Comportamientos en entorno con obstáculos

En cuanto a los comportamientos desarrollados en el entorno con obstáculos 9.11, se observa que, de la misma forma, el algoritmo diferencial recurre a la misma estrategia. Por lo tanto, tiene un comportamiento más estable y más ajustado al entorno . El MAP-Elites por su parte es capaz de llegar a la misma solución y, además, toma un estrategia totalmente opuesta. En este comportamiento, se observa que hace un giro más brusco y es capaz corregir antes de llegar a la zona de peligro.

Figura 9.11 – Trayectorias producidas por MAP-Elites en el entorno con obstáculos.

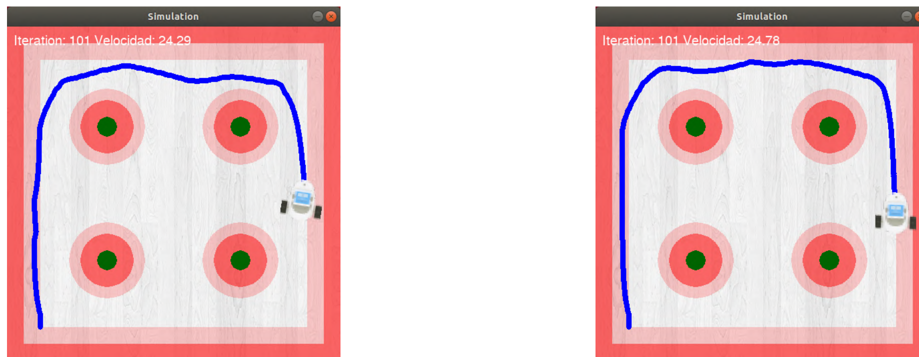


Figura 9.12 – Trayectorias producidas por Evolución Diferencial en el entorno con obstáculos.

9.4.2.3. Comportamientos en entorno con obstáculos en movimiento

Por último, en el entorno con obstáculos móviles, sucede lo mismo que en el caso anterior. En este caso, como los obstáculos son móviles, añade una dificultad extra al problema, y el algorimo diferencial queda atascado en un mínimo local. No es así con MAP-Elites, que es capaz de desarrollar un comportamiento que responde mejor a los obstáculos y por lo tanto alcanzando mejores valores de calidad. En este caso, las trayectorias no muestran adecuadamente el comportamiento, ya que parece que han ocurrido colisiones. Esto se debe a que la captura se ha realizado al final de la evaluación y los obstáculos han estado moviéndose durante el proceso.

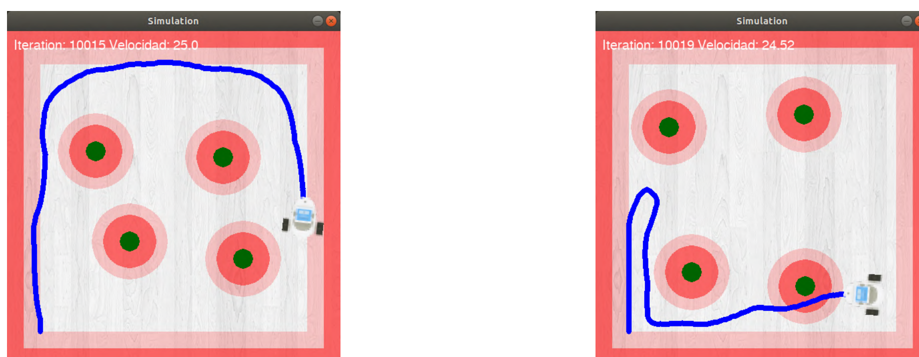


Figura 9.13 – Trayectorias producidas por MAP-Elites en el entorno con obstáculos dinámicos.

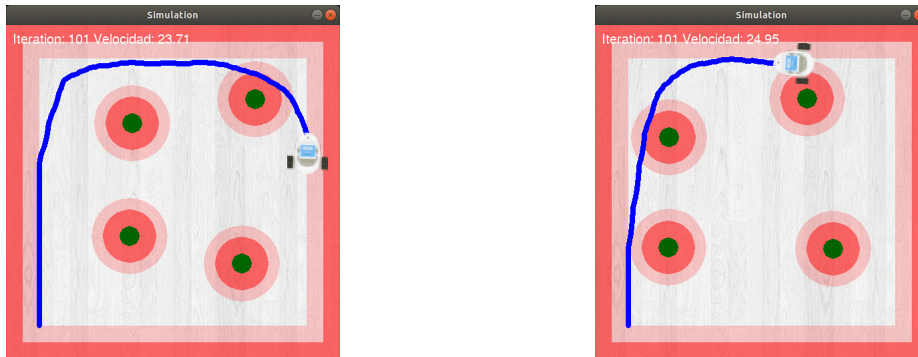


Figura 9.14 – Trayectorias producidas por Evolución Diferencial en el entorno con obstáculos dinámicos.

9.4.2.4. Análisis de Mapas de Calor de MAP-Elites

Además de estos comportamientos, se exportan los mapas de calor generados a partir de MAP-Elites en cada una de las tareas. La estructura que siguen es la misma que en el apartado 8.2.2, a diferencia de que en este caso se trata de un espacio bidimiensional. Hay que aclarar que, aunque se han elegido como variables las coordenadas de la posición final, el mapa está invertido respecto a las imágenes de la simulación. Esto ocurre, debido a que la característica 1, que es la coordenada x, está situado en el eje vertical. Como se puede observar en todos los mapas consigue explorar una gran parte del espacio característico, distinguiéndose las combinaciones con valores de calidad altos. En el caso del entorno vacío 9.15, al tratarse de un problema más sencillo, la zona donde la calidad es más alta es muy extensa. Conforme se aumenta de complejidad, está zona está mucho más limitada. Tanto en el caso de los obstáculos estáticos 9.16 como en el dinámico 9.17, se observa las posiciones finales alcanzadas en los repertorios mostrados con anterioridad, así como otras zonas de menor calidad que darían lugar a otro tipo de trayectorias.

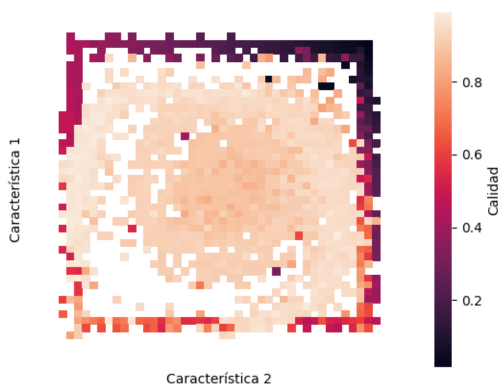


Figura 9.15 – Mapas de soluciones de una ejecución en un entorno vacío.

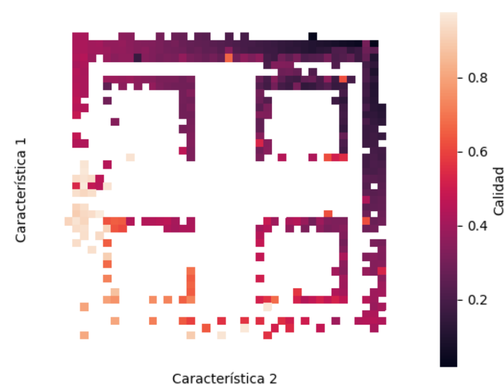


Figura 9.16 – Mapas de soluciones de una ejecución en un entorno con obstáculos.

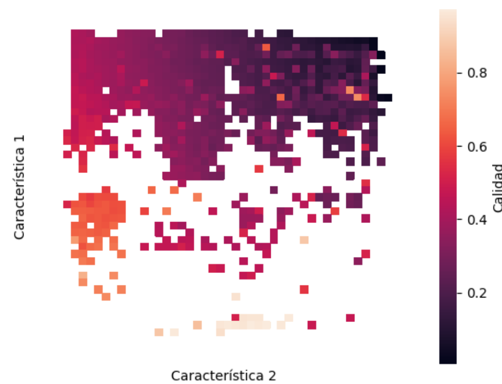


Figura 9.17 – Mapas de soluciones de una ejecución en un entorno con obstáculos móviles.

9.5. Transferencia al entorno real

Tras realizar el proceso de optimización en simulación, se procede a transferir al entorno real los comportamientos evolucionados del apartado anterior. Una vez realizadas una serie de pruebas iniciales, se observa que la estimación obtenida en el apartado 8.4.2, no es suficiente, y el robot real presenta una respuesta en rotación diferente. Para solventar este problema, se utiliza un coeficiente que tiene en cuenta la relación entre la diferencia de las ruedas y una constante para ajustarlo empíricamente.

$$C = \frac{V_1 + V_2}{15 * (V_1 - V_2)} \tag{9.3}$$

$$V_2 = \frac{C - 1}{C + 1} \tag{9.4}$$

siendo V_1 la velocidad más grande de las dos ruedas.

Tras realizar este ajuste final, se prueban los controladores desarrollados. En primer lugar, se evalúan los controladores producidos por el algoritmo diferencial en el caso del entorno vacío. Se observa que el controlador V1 no funciona correctamente y directamente colisiona con la pared. Esto se debe a que es un controlador ideal, por lo tanto no es capaz de dar las órdenes a tiempo y reacciona muy tarde. Esto se puede comprobar en las figuras 9.18 y 9.19. En el caso V2, es capaz de esquivar la pared y sigue una trayectoria similar a la simulación, ya que en este caso si que tiene en cuenta los tiempos, reacciona antes y es capaz de realizar el comportamiento correctamente. Finalmente, con el modelo V3, el robot reacciona antes a la pared y describe una trayectoria diferente, tal como se observa en las figuras 9.20 y 9.21. Por lo tanto, se comprueba que al hacer el controlador más conservador, sigue siendo capaz de realizar la tarea, aunque dando la orden de giro mucho antes.

Figura 9.18 – Trayectoria desarrollada por el controlador V2 en un entorno vacío.

Figura 9.19 – Trayectoria desarrollada por el controlador V2 en un entorno vacío.

Figura 9.20 – Trayectoria desarrollada por el controlador V3 en un entorno vacío.

Figura 9.21 – Trayectoria desarrollada por el controlador V3 en un entorno vacío.

En el caso de los obstáculos, se prueban tanto los controladores resultantes de las tareas en estático como en movimiento de MAP-Elites. En el primer caso, se observa un comportamiento similar al desarrollado en el entorno vacío. Cuando el robot se encuentra con un obstáculo no es capaz de esquivarlo. Esto se debe a que la solución que toma en simulación es ir pegado a la pared para no encontrarse de frente con uno de los cilindros. De este modo, el modelo en simulación no es suficiente para superar la barrera que supone las variaciones del entorno real. Por otro lado, los controladores resultantes del proceso evolutivo en un entorno con obstáculos móviles, reaccionan a estos. Es decir, el hecho de entrenarlo con estos obstáculos dinámicos hace que sea un controlador mucho más robusto, capaz de adaptarse a las variaciones del entorno real. Esto se puede comprobar en las figuras 9.22 para el modelo V2. En cuanto al modelo utilizado, también se observa que sigue el mismo patrón que en el entorno anterior, ya que el controlador reacciona antes los obstáculos 9.23.

Figura 9.22 – Trayectoria seguida por el controlador V2 en un entorno con obstáculos.

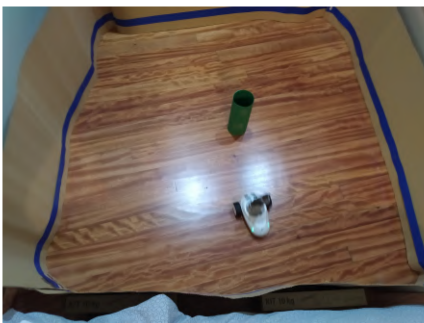
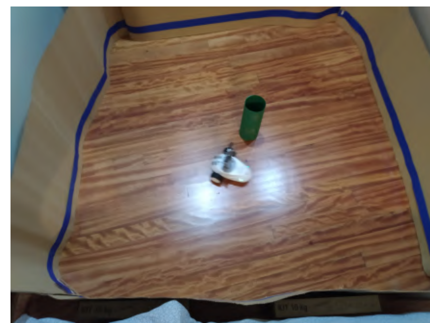



Figura 9.23 – Trayectoria seguida por el controlador V3 en un entorno con obstáculos.

10 CONCLUSIONES Y TRABAJO FUTURO

10.1. Conclusiones

En este apartado se describen las conclusiones sacadas tras la realización de este trabajo desde dos puntos de vista diferentes: uno desde el conocimiento adquirido y otro, desde las conclusiones extraídas de los resultados.

En primer lugar, la realización de este trabajo me ha brindado la oportunidad de adquirir capacidades como la búsqueda y análisis crítico de información. Desde el plano técnico, como el trabajo ha sido desarrollado en su totalidad en Python, he podido aprender a trabajar con este lenguaje de forma estructurada. Dentro de la robótica, he podido desarrollar una metodología de trabajo que tiene como fin el modelado de una unidad robótica real. Durante este proceso he aprendido a desarrollar y optimizar modelos físico-matemáticos y analizar y representar los datos obtenidos a partir de estos. En cuanto a la inteligencia artificial, he profundizado en dos áreas de especial relevancia dentro de este campo, como son los algoritmos evolutivos y las redes neuronales artificiales.

Desde el punto de vista de los resultados obtenidos durante la realización de este trabajo, se pueden comentar una serie de conclusiones. Se ha podido comprobar que los algoritmos QD son capaces de arrojar resultados interesantes ante su aplicación a problemas de robótica autónoma. Produciendo así una serie de comportamientos que no serían posibles a través de métodos de optimización más clásicos, pero que a su vez muestran de la misma forma, un alto rendimiento en las tareas propuestas. Además, se ha podido ver la importancia de una correcta modelización del comportamiento del robot, ya que es clave para su transferencia de un simulador a un entorno real, y como este afecta a las diferentes soluciones desarrolladas.

10.2. Trabajo futuro

En esta sección se tratará las posibles líneas futuras de trabajo, así como aspectos que podrían complementar los resultados obtenidos en futuros proyectos. A continuación se detalla una lista con las diferentes propuestas:

- **Paralelización:** Aplicar este tipo de herramientas de optimización a problemas más complejos requiere unos tiempos de cálculo mayores. Por lo tanto, se propone utilizar algún tipo de infraestructura paralela para aumentar la la eficiencia de la implementación de estos algoritmos.

- **Ampliación de los algoritmos de optimización implementados:** En este trabajo se ha estudiado uno de los algoritmos pertenecientes a los llamados algoritmos QD, pero en la actualidad se vienen desarrollando nuevas alternativas a este, algunas de ellas como resultado de incorporar a este elementos de otras técnicas. Otra alternativa podría ser utilizar los resultados obtenidos con MAP-Elites como punto de partida para otras herramientas de optimización.

- **Comparación con otras técnicas de la inteligencia artificial:** Dentro de este campo existen otras técnicas aplicadas a la robótica autónoma que son relevantes actualmente. Es el caso del aprendizaje reforzado, del que se puede encontrar diversos ejemplos en trabajos actuales. Por lo que sería interesante implementarlo y compararlo con las técnicas aplicadas.

11 ORDEN DE PRIORIDAD ENTRE LOS DOCUMENTOS

1. Memoria
2. Anexos
3. Presupuesto

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

ANEXOS

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBREIRO, S/N

15405 - FERROL

FECHA: SEPTIEMBRE DE 2020

AUTOR: EL ALUMNO

Fdo.: DIEGO JAVIER GUTIÉRREZ BARRIO

Índice del documento ANEXOS

12 DOCUMENTACIÓN DE PARTIDA **95**

12.1 Propuesta inicial de asignación de TFG 95

13 CÓDIGOS DE PROGRAMACIÓN **98**

13.1 Algoritmos Evolutivos 98

13.2 Simulador 103

12 DOCUMENTACIÓN DE PARTIDA

12.1. Propuesta inicial de asignación de TFG

Escuela Universitaria Politécnica Ferrol
UNIVERSIDADE DA CORUÑA

Intranet eup

ESCUELA UNIVERSITARIA POLITÉCNICA

ASIGNACIÓN DE TRABAJO FIN DE GRADO

En virtud de la solicitud efectuada por:
En virtud da solicitude efectuada por:

APELLIDOS, NOMBRE: *Gutiérrez Barrio,Diego Javier*
APELIDOS E NOME:

DNI: [redacted] **Fecha de Solicitud:** OCT2019
DNI: *Fecha de Solicitude:*

Alumno de esta escuela en la titulación de Grado en Ingeniería en Electrónica Industrial y Automática, se le comunica que la Comisión de Proyectos ha decidido asignarle el siguiente Trabajo Fin de Grado:

O alumno de esta escola na titulación de Grado en Enxeñería en Electrónica Industrial e Automática, comunícaselle que a Comisión de Proxectos ha decidido asignarlle o seguinte Traballo Fin de Grado:

Título T.F.G:Desarrollo de un experimento de robótica colectiva mediante evolución distribuida

Número TFG: *770G01A185*

TUTOR:(Titor) *Prieto García,Abraham*

COTUTOR/CODIRECTOR: *Pedro Trueba Martinez*

La descripción y objetivos del Trabajo son los que figuran en el reverso de este documento:

A descrición e obxectivos do proxecto son os que figuran no reverso deste documento.

Ferrol a Miercoles, 6 de Noviembre del 2019

Retirei o meu Traballo Fin de Grado o dia ____ de _____ do ano _________

Fdo: Gutiérrez Barrio,Diego Javier

Documento Generado automaticamente el: 06/11/2019 a las: 12:47:42 desde https://www.eup.udc.es/trabajosfindegrado/

DESCRIPCIÓN Y OBJETIVO:OBJETO: Diseñar e implementar un experimento de coordinación autónoma mediante un equipo de robots reales utilizando algoritmos evolutivos

ALCANCE: Se utilizará un algoritmo evolutivo especialmente adecuado para la optimización de sistemas multirobot para implementar un experimento con un equipo de robots reales.

Los robots a utilizar serán el modelo ROBOBO desarrollado en la propia UDC y que se basa en el uso de una plataforma robótica móvil que transporta un smartphone, que es el que proporciona la sensorización, actuación y procesado al robot.
El experimento tendrá como objetivo mostrar la aplicabilidad de este tipo de algoritmos en un caso real, para lo cual el experimento desarrollado deberá tener los siguientes requisitos:

- Comunicaciones descentralizadas
- Alto grado de autonomía en los robots
- Utilización de las capacidades sensoriales de alto nivel de los ROBOBO: cámara, micrófono, etc
- Utilización de las capacidades de comunicación de los ROBOBO: WI-FI, Bluetooth, etc

Una parte importante de la labor del alumno será diseñar el experimento en base a estos requisitos.

Metodología:

El método de trabajo utilizado se basará en el establecimiento de metas claras y concisas a corto plazo por parte de los directores que supervisarán la consecución de las mismas. El desarrollo de la aplicación se llevará a cabo siguiendo la metodología definida en el Proceso Unificado de Desarrollo de Software. Esta metodología propone el desarrollo de un esquema iterativo e incremental. Las distintas iteraciones se centrarán en aspectos relevantes del software. En cada iteración se llevarán a cabo las fases clásicas de análisis, diseño, implementación y pruebas, de forma que al final del proceso se detecten debilidades, problemas de rendimiento que sean necesarios abordar en la siguiente iteración. Cada iteración incorporará más funcionalidades sobre la anterior, hasta que la última finalizará con un software que implementa la totalidad de las funcionalidades.

Fases:

1- Estudio y familiarización con los algoritmos evolutivos para robótica colectiva, y en particular, de la implementación que proporcionará el tutor
2- Estudio y familiarización con el robot basado en smartphone
3- Diseño del experimento en base a los requisitos del tutor
4- Implementación del experimento y pruebas de optimización
4- Análisis de resultados y refinamiento de la implementación
5- Documentación

Material:

Un ordenador con el IDE de programación y 6 unidades de ROBOBO con 6 smartphones

13 CÓDIGOS DE PROGRAMACIÓN

13.1. Algoritmos Evolutivos

```
0  import math as math
1  import numpy as np
2  import operator as op
3  import logging as logger
4  import seaborn as sns
5  import pandas as pd
6  from mpl_toolkits.mplot3d import Axes3D
7  import datetime
8
9  from matplotlib import pyplot as plt
10 from pandas import DataFrame
11
12 class MAPElites():
13
14     def __init__(self,
15                  iterations,
16                  population_size,
17                  num_bins,
18                  bins,
19                  dimensions,
20                  feature_dimensions,
21                  crossover_rate,
22                  sigma,
23                  mutation_rate,
24                  flag_crossover,
25                  minimization = True):
26
27
28         """
29         Parámetros:
30         - iterations: Número de iteraciones con la que controlar el fin del bucle
31         - population_size: Población inicial
32         - dimensions: Dimensiones del problema
33         - feature_dimensions: Dimensiones del espacio de características
34         - crossover_rate: Ratio de cruce.
35         - flag_crossover: Flag para activar el cruce
36         - mutation_rate : Ratio de mutación
37         - minimization : True = minimizar False = maximizar
38         - num_bins: Número de celdas del mapa
39         - bins: Vector que contiene los intervalos de las celdas
40         """
41
42         self.iterations = iterations
43         self.population_size = population_size
44         self.dimensions = dimensions
45         self.feature_dimensions = feature_dimensions
46         self.crossover_rate = crossover_rate
47         self.flag_crossover = flag_crossover
48         self.mutation_rate = mutation_rate
49         self.minimization = minimization
50         self.num_bins = num_bins
51         self.bins = bins
52         self.best_individual = 0
53         self.replace = 0
54         self.seed = 30
55         self.sigma = sigma
56         self.min_global = 10000
57         self.tiempo_inicial = datetime.datetime.now()
58
59         if self.minimization:
60             self.comp = op.lt
61         else:
62             self.comp = op.gt
63
64         """
65         Matriz de N-Dimensiones, donde N es el número de dimensiones del espacio de atributos,
```

```
66          que contendrá el mapa de celdas. Cada celda tendrá un objeto de la forma:
67          list(np.array([x0,x1,...xn], perf)) , siendo n el número de variables del problema.
68          Sea crea otra matriz de las misma dimensiones con la que facilitar la representación gráfica.
69          """
70
71          self.dimensiones_matriz = (self.num_bins,)*(self.feature_dimensions)
72          self.performances = np.full(self.dimensiones_matriz,np.inf)
73          self.solutions = np.full(self.dimensiones_matriz,np.inf,dtype = object)
74          self.solutions.fill([np.inf,np.inf])
75
76      def mapeo(self, individual):
77          # Mapeo de cada individuo en la celda correspondiente
78          index = self.gentospace(individual)
79          celda = self.solutions[index]
80          if self.comp(self.performance(individual), celda[1]):
81              self.solutions[index] = [individual, self.performance(individual)]
82              self.performances[index] = self.performance(individual)
83              self.replace += 1
84          if self.performances[index] < self.min_global:
85              self.min_global = self.performances[index]
86
87
88      def gentospace(self, individual):
89          # Devuelve el índice del individuo en cada una de los atributos
90          index = tuple()
91          for i in range(self.feature_dimensions):
92              b = np.digitize(individual[i],self.bins,right = False)
93              index = index + (b-1,)
94          return index
95
96      def future_space(self):
97          # Función para calcular el valor del individuo en cada característica.
98          # En este caso como se utiliza las propias variables de la función no se utiliza.
99          pass
100
101     def generate_initial_population(self):
102         # Genera cada solución y la mapea
103         for i in range(0,self.population_size):
104             random_solution = self.generate_random_solution()
105             self.mapeo(random_solution)
106
107     def generate_random_solution(self):
108         new_population = [np.random.uniform(low= -5.12, high= 5.12) for d in range(self.dimensions)]
109         return new_population
110
111     def performance(self, individual):
112         # Función Rastringin
113         fitness = 10*self.dimensions
114         for i in range(len(individual)):
115             fitness += individual[i]**2 - (10*math.cos(2*math.pi*individual[i]))
116         return fitness
117
118     def selection(self, numero_individuos):
119         # Búsqueda de individuo(s) en las celdas ocupadas.
120         # El candidato tendrá la siguiente forma [np.array[x0,x1...],....np.array[x0,x1]]
121         condicion_busqueda = False
122         individuo_aleatorio = tuple()
123         for i in range(numero_individuos):
124             while condicion_busqueda == False:
125                 candidate_index = np.random.randint(self.num_bins, size=self.feature_dimensions)
126                 candidate = self.solutions[tuple(candidate_index)][0]
127                 if self.solutions[tuple(candidate_index)][1] != np.inf:
128                     condicion_busqueda = True
129             individuo_aleatorio = individuo_aleatorio + (candidate,)
130         return individuo_aleatorio
131
132     def mutation(self, individual):
133         # Se utiliza la mutación gaussiana para añadir una pequeña mutación
134         mu = 0
135         if np.random.random() < self.mutation_rate:
136             variation = np.random.normal(mu, self.sigma, self.dimensions)
137             new_individual = individual + variation
138             new_individual = np.clip(new_individual,-5.12,5.12)
139         else:
140             new_individual = individual
141         return new_individual
142
143     def crossover(self, individual):
144         # El cruce de individuos se realiza a través del cruce uniforme
```

```
145         individual1 = individual[0]
146         individual2 = individual[1]
147         for i in range(len(individual)):
148             if np.random.random() < crossover_rate:
149                 individual1[i], individual2[i] = individual2[i], individual1[i]
150         return individual1, individual2
151
152     def ploteo(self):
153
154         # Ratio de sustitución
155         ratio_subs = self.iterations/self.replace
156         print(self.iterations, self.replace)
157         print("Ratio de sustituciones:")
158         print(ratio_subs,"\n")
159
160         # Porcentaje de celdas ocupadas
161         occupied = np.where(self.performances != np.inf)
162         percentege = (len(occupied[0]) / self.performances.size)*100
163         print("Porcentaje de celdas ocupadas:")
164         print(percentege,"\n")
165
166         # Mejor solución
167         index_best_performances = np.where(self.performances == np.amin(self.performances))
168         index_best_solution = tuple()
169         for i in range(len(index_best_performances)):
170             index_best_solution = index_best_solution + (tuple(index_best_performances[i]),)
171         self.best_individual = self.solutions[index_best_solution]
172         best_ind = self.best_individual[0]
173         print("Mejor individuo:")
174         print(best_ind,"\n")
175
176         # Norma
177         norma = np.linalg.norm(best_ind[0])
178         print("Norma:")
179         print(norma,"\n")
180
181         # Guardar medidas
182         self.vector_resultados = [norma, ratio_subs, percentege, best_ind[1]]
183
184         # Redimensionamiento de los datos según feature dimensions.
185         if self.feature_dimensions == 1:
186             df = pd.DataFrame(self.performances)
187             df.replace(np.inf, np.nan, inplace = True)
188         if self.feature_dimensions == 2:
189             df = pd.DataFrame(self.performances)
190             df.replace(np.inf, np.nan, inplace = True)
191         if self.feature_dimensions == 3:
192             data = self.performances.reshape(self.dimensiones_matriz[0]*self.dimensiones_matriz[2],self.dimensiones_matriz[1])
193             df = pd.DataFrame(data)
194             df.replace(np.inf, np.nan, inplace = True)
195         if self.feature_dimensions == 4:
196             data = self.performances.transpose(0,2,1,3).reshape(self.num_bins**2,self.num_bins**2)
197             data_2 = self.solutions.transpose(0, 2, 1, 3).reshape(self.num_bins ** 2, self.num_bins ** 2)
198             df = pd.DataFrame(data)
199             df.replace(np.inf, np.nan, inplace = True)
200         if self.feature_dimensions == 5:
201             data = self.performances.transpose(0,1,3,2,4).reshape(self.num_bins**3,self.num_bins**2)
202             df = pd.DataFrame(data)
203             df.replace(np.inf, np.nan, inplace = True)
204         if self.feature_dimensions == 6:
205             data = self.performances.transpose(0,2,4,1,3,5).reshape(self.num_bins**3,self.num_bins**3)
206             df = pd.DataFrame(data)
207             df.replace(np.inf, np.nan, inplace = True)
208
209         sns.heatmap(df, yticklabels = False, xticklabels = False, square = False, cbar_kws={'label': 'Calidad'})
210         plt.xlabel("Característica 2")
211         plt.ylabel("Característica 1")
212         plt.savefig('ejemplo_map_elites.eps')
213
214     def run(self):
215
216         self.tiempo_inicial = datetime.datetime.now()
217
218         self.generate_initial_population()
219         while True:
220             if self.flag_crossover == True:
221                 individual = self.selection(numero_individuos = 2)
222                 ind = self.crossover(individual)[0]
223                 ind = self.mutation(ind)
```

```
224             else:
225                 individual = self.selection(numero_individuos = 1)[0]
226                 ind = self.mutation(individual)
227             self.mapeo(ind)
228             tiempo_transcurrido = datetime.datetime.now() - self.tiempo_inicial
229             if tiempo_transcurrido.total_seconds() >= 30:
230                 break
231 
232         print("Tiempo transcurrido:")
233         print(tiempo_transcurrido,"\n")
234         self.ploteo()
235         return self.vector_resultados
236 
237 # Future space
238 upperlimit = 5.12
239 lowerlimit = -5.12
240 dimensions = 4
241 feature_dimensions = 2
242 granulation = 0.1
243 bins = np.arange(lowerlimit, upperlimit, granulation)
244 num_bins = len(bins)
245 cross = False
246 
247 # Parámetros del algoritmo evolutivo
248 crossover_rate = 0.2
249 mutation_rate = 0.8
250 sigma = 0.05
251 iterations = 1000
252 population_size = 10000
253 
254 print("Parámetros:")
255 print("Dimensiones = ", dimensions, " Características = ", feature_dimensions,
256 " Granulación = ", granulation, " Población inicial = ", population_size," Sigma = ", sigma, " Mutación = " ,mutation_rate)
257 mp = MAPElites(iterations, population_size, num_bins, bins, dimensions, feature_dimensions, crossover_rate, sigma,
258 mutation_rate, flag_crossover=cross, minimization=True)
259 resultado = mp.run()
```

Código 13.1: Implementación MAP-Elites con función Rastringin.

```
0 import math as math
1 import numpy as np
2 import operator as op
3 import datetime
4 import random
5 
6 class Differential_Evolution():
7 
8     def __init__(self,
9                 iterations,
10                population_size,
11                dimensions,
12                cross_probability,
13                F):
14         self.iterations = iterations
15         self.population_size = population_size
16         self.dimensions = dimensions
17         self.cross_probability = cross_probability
18         self.F = F
19         self.best_fitness = 1000
20         self.best_individual = 0
21 
22     def generate_random_solution(self):
23         new_population = [np.random.uniform(low= -5.12, high= 5.12) for d in range(self.dimensions)]
24         return new_population
25 
26     def generate_initial_population(self):
27         initial_population = []
28         for i in range(0,self.population_size):
29             random_solution = self.generate_random_solution()
30             initial_population.append(random_solution)
31         return np.array(initial_population)
32 
33     def mutation(self, population):
```

```
34         # Selección de individuo aleatorios
35         index = [idx for idx in range(population_size)]
36         random_index = np.random.choice(index, 3, replace=False)
37         target_vectors= population[random_index]
38         mutant = target_vectors[0] + self.F * (target_vectors[1] - target_vectors[2])
39         np.clip(mutant, -5.12, 5.12)
40         return mutant
41 
42     def recombination(self, noisy_vector, individuo):
43         # Cruce binomial
44         cossover_point = np.random.rand(dimensions) < cross_probability
45         trial_vector = np.where(cossover_point, noisy_vector, individuo)
46         return trial_vector
47 
48     def evaluate(self, individual):
49         # Función Rastringin
50         fitness = 10*self.dimensions
51         for i in range(len(individual)):
52             fitness += individual[i]**2 - (10*math.cos(2*math.pi*individual[i]))
53         return fitness
54 
55     def run(self):
56 
57         tiempo_inicial = datetime.datetime.now()
58         population = self.generate_initial_population()
59         while True:
60             for j in range(len(population)):
61                 noisy_vector = self.mutation(population)
62                 trial_vector = self.recombination(noisy_vector, population[j])
63                 if self.evaluate(population[j]) > self.evaluate(trial_vector):
64                     population[j] = trial_vector
65                 if self.best_fitness > self.evaluate(trial_vector):
66                     self.best_fitness = self.evaluate(trial_vector)
67                     self.best_individual = trial_vector
68                     print(self.best_fitness)
69             tiempo_transcurrido = datetime.datetime.now() - tiempo_inicial
70             if self.best_fitness == 0:
71                 break
72             # print(tiempo_transcurrido)
73             if tiempo_transcurrido.total_seconds() >= 100:
74                 break
75 
76         tiempo_transcurrido = datetime.datetime.now() - tiempo_inicial
77         print("Tiempo transcurrido:")
78         print(tiempo_transcurrido,"\n")
79         print("Mejor individuo")
80         print(self.best_individual,self.best_fitness)
81 
82 iterations = 3000
83 dimensions = 7
84 population_size = dimensions*10
85 cross_probability = 0.9
86 F = 0.8
87 
88 de = Differential_Evolution(iterations, population_size, dimensions, cross_probability, F)
89 de.run()
```

Código 13.2: Implementación de Evolución Diferencial con función Rastringin.

13.2. Simulador

```
0  import sys
1  import math
2  import time
3  import pygame
4  import random
5  import numpy as np
6  import pandas as pd
7  import seaborn as sns
8  import matplotlib.pyplot as plt
9  from Environment import Environment
10 from NeuralNetwork import JordanNetwork
11 from Configuration import Configuration
12 from MAPELitesSimulation import MAPElitesSimulation
13 from DifferentialSimulation import DifferentialSimulation
14
15
16 BLACK = (0, 0, 0)
17 RED = (255,0,0)
18 GREY = (206, 204, 202)
19 WHITE = (255, 255, 255)
20 GREEN = (0, 100, 0)
21 BLUE = (0, 0, 255)
22
23 cm_pixel = 0.3
24
25 # Constantes movimiento lineal
26 a = 0.0000175535946480465
27 b = 0.379742197503453
28 c = 2.12361958525124
29 a_positiva = 103.560231228394
30 t0 = 0.1
31
32 class Robot:
33     def __init__(self):
34         # Inicialización
35         self.current_x = 50
36         self.current_y = 450
37         self.sum_v_l = 0
38         self.sum_v_r = 0
39         self.v_current_r = 0
40         self.v_current_l = 0
41         self.obst_col = False
42         self.x_inicial = self.current_x
43         self.y_inicial = self.current_y
44
45         # Inicialización visual
46         self.x_visual = self.current_x
47         self.y_visual = self.current_y
48         self.alpha_visual = -math.pi/2
49
50         self.fitness = []
51         self.inputs = [0]*4
52         self.outputs = [0]*2
53         self.sum_vel = 0
54         self.buffer_v = [[0,0]] * int(0.1/config.dt)
55         self.buffer_sensor = [[0, 0, 0, 0]] * int(0.4/config.dt)
56
57         # Inicialización red neuronal
58         self.nn = JordanNetwork(4, 4, 2)
59
60     def modelo_vel(self, v_percentage):
61
62         if v_percentage == 0:
63             v = 0
64         else:
65             v = math.pow(v_percentage, 2)*a + v_percentage*b + c
66         return v
67
68     def aceleracion(self, v_objetivo, v):
69
70         a_negativa = v/0.208
71
72         if v < v_objetivo:
```

```
73              v += a_positiva*dt
74          elif v > v_objetivo:
75              v -= a_negativa*dt
76              if v < 3:
77                  v = 0
78          return v
79
80      def act(self, v_r, v_l):
81
82          # Cálculo de la nueva velocidad
83          v_array = [self.aceleracion(self.modelo_vel(v_r*100*0.6), self.v_current_r), self.aceleracion(self.modelo_vel(v_l*100*0.6),
    self.v_current_l)]
84          self.v = (v_array[1] + v_array[0])/2
85          self.prueba_array = v_array
86          inc_theta = (v_array[1] - v_array[0])/14.5
87
88          # Actualización posición robot
89          self.alpha_visual += inc_theta
90          self.current_x += self.v/cm_pixel*math.cos(-self.alpha_visual)*dt
91          self.current_y -= self.v/cm_pixel*math.sin(-self.alpha_visual)*dt
92
93          # Límites
94          if self.current_x < 0 : self.current_x = 0
95          if self.current_y < 0: self.current_y = 0
96          if self.current_x > config.width - 30 : self.current_x = config.width - 30
97          if self.current_y > config.height - 30: self.current_y = config.height - 30
98
99          self.v_current_r = v_array[0]
100         self.v_current_l = v_array[1]
101
102     def check_collision(self):
103         # Detección pared
104         if any(x > 0.9 for x in self.inputs):
105             self.obst_col = True
106
107         # Detección obstáculo
108         if pygame.sprite.spritecollide(robobo_sprite, obstacles_list, False):
109             self.obst_col = True
110
111     def reset(self):
112         # Inicialización
113         self.current_x = 50
114         self.current_y = 450
115         self.sum_v_l = 0
116         self.sum_v_r = 0
117         self.v_current_r = 0
118         self.v_current_l = 0
119         self.obst_col = False
120         self.x_inicial = self.current_x
121         self.y_inicial = self.current_y
122         self.x_final = 0
123         self.y_final = 0
124
125         # Inicialización visual
126         self.x_visual = self.current_x
127         self.y_visual = self.current_y
128         self.alpha_visual = -math.pi/2
129
130         self.sum_vel = 0
131         self.outputs = [0]*2
132         self.buffer_v = [[0,0]] * int(0.1/config.dt)
133         self.buffer_sensor = [[0, 0, 0, 0]] * int(0.4/config.dt)
134
135         # Inicialización red neuronal
136         self.nn = JordanNetwork(4, 4, 2)
137
138 class RoboboSprite(pygame.sprite.Sprite):
139     def __init__(self, robobo):
140         super().__init__()
141         self.robobo = robobo
142         self.image_orig = pygame.image.load("/home/sium/Dropbox/Universidad/TFG/scripts/simulador_3/robobo_small.png").convert_alpha()
143         self.image_orig = pygame.transform.scale(self.image_orig, (60, 60))
144         self.image = self.image_orig.copy()
145         self.rect = self.image.get_rect()
146         self.rot = -math.degrees(self.robobo.alpha_visual)
147
148     def update(self):
149         self.rot = -math.degrees(self.robobo.alpha_visual)
150         new_image = pygame.transform.rotozoom(self.image_orig, self.rot, 1.0)
```

```
151
152         self.image = new_image
153         self.rect = self.image.get_rect()
154         self.rect.center = (self.robobo.x_visual, self.robobo.y_visual)
155
156 class Obstacle(pygame.sprite.Sprite):
157     def __init__(self, color, width, height, x_coordenate, y_coordenate, move_obstacles):
158         super().__init__()
159
160         self.image = pygame.Surface([width, height])
161         self.image.fill(WHITE)
162         self.image.set_colorkey(WHITE)
163         self.name = "circle"
164         self.color = color
165         self.radius = width//2
166         self.center = [self.radius, self.radius]
167         self.rect = self.image.get_rect()
168         pygame.draw.circle(self.image, self.color, self.center, self.radius)
169         self.rect.center = (x_coordenate, y_coordenate)
170         self.move_obstacles = move_obstacles
171         self.x_origin = x_coordenate
172         self.y_origin = y_coordenate
173         self.angle = random.uniform(0, math.pi)
174         self.paso = math.pi/200
175
176     def update(self):
177         if self.move_obstacles:
178             x_coordenate = self.x_origin + np.cos(self.angle)*40
179             y_coordenate = self.y_origin + np.sin(self.angle*2)*40
180
181             self.angle += self.paso
182             self.rect.center = (x_coordenate, y_coordenate)
183
184 class circle(pygame.sprite.Sprite):
185     def __init__(self, color, width, height, obstacle, alpha):
186         super().__init__()
187
188         self.image = pygame.Surface([width, height])
189         self.image.fill(WHITE)
190         self.image.set_colorkey(WHITE)
191         self.image.set_alpha(alpha)
192         self.name = "circle"
193         self.color = color
194         self.radius = width//2
195         self.center = [self.radius, self.radius]
196         self.rect = self.image.get_rect()
197         pygame.draw.circle(self.image, self.color, self.center, self.radius)
198         self.obstacle = obstacle
199         self.rect.center = self.obstacle.rect.center
200
201     def update(self):
202         self.rect.center = self.obstacle.rect.center
203
204 class rectangle(pygame.sprite.Sprite):
205     def __init__(self, color, alpha, size):
206         super().__init__()
207
208         self.image = pygame.Surface([config.width, config.height])
209         self.image.fill(WHITE)
210         self.image.set_colorkey(WHITE)
211         self.image.set_alpha(alpha)
212         self.name = "rectangle"
213         self.color = color
214         self.rect = self.image.get_rect()
215         pygame.draw.rect(self.image, self.color, (0, 0, size, config.height),0)
216         pygame.draw.rect(self.image, self.color, (0, 0, config.width, size),0)
217         pygame.draw.rect(self.image, self.color, (config.width - size, 0, size, config.height),0)
218         pygame.draw.rect(self.image, self.color, (0, config.height - size, config.width, size),0)
219
220
221 config = Configuration()
222 dt = config.dt
223
224 # Configuración pantalla
225 display = pygame.display.set_mode((config.width, config.height))
226 pygame.display.set_caption("Simulation")
227 pygame.font.init()
228 font = pygame.font.SysFont('arial', 20)
229 clock = pygame.time.Clock()
```

```
230 pygame.init()
231
232 # Background
233 bg = pygame.image.load("/home/sium/Dropbox/Universidad/TFG/scripts/simulador_3/bg.png")
234
235 # Inicialización sprites
236 obstacle_1 = Obstacle(GREEN, 30, 30, 350, 350, config.move_obstacles)
237 obstacle_2 = Obstacle(GREEN, 30, 30, 150, 350, config.move_obstacles)
238 obstacle_3 = Obstacle(GREEN, 30, 30, 350, 150, config.move_obstacles)
239 obstacle_4 = Obstacle(GREEN, 30, 30, 150, 150, config.move_obstacles)
240
241 # Zona frenado
242 zone_1 = circle(RED, 115, 115, obstacle_1, 50)
243 zone_2 = circle(RED, 115, 115, obstacle_2, 50)
244 zone_3 = circle(RED, 115, 115, obstacle_3, 50)
245 zone_4 = circle(RED, 115, 115, obstacle_4, 50)
246
247 zone_5 = circle(RED, 80, 80, obstacle_1, 125)
248 zone_6 = circle(RED, 80, 80, obstacle_2, 125)
249 zone_7 = circle(RED, 80, 80, obstacle_3, 125)
250 zone_8 = circle(RED, 80, 80, obstacle_4, 125)
251
252 rect_zone_1 = rectangle(RED, 50, 50)
253 rect_zone_2 = rectangle(RED, 125, 25)
254
255 obstacles_list = pygame.sprite.Group()
256 obstacles_list.add(obstacle_1)
257 obstacles_list.add(obstacle_2)
258 obstacles_list.add(obstacle_3)
259 obstacles_list.add(obstacle_4)
260
261 zone_list = pygame.sprite.Group()
262
263 zone_list.add(zone_1)
264 zone_list.add(zone_2)
265 zone_list.add(zone_3)
266 zone_list.add(zone_4)
267 zone_list.add(zone_5)
268 zone_list.add(zone_6)
269 zone_list.add(zone_7)
270 zone_list.add(zone_8)
271 zone_list.add(rect_zone_1)
272 zone_list.add(rect_zone_2)
273
274 robot = Robot()
275 robobo_sprite = RoboboSprite(robot)
276
277 sprites_list = pygame.sprite.Group()
278 sprites_list.add(robobo_sprite)
279
280 Environment = Environment(robot, config, obstacles_list)
281 algorithm = config.algorithm
282 if algorithm == 0:
283     sim = MAPElitesSimulation(config, robot, Environment)
284 elif algorithm == 1:
285     sim = DifferentialSimulation(config, robot, Environment)
286
287 running = True
288
289 t_1 = int(round(time.time() * 1000))
290
291 while running:
292
293     for event in pygame.event.get():
294         if event.type == pygame.QUIT:
295             running = False
296
297     # Evolución
298     sim.main()
299
300     sprites_list.update()
301     obstacles_list.update()
302     zone_list.update()
303
304     if sim.finished:
305         running = False
306
307     if sim.visualize:
308         display.fill(WHITE)
```

```
309         display.blit(bg, (0, 0))
310
311         # Representar elementos del entorno
312         zone_list.draw(display)
313         for cord in range(len(sim.environment.trail)):
314             pygame.draw.circle(display, BLUE, sim.environment.trail[cord], 4)
315         sprites_list.draw(display)
316         obstacles_list.draw(display)
317
318         iteration_str = 'Iteration: ' + str(sim.iteration + 1) + ' ' + 'Velocidad: ' + str(np.round(robot.v, 2))
319         text = font.render(iteration_str, True, WHITE)
320         display.blit(text, [10, 10])
321         pygame.display.flip()
322         clock.tick(config.fps)
323 pygame.quit()
324
325 print(int(round(time.time() * 1000))- t_1)
326
327 sns.set()
328 sns.set_style("whitegrid")
329 x = np.arange(len(sim.mean_fitness))
330
331 if algorithm == 1:
332     plt.fill_between(x, np.array(sim.mean_fitness) - np.array(sim.deviation_fitness), np.array(sim.mean_fitness) + np.array(sim.deviation_fitne
333     plt.plot(x, sim.mean_fitness, '-b', label="calidad media", markevery = 500, marker='o')
334     plt.plot(x, sim.max_fitness, '-r', label="calidad máxima", markevery = 500, marker='s')
335     plt.legend(loc="lower right")
336     plt.ylabel('Calidad')
337     plt.xlabel('Evaluaciones')
338     plt.xlim(0, len(x))
339     plt.ylim(0,1)
340     plt.savefig('calidad_mean_max')
341     plt.clf()
342
343     for i in range(sim.fitnes_matrix.shape[0]):
344         plt.plot(sim.fitnes_matrix[i][:], '-.')
345     plt.ylabel('Calidad')
346     plt.xlabel('Generaciones')
347     plt.xlim(0, config.iterations - 1)
348     plt.ylim(0,1)
349     plt.savefig('calidad_pob')
350     plt.clf()
351
352     for i in range(sim.fitnes_matrix.shape[0]):
353         plt.plot(sim.gen_matrix[i][:], '-.')
354     plt.ylabel('Gen Medio')
355     plt.xlabel('Generaciones')
356     plt.xlim(0, config.iterations - 1)
357     plt.ylim(0,1)
358     plt.savefig('gen_mean')
359     plt.clf()
360
361 else:
362
363     plt.fill_between(x, np.array(sim.mean_fitness) - np.array(sim.deviation_fitness), np.array(sim.mean_fitness) + np.array(sim.deviation_fitne
364     plt.plot(x, sim.mean_fitness, '-b', label="calidad media", markevery = 500, marker='o')
365     plt.plot(x, sim.max_fitness, '-r', label="calidad máxima", markevery = 500, marker='s')
366     plt.legend(loc="lower right")
367     plt.ylabel('Calidad')
368     plt.xlabel('Evaluaciones')
369     plt.xlim(0,config.iterations - 1)
370     plt.ylim(0,1)
371     plt.savefig('map_calidad')
```

Código 13.3: Programa principal y visualizador del simulador.

```
0  import numpy as np
1
2  def sigmoid(x):
3      return 1 / (1 + np.exp(-x))
4
5  def tanh(x):
6      return np.tanh(x)
7
8  def ReLU(x):
9      return x * (x > 0)
10
11 def softmax(x):
12     e = np.exp(x - np.max(x))  # prevent overflow
13     if e.ndim == 1:
14         return e / np.sum(e, axis=0)
15     else:
16         return e / np.array([np.sum(e, axis=1)]).T  # ndim = 2
17
18 # -----------------------------------------------------------------------------
19 # Jordan recurrent network
20 # Copyright (C) 2011  Nicolas P. Rougier
21 #
22 # Distributed under the terms of the BSD License.
23 # -----------------------------------------------------------------------------
24 # This is an implementation of the multi-layer perceptron with retropropagation
25 # learning.
26 # -----------------------------------------------------------------------------
27 class JordanNetwork:
28
29     def __init__(self, *args):
30
31         self.shape = args
32         n = len(args)
33
34         self.size = 0
35         # Build layers
36         self.layers = []
37         # Input layer (+1 unit for bias
38         #              +size of oputput layer)
39         self.layers.append(np.ones(self.shape[0]+1+self.shape[-1]))
40         # Hidden layer(s) + output layer
41         for i in range(1,n):
42             self.layers.append(np.ones(self.shape[i]))
43         # Build weights matrix (randomly between -0.25 and +0.25)
44         self.weights = []
45         for i in range(n-1):
46             self.weights.append(np.zeros((self.layers[i].size,
47                                           self.layers[i+1].size)))
48             self.size += self.weights[i].size
49         # Reset weights
50         self.reset()
51
52     def reset(self):
53         ''' Reset weights '''
54
55         for i in range(len(self.weights)):
56             Z = np.random.rand( len(self.layers[i]), len(self.layers[i+1]) )
57             self.weights[i][:] = (2*Z-1)*0.25
58
59     # change weights from genes [0,1] -> [-5,5]
60     def update_weights(self, weights):
61
62         if len(weights) != self.size:
63             raise ValueError("Wrong number of parameters")
64         start_index = 0
65         end_index = 0
66         for i in range(len(self.weights)):
67             end_index = start_index + self.weights[i].size
68             self.weights[i] = ((weights[start_index:end_index].reshape(self.weights[i].shape))*2.0 - 1.0)*5.0
69             start_index = end_index
70
71     def propagate_forward(self, data):
72         ''' Propagate data from input layer to output layer. '''
73
74         # Set input layer with data
75         self.layers[0][0:self.shape[0]] = data
76         # and output layer
77         self.layers[0][self.shape[0]:-1] = self.layers[-1]
```

```
        
        # Propagate from layer 0 to layer n-1 using sigmoid as activation function
        for i in range(1,len(self.shape)):
            # Propagate activity
            self.layers[i][...] = sigmoid(np.dot(self.layers[i-1],self.weights[i-1]))

        # Return output
        return self.layers[-1]
```

Código 13.4: Código de la red neuronal recurrente tipo Jordan.

```
import diff
import map_elites
import numpy as np
from map_elites import MAPElites

class MAPElitesSimulation:
    def __init__(self, config, robot, environment):
        self.index = 0
        self.visual_index = 0
        self.steps = 0
        self.iteration = 0
        self.map = False
        self.finished = False
        self.visualize = False
        self.robot = robot
        self.config = config
        self.environment = environment
        self.dimensions = self.robot.nn.size

        self.max_fitness = []
        self.mean_fitness = []
        self.deviation_fitness = []

        self.robot.weights = diff.generate_initial_population(self.config.low, self.config.high, self.dimensions, self.config.population_size)
        self.current_weights = self.robot.weights[0]

        bins = [np.arange(self.config.lowerlimit, self.config.upperlimit, self.config.granulation)]*2
        num_bins = len(bins[0])
        self.mp = MAPElites(self.config.population_size, num_bins, bins, self.dimensions, self.config.feature_dimensions, self.config.crossove
        self.config.mutation_rate, flag_crossover=self.config.cross, minimization=True)

    def evolution(self):
        if self.mp.flag_crossover == True:
            individual = self.mp.selection(numero_individuos = 2)
            ind = self.mp.crossover(individual)[0]
            self.robot.current_weights = self.mp.mutation(ind)
        else:
            individual = self.mp.selection(numero_individuos = 1)[0]
            self.robot.current_weights = self.mp.mutation(individual)

        self.robot.nn.update_weights(self.robot.current_weights)

    def evaluation(self):
        self.mp.mapeo(self.robot.current_weights, self.feature_space(), self.calculate_fitness())
        self.robot.fitness = self.mp.performances

        if self.map:
            # En el caso de MAP-Elites es necesario quitar los np.inf que cubren la matriz
            self.mean_fitness.append(np.mean(self.robot.fitness[self.robot.fitness != np.inf]))
            self.deviation_fitness.append(np.std(self.robot.fitness[self.robot.fitness != np.inf]))
            self.max_fitness.append(np.amax(self.robot.fitness[self.robot.fitness != np.inf]))

    def calculate_fitness(self):

        v_mean = (self.robot.sum_vel/self.steps)
        v_diff = 1 - np.sqrt(np.absolute(self.robot.sum_v_l/(self.steps + 1) - self.robot.sum_v_r/(self.steps + 1)))
        fitness = v_mean * v_diff * self.steps/self.config.evaluation_steps

        return fitness

    def feature_space(self):
        features = [self.robot.x_final, self.robot.y_final]
```

```
62          return features
63
64      def select_best(self):
65          # Selección de mejores individuos para su representación
66          self.best_index = np.argwhere((self.robot.fitness > 0.8*self.max_fitness[-1]) & (self.robot.fitness != np.inf))
67
68          # Selección de la mejor solución
69          self.best_individual_index = np.argwhere(self.robot.fitness == self.max_fitness[-1])
70          np.save("best_ind.npy", self.mp.solutions[self.best_individual_index[0][0]][self.best_individual_index[0][1]][0])
71
72      def main(self):
73          if self.steps == 0:
74              if self.iteration < self.config.iterations and self.map:
75                  self.evolution()
76              else:
77                  self.robot.current_weights = self.robot.weights[self.index]
78                  self.robot.nn.update_weights(self.robot.current_weights)
79
80              if self.visualize:
81                  self.robot.nn.update_weights(self.mp.solutions[self.best_index[self.visual_index][0]][self.best_index[self.visual_index][1]][0])
82
83          self.environment.simulate_step(self.steps)
84
85          self.steps += 1
86
87          if self.steps == self.config.evaluation_steps or self.robot.obst_col:
88              self.robot.x_final = self.robot.current_x
89              self.robot.y_final =  self.robot.current_y
90              self.environment.trail = []
91
92              if self.visualize == False:
93                  self.evaluation()
94              else:
95                  self.visual_index += 1
96
97                  if self.visual_index == len(self.best_index):
98                      self.finished = True
99
100             self.steps = 0
101             self.environment.t = 0
102             self.robot.reset()
103
104             if self.map == False:
105                 self.index  += 1
106             else:
107                 self.iteration += 1
108                 print("Iteración:" + "" +   str(self.iteration) + '/' + str(self.config.iterations))
109
110             # Recorrer población inicial
111             if self.index == self.config.population_size:
112                 self.map = True
113                 self.index = 0
114
115             # Condición final evolución
116             if self.iteration == self.config.iterations:
117                 self.select_best()
118                 self.mp.ploteo()
119                 self.visualize = True
```

Código 13.5: Código del proceso evolutivo para MAP-Elites.

```
0  import math as math
1  import numpy as np
2  import operator as op
3  import logging as logger
4  import seaborn as sns
5  import pandas as pd
6  from mpl_toolkits.mplot3d import Axes3D
7  import datetime
8
9  from matplotlib import pyplot as plt
10 from pandas import DataFrame
11
```

```
12 class MAPElites():
13
14     def __init__(self,
15                  population_size,
16                  num_bins,
17                  bins,
18                  dimensions,
19                  feature_dimensions,
20                  crossover_rate,
21                  sigma,
22                  mutation_rate,
23                  flag_crossover,
24                  minimization):
25
26
27         """
28         Parámetros:
29         - iterations: Número de iteraciones con la que controlar el fin del bucle
30         - population_size: Población inicial
31         - dimensions: Dimensiones del problema
32         - feature_dimensions: Dimensiones del espacio de caracerísticas
33         - crossover_rate: Ratio de cruce.
34         - flag_crossover: Flag para activar el cruce
35         - mutation_rate : Ratio de mutación
36         - minimization : True = minimizar False = maximizar
37         - num_bins: Número de celdas del mapa
38         - bins: Vector que contiene los intervalos de las celdas
39         """
40
41         self.population_size = population_size
42         self.dimensions = dimensions
43         self.feature_dimensions = feature_dimensions
44         self.crossover_rate = crossover_rate
45         self.flag_crossover = flag_crossover
46         self.mutation_rate = mutation_rate
47         self.minimization = minimization
48         self.num_bins = num_bins
49         self.bins = bins
50         self.best_individual = 0
51         self.replace = 0
52         self.seed = 30
53         self.sigma = sigma
54         self.min_max_global = 0
55         self.tiempo_inicial = datetime.datetime.now()
56
57         if self.minimization:
58             self.comp = op.lt
59         else:
60             self.comp = op.gt
61
62         """
63         Matriz de N-Dimensiones, donde N es el número de dimensiones del espacio de atributos,
64         que contendrá el mapa de celdas. Cada celda tendrá un objeto de la forma:
65         list(np.array([x0,x1,...xn], perf)) , siendo n el número de variables del problema.
66         Sea crea otra matriz de las misma dimensiones con la que facilitar la representación gráfica.
67         """
68
69         self.dimensiones_matriz = (self.num_bins,)*(self.feature_dimensions)
70         self.performances = np.full(self.dimensiones_matriz,np.inf)
71         self.solutions = np.full(self.dimensiones_matriz,np.inf,dtype = object)
72         self.solutions.fill([np.inf,np.inf])
73
74     def mapeo(self, individual, features, fitness):
75         # Mapeo de cada individuo en la celda correspondiente
76         index = self.gentospace(individual, features)
77         celda = self.solutions[index]
78         if fitness > celda[1] or celda[1] == np.inf:
79             self.solutions[index] = [individual, fitness]
80             self.performances[index] = fitness
81             self.replace += 1
82         if self.comp(self.performances[index], self.min_max_global) == False:
83             self.min_max_global = self.performances[index]
84
85     def gentospace(self, individual, features):
86         # Devuelve el índice del individuo en cada una de los atributos
87         index = tuple()
88         for i in range(self.feature_dimensions):
89             b = np.digitize(features[i],self.bins[i],right = False)
90             index = index + (b-1,)
```

```
        return index

    def performance(self, individual):
        # Cálculo de la performance. Se utilizar como benchmark la función Rastrigin de n-dimensiones.unif
        fitness = 10*self.dimensions
        for i in range(len(individual)):
            fitness += individual[i]**2 - (10*math.cos(2*math.pi*individual[i]))
        return fitness

    def selection(self, numero_individuos):
        # Búsqueda de individuo(s) en las celdas ocupadas.
        # El candidato tendrá la siguiente forma [np.array[x0,x1...],....np.array[x0,x1]]
        condicion_busqueda = False
        individuo_aleatorio = tuple()
        for i in range(numero_individuos):
            while condicion_busqueda == False:
                candidate_index = np.random.randint(self.num_bins, size=self.feature_dimensions)
                candidate = self.solutions[tuple(candidate_index)][0]
                if self.solutions[tuple(candidate_index)][1] != np.inf:
                    condicion_busqueda = True
            individuo_aleatorio = individuo_aleatorio + (candidate,)
        return individuo_aleatorio

    def mutation(self, individual):
        # Se utiliza la mutación gaussiana para añadir una pequeña mutación
        mu = 0
        if np.random.random() < self.mutation_rate:
            variation = np.random.normal(mu, self.sigma, self.dimensions)
            new_individual = individual + variation
            new_individual = np.clip(new_individual,0,1)
        else:
            new_individual = individual
        return new_individual

    def crossover(self, individual):
        # El cruce de individuos se realiza a través del cruce uniforme
        individual1 = individual[0]
        individual2 = individual[1]
        for i in range(len(individual)):
            if np.random.random() < crossover_rate:
                individual1[i], individual2[i] = individual2[i], individual1[i]
        return individual1, individual2

    def ploteo(self):

        # Redimensionamiento de los datos según feature dimensions.
        if self.feature_dimensions == 1:
            df = pd.DataFrame(self.performances)
            df.replace(np.inf, np.nan, inplace = True)
        if self.feature_dimensions == 2:
            df = pd.DataFrame(self.performances)
            df.replace(np.inf, np.nan, inplace = True)
        if self.feature_dimensions == 3:
            data = self.performances.reshape(self.dimensiones_matriz[0]*self.dimensiones_matriz[2],self.dimensiones_matriz[1])
            df = pd.DataFrame(data)
            df.replace(np.inf, np.nan, inplace = True)
        if self.feature_dimensions == 4:
            data = self.performances.transpose(0,2,1,3).reshape(self.num_bins**2,self.num_bins**2)
            data_2 = self.solutions.transpose(0, 2, 1, 3).reshape(self.num_bins ** 2, self.num_bins ** 2)
            df = pd.DataFrame(data)
            df.replace(np.inf, np.nan, inplace = True)
        if self.feature_dimensions == 5:
            data = self.performances.transpose(0,1,3,2,4).reshape(self.num_bins**3,self.num_bins**2)
            df = pd.DataFrame(data)
            df.replace(np.inf, np.nan, inplace = True)
        if self.feature_dimensions == 6:
            data = self.performances.transpose(0,2,4,1,3,5).reshape(self.num_bins**3,self.num_bins**3)
            df = pd.DataFrame(data)
            df.replace(np.inf, np.nan, inplace = True)

        sns.heatmap(df, yticklabels = False, xticklabels = False, square = False, cbar_kws={'label': 'Calidad'})
        plt.xlabel("Característica 2")
        plt.ylabel("Característica 1")
        plt.savefig('ejemplo_map_elites.png')
        plt.clf()
```

Código 13.6: Código de los operadores evolutivos de MAP-Elites.

```
import diff
import numpy as np

class DifferentialSimulation:
    def __init__(self, config, robot, environment):
        self.index = 0
        self.steps = 0
        self.iteration = 0
        self.finished = False
        self.visualize = False
        self.robot = robot
        self.config = config
        self.environment = environment
        self.dimensions = self.robot.nn.size
        self.robot.weights = diff.generate_initial_population(config.low, config.high, self.dimensions, config.population_size)
        self.current_weights = self.robot.weights[0]

        self.max_fitness = []
        self.mean_fitness = []
        self.deviation_fitness = []
        self.fitnes_matrix = np.full((self.config.population_size, self.config.iterations),np.inf)
        self.gen_matrix = np.full((self.config.population_size, self.config.iterations),np.inf)

    def evolution(self):

        # Mutación
        noisy_vector = diff.mutation(0, 1, self.robot.weights, self.config.population_size, self.config.F)
        self.robot.current_weights = diff.recombination(noisy_vector, self.robot.weights[self.index], self.dimensions, self.config.cross_prob

        # Actualización pesos de la red
        self.robot.nn.update_weights(self.robot.current_weights)

    def calculate_fitness(self):

        v_mean = (self.robot.sum_vel/self.steps)
        v_diff = 1 - np.sqrt(np.absolute(self.robot.sum_v_l/(self.steps + 1) - self.robot.sum_v_r/(self.steps + 1)))
        fitness = v_mean * v_diff * self.steps/self.config.evaluation_steps

        return fitness

    def evaluation(self):

        # Evaluación
        if self.iteration > 0:
            # Maximización
            if self.calculate_fitness() > self.robot.fitness[self.index]:
                self.robot.fitness[self.index] = self.calculate_fitness()
                self.robot.weights[self.index] = self.robot.current_weights
        else:
            self.robot.fitness.append(self.calculate_fitness())
            self.robot.weights[self.index] = self.robot.current_weights

        #print(self.calculate_fitness())

        if self.iteration < self.config.iterations:
            self.fitnes_matrix[self.index][self.iteration] = self.robot.fitness[self.index]
            self.gen_matrix[self.index][self.iteration] = sum(self.robot.weights[self.index])/self.dimensions
            self.mean_fitness.append(np.mean(self.robot.fitness))
            self.max_fitness.append(max(self.robot.fitness))
            self.deviation_fitness.append(np.std(self.robot.fitness))

    def select_best(self):
        # Selección de la mejor solución
        self.best_individual_index = np.argwhere(self.robot.fitness == self.max_fitness[-1])
        np.save("best_ind.npy", self.robot.weights[self.best_individual_index[0][0]])

    def main(self):
        if self.steps == 0:
            if self.iteration < self.config.iterations:
                self.evolution()
            else:
                self.robot.nn.update_weights(self.robot.weights[self.index])

        self.environment.simulate_step(self.steps)
        self.steps += 1

        if self.steps == self.config.evaluation_steps or self.robot.obst_col:
            self.robot.x_final = self.robot.current_x
```

```
78            self.robot.y_final =  self.robot.current_y
79            self.environment.trail = []
80
81            if self.visualize == False:
82                self.evaluation()
83
84            self.steps = 0
85            self.environment.t = 0
86            self.robot.reset()
87
88            self.index += 1
89
90            # Recorrer miembros población
91            if self.index == self.config.population_size:
92                self.iteration += 1
93                self.index = 0
94                print("Iteración:" + "" +   str(self.iteration) + '/' + str(self.config.iterations))
95
96            # Condición final evolución
97            if self.iteration == self.config.iterations:
98                self.select_best()
99                self.visualize = True
100
101           # Condición final evolución
102           if self.iteration == self.config.iterations + 1:
103               self.finished = True
```

Código 13.7: Código del proceso evolutivo para la Evolución Diferencial.

```
0 import math
1 import numpy as np
2 import operator as op
3
4
5 def generate_random_solution(low_limit, high_limit, dimensions):
6     new_population = np.random.uniform(low=low_limit, high=high_limit, size = dimensions)
7     return new_population
8
9 def generate_initial_population(low_limit, high_limit, dimensions, population_size):
10    initial_population = []
11    for i in range(0, population_size):
12        random_solution = generate_random_solution(
13            low_limit, high_limit, dimensions)
14        initial_population.append(random_solution)
15    return np.array(initial_population)
16
17 def mutation(low_limit, high_limit, population, population_size, F):
18    # Selección de individuo aleatorios
19    index = [idx for idx in range(population_size)]
20    random_index = np.random.choice(index, 3, replace=False)
21    target_vectors = population[random_index]
22    mutant = target_vectors[0] + F * \
23        (target_vectors[1] - target_vectors[2])
24    mutant = np.clip(mutant, low_limit, high_limit)
25    return mutant
26
27 def recombination(noisy_vector, individuo, dimensions, cross_probability):
28    cossover_point = np.random.rand(dimensions) < cross_probability
29    trial_vector = np.where(cossover_point, noisy_vector, individuo)
30    return trial_vector
```

Código 13.8: Código de los operadores evolutivos de Evolución Diferencial.

```
0  import math
1  import numpy as np
2
3  TWOPI = 2*np.pi
4
5  def normalize_angle(angle):
6      # reduce the angle
7
8      angle = angle %TWOPI
9      # en python %es modulo entonces 0 <= angle < 360
10
11     # force into the minimum absolute value residue class, so that -180 < angle <= 180
12     if angle > np.pi:
13         angle -= TWOPI
14     return angle
15
16 class Environment:
17     def __init__(self, robot, config, obstacles):
18         self.t = 0
19         self.dt = config.dt
20         self.robot = robot
21
22         # parametros para sensor rendija
23         self.alpha_max = math.radians(25)
24         self.tam_rendija = 4
25         self.H0 = 1
26         self.w0 = self.tam_rendija
27         self.width = config.width
28         self.height = config.height
29         self.obstacles = obstacles
30         self.radio_robobo = 30
31         self.trail = []
32
33     def simulate_step(self, step):
34
35         # Actualizar sensores robo
36         self.actualizar_sensores_de_robot()
37         self.robot.buffer_sensor.append(self.robot.inputs)
38         normal_input = np.array(self.robot.buffer_sensor[step])
39
40         #Red neuronal (Entrada: Rendija, Salida: Velocidad ruedas)
41         self.robot.outputs = self.robot.nn.propagate_forward(normal_input)
42
43         self.robot.buffer_v.append([self.robot.outputs[0] , self.robot.outputs[1]])
44
45         self.robot.sum_vel += (self.robot.outputs[0] + self.robot.outputs[1])/2
46         self.robot.sum_v_l += self.robot.outputs[1]
47         self.robot.sum_v_r += self.robot.outputs[0]
48
49         # Actuadores
50         self.robot.act(self.robot.buffer_v[step][0],  self.robot.buffer_v[step][1])
51         self.robot.check_collision()
52
53         # Actualizar copia gráfica
54         self.update_visual()
55
56         self.t += self.dt
57
58     def update_visual(self):
59         self.robot.x_visual = self.robot.current_x
60         self.robot.y_visual = self.robot.current_y
61         self.trail.append((int(self.robot.x_visual),int(self.robot.y_visual)))
62
63     def actualizar_sensores_de_robot(self):
64
65         x_j = self.robot.current_x
66         y_j = self.robot.current_y
67         alpha_j = self.robot.alpha_visual
68         self.D0 = self.radio_robobo
69         self.D0_squared = self.D0*self.D0
70
71         array_x_w = []
72
73         for obstacle in self.obstacles:
74             x_i = obstacle.rect.center[0]
75             y_i = obstacle.rect.center[1]
76             alfa_ij = math.atan2(y_i - y_j, x_i - x_j)
77             alfa_rij = normalize_angle(alfa_ij - alpha_j)
```

```
78              if abs(alfa_rij) <= self.alpha_max:
79                  x_ij = (alfa_rij + self.alpha_max)/(2*self.alpha_max)
80                  D_ij = (x_i - x_j)*(x_i - x_j) + (y_i - y_j)*(y_i - y_j)
81                  w_ij = self.w0 * self.D0_squared/D_ij
82                  array_x_w.append((x_ij, w_ij))
83
84          alpha_ray_izq = normalize_angle(alpha_j - self.alpha_max)
85          alpha_ray_der = normalize_angle(alpha_j + self.alpha_max)
86
87          di_j = self.corte_pared_distancia(x_j, y_j, alpha_ray_izq)
88          dd_j = self.corte_pared_distancia(x_j, y_j, alpha_ray_der)
89          #print(di_j, dd_j)
90
91          HI_j = self.H0*self.D0*2 / di_j
92          HD_j = self.H0*self.D0*2 / dd_j
93          #print(HI_j,HD_j)
94
95          self.actualizar_rendija(self.robot, HI_j, HD_j, array_x_w)
96
97      def detectar_corte_pared(self, x, y, robot, alpha, alpha_max):
98
99          alpha1 = normalize_angle(alpha + alpha_max)
100         dist_corte1 = self.corte_pared_distancia(x, y, alpha1)
101         if dist_corte1 > 0 and dist_corte1 < self.robot.distance_closest:
102             self.robor.distance_closest = dist_corte1
103             self.robor.angle_closest = alpha1
104
105         alpha2 = normalize_angle(alpha - alpha_max)
106         dist_corte2 = self.corte_pared_distancia(x, y, alpha2)
107         if dist_corte2 > 0 and dist_corte2 < self.robor.distance_closest:
108             self.robor.distance_closest = dist_corte2
109             self.robor.angle_closest = alpha2
110
111     def actualizar_rendija(self, robot, HI_j, HD_j, array_xsws):
112
113         # de i 0 a 4 (rendija)
114         for i in range(self.tam_rendija):
115             v = 0
116             for j in range(len(array_xsws)):
117                 (x,w) = array_xsws[j]
118                 v += max(0,min(1,self.tam_rendija*(0.5/self.tam_rendija - (abs(x-(i+0.5)/self.tam_rendija) - w/2))))
119             if v > 0.001:
120                 v = max(-v, -1)
121             else:
122                 v = HI_j + i * (HD_j - HI_j) / (self.tam_rendija-1)
123                 if v > 1:
124                     v = 1.0
125             self.robot.inputs[i] = v
126
127             # alpha_ray en rad
128     def corte_pared_distancia(self, x_i, y_i, alpha_ray):
129
130         xc = math.inf
131         yc = math.inf
132         dc = math.inf
133
134         cos_alpha = math.cos(alpha_ray)
135         sin_alpha = math.sin(alpha_ray)
136
137         #print("Corte pared x", x_i, ", y", y_i, ", alpha_ray ( )", math.degrees(alpha_ray))
138         (dN, xuN, yuN) = intersection_metodo_nuevo(x_i, y_i, alpha_ray, cos_alpha, sin_alpha, 0, 0, self.width, 0)
139         if dN > 0 and dN < dc:
140             xc = xuN
141             yc = yuN
142             dc = dN
143         #print("Norte xc, yc, dc", xuN , yuN, dN)
144         (dS, xuS, yuS) = intersection_metodo_nuevo(x_i, y_i, alpha_ray, cos_alpha, sin_alpha, 0, self.height, self.width, self.height)
145         if dS > 0 and dS < dc:
146             xc = xuS
147             yc = yuS
148             dc = dS
149         #print("Sur xc, yc, dc", xuS, yuS, dS)
150         (dE, xuE, yuE) = intersection_metodo_nuevo(x_i, y_i, alpha_ray, cos_alpha, sin_alpha, 0, 0, 0, self.height)
151         if dE > 0 and dE < dc:
152             xc = xuE
153             yc = yuE
154             dc = dE
155         #print("Este xc, yc, dc", xuE, yuE, dE)
156         (dO, xuO, yuO) = intersection_metodo_nuevo(x_i, y_i, alpha_ray, cos_alpha, sin_alpha, self.width, 0, self.width, self.height)
```

```
157         if dO > 0 and dO < dc:
158             xc = xuO
159             yc = yuO
160             dc = dO
161         #print("Oeste xc, yc, dc", xuO, yuO, dO)
162         #print("x corte, y corte, d", xc, yc, dc)
163         return dc
164 
165     def crear_array_contodo(self, HI_j, HD_j, array_xsws):
166 
167         array = [0] * self.tam_rendija
168         for i in range(self.tam_rendija):
169             v = 0
170             for ix in range(len(array_xsws)):
171                 x = array_xsws[ix][0]
172                 w = array_xsws[ix][1]
173                 v += max(0,min(1,1 - (abs(x-i) - w/2)))
174                 #print(1 - (abs(x-i) - w/2), abs(x-i))
175                 #print(x, w, v)
176             if v > 0.001:
177                 v = max(-v, -1)
178             else:
179                 v = HI_j + i * (HD_j - HI_j) / (self.tam_rendija-1)
180             array[i] = v
181 
182         return array
183 
184     def combinar_pared_vecinos(self, arraypared, arrayvecinos):
185         array = arraypared
186         for i in range(len(arraypared)):
187             if arrayvecinos[i] > 0.001:
188                 array[i] = -arrayvecinos[i]
189                 # cortar por -1
190                 if array[i] < -1:
191                     array[i] = -1
192             else:
193                 array[i] = arraypared[i]
194                 # cortar pr 1
195                 if array[i] > 1:
196                     array[i] = 1
197         return array
198 
199 def intersection_metodo_nuevo(x1, y1, alpha, cos_alpha, sin_alpha, x2a, y2a, x2b, y2b):
200 
201     angulo_recta2 = math.atan2(y2b - y2a, x2b - x2a)
202     dxa = cos_alpha
203     dya = sin_alpha
204 
205     dxb = math.cos(angulo_recta2)
206     dyb = math.sin(angulo_recta2)
207 
208     divisor = dxa*dyb - dxb*dya
209     if divisor != 0:
210         t = (dyb*(x2a-x1)-dxb*(y2a-y1))/divisor
211         vx = dxa*t
212         vy = dya*t
213 
214         # si d esta hacia atras signo negativo
215         d = math.copysign(math.hypot(vx, vy), vx*cos_alpha + vy*sin_alpha)
216         return (d, x1 + vx, y1 + vy)
217     else:
218         return (math.inf, math.nan, math.nan)
```

Código 13.9: Código del entorno de simulación.

```
0  class Configuration():
1      def __init__(self):
2          # Parámetros Simulador
3          # Original 500 x 500
4          self.width = 500
5          self.height = 500
6          self.fps = 60
7          self.dt = 1/self.fps
8          self.evaluation_steps = 700
9          self.move_obstacles = True
10         self.algorithm = 1
11
12         # Parámetros EAs Generales
13         self.population_size = 100
14         self.iterations = 100
15
16         # Parámetros Differential Evolution
17         self.low = 0
18         self.high = 1
19         self.F = 0.1
20         self.cross_probability = 0.9
21
22         # Parámetros MAP-Elites
23
24         # Feature Space
25         self.upperlimit = self.height
26         self.lowerlimit = 0
27         self.feature_dimensions = 2
28         self.granulation = 10
29
30         # Parámetros Evolución
31         self.crossover_rate = 0.2
32         self.mutation_rate = 0.8
33         self.sigma = 0.05
34         self.cross = False
```

Código 13.10: Código de la configuración del simulador.

```
0  import sys
1  import os
2  import math
3  import time
4  import numpy as np
5  import cv2
6  import imageprocessing
7  import pandas as pd
8  sys.path.append(os.path.join(os.path.dirname(
9      '_file_'), '.', 'robobo.py-master'))
10
11 sys.path.append(os.path.join(os.path.dirname('_file_'), '.', 'robobo-python-video-stream-master'))
12
13 from utils.Wheels import Wheels
14 from utils.LED import LED
15 from utils.Color import Color
16 from NeuralNetwork import JordanNetwork
17 from robobo_video.robobo_video import RoboboVideo
18 from Robobo import Robobo
19
20 tilt_speed = 15
21 angle = 85
22 k_blue = 1363
23 delay = 16
24
25 # Configuración
26 rob = Robobo("192.168.1.167")
27 videoStream = RoboboVideo("192.168.1.167")
28 rob.connect()
29 videoStream.connect()
30
31 # Posicionar Smartphone
32 rob.moveTiltTo(angle, tilt_speed)
33 rob.wait(2)
```

```
34
35 # Configuración red
36 weights = np.load('pared.npy')
37 nn = JordanNetwork(4, 4, 2)
38 nn.update_weights(weights)
39
40 t_last_frame = int(round(time.time() * 1000))
41 t_initial = t_last_frame
42
43 while True:
44     t = int(round(time.time() * 1000))
45
46     if t > t_last_frame + delay:
47
48         img = videoStream.getImage()
49
50         # Procesamiento
51         t_1 = int(round(time.time() * 1000))
52
53         cord = 0
54         while cord < 4:
55             vert = img[:,36 + 72*cord:36 + 72*cord + 1,:]
56             res_blue = imageprocessing.mask_blue(vert)
57
58             # Obtención de la franja horizontal
59             for i in range(0,res_blue.shape[0]):
60                 if res_blue[i][0][0] > 0 and res_blue[i-1][0][0] == 0:
61                     initial_point = i
62                     found = cord
63                     cord = 4
64                     break
65                 else:
66                     initial_point = 0
67                     found = 0
68             cord += 1
69
70         h = initial_point + 20
71         hor = img[h:h+1,:,:]
72         res_hor_blue = imageprocessing.mask_blue(hor)
73         res_hor_green = imageprocessing.mask_green(hor)
74
75         # Rellernar celdas de la rendija
76         inputs = []
77         for celda in range(4):
78             w = imageprocessing.sub_pixel_filter(res_hor_green[:,72*celda:(celda + 1)*72,:])/72
79             if w > 0.001:
80                 inputs.append(-w)
81             else:
82                 estimation = k_blue/(imageprocessing.sub_pixel_filter(res_blue))
83                 distance = 0.00000918*math.pow(estimation,3)-0.00154*math.pow(estimation,2) +1.08*estimation-1.37
84                 inputs.append(1 -np.tanh(distance*1/150))
85
86         v_r, v_l = nn.propagate_forward(inputs)
87
88         # Post-procesamiento de la velocidad
89         if v_r >= v_l:
90             v_max = v_r
91             v_min = v_l
92         else:
93             v_max = v_l
94             v_min = v_r
95
96         c_20 = (v_max + v_min)/(20*(v_max - v_min))
97
98         v_mod = v_max*(c_20 - 1)/(c_20 + 1)
99
100        if v_r >= v_l:
101            v_r = v_max
102            v_l = v_mod
103        else:
104            v_l = v_max
105            v_r = v_mod
106
107        rob.moveWheels(int(v_r*60), int(v_l*60))
108
109        """
110        img_green = imageprocessing.mask_green(img)
111        img_green = cv2.cvtColor(img_green, cv2.COLOR_HSV2BGR)
112        img_blue = imageprocessing.mask_blue(img)
```

```
113         img_blue = cv2.cvtColor(img_blue, cv2.COLOR_HSV2BGR)
114 
115 
116         cv2.namedWindow('imagen', cv2.WINDOW_NORMAL)
117         cv2.imshow('imagen', img_blue + img_green)
118         key = cv2.waitKey(1) &0xFF
119         if key == ord('q'):
120             videoStream.disconnect()
121             break
122         """
123 
124         t_last_frame = t
125 
126 videoStream.disconnect()
```

Código 13.11: Código de la prueba de los controladores en el robot real.

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

PLANOS

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBREIRO, S/N

15405 - FERROL

FECHA: SEPTIEMBRE DE 2020

AUTOR: EL ALUMNO

Fdo.: DIEGO JAVIER GUTIÉRREZ BARRIO

Este documento no se aplica a este tipo de trabajos.

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

PLIEGO DE CONDICIONES

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBREIRO, S/N

15405 - FERROL

FECHA: SEPTIEMBRE DE 2020

AUTOR: EL ALUMNO

Fdo.: DIEGO JAVIER GUTIÉRREZ BARRIO

Índice del documento PLIEGO DE CONDICIONES

14 PLIEGO DE CONDICIONES 129

14 PLIEGO DE CONDICIONES

Este documento no se aplica a este tipo de trabajos.

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

MEDICIONES

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBREIRO, S/N

15405 - FERROL

FECHA: SEPTIEMBRE DE 2020

AUTOR: EL ALUMNO

Fdo.: DIEGO JAVIER GUTIÉRREZ BARRIO

Índice del documento MEDICIONES

15 Mediciones **135**

15 Mediciones

Este documento no se aplica en este tipo de trabajos

TÍTULO: **Desarrollo de un experimento de robótica colectiva mediante evolución distribuida.**

PRESUPUESTO

PETICIONARIO: **ESCUELA UNIVERSITARIA POLITÉCNICA**

AVDA. 19 DE FEBREIRO, S/N

15405 - FERROL

FECHA: SEPTIEMBRE DE 2020

AUTOR: EL ALUMNO

Fdo.: DIEGO JAVIER GUTIÉRREZ BARRIO

Índice del documento PRESUPUESTO

16 PRESUPUESTO **141**

16 PRESUPUESTO

Elemento	Ud	Precio
Mano de obra	400 horas	40 €/h
Robobo	1	329 €/ud
Smartphone	1	120 €/ud
Cinta	3 metros	4 €/m
Cajas de cartón	5	0,23 €/ud
Total		16462.15 €

Tabla 16.1 – Tabla de presupuesto.