TRABAJO FIN DE GRADO

Grado en Ingeniería Electrónica Industrial

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Control de flotas de robots utilizando el software CoppeliaSim en Python

Curso: 2020/2021

Modalidad TFG: Trabajo técnico

Alumno/a:

Jesús Aporta Costela

Director/es:

María del Mar Castilla Nieto

José Carlos Moreno Úbeda

UNIVERSIDAD DE ALMERÍA

ESCUELA SUPERIOR DE INGENIERÍA

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL

TRABAJO FIN DE GRADO

Control de flotas de robots utilizando el software CoppeliaSim en Python

---

Alumno: *Jesús Aporta Costela*

Director: *María del Mar Castilla Nieto*

Codirector: *José Carlos Moreno Úbeda*

Fecha: *Julio 2021*

Jesús Aporta Costela	*María del Mar Castilla Nieto*	*José Carlo Moreno Úbeda*

Quisiera dar gracias a todas las personas que me han apoyado en esta etapa, mi familia, amigos y compañeros que siempre han estado conmigo.

Al grupo de investigación de Automática, Robótica y Mecatrónica (ARM) de la Universidad de Almería, por su labor docente a lo largo de estos años de carrera y un agradecimiento en especial a José Carlos Moreno Úbeda y María del Mar Castilla Nieto por su tiempo y por guiarme haciendo este TFG.

SIGLAS	SIGNIFICADO	TRADUCCIÓN
TFG	Trabajo de Fin de Grado	-
BSA	Backtracking Search Algorithm	Algoritmo de búsqueda de retroceso
PSO	Particle Swarm Optimization	Optimización de enjambre de partículas
MCMM	Multi-hop Collaborative Min-Max	Min-Max Multi-salto colaborativo
POO	Programación Orientada a Objetos	-
IDE	Integrated Development Environment	Entorno de programación
GUI	Graphical User Interface	Interfaz de usuario
IR	Infrared Ray	Rayos infrarrojos
OHC	OverHead Controller	Controlador suspendido

SIMBOLO	SIGNIFICADO
Ts	Tiempo que se toma para realizar el método Sum-Dist
r	Nombre identificación de un robot ancla
i	Número de un robot nodo
l	Distancia recorrida de un mensaje
d	Distancia del remitente de un mensaje
$B_{i,r}$	Cuadro delimitador de un nodo
S_i	Intersección de cuadros delimitadores de un nodo
$\in$	Robot nodo cuando es robot ancla auxiliar
β	Límite de alineamiento
ζ	Factor de confianza
P	Matriz *P* de población
P_{hist}	Matriz *P* de población histórica
Q	Coeficiente de cruzado
$X_{Y,D}$	Matriz de mapeo

Abstract

Nowadays mobile robotics is having an important role, at industrial and social level. More specifically, one of the fields that is currently emerging is swarm robotics. The main objective of this diploma thesis is the development of a swarm robotics algorithm to control minirobots fleets, specifically Kilobots, both at the simulation and real levels.

In this diploma thesis a bibliographic study about swarm robotics algorithms has been developed, in addition, the algorithm "distributed and resilient localization" for a self-location based on the adjacent robots and the "wave" algorithm is implemented for a joint navigation of a robot's swarm. The simulation is carried out with CoppeliaSim Edu software making use of Kilobots and, afterwards, real tests have been performed using real Kilobots.

The programming of Kilobots have been developed in CoppeliaSim with Python since it offers several tools for a simple programming. To use Python in CoppeliaSim, Anaconda software has been used. It is a software to manage several libraries from Python besides offering several programming environments like for example Jupyter.

Keywords: Swarm technology, swarm robotics, fleet control, Kilobots, CoppeliaSim, multirobot systems.

Resumen

La robótica móvil está teniendo un papel importante en la actualidad tanto a nivel industrial como social. Más concretamente, una de las áreas que está emergiendo actualmente es la robótica de enjambre o control de flotas de robots. Este Trabajo de Fin de Grado (TFG) trata sobre la implementación de algoritmos de tecnología de enjambre dirigidos al control de flotas de minirobots, concretamente Kilobots, tanto a nivel de simulación como de robots reales.

En este TFG se realiza un estudio bibliográfico sobre algoritmos que usan tecnología de enjambre, se implementa el algoritmo "distributed and resilient localization" para una localización propia en base a los robots adyacentes, y el algoritmo "wave" para una navegación conjunta de una flota de robots. La simulación se lleva a cabo con el software *CoppeliaSim Edu* haciendo uso de robots Kilobots y a partir de esta simulación se hace un ensayo con Kilobots reales de los algoritmos mencionados.

La programación de los Kilobots en CoppeliaSim se desarrolla en lenguaje Python puesto que ofrece una serie de herramientas para hacer una programación más sencilla. Para el uso de Python en CoppeliaSim se hace uso de Anaconda, un software para administrar las diferentes librerías utilizadas en Python, que además ofrece varios editores de programación, como por ejemplo Jupyter.

Palabras clave: Tecnología de enjambre, control de flotas, Kilobots, CoppeliaSim, sistemas multi robot.

1.1. Motivación

Los robots están siendo utilizados en todos los ámbitos que nos rodean, en hospitales, océanos, casas, escuelas, industria, etc. sirviendo para un amplio abanico de propósitos desde salvar vidas en medicina hasta combatir el fuego en un incendio **[1].** Juegan un papel clave en el mundo de la salud, transporte y la exploración del espacio por lo que van a tener un papel aún más importante en nuestra vida futura.

Una parte de la robótica que está en auge y tiene potencial en aplicaciones a cualquier nivel es la de control de flotas o tecnología de enjambre **[2]**. Esta tecnología usa una flota de robots cuya función es trabajar de forma conjunta para realizar una tarea. Estos sistemas de robots son robustos, escalables y flexibles, permitiendo una multitud de usos como la exploración, vigilancia, búsqueda, rastreo, limpieza y transporte de objetos, entre otros **[3]**.

1.2. Objetivos

El objetivo fundamental de este Trabajo Fin de Grado (TFG) es el diseño de un sistema de robots con limitaciones sensoriales que, trabajando con inteligencia de enjambre, sea capaz de llevar a cabo tareas de forma conjunta, coordinando a todos sus integrantes. Llevar a cabo este TFG conlleva un estudio de la tecnología de enjambre, por lo que se profundizará en ésta, en los algoritmos que permiten un trabajo conjunto de robots y en sus posibles utilidades en un caso real.

La implementación se ha realizado en dos etapas, una de simulación que se ha desarrollado en el software CoppeliaSim **[4]** y otra con robots reales llamados Kilobots, véase la figura 1.1.

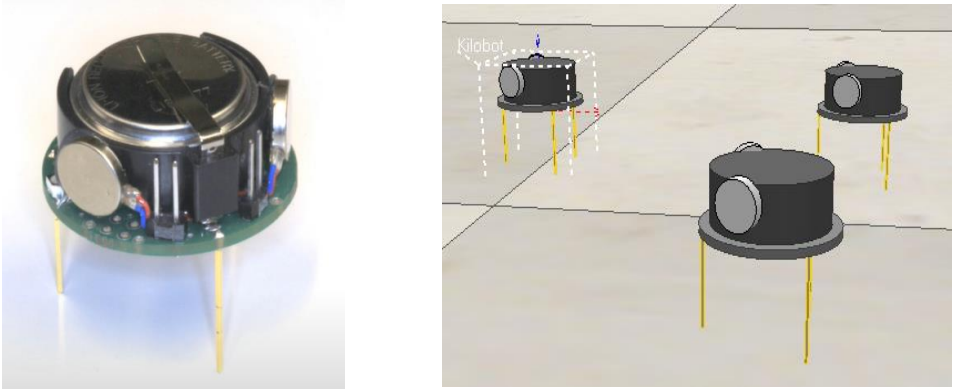


Figura 1.1. Kilobot real (izquierda) y modelo de Kilobot disponible en el software CoppeliaSim (derecha)

Para alcanzar este objetivo se han llevado a cabo las siguientes etapas:

- Estudio de la robótica de enjambre, búsqueda de información de los diferentes algoritmos en los que un grupo de robots trabaje de forma conjunta para la realización de una tarea.
- Búsqueda de información del funcionamiento del software CoppeliaSim para la modelación de sistemas robotizados en simulación.
- Diseño de una escena con robots Kilobots dispuesta para la implementación de algoritmos con tecnología de enjambre.
- Aprendizaje del software Anaconda y del lenguaje de programación Python.
- Estudio de los robots Kilobots: búsqueda de información y estudio de sus capacidades para la elección de un algoritmo adecuado.
- Programación de los algoritmos para la simulación de los Kilobots en CoppeliaSim.

- Ensayos en simulación de los algoritmos y comprobación del correcto funcionamiento.
- Análisis detallado del funcionamiento de la parte software y hardware de los Kilobots.
- Configuración de Eclipse para programar los Kilobots.
- Programación de los algoritmos para que funcionen en Kilobots reales, haciendo las modificaciones pertinentes debido a la diferencia entre el Kilobot real y el de simulación proporcionado por CoppeliaSim.
- Ensayos en caso real y comprobación del correcto funcionamiento.
- Comparación de los ensayos en simulación y los reales y discusión de los resultados

1.3. Contexto

La robótica de enjambre está inspirada en los comportamientos sociales de los animales. Las hormigas, abejas, pájaros, peces y ovejas son ejemplos de cómo una unión de individuos simples puede realizar una tarea de forma conjunta de mayor complejidad que las que podrían realizar cada uno de ellos por separado **[3]**. En la figura 1.2. se muestran ejemplos de estos comportamientos, como un enjambre de abejas que trabaja de forma conjunta para la recolección de miel, un conjunto de peces que se agrupa para protegerse de los depredadores y una bandada de pájaros o un rebaño de ovejas que siguen una trayectoria en grupo para evitar perderse.

Figura 1.2. Ejemplos de enjambres en la naturaleza, empezando desde arriba a la izquierda: Enjambre de abejas, Banco de peces, Bandada de pájaros, Rebaño de ovejas.

La robótica de enjambre está definida como "*un novedoso enfoque para la coordinación de sistemas robotizados que cuentan con un gran número de robots*" y como "*el estudio de cómo diseñar un número de agentes simples de modo que un comportamiento colectivo deseado emerja de la interacción entre los propios agentes y el medio que les rodea*" **[3]**. Otra manera de definir la robótica de enjambre es la de un estudio de cómo hacer que los robots colaboren de forma colectiva para resolver una tarea que de forma individual sería imposible de realizar **[2]**.

Un sistema de enjambre de robots tiene tres ventajas principales mencionadas anteriormente **[2]**:

- **Robustez**: Consiste en proporcionar seguridad ante fallas. En caso de fallo de algún robot, se puede sustituir por otro al ser todos iguales. El hecho de ser un sistema descentralizado lo hace mucho más robusto .
- **Flexibilidad:** Debido a la homogeneidad de los individuos del conjunto, cada robot puede adaptarse de forma que el enjambre pueda cumplir la tarea sin necesidad de especialización en términos de hardware. Los robots superan las limitaciones de sus capacidades mediante la cooperación.
- **Escalabilidad**: Puesto que cada robot trabaja de forma independiente, el tamaño del enjambre es completamente ajustable.

Los algoritmos que usan inteligencia de enjambre hacen uso de los robots como una red neuronal. Más concretamente, se puede usar una flota de robots para que resuelva tareas de forma colectiva usando un método en el que el conjunto de robots representa una red neuronal y en la que cada robot representa una neurona. El conjunto de robots trabaja de forma que se intercambian información entre ellos para obtener información de su entorno y saber qué acciones hay que llevar a cabo para hacer la tarea para la que están programados **[5]**. En la figura 1.3. se expone un esquema para la programación de un enjambre de robots para que actúen como red neuronal. Primeramente, se hace la programación necesaria para a continuación colocarlos aleatoriamente en un área de trabajo y a partir de la información captada del entorno por el grupo, llevar a cabo una tarea para la que están programados que se decide entre todo el enjambre.

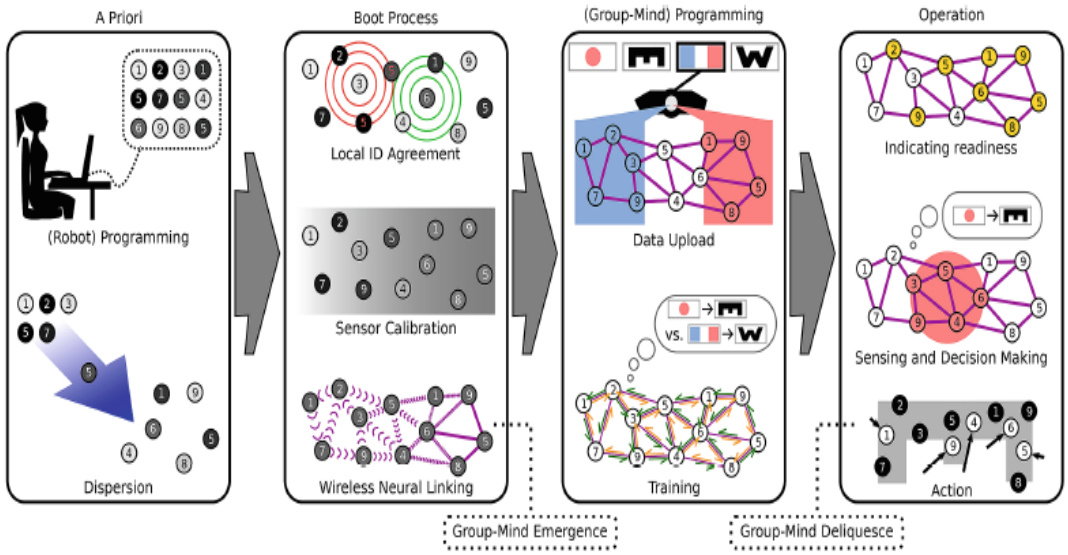


Figura 1.3. Esquema de programación de un enjambre de robots para que actúen como una red neuronal
Fuente: **[5]**

En la actualidad la aplicación de la tecnología de enjambre a la robótica está en proceso de maduración. Una de las utilidades es la de hacer uso de una flota de robots que de manera cooperativa encuentra una solución a un problema de optimización de rutas en un entorno con obstáculos, ya que la clave de los problemas de optimización de rutas es la de obtener una descripción del entorno que se desea encontrar una ruta. Un solo robot explorando un entorno es más lento que un enjambre explorando el mismo entorno. En un enjambre cada robot trabajaría de forma autónoma abarcando más terreno e intercambiando la información de la trayectoria recorrida con todo el enjambre sin necesidad de que todos los robots recorran todo el entorno. Una vez recopilada la información se encuentra la ruta a partir de otros algoritmos. El problema de estos algoritmos es que se necesita información previa del terreno para poder implementarse, además de que los robots deben ser supervisados desde una computadora principal **[6]**.

Es posible usar un algoritmo de este tipo para mapear una zona sin necesidad de información previa y así encontrar una solución a un problema sin la utilización de una computadora principal que los coordine, esto se consigue haciendo que los robots compartan la información obtenida del entorno entre ellos haciendo así que el enjambre alcance una solución a un problema dado. Un ejemplo de aplicación de este método es el de un algoritmo que usa control difuso para el rescate de personas desaparecidas en un lugar devastado. Al ser una zona totalmente desconocida y los robots trabajar de forma autónoma, se necesita un robot con una red sensorial sofisticada para poder obtener una amplia información del entorno o bien un enjambre de robots más sencillos con capacidad de comunicación entre sí **[7]**.

Actualmente se está empezando a usar la tecnología de realidad aumentada. Se hace uso de realidad virtual que se combina con la realidad para obtener información del entorno, para hacer todo esto se necesita de un sistema de comunicación de robots con una computadora y una cámara capaz de detectar códigos QR, estos códigos QR se pueden imprimir en papel y pegarse a la base superior del robot como se muestra en la figura 1.4., de tal forma que pueda ser detectado este código por la cámara pudiendo hacer un seguimiento del comportamiento del robot y proporcionarle información **[8] [9]**.

Figura 1.4. Ejemplo de un robot con un código QR para ser detectado por una cámara
Fuente:**[9]**

Para la implementación de los algoritmos mencionados es necesario contar con robots diseñados para este fin. Existen varios tipos de robots con una amplia variedad de cualidades para poder trabajar con tecnología de enjambre, como por ejemplo:

- ***Khepera IV*** (figura 1.5.): Este robot tiene cualidades para trabajar en enjambre ya que es compacto, tiene varias formas de comunicarse con otros robots como wifi y bluetooth, sensores infrarrojos para detección de obstáculos, cámara a color, micrófono, altavoces, giroscopio y acelerómetro además de procesador con gran potencia. El problema de este robot es que al tener tantos componentes su precio es elevado, aunque se está comercializando **[11]**.

Figura 1.5. Robot Khepera IV
Fuente:**[11]**

- ***I-Swarm*** (figura 1.6.): Un robot extremadamente pequeño, de dimensiones 3x3x3 mm, que no tiene batería, funcionando a partir de una célula solar. Su mecanismo de movimiento se basa en unas pequeñas barras piezoeléctricas que vibran permitiendo el movimiento del robot. Se pueden comunicar entre ellos por infrarrojos. Al ser un robot muy simple es muy barato, pero tiene su propio chip programable y ninguna compañía ha querido invertir todavía en el desarrollo de este chip **[2].**

Figura 1.6. Robot I-Swarm
Fuente:**[2]**

- ***Alice*** (figura 1.7.): Es un robot de dimensiones también muy pequeñas (22x21x20 mm) pero este tiene mejores prestaciones ya que se mueve con ruedas, tiene sensores infrarrojos que no solo permiten la comunicación sino detección de obstáculos y un módulo de radio que permite comunicación con un controlador principal o con otro robot lejano **[12]**.

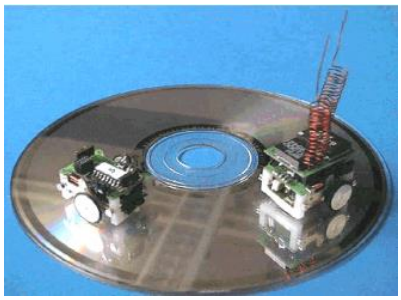


Figura 1.7. Robots Alice comparados con un CD
Fuente: **[12]**

Sin embargo, uno de los robots más famosos especializados en tecnología de enjambre es el Kilobot (figura 1.8.). Son robots de bajo coste, que han sido diseñados en la universidad de Harvard por Radhika Nagpal y Michael Rubenstein y sus cualidades a destacar, además de su bajo coste, son la capacidad de movimiento, detección de la luz ambiental, un LED RGB, detección de la distancia de un Kilobot vecino y la capacidad de comunicarse con otros Kilobots que estén en un radio de hasta 7 cm **[2][10]**. Puesto que este TFG se desarrolla con este robot, se hablará detalladamente de él más adelante.

Figura 1.8. Enjambre de Kilobots
Fuente: **[10]**

Uno de los apartados más importantes en la tecnología de enjambre es el de la comunicación entre robots. Un método para propagar un mensaje entre el enjambre es el algoritmo Wave **[13]**. El funcionamiento de este algoritmo es el siguiente: un robot iniciador empieza enviando un mensaje a sus vecinos y éstos, a su vez, replican el mensaje a sus adyacentes simulando el comportamiento de una onda, de ahí su nombre. Se puede usar además de para la propagación de mensajes, para crear varios grupos de robots dentro de un enjambre. Estos grupos están ordenados de forma jerárquica en la que un robot obedece a un robot líder que tiene varios esclavos y estos esclavos a su vez pueden hacer de líder para otros robots esclavos creando así una jerarquía. Con esto se consigue que varios grupos puedan ejecutar tareas que requieran de varios grupos de robots o tareas diferentes asignadas a cada grupo **[13][14]**. En la figura 1.9. se muestra cómo se crean dos grupos, celeste y morado, dentro de un enjambre. Cada grupo tiene su líder, S_A para el grupo celeste y S_B para el grupo morado y sus esclavos son los robots con el número 1 que además de ser esclavos se encargan de guiar a los robots con el número 2.

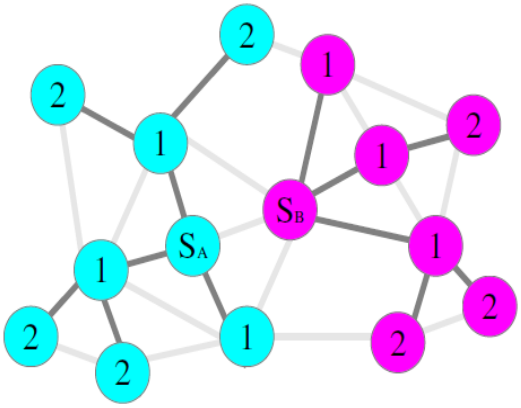


Figura 1.9. Ilustración de división de un enjambre por algoritmo wave
Fuente:**[13]**

Otro método para crear grupos dentro de un enjambre seria mediante el uso de tokens virtuales **[15]**. Los robots pasan los tokens a través de mensajes que viajan por el enjambre buscando una posición óptima para convertirse en un token estático. Lo que rige el movimiento de los tokens es la función densidad, que determina cuándo y hacia donde se mueven los tokens. La condición para convertirse en un token estático es la de encontrar una posición donde el valor de la función densidad en la posición actual del token es el mejor comparado con el valor de los robots adyacentes. Se puede utilizar para dividir un enjambre de robots en 2 o más grupos de forma equitativa al contrario que en el algoritmo wave. Este método es rápido y robusto, pero es más complejo de programar y necesita más potencia computacional a diferencia del algoritmo wave **[15]**.

Otro aspecto importante en la robótica de enjambre es el de la localización de cada entidad en el enjambre, esto se puede hacer utilizando varias técnicas de localización, la más simple seria a partir de un controlador principal que provee información a cada robot con su posición actual **[2]**. Otra técnica utilizada consiste en implantar un sistema de odometría con el que el robot sabe su posición en todo momento, pero el problema de estos sistemas es que suelen conllevar a errores acumulados debido al movimiento como por ejemplo en el caso de que una rueda patine. Además, este sistema solo funciona en robots con movilidad sobre ruedas ya que los cálculos de la posición se hacen dependiendo del giro de cada rueda **[2]**.

Se puede usar la tecnología de enjambre para la localización con el uso de algoritmos y la comunicación entre los robots. Estos hacen uso de los mensajes que envían los robots, que contienen información de su distancia para estimar su propia posición en base a un robot de referencia. Haciendo uso del algoritmo Sum-Dist **[23]** se calcula la distancia que viajan los mensajes para obtener una estimación de

la distancia a la que se encuentran los robots referencia del enjambre. Posteriormente, se usa el algoritmo Min-Max **[23]** para calcular una estimación de la posición. Como esta estimación es poco precisa, se usan otros métodos de refinamiento como el algoritmo *Backtracking Search Algorithm* (BSA) **[24]** o el *Particle Swarm Optimization* (PSO) **[16]**. Ambos están inspirados en el comportamiento de las abejas. Las abejas se dividen en grupos en función de su capacidad para solucionar un determinado problema/tarea. Estos grupos mutan atrayendo a otras abejas, que podrán ocupar un lugar en el grupo si resuelven el problema mejor que alguna abeja del grupo original sustituyendo esta abeja por la abeja mutada **[16]**.

Una de las típicas tareas que se le suelen asignar a los robots con tecnología de enjambre es la de exploración, por lo que hay multitud de algoritmos diseñados para este fin. Algunos a destacar son:

- **Modelo SkyBat**: Este modelo está inspirado en el comportamiento de los murciélagos para encontrar refugio. Con este algoritmo, los robots buscan una solución de posicionamiento en un entorno donde hay más de una solución válida, los robots trabajan de forma autónoma sin líder y buscando siempre una mejor solución. El funcionamiento de este modelo se basa en robots que primeramente comienzan en una zona de inicio y comienzan a moverse en el medio aleatoriamente siguiendo trayectorias lineales buscando posibles soluciones. Una vez se encuentra una solución se coloca en esta y empieza a retransmitirla durante un tiempo para atraer a otros robots hacia esta solución, en el caso de que un robot encuentre varias soluciones, el robot se mueve hacia la solución con más robots. El tiempo de retransmisión de la solución depende de la calidad de la solución, a mayor calidad mayor tiempo de retransmisión asegurando así que la mejor solución sea la más retransmitida **[17]**.

- **Algoritmo basado en feromonas virtuales:** Los robots obtienen información del entorno basándose en feromonas virtuales que son soltadas por robots exploradores que van indicando regiones a las que acercarse o alejarse creando así una especie de red neuronal robótica que describe el espacio total de trabajo. En la figura 1.10. se puede ver un ensayo del algoritmo de búsqueda por feromonas donde se muestra el rastro dejado y el camino que toman los robots en base a estas feromonas **[18]**.

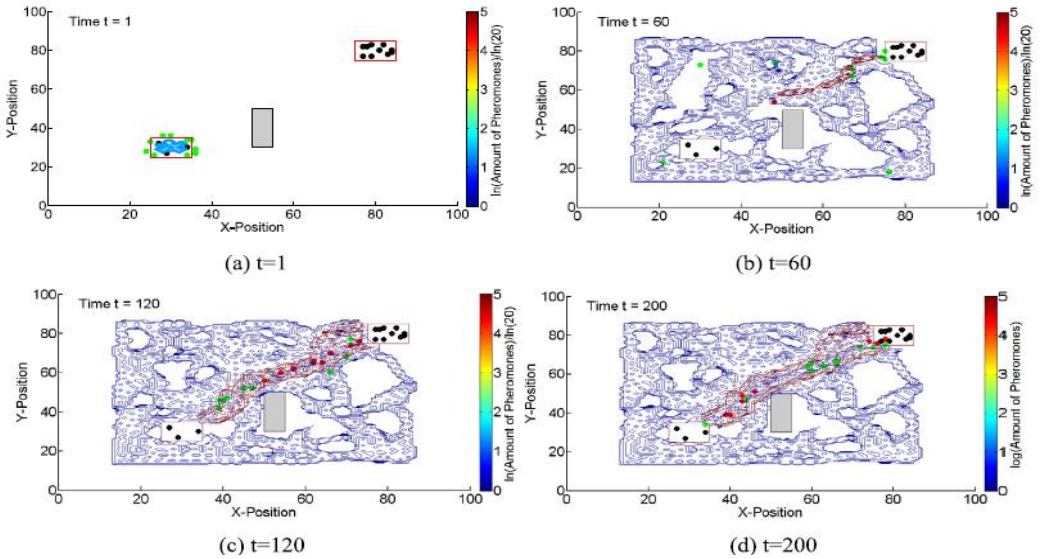


Figura 1.10. Ejemplo de uso del algoritmo de búsqueda de objetivos por feromonas
Fuente:**[18]**

Para poder implementar un movimiento hacia un objetivo, hacen falta algoritmos que rijan el movimiento de los robots. Hay varias formas de hacerlo, algunos ejemplos son los basados en campos de fuerza o el *Morphogen Gradient Following* **[19]**. En este último se utiliza el conjunto de lecturas de los robots para crear un campo con un gradiente dirigido hacia un objetivo.

Un aspecto importante para la realización de este TFG es el de la simulación. Hay cientos de aplicaciones software de simulación de robots, pero no todas son aptas para la simulación de robótica de enjambre, ya que muchas de ellas se centran en células robóticas o en otro tipo de robots **[2]**. Los simuladores más importantes relacionados con el objetivo de este TFG son:

- ***ARGos***: Es un simulador con multi-físicas, que puede simular un gran enjambre de robots de cualquier tipo de forma eficiente. Usa un método de paralelización en el que permite que el bucle principal se distribuya en varios sub-bucles para así poder simular múltiples robots **[2][20]**.
- ***Webots***: Este simulador provee herramientas para poder crear robots definidos por el usuario y simularlos. Es de propósito general y se usa tanto en la industria como en el mundo de la educación **[21]**.
- ***Kilombo***: Simulador especializado en la simulación de Kilobots. Está desarrollado en el mismo lenguaje que los Kilobots, por lo que se le puede extrapolar el código a un Kilobot real sin necesidad de adaptarlo. Permite simular 1000 Kilobots a una velocidad 100 veces mayor que en la realidad **[22]**.

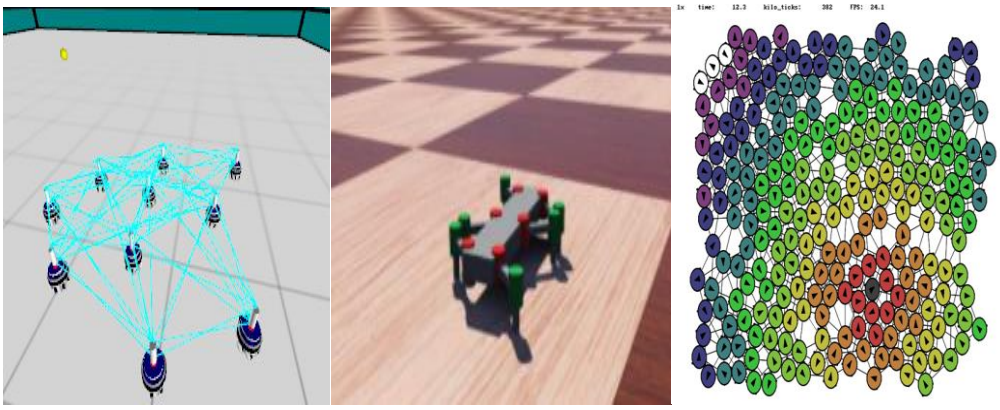


Figura 1.11. Imágenes de los simuladores, empezando desde izquierda, *ARGos*, *Webots* y *Kilombo*

El desarrollo de este TFG se realiza con **CoppeliaSim**, un software de simulación de robots en el que éstos se controlan individualmente, ya que cada uno funciona con su propio script. CoppeliaSim proporciona una API que permite que se puedan implementar los scripts de cada robot en distintos lenguajes como C, C++, Python o Java. De este software se hablará más detalladamente en los siguientes apartados **[4]**.

1.4. Resumen de resultados

Se han alcanzado los objetivos mencionados anteriormente puesto que se ha programado en simulación y en caso real un enjambre de robots con algoritmos de inteligencia de enjambre. El estudio de la robótica de enjambre ha mejorado la comprensión del funcionamiento de los robots y del trabajo cooperativo entre ellos, así como el potencial que puede tener en un futuro. Al trabajar con variedad de herramientas como el simulador CoppeliaSim, librerías, entornos de programación, diferentes lenguajes de programación y robots reales, ha permitido alcanzar un conocimiento del ámbito de la robótica útil para casos más complejos.

Se han implementado dos algoritmos, uno de posicionamiento, viendo los inconvenientes y ventajas de utilizarlo frente a otros métodos para la localización de los robots, y comprobando las diferencias entre el caso real y el simulado, siendo los resultados bastante satisfactorios. Se han hecho diferentes ensayos para validar este algoritmo usando diferentes posiciones de un robot.

La programación del segundo algoritmo ha sido más fácil, pero con diferencias más notables entre el caso real y la simulación, pero esto es debido a que en la simulación todos los robots se mueven de forma ideal y el sensor de luz diseñado en CoppeliaSim difiere mucho en funcionamiento al del robot real. Se han programado dos algoritmos de movimiento, basados en el seguimiento de un robot líder que guía al enjambre hacia la luz.

El primer algoritmo de réplica de movimientos, ha obtenido resultados muy parecidos a los de simulación, incluyendo ruido con el que los Kilobots se desorientaban y no llegaban a un resultado óptimo del caso ideal. El segundo algoritmo basado en un sistema realimentado no se ha podido implementar en los Kilobots reales debido a que al ser un algoritmo más complejo no había suficiente espacio en la memoria de estos.

El enlace https://youtu.be/yYqOI8ZMM4g lleva a un video con un resumen de los ensayos que se han realizado tanto en simulación como con Kilobots reales de los algoritmos en los que un Kilobot líder guía a un enjambre hacia un foco de luz.

1.5. Planificación temporal

Fecha	*Tarea*	*Dias*
Septiembre 2020	• Búsqueda de información de la robótica de enjambre y Kilobots	3
	• Estudio de la robótica de enjambre y de los algoritmos a programar	7
Octubre 2020	• Preparación de CoppeliaSim y Anaconda, comprensión funcionamiento de CoppeliaSim	2
	• Aprendizaje básico del lenguaje Python, diseño del área de trabajo en CoppeliaSim y preparación de la programación.	8
	• Programación algoritmo de localización (etapa I) (simulación)	5
Noviembre 2020	• Programación algoritmo de localización (etapa II) (simulación)	1
	• Programación algoritmo de localización (etapa III) (simulación)	8
	• Programación algoritmo de localización (etapa IV) (simulación)	5
Enero 2021	• Programación de algoritmo de localización (completo) y ensayos del algoritmo completo	4
	• Programación de algoritmo de enjambre hacia la luz (simulación) y ensayos del algoritmo	5
	• Preparación del entorno de programación Eclipse, estudio del funcionamiento de los Kilobots reales y su calibración	6
Febrero 2021	• Programación algoritmo de localización real (etapa I y II) y ensayos con este algoritmo	5
	• Programación de algoritmo de enjambre hacia la luz y su posterior ensayo con este	7
Marzo 2021	• Redacción de la memoria	16

Tabla 1.1. Planificación temporal

La elaboración del TFG total ha sido de 82 días y con una media diaria de 4 horas, el tiempo en horas total consumido ha sido de 328 horas.

1.6. Competencias utilizadas

Las competencias básicas utilizadas en este TFG son las definidas en el R.D. 1393/2007, de 29 de octubre:

- **Poseer y comprender conocimientos (CB1):** Que los estudiantes hayan demostrado poseer y comprender conocimientos en un área de estudio que parte de la base de la educación secundaria general, y se suele encontrar a un nivel que, si bien se apoya en libros de texto avanzados incluye también algunos aspectos que implican conocimientos procedentes de la vanguardia de su campo de estudio.
- **Aplicación de conocimientos (CB2):** Que los estudiantes sepan aplicar sus conocimientos a su trabajo o vocación de una forma profesional y posean las competencias que suelen demostrarse por medio de la elaboración y defensa de argumentos y la resolución de problemas dentro de su área de estudio. *Competencia básica para la implementación de los algoritmos utilizados.*
- **Capacidad de emitir juicios (CB3):** Que los estudiantes tengan la capacidad de reunir e interpretar datos relevantes (normalmente dentro de su área de estudio) para emitir juicios que incluyan una reflexión sobre temas relevantes de índole social, científica o ética. *Competencia utilizada para la interpretación de los resultados obtenidos.*
- **Capacidad de comunicar y aptitud social (CB4):** Que los estudiantes puedan transmitir información, ideas, problemas y soluciones a un público tanto especializado como no especializado. *Competencia relacionada con la redacción de la memoria.*
- **Habilidad para el aprendizaje (CB5):** Que los estudiantes hayan desarrollado aquellas habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía.

Las competencias transversales necesarias para la realización de este TFG:

- **Conocimientos básicos de la profesión (UAL1):** Conocimiento, habilidades y actitudes que posibilitan la comprensión de nuevas teorías, interpretaciones, métodos y técnicas dentro de los diferentes campos disciplinares, conducentes a satisfacer de manera óptima las exigencias profesionales. *Competencia utilizada en los estudios y búsquedas de información vinculados al TFG.*
- **Habilidad en el uso de las TIC (UAL2):** Utilizar las Técnicas de Información y Comunicación (TICs) como una herramienta para la expresión y la comunicación, para el acceso a fuentes de información, como medio de archivo de datos y documentos, para tareas de presentación, para el aprendizaje, la investigación y el trabajo cooperativo. *Competencia utilizada para la búsqueda de información a través de la biblioteca de la universidad y las bases de datos ScienceDirect y Springer.*
- **Capacidad para resolver problemas (UAL3):** Capacidad para identificar, analizar, y definir los elementos significativos que constituyen un problema para resolverlo con rigor. *Esta competencia está relacionada con la elección de materiales y métodos utilizados para resolver los problemas propuestos como el de posicionamiento u orientación de un enjambre.*
- **Comunicación oral y escrita en la propia lengua (UAL4):** Comprender expresar con claridad y oportunidad las ideas, conocimientos, problemas y soluciones a un público más amplio, especializado o no especializado (y sentimientos a través de la palabra, adaptándose a las características de la situación y la audiencia para lograr su comprensión y adhesión). *Se utiliza en la redacción de la memoria.*
- **Capacidad de crítica y autocrítica (UAL5):** Es el comportamiento mental que cuestiona las cosas y se interesa por los fundamentos en los que se asientan las ideas, acciones y juicios, tanto propios como ajenos. *Se utiliza en el desarrollo de todo el TFG tanto para el estudio como en la parte de ensayos en simulación y reales cuestionando los resultados obtenidos.*
- **Conocimiento de una segunda lengua (UAL7):** Entender y hacerse entender de manera verbal y escrita usando una lengua diferente a la propia. *La mayoría de los artículos académicos en donde se basan los estudios de este TFG están en ingles por lo tanto es imprescindible comprender este idioma.*
- **Compromiso ético (UAL8):** Capacidad para pensar y actuar según principios de carácter universal que se basan en el valor de la persona y se dirigen a su pleno desarrollo. *Actuar de forma ética en la escritura de la memoria sin hacer plagio.*

- **Capacidad para aprender a trabajar de forma autónoma (UAL9):** Capacidad para diseñar, gestionar y ejecutar una tarea de forma personal. *El desarrollo del TFG se ha hecho de forma autónoma eligiendo algoritmos y materiales necesarios para el TFG.*

Las competencias especificas referentes al grado en ingeniería electrónica industrial utilizadas:

- **Conocimiento en materias básicas y tecnológicas (CT3)**, que les capacite para el aprendizaje de nuevos métodos y teorías, y les dote de versatilidad para adaptarse a nuevas situaciones. *Competencia básica para comprender los algoritmos y la realización de los estudios.*
- **Capacidad de resolver problemas con iniciativa (CT4)**, toma de decisiones, creatividad, razonamiento crítico y de comunicar y transmitir conocimientos, habilidades y destrezas en el campo de la Ingeniería Industrial. *Se ha utilizado para la implementación de los diferentes algoritmos tanto en simulación como en los casos reales.*
- **Capacidad de analizar y valorar el impacto social y medioambiental (CT7)** de las soluciones técnicas. *Utilizada en el uso de este TFG para trabajos futuros.*
- **Capacidad para aplicar los principios y métodos de la calidad (CT8)**. *Competencia utilizada en la elección y uso de los métodos utilizados.*
- **Capacidad de trabajar en un entorno multilingüe y multidisciplinar (CT10).** *La mayoría de los materiales utilizados se encuentran en inglés.*
- **Capacidad para la resolución de los problemas matemáticos que puedan plantearse en la ingeniería (CB1).** Aptitud para aplicar los conocimientos sobre: álgebra lineal; geometría; geometría diferencial; cálculo diferencial e integral; ecuaciones diferenciales y en derivadas parciales; métodos numéricos; algorítmica numérica; estadística y optimización. *En el apartado de programación se ha hecho uso de diferentes herramientas matemáticas para el cálculo de la posición.*
- **Conocimientos básicos sobre el uso y programación de los ordenadores (CB3)**, sistemas operativos, bases de datos y programas informáticos con aplicación en ingeniería. *Competencia básica utilizada para la programación de los Kilobots.*
- **Conocimientos y capacidades para organizar y gestionar proyectos (CRI12).** Conocer la estructura organizativa y las funciones de una oficina de proyectos. *Utilizada en la redacción de la memoria.*
- **Conocimiento de los fundamentos y aplicaciones de la electrónica digital y microprocesadores (CTEE3).** *Ha sido necesario para la programación de los Kilobots comprender el funcionamiento del microprocesador de este.*
- **Conocimiento y capacidad para el modelado y simulación de sistemas (CTEE7).** *Se utiliza en el desarrollo del apartado de simulación con CoppeliaSim.*
- **Conocimientos de principios y aplicaciones de los sistemas robotizados (CTEE9).** *Puesto que el TFG se ha desarrollado con un enjambre de robots se necesita de estos conocimientos.*
- **Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario (TFG),** consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería Industrial de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas. *Utilizado en la redacción de la memoria y exposición del TFG.*

1.7. Estructura de la memoria

Este TFG está estructurado en 6 capítulos. El **primer capítulo** es una introducción en la que se expone principalmente el contexto de la tecnología de enjambre, describiendo brevemente lo que es la tecnología de enjambre, varios robots con tecnología apta para funcionar en forma de enjambre y algoritmos relacionados con estos. En este capítulo también se motiva el trabajo, se resumen los resultados y se muestran las competencias utilizadas.

El **segundo capítulo** expone los materiales utilizados y aplicaciones software además de una breve explicación de la elección de estos. Se explica el algoritmo "*distributed and resilient localization*" y

"*Wave*" y el porqué de la elección de estos dos algoritmos. Una descripción de las aplicaciones software utilizadas: Anaconda, KiloGUI, Eclipse y CoppeliaSim, además de todas las librerías y lenguajes de programación utilizados en estas dos últimas. También se describe el Kilobot y todo lo relacionado con él a nivel de hardware.

El apartado de simulación se desarrolla en el **tercer capítulo,** que es donde se empieza a trabajar con el software de simulación CoppeliaSim, se muestra como se ha preparado el software para el trabajo con los Kilobots y también la programación de los algoritmos que se van a simular con los robots.

La parte donde se trabaja con los robots reales es el **cuarto capítulo** y se centra en la preparación para los ensayos reales con los Kilobots, exponiendo la calibración y la adaptación de los algoritmos utilizados en simulación para que funcionen en los robots reales.

El **quinto capitulo** expone los resultados conseguidos y se discute sobre estos resultados analizando las variaciones y posibles mejoras.

En el **sexto capitulo** se hace una conclusión del TFG y se proponen futuras líneas de trabajo relacionadas.

En este capítulo se hace una descripción de las herramientas y métodos utilizados y una explicación del porqué se han seleccionado estas.

2.1. Kilobot real

El Kilobot fue diseñado por el departamento de investigación de la universidad de Harvard, su diseño se centró principalmente en trabajar con algoritmos de robótica de enjambre para la experimentación y prueba de estos algoritmos. En el diseño del Kilobot se tuvo en cuenta la funcionalidad y el coste, ya que normalmente un robot sofisticado puede llegar a tener un coste elevado y un gran número de estos robots podría resultar en un precio desorbitado **[35][36]**.

2.1.1. Hardware

Dependiendo del modelo tiene variaciones en el sensor de luz, pero el Kilobot usado en este TFG es como el de la siguiente figura:

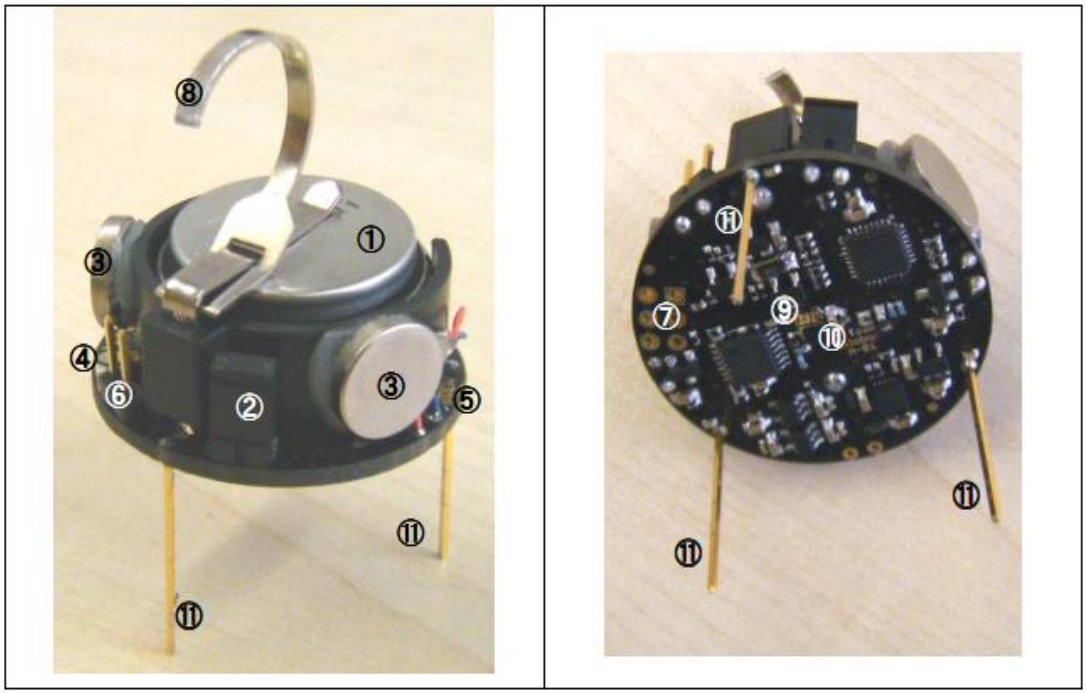


Figura 2.1. Vista isométrica (Izquierda) e inferior (Derecha) de un Kilobot.
Fuente: **[35]**

Apoyándose en la Figura 2.1. se realiza una descripción del hardware básico:

1- **Batería**: Una batería de litio de 3.4V y 160 mAh aproximadamente dependiendo de su actividad puede llegar a durar hasta 3 meses. Para la gestión de esta batería tiene 3 reguladores de voltaje uno para el microcontrolador y los otros 2 para los motores vibratorios.

2- **Jumper de encendido:** Es un conector manual para encender o apagar el Kilobot, es necesario que esté conectado para poder cargar el Kilobot.

3- **Motor vibratorio:** Los motores vibratorios son los encargados del movimiento, dependiendo de la configuración puede hacer que el Kilobot rote en sentido horario, antihorario y avanzar en línea recta. Su intensidad de vibración está regulada en 255 niveles, aunque es extraño llegar a valores muy altos.

4- **LED RGB**: Un LED RGB para generar cualquier color para interactuar con el usuario.

5- **Sensor de Luz**: Un sensor que detecta la luz del ambiente que lo rodea, este sensor puede variar dependiendo del modelo de Kilobot.

6- **Conector Serial**: Este conector se usa para el modo ***debug*** en el que un Kilobot se puede comunicar directamente con el controlador OHC y este con una computadora.

7- **Puerto de programación directa**: Un puerto para la programación directa, se usa principalmente para cargar el firmware del Kilobot.

8- **Pestaña de carga**: Esta pestaña se usa en la carga del Kilobot, ya que el cargador consta de dos varas flotantes dispuestas para asentar el Kilobot con el uso de esta pestaña.

9- **Transmisor de rayos infrarrojos (IR)**: La comunicación entre Kilobots está basada en rayos infrarrojos. Mas concretamente, el transmisor apunta hacia el suelo rebotando este rayo hacia todas direcciones y recibiendo este rayo el receptor de IR, véase figura 2.2. El alcance de los rayos varía en función del medio en el que rebota el rayo, pero normalmente suele ser de unos 7cm.

10- **Receptor de IR**: Este receptor es el que capta el conjunto de rayos, está conectado a uno de los puertos del microcontrolador.

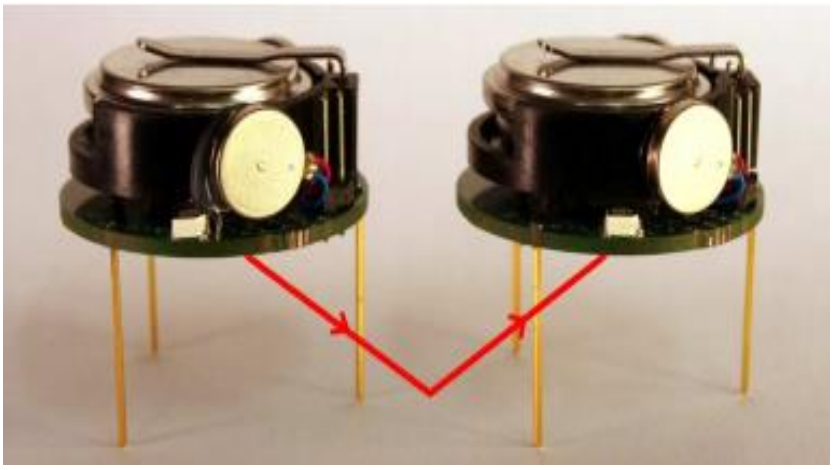


Figura 2.2. Ejemplo de la trayectoria de un rayo infrarrojo entre Kilobots.
Fuente: **[35]**

11- **Patillas**: Sirven como apoyo para el Kilobot, las piernas traseras también tienen la función de conector del polo negativo de la batería para su conexión en la estación de carga.

El microcontrolador que gobierna el Kilobot es un *ATMega 328p* de 8bit y a una frecuencia de reloj de 8MHz. Sus principales características son **[37]**:

- **32 KBytes de memoria flash:** En esta memoria es donde se guardará el código para programar los algoritmos.
- **1 KByte de memoria EEPROM:** Donde se almacena los ajustes de calibración y otros datos no volátiles.
- **2 KBytes de memoria SRAM:** Para el procesamiento de datos.
- **3 puertos**: Se usa en la conexión con los diferentes periféricos como los motores, el transmisor y el receptor de IR.

- **Un puerto Serial USART**: Para el uso del modo ***debug***.

- **Convertidor ADC de 10 bits**: Para el sensor de la luz de ambiente se necesita un convertidor de analógico a digital esto se consigue gracias a este convertidor ya integrado.

2.1.2. OverHead Controller (OHC)

El OHC es el que permite comunicar los Kilobots con un ordenador, esto se consigue a través de los infrarrojos normalmente o de forma física a través del puerto serial o el puerto de programación directa.

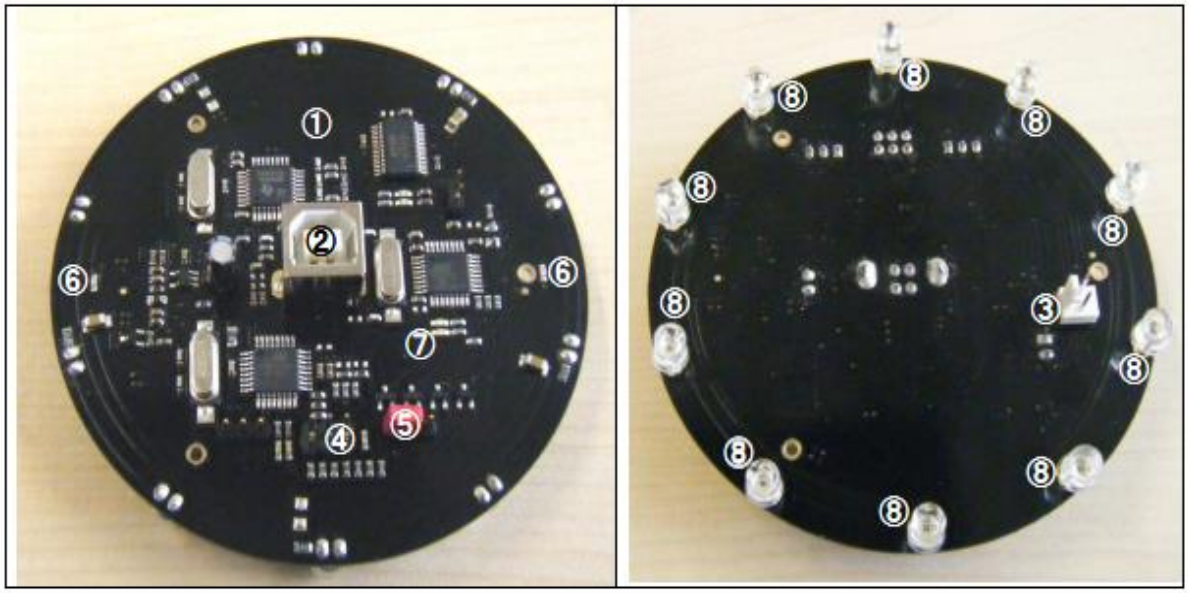


Figura 2.3. Vista superior (Izquierda) e inferior (Derecha) del OHC.
Fuente: **[35]**

Sus principales componentes son los que se muestran en la figura 2.3:

1- Placa del controlador.

2- Conector para un cable USB: Para la conexión por cable USB a un ordenador.

3- Conector para comunicación serial a un Kilobot: Este conector se utiliza cuando se quiera usar el modo ***debug*** del Kilobot.

4- Conector para programar el firmware: Cuando se necesite programar el firmware de un Kilobot se usa este puerto.

5- Jumper para modo programar/firmware: Cambiando la posición de este jumper se habilita el modo programación del firmware.

6- Diodos para test de conexión del OHC

7- LED de encendido

8- IR LED: Con estos LED se comunica el controlador con un grupo de Kilobots esto le permite programar un enjambre de Kilobots ya que se comunica de forma simultánea con todos ellos.

Para poder usar el software de "*KiloGUI*" que es el encargado de la comunicación de los Kilobots se necesita conectar este controlador al ordenador y que este controlador apunte con los LED IR hacia los Kilobots.

2.1.3. Software KiloGUI

Para el control de los Kilobots se hace uso de la aplicación "*KiloGUI*", que se puede descargar desde la página principal de Kilobotics **[38]** que además de contener esta aplicación contiene una serie de ejemplos y documentación importante para el uso de los Kilobots **[38]**.

Una vez se abre la aplicación se muestra la ventana que se puede observar en la figura 2.4:

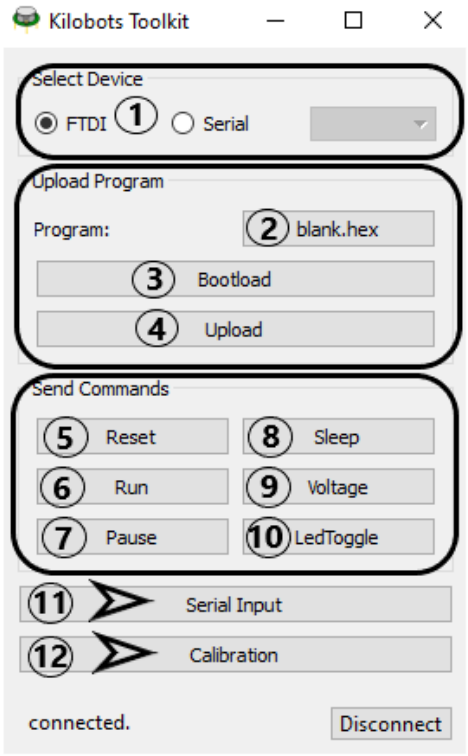


Figura 2.4. Ventana principal aplicación KiloGUI

- **Select Devide**: Esta sección es para elegir como se comunica el controlador con el Kilobot que puede ser FTDI **(1)** (a través de los LED IR) o por la conexión serie.

- **Upload Program**: Sección para cargar los archivos programados al Kilobot

  2- **Program**: Presionando aquí se selecciona el archivo que se desea cargar.

  3- **Bootload:** Si se presiona aquí ordena al Kilobot que se ponga en modo programación, y así estar dispuesto a cargar el código seleccionado en **(2).**

  4- **Upload:** Botón para iniciar la carga del código.

- **Send Commands:** Esta sección contiene varias funcionalidades para gestionar funciones del Kilobot.

  5- **Reset:** Reinicia un Kilobot y lo pone en modo espera.

6- **Run:** Inicia el código almacenado en la memoria flash.

7- **Pause:** Pausa la ejecución del código que se esté llevando a cabo.

8- **Sleep**: Pone a los Kilobots en modo dormir para un bajo consumo.

9- **Voltage**: Muestra la carga de un Kilobot dependiendo del color del LED.

10- **LedToggle**: Botón para probar la conexión del controlador con el ordenador, pulsándolo hace que se encienda una luz del controlador.

- **Serial input:** Presionando el botón **11** aparece una ventana como la mostrada en la figura 2.5 que muestra lo que está mandando un Kilobot conectado por el cable serie al controlador.

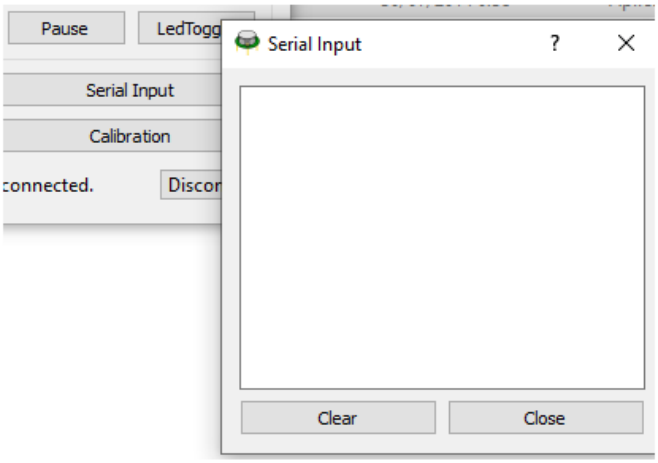


Figura 2.5. Ventana puerto serie KiloGUI

- **Calibration:** Pulsando el botón **12** aparece una ventana (figura 2.6) para la calibración de los Kilobots. Los valores que se le asignen en esta ventana son almacenados para su posterior uso en el código. El Kilobot necesita ser calibrado ya que no todos los Kilobots se mueven de la misma forma debido a diferencias en el hardware como la posición de los motores o la posición de las patillas.

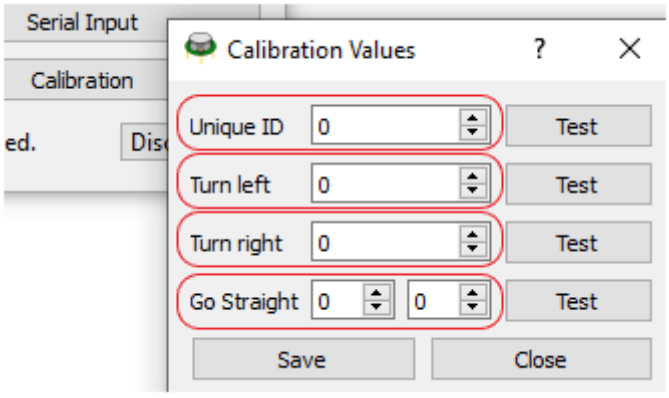


Figura 2.6. Ventana calibración Kilobots

- **Unique ID:** Con esta casilla se le asigna un número de identificación al Kilobot que se esté calibrando.

- **Turn left:** Se le asigna un valor al motor izquierdo con el que el Kilobot gire de forma factible en sentido antihorario, este valor está comprendido entre 0 y 255 pero para un funcionamiento óptimo y teniendo en cuenta la superficie, el valor suele rondar en torno a 70.

- **Turn right:** Es equivalente a **Turn left** pero para el giro en sentido horario.

- **Go Straight:** Se configura la vibración del motor izquierdo y derecho para asignar el valor con el que el Kilobot avanza en línea recta.

Una vez calibrados todos los parámetros se pulsa ***Save*** si se quiere almacenar la configuración en la memoria ROM del Kilobot o ***Close*** si se quiere descartar. Hay que tener en cuenta que la calibración se tiene que hacer robot a robot debido a las diferencias entre estos y para asignarle un ID distinto a cada uno.

2.2. CoppeliaSim Edu

CoppeliaSim es un software que proporciona las herramientas necesarias para programar, simular y verificar el comportamiento de un robot. Esto se consigue con una variedad de características que tiene integradas, las más relevantes son **[25]**:

- **Multiplataforma**, se puede usar en tres sistemas operativos Windows, MacOS y Linux.
- **Varios lenguajes de programación**, C, C++, Java, Lua, Matlab y Python. En este último lenguaje se desarrollará la programación de los Kilobots.
- **Variedad de funciones API** que son las utilizadas para gobernar los robots.
- **4 motores físicos** (ODE, Bullet, Vortex, Newton) estos motores son los encargados de simular la cinemática y dinámica en tiempo real de un robot.
- **Detección de obstáculos y cálculo de la distancia mediante sensores programables.**
- **Sensores de visión** para procesar imágenes de la simulación como objetos o formas.
- **Registro y visualización de datos.**
- **Simulación de partículas**, para la simulación de agua o aire.

Se ha escogido este simulador por ofrecer la posibilidad de programar flotas de robots y además proporcionar las herramientas necesarias para poder programar los robots en el lenguaje Python.

2.2.1. API B0-based remoted

La API B0-based remoted permite interactuar con el simulador de CoppeliaSim desde una aplicación externa. Esto se consigue definiendo dos entidades separadas: la del *cliente* y la del *servidor*. La entidad del *cliente* es la llamada aplicación que pueda ser programada en los diferentes lenguajes que permite CoppeliaSim. La entidad *servidor* es la encargada de transcribir lo programado por el *cliente* al lenguaje "Lua" de CoppeliaSim **[26]**.

La comunicación entre cliente/servidor se consigue escribiendo en la aplicación cliente las funciones que interactúan con el servidor (CoppeliaSim). Como hay una amplia variedad de funciones, a continuación solo se explican las funciones que se utilizan en este TFG **[27]**:

- **`simxServiceCall`**: En algunas funciones es necesario que se le especifique el canal de comunicación que se utilizara en la llamada de la función, se puede hacer con cinco funciones `simxServiceCall`, `simxDefaultPublisher`, `simxDefaultSubscriber`, `simxCreatePublisher` y `simxCreateSubscriber`. Todas las funciones ejecutan la función de diferente forma y ofrecen distintas posibilidades. La función **`simxServiceCall`** permite ejecutar la función en modo bloqueo (en modo bloqueo CoppeliaSim y la aplicación remota se sincronizan para obtener un resultado al mismo tiempo) y el comando a ejecutar en CoppeliaSim, y devuelve una respuesta al cliente. Esta función se utiliza cuando se necesita una única respuesta del servidor (CoppeliaSim) **[25]**.

- **`simxDefaultPublisher`:** A diferencia de la función anterior, ésta ejecuta las funciones en modo no bloqueo. Con esta forma de comunicación se ejecuta la función en la aplicación mientras se ejecuta la función en CoppeliaSim. Este modo de comunicación se usa en funciones que no necesitan una respuesta **[25]**.

- **`simxGetObjectHandle`:** Esta función devuelve el número de identificación de cualquier objeto en la escena. Requiere que se le introduzca una lista con el nombre que tiene en la escena de CoppeliaSim en el primer elemento y el canal de comunicación en el segundo. Devuelve una lista con 2 elementos, el primero un booleano indicando si la comunicación ha sido un éxito y el segundo el número de identificación del objeto con el nombre que se le ha introducido.

- **`simxSetJointTargetVelocity`:** Establece una velocidad a una articulación. Necesita que se le introduzca el número de identificación de la articulación, la velocidad y el modo de comunicación. Devuelve una lista con un solo valor, un booleano que indica si la comunicación ha sido un éxito.

- **`simxReadProximitySensor`:** Devuelve información detallada de las lecturas de un sensor. Se le introduce el numero identificador del sensor y el canal de comunicación, si se llama con éxito devuelve una lista con 6 elementos. El primer elemento indica si la función se llamó con éxito, el segundo un número que puede ser 0 o 1 para indicar si se ha detectado algo, que pueda detectar, un tercer elemento con la distancia del punto detectado. El cuarto elemento contiene una lista con el punto en coordenadas cartesianas relativo al sensor, el quinto el número de identificación del objeto detectado y el último elemento es un vector normal a la superficie del objeto detectado relativo al sensor.

- **`simxCheckProximitySensor`:** Esta función es análoga a la anterior pero solo detecta un objeto que se le introduzca a la función. Se le introduce tres parámetros: el número identificador del sensor, el del objeto a detectar y el canal de comunicación. Devuelve cuatro elementos un booleano indicando el éxito de la comunicación, un número indicando si se ha detectado el objeto introducido, un número con la distancia a la que se encuentra el objeto y una lista indicando la posición relativa al sensor de este objeto.

- **`simxGetSimulationTime`:** Para saber el tiempo de simulación, se usa para programar los temporizadores de un robot. Se le introduce el canal de comunicación y devuelve una lista con un booleano para indicar el estado de la comunicación y un número con el tiempo de simulación actual en CoppeliaSim.

- **`simxStartSimulation`:** Se utiliza para iniciar la simulación desde la aplicación que llama esta función. Necesita una entrada indicando el canal de comunicación y devuelve el estado de la comunicación y un número para indicar si se puede o no realizar la acción.

- **`simxStopSimulation`:** función análoga a la anterior, pero se utiliza para parar la simulación, los parámetros de entrada y salida son los mismos.

2.2.2. Interfaz CoppeliaSim

La interfaz de CoppeliaSim está compuesta por diversos elementos, CoppeliaSim trabaja con escenas que son las que albergan todos los diferentes elementos necesarios para la simulación. En la figura 2.7 se describen los principales elementos y herramientas que contiene una escena:

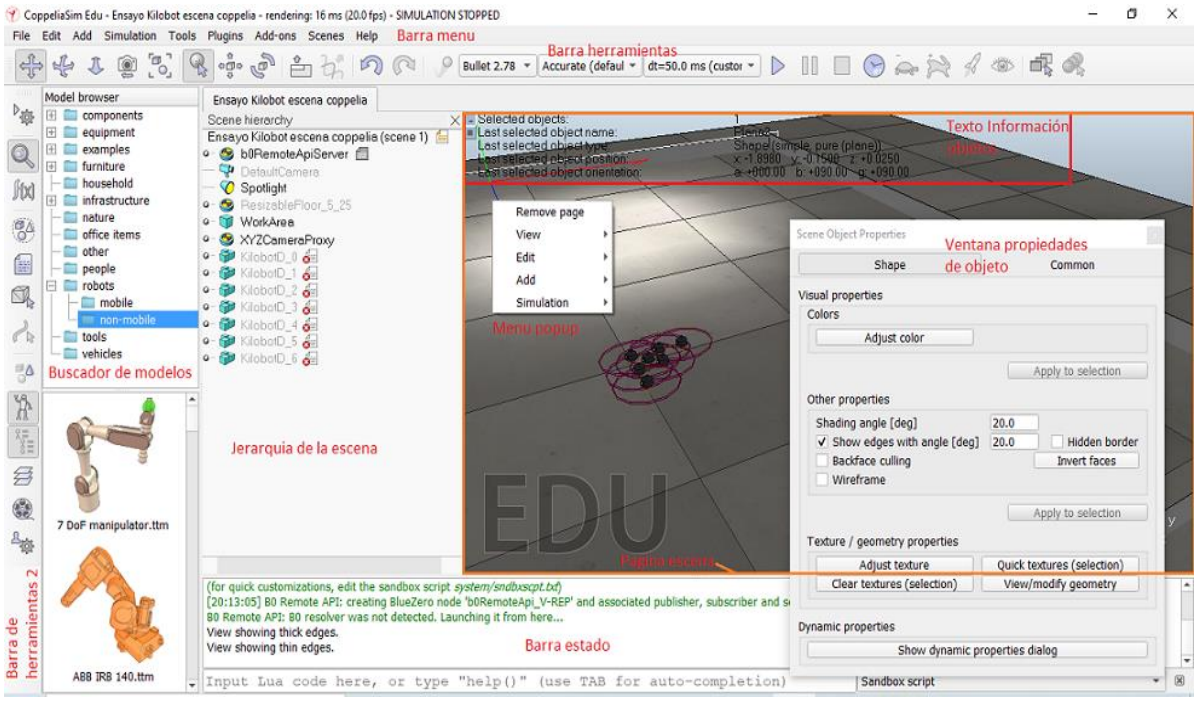


Figura 2.7. Escena de CoppeliaSim

- **Barra menú:** Esta barra contiene todas las funcionalidades, información y herramientas de la simulación. Las principales funciones son para crear, guardar o cargar escenas, herramientas de la simulación, añadir/modificar objetos o formas y menú de ayuda.

- **Barra de herramientas:** Contiene variedad de herramientas para la gestión de la simulación, para el movimiento de los objetos y gestión de los motores físicos véase figura 2.8.

Figura 2.8. Barra de herramientas

- **Barra de herramientas 2:** Contiene herramientas para editar objetos, visualización de elementos, grabar escenas y configuraciones de usuario y simulación, véase figura 2.9:

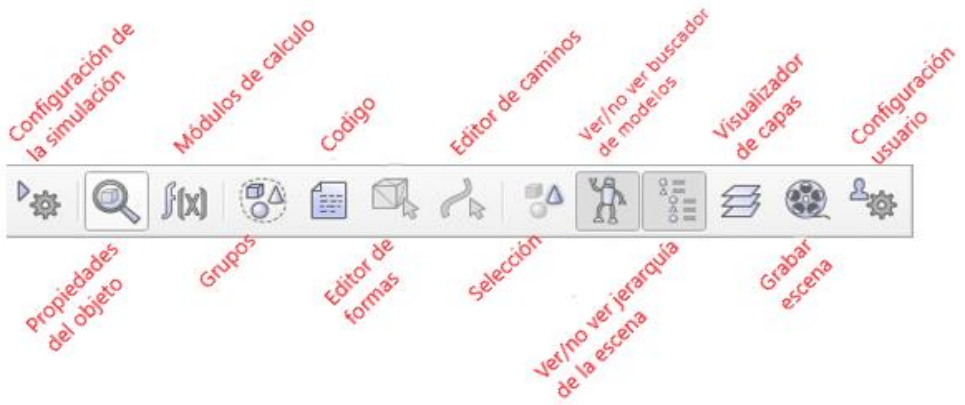


Figura 2.9. Barra de herramientas 2

- **Buscador de modelos:** Herramienta para buscar modelos ya preparados en CoppeliaSim. CoppeliaSim incluye variedad de formas, objetos, sensores, robots y una amplia variedad de componentes para realizar una simulación sin la necesidad de diseñar todos los objetos.

- **Jerarquía de escena:** Con esta ventana se puede visualizar la jerarquía que tiene un objeto. Los objetos en CoppeliaSim pueden estar ordenados jerárquicamente como objetos padre o hijos, los hijos son los que están ligados a un objeto padre creando una dependencia con éste y estos objetos padres pueden ser también objetos hijo al estar ligados a otros objetos padre creando así una jerarquía. Ésto es útil para crear grupos y darle cualidades al grupo entero o solo a objetos puntuales dependiendo de si se elige a un objeto padre con sus objetos hijos como grupo o a un solo objeto hijo en la figura 2.10 muestra un ejemplo de una jerarquía de escena más detalladamente.

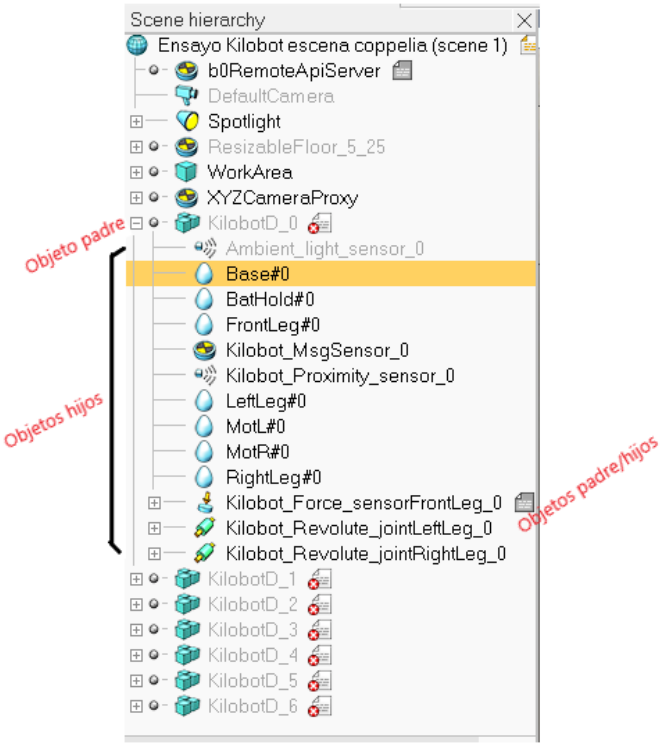


Figura 2.10. Jerarquía de escena

- **Barra de estado:** Ventana para visualizar todas las acciones que el usuario hace en la escena además de los errores que puedan ocurrir en esta.

- **Menú popup (desplegable):** Este menú desplegable se muestra cuando se hace clic con el botón derecho, dependiendo de en la ventana que se presione muestra una variedad de herramientas vinculadas a esa ventana.

- **Texto información de objetos:** Muestra información del ultimo objeto seleccionado en la escena como su posición, orientación, nombre o tipo de objeto.

- **Ventana propiedades de objeto:** Ventana importante para configurar el comportamiento de cualquier objeto, se puede acceder de dos formas seleccionándola en la barra de menú o haciendo clic dos veces en la imagen del objeto de la jerarquía de escena. En la figura 2.11 se muestra las diferentes formas de acceder a la ventana de propiedades de objeto.

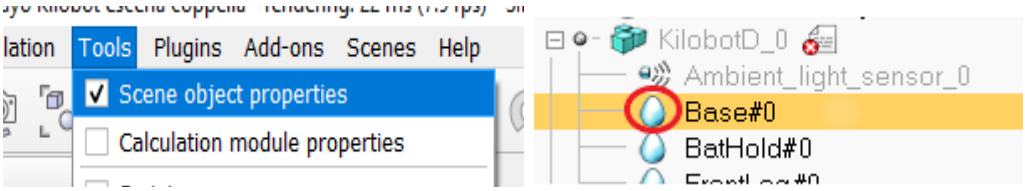


Figura 2.11. Desplegar ventana de propiedades de objeto

- **Página de escena:** Página para la visualización real de la escena. Esta página es la que se graba si se desea grabar la simulación de la escena.

2.2.3. Objetos de escena

Para la construcción de la simulación se usan principalmente los objetos de escena. Cuando se genera uno de estos objetos aparece tanto en la página de escena como en la jerarquía. Los objetos de escena pueden ser de varios tipos, pero los usados principalmente en este TFG son los de la figura 2.12:

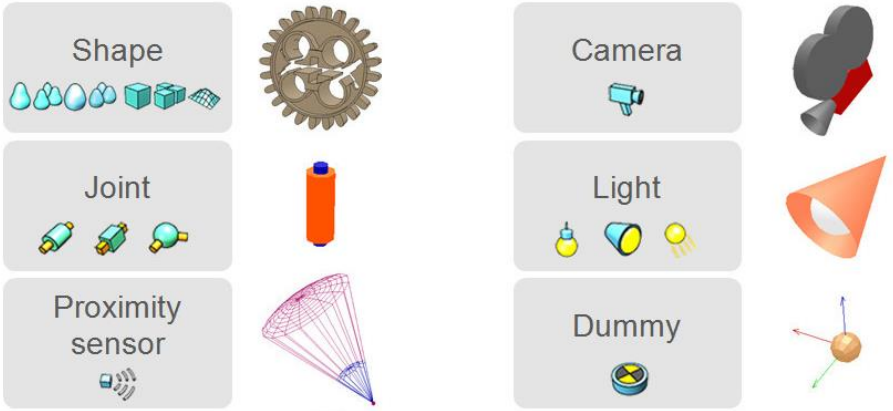


Figura 2.12. Principales objetos de escena
Fuente: **[25]**

- **Shape(Formas)**: Las formas son un objeto solido compuesto por caras triangulares. A estos objetos se le puede dar configuraciones dinámicas accediendo a la ventana de **propiedades del objeto**, además de esta configuración se le puede cambiar varios atributos como cambios en la visualización del objeto o definir a la capa a la que pertenece. Estos objetos pueden ordenarse jerárquicamente para hacer que unos dependan de otros y crear objetos más complejos como robots.

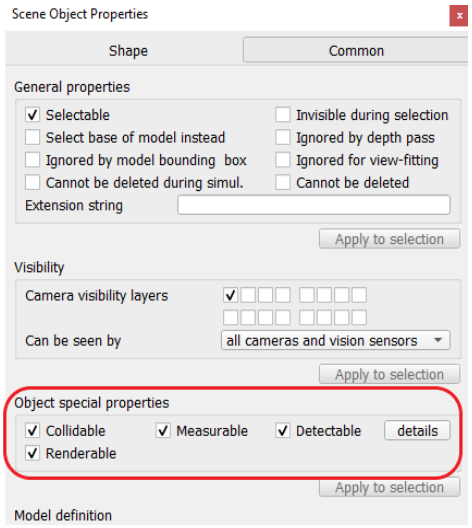


Figura 2.13. Ventana de propiedades de objeto

Los objetos *formas* como muestra la figura 2.13 pueden ser capaces de colisionar (**Collidable**), medibles (**Measurable**), detectables (**Detectable**) y capaces de renderizarse (**Renderable**).

Dependiendo lo que se marque en la ventana, los objetos interactúan de forma diferente con el entorno que les rodea.

- **Collidable**: Al marcar esta casilla se le atribuye la capacidad de poder colisionar con otros objetos con este mismo atributo.

- **Measurable**: Cuando se quiera hacer uso de una función en la que se calcule la distancia entre dos objetos estos dos objetos deben tener marcada esta casilla indicando que son medibles.

- **Detectable**: Para que un sensor de proximidad pueda detectar una forma esta debe tener marcada esta casilla.

- **Renderable**: Esta no se tendrá en cuenta en este TFG ya que se utiliza cuando se desea que un objeto se pueda ver por un objeto escena del tipo sensor de visión.

Además de todas estas características a los objetos forma se le pueden dar cualidades dinámicas marcando la casilla de la figura 2.14:

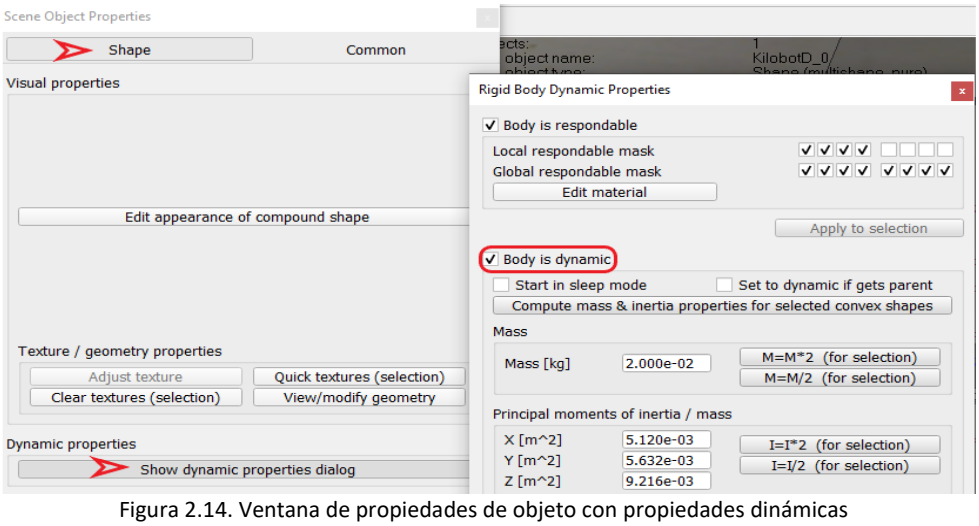


Figura 2.14. Ventana de propiedades de objeto con propiedades dinámicas

Marcando la casilla "**Body is dynamic**" se le atribuye al objeto cualidades dinámicas, estas cualidades se pueden cambiar en esta misma ventana cualidades como su peso o su momento de inercia. Para que a un objeto le afecte la gravedad tiene que ser dinámico, esta capacidad se puede aprovechar para crear un objeto flotante desmarcando esta casilla.

- **Cámara (Camera):** Una cámara es un objeto que permite ver la simulación desde diferentes puntos de vista o hacer que esta cámara siga a una determinada forma para poder ver su comportamiento en simulación sin necesidad de seguirlo manualmente.

- **Articulación (Joint):** Con las articulaciones se puede dotar a un objeto de movimiento, esto se consigue dándole a un objeto padre una articulación que a la vez tiene un objeto hijo, este objeto hijo es el que se mueve y estará vinculado a esta articulación junto con su objeto padre.

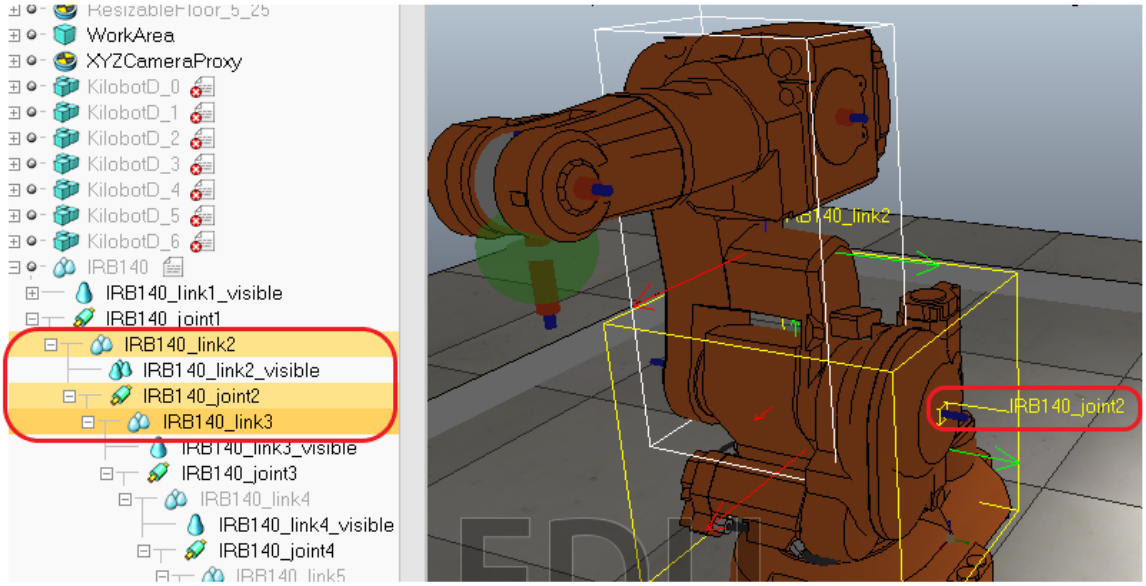


Figura 2.15. Imagen de una articulación del robot IRB 140 de ABB y su jerarquía

En la figura 2.15 se puede ver como a partir de la jerarquía se puede crear una articulación entre dos objetos. Los tipos de articulación son de revolución, prismática y esférica. Se puede definir su comportamiento con varios modos, pero el más importante es el modo pasivo que es el que hace que se pueda gobernar su comportamiento a través de los scripts y con funciones.

- **Luz (Light):** Las luces son objetos que permiten visualizar la escena iluminándola. Hay varios tipos de luz, pero se utiliza el foco para iluminar una parte de la escena ya que el caso de ensayo con Kilobots reales se realiza con un foco.

- **Sensor de proximidad (Proximity sensor):** Con este objeto se permite detectar otros objetos que tengan la característica de ser detectables (detectable). Para incluir un sensor a un objeto se le añade como un objeto hijo y se selecciona el tipo y rango de detección que tiene.

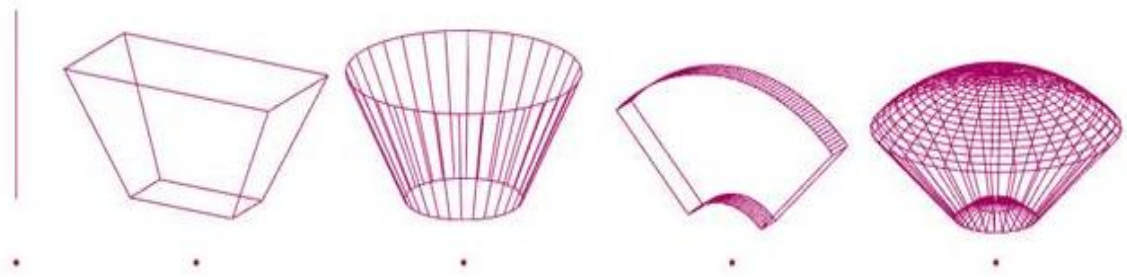


Figura 2.16. Tipos de sensores de proximidad. De izquierda a derecha tipo rayo, pirámide, cilindro, disco y cono. Fuente: **[25]**

Los tipos que se utilizan son del tipo disco y cono como muestra la figura 2.16. En la figura 2.17 se muestra la ventana para definir su espacio de detección.

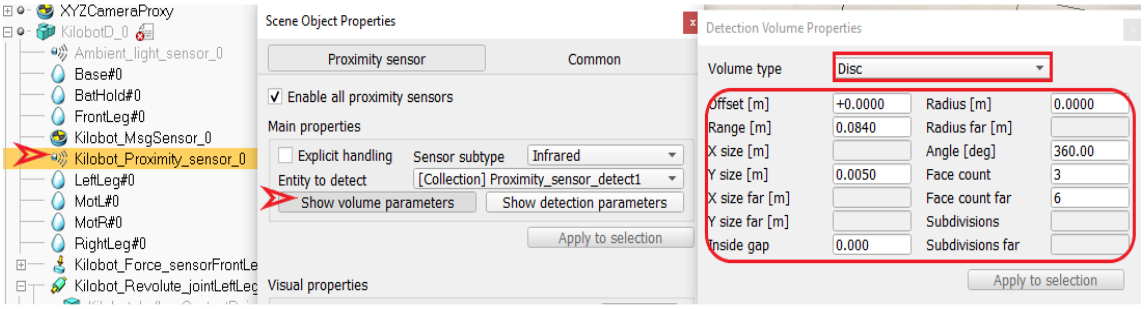


Figura 2.17. Ventana de propiedades de un sensor

- **Objeto ficticio (Dummy):** Es el objeto más simple que se encuentra en CoppeliaSim y su uso normalmente es para tomarlo como una referencia dándole al objeto al que se le vincula como hijo un apoyo ya que se le puede añadir distintos propósitos al del objeto al que está vinculado. Los objetos ficticios pueden tener la capacidad de colisionar, ser medibles y detectables, pero no visibles por sensores de visión.

2.2.4. Kilobots simulados

El robot Kilobot de CoppeliaSim tiene varias características y diferencias con el robot real, para añadir un Kilobot a la escena basta con buscarlo en el **model browser** y añadirlo arrastrándolo. Mirando su árbol jerárquico se pueden ver sus cualidades físicas y observando su código por defecto la manera que tiene de funcionar.

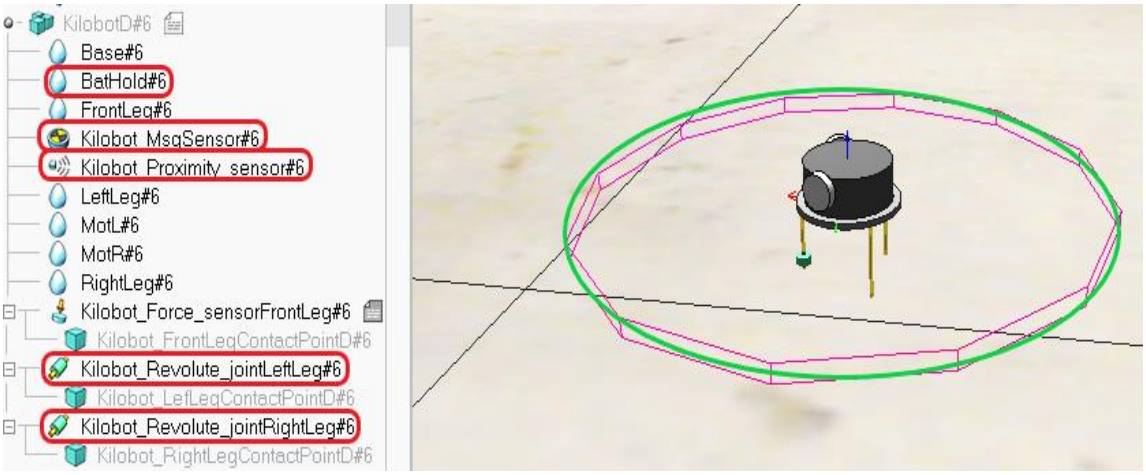


Figura 2.18. Jerarquía e imagen de un Kilobot en CoppeliaSim

Los objetos hijos por destacar son:

- **`BatHold`**: Objeto forma que además de actuar como el cuerpo del Kilobot funciona también como un led RGB ya que el código del Kilobot contiene una función para cambiar el color de este objeto. El problema de esta función es que no tiene una función análoga en la **API B0-based remoted** para cambiar el color por lo que no se ha usado esta funcionalidad del Kilobot.

- **`Kilobot_Revolute_jointLeftLeg/RightLeg`:** Estos objetos articulación son los que hacen que un Kilobot simulado se mueva igual que uno real. Se mueven imitando una vibración de los objetos hijos que tienen vinculado. Estos robots simulados se pueden mover a una velocidad muy superior a la de un Kilobot real.

- **`Kilobot_Proximity_sensor`:** El Kilobot real no tiene un sensor de proximidad verdaderamente, pero para que un Kilobot en simulación pueda comunicarse con otro, tiene que definirse el rango con el que se puede comunicar con otro Kilobot. Para conseguir esto se hace uso de este sensor que funciona como referencia del espacio (área verde de la figura 2.18) con el que se pueden comunicar dos Kilobots, solo pudiendo existir interacción entre los Kilobots dentro de este espacio.

- **`Kilobot_MsgSensor`:** Este objeto ficticio es la referencia que hace que lo detecte el sensor de proximidad de otro Kilobot.

La principal desventaja del Kilobot de CoppeliaSim es que no tiene un sensor de luz como el Kilobot real, pero se puede solucionar añadiéndole un sensor tal y cómo se explica en capítulos posteriores. Otra de las diferencias es la de movimiento, mientras que un Kilobot real solo puede mover una u otra patilla, el Kilobot de CoppeliaSim puede mover las dos para crear movimientos diferentes.

2.3. Python

Python es un lenguaje de alto nivel de propósito general ya que se puede utilizar tanto para programación de PCB como para programación de complejos programas en diferentes plataformas. Las principales ventajas que lo hacen tan práctico y famoso es su flexibilidad, sencillez y su facilidad de aprendizaje Es *Open Source* y por tanto puede ser usado libremente por cualquier persona o empresa **[28]**.

Las principales ventajas del uso de Python son **[29]**:

- **Simplicidad y rapidez:** Al ser un lenguaje muy simple no requiere de muchas líneas para la programación de una función.

- **Lenguaje flexible:** Tiene variedad de herramientas para hacer una programación cómoda, ya que es capaz de detectar el tipo de dato de una variable sin necesidad de especificar el tipo de dato a diferencia de otros lenguajes.

- **Ordenado:** Es un lenguaje muy limpio y fácil de entender.

- **Multiplataforma:** Python es admitido en casi todos los sistemas operativos que existen y además contiene una amplia gama de librerías para apoyar cada vez más su funcionalidad.

- **Multiparadigma:** Esto hace que pueda soportar varios paradigmas como la Programación Orientada a Objetos (POO), estructurada, imperativa y funcional. Esta característica es importante ya que la programación en CoppeliaSim se hace con POO.

- **Comunidad:** La comunidad detrás de Python es la ventaja que más destaca, ya que ésta se encarga de actualizar y mejorar las librerías y variedad de foros para proporcionar ayuda a los nuevos programadores.

El principal defecto de elegir este lenguaje es que los Kilobots reales se programan en lenguaje C, pero extrapolarlo será sencillo ya que el lenguaje es fácil de entender y servirá de guía para la programación en C.

2.3.1. Anaconda

Anaconda es un software de código abierto que contiene más de 7500 paquetes de código abierto, que contienen diferentes librerías y entornos de programación (IDE). Contiene una interfaz de usuario (GUI) llamada **AnacondaNavigator**, que es una herramienta que ayuda a gestionar los entornos de programación con las diferentes librerías de una forma rápida y sencilla **[30]**.

Las principales ventajas que tiene son:

- Es de código abierto y tiene un manual muy detallado además de una gran comunidad.
- Es multiplataforma, ya que permite ser instalado en los sistemas operativos más usados (Windows, Linux y MacOS).
- Contiene varios IDE como RStudio, PyCharm Professional, Spyder, JupyterLab y Jupyter. En este último es en el que se ha desarrollado la programación de CoppeliaSim.
- Su GUI permite una fácil gestión de los paquetes, manteniéndolos actualizados e instalando los necesarios en cada entorno.
- Gran variedad de paquetes centrados en el *machine learning*.
- Los proyectos que se realizan con anaconda son portables por lo que se pueden compartir de forma fácil además de poder ejecutarse paralelamente junto con otros proyectos.

La instalación de Anaconda y configuración de los paquetes utilizados en este TFG se muestra en el Anexo I.

2.3.2. Programación Orientada a Objetos (POO)

La POO es un paradigma de programación que hace uso de las clases y objetos para realizar una programación sencilla y reutilizable. Este concepto de programación está centrado en los objetos lo que lo hace ideal para los simuladores con sistemas robotizados de enjambres ya que se toma a cada individuo como un objeto y así se puede reutilizar el código con todo el enjambre.

La programación está basada en varias técnicas **[31]**, siendo las principales:

- **Encapsulamiento**: Toda la información importante la contiene el objeto y solo se puede acceder a ella con métodos asegurando que no se pueda cambiar desde fuera.
- **Abstracción**: Es la forma que tiene el usuario de interactuar con el objeto mediante los atributos o métodos. Se basa en construir una variedad de códigos simples que interactuando entre ellos representa un código más complejo.

- **Herencia**: Al igual que en CoppeliaSim se usa una jerarquía. Las clases son una jerarquía en la que un objeto contiene atributos comunes con otros objetos de la misma clase permitiendo reutilizar el código.

- **Polimorfismo:** Es la capacidad de compartir un comportamiento entre objetos, que, aunque compartan un método hace que cada objeto lo pueda usar de diferente modo.

Para la POO se usa una sintaxis diferente en cada lenguaje de programación, la sintaxis básica utilizada en Python es:

- **`class Clase`**: Para definir una clase en la que se define los Kilobots, por ejemplo.

- **`def __init__(Atributo1, Atributo2…):`** Esto es un constructor, que genera los atributos introducidos cuando se crea un objeto de la clase que contiene este constructor.

- **`def Método`**: Esta es la sintaxis para generar un método, un método es como una función que puede llevar a cabo el objeto clase que la contiene.

- **`self.Atributo/Método`**: Para llamar a un método o un atributo dentro de un método se escribe con esta sintaxis, también se usa para crear atributos dentro de la clase.

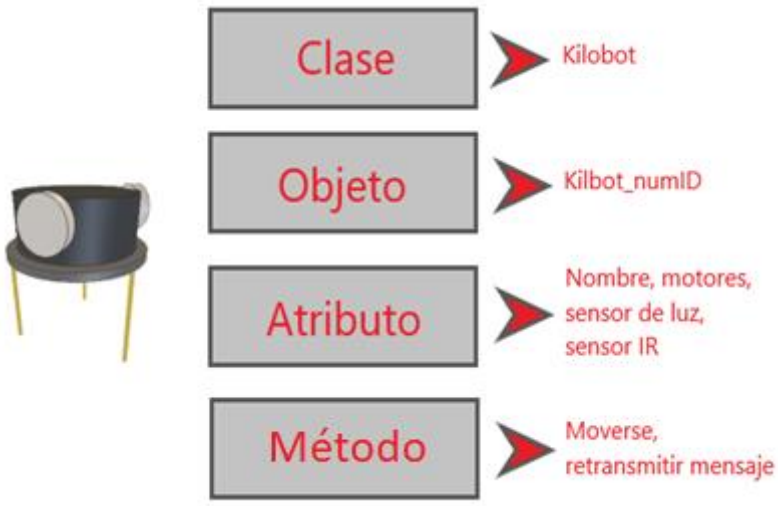


Figura 2.19. Ilustración de POO en un Kilobot

En la figura 2.19 se puede ver cómo se construiría una clase Kilobot con su diferentes atributos y métodos. Los objetos serían los diferentes robots del enjambre que se identificarían por su número de identificación. Gracias a la POO, programando un solo Kilobot se genera un código que se podría reutilizar para todos los Kilobots del enjambre cambiando únicamente el valor de los parámetros del constructor.

2.3.3. Librerías

Las librerías usadas en Python son:

- **`random`**: Esta es la librería estándar que posee Python para generar valores aleatorios **[32]**.

- **`numpy`:** Es un paquete para el procesamiento de datos en forma de matriz **[33]**, aunque su uso principal en este TFG es para las funciones aleatorias:

- **`numpy.random.uniform`**: Para generar una distribución aleatoria uniforme.
- **`numpy.random.normal`:** Permite usar una función de distribución normal.

- **`debugger`:** De esta librería se usa la función **`Tracer`** para la depuración del código y la comprobación de su correcto funcionamiento **[34]**

2.4. Eclipse

Eclipse es un IDE diseñado por IBM y programado en Java que proporciona una variedad de herramientas para la programación en C y C++ **[39]**. En este apartado se muestran sus principales características y herramientas.

La programación de Eclipse es en código abierto lo que permite que trabaje con colecciones de complementos o herramientas, por lo que se pueden seleccionar diferentes complementos necesarios en cada proyecto. Para la programación de los Kilobots se usará la del complemento ***winAVR*** que es un *plug-in* para la programación de microcontroladores Atmel.

Los proyectos se pueden crear desde cero o cargar uno ya existente exportado desde otro usuario, la principal vista cuando se carga un proyecto se llama banco de trabajo y se encuentra dividida en seis secciones, véase figura 2.20:

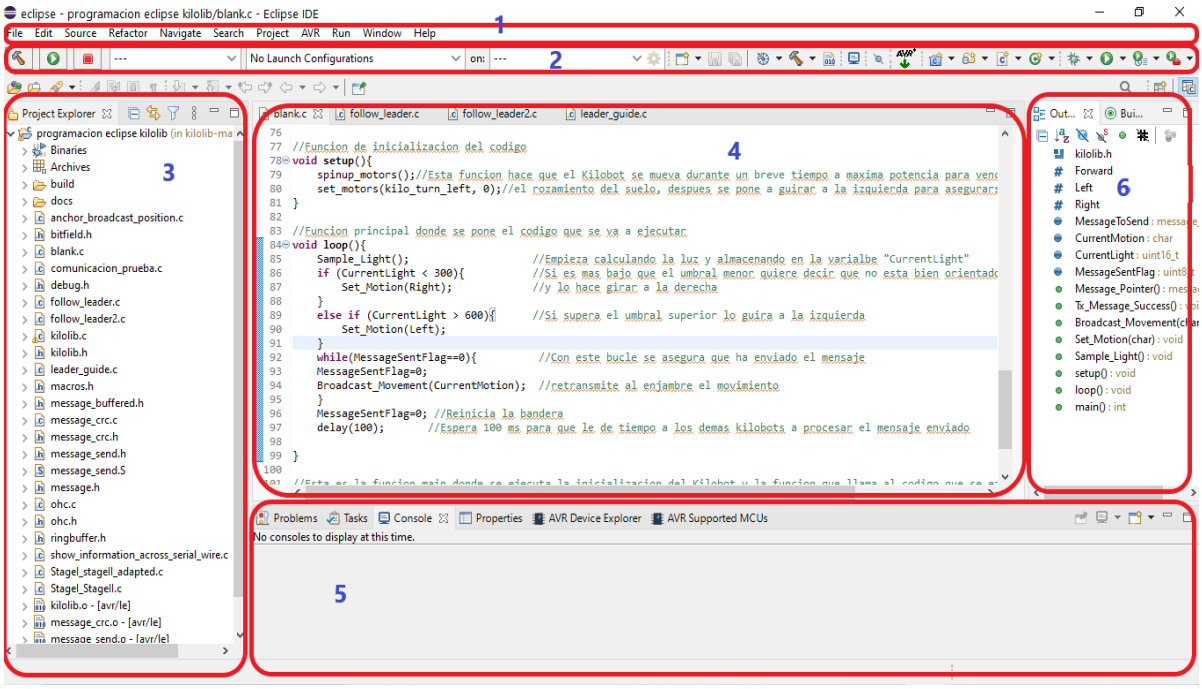


Figura 2.20. Ventana banco de trabajo de Eclipse

1- **Barra menú:** Proporciona acceso a todos los complementos, configuraciones y herramientas de Eclipse como gestión de proyectos, configuración del ensamblador, herramientas de edición de código, herramientas de gestión de complementos y menú de ayuda.

2- **Barra de herramientas:** Es una barra de herramientas personalizable donde se le pueden agregar y quitar las funciones que el usuario considere oportuno, normalmente tiene herramientas vinculadas con el ensamblador, gestión de proyectos y herramientas de complementos.

3- **Explorador del proyecto:** Ventana para la gestión de los archivos generados en cada proyecto.

4- **Editor:** Eclipse proporciona un editor para que el usuario cree su propio código, las principales características es un sistema de ayuda para el caso de errores en la sintaxis del código y resaltado para diferenciar los diferentes elementos en el código.

5- **Marcadores:** Tiene dos ventanas para la inclusión de diferentes marcadores asignados por el usuario. Estos marcadores pueden ser una consola, visor de errores, vista de variables en el código o marcadores de complementos incluidos por el usuario.

6- **Marcadores 2:** Esta es la segunda ventana de marcadores, tiene las mismas funciones que la primera pero en diferente ubicación en el banco de trabajo.

2.4.1. WinAVR

Los Kilobots admiten archivos de extensión *"**hex**"* que son archivos en hexadecimal, para generar estos tipos de archivos se hace la programación en C en el IDE ***Eclipse*** del código que se desea cargar en el Kilobot y con la ayuda de los complementos proporcionados por ***WinAVR*** se generan estos archivos. ***Eclipse*** no contiene estas herramientas en su aplicación por defecto por eso es necesario el uso de ***WinAVR*** que es un instalador de los complementos necesarios para generar códigos legibles para un microcontrolador Atmel.

WinAVR proporciona una serie de herramientas para la programación de sistemas embebidos con microcontroladores Atmel, sus principales herramientas son **[40][41]**:

- **avr-gcc:** Es el compilador de AVR y la herramienta principal para la creación, simulación y comprobación de códigos para microcontroladores.
- **avr-libc:** Librería de apoyo para **avr-gcc** que permite la programación en C de microcontroladores Atmel de 8 bits.
- **avr-as:** Es el ensamblador y el encargado de pasar el código C al código maquina (genera el archivo *hex).*
- **avrdude:** Es una herramienta para la manipulación de la ROM y la EEPROM de un microcontrolador.
- **avr-gdb:** Contiene las herramientas necesarias para la depuración de códigos.

La configuración de *Eclipse* para que utilice estos complementos y la instalación de *WinAVR* se ha incluido en el **Anexo I.**

2.5. Lenguaje C

El lenguaje C es un lenguaje de medio nivel creado en los 70 por Brian Kernighan y Dennis Ritchie, se diseñó este lenguaje porque los que había por aquel entonces eran lenguajes complejos en los que resultaba difícil de comprender el enmarañado código creado, lo que complicaba la compresión del mismo por parte de otro usuario que no fuese su creador. Este lenguaje ofrece flexibilidad y baja comprobación de errores (como el tamaño de los vectores o si una variable pertenece al área de datos al que está referida) **[41]**.

Este lenguaje es el precursor del uso de una sintaxis que está basada en bucles, condicionales y funciones alrededor de una función `main` principal. El lenguaje C tiene una amplia variedad de librerías para todo tipo de usos, pero las usadas en este TFG son las necesarias para la programación de los

Kilobots y que se explican a continuación. Estas librerías se pueden descargar en la página de Kilobotics **[38].**

2.5.1. Librería `Kilolib`

Esta librería proporciona las funciones y estructuras de datos necesarias para la lectura de los sensores de los Kilobots y para interactuar con sus diferentes periféricos, como el transmisor y receptor de IR, el control de los motores vibratorios para su movimiento o el LED RGB. Antes de entrar en detalle en las funciones hay que explicar las estructuras de datos y las variables utilizadas ya que son un factor importante para tener en cuenta, ya que el Kilobot solo admite 32Kbytes de código, siendo muy importante, por lo tanto, ahorrar al máximo en espacio. Por esta razón los tipos de variables se tienen que elegir de la mejor forma posible.

Los tipos de declaraciones de variables disponibles son **[42]**:

- **`uint8_t`:** Son variables enteras de 8 bits sin signo por lo que su rango de valores es de 0 a 255.
- **`uint16_t`:** Variable entera de 16 bits sin signo con rango de valores de 0 a $(2^{16} - 1)$ que es un valor máximo de 65535.
- **`int8_t`:** La diferencia con **`uint8_t`** es que tiene signo y su rango es de -127 a 127.
- **`int16_t`:** Al igual que la anterior es con signo y 16 bits con rango de -32767 a 32767.
- **`message_t`:** Cuando se genera una variable de este tipo se crea una estructura de datos de 12 bytes para el envío/recepción de mensajes que está dividido en 3 campos:

  - **`data[9]`:** Un vector **`uint8_t`** de 9 elementos por lo que ocupa 9 bytes y su uso es para guardar datos definidos por el usuario.

  - **`type`:** Describe el tipo de mensaje ocupa 1 byte de tipo **`uint8_t`**.

  - **CRC:** Se usa en la función **`message_crc()`** para comprobar si un mensaje es correcto o no, es del tipo **`uint16_t`** por lo que ocupa 2 bytes.

- **`distance_measurement_t`:** Esta variable se utiliza en la función de recepción de mensajes. Cada vez que un Kilobot recibe un mensaje de otro Kilobot toma dos medidas que representan la intensidad de la señal del mensaje y en base a estas dos medidas se puede estimar la distancia a la que se encuentra el remitente.

Las funciones que se utilizan **[42]**:

- **`delay(uint16_t tiempo)`:** Pausa la ejecución del programa durante un tiempo indicado en milisegundo en el parámetro ***`tiempo`*** introducida por el usuario**.** Es una función vacía por lo que no devuelve nada.
- **`estimate_distance(distance_measurement_t distancia)`:** Estima la distancia en milímetros de una señal introducida **distancia** proveniente de la recepción de un mensaje. Devuelve un **`uint8_t`** con la distancia a la que se encuentra el remitente del mensaje.
- **`get_ambientlight()`:** Cada vez que se llama a esta función devuelve la lectura del sensor de luz. El sensor está conectado a uno de los convertidores del procesador, estos puertos son de 10

bits por lo que su rango de valores es desde -1 (cuando el convertidor no está disponible) hasta $(2^{10} - 1)$ que es 1023.

- **`kilo_init()`:** Función para inicializar el hardware del Kilobot.

- **`kilo_start(setup(),loop())`**: Provoca que el Kilobot entre en bucle de la forma que se ejecute una vez la función introducida **`setup()`** y seguidamente una continua ejecución de la función **`loop()`**. Usando el OHC es posible parar o detener este bucle.

- **`message_crc(message_t mensaje)`** : Función que calcula el **CRC** de un mensaje. Devuelve un valor **`uint16_t`** que se tiene que almacenar en el elemento **CRC** de la variable ***`mensaje`***, una vez hecho esto el mensaje está listo para ser enviado. Esta función comprueba la validez de un mensaje en el caso de que se corrompa el mensaje el receptor del mensaje lo descarta dependiendo del valor del **CRC**.

- **`set_motors(uint8_t izquierda,uint8_t derecha)`** : Ajusta la vibración de los motores del Kilobot. Tiene dos parámetros de entrada un valor ***`izquierda`*** que establece la vibración del motor izquierdo del Kilobot y otro valor ***`derecha`*** para el motor derecho. Normalmente los valores de ***`izquierda`*** y ***`derecha`*** suelen ser los valores de calibración asignados.

- **`spinup_motors()`** : Activa los motores a máxima potencia durante 15 milisegundos para vencer el rozamiento del suelo. Es conveniente llamarlo antes de una función de movimiento del Kilobot.

Otras variables definidas para usar en el Kilobot:

- **`kilo_message_rx`**: Devolución que se activa cuando llega un mensaje valido. Recibe dos parámetros que es una función con un puntero donde almacenar el mensaje y otro puntero donde almacenar la distancia medida de la señal. Se tiene que registrar después de la función **`kilo_init()`**.

- **`kilo_message_tx`**: Es análoga a **`kilo_message_rx`** pero se activa con los mensajes enviados.

- **`kilo_message_tx_success`:** Devolución para la comprobación del correcto envío de mensajes.

- **`kilo_turn_left`:** Variable donde esta almacenado el valor calibrado del giro a izquierdas, es del tipo **`uint8_t`**.

- **`kilo_turn_right`:** Variable donde esta almacenado el valor calibrado del giro a izquierdas, es del tipo **`uint8_t`**.

- **`kilo_uid`:** Variable donde esta almacenado el valor del número de identificación asignado en la calibración, es del tipo **`uint16_t`**.

2.5.2. Librería debug

Librería para activar el modo de depuración del código del Kilobot real. Se usa principalmente para ver los valores de las variables de un Kilobot que esté conectado a través del cable serie con el OHC.

Esta librería solo contiene la función **`debug_init(),`** encargada de inicializar el hardware para la comunicación serie del Kilobot. Un ejemplo de programación para activar el modo depuración se muestra en la figura 2.21:

```
#include <kilolib.h>
#define DEBUG       //se necesita definir la variable DEBUG para habilitar el modo debug
#include <debug.h>  //Inclusión de la librería
void setup() {
}

void loop() {
    printf("%d\n", kilo_uid); //Se usa printf para enviar por el serial el ID del kilobot
}
int main() {
    kilo_init();
    debug_init();             //Funcion para inicializar el hardware del kilobot para el modo debug
    kilo_start(setup, loop);
    return 0;
}
```

Figura 2.21. Código de ejemplo para el uso del modo debug.

En la figura se muestra como inicialmente se define la variable **`DEBUG`** esto se hace para la comunicación serie y debe ser definida antes de la inclusión de la librería **`debug.h`**. Para enviar información al ordenador se usa **`printf()`** con la información a mostrar en la ventana **Serial input** de la aplicación **KiloGUI** explicada anteriormente.

2.6. Algoritmo "distributed and resilient localization"

La localización de los robots es un factor importante para tener en cuenta en el mundo de la robótica móvil, ya que se debe tener conciencia de donde se encuentran los robots en todo momento para tener una idea del comportamiento que está teniendo el propio robot y no se produzca ningún comportamiento indeseado, fallas o accidentes. Muchas aplicaciones requieren que los robots sepan su propia posición por lo que haciendo uso del método "***distributed and resilient localization***" **[23]** un robot con limitaciones sensoriales puede saber su propia posición. Este método se divide en varias etapas que utiliza una variedad de algoritmos empíricos y probabilísticos para poder saber su propia posición en base a una referencia **[23]**. Se ha escogido este algoritmo puesto que lo único que se necesita para su funcionamiento es un robot con la capacidad de mandar/recibir mensajes y saber la distancia a la que se encuentra el remitente de estos mensajes.

Para el uso del método "***distributed and resilient localization***" es necesario que el enjambre este dividido entre robots denominados anclas y los robots nodos. Los *anclas* conocen su propia posición y son la referencia de los robots nodos que son los que obtendrán su propia posición en base a los robots anclas. Para que se lleve a cabo esto, el algoritmo está dividido en varias etapas por lo que procederá a explicar cada una de ellas:

2.6.1. Algoritmo Sum-Dist (Etapa I)

En la primera etapa se hace uso del algoritmo **Sum-Dist [23]**, que utiliza la capacidad de un robot de enviar/recibir mensajes con sus vecinos para determinar la distancia que han recorrido los mensajes enviados por el enjambre. Esta información se usará posteriormente en las siguientes etapas.

Los que inician el proceso son los robots anclas retransmitiendo en un mensaje su identificación, su posición en coordenadas cartesianas y la distancia que ha recorrido ese mensaje (al ser los que inician el proceso esta distancia es cero). Los mensajes llegan a los robots nodos que almacenan en una lista todos los mensajes anclas (actualizando en esta lista los mensajes del ancla si es necesario ya que se escogen los mensajes que hayan recorrido menos distancia) y después replican el mensaje a sus vecinos, pero esta vez añadiéndole la distancia a la que se encontraba el remitente del mensaje.

Este algoritmo se ejecuta hasta un tiempo ***Ts***, dependiendo su velocidad de ejecución de este tiempo. Este algoritmo no tiene mucha carga computacional ya que es fácil por lo que el tiempo empleado no viene limitado por complejos cálculos o iteraciones **[23]**. El tiempo seleccionado no debe ser bajo para proporcionarle tiempo a todos los robots nodos a intercambiarse la información de todos los robots anclas con la distancia de sus mensajes, pero tampoco puede ser un tiempo elevado para que no emplee un tiempo excesivo en ejecutar este algoritmo. El algoritmo completo viene descrito en la figura 2.22:

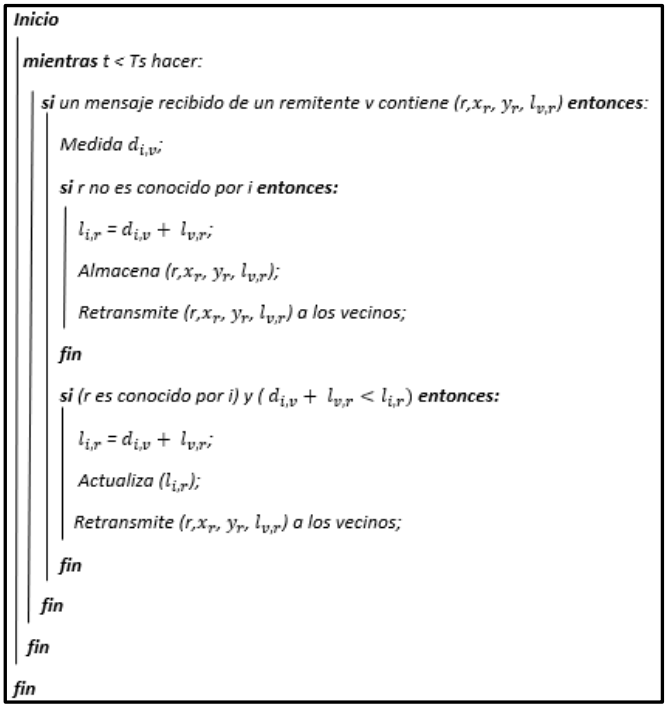


Figura 2.22. Algoritmo Sum-dist

2.6.2. Método Min-Max (Etapa II)

Para una estimación inicial de la posición se hace uso del método **Min-Max**, este método se realiza en tres partes:

- Los robots nodos calculan su cuadrado delimitador $\boldsymbol{B_{i,r}}$ usando la información obtenida de las distancias y posiciones de los robots anclas. El cuadro delimitador es el área donde se puede encontrar el robot nodo respecto del ancla r del que se calcule este cuadrado y se calcula con la ecuación (2.1):

$$B_{i,r}: [x_r - l_{i,r}, y_r - l_{i,r}] \times [x_r + l_{i,r}, y_r + l_{i,r}] \tag{2.1}$$

- Una vez calculado todos los cuadros delimitadores, se halla la intersección de todos estos. Esta intersección se le llama región $\boldsymbol{S_i}$ y en ella se encuentra la posición del robot nodo, esta región se calcula a partir de la ecuación (2.2):

$$S_i = [\max(x_r - l_{i,r}), \max(y_r - l_{i,r})] \times [\min(x_r + l_{i,r}), \min(y_r + l_{i,r})] \tag{2.2}$$

  Donde $(\max(x_r - l_{i,r}), \max(y_r - l_{i,r}))$ se calcula como el máximo de todas las primeras coordenadas de los $\boldsymbol{B_{i,r}}$ calculados en la primera parte y $(\min(x_r + l_{i,r}), \min(y_r + l_{i,r}))$ se calcula con las segundas coordenadas.

- Una vez se ha obtenido $\boldsymbol{S_i}$ la estimación del punto es el centro de la región que se calcula a partir de las ecuaciones (2.3):

$$x_i = \frac{S_{i,x_{max}} + S_{i,x_{min}}}{2} \qquad y_i = \frac{S_{i,y_{max}} + S_{i,y_{min}}}{2} \tag{2.3}$$

En la figura 2.23. se muestra el método **Sum-Dist** de un robot nodo $\boldsymbol{u_i}$ junto a tres robots anclas $\boldsymbol{r_1}, \boldsymbol{r_2}$ y $\boldsymbol{r_3}$, se puede ver como se crea un cuadrado $\boldsymbol{S_i}$ a partir de las distancias a las que se encuentran los robots anclas. La distancia a la que se encuentran cada nodo crea un cuadrado y la intersección de estos tres cuadrados crean el cuadrado $\boldsymbol{S_i}$ que es la estimación de la posición del robot nodo $\boldsymbol{u_i}$.

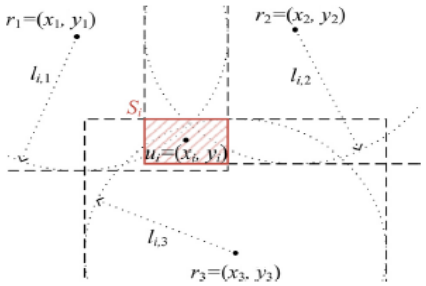


Figura 2.23. Ilustración gráfica del método Min-Max
Fuente: **[23]**

2.6.3. Multi-hop Collaborative Min-Max (MCMM) (Etapa III)

Como se ha visto en la etapa 1 se hace uso de un tiempo ***Ts*** de ejecución del algoritmo **Sum-Dist**, lo cual puede ocasionar que el enjambre no tenga el suficiente tiempo para compartir la información de todos los robots anclas. Por lo tanto, este problema puede llevar a errores en el cálculo de la posición de un robot nodo en la etapa 2 ya que se necesitan por lo menos la información de 3 robots anclas para poder calcular una posición óptima. La figura 2.24 muestra el área que genera teniendo la información de diversos robots anclas:

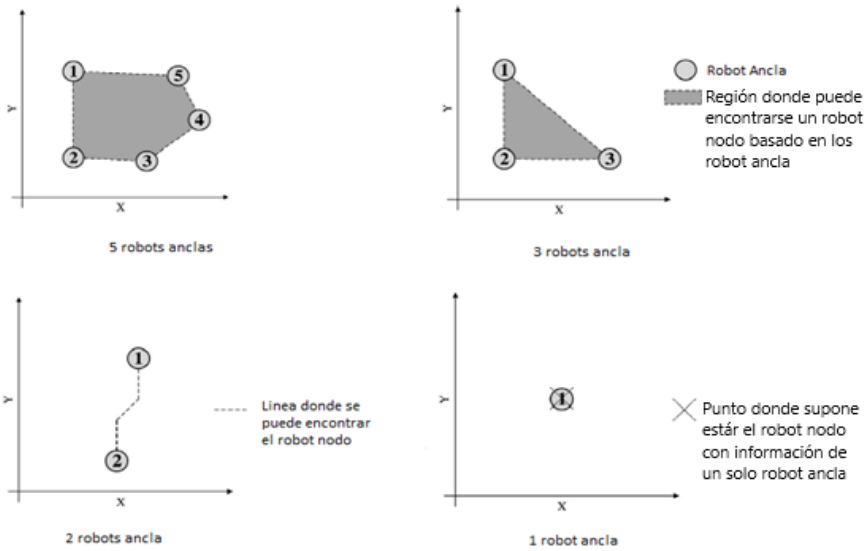


Figura 2.24. Región de la posición de un robot nodo basado en la información de diversos robots anclas
Fuente:**[23]**

Como se puede ver en la imagen, el caso de 1 y 2 robots anclas hace que no se pueda calcular la posición por la escasa información, pero haciendo uso del algoritmo MCMM se puede solucionar. Este algoritmo hace uso de los robots nodos adyacentes usándolos como anclas provisionales para calcular su propia posición.

Al principio del algoritmo se calcula la longitud del área delimitadora S_i mediante las ecuaciones (2.4):

$$\Delta x_i = S_{i,x_{min}} - S_{i,x_{max}} \qquad \Delta y_i = S_{i,y_{min}} - S_{i,y_{max}} \tag{2.4}$$

Esta información se comparte con todos los robots nodos vecinos. Una vez compartida esta información el robot nodo selecciona el nodo con $\Delta \boldsymbol{x_i}$ más pequeña y lo elige ancla auxiliar ($\in_{\boldsymbol{x}}$). Esta operación se repite, pero usando los robots nodos con el menor Δy_i. Una vez seleccionados los robots anclas auxiliares $\in_{\boldsymbol{x}}$ y $\in_{\boldsymbol{y}}$ se ejecuta el algoritmo **Sum-Dist** y a continuación se recalculan los parámetros $\Delta \boldsymbol{x_i}$, $\Delta \boldsymbol{y_i}$,$\boldsymbol{x_i}$ e $\boldsymbol{y_i}$ haciendo uso del método **Min-Max** pero solo se actualiza la posición si:

(I) El robot nodo i tiene menos de 3 nodos anclas como referencia en la primera etapa (Etapa I)
(II) No es posible verificar una buena alineación entre el robot nodo ***i*** y dos robots anclas ***r*** y ***r'***. Para verificar esta alineación se utiliza la ecuación (2.5):

$$\frac{l_{i,r} + l_{i,r'}}{\sqrt{(x_r - x_{r'})^2 + (y_r + y_{r'})^2}} - 1 \geq \beta \tag{2.5}$$

Dónde $\boldsymbol{\beta}$ es el límite de alineamiento que se ajusta de forma empírica. En la figura 2.25 se representa el algoritmo MCMM donde ***n*** es el número de robots nodos:

Inicio
- *Calcula x_i, y_i, Δx_i y Δy_i;*
- ***desde*** *j =1 hasta n* ***hacer****:*
  - *Compartir con el enjambre x_i, y_i, Δx_i y Δy_i;*
  - *Seleccionar $\in_x$;*
  - *Seleccionar $\in_y$;*
  - *Ejecutar algoritmo Sum-Dist con los robots anclas auxiliares $\in_x$ y $\in_y$;*
  - ***si*** *se dan las condiciones (I) y (II):*
    - *Ejecuta el método Min-Max;*
    - *Actualiza x_i, y_i, Δx_i y Δy_i ;*
  - ***fin***
- ***fin***

fin

Figura 2.25. Algoritmo MCMM

Una vez terminado el algoritmo **MCMM** se comparte con todo el enjambre la posición calculada para su posterior uso en la etapa 4. Antes de entrar en la etapa 4 se calcula el factor de confianza ($\boldsymbol{\zeta}$) este factor se utiliza para ver el grado de acierto en el cálculo de la posición y se usará en la etapa 4. El cálculo del factor de confianza se basa en el área de intersección de un robot nodo y se calcula con la ecuación (2.6):

$$\zeta_i = 1 + |S_i| \tag{2.6}$$

En el caso de un robot ancla la ecuación (2.7):

$$\zeta_r = 1 \tag{2.7}$$

2.6.4. Backtracking Search Algorithm (BSA) (Etapa IV)

El algoritmo **BSA** es un algoritmo utilizado para optimizar funciones numéricas. Mas concretamente, es un *algoritmo evolutivo* que permite optimizar problemas no lineales o con varias dimensiones, como es el caso de la posición de un robot ya que hay que optimizar la posición x e y. Los algoritmos evolutivos están basados en la aleatoriedad, en la que en cada iteración se genera una multitud de soluciones y a partir de estas soluciones, por un proceso de mutación, en la siguiente iteración se genera otra variedad de soluciones basada en la anterior y seleccionando continuamente la que mejor optimice la función objetivo a optimizar **[24]**.

El algoritmo **BSA** está dividido en 5 pasos o etapas como se muestra en la figura 2.26:

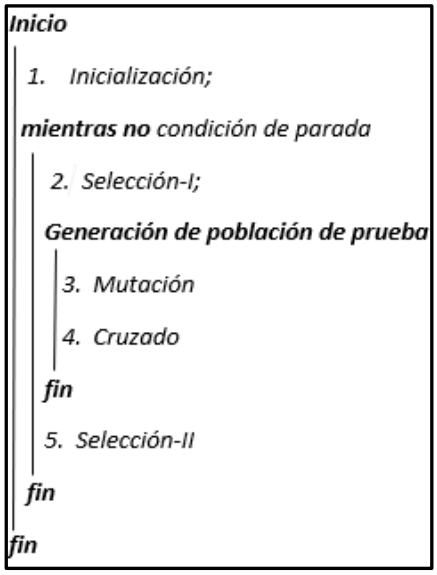


Figura 2.26. Algoritmo BSA

1. ***Inicialización*:** Se genera una población $\boldsymbol{P}$ que representa un conjunto de posibles soluciones del problema a optimizar donde ***i*** es la cantidad de soluciones deseada y ***j*** la dimensión del problema. Esta población se genera con la ecuación (2.8) que describe una función de distribución uniforme ($\boldsymbol{U}$):

$$P \sim U(menor, mayor) \tag{2.8}$$

2. ***Selección-I*:** En este paso se selecciona la población historia ($\boldsymbol{P_{hist}}$), generándola según la ecuación (2.9):

$$P_{hist} \sim U(menor, mayor) \tag{2.9}$$

   A continuación, con un criterio aleatorio descrito en la ecuación (2.10), se mezcla la población $\boldsymbol{P_{hist}}$ generada en (2.9) con la población $\boldsymbol{P}$:

$$si\ a < b\ entonces\ P_{hist} \coloneqq P|a, b \sim U(0,1) \tag{2.10}$$

   Esto se hace con cada solución de la población y una vez evaluada cada una de estas soluciones para finalizar se baraja la matriz población $\boldsymbol{P_{hist}}$.

3. ***Mutación*:** Para crear una nueva versión de la actual población con la que generar nuevas soluciones posibles se genera la población mutación ($\boldsymbol{P_{mod}}$) con la ecuación (2.11):

$$P_{mod} = P + \eta\Gamma(P_{hist} - P) \tag{2.11}$$

Donde $\boldsymbol{\eta}$ se ajusta de forma empírica y $\boldsymbol{\Gamma}$ es una distribución normal descrita en la ecuación (2.12):

$$\Gamma \sim N(0.1) \tag{2.12}$$

La población $\boldsymbol{P_{mod}}$ es una matriz generada de forma aleatoria basada en valores de la población $\boldsymbol{P_{hist}}$ y $\boldsymbol{P}$.

4. ***Cruzado*:** El cruzado es un algoritmo que selecciona de forma aleatoria entre la población mutada $\boldsymbol{P_{mod}}$ las posibles soluciones que mejoran las de $\boldsymbol{P}$. Esto se hace mediante un proceso de selección basado en dos estrategias aleatorias creando una nueva población $\boldsymbol{P_{new}}$ que será la que se enfrentará a la población $\boldsymbol{P}$ seleccionando las soluciones que optimicen mejor el problema, véase figura 2.27:

```
Inicio
  X_{1:Y,1:D} = 1;
  si a < b|a, b ~ U(0,1) entonces:
    Estrategia I:
    desde i=1 hasta Y hacer:
      X_{i,u_{(1:[Q*rnd*D])}} = 0|u = alternar(1,2,3 ..., D);
    fin
  fin
  sino:
    Estrategia II:
    desde i=1 hasta Y hacer:
      X_{i,rndi(D)} = 0;
    fin
  fin
  P_new = P_mod;
  desde i=1 hasta Y hacer:
    desde j=1 hasta D hacer:
      si X_{i,j} = 1 entonces:
        P_new_{i,j} = P_{i,j};
      fin
    fin
  fin
fin
```

Figura 2.27. Algoritmo de cruzado

Inicialmente, se genera una matriz de mapeo de unos $\boldsymbol{X_{Y,D}}$ de dimensión $\boldsymbol{Y} \times \boldsymbol{D}$ donde $\boldsymbol{Y}$ es el numero de soluciones y **D** es la dimensión del problema a optimizar. Esta matriz se utiliza para saber cuál de las soluciones de la población $\boldsymbol{P_{mod}}$ mutada serán enfrentadas con las de $\boldsymbol{P}$.

A continuación, se selecciona siguiendo una distribución uniforme que estrategia se tomará para la selección de que valores mutados serán los elegidos para enfrentarlos:

- **Estrategia I**, esta estrategia permite que una solución sea elegida, pero puede ser elegida desde 1 hasta D o solo algunos de los elementos de la solución que se está valorando. $\boldsymbol{Q}$ es el coeficiente de cruzado que se elige de forma empírica y rnd es una distribución uniforme entre 0 y 1 ($rnd \sim U(0,1)$).

- La **Estrategia II** elige una solución, pero solo uno de sus elementos es elegido para que mute. La función $rndi(D)$ es una distribución uniforme entre 0 y D ($rndi \sim U(0, D)$) esto hace que solo uno de los elementos de cada sujeto mute.

Una vez se hayan seleccionado que valores son los que mutaran usando la matriz $\boldsymbol{X_{Y,D}}$, se genera la población $\boldsymbol{P_{new}}$ y se recorre cada elemento y dependiendo si ese mismo elemento ($\boldsymbol{i}, \boldsymbol{j}$) de la matriz de mapeo es un 1 o 0 este elemento será seleccionado para mutar si es 0 o no mutar si es 1.

5. ***Selección* II**: Una vez generada la población $\boldsymbol{P_{new}}$ se evalúa cada solución $\boldsymbol{i}$ de la población $\boldsymbol{P_{new}}$ y $\boldsymbol{P}$ en el problema a optimizar y si la solución $\boldsymbol{i}$ de la población $\boldsymbol{P_{new}}$ es mejor que la $\boldsymbol{i}$ de $\boldsymbol{P}$, ésta será sustituida por la solución de la población mutada $\boldsymbol{P_{new}}$.

2.6.5. Funciones para optimizar con el algoritmo BSA

El algoritmo **BSA** se utiliza para optimizar una función, esta función puede ser de dimensión $\boldsymbol{D}$ que en este caso $D = 2$ ya que lo que se tiene que optimizar son las coordenadas $\boldsymbol{x_i}$ e $\boldsymbol{y_i}$. Mas concretamente, la función a optimizar tiene como objetivo minimizar el error de las distancias a las que se encuentra un robot nodo de sus vecinos. La función ($\boldsymbol{f}$) consta de dos términos, el error que hay de un robot nodo a sus nodos vecinos de un salto ($\boldsymbol{f}_1$) y el error de un robot a sus nodos de dos saltos ($\boldsymbol{f}_2$):

- **Error de vecinos de un salto** ($\boldsymbol{f}_1$)**:** Se calcula con las distancias a las que se encuentran los vecinos a un salto ($\boldsymbol{v}$). Cuando se habla de vecinos a un salto quiere decir de robots nodos, junto al robot que se quiere calcular la posición, a los que están en rango para enviar/recibir mensajes sin necesidad de robots nodos intermediaros.

  Para calcular el error se resta a la distancia real medida la distancia entre las posiciones calculadas en los apartados anteriores según la ecuación (2.13):

$$g_{i,v} = d_{i,v} - \|p_v - p_i\| \tag{2.13}$$

  Esto es para el error de un solo nodo vecino, pero el robot vecino también puede ser el caso que no sepa su posición por lo que se tiene que hacer uso del factor de confianza ($\boldsymbol{\zeta_v}$) por lo que la ecuación (2.14) proporciona el error de vecinos de un salto:

$$f_1 = \sum_{v \epsilon V_i} \frac{g_{i,v}^2}{\zeta_v} \tag{2.14}$$

Para una mejor comprensión del error $\boldsymbol{g_{i,v}}$ que se produce en la posición calculada de un robot, se hace uso de la figura 2.28:

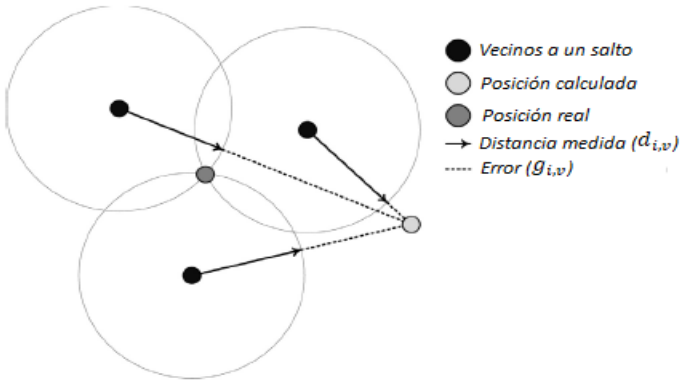


Figura 2.28. Representación gráfica del cálculo del error $g_{i,v}$ de vecinos a un salto
Fuente:**[23]**

- **Error de vecinos de dos saltos ($\boldsymbol{f_2}$):** Se calcula basándose en la distancia máxima con la que un robot puede enviar/recibir mensajes con sus vecinos. La distancia $\boldsymbol{L}$ es el rango máximo con el que un robot se puede comunicar con otro, por lo que la distancia a la que puede estar un vecino de dos saltos ($\boldsymbol{w}$) está comprendida entre $L \leq \|p_w - p_i\| \leq 2L$ y el cálculo del error ($\boldsymbol{h_{i,w}}$) está definido como en la ecuación (2.15):

$$h_{i,w} = \max(0, L - \|p_w - p_i\|, \|p_w - p_i\| - 2L)^2 \tag{2.15}$$

Con (2.16) se calcula el error de vecinos de dos saltos $\boldsymbol{f_2}$:

$$f_2 = \sum_{w \epsilon W_i} \frac{h_{i,w}}{\zeta_v} \tag{2.16}$$

En la figura 2.29. se puede ver que cuando una posición está bien calculada ($h_{i,w} = 0$) es porque se encuentra en la posición fuera del rango de comunicación. Sin embargo, pero si hay error quiere decir que o la posición calculada ($h_{i,w} = b^2$) está dentro del rango o está más lejos de 2 saltos $h_{i,w} = a^2$ por lo que estaría mal calculado en ambos casos y habría error

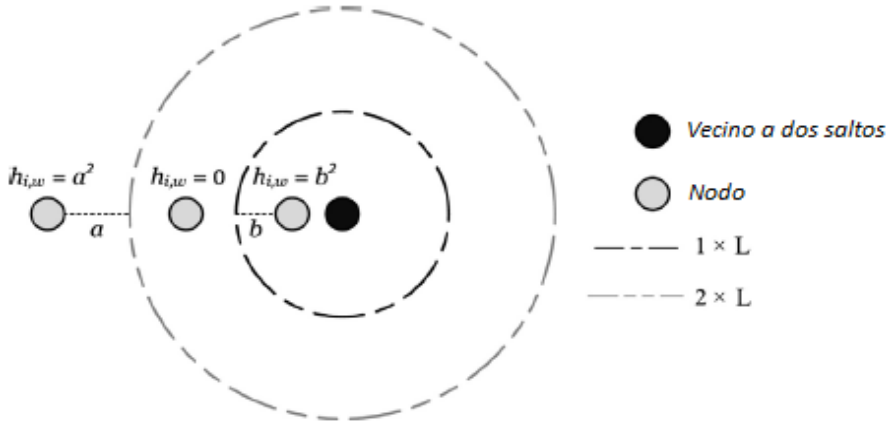


Figura 2.29. Representación gráfica del cálculo del error $h_{i,w}$ de vecinos a dos saltos
Fuente:**[23]**

Finalmente, la función objetivo a optimizar con el algoritmo BSA es la (2.17) que es la suma de (2.16) y (2.14):

$$f = \min(f_1 + f_2) \tag{2.17}$$

2.7. Algoritmo "Wave"

Para la comunicación entre robots se utiliza el algoritmo "Wave" ya que es el ideal para trabajar con robots de enjambre. Consiste básicamente en pasar mensajes entre todos los miembros del enjambre con un robot que inicie este mensaje y todos los demás pasándolos a sus vecinos imitando el comportamiento de una onda. Para el funcionamiento de este algoritmo se necesita la lista de vecinos que se puede descubrir con el siguiente algoritmo, véase figura 2.30:

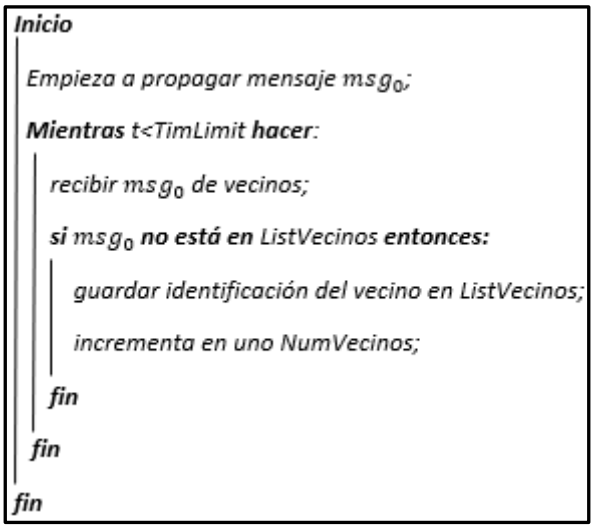


Figura 2.30. Algoritmo descubrir vecindario

La lista de vecinos se descubre con un límite tiempo ***TimLimit***, en el mensaje msg_0 se guarda la identificación del remitente para que los vecinos puedan almacenarlo en ***ListVecinos*** y así tener una lista de sus robots vecinos. Esta lista también será útil para el algoritmo "***distributed and resilient localization***" visto en el apartado anterior ya que se necesita saber cuáles son los vecinos a un salto y a dos.

El algoritmo Wave empieza con un iniciador denomina robot líder que será el encargado de guiar al enjambre hasta un foco de luz. Este algoritmo se usa para resolver tareas de forma conjunta en la que el robot líder ordena al enjambre lo que tiene que hacer, estas órdenes se propagan de forma que siguen la estructura del algoritmo Wave. En la figura 2.31. se representa el algoritmo para la toma de decisiones. Si el robot es el líder o iniciador envía mensajes con las órdenes que tiene que hacer el enjambre y después espera una realimentación proveniente del enjambre para saber qué acciones tomar en base a los mensajes del enjambre. Si no es robot iniciador o líder recibe el mensaje con la orden a llevar, pero antes de llevar a cabo las acciones se tiene que configurar si el remitente del mensaje es su padre, en el caso de que no tenga lo guarda como su padre y todos los mensajes provenientes de otros robots que no sean su padre los descarta. Una vez se han hecho las acciones u órdenes que contenía el mensaje envía un mensaje de realimentación al enjambre con destino al robot padre.

```
Inicio
    si eres iniciador o líder entonces:
        propagar mensaje con ordenes;
        recibir mensaje con realimentación de los vecinos;
        tomar decisión;
    fin
    sino:
        recibir mensajes;
        si no tiene padre entonces:
            define al remitente como padre;
            propaga el mensaje;
        fin
        enviar mensaje de realimentación al padre;
        esperar a un nuevo mensaje;
    fin
fin
```

Figura 2.31. Algoritmo Wave para toma decisiones

En este capítulo se explica el apartado de simulación de los Kilobots. Los pasos realizados en CoppeliaSim para la puesta en escena de los Kilobots, explicación del código empleado y sus diferentes funciones en la programación del algoritmo "***distributed and resilient localization***" y el programa para guiar el enjambre hacia un foco de luz.

3.1. Preparación de la escena en CoppeliaSim

La preparación de la escena se realiza creando un área de trabajo cuadrada y delimitada que contiene un foco de luz y varios Kilobots. Además, se le añade a los Kilobots un sensor de luz necesario para la programación del programa del enjambre que se mueve hacia un foco de luz.

3.1.1. Área de trabajo

Para la preparación del área de trabajo se añadirán 4 objetos principales a la escena:

- **b0RemoteApiServer**: Este objeto crea el servidor para la comunicación entre **CoppeliaSim** y una aplicación externa, que en este caso es ***Jupyter***. Los pasos para añadirlo se resumen a continuación:

  Se busca en el *model browser* de CoppeliaSim el objeto B0 remote API Server como se muestra en la figura 3.1:

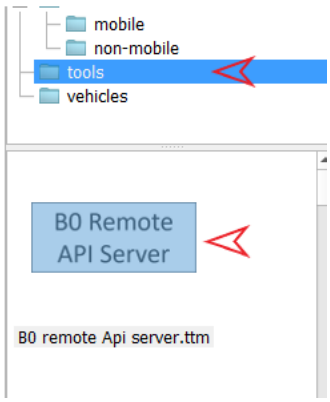


Figura 3.1. Pasos para incluir el objeto B0 remote API Server

  Este objeto simplemente se arrastra hasta la escena y se configuran los nombres del nodo para su posterior uso en ***Jupyter***. Para configurar los nombres se selecciona el objeto creado en la ventana de jerarquía de objetos y aparece la figura 3.2:

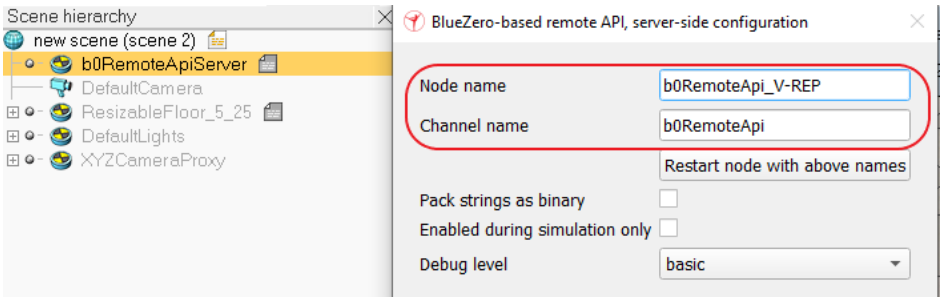


Figura 3.2. Ventana del objeto B0 remote API Server

  Los apartados *Node name* y *Channel name* son los que se utilizan para la comunicación con ***Jupyter***. Estos nombres se usarán en la función de la figura 3.3 para definir el canal de comunicación:

```python
# Carga de las diferentes librerias que se utilizan
import b0RemoteApi
import math
import time
import random
import numpy as np
from IPython.core.debugger import Tracer

#Definicion de la conexión con Coppelia
client = b0RemoteApi.RemoteApiClient('b0RemoteApi_V-REP','b0RemoteApi',60,True,60)
```

Figura 3.3. Script definición del cliente para CoppeliaSIm

Como se ve en el script de la figura 3.3. inicialmente, se importa la librería de comunicación b0RemoteApi y a continuación se define el canal cliente con los nombres del servidor(b0RemoteApi_V-REP) y el nombre del canal (b0RemoteApi), que son los configurados en la figura 3.2.

Además de añadir este objeto se tiene que habilitar la comunicación externa a través de b0 remote API véase figura 3.4:

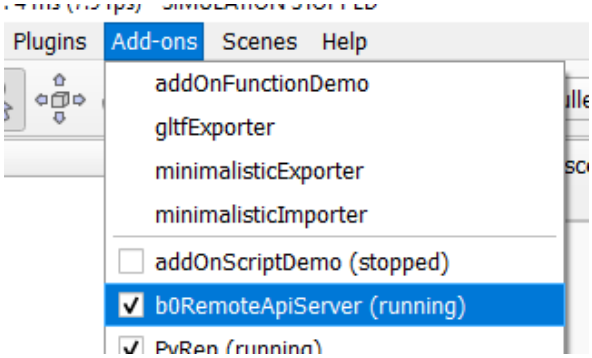


Figura 3.4. Habilitar la comunicación vía b0 Remote API

- **Foco de Luz:** Se eliminan las fuentes de luz que tiene la escena por defecto y se añade un foco de luz para simular el ensayo real. En la figura 3.5 se muestran los pasos para añadirlo usando el menú desplegable que se muestra al pulsar el botón derecho del ratón:

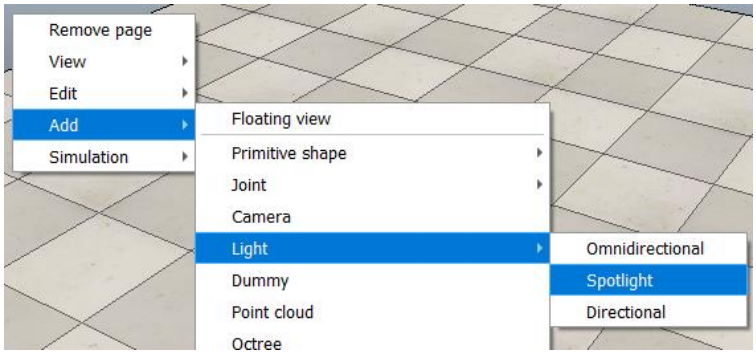


Figura 3.5. Incluir un foco de luz a la escena

A este foco de luz se le añade un *objeto dummy* en la misma posición que el foco y se le vincula en la jerarquía de escena al foco. Este *objeto dummy* se utiliza para que sea detectado por el sensor de luz del Kilobot.

Para hacer esto se abre el menú desplegable haciendo clic con el botón derecho en la ventana de simulación, y se selecciona *add* y el objeto *dummy* como se muestra en la figura 3.6:

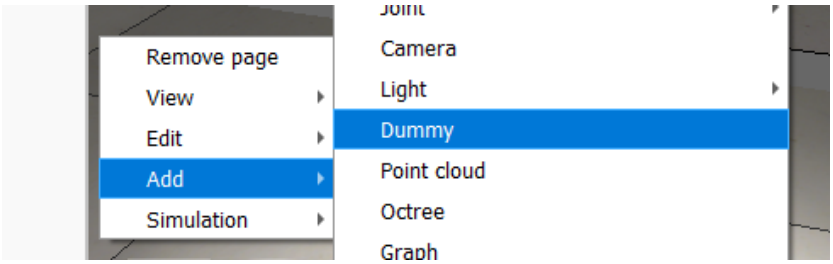


Figura 3.6. Agregar un objeto dummy a la escena

El objeto que acaba de aparecer se coloca en la misma posición que el foco para que actúe como señal para el sensor del Kilobot. Para colocarlo en la misma posición se hace uso de la herramienta de *movimiento* de la *barra de herramientas*. Para hacer esto se abre la herramienta de *movimiento de objetos* y se selecciona el objeto *dummy* en la jerarquía y a continuación el foco, y en la pestaña *position* de la pestaña de movimiento de objetos se hace clic izquierdo en el botón *Apply to selection*, haciendo esto se coloca el objeto *dummy* en la misma posición que el foco, en la figura 3.7 se muestran los pasos a seguir:

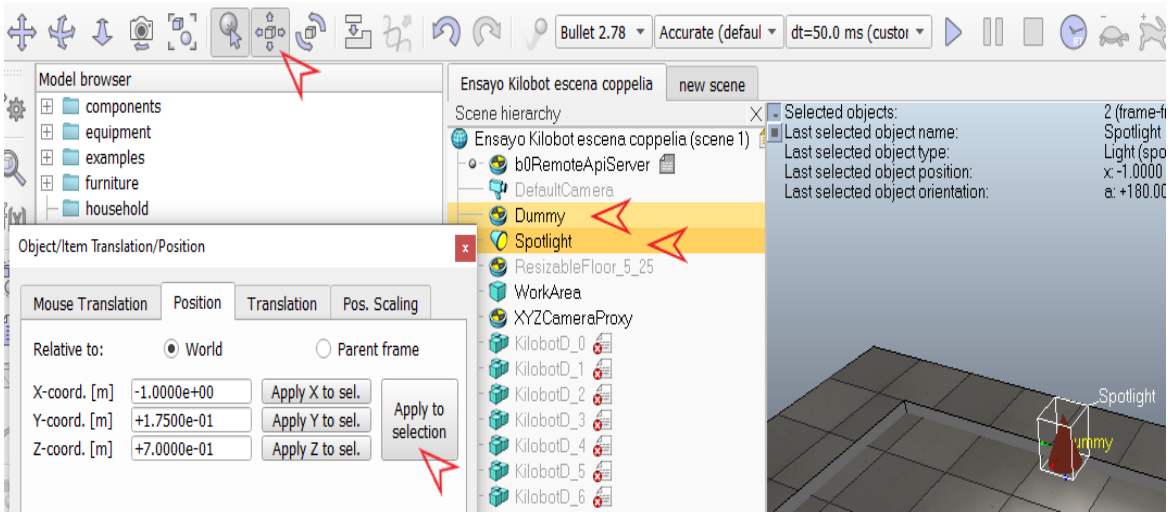


Figura 3.7. Pasos para mover el objeto dummy a la misma posición que el foco de luz

Este objeto *dummy* se agrega en la jerarquía del foco de luz y una vez colocado en la misma posición que la del foco se le cambia el nombre a “Sensor_light_detect”. En la figura 3.8 se muestra una imagen del foco (*Spotlight*) y su jerarquía con el *objeto dummy* llamado “Sensor_light_detect”.

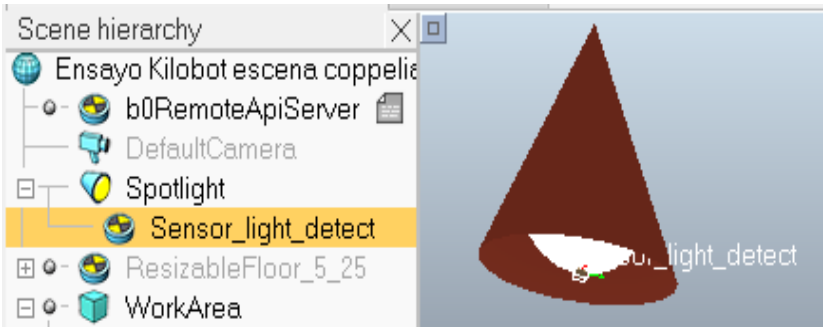


Figura 3.8. Jerarquía e imagen de un foco en CoppeliaSim

- **Caja delimitadora**: Esta caja define el área de trabajo para que los Kilobots no se salgan de la zona de trabajo durante las simulaciones. Se ha creado con 4 *objetos forma (shape)* planos unidos jerárquicamente con una dimensión de 3 m de largo y 10 cm de alto véase figura 3.9.

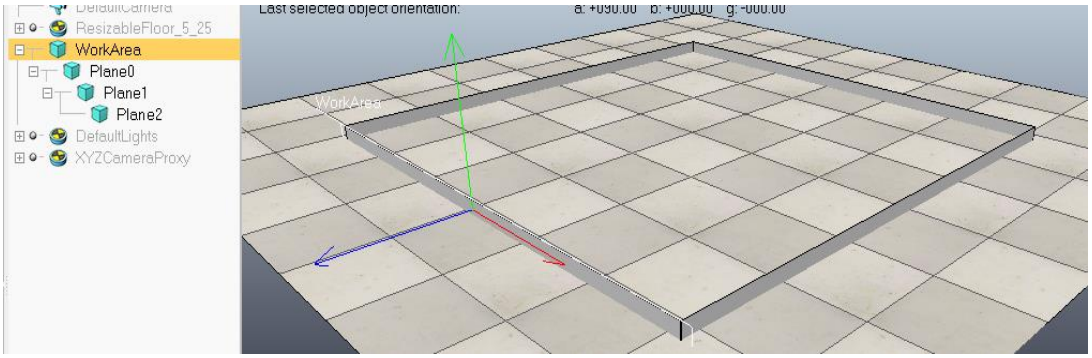


Figura 3.9. Caja delimitadora

Para que el objeto no sea estático, en la ventana *propiedades de objeto* de la caja delimitadora se le ha desmarcado la casilla "*body is dinamic*" de este modo no le afecta la gravedad y no puede ser movido en la simulación por algún robot. Además de desmarcar esta casilla, se marca también la casilla "*collidable*" de la ventana "*common"* en la ventana *propiedades de objeto*.

- Por último, se agregan a la escena 7 Kilobots. Para ello, se busca en el ***model bowser*** y se agrega arrastrando la figura a la ventana de simulación. La distribución de los Kilobots se muestra en la figura 3.10:

Figura 3.10. Disposición de los Kilobots en CoppeliaSim

Para comprobar el algoritmo "**distributed and resilient localization**" se anota la posición en la que se encuentra cada Kilobot, véase tabla 3.1:

Nº ID Kilobot	Coordenada x(m)	Coordenada y(m)
0	0	0
1	0	0.075
2	0	0.14
3	0.075	0
4	0.14	0
5	0.30	0.30
6	0.65	0.65

Tabla 3.1. Coordenadas de los Kilobots en CoppeliaSim

Y para la simulación del código en el que un Kilobot guía al enjambre hacia un foco de luz es necesario orientar todos los Kilobots hacia el mismo sentido. Para hacer esto se hace uso de la herramienta de orientación de objetos y de la jerarquía de escena de los robots:

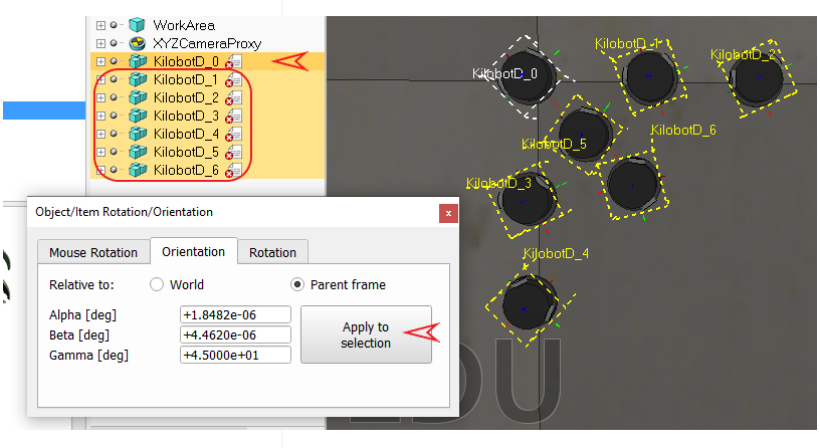


Figura 3.11. Orientación de los Kilobots

En la figura 3.11 se muestran los pasos a seguir, primero se seleccionan todos los Kilobots que se quieren orientar y por último al Kilobot líder con la orientación deseada para el enjambre. Posteriormente, con la herramienta de orientación de la barra de herramientas en la pestaña de *Orientation* se selecciona *Apply to selection*. Una vez se hace esto todos los Kilobots quedan orientados de la misma forma que el Kilobot líder.

Para terminar, se tienen que deshabilitar el uso del script Lua del Kilobot para poder usar los scripts creados con Python. Para hacer esto se selecciona en la pestaña de tools>script y se deshabilitan los scripts de los Kilobots como muestra la figura 3.12:

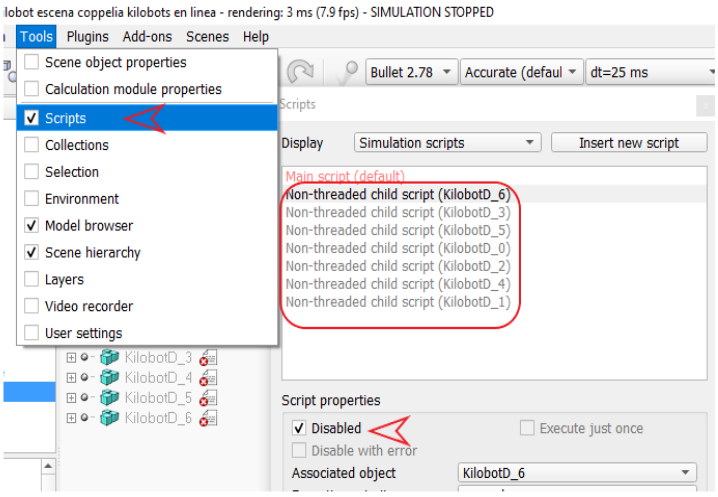


Figura 3.12. Deshabilitar el script de Lua de los Kilobots

3.1.2. Adición de sensor de luz a los Kilobots

El Kilobot de CoppeliaSim no incluye un sensor de luz, por lo que se tiene que añadir de forma manual un sensor que detecte el foco creado en el paso anterior. Este paso se realiza antes de la colocación y orientación de los Kilobots. Primeramente, se añade un solo Kilobot y se le añade el sensor de luz y se réplica en la escena copiando y pegando este Kilobot.

Para agregar el sensor de luz, se selecciona el Kilobot y se hace clic con el botón derecho del ratón para mostrar el menú desplegable y se selecciona la opción de añadir un sensor de proximidad cilíndrico, véase figura 3.13:

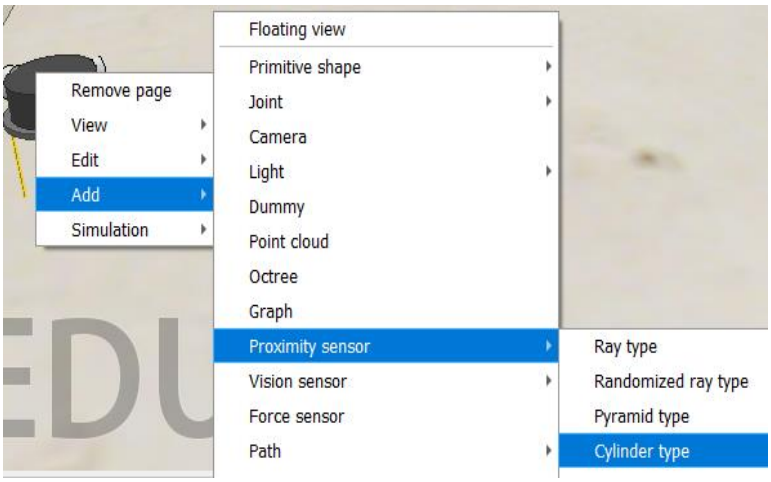


Figura 3.13. Agregar un sensor de proximidad cilíndrico

Este sensor se configura para que detecte el *objeto dummy* llamado "sensor_light_detec", ya que este objeto tiene la misma posición que la del foco de luz que se ha añadido en el apartado anterior. De esta forma, se consigue que los Kilobots puedan detectar el foco de luz cuando entra en el rango del sensor y así simular el sensor de luz del Kilobot real. El rango del sensor seleccionado es el mostrado en la figura 3.14:

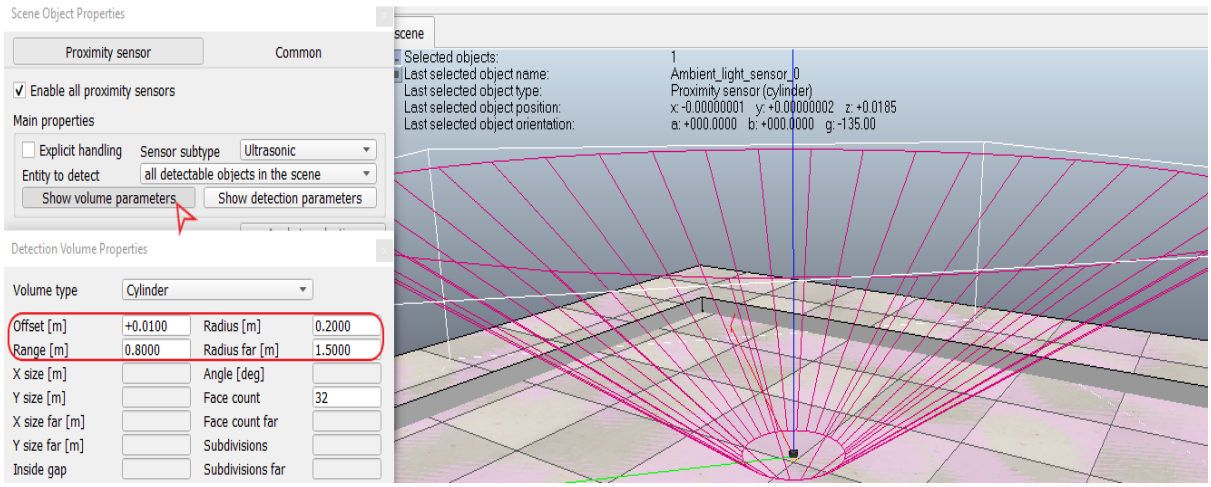


Figura 3.14. Configuración del sensor de luz

Tiene una altura de 0,8 m, un radio inferior de 0,2 m y un radio superior de 1,5 m, dando al sensor el rango mostrado en la figura 3.14. Una vez hecho esto se selecciona el objeto que tiene que detectar, que es el objeto *dummy* "sensor_light_detec" como muestra la figura 3.15:

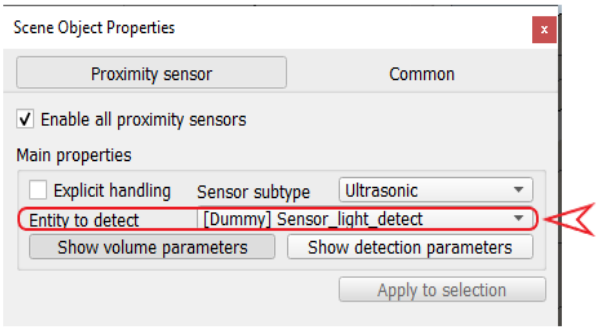


Figura 3.15. Seleccionar el objeto a detectar

3.2. Preparación de Python

Para la programación en Python se usa la aplicación web ***Jupyter***, un compilador que a partir de celdas de código permite ejecutar estas secciones de código sin tener que ejecutar un código completo. Para la POO se crea la clase "Kilobot" que contiene los diferentes métodos y atributos para su programación.

3.2.1. Constructor Kilobot

El constructor se usa para generar por defecto los diferentes atributos de una clase. En este constructor se le asignará el nombre de los diferentes atributos del Kilobot en CoppeliaSim como atributos de entrada de cada Kilobot (nombre ID, nombre del motor derecho, nombre del motor izquierdo y nombres de los sensores). En la figura 3.16 se muestra un fragmento de ejemplo con el código de los nombres de los atributos del KilobotD_0:

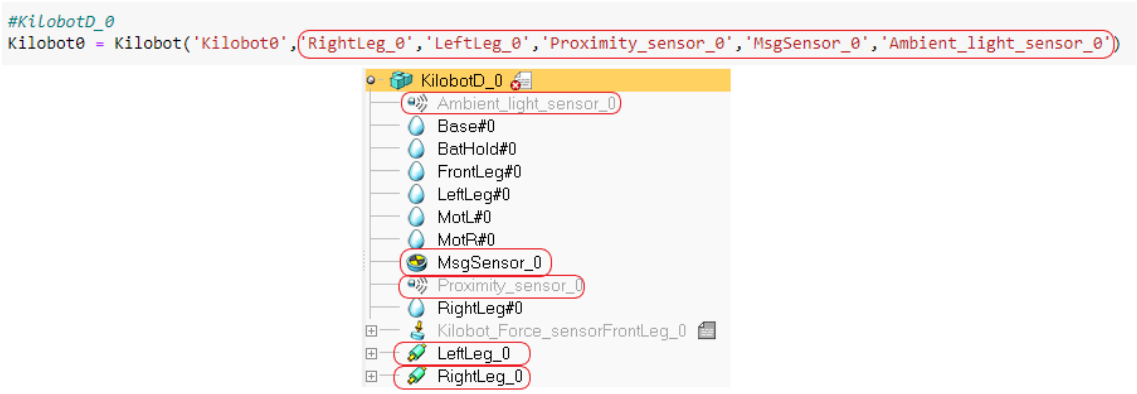


Figura 3.16. Constructor y ejemplo de uso con la jerarquía del KilobotD_0

El constructor crea los atributos que contiene el Kilobot que son comunes en todos los Kilobots. Los que se definen proveniente de CoppeliaSim son los objetos que pueden interactuar en la escena. Estos atributos se definen como el mando(handle) del objeto de CoppeliaSim usando la función **`simxGetObjectHandle`**. Además de todos los mandos de cada objeto también se generan una variedad de atributos como banderas, memoria y parámetros importantes para la programación de los métodos utilizados. Todos estos atributos vienen explicados detalladamente en el **Anexo II** que contiene el código con comentarios explicando para que se utiliza cada atributo.

3.2.2. Métodos básicos y funciones utilizadas

En esta sección se hará un breve resumen de los métodos utilizados:

- **`Initialization(flagNoise)`**: Método que inicializa el Kilobot, pone su velocidad a 0 y si es el KilobotD_0 lo asigna como líder, si el parámetro de entrada ***`flagNoise`*** es **`True`** se le añade ruido de movimiento al Kilobot.

- **`Set_velocity(motor,velocity)`**: Para controlar la velocidad de los motores, con el parámetro de entrada ***`motor`*** se señaliza el motor al que se le tiene que asignar una velocidad introducida ***`velocity`***. Se debe tener en cuenta que lo que permite el movimiento son la vibración de las patillas en el Kilobot de simulación por lo que si vibra el motor izquierdo (patilla izquierda) este girara a la derecha al contrario que en un Kilobot real que si vibra el motor izquierdo gira a la izquierda.

- **`Turn_left()`**: Cada vez que se llama este método el Kilobot gira a izquierda a máxima velocidad.

- **`Turn_right()`**: Análoga a la anterior pero con la diferencia de que gira a la derecha.

- **`Straight(velocity)`**: Método para hacer que el Kilobot avance en línea recta con una velocidad ***`velocity`*** introducida.

- **`Light_sensor()`**: Este método devuelve la lectura del sensor de luz. Como es una simulación del sensor real devuelve un valor de proximidad y las coordenadas relativas al Kilobot del foco de luz. Si no detecta el sensor devuelve -1.

- **`Broadcast_mode(message)`**: Retransmite un mensaje ***`message`*** a los Kilobots adyacentes.

- **`Listen_mode()`**: Pone el Kilobot en modo escucha habilitando la recepción de mensajes.

- **`Store_data(data)`:** Guarda un dato ***`data`*** en la memoria de datos. Este método se utiliza en la recepción de mensajes.

- **`Delete_memory()`:** Borra los mensajes recibidos de la memoria.

- **`Variable_timer(timeSelect)`:** Un temporizador con una duración determinada por el parámetro de entrada ***`timeSelect`***. Este método devuelve **`False`** siempre que no ha pasado el tiempo determinado. Una vez se alcanza el tiempo ***`timeSelect`*** devuelve **`True`** y se reinicia.

- **`Neighborhood_discover(Ts)`:** Es un algoritmo que proviene de **[13]** que se utiliza para descubrir los Kilobots próximos al que inicia este método. Este algoritmo retransmite su nombre durante un tiempo ***`Ts`*** introducido, revisa los mensajes que le han llegado y agrega los nombres a una lista de vecinos junto con la distancia a la que se encuentran.

- **`Node_two_hope_discover(Ts)`**: Método para crear una lista con los Kilobots que se encuentran a dos saltos, esto quiere decir, una lista con los Kilobots vecinos que son vecino del Kilobot que inicia este método o una lista con los Kilobots que están en rango de comunicación con los vecinos del Kilobot que inicia este método. Para llevarse a cabo este método se inicia el algoritmo **`Neighborhood_discover(Ts)`** anteriormente explicado, para obtener la lista de vecinos. Una vez hecho esto se retransmite esta lista a los vecinos y se comprueba la memoria con los mensajes que le han llegado, se revisa el remitente del mensaje y si pertenece a la lista de vecinos y si es vecino se usa el contenido del mensaje, que es la lista de vecinos, y se añade a la lista **`NodeTwoHope`** que serán los vecinos a dos saltos.

- **`Swarm_state()`:** Este método se utiliza para que todo el enjambre este coordinado y que salten de una función a otra de forma conjunta. Se retransmite un mensaje con la información del estado (atributo **`state`**) este mensaje llega a los Kilobots junto con la ID del remitente y se crea un diccionario. Una vez que se contiene todo el enjambre en el diccionario con el estado en **`True`** se devuelve **`True`** para indicar que todo el enjambre está listo y se reinicia el diccionario y el atributo estado.

- **`Clear_same_message()`:** En ocasiones se retransmitía varias veces el mismo mensaje, para evitar esto y que haya una gran cantidad de mensajes en la memoria de los Kilobots se usa esta función que borra los mensajes repetidos.

Estos eran los métodos relacionados con los Kilobots y las acciones que pueden llevar a cabo. A continuación se explican otras funciones utilizadas en la programación de los Kilobots:

- **`alpha(coordinate_x,coordinate_y)`:** Función que devuelve el ángulo entre el eje X y dos coordenadas ***`coordinate_x`*** y ***`coordinate_y`*** introducidas. Este ángulo estará comprendido entre 0 y 360º, se utilizará en el sensor de luz para simular el sensor real.

- **`broadcast_message(Kilobot,message)`**: Esta función se llama cuando un Kilobot ***`Kilobot`*** desea retransmitir un mensaje. Esta función revisa cuales son los Kilobots adyacentes a este ***`Kilobot`*** y les escribe en su memoria el mensaje ***`message`*** retransmitido por el ***`Kilobot`***.

- **`message_on_buffer(Kilobot,message,distance,sender)`**: Función que se utiliza en la anterior ya explicada para escribir en la memoria del Kilobot ***`Kilobot`*** el mensaje ***`message`***. Escribe en la memoria del ***`Kilobot`*** un vector que contiene el mensaje ***`message`***, la distancia a la que se encuentra el remitente ***`distance`*** y la ID del remitente ***`sender`***.

- **`reset_SwarmState()`**: Se utilizar para reiniciar el estado **`state`** de los Kilobots.

- **`clean_SwarmState(Kilobot)`:** Reinicia el diccionario del Kilobot ***`Kilobot`*** introducido que contiene los estados de todos los Kilobots.

3.3. Programación del algoritmo "distributed and resilient localization"

El algoritmo "**distributed and resilient localization**" se ha adaptado para poder ser programado con POO. En esta sección se explica detalladamente como se ha programado, pero en el **Anexo II** se ha incluido todo el código con comentarios detallados para su comprensión.

3.3.1. Estimación de la distancia (Etapa I)

Para una primera recopilación de datos de las posiciones de los Kilobots se hace uso del algoritmo **Sum-dist** ya explicado. Para programarlo en POO se tiene que comprobar si el Kilobot es un nodo o un ancla con un condicional. Si es un ancla tan solo retransmite sus coordenadas o si es un nodo se ejecuta el algoritmo **Sum-dist**.

El nombre del método en la programación del código es el de **`Sum_dist(Ts)`** que es el que se muestra en la figura 3.17:

```
#Algoritmo Suma-distancia para un nodo desconocido (STAGE-I)
def Sum_Dist(self,Ts):
    if(self.Anchor==True or self.AnchorAux==True):  #Si es un ancla retransmite sus datos de nombre coordenadas
        self.Broadcast_mode([self.nameKilobot,self.Coordinates[0],self.Coordinates[1],0])#y distancia del mensaje
        if(self.Variable_timer(Ts)):
            return True
        else:
            return False
    else:
```

Figura 3.17. Inicio programación Sum-dist

El parámetro que se introduce ***`Ts`*** es el tiempo que se ejecutará el algoritmo **Sum-dist**. Cómo se muestra al inicio del código se comprueba si es un Kilobot ancla o ancla auxiliar si los atributos **`Anchor`** o **`AnchorAux`** respectivamente son **`True`**. Haciendo uso del temporizador **`Variable_timer()`** se comprueba si ha pasado el tiempo ***`Ts`***, que cuando no ha pasado dicho tiempo devuelve **`False`** y una vez pasado este tiempo devuelve **`True`**.

El código implementado en los Kilobots nodo es el que se muestra en la figura 3.18:

```
else:
    if(not(self.Variable_timer(Ts))): #Revisa que no ha pasado el tiempo (recomendado Ts=7)
        anchorList = self.AnchorList  #Almacena la variable anchorlist que se va a utilizar
        if(self.MemoryCount!=0):      #Comprueba si hay algún mensaje
            for n in range(self.MemoryCount): #El bucle es para revisar todos los mensajes
                message = self.Memory[n][0]   #Almacena el mensaje
                if(len(message)==4):          #Comprueba que tiene la estructura del algoritmo
                    Div=self.Memory[n][1]     #Almacena la distancia a la que está el remitente del mensaje
                    if not(message[0] in anchorList):  #Comprueba si lo tiene en la lista de anclas
                        Lir = message[3]+Div  #Si no lo tiene almacena la distancia a la que está el ancla sumandole
                                              #la del nodo
                        anchorList.setdefault(message[0],[message[1],message[2],Lir])#Guarda el ancla y su información
                        self.Broadcast_mode([message[0],message[1],message[2],Lir])  #Retrasmite un mensaje con la
                                                                                     #información del ancla
                    else:     #Si conoce el ancla
                        value = anchorList[message[0]]    #Almacena los valores del mensaje referentes al ancla
                        if(Div+message[3]<value[2]):      #Comprueba si la distancia que ha recorrido es más larga
                            Lir = message[3]+Div          #Si es una distancia más corta lo actualiza
                            anchorList.update({message[0]:[message[1],message[2],Lir]})
                            self.Broadcast_mode([message[0],message[1],message[2],Lir]) #Retransmite otra vez el ancla
        self.Listen_mode()            #Lo pone en escucha
        self.AnchorList = anchorList  #Almacena en memoria el nuevo diccionario
        return False                  #Devuelve false si no ha terminado el tiempo
    else:
        self.Delete_memory()     #Borra los mensajes que le han llegado cuando termina el algoritmo
        return True              #Devuelve True cuando ha pasado el tiempo
```

Figura 3.18. Programación Sum-dist de los nodos

Se comprueba inicialmente si ha pasado el tiempo seleccionado ***Ts*** y si ha llegado algún mensaje comprobando el contador de memoria de mensajes (**`MemoryCount`**). A continuación, se revisan todos los mensajes con un bucle **`for`** leyendo todos los mensajes en memoria y siguiendo el esquema de programación de la figura 2.1. se ejecuta con todos los mensajes el algoritmo **Sum-dist**, en el caso de que no haya pasado el tiempo ***Ts*** devuelve **`False`** y si lo ha pasado devuelve **`True`** para señalizar que ha terminado el tiempo y por lo tanto el fin del algoritmo.

3.3.2. Primer cálculo de la posición (Etapa II)

Con este algoritmo se consigue una primera estimación de la posición del que lo inicia. Se necesita la lista **`AnchorList`** que contiene la información de la distancia a la que se encuentran los Kilobots anclas, esta lista se calcula previamente en el **Etapa I**. Haciendo uso de la figura 3.19 se procede a explicar la programación:

```
#Algoritmo Min-Max para la obtención de las coordenadas Ui que son las coordenadas del Kilobot(nodo) en bruto (STAGE-II)
def Min_Max(self):
    anchorList = self.AnchorList      #Carga la lista de anclas
    values = []                       #Se tiene que extraer con un bucle for porque la función values()
    for i in anchorList.values():     #da una direccion y no una lista
        values.append(i)              #Extrae de la dirección de todos las anclas la distancia Lir,xr e yr
    Bir1x = []                        #Crea una lista donde se almacenará todos los valores xr-Lir
    Bir1y = []                        #de la misma forma con yr-Lir
    Bir2x = []                        #y con xr+Lir e yr+Lir
    Bir2y = []                        #Estas matrices son las coordenadas del área cuadrada donde estará el nodo
    for i in range(len(values)):
        Bir1x.append(values[i][0]-values[i][2])   #Almacena los valores de todos los nodos
        Bir1y.append(values[i][1]-values[i][2])
        Bir2x.append(values[i][0]+values[i][2])
        Bir2y.append(values[i][1]+values[i][2])
    Si = [[max(Bir1x),max(Bir1y)],[min(Bir2x),min(Bir2y)]]  #Se obtiene el máximo de BR1x y bir1y y
                                                            #Los mínimos Br2x y Bir2y
    Deltax = Si[1][0]-Si[0][0]                              #Deltax y Deltay son los valores de las dimensiones de la
                                                            #caja que contiene Ui estos dos datos son necesarios para
    Deltay = Si[1][1]-Si[0][1]                              #ejecutar el algoritmo MCMM
    Ui = [(Si[0][0]+Si[1][0])/2,(Si[0][1]+Si[1][1])/2]      #Ui son las coordenadas en bruto que son la media de la suma
    return [Ui,[Deltax,Deltay]]  #de los valores x y la suma de los valores y, además de devolver también Deltax y Deltay
```

Figura 3.19. Programación Min-Max

Acorde con el algoritmo **Min-Max**, el método **`Min_Max()`** utiliza cuatro matrices para el cálculo, dos de estas matrices para calcular la amplitud en el eje x de las anclas y las otras dos para la amplitud en el eje y. Cuando se han formado estas matrices se hace uso de la función **`max()`** y **`min()`** para obtener

el máximo y el mínimo de una matriz respectivamente. A continuación, se calcula el área delimitadora S_i que es el área donde se encuentra las coordenadas del Kilobot. Este método devuelve las coordenadas estimadas u_i que son el centro del área S_i y el valor Δx_i y Δy_i para su posterior uso en el **Etapa III.**

3.3.3. Mitigación de fallos en Etapa I (Etapa III)

La mitigación de fallos se lleva a cabo con el método **`MCMM(Ts)`**, que es el **algoritmo MCMM** ya explicado. Este algoritmo lo ejecutan todos los Kilobots incluidos los Kilobots ancla. En la figura 3.20 se muestra la primera parte del código para su explicación:

```
#Algoritmo MCMMM(Multi-hop collaborative Min-Max) este algoritmo sirve cuando se tienen insuficientes
#anclas en el STAGE-I (STAGE-III)
def MCMM(self,Ts):
    if(not(self.FlagEx_me)):                    #Revisa si la bandera de la función Exchange_messages es True
        dates = [self.Coordinates,self.Delta]   #Almacena los datos que va a compartir con los nodos
        if(self.Exchange_messages(dates)==True):#Activa la función en la que los Kilobots intercambian
            self.FlagEx_me = True               #información de los nodos
            self.Delete_memory()                #Pone la bandera en True para que no vuelva a entrar
    elif(not(self.ElectEx_Ey)):       #Comprueba con la bandera ElctEx_Ey si se ha elegido un Ex y Ey
        self.ElectEx_Ey = True        #Pone en True la bandera que indica si se ha elegido uno
        Ex = self.Elect_ex()          #Activa las funciones que eligen Ex y Ey
        Ey = self.Elect_ey()
        if(Ex==self.nameKilobot or Ey==self.nameKilobot): #Si el nombre elegido coincide con el propio
            self.AnchorAux = True                         #se activa como nodo auxiliar
        elif((Ex==True) or (Ey==True)): #Si Ex o Ey son True quiere decir q la lista esta vacía por lo que
            self.FlagEx_me = False      #ya se han evaluado todos los nodos
            self.Flagx = False          #Inicializa todas las variables utilizadas y devuelve True indicando
            self.Flagy = False          # que ha terminado
            self.ListEx = []
            self.ListEy = []
            self.Delete_memory()
            self.State = False
            self.ElectEx_Ey = False
            return True
        else:
            self.AnchorAux = False          #Si no es de ningún otro valor se inicializa el ancla auxiliar
    if(self.ElectEx_Ey and self.FlagEx_me): #Si ya se ha compartido la información y elegido los anclas auxiliar
```

Figura 3.20. Programación MCMM elección de Ex y Ey

Primero se revisa si la bandera **`FlagEx_me`** es **`True`**, en el caso que no lo sea se llama al método **`Exchange_messages(dates)`**. Este método comparte con todo el enjambre la información ***dates*** que contiene las coordenadas y los valores Δx_i y Δy_i de cada nodo, esta información se utiliza para la elección del nodo auxiliar ($\in$).

Una vez se ha compartido toda la información se comprueba con la bandera **`ElectEx_Ey`** si se ha seleccionado algún nodo como auxiliar. Si no es el caso se eligen los nodos auxiliares Ex y Ey ($\in_x, \in_y$) a partir de los métodos **`Elect_ex`** y **`Elect_ey`**. Estos métodos seleccionan los nodos auxiliares de menor a mayor sin repetirlos. Si las funciones de elección de nodos han seleccionado ya todos los nodos disponibles, devuelve **`True`** indicando que ya ha terminado el **algoritmo MCMM** por lo que al terminar este el método **`MCMM`** devuelve **`True`**. Si se ha asignado a si mismo como nodo auxiliar lo almacena en la variable **`AnchorAux`** para cuando se inicie el método **`Sum_dist()`** actúe como un ancla.

Cuando la bandera **`ElectEx_Ey`** es **`True`** quiere decir que ya se ha seleccionado un nodo auxiliar, por lo que en ese caso salta al siguiente condicional, que es donde se comprueba si ya se ha compartido información de los nodos y los nodos auxiliares para iniciar el método **`Sum_dist()`** con los nodos auxiliares.

```
if(self.ElectEx_Ey and self.FlagEx_me): #Si ya se ha compartido la información y elegido los anclas auxiliar
    if(self.Sum_Dist(Ts)==True):        #Llama a la funcion Sum_dist y si termina (devuelve True)
        self.ElectEx_Ey = False         #Inicializa la bandera para elegir un nuevo ancla auxiliar
        self.AnchorAux = False
        if(self.Anchor!=True):           #Si no es ancla
            self.Clear_AnchorListAux()   #Limpia la lista de anclas de las auxiliares
            if(len(self.AnchorList)<3):  #revisa la primera condición del algoritmo(si tiene menos de tres anclas)
                values=[]
                coordinates = self.Min_Max() #Llama a la función min_max para almacenar las nuevas x,y e delta
                for i in self.AnchorList.values():
                    values.append(i)
                if(len(self.AnchorList)==1):  #Y calcula la beta dependiendo si tiene 1 o dos anclas
                    lir = values[0][2]
                    xr = values[0][0]
                    yr = values[0][1]
                    beta = (lir/(((xr**2)+(yr**2))**(1/2)))-1
                elif(len(self.AnchorList)==2):
                    lir1 = values[0][2]
                    lir2 = values[1][2]
                    xr1 = values[0][0]
                    xr2 = values[1][0]
                    yr1 = values[0][1]
                    yr2 = values[1][1]
                    beta = ((lir1+lir2)/((((xr1-xr2)**2)+(yr1-yr2)**2)**(1/2)))-1
                else:
                    beta = -1          #Si no encuentra ningún ancla beta es -1
                if(beta<=self.__Beta): #Si cumple la condición de la beta calculada es menor que la constante beta
                    self.Coordinates = coordinates[0]#Actualiza los valores de las coordenadas y deltas
                    self.Delta = coordinates[1]
    self.Delete_memory()
return False
```

Figura 3.21. Programación MCMM cálculo beta

En la figura 3.21. se muestra la segunda parte del método **`MCMM`** donde se ejecuta el método **`Sum_Dist()`** y se comprueban las condiciones **(I)** y **(II)** explicadas en el apartado 2.1.3 para la actualización de la posición. Para el cálculo del valor de ***Beta*** usado en la condición **(II)** se tiene en cuenta el número de anclas que se han encontrado en la ejecución del método **`Sum_Dist()`**, ya que dependiendo del número de anclas se utiliza una formula distinta. Una vez revisadas las condiciones y el cálculo de ***Beta*** se reinicia la bandera **`ElectEx_Ey`** para la elección de un nuevo Kilobot ancla auxiliar y devuelve **`False`** para indicar que el método no ha terminado.

3.3.4. Cálculo del factor de confianza

Para el refinamiento de la posición se requiere del factor de confianza de cada nodo. Para el cálculo de éste se usa el método **`Confidance_factor()`** en el que además de calcularse el factor de confianza (que se calcula en base de los propios valores Δx_i y Δy_i) se comparte con todo el enjambre el valor del factor de confianza.

```
#Cálculo del factor de confianza para su uso en el BSA y lo comparte con el enjambre
def Confidance_factor(self):
    if(len(self.UnknowNodeList)<self.__UnknowNodeNumber): #Comprueba que la lista de nodos tiene el mismo
        if(self.Anchor!=True):                             #número de elementos que nodos desconocidos hay
            confidanceFactor = 1+(self.Delta[0]*self.Delta[1])     #Calcula el factor de confianza
            self.ConfidenceFactor = confidanceFactor
            self.UnknowNodeList.update({self.nameKilobot:[self.Coordinates,confidanceFactor]}) #Se incluye en la lista
        for i in self.Memory:      #Comprueba la memoria los mensajes que le han llegado
            if not('Sw_st'in i[0]):#Si no es un mensaje de la función 'Swarm_state'('Sw_st') quiere
                                   #decir que si es de esta función
                self.UnknowNodeList.update(i[0]) #Actualiza la lista con el mensaje
        self.Broadcast_mode(self.UnknowNodeList) #y comparte la lista
        self.Listen_mode()
        self.Delete_memory()
        return False
    else:   #Si ya tiene el mismo número de elementos en el diccionario de
            #'UnknowNodeList' que número de nodos desconocidos
        self.Delete_memory()  #Borra la memoria
        return True           #y devuelve true para indicar que ha terminado
```

Figura 3.22. Programación del cálculo del factor de confianza

En la figura 3.22. se muestra la programación del código, primero se comprueba si se tiene la información de todos los nodos comparando la longitud de la lista de nodos con el número de nodos totales. Si no es un ancla se cambia el valor del **factor de confianza**, el cambio se realiza en base a los valores

de delta y esto se comparte con el enjambre. Una vez compartido y si se tiene el factor de confianza de todos los nodos, devuelve **`True`** para indicar que se ha terminado y está listo para pasar a la siguiente etapa.

3.3.5. Refinamiento de posición (Etapa IV)

El refinamiento a partir del algoritmo BSA se realiza con el método **`Refinament_with_BSA()`**. Se realiza apoyándose en la figura 2.26**.** En la figura 3.23 se muestra la programación de la primera etapa de inicialización:

```
#Algoritmo para refinar los resultados con BSA (STAGE-IV)
def Refinament_with_BSA(self):
    if(self.Anchor==True):        #Comprueba si es ancla
        self.Clear_same_message() #Borra mensajes repetidos
        for i in self.Memory:     #y los retrasmite por si los necesita alguien
            self.Broadcast_mode(i[0])
        self.Delete_memory()
        self.Listen_mode()
        return True               #Devuelve true
    else:                         #Si no es un ancla
        if not(self.FlagInitP):   #Revisa si esta inicializada la matriz P comprobando la bandera 'FlagInitP'
            self.P,self.Phist = self.Initialize_P()  #Inicializa la matriz P y Phist llamando a la función 'Initialize_P'
            self.FitnessP = self.Compute_fitness(self.P) #Y calcula el error de todas las coordenadas
            self.FlagInitP = True   #Almacena en la bandera que ya se han inicializado
        else:
```

Figura 3.23. Programación de la inicialización del algoritmo BSA

Además de la inicialización, se comprueba si el que está ejecutando el algoritmo es un ancla, si es el caso tan solo réplica los mensajes que le llegan y devuelve **`True`**. Si no lo es genera la matriz de población $\boldsymbol{P}$ y $\boldsymbol{P_{hist}}$(**`P`** y **`Phist`**) con el método **`Initialize_P()`**. Esta función crea la matriz inicialización de forma aleatoria basándose en su posición con las reglas para generarla ya explicadas en el apartado 2.6.4**.**

Una vez se realiza la inicialización se genera una matriz con el error que se crea con cada elemento de la matriz $\boldsymbol{P}$. Para esto se usa el método **`Compute_fitness(P)`**, que genera una matriz con el error de cada una de las coordenadas generadas. Para el cálculo del error se hace uso del método para el cálculo del error de unas coordenadas explicado en 2.6.4, en el código se ha implementado este método con el método **`Compute_error(`*`inCoordinates`*`)`**, su código se muestra en las siguientes figuras:

```
#Calcula el error de unas coordenadas introducidas
def Compute_error(self,inCoordinates):
    errorG = 0
    for i in self.NeighborsList:       #Recorre la lista de vecinos
        if(i in self.__AnchorsName):   #Si es un ancla su factor de confianza y sus coordenadas son fijas
            confidance = 1
            coordinates = self.__AnchorsName[i]
        else:
            dates = self.UnknowNodeList[i]  #Si no lo es los datos del vecino los tiene en el diccionario
            confidance = dates[1]           #de nodos desconocidos
            coordinates = dates[0]
        pv_pi =[inCoordinates[0]-coordinates[0],inCoordinates[1]-coordinates[1]] #Calcula el vector pv-pi
                                                  #(pv=posicion vecino, pi=posición introducida)
        module=(((pv_pi[0])**2)+(pv_pi[1])**2)**(1/2) #Calcula el módulo del vector
        error = self.NeighborsList[i]-module    #Calcula el error con la distancia a este vecino
        errorG = errorG+((error**2)/confidance) #y lo almacena en errorG dividiendo el error por el factor de confianza
    errorH = 0
```

Figura 3.24. Código para el cálculo del error f_1(errorG)

En la figura 3.24 se muestra la programación del cálculo del primer error **`errorG`** ($\boldsymbol{f_1}$), éste se calcula a partir de la distancia de los nodos vecinos cuyo cálculo matemático está ya explicado en el apartado 2.6.5. La figura 3.25 muestra la programación para el cálculo del **`errorH`** ($\boldsymbol{f_2}$), para el uso de este método se necesita previamente una lista con los nodos vecinos y los nodos a dos saltos. Una vez calculado el error de las coordenadas introducidas ***`inCoordinates`***, devuelve la suma de estos dos errores **`errorT`**.

```
errorH = 0
for i in self.NodeTwoHope:
    if(i in self.__AnchorsName):    #El cálculo de este error es análogo al anterior
        confidance = 1              #pero usando la lista de nodos a dos saltos
        coordinates = self.__AnchorsName[i]
    else:
        dates = self.UnknowNodeList[i]
        confidance = dates[1]
        coordinates = dates[0]
    pw_pi = [inCoordinates[0]-coordinates[0],inCoordinates[1]-coordinates[1]]
    module = ((pw_pi[0]**2)+pw_pi[1]**2)**(1/2)
    error = (max([0,self.__ComunicationScope-module,module-2*self.__ComunicationScope]))**2 #El error es el maximo
                                                                                            #de estos tres valores
    errorH = errorH+(error/confidance)
errorT = errorG+errorH
return errorT
```

Figura 3.25. Código para el cálculo del error f_2(errorH)

```
        self.Delete_memory() #Borra memoria
        if(self.Counter<=self.__NumberCycles): #Comprueba el número de ciclos que lleva
            counter = self.Counter  Condición de parada
            for r in range(self.__IterationsBSA): #Este ciclo es para repetir el BSA
              2. self.Selection_I()     #Llama a la primera función 'Selection_I'
              3. PMod = self.Mutation() #ahora a la función de mutación y almacena el valor de la matriz Pmod
              4. PNew = self.Crossover_operation(PMod) #Con la matriz Pmod calculada llama a la función
                                                       #de cruce devolviendo Pnew
              5. minimal = self.Selection_II(PNew) #Con la matriz Pnew se llama a la función 'Selection_II'
                                                   #devolviendo el mínimo valor de error
            indexMinimal = self.FitnessP.index(minimal) #Con ese valor mínimo se busca su índice en la
                                                        #matriz de error 'FitnessP'
            coordinatesMinimal = self.P[indexMinimal]   #y el valor de sus coordenadas se encuentra con ese índice
            if(coordinatesMinimal!=self.Coordinates):   #Si son diferentes a las actuales se actualizan los valores
Mejor solución encontrada
                self.Coordinates = self.P[indexMinimal] #Las coordenadas
                self.UnknowNodeList.update({self.nameKilobot:[self.Coordinates,self.ConfidenceFactor,min(self.FitnessP)]})
                                                        #y la matriz de los nodos con su nuevo valor calculado
                self.Broadcast_mode(['BSA',self.nameKilobot,self.UnknowNodeList]) #Se retransmite el diccionario de nodos
                self.Listen_mode()  #Y lo pone en escucha
            counter+=1
            self.Counter = counter  #Se almacena que ha hecho un ciclo
        else:
            self.Delete_memory()    #Si termina todos los ciclos devuelve True y borra la memoria
            return True
return False   #Para cualquier otro resultado devuelve False
```

Figura 3.26. Programación del algoritmo BSA

La figura 3.26 muestra el código donde se ejecutan las etapas restantes del algoritmo BSA. Antes de ejecutar esta parte del código se comprueba en la memoria de mensajes si ha habido alguna actualización de las coordenadas de los nodos vecinos. Se empieza revisando la **condición de parada**. Esta condición viene dada por el número de veces que se tiene que ejecutar el algoritmo BSA. Este número de veces se ajusta de forma empírica y en este caso se ha fijado a 20 veces. Una vez se revisa se entra en un bucle **`for`** que repite un numero de **`IterationsBSA`** veces los ciclos de mutación y selección. El numero **`IterationsBSA`** se ha ajustado a 10. Cada iteración sigue la estructura de la figura 2.26, (1) llama al método **`Selection_I()`** (**2**) elige la matriz P_{hist} (3). A continuación se genera la matriz de mutación con el método **`Mutation()`** este método devuelve una matriz de mutación siguiendo los pasos explicados en el apartado 2.6.4. Con las dos matrices ya generadas se procede a la fase de cruzado con **`Crossover_operation(PMod)`** y a la de selección final **`Selection_II(Pnew)`** (ya explicadas en el apartado 2.6.4), éste último método entrega las coordenadas con el menor error y en el caso de que sea mejor que las coordenadas ya existentes se actualizan.

3.3.6. Algoritmo completo

Para la ejecución del algoritmo completo se usa el método **`Complete_algorithm(Ts)`**. Este método ejecuta cada etapa del algoritmo "**distributed and resilient localization**" de forma ordenada y conjunta para que los Kilobots no estén en diferentes etapas y todos vayan saltando de etapa a etapa de forma conjunta.

```
#Algoritmo completo de localización de los kilobots
def Complete_algorithm(self,Ts):  #El algoritmo completo se ejecutara a partir de banderas que indiquen que
                                  #han ido haciendo cada etapa
    if(self.FlagStageI):          #Comprueba la bandera de la primera etapa
        if(self.Sum_Dist(Ts)):    #Una vez la primera función del algoritmo termina devuelve True
            self.FlagStageI = False #Desactiva la bandera de la primera etapa
            self.FlagStageIII = True #Activa la de la tercera ya que la segunda solo la hacen los nodos y
            if(self.Anchor!=True):   #es un cálculo rápido
                self.Coordinates,self.Delta=self.Min_Max()  #Aqui calcula las coordenadas los nodos desconocidos
            print('StageII; '+self.nameKilobot +':'+str(self.Coordinates)+'  Number of Anchors:'+str(len(self.AnchorList)))
            #Muestra en pantalla las coordenadas y el número de anclas detectados
            self.Delete_memory()     #Borra memoria
            return False
    elif(self.FlagStageIII):  #De forma análoga a la anterior se entra en la etapa 3
        if(self.MCMM(Ts)):
            self.FlagStageIII = False    #Una vez termina se almacena
            self.FlagConfidence = True   #Se activa la etapa en la que se comparte el factor de confidencia
            self.UnknowNodeList = {}     #Se reinicia el diccionario de información de los nodos
            self.State = True            #Y pone el estado que está listo para saltar a la siguiente etapa
            self.Delete_memory()
            print('StageIII; '+self.nameKilobot +':'+str(self.Coordinates))
            return False
    elif(self.FlagConfidence):
```

Figura 3.27. Programación del algoritmo completo (Etapa I,II y III)

En la figura 3.27. se muestra el inicio del código y como a partir de condicionales y la comprobación de las banderas de cada etapa, salta de una etapa a otra dependiendo del estado de estas banderas.

```
elif(self.FlagConfidence):
    if(self.State==True):      #Comprueba si está listo para la siguiente etapa y activa la función 'Swarm_state'
        if(self.Swarm_state()):#para que todos los kilobots pasen al mismo tiempo a esta etapa y no se mezclen datos
            self.State = False #Cuando devuelve true pone el estado en false para pasar a la siguiente etapa
            self.Delete_memory()
    elif(self.Confidance_factor()): #Llama a la función que calcula y comparte con todo el enjambre el factor
        self.FlagNeigborhood = True #de confianza
        self.FlagConfidence = False #Activa la bandera de la siguiente etapa y desactiva la del factor de confianza
        self.State = True
        self.Delete_memory()
    return False
elif(self.FlagNeigborhood):   #Este algoritmo es análogo al anterior, se llama a la función 'Swarm_state' para
    if(self.State==True):     #que todos los kilobots estén en la misma etapa
        if(self.Swarm_state()):
            self.State = False
            self.Delete_memory()
    elif(self.Node_two_hope_discover(Ts)): #Aquí activa la función que hace descubrir los nodos a 1 y 2 saltos
        self.FlagNeigborhood = False  #Desactiva la bandera de descubrir el vecindario y lo pone listo para saltar
        self.State = True             #a la siguiente etapa
        self.Delete_memory()
        return False
else:
```

Figura 3.28. Programación del algoritmo completo (factor de confianza y descubrimiento de vecindario)

Además de contener las etapas, también incluye otros métodos como compartir con el enjambre la información del factor de confianza de cada Kilobot o el descubrimiento de los vecinos a 1 y 2 saltos además de la distancia a la que se encuentran los vecinos a un salto. Esta programación se expone en la figura 3.28. Se puede ver que los saltos de estos métodos se llevan también a cabo a partir de banderas, pero también tiene un condicional en el interior que revisa el estado del Kilobot que lo inicia, una vez que el estado es **`True`** indica que ha terminado esa método o etapa y empieza la ejecución del método **`Swarm_state()`** para coordinar el salto de todo el enjambre de una etapa o método a otra.

```
else:
    if(self.State==True):
        if(self.Swarm_state()): #Esta parte es análoga a la anterior que sirve para que todos los kilobots estén en la
            self.State = False  #la misma etapa
            self.Delete_memory()
            if(self.FlagStageIV==True): #Aquí la diferencia es que la bandera de la fase IV se utiliza para indicar
                return True             #que se ha terminando el algoritmo y devuelve True
            else:                       #Con estos condicionales obliga a todos los kilobots a terminar al mismo tiempo
                return False
    elif(self.Refinament_with_BSA()):  #Se activa la función BSA para refinar la posición
        self.State = True              #Cuando termina lo pone en listo para saltar
        self.FlagStageIV = True        #Y activa la bandera indicando que ya ha terminado
        self.Delete_memory()
        return False
```

Figura 3.29. Programación del algoritmo completo (Etapa IV)

En la figura 3.29 se muestra la programación de la última etapa, la de refinamiento con el algoritmo BSA. Esta etapa tiene la misma estructura que las anteriores, a partir de banderas y del método **`Swarm_state()`**, pero con la diferencia de que cuando termina devuelve **`True`** para indicar que ya ha terminado el algoritmo.

3.4. Guiar a un enjambre de Kilobots hacia un foco de luz

Para llevar a cabo esta tarea se han realizado dos métodos diferentes dependiendo de si el Kilobot es el líder (el que guía a los demás Kilobots) o si es un Kilobot seguidor. Además de esto se han implementado dos métodos para la tarea de llevar el enjambre hacia la luz, ya que el **método de replicación de órdenes** es susceptible a fallos en los casos en los que se considera el ruido.

3.4.1. Método de replicación de órdenes

Este método hace uso del algoritmo "**Wave**" para la comunicación entre Kilobots. El algoritmo consiste en un Kilobot que guía al enjambre hacia un foco de luz de forma que comparte con el enjambre los movimientos que se está llevando a cabo para que así los demás Kilobots puedan copiarlos y moverse de forma coordinada y conjunta hacia un mismo sitio.

Como en el caso real se ha preparado dos códigos uno para el Kilobot líder y otro para el Kilobot seguidor:

- **Método para Kilobot líder**: El método que se utiliza en el código se llama **`Leader_replicate()`**, este método consiste en tomar la lectura del sensor de luz y a partir de la lectura girar a izquierda o derecha. A continuación, se comparte este movimiento con el enjambre. En la figura 3.30 se muestra el código completo:

```
#Función para guiar el enjambre hacia la luz como los Kilobots reales
def Leader_replicate(self):
    self.EnableReceived = 0                 #Desactiva la escucha de mensajes
    readings = self.Light_sensor()[1]       #Obtiene las lecturas del sensor de luz(coordenadas x e y relativas a el)
    angle = alpha(readings[0],readings[1])  #Calcula el ángulo de apertura respecto al foco
    if(self.Flag==False):                   #Inicialización del Kilobot
        self.Flag = True                    #Señaliza en la bandera que ya está inicializado
        self.Broadcast_mode('Left')         #Empieza moviéndose a izquierda y lo retransmite al enjambre
        self.Turn_left()                    #Llama a la función de girar a izquierdas
    else:                                   #Si ya está inicializado
        if(angle<300 and angle>180):        #Comprueba que el ángulo está en el rango de giro a derechas
            self.Broadcast_mode('Right')    #Si lo está lo retransmite
            self.Turn_right()               #y empieza a girar a derechas
        elif(angle>60 and angle<180):       #Si está en el rango de giro a izquierdas
            self.Broadcast_mode('Left')     #Lo retransmite y gira a izquierdas
            self.Turn_left()
```

Figura 3.30. Programación del Kilobot Líder en el método por réplica

El código empieza leyendo el sensor de luz que devuelve la posición relativa al Kilobot donde se encuentra el foco de luz. Para hacer un cálculo parecido al caso real se ha programado de forma que el Kilobot avance hacia el foco de luz con movimientos oscilatorios de derecha a izquierda. Como se da la posición del foco de luz, se calcula el ángulo que hay entre el **eje X** del sensor de luz y el **vector resultante** de la posición del foco de luz. En la figura 3.31 se puede ver una imagen para su mejor comprensión:

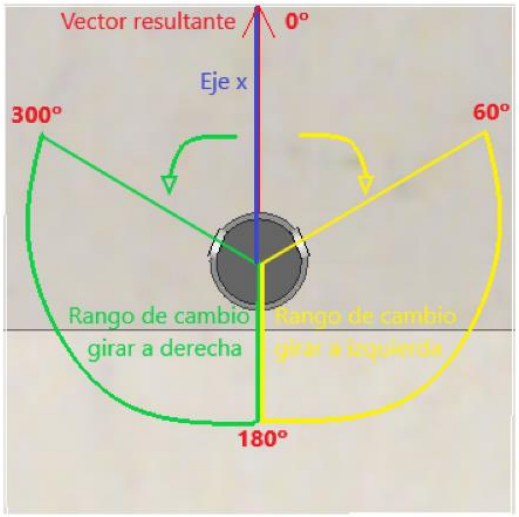


Figura 3.31. Imagen de un Kilobot con vectores del sensor y rangos de cambio de movimiento

El **vector resultante** de la posición del foco de luz es la dirección de avance, el cálculo se realiza entre el **eje x** (eje de orientación del Kilobot) y el **vector resultante**. Cuando el ángulo está entre 60º y 180º quiere decir que el **eje x** está en el **rango de cambio para girar a izquierda** por lo que ordena el cambio de movimiento y empieza a girar a la izquierda. De forma análoga sucede con **el rango de cambio para girar a derechas,** en este caso con los límites del ángulo entre 180º y 300º. Cada vez que se realiza un cambio en el movimiento se usa la función **`Broadcast_mode('Left'/'Right')`** para retransmitirlo al enjambre, se retransmite '***Left***' cuando está girando a izquierdas y '***Right***' cuando gira a derechas, esto permite al Kilobot seguidor copiar esta orden y así realizar el mismo movimiento.

- **Método para Kilobots seguidores**: El Kilobot que llame a este método sigue las órdenes del primer Kilobot que le envíe un mensaje. En la figura 3.32 se muestra el código de este método para su explicación:

```
#Función en la que los kilobots siguen a un líder
def Follower_replicate(self):
    if(self.MemoryCount!=0):                    #Comprueba si ha llegado algún mensajes
        if(self.Father==False):                 #Si no tiene padre
            self.Father = self.Memory[0][2]     #Asigna al remitente del primer mensaje como padre
        for message in self.Memory:             #Revisa todos los mensajes que le han llegado
            if(message[2]==self.Father):        #Si el remitente del mensaje es el padre
                self.Broadcast_mode(message[0]) #Retransimte la orden de giro del padre
                if(message[0]=='Left'):         #Si la orden es girar a izquierdas
                    self.Turn_left()            #Gira a izquierdas
                else:                           #Sino quiere decir que gire a derechas
                    self.Turn_right()
    self.Listen_mode()                          #Pone en modo escucha
    self.Delete_memory()                        #Borra la memoria de mensajes
```

Figura 3.32. Programación del método por réplica de órdenes para Kilobots seguidores

Inicia el código comprobando si le ha llegado algún mensaje y siguiendo la estructura del algoritmo "**Wave**" comprueba si el Kilobot tiene un Kilobot padre, si no lo tiene asigna como padre al remitente del primer mensaje que le ha llegado. Una vez asignado el Kilobot padre solo obedece los mensajes que le llegan de este y los demás los descarta, revisa toda la memoria de mensajes y hace las acciones que le ordena el Kilobot padre, esta orden la réplica al enjambre. Se replica el mensaje puesto que algún Kilobot puede tener un seguidor al que replicar el mensaje como Kilobot padre y así sucesivamente para llegar a todos los Kilobots del enjambre. Si en el mensaje contiene la orden '***left***' el robot gira a izquierda y sino gira a derecha.

3.4.2. Método basado en la distancia con realimentación

Este método hace también uso del algoritmo "Wave" pero incluyendo también la parte de realimentación, a diferencia del método anterior éste es más complejo y necesita más capacidad computacional y para el caso de los Kilobots reales no se puede implementar debido a su limitada memoria para el código. El funcionamiento de este método se basa únicamente en las distancias a las que se encuentran los robots vecinos. El Kilobot líder guía a todo el enjambre hacia la luz, pero también comprueba a la distancia a la que se encuentran los Kilobots seguidores de su Kilobot padre y si la distancia no es lo suficientemente cercana, para a todo el enjambre y envía una orden al Kilobot rezagado indicando que se aproxime a su Kilobot padre, una vez que está próximo a su padre el Kilobot líder inicia otra vez la marcha.

Al igual que en el método anterior el Kilobot padre tiene un método y los seguidores otro:

- **Método para el Kilobot Líder**: En el código se llama **`Leader_guide()`**, con este método se consigue que el Kilobot guíe el enjambre hacia la luz. En la figura 3.33 se muestra el código para su explicación:

```
#Función para guiar el enjambre hacia la luz con realimentación
def Leader_guide(self):
    if(self.MemoryCount==0):                        #Comprueba si hay algún mensaje
        self.Move_to_light()                        #Si no hay mensajes mueve hacia la luz
        self.Broadcast_mode([True,self.nameKilobot]) #y retransmite al enjambre que se muevan
    else:                                           #Si hay algún mensaje
        order = self.Revise_messages()              #Revisa los mensajes para saber qué orden hacer
        if(order==True):                            #Si la orden=True quiere decir que el enjambre está listo
            self.Move_to_light()                    #como está listo mueve el enjambre hacia la luz
        else:                                       #Si no está listo el enjambre se detiene para esperar al
            self.Straight(0)                        #al kilobot rezagado
        self.Broadcast_mode([order,self.nameKilobot])#retransmite la orden o el kilobot que se tiene que mover
    self.Listen_mode()        #lo pone en modo escucha
    self.Delete_memory()      #y borra la memoria
```

Figura 3.33. Programación del líder para el método realimentado

Se inicia comprobando que le ha llegado algún mensaje, si no le ha llegado envía la orden de movimiento e inicia el método **`Move_to_light()`**. Este método usa las lecturas del sensor de luz y su posición para moverse hacia la luz (de forma análoga al método explicado en la replicación de órdenes del apartado anterior). Cuando calcula una orden la retransmite al enjambre, con un vector de dos elementos, en el primer elemento esta la orden y en el segundo su nombre de ID. Las órdenes pueden ser dos:

- **`True`**: Esta orden significa que todos los Kilobots se encuentran próximos y se puede mover el enjambre.

- ***"Nombre ID del Kilobot rezagado"***: Esta orden se da cuando le llega por realimentación que un Kilobot está rezagado, por lo que envía el nombre de este Kilobot para indicar que se tiene que acercar a su Kilobot padre para poder proseguir con el avance.

Para saber cuál de estas dos órdenes utilizar se hace uso del método **`Revise_messages()`**, que se muestra en la figura 3.34.

```
#Función para revisar los mensajes y decidir qué hacer dependiendo de las circunstancias
def Revise_messages(self):
    if(self.Reading!=0):           #Revisa si hay algún Kilobot rezagado
        for message in self.Memory:#Si lo hay revisa en la memoria los mensajes de realimentación
            if(len(message[0])>2): #y si hay alguno que indique que el kilobot rezagado ya está cerca
                if(message[0][3]==self.Reading and message[0][1]<self.__ComunicationScope-0.02):
                    self.Reading = 0 #Lo vuelve a 0
                    return True      #devuelve True y comienza a moverse
        return self.Reading  #si no devuelve el Kilobot rezagado para que se acerque
    else:
        for message in self.Memory:#Revisa la memoria para comprobar si algún Kilobot está rezagado respecto a su padre
            if(len(message[0])>2):      #Comprueba que es un mensaje de realimentación con la longitud de este
                if(message[0][1]>self.__ComunicationScope-0.005): #Si algún kilobot esta más lejos que el
                    self.Reading = message[0][3]                  # rango de comunicación
                    return message[0][3] #Retransmite la orden de movimiento a ese kilobot devolviendo el nombre de este
        return True                       #Si no hay ningún Kilobot rezagado devuelve True para indicar que se mueva
    return self.Reading
```

Figura 3.34. Código para revisar la lista de mensajes llegados al líder

Primeramente, revisa con la variable **`Reading`** si algún Kilobot está rezagado. Comprueba, en los mensajes que le han llegado, los mensajes de realimentación que son los mensajes con más de 2 elementos (proveniente de seguidores) la estructura de estos mensajes son un vector de 4 elementos que contiene la orden del líder, la distancia a la que se encuentra su padre, el nombre del padre y el nombre del Kilobot que genera el mensaje. Estos mensajes son revisados en el primer bucle **`for`**, si el mensaje es del robot rezagado y la distancia a la que se encuentra de su padre está a 2 cm más cerca del alcance máximo comunicación (**`ComunicationScope`**) significa que ya no está rezagado y reinicia la variable **`Reading`** devolviendo **`True`** indicando que el enjambre puede iniciar la marcha. Si no hay ningún Kilobot rezagado, revisa en los mensajes si algún Kilobot está por encima del rango máximo de comunicación y lo asigna como rezagado y devuelve el nombre de este como orden que comparte con el enjambre para indicar que está rezagado y se tiene que acercar a su padre. Si no encuentra ningún rezagado devuelve **`True`** para indicar al enjambre que tiene que avanzar.

Si la orden es distinta de **`True`** el Kilobot líder se detiene para esperar al Kilobot rezagado. Las órdenes del líder es un vector de dos elementos con la orden y el nombre del Kilobot líder y una vez decidida la orden la comparte con los vecinos que estos a su vez la propagan a los demás como se verá a continuación.

- **Método para Kilobots seguidores**: En el código se llama **`Follow_leader()`**. Este método hace que los Kilobots seguidores sigan las órdenes provenientes del Kilobot líder. Para la explicación del método se hace uso de la figura 3.35. que muestra el código completo. Primeramente, se asigna como padre al remitente del primer mensaje que le ha llegado, a partir de este punto solo obedecerá las órdenes proveniente de este. En la comprobación de mensajes se hacen dos acciones una para los mensajes del padre y otra para los Kilobots hijos que lo tengan asignado como padre. Los mensajes del padre pueden tener 3 posibles acciones:

  - **Orden `True`**: Esta orden indica que no hay ningún Kilobot rezagado y que tiene que avanzar todo el enjambre. El Kilobot responde a esta orden aproximándose al remitente del mensaje (Padre), replicando la orden y enviando un mensaje de realimentación que contiene la orden, la distancia a la que se encuentra el padre, el nombre del padre y su propio nombre.

  - **Orden "*`Nombre ID del propio Kilobot`*"**: Si le llega como orden el propio nombre del Kilobot quiere decir que está rezagado, por lo que le ordena que se acerque al padre. El Kilobot remite una orden de avance (**`True`**) para sus Kilobots hijos y un mensaje de realimentación como el descrito anteriormente en la orden **`True`**.

  - **Orden "*`Nombre ID de un Kilobot rezagado`*"**: Si le llega otro nombre ID quiere decir que un Kilobot está rezagado y que por lo tanto hay que esperarlo por lo que detiene el avance y réplica el mensaje.

```
#Función en la que los kilobots siguen a un lider como en un caso real
def Follow_leader(self):
    if(self.MemoryCount!=0):      #Comprueba si hay algún mensaje
        if(self.Father==False):   #Si no tiene padre se lo asigna como el remitente del primer mensaje
            self.Father = self.Memory[0][2]
        for message in self.Memory:  #Bucle que revisa los mensajes de la memoria
            if(message[2]==self.Father and len(message[0])<3):#Si algún mensaje es del padre y no es de realimentación
                if(message[0][0]==True):      #Comprueba si la orden es True que indica que se mueva
                    self.Aproach(message[1])  #Se mueve en dirección del que le ordena la orden basándose en
                                              #la distancia de este mensaje
                    self.Broadcast_mode(message[0])#Retransmite la orden para los hijos
                    self.Broadcast_mode([True,message[1],self.Father,self.nameKilobot]) #Y la realimentación para
                                                                                        #el padre
                elif(message[0][0]==self.nameKilobot):#Si el mensaje contiene su nombre por orden quiere decir
                    self.Aproach(message[1])          #que está rezagado por lo que le indica que se acerque
                    self.Broadcast_mode([True,self.nameKilobot]) #Retransmite la orden para los hijos
                    self.Broadcast_mode([True,message[1],self.Father,self.nameKilobot])#Y la realimentación para
                                                                                       #el padre
                else:               #Si le llega otra orden diferente se para (indica que la orden no es para él)
                    self.Straight(0) #entonces se para esperando a que se mueva el padre otra vez
                    self.Broadcast_mode(message[0])#Retransmite la orden para los hijos
            if(len(message[0])>3):     #Comprueba los mensajes de realimentación
                if(message[0][2]==self.nameKilobot): #si algún mensaje indica que el padre del remitente es el mismo
                    self.Broadcast_mode(message[0])  #Lo retransmite (es el mensaje de un hijo)
    self.Listen_mode()    #Lo pone en modo escucha y borra memoria
    self.Delete_memory()
```

Figura 3.35. Programación de los seguidores por el método realimentado

Para los mensajes provenientes de los hijos se comprueban que sean mensajes de realimentación con la longitud del mensaje ya que tienen una estructura de cuatro elementos. Si el mensaje es de realimentación se comprueba que el remitente del mensaje lo tenga asignado como padre, si se da este caso réplica este mensaje para que le llegue al Kilobot líder para poder evaluar la acción que se llevara a cabo en base a estos mensajes de realimentación.

En la figura 3.36 se muestra el código para que un Kilobot se aproxime a otro introduciéndole la distancia a la que se encuentra el remitente del mensaje.

```
#Función para acercarse a un objetivo remitente del valor introducido "distance"
def Aproach(self,distance):
    if(distance<=0.036):          #Si la distancia es menor a 0.036m no se acerca mas
        self.Straight(0)          #Lo para
    else:
        if(self.Variable_timer(0.5)): #El muestreo se realiza cada 0.5s
            if(distance>=self.Reading):   #Si la distancia introducida es mayor que hace 0.5s realiza el cambio
                if(self.Motor=='Left'):   #Lo que hay que alternar de motor, si el motor que estaba
                    self.Turn_right()     #moviendose es el izquierdo cambia el sentido de giro a derecha
                    self.Motor='Right'    #Anota el actual sentido de giro
                else:                     #En otro caso
                    self.Turn_left()      #Cambia el sentido de giro a izquierda
                    self.Motor='Left'     #y anota el sentido
            self.Reading = distance       #Actualiza la nueva lectura
```

Figura 3.36. Función para que un Kilobot se aproxime a otro en base a la distancia

Se le introduce como parámetro de entrada la distancia del remitente, se comprueba si está demasiado cerca como para acercarse y si lo está se detiene. Sino se activa un temporizador ya que el cambio de movimiento se realiza cada 0.5 segundos, esto evita que se hagan cambios rápidos sin que le dé tiempo a haber cambios significativos en la distancia. Si la distancia es igual o mayor que la distancia almacenada cambia el sentido de giro y actualiza la distancia almacenada a la actual. Este bucle se repite continuamente consiguiendo acercarse al remitente alternando los movimientos de izquierda y derecha.

En este capítulo se explica el código utilizado en los Kilobots reales, su programación y preparación del área de trabajo. Debido a las limitaciones de memoria del Kilobot se ha tenido que adaptar el código para que funcione en los Kilobots reales dando a variaciones significativas en el código comparado con el código de simulación.

4.1. Calibración y disposición del área de trabajo

La calibración se realiza manualmente con la aplicación **KiloGUI** ya explicada en capitulos anteriores. La superficie en la que se ha realizado los ensayos es de cristal, en la tabla 4.1 se muestran los valores de calibración usados en los Kilobots:

ID Kilobot	Giro a izquierda	Giro a derecha	Avance lineal	
0	74	79	65	66
1	70	71	62	64
2	74	72	61	59
3	71	91	56	71
8	61	72	58	62
9	70	68	62	63

Tabla 4.1. Valores de calibración de los Kilobots

Para preparar el área de trabajo en el algoritmo en el que los Kilobots van hacia la luz, se utiliza una mesa con un foco de luz colocado sobre una esquina del área de trabajo y para los ensayos del algoritmo "**distributed and reisilent localization**" se utiliza como superficie de trabajo papel milimétrico como se muestra en la figura 4.1:

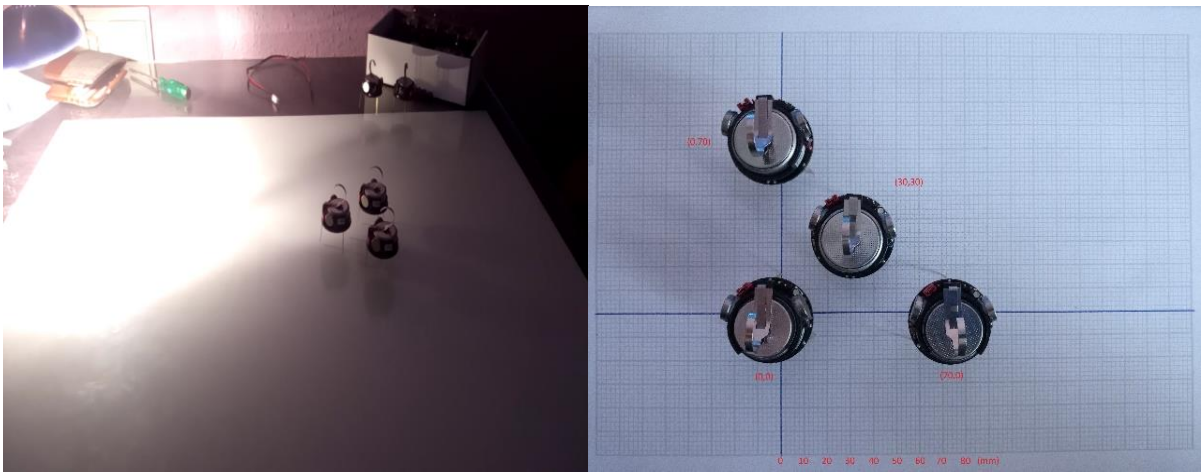


Figura 4.1. Imagen del área de trabajo para los ensayos, izquierda método réplica de órdenes y derecha superficie con papel milimétrico

4.1.1. Funciones comunes usadas en todo el código

Antes de entrar en el código se explican varias funciones que serán comunes en el código ya que son funciones para la comunicación entre Kilobots que serán la base de estos algoritmos:

- **`Broadcast_Mode (Data)`**: Esta función retransmite un dato ***`Data`*** a los Kilobots vecinos, la estructura que tiene se muestra en la figura 4.2:

```
//Función para retransmitir las coordenadas
void Broadcast_Mode(uint8_t data[4]){
    MessageToSend.type = NORMAL;                    //Inicialización del mensaje a enviar
    uint8_t count=0;                                //Este contador se utiliza en el bucle while
    while(count<4){                                 //Con este bucle recorre el mensaje a enviar para almacenarlo
        MessageToSend.data[count] = data[count];    //Lo almacena en la variable "MessageToSend"
        count++;
    }                                               //La función message_crc se tiene que ejecutar y guardar en
    MessageToSend.crc = message_crc(&MessageToSend); //el mensaje a enviar para que el receptor no lo descarte

}
```

Figura 4.2. Código ejemplo de la función para retransmitir mensajes

En el ejemplo de la figura 4.2 se usa una variable **MessageToSend** del tipo **message_t** para almacenar el mensaje a enviar. Primero se define el tipo de mensaje con el atributo **.type** siempre se define como **NORMAL.** Las variables del tipo **message_t** pueden almacenar un total de 9 datos de 8 bits cada uno, por lo que se recorre los atributos **.data** almacenando el dato ***data*** en la variable **MessageToSend** y a continuación se llama a la función **message_crc(*MessageToSend*)** , para guardar el **CRC** en la variable del mensaje y enviar el mensaje por IR.

- ***Message_pointer()** : Esta función se usa conjunto a la función **kilo_message_tx** para registrar la devolución de la función ***Message_pointer()** como la variable encargada de retransmitir los mensajes. Cada vez que se cambien los valores de esta variable se retransmitirá por el puerto IR.

- **Tx_Message_Success()** : Función para registrar que un mensaje se ha retransmitido con éxito se usa de forma conjunta con la función **kilo_message_tx_success** de forma análoga a la función anterior (***Message_pointer()**). Cada vez que se retransmite un mensaje el Kilobot llama a esta función de forma autónoma, esta función contiene una bandera para poder llevar un seguimiento de si se ha retransmitido el mensaje y cada vez que se envía un mensaje se tiene que reiniciar esta bandera.

- **Message_Store** *(*message,*distance)* : Función para leer la memoria donde ha llegado un mensaje, se tiene que registrar con la función **kilo_message_rx**. En la figura 4.3 se muestra un ejemplo de uso:

```
//Función para almacenar el mensaje que le ha llegado en la variable "RecivedMessage"
void Message_Store(message_t *message, distance_measurement_t *distance_measurement){
    NewMessageFlag = 1;                     //Señaliza que hay un nuevo mensaje
    RecivedMessage[0] = message->data[0];   //Lo almacena en la variable "RecivedMessage"(Aqui la ID del Ancla)
    RecivedMessage[1] = message->data[1];   //Aquí la posición x en mm
    RecivedMessage[2] = message->data[2];   //Aquí la posición y en mm
    Distance = estimate_distance(distance_measurement); // Aquí la distancia a la que se encuentra el remitente
}
```

Figura 4.3. Código ejemplo de la función que lee un mensaje llegado

Siguiendo la estructura para leer un mensaje de entrada ****message*** que le ha llegado por IR, se va almacenando en una variable como en el caso del ejemplo en la variable **RecivedMessage** además de esto se calcula la distancia del remitente con la entrada ****distance*** y el uso de la función **estimate_distance** *(*distance)* y esta devuelve un valor que se puede almacenar en otra variable. Esta función se llama de forma autónoma por el Kilobot y se señaliza la recepción de un mensaje con la bandera **NewMessageFlag**. Comprobando esta bandera se puede saber si le ha llegado un mensaje al Kilobot y que está listo para su uso en la variable que se le ha asignado, en el caso del ejemplo las variables **RecivedMessage** y **Distance**, una vez usados los datos y si no se necesitan más se reinicia la bandera indicando que no hay ningún mensaje nuevo permitiendo la recepción de un nuevo mensaje.

- **Set_Motion(*NewMotion*)** : Esta función controla el movimiento del Kilobot. Permite al Kilobot moverse alternando el giro de izquierda y derecha. En la figura 4.4 se muestra el código:

```
//Función que cambia los motores para moverse hacia un lado u otro dependiendo de la entrada "NewMotion"
void Set_Motion(char NewMotion){
    if (CurrentMotion != NewMotion){   //Solo se mueve si cambia de movimiento
        CurrentMotion = NewMotion;     //Si es diferente la actualiza
        spinup_motors();//Función para que el Kilobot se mueva durante un breve tiempo a máxima potencia para vencer el rozamiento del suelo
        if (CurrentMotion == Left){  //Si el movimiento es Left gira a la izquierda
            set_motors(kilo_turn_left, 0);//para el motor derecho y lo hace girar con el valor almacenado en la calibración
        }
        else if (CurrentMotion == Right){  //Si el movimiento es Right gira a la derecha
            set_motors(0, kilo_turn_right);//Análogo a la anterior
        }
        Broadcast_Movement(CurrentMotion);//Una vez actualizado el movimiento lo retransmite al enjambre
        MessageSentFlag=0;
    }
}
```

Figura 4.4. Código de la función Set_Motion

El carácter de entrada ***NewMotion*** define si se tiene que cambiar de movimiento dependiendo de si es diferente al actual movimiento. Si ***NewMotion*** es diferente del actual movimiento (**CurrentMotion**), la variable **CurrentMotion** se actualiza como ***NewMotion*** y se cambia el sentido de giro en función de la variable **CurrentMotion**. Una vez cambiado el giro se retransmite al enjambre el actual sentido de giro ya que esta función se utiliza en el método para guiar a los Kilobots hacia un foco de luz y se tiene que retransmitir la orden para que lo puedan copiar los Kilobots seguidores.

- **loop():** Esta función contiene el algoritmo principal de cada método y varía dependiendo de lo que se desea programar. Se ejecuta en bucle a través de la función "**kilo_start()**".

- **setup():** Función que se ejecuta una vez al inicio de la ejecución del código. Contiene la inicialización del Kilobot dependiendo del método que se quiera ejecutar. Al igual que la función **loop()** se ejecuta a través de la función **kilo_start()**.

- **main()**: Función principal del código donde se haya el código de preparación y la función **loop()** que se ejecutará de forma continua, un ejemplo se muestra en la figura 4.5:

```
//Esta es la funcion main donde se ejecuta la inicializacion del Kilobot y la funcion que llama al codigo que se ejecutara
int main(){
    kilo_init();                                //Función de inicialización del hardware del Kilobot
    kilo_message_tx = Message_Pointer;          //registra la funcion de llamada
    kilo_message_tx_success = Tx_Message_Success;  //registra la funcion de exito de llamada
    kilo_message_rx = Message_Store;            //registra la funcion de llegada de un mensaje
    kilo_start(setup, loop);                    //Función para inciar el bucle principal y la inicialización
    return 0;
}
```

Figura 4.5. Código ejemplo de la función main

En esta función se ejecuta la inicialización del hardware con la función **kilo_init()** y se registran las funciones referentes a la comunicación antes ya explicadas, una vez hecho esto se llama a la función "**kilo_start(*setup*,*loop*)**" ya explicada en apartados anteriores, que ejecutará una sola vez la función ***setup*** y de forma continua el código en la función ***loop*** introducida.

4.2. Adaptación "distributed and resilient localization"

Debido a las limitaciones de memoria de los Kilobot solo se puede programar una parte de este algoritmo. Se implementan las dos primeras etapas (Etapas) del algoritmo, con tres Kilobots anclas y un Kilobot nodo. Se ha realizado con un solo Kilobot nodo ya que para mostrar los resultados se usa el modo **debug** y el cable serial para mostrar los resultados en la pantalla del ordenador al que se conecta el OHC.

Para la programación de los Kilobots reales, se usa un código para los Kilobots anclas y otro diferente para los Kilobots nodo que se exponen a continuación (para una comprensión más detallada se puede ver el código comentado en el **Anexo III**).

4.2.1. Código para los Kilobots anclas

El código para los Kilobots anclas es muy simple, ya que únicamente lo que realizan es compartir con el enjambre su posición en el eje x e y, su ID de identificación y la distancia recorrida por el mensaje (que siempre es 0 debido a que solo hay un Kilobot nodo), aunque no se use este último dato se ha añadido por respetar la estructura del algoritmo. En la figura 4.6 se muestra la programación del Kilobot Ancla real:

```
//Función de inicialización del código
void setup(){                 //Inicializa las coordenadas de los anclas dependiendo de su número de ID
    if(kilo_uid==0){
        Coordinates[0] = 0; //Si es el Kilobot 0 se pone en la posición (0,0)mm
        Coordinates[1] = 0;
    }
    else if(kilo_uid==1){
        Coordinates[0] = 70; //Si es el Kilobot 1 se pone en la posición (70,0)mm
        Coordinates[1] = 0;
    }
    else if(kilo_uid==2){
        Coordinates[0] = 0; //Si es el Kilobot 2 se pone en la posición (0,70)mm
        Coordinates[1] = 70;
    }
}

//Función principal donde se pone el código que se va a ejecutar
void loop(){                          //Esta estructura de mensajes es la que se hace en el algoritmo
    uint8_t v[4] = {kilo_uid,Coordinates[0],Coordinates[1],0}; //Almacena su ID y sus coordenadas y la distancia Lr en la variable v
    Broadcast_Mode(v);                                         //Llama a la función que lo retransmite
    MessageSentFlag = 0;
    delay(rand_soft()*2);       //Espera un tiempo aleatorio hasta enviar un nuevo mensaje (se hace para que no se solapen)
}
```

Figura 4.6. Código principal para un Kilobot ancla

Como se muestra en la figura en la función **`loop()`** es donde se ejecuta el código principal, y en la función **`setup()`** se definen las coordenadas en las que se encuentra el Kilobot ancla dependiendo del número ID ya calibrado en la etapa de calibración. La función **`loop()`** se ejecuta continuamente, lo que hace primeramente es guardar en un vector ***v*** de 4 elementos los datos número ID, coordenada x, coordenada y y distancia recorrida por el mensaje. Lo retransmite al enjambre con la función **`Broadcast_Mode(v)`** y espera un tiempo con la función **`delay(number)`** que para los ensayos reales ***number*** es un número aleatorio proporcionado por la función **`rand_soft()`**.

4.2.2. Código para el Kilobot nodo

Debido a las limitaciones del Kilobot solo se ha programado la primera y segunda etapa (**Etapa I** y **Etapa II**) del algoritmo "**distributed and resilient localization**". Este código se ha usado en un solo Kilobot conectado al controlador OHC para poder comprobar los resultados en el ordenador. La programación de estas dos etapas se muestran en la figura 4.7:

```
void loop(){
  //Stage I del algoritmo
  if(Anchors[0][3]==0 || Anchors[1][3]==0 || Anchors[2][3]==0){//Si no tiene los datos de algun ancla no para de repetir el proceso
      if(NewMessageFlag==1){                                   //Comprueba si hay un nuevo mensaje
          NewMessageFlag = 0;                                  //Reinicia la bandera
          Anchors[RecivedMessage[0]][0]=RecivedMessage[0]; //Almacena la ID del remitente
          Anchors[RecivedMessage[0]][1]=RecivedMessage[1]; //Su posición en x
          Anchors[RecivedMessage[0]][2]=RecivedMessage[2]; //Su posición en y
          Anchors[RecivedMessage[0]][3]=Distance;          //Y la distancia recorrida por el mensaje
      }
  }
  //Stage II del algoritmo
  else if(FlagMin_Max==0){                    //Una vez terminada la fase de recopilación de datos de los anclas
      Min_Max();                              //Se inicia el metodo Min_Max para la obtención de las coordenadas
      printf ( "Coordenadas nodo: (%d,%d) " , Coordinates[0],Coordinates[1]); //Envía por el serial las coordenadas para que se puedan ver
      FlagMin_Max = 1;                //Pone la bandera en 1 para que pare de realizar el código indicando que ha terminado
  }
}
```

Figura 4.7. Código principal para Kilobot nodo

En la primera etapa (**Etapa I**) se ha programado el **algoritmo sum-dist** pero sin temporizador ya que la función que tiene integrada el Kilobot para usar un temporizador ocupa mucho espacio de memoria de programación y por lo tanto no se podría programar la segunda etapa, en sustitución a esto se ha puesto una condición para terminar esta etapa, esta condición es que en la lista donde se almacena toda la información de los anclas se tiene que tener la información de todos los Kilobots anclas del ensayo. Una vez revisada la condición de fin del algoritmo se revisa si le ha llegado algún mensaje, si es el caso se almacena en la lista de anclas **`Anchors`**, para almacenar la información se ha utilizado el numero ID de los Kilobots como índice por lo que es muy importante que su número ID sea 0,1 y 2 puesto que si no lo son se saldría del rango de la matriz de información de los Kilobots anclas **`Anchors`**.

Una vez se tiene la información de todos los Kilobots anclas, se salta a la etapa 2 (**Etapa II**). Esta etapa tiene una comprobación de la bandera **`FlagMin_Max`** para que no se repita el proceso continuamente y envíe continuamente la información por el puerto serie saturando la ventana que muestra estos resultados. En la figura 4.8 se muestra el código del método **Min-Max** que también ha tenido que ser modificado. Primero crea las matrices que contienen los rangos de los cuadrados delimitadores B_i de cada ancla y después calcula la intersección S_i haciendo uso de la función **`Max()`** y **`Min()`** que devuelve el máximo y mínimo de un vector de tres elementos. Una vez calculado la intersección almacena las coordenadas del centro de este cuadrado intersección.

```
//Función que calcula la posición con los datos de los anclas usando el método Max_min
void Min_Max(){
    //Aquí almacena los valores de los cuadros delimitadores Bi de un nodo
    int8_t Bir1x[3] = {Anchors[0][1]-Anchors[0][3],Anchors[1][1]-Anchors[1][3],Anchors[2][1]-Anchors[2][3]};
    int8_t Bir1y[3] = {Anchors[0][2]-Anchors[0][3],Anchors[1][2]-Anchors[1][3],Anchors[2][2]-Anchors[2][3]};
    int8_t Bir2x[3] = {Anchors[0][1]+Anchors[0][3],Anchors[1][1]+Anchors[1][3],Anchors[2][1]+Anchors[2][3]};
    int8_t Bir2y[3] = {Anchors[0][2]+Anchors[0][3],Anchors[1][2]+Anchors[1][3],Anchors[2][2]+Anchors[2][3]};
    int8_t Si[2][2];       //Aquí se almacena la intersección de los cuadros
    Si[0][0] = Max(Bir1x); //Calculando su máximo
    Si[0][1] = Max(Bir1y);
    Si[1][0] = Min(Bir2x); //y su mínimo
    Si[1][1] = Min(Bir2y);
    Coordinates[0] = (Si[0][0]+Si[1][0])/2; //calcula el centro de dicho cuadrado que será la posición estimada
    Coordinates[1] = (Si[0][1]+Si[1][1])/2;
}
```

Figura 4.8. Código método Min-Max para Kilobot real

Una vez ha finalizado el método **Min-Max,** se envían las coordenadas del centro del cuadrado intersección por el cable serial haciendo uso de la función **`printf()`** y se pone la bandera **`FlagMin_Max`** a uno indicando que ha terminado el cálculo de su posición.

4.3. Caso de enjambre que se mueve hacia la luz

Al igual que en el caso anterior, no se puede realizar la programación del modelo realimentado, ya que el código para el Kilobot líder ocupa más de los 32Kb permitidos en la programación de un Kilobot. El modelo que si se puede implementar es el de réplica de movimientos ya que es un método muy simple y ocupa poco espacio.

4.3.1. Código de réplica de movimientos para Kilobot líder

Este código permite al Kilobot líder guiar al enjambre hacia un foco de luz alternando el giro del Kilobot al igual que en la simulación. Al igual que en la simulación el Kilobot tiene un rango de giro a izquierda y otro de giro a derechas, estos rangos vienen dados por la lectura del sensor de luz que dependiendo de la cantidad de luz llegada da un valor u otro aprovechando la arquitectura del Kilobot y la posición del sensor de luz. Estos valores han sido ajustados en base a un código de ejemplo proveniente de la página web "**labs**" de Kilobotics **[43]**. En la figura 4.9 se muestra el código principal del método de réplica de movimientos para el Kilobot líder:

```
//Función de inicialización del código
void setup(){
    spinup_motors();//Esta función hace que el Kilobot se mueva durante un breve tiempo a maxima potencia para vencer el rozamiento del suelo
    set_motors(kilo_turn_left, 0);//el rozamiento del suelo, después se pone a girar a la izquierda para asegurar movimiento
}

//Función principal donde se pone el código que se va a ejecutar
void loop(){
    Sample_Light();                 //Empieza calculando la luz y almacenando en la variable "CurrentLight"
    if (CurrentLight < 300){        //Si es más bajo que el umbral menor quiere decir que no está bien orientado
        Set_Motion(Right);          //y lo hace girar a la derecha
    }
    else if (CurrentLight > 600){   //Si supera el umbral superior lo gira a la izquierda
        Set_Motion(Left);
    }
}
```

Figura 4.9. Código principal del líder para que un Kilobot líder guie a un enjambre hacia la luz

Se inicializa girando a izquierdas para asegurar siempre que empieza en movimiento. Inicia el código con la función **`Sample_light()`**, esta función calcula la lectura del sensor de luz y lo almacena en la variable **`CurrentLight`** la función **`Sample_light()`** calcula la lectura del sensor haciendo la media de varias muestras recogidas del sensor de luz. Si la lectura del sensor está en el rango de giro a derechas (300) ordena que se gire a derecha con la función **`Set_Motion()`** y si está en el rango de giro a izquierdas (600) gira a izquierdas. Hay que tener en cuenta que dentro de la función **`Set_Motion()`** se llama a la función **`Broadcast_Mode()`** para retransmitir al enjambre el sentido en el que se está girando que además de compartir el sentido de giro también se comparte el numero ID del Kilobot para que los Kilobots seguidores puedan identificar los mensajes del Kilobot padre.

4.3.2. Código de réplica de movimientos para Kilobot seguidor

Los Kilobots seguidores al igual que en simulación seguirán parte de la estructura del algoritmo "**Wave**", copiando las órdenes provenientes de un Kilobot padre asignado como el primer Kilobot que le envíe un mensaje. Los mensajes contienen el sentido de giro con el que está girando el remitente además del número ID del Kilobot. Una vez se haya ajustado el sentido de giro se retransmite al enjambre el sentido con el que está girando por si algún Kilobot seguidor lo tiene asignado como padre.

```
//Función de inicialización del código
void setup(){
    spinup_motors();//Función para que el Kilobot se mueva durante un breve tiempo a maxima potencia para vencer el rozamiento del suelo
    set_motors(kilo_turn_left, 0);//el rozamiento del suelo, después se pone a girar a la izquierda para asegurarse movimiento
}

//Función principal donde se pone el código que se va a ejecutar
void loop(){
    if(NewMessageFlag==1){                  //Si hay mensaje comprobando el estado de la bandera
        NewMessageFlag = 0;                 //Reinicia la bandera
        if(Father==-1){                     //Si no tiene padre
            Father = RecivedMessage[0];     //lo almacena al primero que le envíe el mensaje
            }
        if(RecivedMessage[0]==Father){      //Si el mensaje es del padre
            Set_Motion(RecivedMessage[1]); //Lo lee y modifica el movimiento de acuerdo con el asignado
            }
        }
}
```

Figura 4.10. Código principal del seguidor para que el enjambre vaya hacia la luz guiado por un líder

En la figura 4.10. se muestra el código principal, se inicializa girando a izquierda como el Kilobot líder. El Kilobot comprueba si le ha llegado algún mensaje, una vez le llega el primer mensaje si no tiene padre (**`Father=-1`**) se guarda el remitente del primer mensaje en la variable **`Father`**, después revisa si el remitente de los mensajes es el padre y si lo es actualiza el sentido de giro copiando el retransmitido por el Kilobot asignado como padre. Nótese que en la función **`Set_motion()`** al igual que en el apartado anterior retransmite el sentido de giro y la ID del Kilobot.

En este capítulo se muestran los resultados obtenidos por los ensayos realizados en simulación y los realizados con Kilobots reales además se discute de las diferencias y problemas encontrados durante el desarrollo de ambos casos.

5.1. Resultados de la simulación

Durante la simulación mediante CoppeliaSim se han encontrado problemas por limitaciones del ordenador, ya que al usarse POO se tiene que actualizar constantemente los estados de los Kilobots y al ejecutarse el código conjunto a la simulación se producían unos pequeños retardos debidos a la comunicación entre Jupyter y CoppeliaSim, para solucionar esto se ha ejecutado la simulación a una velocidad prudente para darle margen de tiempo a Jupyter a ejecutar el código y reducir al máximo el retardo entre la ejecución del código y la simulación de CoppeliaSim.

5.1.1. Ensayos "distributed and resilient localization"

Para el algoritmo "**distributed and resilient localization**" se han realizado varios ensayos variando la posición de un Kilobot al igual que en el caso real y así poder comparar resultados. Además de variar la posición también se ha variado el tiempo de ejecución de los algoritmos **Sum-Dist** y así ver las variaciones debido al tiempo.

Para la simulación de los ensayos se ha usado el método **`Complete_algorithm(Ts)`**, que dependiendo del ensayo se ha seleccionado distinto tiempo ***Ts***. En todos los ensayos se ha tomado como Kilobots ancla los **Kilobots 0, 2 y 4** tomando al Kilobot 0 como eje de referencia por lo que su posición es la (0,0).

- **Ensayo 1**: En el primer ensayo se ha realizado con un tiempo ***Ts*** de 5 segundos y una distribución de **4 Kilobots nodo** como la mostrada en la figura 5.1:

Figura 5.1. Disposición del ensayo 1 y 2

Las coordenadas de los Kilobots es la que se definió por defecto y vienen descritas en la figura 3.11. proveniente de capítulos anteriores. Los resultados obtenidos se muestran en la tabla 5.1:

ID Kilobot	Numero de anclas	Etapa II (mm)	Etapa III (mm)	Etapa IV (mm)
1	2	(0,70)	(0,70)	(-0.5,70)
3	2	(70,0)	(70,0)	(70,0)
5	3	(31,31)	(31,31)	(25,33.4)
6	3	(48.5,48.5)	(48.5,48.5)	(54.4,59)

Tabla 5.1. Resultados ensayo 1(Simulación)

Con la tabla 5.2 se puede comparar el error cometido en cada caso:

ID Kilobot	Coordenadas reales (mm)	Etapa II Error (mm)	Etapa III Error (mm)	Etapa IV Error (mm)
1	(0,75)	(0,5)	(0,5)	(0.5,5)
3	(75,0)	(5,0)	(5,0)	(5,0)
5	(30,30)	(1,1)	(1,1)	(5,3.4)
6	(65,65)	(16.5,16.5)	(16.5,16.5)	(10.6,6)

Tabla 5.2. Error cometido en ensayo 1 (simulación)

Como se puede ver en la tabla de error la ejecución del algoritmo BSA (**Etapa IV**) no siempre mejora el resultado ya que para el Kilobot 1 y 5 ha aumentado el error al realizar esta etapa de refinamiento. Esto se puede solucionar proporcionando al algoritmo más iteraciones para que pueda afinar más el resultado, pero esto llevaría a una mayor carga computacional y se tardaría más en el proceso.

- **Ensayo 2**: Este segundo ensayo se realiza con una misma distribución de Kilobots, pero con un tiempo Ts de 7 segundos para darle a todos los Kilobots margen de tiempo para que le llegue información de todos los Kilobots anclas del enjambre y comprobar si mejoran los resultados en comparación con el ensayo anterior. En la tabla 5.3 se muestran los resultados obtenidos:

ID Kilobot	Numero de anclas	Etapa II (mm)	Etapa III (mm)	Etapa IV (mm)
1	3	(16,75)	(16,75)	(3.2,75)
3	3	(75,16)	(75,16)	(75,-0.5)
5	3	(31,31)	(31,31)	(24.3,33.7)
6	3	(48.5,48.5)	(48.5,48.5)	(61.2,56.3)

Tabla 5.3. Resultados ensayo 2 (simulación)

ID Kilobot	Coordenadas reales (mm)	Etapa II Error (mm)	Etapa III Error (mm)	Etapa IV Error (mm)
1	(0,75)	(16,0)	(16,0)	(3.2,0)
3	(75,0)	(0,16)	(0,16)	(0,0.5)
5	(30,30)	(1,1)	(1,1)	(5.7,3.7)
6	(65,65)	(16.5,16.5)	(16.5,16.5)	(3.8,8.7)

Tabla 5.4. Error cometido en ensayo 2 (simulación)

Como se puede ver en la tabla 5.4 el Kilobot 1, 3 y 6 han mejorado el error considerablemente. A diferencia del ensayo anterior, en el **Etapa II** se comete más error en el Kilobot 1 y 3. El **Etapa III** en este ensayo es irrelevante ya que al tener información de 3 anclas no se realiza esta etapa al no ser necesaria. Los resultados del error son mejores en el **Etapa IV** en este ensayo que en el anterior, pero esto es mera casualidad ya que se refina la posición de una manera aleatoria y no influye el número de anclas detectados.

- **Ensayo 3**: Se ha realizado este ensayo con el mismo número de **Kilobots nodo**, pero con una distribución diferente y un tiempo Ts de 7 segundos, la distribución es la de la figura 5.2:

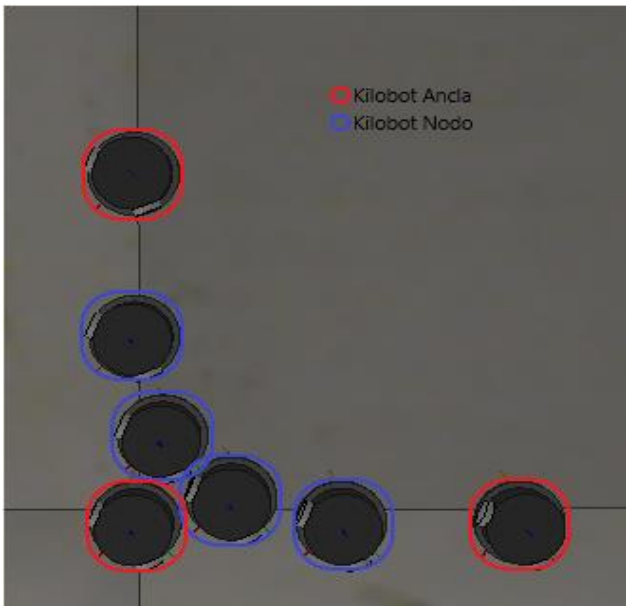


Figura 5.2. Disposición del ensayo 3

La variación en la distribución es del Kilobot 5 y 6, su posición es (10,35) y (35,10) respectivamente. Se han hecho con estas coordenadas para comparar los resultados con el caso real. En la tabla 5.5 se muestran los resultados obtenidos:

ID Kilobot	Numero de anclas	Etapa II (mm)	Etapa III (mm)	Etapa IV (mm)
1	3	(17,70)	(17,70)	(-6,70)
3	3	(70,17)	(70,17)	(70,17)
5	3	(18,35)	(18,35)	(0.4,34.3)
6	3	(35,18)	(35,18)	(35,18)

Tabla 5.5. Resultados ensayo 3 (simulación)

ID Kilobot	Coordenadas reales (mm)	Etapa II Error (mm)	Etapa III Error (mm)	Etapa IV Error (mm)
1	(0,75)	(17,5)	(17,5)	(6,5)
3	(75,0)	(5,17)	(5,17)	(5,17)
5	(10,35)	(8,0)	(8,0)	(9.6,0.7)
6	(35,10)	(0,8)	(0,8)	(0,8)

Tabla 5.6. Error cometido en ensayo 3 (simulación)

Se puede observar en la tabla 5.6 que al igual que en los ensayos anteriores ocurre lo mismo. Por lo que se muestra en la simulación los Kilobots que se encuentran colocados más céntricos en la región creada por los tres Kilobots ancla obtienen mejores resultados en el cálculo de las coordenadas que los que se encuentran en los límites de esta región. Pero los Kilobots que se encuentran en los limites les funciona mejor el **Etapa IV** aunque esto se debe simplemente a casualidad ya que la actualización de la posición es al azar, en **[24]** se realizan ensayos de localización parecidos ya que se realizan con Kilobots, pero con otra herramienta de simulación y con un enjambre mayor. Los resultados obtenidos son parecidos a los obtenidos en estos ensayos, en la figura 5.3. se muestra uno de los ensayos realizados en **[24]** con 32 Kilobots nodos y 4 Kilobots anclas y muestra como los Kilobots que están colocados de forma más céntrica en el enjambre obtienen mejores resultados en el cálculo de las coordenadas.

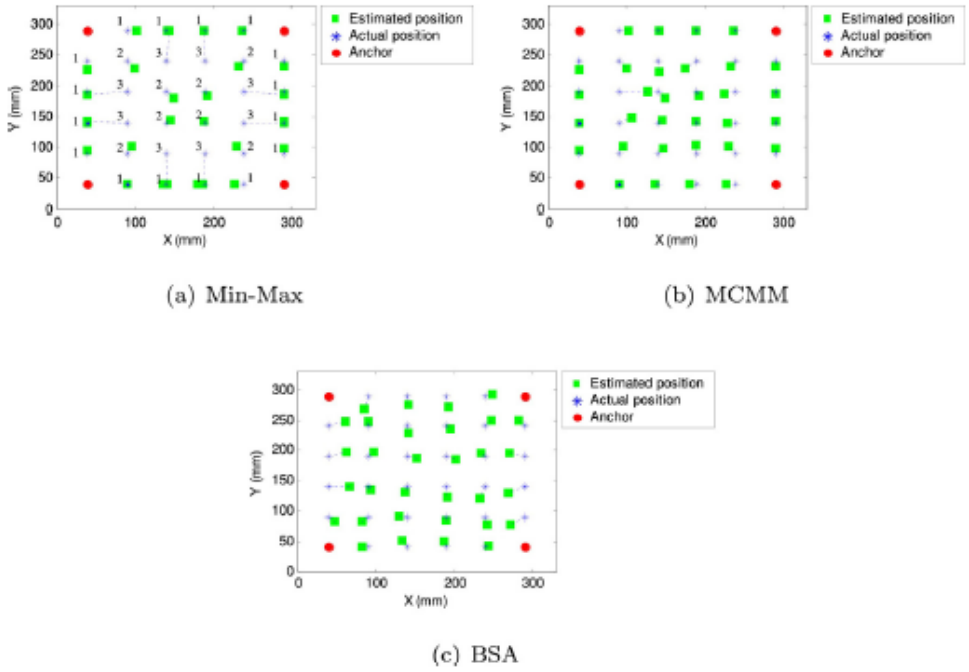


Figura 5.3. Resultados del ensayo realizado con 32 Kilobots nodo y 4 anclas
Fuente: **[24]**

5.1.2. Ensayo con método de réplica de órdenes

Con este método se han hecho tres ensayos, dos de estos ensayos con ruido. Este ruido simula las diferencias estructurales del Kilobot real que hacen que los movimientos se realicen a diferente velocidad entre Kilobots. Para simular este ruido se le introduce a la función inicialización del Kilobot el valor **`True`**, con esto se le suma a la velocidad de giro un valor aleatorio (valor aleatorio distinto para cada sentido de giro).

- **Ensayo 1:** En este ensayo se comprueba que el método funciona correctamente y sin tener cuenta el ruido. Se hace con la disposición mostrada en la figura 5.1. en la que tiene que guiar a todo el enjambre hacia un foco de luz. Los resultados obtenidos son ideales y solo sirven para comprobar que el código es correcto. También se puede apreciar en la simulación de este ensayo el retardo que hay entre el código y la simulación de CoppeliaSim ya que se puede ver como los Kilobots cambian de movimiento con una pequeña descoordinación de tiempo debido al retraso en la ejecución del código.

- **Ensayo 2:** A diferencia del ensayo anterior se realiza con el mismo número de Kilobots y disposición, pero con la diferencia de incluirle el ruido a los Kilobots. El proceso de simulación se muestra en la figura 5.4:

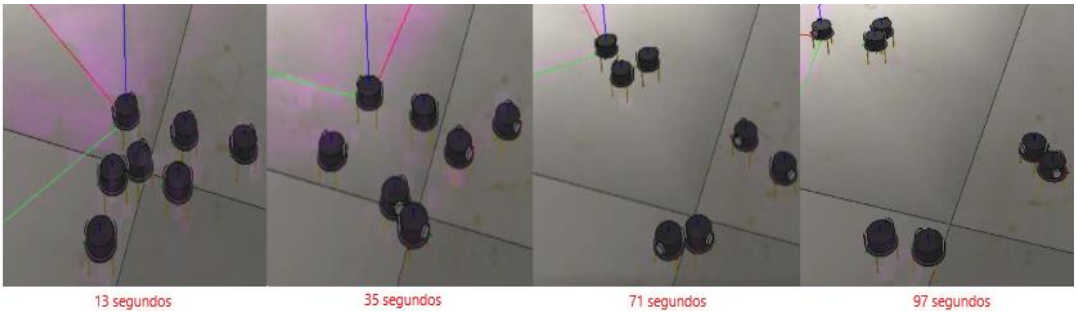


Figura 5.4. Simulación ensayo 2 método réplica de órdenes

Como se puede ver en la figura al no haber realimentación los Kilobots más lejanos al Líder se desorientan rápidamente y solo los más cercanos al líder lo acompañan hasta llegar al foco de luz, pero llegan de forma descoordinada debido al ruido.

- **Ensayo 3:** Este ensayo se realiza con una disposición diferente y con cinco Kilobots, la disposición es en fila y los resultados obtenidos se muestran en la siguiente figura:

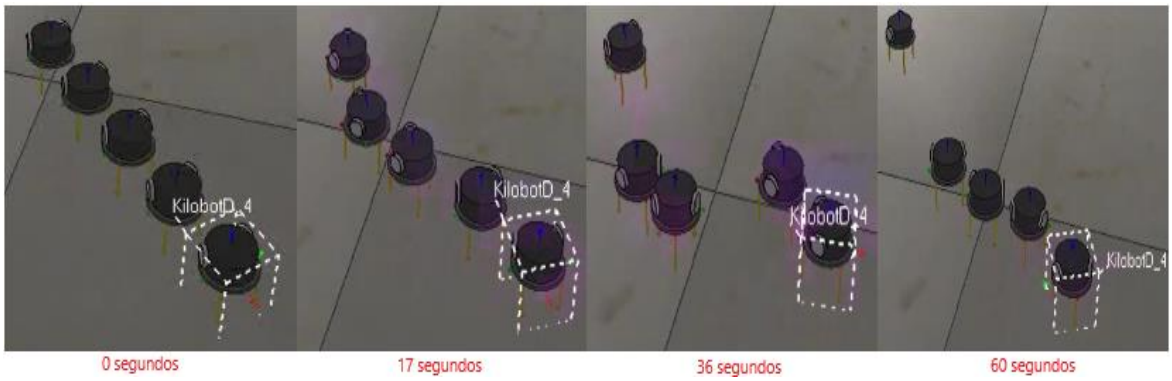


Figura 5.5. Simulación ensayo 3 método réplica de órdenes

La figura 5.5 muestra como debido al ruido el segundo Kilobot pierde los mensajes recibidos del Kilobot líder por alejarse este del rango de comunicación y al ser el padre de los siguientes Kilobots, el enjambre queda rápidamente desorientado y sin ninguna orden nueva para poder acercarse al foco de luz, dando vueltas sobre sí mismo en el sentido de la última orden reconocida.

5.1.3. Ensayos con método con realimentación

Con este método se han realizado dos ensayos con ruido, pero debido a la complejidad del código se tiene que realizar con 2 Kilobots seguidores y la velocidad de simulación tiene que ser lenta ya que debido al retraso de la ejecución del código cuanto más extenso y complejo es el código más retraso hay en procesar las órdenes los Kilobots. Si se le añade muchos Kilobots la carga computacional es demasiada para el ordenador y no funciona el código correctamente por este motivo se realiza con únicamente 2 Kilobots seguidores. Las simulaciones han sido mucho más largas no solo por la velocidad de simulación sino también porque este método al ser realimentado es más lento al tener que esperar que los Kilobots rezagados se acerquen al Kilobot líder.

- **Ensayo 1**: En este ensayo se ha dispuesto los Kilobots para que los dos Kilobots sean hijos del Kilobot líder y que los dos los tengan asignado como padre. En la figura 5.6 se puede ver como el **Kilobot líder** se acerca cada vez más a la luz conjunto a dos Kilobots seguidores. Este proceso es mucho más lento debido a la parte de realimentación ya que cuando un Kilobot se aleja del rango límite establecido, el **Kilobot líder** ordena a todo el enjambre detenerse excepto al Kilobot que estaba rezagado, que le ordena que se acerque. Este método también es más lento porque la realimentación se hace en base a la distancia y en ocasiones los Kilobots seguidores debido a su estructura hacen que los Kilobots avancen en dirección contraria al **Kilobot líder**, aunque finalmente cuando llegan al límite de comunicación terminan dándose la vuelta. Para arreglar este problema se incrementó el tiempo de muestra para leer la distancia a la que se encuentra el Kilobot padre y aunque mejora los resultados algunas veces ocurría este mismo proceso, pero de forma más breve y menos constante.

Figura 5.6. Simulación ensayo 1 método realimentado

- **Ensayo 2**: Este ensayo ha sido realizado para comprobar que funciona el método de realimentación padre/hijo. El resultado de la simulación es el de la figura 5.7:

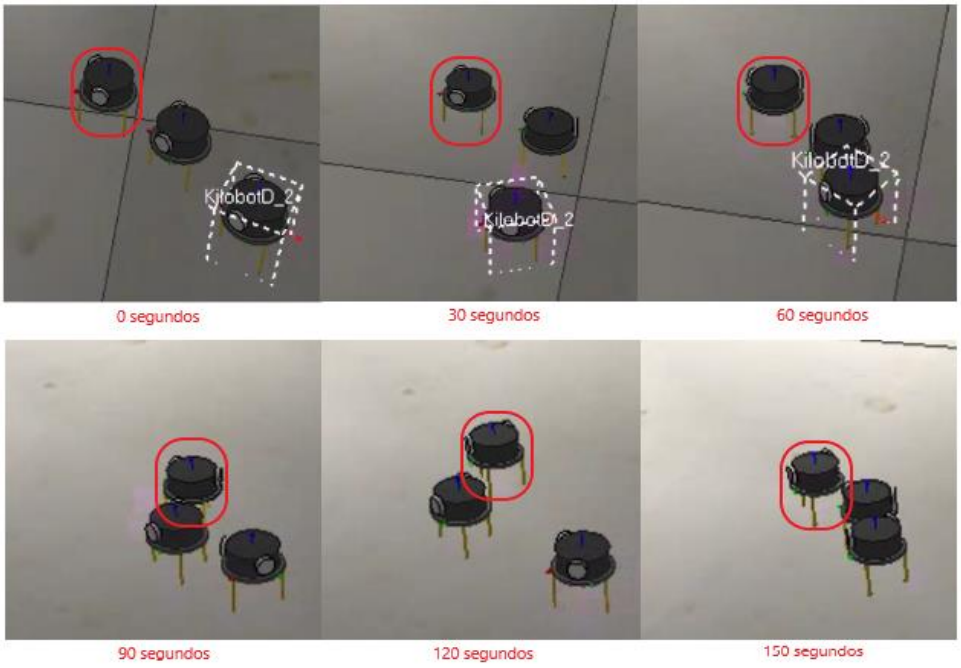


Figura 5.7. Simulación ensayo 2 método realimentado

En este ensayo el Kilobot 2 tiene como Kilobot padre un Kilobot hijo del **Kilobot líder,** actuando de puente para que le llegue la realimentación del Kilobot 2 a través de este Kilobot que es el que se encuentra entre los dos en la figura. Con esta estructura el enjambre ha tardado menos en ir hacia la luz, esto se debe a que tiene más espacio y no se chocan los Kilobots hijos entre ellos al contrario que en el primer ensayo, que uno de los problemas es que en ocasiones se chocaban los Kilobots hijos entre ellos dificultando el avance hacia el **Kilobot líder** haciendo más lento el proceso.

5.2. Resultados de los ensayos reales

Para la simulación del algoritmo "**distributed and resilient localization**" se han hecho varios ensayos de la etapa I y II variando la posición de un Kilobot (el conectado por el cable serie). En el caso de los Kilobots que van hacia un foco de luz solo se ha podido hacer ensayos con el **método de réplica de movimientos** ya que el método realimentado no se podía implementar debido a las limitaciones de memoria del Kilobot.

5.2.1. Ensayos Etapa I y Etapa II

Para probar estas dos etapas se han hecho tres ensayos:

- **Ensayo 1**: En este ensayo se han colocado los Kilobots con la distribución de la figura 5.8:

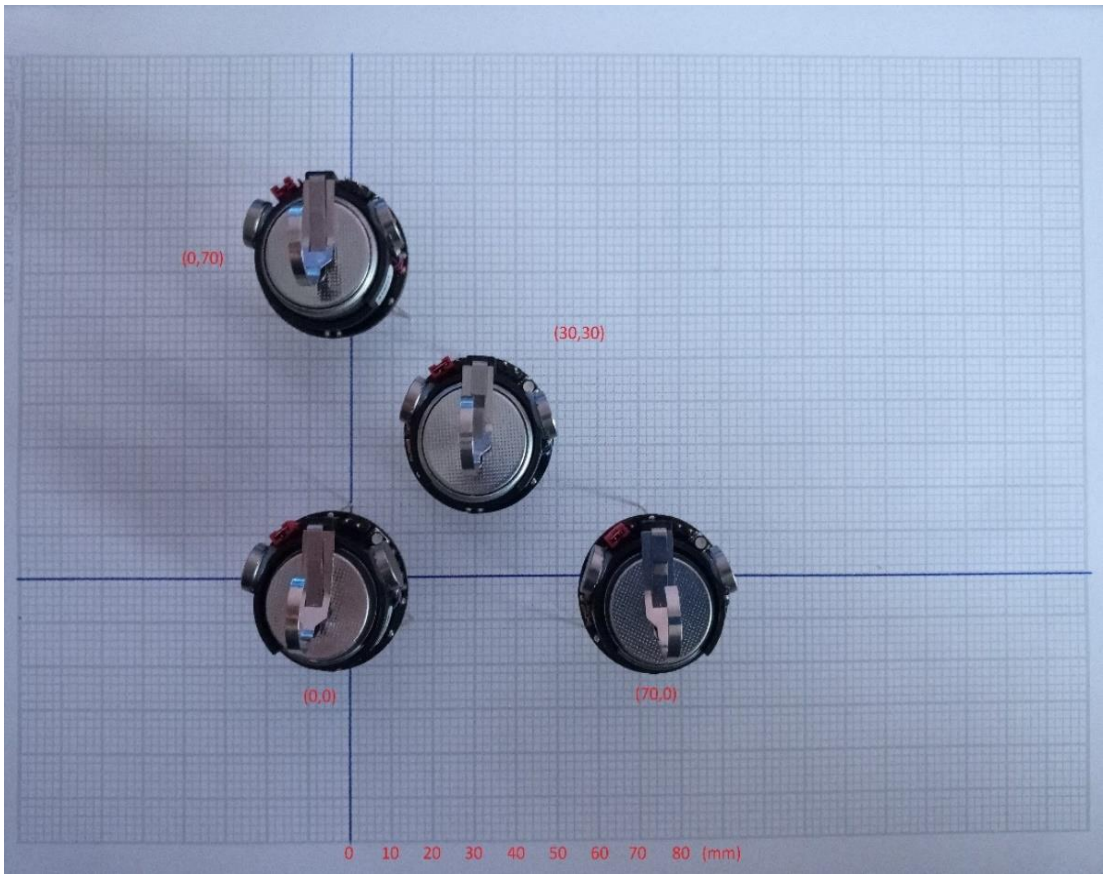


Figura 5.8. Ensayo 1 Etapa I y II caso real

Los Kilobots definidos como anclas son los que se encuentran en la posición (0,0), (0,70) y (70,0) y el Kilobot nodo que calcula su propia posición es el que se encuentra en la posición (30,30). Observando los resultados obtenidos a través del cable serie (figura 5.9) se ve que el error obtenido es de (3,3) milímetros en las coordenadas, un resultado muy aproximado al de la simulación para la posición (30,30) en el Etapa II (tabla 5.2) que tienen un error de 1 milímetro en cada coordenada. Las diferencias entre la simulación y el caso real se deben a la diferencia entre la localización de los Kilobots anclas, por la disposición de los Kilobots en el caso real que al ser ajustados manualmente puede que la posición no sea exactamente la mostrada y por el cálculo de la distancia a la que se encuentra el remitente de un mensaje que no es del todo exacta. Aunque tenga estas deficiencias el cálculo es bastante satisfactorio.

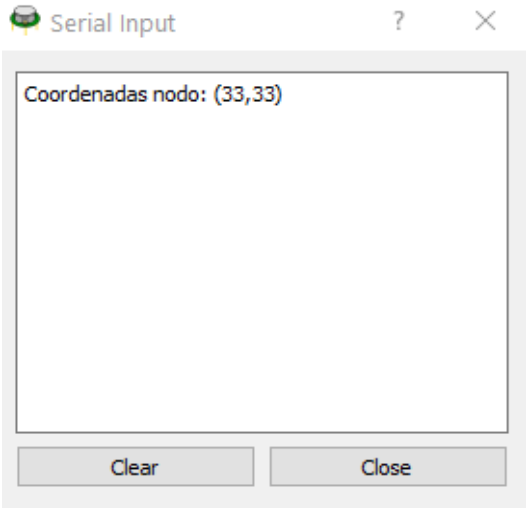


Figura 5.9. Coordenadas obtenidas por cable serie Ensayo 1

- **Ensayo 2**: En este ensayo se ha variado la posición del Kilobot Nodo para probar el método cerca de los límites de la región creada por los Kilobots anclas. La disposición es la de la figura 5.10:

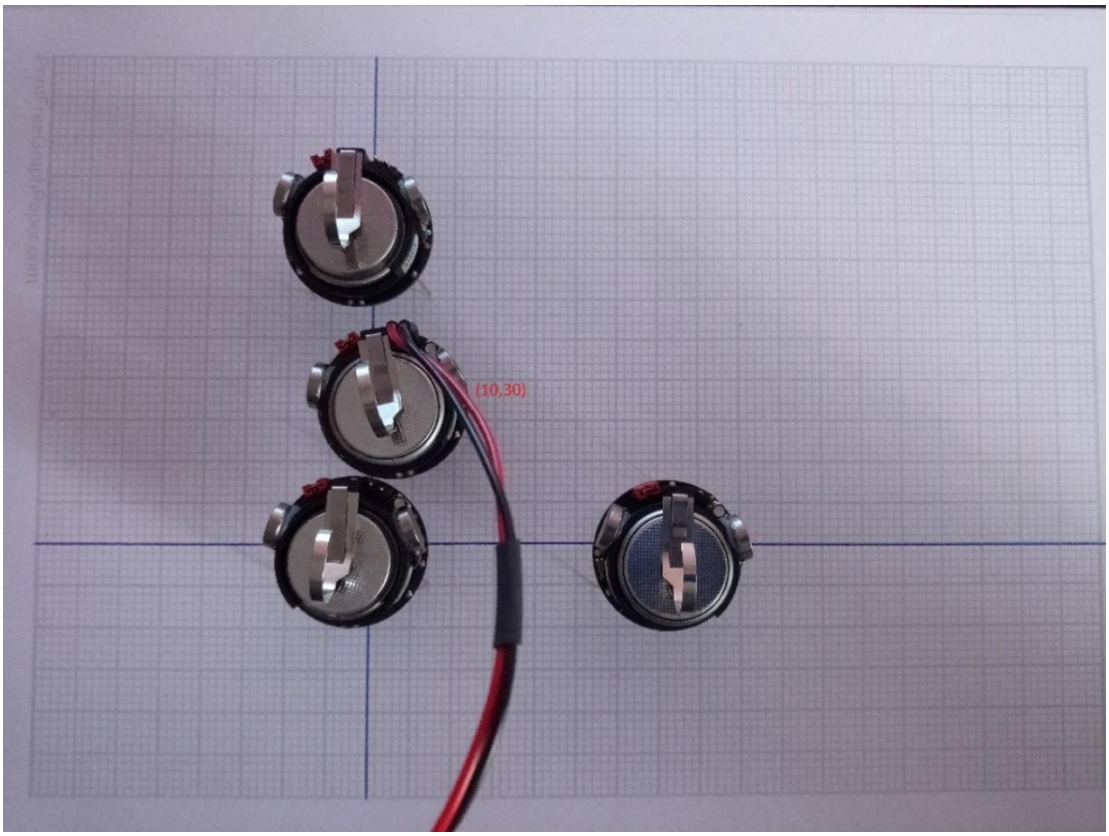


Figura 5.10. Ensayo 2 Etapa I y II caso real

Los resultados obtenidos en este ensayo han tenido mayor error como se muestra en la figura 5.11 en la que se ha obtenido unas coordenadas estimadas de (25,35) habiendo mucho más error que en el ensayo anterior, esto demuestra que al igual que en la simulación se comete más error cerca de los límites de la región creada por los Kilobots anclas. Además de esto se tiene que añadir a los motivos de error los explicados anteriormente, colocar de forma manual el Kilobot y el cálculo de la distancia del remitente que es una aproximación y no es del todo exacta.

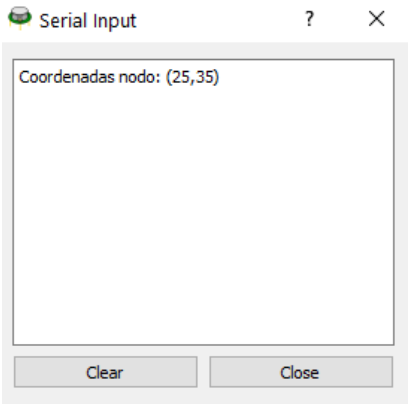


Figura 5.11. Coordenadas obtenidas por cable serial Ensayo 2

- **Ensayo 3**: Al igual que en el ensayo 2 se ha hecho cerca de los límites, pero cerca del eje x del enjambre. La distribución de los Kilobots es la mostrada en la figura 5.12.

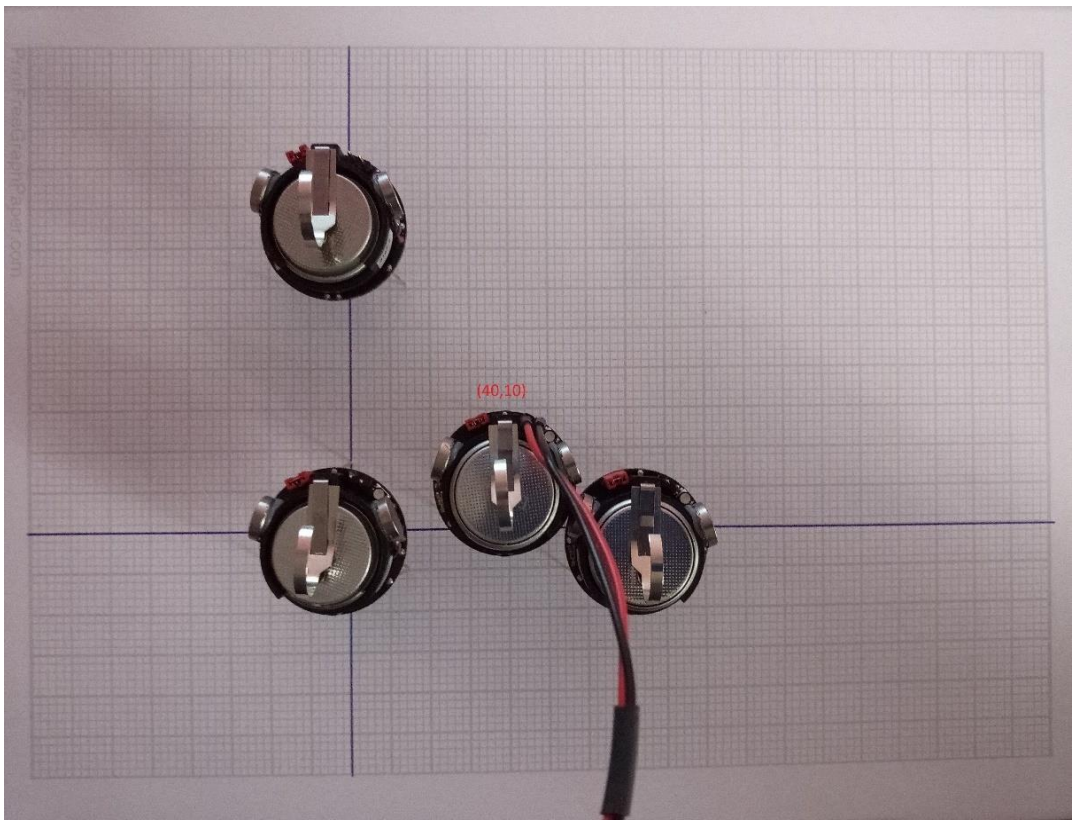


Figura 5.12. Ensayo 3 Etapa I y II caso real

Los resultados obtenidos (figura 5.13.) son parecidos al caso anterior en los que también se comete un error considerable.

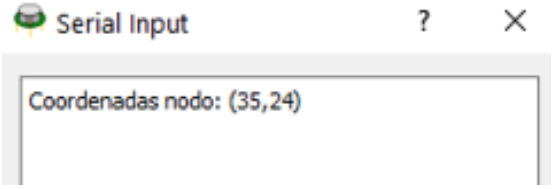


Figura 5.13. Coordenadas obtenidas por cable serial Ensayo 3

A continuación, se muestra una tabla resumen (tabla 5.7) de los resultados obtenidos:

Ensayo	Coordenadas reales (mm)	Etapa II (mm)	Error (mm)	Simulación Coordenadas (mm)	Simulación Estimación (mm)	Simulación Error (mm)
1	(30,30)	(33,33)	(3,3)	(30,30)	(31,31)	(1,1)
2	(10,30)	(25,35)	(15,5)	(10,35)	(18,35)	(8,0)
3	(40,10)	(35,24)	(5,14)	(35,10)	(35,18)	(0,8)

Tabla 5.7. Resultados obtenidos

Como es de esperar en la simulación los resultados obtenidos tienen menos error que en el caso real, pero estos resultados son bastante satisfactorios considerando las varias fuentes de ruido ya comentadas.

5.2.2. Ensayos Kilobots con método de réplica de movimientos

Para la simulación de los Kilobots que se dirigen hacia un foco de luz se realiza a través del método de réplica de movimientos realizando con este, 4 ensayos:

- **Ensayo 1**: En este ensayo es una primera toma de contacto con el método, por lo que se realiza con un Kilobot líder y un Kilobot seguidor. Los resultados se muestran en la figura 5.14:

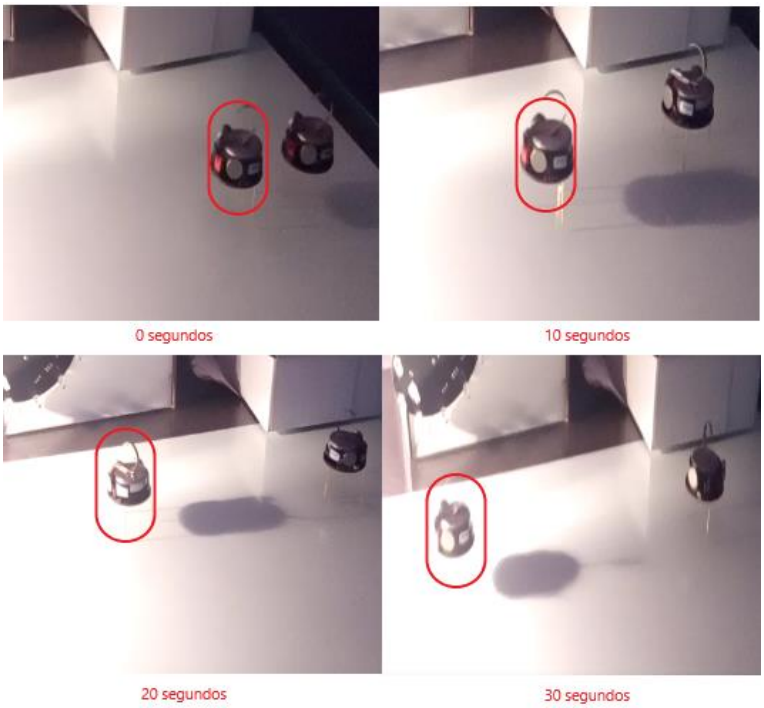


Figura 5.14. Resultados método réplica de movimientos ensayo 1

  En este ensayo debido a las diferencias en el movimiento entre el **Kilobot líder** y el seguidor, se ha perdido la conexión entre estos dos y el Kilobot seguidor se ha quedado rezagado sin poder recibir nuevas órdenes del **Kilobot líder**, aunque como prueba se ha desarrollado correctamente ya que al principio de la simulación replicaba los movimientos.

- **Ensayo 2**: En este ensayo se ha aumentado el número de Kilobots seguidores a dos y se ha intentado ajustar la calibración del movimiento para que sean los más parecidos posibles al **Kilobot líder**, los resultados son los de la figura 5.15:

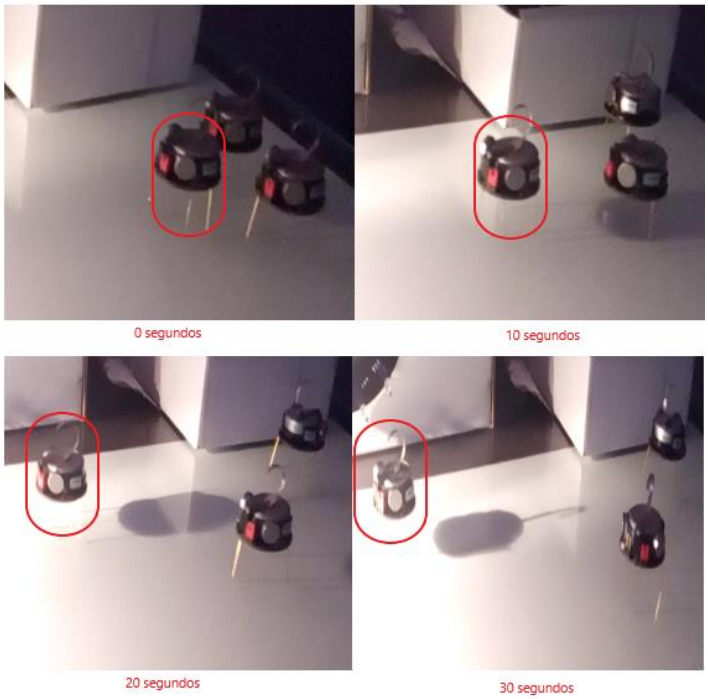


Figura 5.15. Resultados método réplica de movimientos ensayo 2

En este se han conseguido mejores resultados, ya que al principio si responde bien uno de los Kilobots seguidores, pero llega un punto en el que se acaba desorientado por los mismos motivos que en el **ensayo 1**.

- **Ensayo 3:** En este ensayo se han añadido dos Kilobots más seguidores para comprobar si algún Kilobot pude seguir al **Kilobot líder** obteniendo el resultado de la figura 5.16:

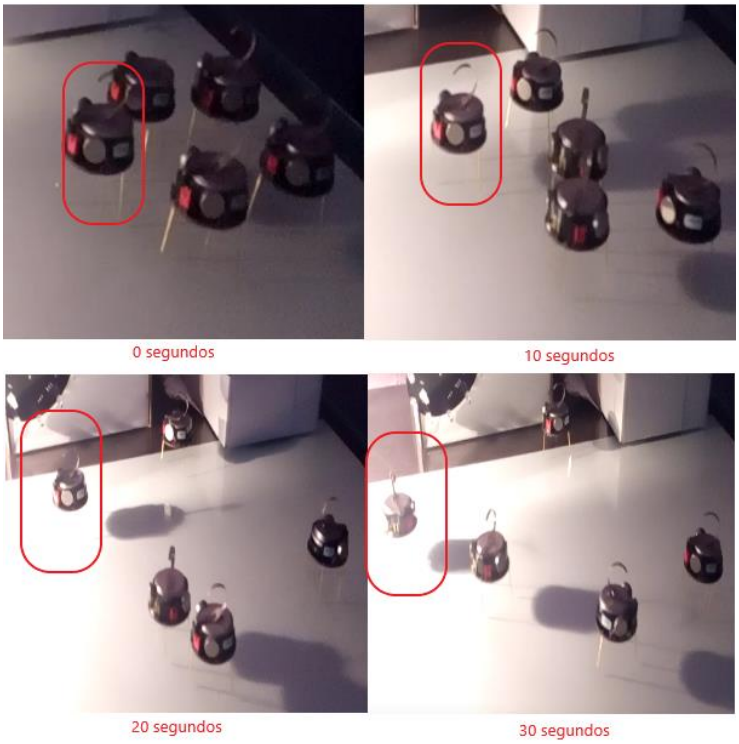


Figura 5.16. Resultados método réplica de movimientos ensayo 3

En este ensayo uno de los Kilobots seguidores si alcanza el foco de luz, pero con muchas dificultades debido a las mismas causas que en ensayos anteriores. En estos tres ensayos ocurre aproximadamente lo mismo que en la simulación del método de réplica de movimientos con ruido. La mayoría de los Kilobots se desorientan por las diferencias en el movimiento y se acaba perdiendo la sincronización del enjambre provocando que se separen del **Kilobot líder** y no poder recibir órdenes del éste.

- **Ensayo 4:** En este ensayo se realiza para comprobar el funcionamiento del algoritmo "Wave", para asignar el Kilobot padre. Se colocan los Kilobots en cadena y se inicia el método dando como resultado los mostrados en la figura 5.17:

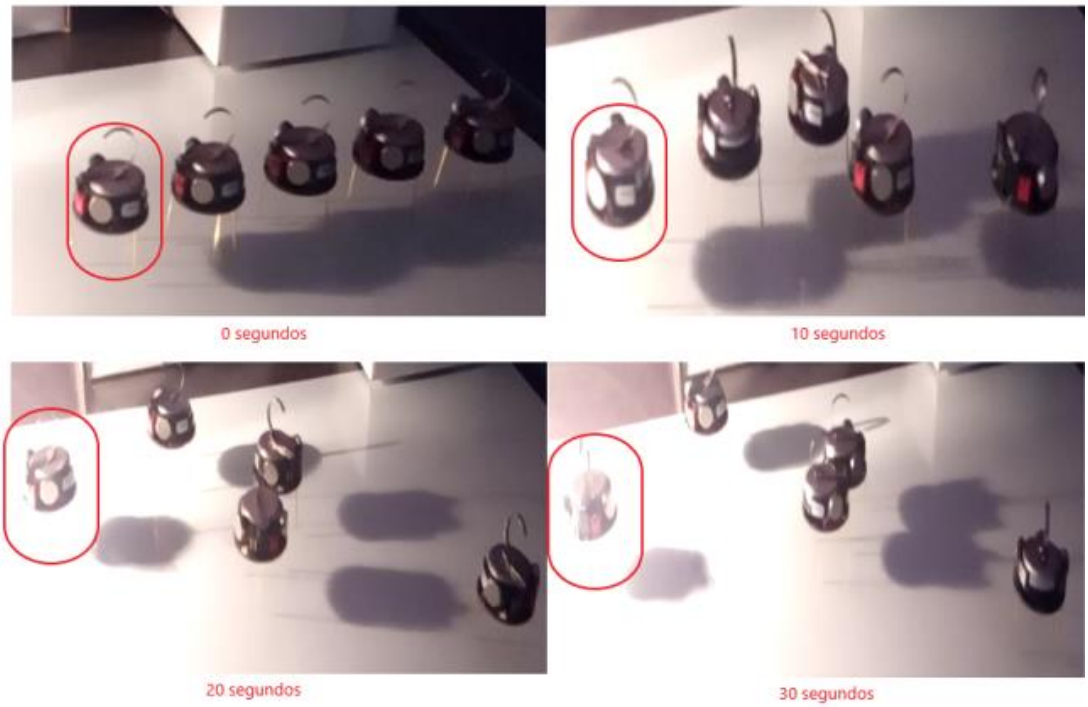


Figura 5.17. Resultados método réplica de movimientos ensayo 4

Los resultados obtenidos son los mismos que en simulación, los últimos Kilobots siguen las órdenes del **Kilobot líder** a través de Kilobots padre/hijo, pero debido al ruido acaban desorientándose. El **método realimentado** basado en las distancias no se ha podido implementar debido a las limitaciones de espacio en memoria del Kilobot por lo que no se ha podido realizar ningún ensayo para probar este método.

Las conclusiones se centran en la evaluación del algoritmo "**distributed and resilient localization**", el algoritmo "**Wave**" para la comunicación entre robots, el uso de **Python** como lenguaje de programación, **CoppeliaSim** y los **Kilobots** además de los futuros trabajos que se puedan realizar con cada una de estas herramientas usadas en el TFG.

6.1. Conclusiones

El algoritmo "**distributed and resilient localization**" puede tener un desarollo interesante en el mundo de la robótica móvil ya que se puede implementar fácilmente en cualquier robot. El problema de este algoritmo es que necesita de puntos de referencia para que el robot pueda localizarse por sí mismo, por lo que depende de otros robots ancla que sepan su posición o se hace uso de unas balizas que actúen como anclas. Otro de los problemas de este algoritmo es que no proporciona una localización totalmente exacta, por lo que se puede implementar en sistemas en el que la posición no tiene que ser del todo exacta o se puede implementar como apoyo a otro algoritmo de localización para verificar un buen cálculo de la posición, empleando técnicas de fusión sensorial.

Con el algoritmo "**Wave**" se comprueba que la comunicación en un enjambre se puede realizar fácilmente con la asignación padre/hijo a los robots, configurando un grupo ordenado que colabora para realizar una tarea dirigida por un robot líder. Un problema de este algoritmo es que solo el robot líder es el que evalúa y toma órdenes para el enjambre en base a la situación en la que se encuentra el enjambre, por lo que la carga computacional que soporta este robot es elevada en comparación con los robots seguidores. Además, estos robots seguidores no pueden trabajar de forma autónoma debido a que solo pueden seguir órdenes del robot líder.

Python al ser un lenguaje de alto nivel ha simplificado la programación de los Kilobots en CoppeliaSim**.** Con ayuda de Anaconda que proporciona un entorno para la gestión de librerías fácil e intuitivo y la IDE Jupyter, que con su sistema para distribuir el código por bloques permite ejecutar unas partes de código u otras dependiendo de la simulación que se necesita realizar, se ha podido realizar una programación más fácil. El uso de CoppeliaSim y Python para la simulación de enjambres da un resultado satisfactorio, pero se han producido pequeños retrasos entre la ejecución del código y la simulación. Esto también puede ser debido al ordenador utilizado (Intel Core I5, con tarjeta gráfica Intel® UHD Graphics 620). Los ensayos se han realizado con siete Kilobots. Si se le hubiese añadido más Kilobots a los ensayos en los que el Kilobot iba hacia la luz con el método con realimentación, habría tenido mucha carga computacional para el ordenador y no se habría conseguido una simulación correcta, por lo que para la simulación de grandes enjambres de robots con códigos medianamente complejos debe de usarse una herramienta de simulación más simple o un ordenador con altas prestaciones para no tener problemas.

Los **Kilobots** han resultado ser una herramienta útil para hacer unos primeros ensayos con algoritmos de tecnología de enjambre y aunque se tienen bastantes limitaciones debido a la memoria de código que puede almacenar, pueden usar para realizar unos primeros ensayos de prueba para algoritmos simples de tecnología de enjambre. Los principales algoritmos que se pueden ensayar en este tipo de robots son los relacionados con la comunicación y si algún algoritmo está relacionado con el desplazamiento es mejor si el Kilobot se mueve de forma autónoma sin depender de otro **Kilobot** en movimiento.

6.2. Futuros trabajos

Como futuros trabajos se podría probar el algoritmo "**distributed and resilient localization**" junto a otro algoritmo de localización y que a partir de este puedan reordenarse o hacer una forma en concreto. En cualquier algoritmo en el que se necesite intercambio de información mediante un enjambre

se puede realizar la gestión de la comunicación en el enjambre con el algoritmo "**Wave**" ya que como se ha visto es un método robusto para gestionar la comunicación en un enjambre. Ya que CoppeliaSim no tiene limitaciones, se puede implementar un algoritmo más complejo para un enjambre de Kilobots u otro tipo de robot con los que estén dispuestos para trabajar con tecnología de enjambre. Si se elige un nuevo tipo de robot con mejores capacidades sensoriales se puede realizar también a partir de una POO y Python una programación fácil que después se podrá extrapolar fácilmente al robot real.

[1] CORRELL, N., SCHWAGER, M., OTTE, M. "*Proceedings in Advanced Robotics: Distributed Autonomous Robotic Systems"*, 2019, num.9, ISBN 978-3-319-73006-6

[2] HAMANN, H. "*Swarm Robotics: A Formal Approach"*. Lübeck: Institute of Computer Engineering, 2018.

[3] BRAMBILLA, M., FERRANTE, E., BIRATTARI, M. *"Swarm robotics: a review from the swarm engineering perspective"*. *Swarm Intell*, 2013, num 7, p. 1–41.

[4] «CoppeliaSim» [En línea]. Disponible en: https://www.coppeliarobotics.com. [Último acceso: 15 marzo 2021].

[5] OTTE, M. "*Collective Cognition and Sensing in Robotic Swarms via an Emergent Group-Mind", International Symposium on Experimental Robotics*, 2016, Aerospace Engineering Sciences Department, University of Colorado at Boulder, Boulder, USA.

[6] YONG, W., XIAOLEI, M., ZHIBIN, L., YONG, L., MAOZENG, X., YINHAI, W. "*Profit distribution in collaborative multiple centers vehicle routing problema"*, *Journal of Cleaner Production*, 2017, num. 144, p. 203-219, ISSN 0959-6526.

[7] AHMAD, D., MEH, J., KASHIF, Z., ABBAS, K., DINESH K. S. "*Behavior-based swarm robotic search and rescue using fuzzy controller*", *Computers & Electrical Engineering*, 2018, num. 70, p. 53-65, ISSN 0045-7906.

[8] ALMEIDA, J.P.L.S., NAKASHIMA, R.T., NEVES-JR, F." Autonomous Navigation of Multiple Robots with Sensing and Communication Constraints Based on Mixed Reality" J Control Autom Electr Syst, 2020, num. 31, p. 1165–1176.

[9] KOZHEMYAKIN, I., SEMENOV, N.,RYZHOV, V.,CHE-MODANOV, M. "Multi-agent Control System of a Robot Group: Virtual and Physical Experiments", Interactive Collaborative Robotics, Lecture Notes in Computer Science, 2019, num. 11659.

[10] «Kilobot», K-Team [En línea]. Disponible en: https://www.k-team.com/mobile-robotics-products/kilobot. [Último acceso: 15 marzo 2021].

[11] «Khepera IV », K-Team [En línea]. Disponible en: https://www.k-team.com/khepera-iv . [Último acceso: 15 marzo 2021].

[12] CAPRARI, G., BALMER, P., PIGUET, R., SIEGWART, R. "*The autonomous microbot 'Alice': A platform for scientific and commercial applications", Proceedings of the Ninth International Symposium on Micromechatronics and Human Science*, 1998, Nagoya, Japan, p. 231–235.

[13] SILVA JR, L., NEDJAH, N. "*Wave Algorithm for Recruitment in Swarm Robotics"*, Springer International Publishing, 2015, Switzerland.

[14] SILVA JR, L., NEDJAH, N. "*Wave algorithm applied to collective navigation of robotic swarms" Applied Soft Computing*, 2017, num. 57, p. 698-707, ISSN 1568-4946.

[15] BULLA CRUZ, N., NEDJAH, N., MOURELLE, L. "*Robust distributed spatial clustering for swarm robotic based systems" Applied Soft Computing*, 2017, num. 57, p. 727-737, ISSN 1568-4946.

[16] OLIVEIRA DE SÁ, A., NEDJAH, N., MOURELLE, L. "*Distributed efficient localization in swarm robotic systems using swarm intelligence algorithms*", *Neurocomputing*, 2016, num. 172, p. 322-336, ISSN 0925-2312.

[17] ZELENKA, J., KASANICKY, T., BUDINSKÁ, I., NADO, L., KANUCH, P. "*SkyBat: A Swarm Robotic Model Inspired by Fission-Fusion Beha-viour of Bats" N. A. Aspragathos*, 2019, num. 67, p. 521–528.

[18] SONG, Y., FANG, X., LIU, B., LI, C., LI, Y., YANG, S. "*A novel foraging algorithm for swarm robotics based on virtual phe-romones and neural network" Applied Soft Computing*, 2020, num. 90, p. 106-156, ISSN 1568-4946.

[19] OH, H., SHIRAZ A. R., JIN, Y. "*Morphogen diffusion algorithms for tracking and herding using a swarm of kilobots" Soft Comput*, 2018, num. 22, p. 1833–1844.

[20] «About ARGos», ARGos [En línea]. Disponible en: https://www.argos-sim.info/concepts.php . [Último acceso: 15 marzo 2021].

[21] «Webots» [En línea]. Disponible en: https://cyberbotics.com/ . [Último acceso: 15 marzo 2021].

[22] «Robotics», Cornell University [En línea]. Disponible en: https://arxiv.org/abs/1511.04285 . [Último acceso: 15 marzo 2021].

[23] OLIVEIRA DE SÁ, A., NEDJAH, N., MACEDO MOURELLE, L. "*Distributed and resilient localization algorithm for Swarm Robotic Systems" Applied Soft Computing*, 2017, num. 57, p. 738-750, ISSN 1568-4946.

[24] CIVICIOLGLU, P. *"Backtracking Search Optimization Algorithm for numerical optimization problems" Applied Mathematics and Computation*, 2013, num. 219, p. 8121-8144, ISSN 0096-3003.

[25] «Manual de usuario», CoppeliaSim [En línea]. Disponible en: https://www.coppeliarobotics.com/helpFiles/index.html. [Último acceso: 25 marzo 2021].

[26] «Bo-based remoted API», CoppeliaSim [En línea]. Disponible en: https://www.coppeliarobotics.com/helpFiles/en/b0RemoteApiOverview.htm . [Último acceso: 25 marzo 2021].

[27] «Bo-based remoted API, Python», CoppeliaSim [En línea]. Disponible en: https://www.coppeliarobotics.com/helpFiles/en/b0RemoteApi-python.htm . [Último acceso: 25 marzo 2021].

[28] «Introducción Python», Programación en Python nivel básico [En línea]. Disponible en: https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion1/introduccion.html . [Último acceso: 29 marzo 2021].

[29] «Ventajas y desventajas de Python», Programación en Python nivel básico [En línea]. Disponible en: https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion1/ventajas_desventajas.html . [Último acceso: 29 marzo 2021].

[30] «Documentación Anaconda», Anaconda [En línea]. Disponible en: https://docs.anaconda.com/anaconda/ . [Último acceso: 29 marzo 2021].

[31] «Programación orientada a objetos en Python», Programación en Python nivel básico [En línea]. Disponible en: https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion9/poo.html . [Último acceso: 29 marzo 2021].

[32] «Biblioteca Random», Introducción a la programación con Python [En línea]. Disponible en: https://www.mclibre.org/consultar/python/index.html . [Último acceso: 29 marzo 2021].

[33] «Librería Numpy», AprendeIAI [En línea]. Disponible en: https://aprendeia.com/introduccion-a-numpy-python-1/ . [Último acceso: 29 marzo 2021].

[34] «Module: core.debugger », IPython Documentation [En línea]. Disponible en: https://ipython.readthedocs.io/en/stable/api/generated/IPython.core.debugger.html . [Último acceso: 29 marzo 2021].

[35] «Manual Kilobot», K-Team [En línea]. Disponible en: http://ftp.k-team.com/kilobot/user_manual/Kilobot_UserManual.pdf . [Último acceso: 29 marzo 2021].

[36] RUBENSTEIN, M., AHLER, C., HOFF, N., CABRERA, A., NAGPAL, R. "*Kilobot: A low cost robot with scalable operations designed for collective behaviors*", *Robotics and Autonomous Systems,* 2014, num.62, p.966-975, ISSN 0921-8890.

[37] «ATMega 328 datasheet», alldatasheet [En línea]. Disponible en: https://pdf1.alldatasheet.com/datasheet-pdf/view/422609/ATMEL/ATMEGA328.html . [Último acceso: 29 marzo 2021].

[38] «Downloads», Kilobotics [En línea]. Disponible en: https://kilobotics.com/download . [Último acceso: 29 marzo 2021].

[39] «Eclipse» [En línea]. Disponible en: https://www.eclipse.org/ . [Último acceso: 29 marzo 2021].

[40] «Manual de instalación» WinAVR [En línea]. Disponible en: http://winavr.sourceforge.net/install_config_WinAVR.pdf. [Último acceso: 2 abril 2021].

[41] «AVR Libc» [En línea]. Disponible en: http://www.nongnu.org/avr-libc/ . [Último acceso: 2 abril 2021].

[42] «ApiDocs», Kilobotics [En línea]. Disponible en: https://kilobotics.com/docs/index.html . [Último acceso: 29 marzo 2021].

[43] «Labs», Kilobotics [En línea]. Disponible en: https://kilobotics.com/labs . [Último acceso: 12 abril 2021]

Anaconda

Para gestionar las librerías Anaconda hace uso de entornos. Instalado **Python** y las librerías necesarias para el desarrollo del trabajo que se desea hacer, se puden tener varios entornos con diferentes librerías seleccionadas dependiendo de la finalidad del trabajo a realizar.

La creación del entorno se puede hacer a partir de la **ventana de comandos de Windows** o con la **UI Anaconda Navigator,** aquí se explica a través de la UI:

1- Abrir la **UI Anaconda Navigator** y seleccionar la pestaña de entornos (**Environment**), la figura A.1 muestra una imagen de ejemplo:

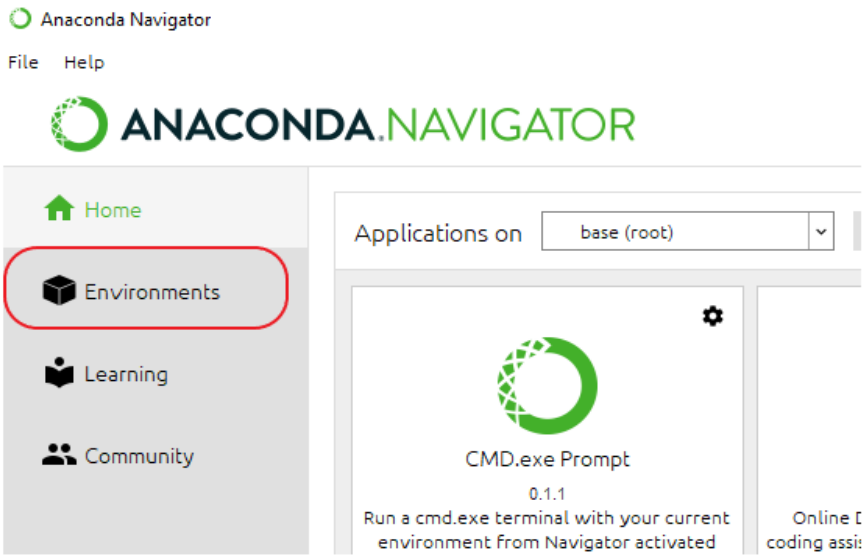


Figura A.1. Menú de selección de Anaconda

2- Esta pestaña muestra todos los entornos instalados y las diferentes librerías que contienen, para crear una se selecciona el botón de crear un nuevo entorno (**Create**), esto se muestra en la figura A.2:

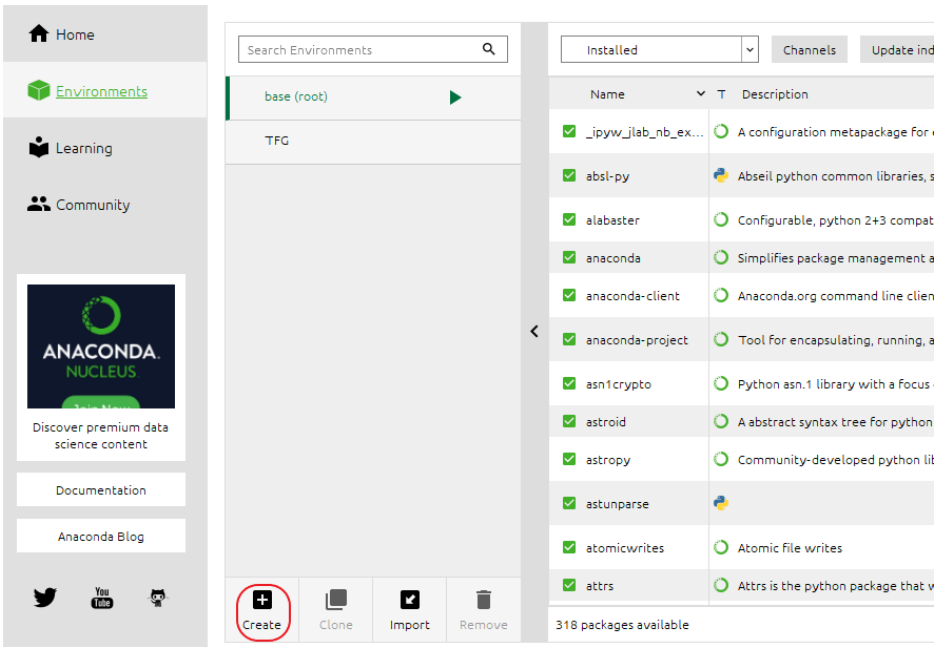
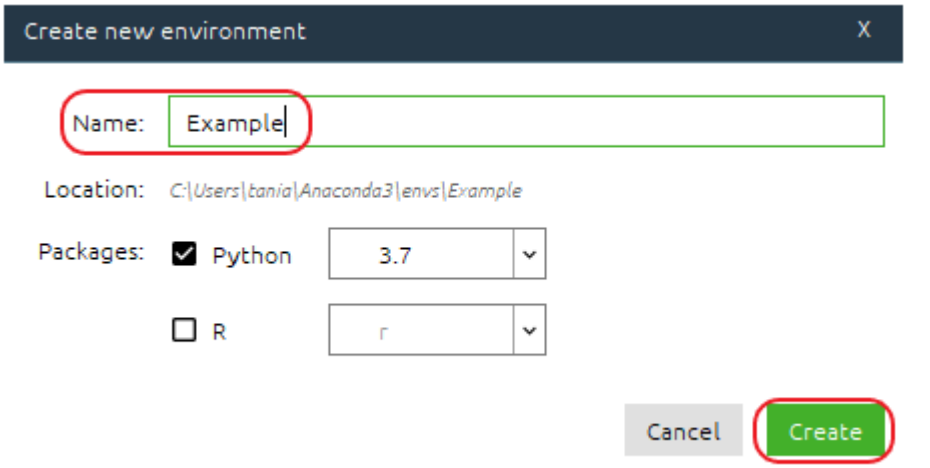


Figura A.2. Creación entorno Anaconda

3- Aparece una ventana como la de la figura A.3 se escribe el nombre del entorno y se le da al botón créate:



7

Figura A.3. Crear un nuevo entorno en Anaconda

4- Después de crearse el entorno se vuelve a la pestaña principal (**Home**) y se instala el entorno **Jupyter** haciendo clic en el botón **Intall** como se muestra en la figura A.4:

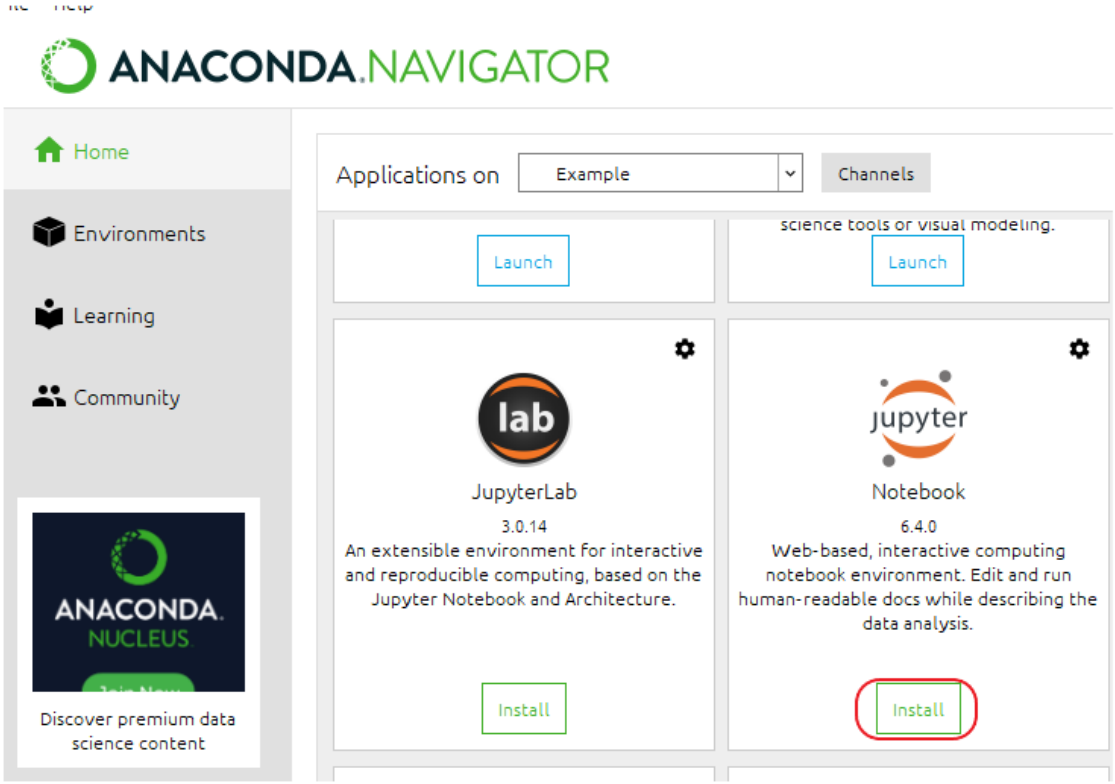


Figura A.4. Instalar Jupyter en Anaconda

5- Una vez instalado **Jupyter** ya se puede empezar a programar con él. Pero para el desarrollo de este TFG se deben instalar dos librerías que se utilizaran. Para su instalación se hace a través de una ventana terminal de **Windows**, que se puede abrir una con el buscador de Windows o se puede hacer a partir de la UI seleccionando **CMD.exe Prompt** que abre una ventana terminal con el directorio el entorno seleccionado como muestra la figura A.5:

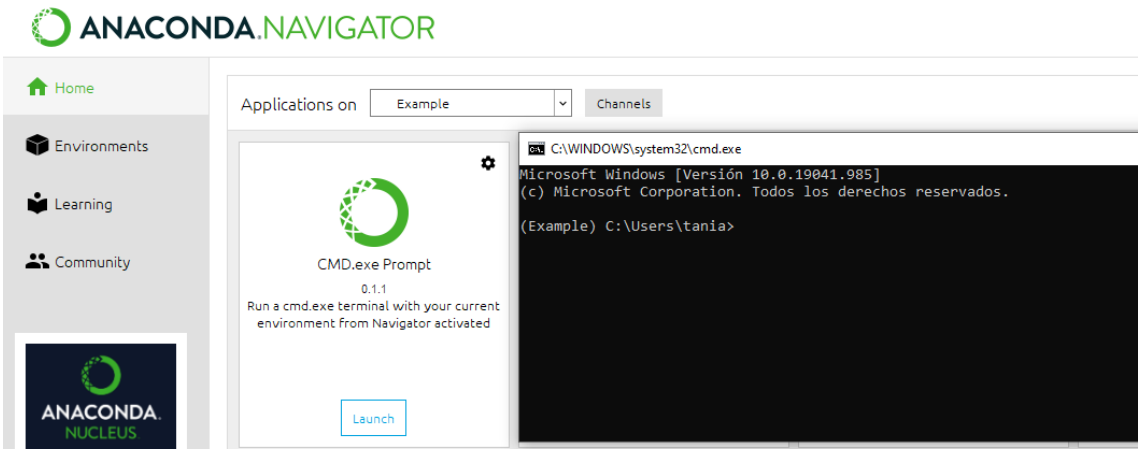


Figura A.5. Ventana terminal para instalar librerías

6- Para instalar las librerías necesarias basta con escribir el comando **pip install *Nombre librería*,** para poder utilizar **Jupyter** junto a **CoppeliaSim** se instala la librería **msgpack.** También se utiliza la librería **numpy** por lo que se debe instalar con este mismo método como se muestra en la figura A.6:

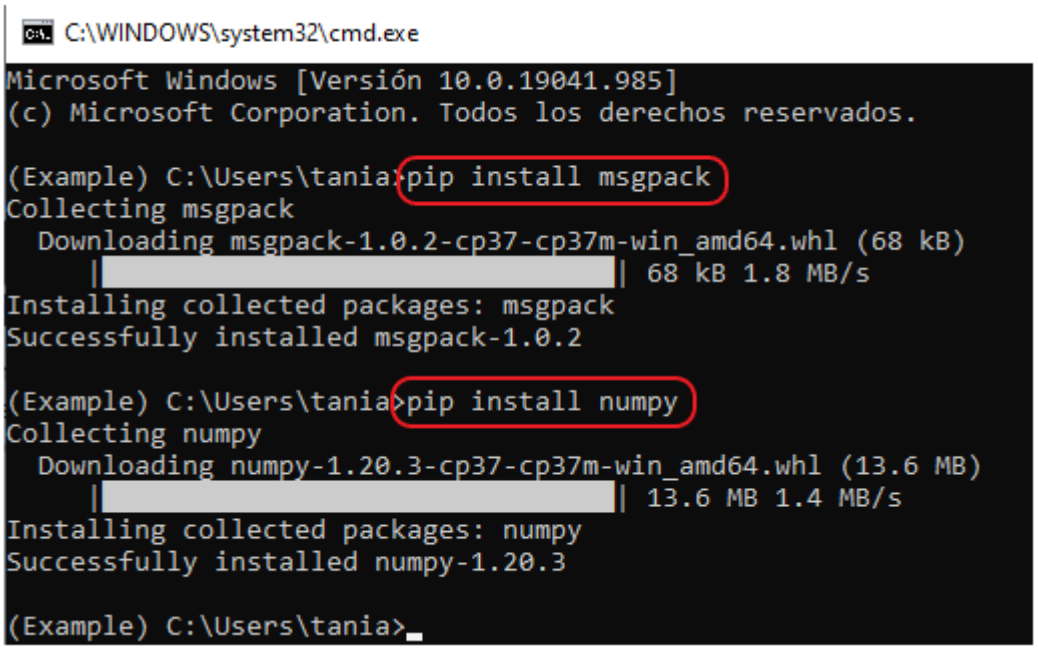


Figura A.6. Resultado ventana terminal después de instalar las librerías msgpack y numpy

7- Para finalizar para poder usar la librería B0-Based Remote API, en la carpeta donde se generan los códigos se tienen que añadir los siguientes archivos:

- b0.py
- b0RemoteApi.py
- b0.dll
- boost_date_time-vc141-mt-x64-1_70.dll
- boost_filesystem-vc141-mt-x64-1_70.dll

- boost_program_options-vc141-mt-x64-1_70.dll
- boost_regex-vc141-mt-x64-1_70.dll
- boost_thread-vc141-mt-x64-1_70.dll
- libzmq-mt-4_3_2.dll
- lz4.dll
- zlib1.dll

Todos estos archivos se encuentran en el directorio donde este instalado CoppeliaSim, que normalmente es **C:\Program Files\CoppeliaRobotics\CoppeliaSimEdu** y en la ruta **C:\Program Files\CoppeliaRobotics\CoppeliaSimEdu\programming\b0RemoteApiBindings\python**

Eclipse

Los códigos que se cargaban a los Kilobots a través de la aplicación **KiloGUI** son códigos con la extensión **.hex**. Esta extensión es un archivo con código hexadecimal que contiene todas las órdenes que tiene que hacer el Kilobot. Para generar este archivo se debe instalar **WinAVR** y preparar el entorno de programación **Eclipse:**

1- Primeramente, se descarga e instala **Eclipse** descargándose desde la página https://www.eclipse.org/downloads/ ,la instalación es un proceso fácil que no necesita mucho detalle ya que tan solo es seguir los pasos del instalador.

2- Se instala **WinAVR** accediendo a la página web https://sourceforge.net/projects/winavr/files/latest/download se descarga el instalador. Una vez se haya descargado el instalador, se instala de forma sencilla con tan solo seguir los pasos del instalador.

3- Además de descargarse estas dos aplicaciones, se necesita descargarse las librerías que se utilizaran en el Kilobot estas librerías se encuentran en el enlace https://github.com/acornejo/kilolib ,además de las librerías necesarias, también contienen el archivo "**Makefile**" que permite que se generen los archivos **.hex**. Una vez se descargue el archivo se crea la carpeta donde está el entorno de programación y se extrae el archivo rar descargado en esta carpeta.

4- Una vez ya este todo instalado y descargado se inicia Eclipse y se crea el entorno de programación como muestra la figura A.7 en la carpeta donde se descargaron las librerías:

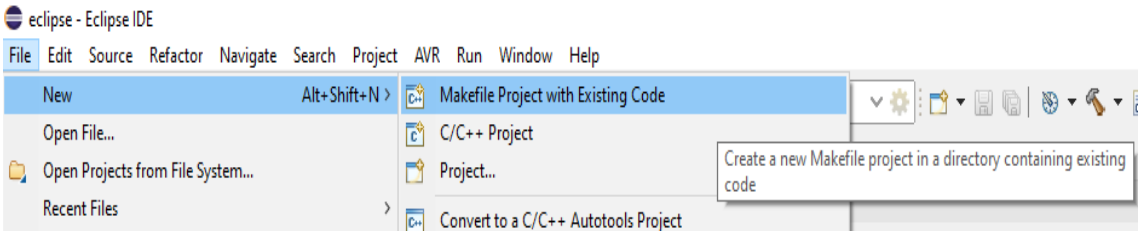


Figura A.7. Crear un nuevo entrono de programación en Eclipse

Se pone un nombre al proyecto, se busca la ruta donde están las librerías junto al archivo "**Makefile**" y se selecciona la herramienta para compilar **AVR-GCC Toolchain** como se muestra en la figura A.8:

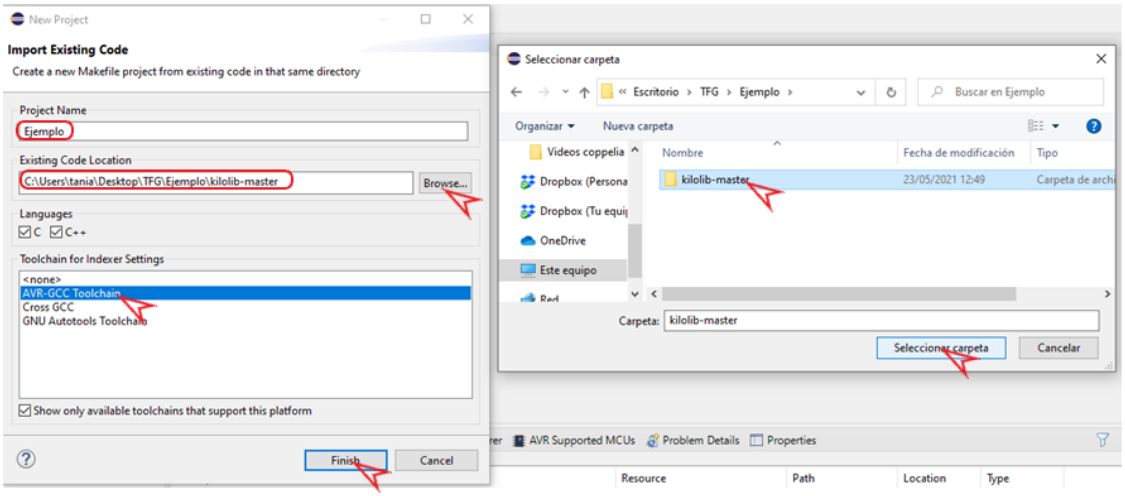


Figura A.8. Selección de las librerías utilizas para la programación en Eclipse

5- Una vez se crea el entorno de programación ya se puede crear el código de programación usando el archivo "**blank.c**", un ejemplo del archivo que se genera cuando se compila se muestra en la figura A.9:

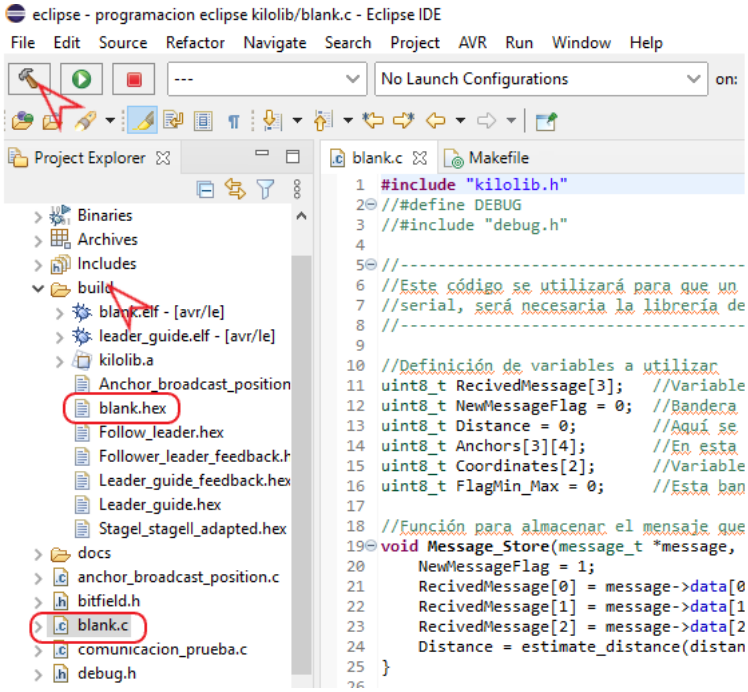


Figura A.9. Ilustración del árbol de archivos con el archivo blank

Todos los códigos que se desean realizar se escriben en este archivo (**blank.c**) y una vez terminado se compila con el botón de compilar y el archivo **(blank.hex**) es el que se utilizara en la aplicación **KiloGUI** para transmitir al Kilobot. Si se desea conservar el archivo y generar distintos códigos tan solo se tiene que cambiar el nombre del archivo "**Blank.hex**"(**importante, se cambia el nombre del archivo con extensión .hex no el de extensión .c**) para generar un nuevo código con distintas órdenes.

A la hora de la compilación de los archivos se muestra el error mostrado en la figura A.10 que impide que se genere el archivo **blank.hex**:

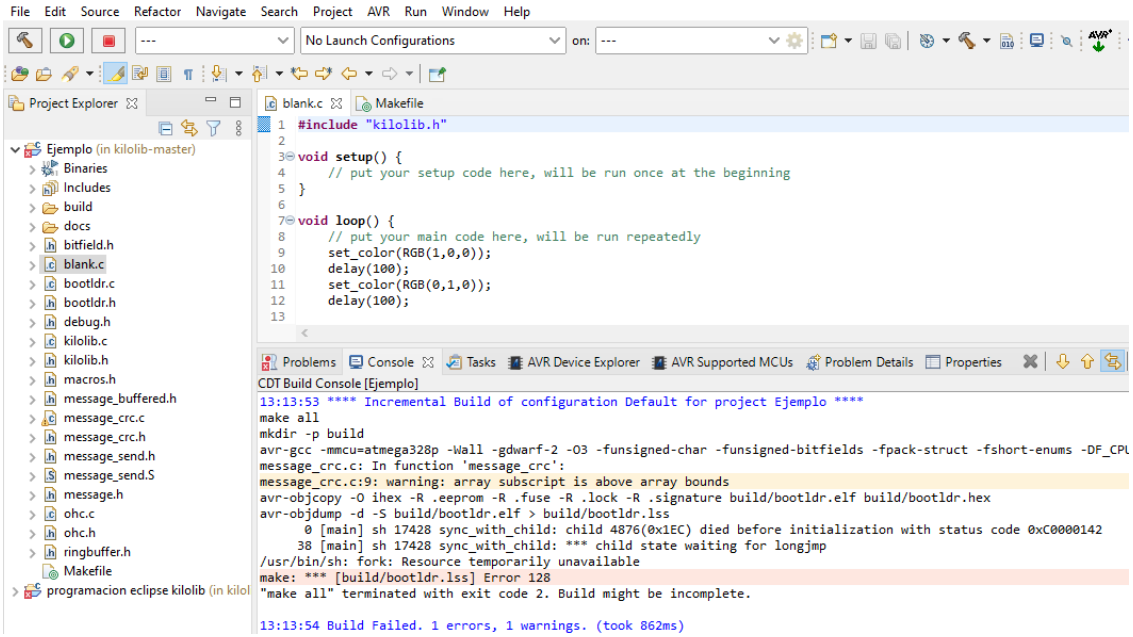


Figura A.10. Error en compilación debido al archivo bootloader

Como se ve en la consola de errores hay un conflicto cuando compila el archivo “bootldr”, este archivo está relacionado con el bootloader del Kilobot. Como solo interesa el archivo **blank.hex**, se modifica el constructor “**Makefile**” para que solo genere el archivo blank.hex.

Esto se consigue abriendo el archivo “**Makefile**” que muestra el código de la figura A.11:

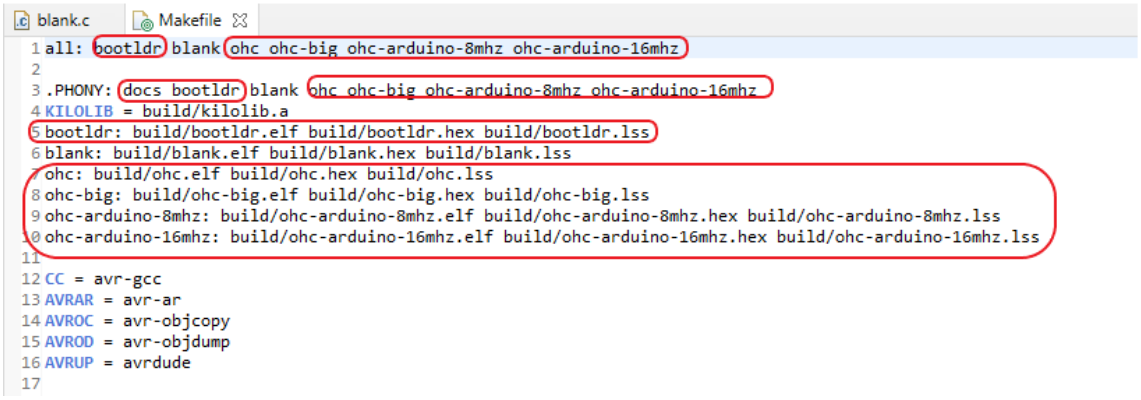


Figura A.11. Modificación del archivo Makefile (1)

En la figura A.11. y A.12. se muestra **todo lo escrito que no esté relacionado con el archivo blank** (código rodeado por un círculo rojo) y que por lo tanto se debe borra para que no se compile. Una vez se haya borrado todo lo no relacionado con el archivo **blank** se puede compilar todo con normalidad como se ha mostrado en el **paso 5** mostrado anteriormente.

blank.c Makefile

```
build/ohc.elf: ohc.c message_crc.c message_send.S | build
    $(CC) $(CFLAGS) $(OHC_FLAGS) -o $@ ohc.c message_crc.c message_send.S

build/ohc-big.elf: ohc.c message_crc.c message_send.S | build
    $(CC) $(CFLAGS) $(OHC_BIG_FLAGS) -o $@ ohc.c message_crc.c message_send.S

build/ohc-arduino-8mhz.elf: ohc.c message_crc.c message_send.S | build
    $(CC) $(CFLAGS) $(OHC_ARDUINO_FLAGS) -o $@ ohc.c message_crc.c message_send.S

build/ohc-arduino-16mhz.elf: ohc.c message_crc.c message_send.S | build
    $(CC) $(CFLAGS) $(OHC_ARDUINO_FLAGS) -DARDUINO_16MHZ -o $@ ohc.c message_crc.c message_send.S

build/bootldr.elf: bootldr.c kilolib.c message_crc.c | build
    $(CC) $(CFLAGS) $(BOOTLDR_FLAGS) -o $@ bootldr.c kilolib.c message_crc.c

program-ohc: build/ohc.hex
    $(AVRUP) -p m328  $(PFLAGS) -U "flash:w:$<:i"

program-ohc-big: build/ohc-big.hex
    $(AVRUP) -p m328  $(PFLAGS) -U "flash:w:$<:i"

program-ohc-arduino-8mhz: build/ohc-arduino-8mhz.hex
    $(AVRUP) -p m328p $(PFLAGS) -U "flash:w:$<:i"

program-ohc-arduino-16mhz: build/ohc-arduino-16mhz.hex
    $(AVRUP) -p m328p $(PFLAGS) -U "flash:w:$<:i"

program-boot: build/bootldr.hex
    $(AVRUP) -p m328p $(PFLAGS) -U "flash:w:$<:i"

program-blank: build/blank.hex build/bootldr.hex
    $(AVRUP) -p m328p $(PFLAGS) -U "flash:w:build/blank.hex:i" -U "flash:w:build/bootldr.hex"
```

Figura A.12. Modificación del archivo Makefile (2)

```
# Carga de las diferentes librerías que se utilizan
import b0RemoteApi
import math
import time
import random
import numpy as np
from IPython.core.debugger import Tracer

#Definición de la conexión con Coppelia
client = b0RemoteApi.RemoteApiClient('b0RemoteApi_V-REP','b0RemoteApi',60,True,60)#Se crea el cliente para la comunicación

#-----------------------------------------------------------------------------------------------------------------------
#----------------------------------------Configuración de los kilobots--------------------------------------------------
#-----------------------------------------------------------------------------------------------------------------------

#Creacion de la clase Kilobot
class Kilobot:
    #Constructor del Kilobot:
    def __init__(self,nameKilobot,nameRightMotor,nameLeftMotor,nameProximitySensor,nameMsgSensor,nameLightSensor):
        self.nameKilobot = nameKilobot   #El nombre del kilobot es el asignado como clase
        self.nameRightMotor = nameRightMotor
        self.nameLeftMotor = nameProximitySensor
        self.nameProximitySensor = nameProximitySensor
        self.nameMsgSensor = nameMsgSensor
        self.nameLightSensor = nameLightSensor

        #Handles de los componentes a usar
        self.ProximitySensorHandle = client.simxGetObjectHandle(nameProximitySensor,client.simxServiceCall())
        self.MsgSensorHandle = client.simxGetObjectHandle(nameMsgSensor,client.simxServiceCall())
        self.LightSensorHandle = client.simxGetObjectHandle(nameLightSensor,client.simxServiceCall())
        self.LeftMotorHandle = client.simxGetObjectHandle(nameLeftMotor,client.simxServiceCall())
        self.RightMotorHandle = client.simxGetObjectHandle(nameRightMotor,client.simxServiceCall())

        #Constantes
        self.__HalfDiameter = 0.034/2                #Radio del Kilobot
        self.__RatioMotor = 10/255                   #Se ha usado el ratio del código de Coppelia
        self.__MaxVelocity = 255                     #Indica la máxima velocidad que puede alcanzar un motor
        self.__AnchorsNumber = 3                     #Numero de anclas que habrá
        self.__AnchorsName = {'Kilobot0':[0,0],'Kilobot2':[0,0.14],'Kilobot4':[0.14,0]} #Nombres de los anclas
                                                                                        #de referencia y su ubicación
        self.__UnknowNodeNumber = 4                  #Numero de nodos desconocidos
        self.__Beta = -0.05                          #Factor para saber si unas coordenadas son válidas en el MCMM
        self.__ComunicationScope = 0.07              #Alcance de la comunicación de un Kilobot a otro
        self.__Eta = 1                               #Este valor se usa en el algoritmo BSA en la parte de mutación
        self.__Q = 1                                 #Valor de control para la función de cruce del BSA
        self.__NumberCycles = 20                     #Número de veces que se ejecutara el algoritmo BSA completo
        self.__IterationsBSA = 10                    #Este valor controla el número de veces que variara
                                                     #la matriz de mutación

        #Otros parametros del Kilobot
        self.Father = False            #Variable donde se almacena el nombre del Kilobot padre
        self.Counter = 0               #Un contador que se usara para contar eventos
        self.Counter2 = 0              #Otro contador
        self.StoreData = []            #Este dato se usara para almacenar datos
        self.Timer = 0.0               #Para llevar una cuenta interna cuando haga falta
        self.Timer2 = 0.0              #Para llevar una cuenta interna cuando haga falta
        self.Reading = 0.0             #Variable para almacenar un dato de un sensor
        self.LastReading = 0.0         #Para almacenar un dato antiguo de un sensor
        self.Motor = 0                 #Para indicar que motor ha sido usado
        self.Reference = False         #Para indiciar si es un nodo de referencia o no
        self.Coordinates = [0,0]       #Coordenadas en x e y
        self.AnchorList = {}           #Aqui almacena los mensajes de los anclas con sus coordenadas y
                                       #La distancia recorrida del mensaje
        self.UnknowNodeList = {}       #Almacenamiento de los nodos que desconocen su posición
        self.Anchor = False            #Aqui almacena si es un ancla o no
        self.AnchorAux = False         #Esta variable se usa en el algoritmo MCMM para poner
                                       #un nodo desconocido como ancla provisionalmente
        self.UnknowNodeReady ={}       #Diccionario donde se almacena si un nodo está listo
        self.NodeAnchorsLess = 0
        self.Delta = []                #Aqui almacena el valor delta x e y
        self.ListEx = []               #Estas variables sirven para la elección de Ex y Ey en el MCMM
        self.ListEy = []
        self.ConfidenceFactor = 0      #Factor de confianza para el algoritmo BSA
        self.NodeTwoHope = []          #Lista de nodos que están a 2 saltos
        self.P = []                    #Matriz P para el BSA
        self.Phist = []                #Matriz Phist para el BSA
        self.FitnessP = []             #Matriz Pfitnes que calcula guarda el error para el BSA
        self.Pmod = []                 #Matriz P mutada de BSA
        self.LeftNoise = 0             #Simulacion del ruido en los motores del Kilobot
        self.RightNoise = 0            #este ruido viene generado por las diferencias en el hardware o en la calibración
        self.FatherDistance = 0        #Distancia donde se encuentra el Kilboot padre
```

```
    #Banderas de señalizacion de eventos
    self.Flag  = False          #Bandera que indica algún evento
    self.Flag2 = False
    self.FlagEx_me = False      #Bandera que indica si se ha compartido los mensajes en el MCMM
    self.Flagx = False          #Bandera para indicar que se ha cargado la lista Ex
    self.Flagy = False          #Analoga a la anterior pero con la lista Ey
    self.ElectEx_Ey = False     #Indica que se ha elegido un ancla Ex y Ey
    self.FlagInitP = False      #Bandera para señalizar que ya se ha inicializado la población
    self.FlagStageI = True      #Banderas para poder cambiar de Stage todos al mismo tiempo
    self.FlagStageIII = False
    self.FlagStageIV = False
    self.FlagConfidence = False
    self.FlagNeigborhood = False
    self.FlagAdvance = False

    #Comunicacion con otros kilobots
    self.EnableReceived = 1    #Para indicar si recibe o no mensajes (0 = no), por defecto está escuchando
    self.State = False         #Para saber si está listo para una siguiente acción (True = listo)
    self.Memory = []           #Memoria de los mensajes que le llegan
    self.MemoryCount = 0       #Contador de memoria
    self.NeighborsList = {}    #Lista para almacenar los vecinos que tiene cerca un Kilobot
    self.SwarmState = {}       #Aqui almacena el estado de los kilobots para saber si están listos

    #Parametros para movimiento
    self.RightMotorVelocity = 0       #Estas variables sirven para indicar la velocidad que lleva el Kilobot
    self.LeftMotorVelocity = 0
    self.Velocity = 0

#------------------------------------------------------------------------------------------------------------------------
#----------------------------------------Metodos para los Kilobots-------------------------------------------------------
#------------------------------------------------------------------------------------------------------------------------

#Función para inicializar todas las variables
def Initialization(self,flagNoise):
    self.Straight(0)
    if(self.nameKilobot=='Kilobot0'):
        self.Father = self.nameKilobot
    else:
        self.Father = False
    if(flagNoise==True):
        self.LeftNoise = random.randrange(-30,30,1)        #El ruido es un valor aleatorio entre -30 y 30
        self.LeftNoise = random.randrange(-30,30,1)        #este valor es fijo
    else:
        self.LeftNoise = 0
        self.RightNoise = 0
    self.Counter = 0
    self.StoreData = []
    self.Timer = 0.0
    self.Timer2 = 0.0
    self.Reading = 0.0
    self.LastReading = 0.0
    self.Flag  = 0
    self.Flag2 = 0
    self.Motor = 0
    self.Reference = False
    self.AnchorList = {}
    self.UnknowNodeList = {}
    if(self.nameKilobot in self.__AnchorsName):                #Inicializa los anclas
        self.Anchor = True
        self.Coordinates = self.__AnchorsName[self.nameKilobot]
        self.ConfidenceFactor = 1
    else:
        self.Anchor = False
        self.Coordinates = []
        self.ConfidenceFactor = 0
    self.AnchorAux = False
    self.UnknowNodeReady = {}
    self.NodeAnchorsLess = 0
    self.SwarmState = {self.nameKilobot:self.State}
    self.Delta = []
    self.ListEx = []
    self.ListEy = []
    self.Flagx = False
    self.Flagy = False
    self.FlagEx_me = False
    self.ElectEx_Ey = False
    self.FlagInitP = False
```

```
        self.FlagStageI = True
        self.FlagStageIII = False
        self.FlagStageIV = False
        self.FlagConfidence = False
        self.FlagNeigborhood = False
        self.FlagAdvance = False
        self.NodeTwoHope = []
        self.P = []
        self.Phist = []
        self.FitnessP = []
        self.Pmod = []
        #Comunicacion con otros kilobots
        self.EnableReceived = 1
        self.State = False
        self.Memory = []
        self.MemoryCount = 0
        self.NeighborsList = {}
        #Parametros para movimiento
        self.RightMotorVelocity = 0
        self.LeftMotorVelocity = 0
        self.Velocity = 0

    #Función para establecer a un motor una velocidad se selecciona el motor y la velocidad
    #como el motor puede tener 255 velocidades se tiene que dar un número entre 0 y 255
    #(255=22.5º/s en rotación aproximadamente)
    def Set_velocity(self,motor,velocity):
        if (velocity<0):
            velocity=0      #Se configura el máximo y mínimo de velocidad
        if(velocity>self.__MaxVelocity):
            velocity = self.__MaxVelocity
        if (motor == 'Left'):
            self.LeftMotorVelocity = velocity
            _,LeftMotorHandle = self.LeftMotorHandle
            velocity = self.__RatioMotor*(velocity+self.LeftNoise)   #Se le añade el ruido a la velocidad
            return client.simxSetJointTargetVelocity(LeftMotorHandle,velocity,client.simxDefaultPublisher())
        else:
            self.RightMotorVelocity = velocity
            _,RightMotorHandle = self.RightMotorHandle
            velocity = self.__RatioMotor*(velocity+self.RightNoise)  #Se le añade el ruido a la velocidad
            return client.simxSetJointTargetVelocity(RightMotorHandle,velocity,client.simxDefaultPublisher())

    #Función para que gire a la izquierda
    def Turn_left(self):
        self.Set_velocity('Left',0)
        self.Set_velocity('Right',255)

    #Función para que solo gire a la derecha
    def Turn_right(self):
        self.Set_velocity('Left',255)
        self.Set_velocity('Right',0)

    #Función para que el kilobot avanze recto, también se puede usar para pararlo por completo
    def Straight(self,velocity):          #Si se activa a máxima velocidad avanza a 1 cm/s aproximadamente
        self.Velocity = velocity
        self.Set_velocity('Left',velocity)    #Avanza en dirección -x del eje de coordenadas del kilobot
        self.Set_velocity('Right',velocity)   #en dirección x del eje del sensor de proximidad, y en el eje 'x'
                                              #del sensor de luz
        return True

    #Función que devuelve la distancia de la luz (a menor distancia más luminosidad) y posición en los ejes
    def Light_sensor(self):
        _,LightSensorHandle = self.LightSensorHandle
        reading=client.simxReadProximitySensor(LightSensorHandle,client.simxServiceCall())
        if(reading[1]==1):
            distance = [round(reading[2],4),reading[3]] #La programación del sensor es diferente al real, ya que
                                                        #no se puede implementar el sensor real en Coppelia.
        else:
            distance = [-1,[0,0,0]]
        return distance

    #Función que pone el Kilobot en modo habla y retransmite un mensaje (este código difiere de uno real)
    def Broadcast_mode(self,message):
        self.EnableReceived = 0          #Establece que no puede recibir ningún mensaje
        broadcast_message(self,message)  #Esta función representa al mensaje viajando por el ambiente
        return True
```

```
#Poner a un Kilobot en modo escucha
def Listen_mode(self):
    self.EnableReceived = 1 #Habilita la escucha
    return True

#Función para almacenar un dato internamente en la memoria del Kilobot(para la recepción de mensajes)
def Store_data(self,data):
    self.Memory.append(data)      #Añade el dato introducido a la memoria
    counter = self.MemoryCount
    self.MemoryCount = counter+1  #Y sube en uno el contador de memoria
    return True

#Borrar memoria de los mensajes llegados
def Delete_memory(self):
    self.Memory = []
    self.MemoryCount = 0
    return True

#Función de temporizador, se introduce un numero y devuelve True cuando ha pasado el tiempo introducido
def Variable_timer(self,timeSelect):
    time = client.simxGetSimulationTime(client.simxServiceCall())[1] #Almacena el tiempo actual
    if(self.Timer==0):     #Comprueba si el temporizador esta incializado
        self.Timer = time  #Se asigna el tiempo actual para contar
        return False       #Devuelve False para indicar que no ha pasado el tiempo introducido
    elif(time-self.Timer>=timeSelect):  #Comprueba si ha pasado el tiempo introducido
        self.Timer = 0    #Si ha pasado se reinicia el temporizador
        return True       #Devuelve True
    else:                 #Esta programacion del temporizador en un Kilobot real se tendria que hacer de otra forma
        return False      #con un contador interno del Kilobot y la velocidad del procesador de este

#Algoritmo para almacenar los vecinos adyacentes en una lista (Wave Algorithm for Recruitment)
def Neighborhood_discover(self,Ts):
    if(self.Flag==0): #Comprueba si es la primera vez que pasa por esta función
        self.Broadcast_mode(self.nameKilobot)    #Pone al Kilobot a retransmitir su nombre
        self.Listen_mode()                       #Lo pone en modo escucha para espera una respuesta de los vecinos
        self.Flag = 1                            #Almacena que ha retransmitido
    if(not(self.Variable_timer(Ts))): #Este es el condicional del bucle que depende de
                                      #un tiempo dado que está esperando a que le respondan
        for i in self.Memory:         #Aquí registra si le ha llegado en la memoria los mensajes que le han llegado
            if(not(i[2] in self.NeighborsList)): #Comprueba si lo tiene en la lista o si todavía no le ha llegado nada
                self.NeighborsList.update({i[2]:i[1]}) #El vecino lo va añadiendo a la lista junto a su distancia
        self.Delete_memory()
        return False
    else:
        self.Listen_mode()
        self.Flag=0                       #Aquí reinicia todas las variables utilizadas
        self.Buffer = [False,False,False]
        self.Delete_memory()
        return True

#Función para almacenar los kilobots que están a 2 saltos del que la inicia
def Node_two_hope_discover(self,Ts):
    if(self.Flag2==False):                  #Primero se necesita la lista de vecinos por lo que
        if(self.Neighborhood_discover(Ts)):  #inicia el algoritmo Neighborhood_discover()
            self.Flag2 = True               #Almacena en esta bandera que se ha realizado
        return False
    elif(not(self.Variable_timer(Ts))):
        for message in self.Memory:               #Revisa en la memoria los mensajes que le
            if('Tw_dis' in message[0]):           #han llegado que son la lista de sus vecinos
                NeighborList = message[0][2]
                for ID in NeighborList:          #Con este bucle revisa la lista
                    if (not(ID in self.NodeTwoHope)and not(ID in self.NeighborsList) and ID!=self.nameKilobot):
                        self.NodeTwoHope.append(ID)       #Lo almacena si no está en su lista de vecinos y
                                                          #si no está ya en la lista
        self.Broadcast_mode(['Tw_dis',self.nameKilobot,self.NeighborsList])   #Despues retransmite su lista de vecinos
        self.Listen_mode()
        self.Delete_memory()
        return False
    else:
        self.Delete_memory()
        self.Flag2 = False    #Cuando termina lo inicializa todo y devuelve true
        return True

#Algoritmo Suma-distancia para un nodo desconocido (STAGE-I)
def Sum_Dist(self,Ts):
    if(self.Anchor==True or self.AnchorAux==True):  #Si es un ancla retransmite sus datos de nombre coordenadas
        self.Broadcast_mode([self.nameKilobot,self.Coordinates[0],self.Coordinates[1],0])#y distancia del mensaje
        if(self.Variable_timer(Ts)):
            return True
```

```
            else:
                return False
        else:
            if(not(self.Variable_timer(Ts))): #Revisa que no ha pasado el tiempo (recomendado Ts=7)
                anchorList = self.AnchorList  #Almacena la variable anchorlist que se va a utilizar
                if(self.MemoryCount!=0):      #Comprueba si hay algún mensaje
                    for n in range(self.MemoryCount): #El bucle es para revisar todos los mensajes
                        message = self.Memory[n][0]   #Almacena el mensaje
                        if(len(message)==4):          #Comprueba que tiene la estructura del algoritmo
                            Div=self.Memory[n][1]     #Almacena la distancia a la que está el remitente del mensaje
                            if not(message[0] in anchorList):  #Comprueba si lo tiene en la lista de anclas
                                Lir = message[3]+Div  #Si no lo tiene almacena la distancia a la que está el ancla sumandole
                                                      #la del nodo
                                anchorList.setdefault(message[0],[message[1],message[2],Lir])#Guarda el ancla y su información
                                self.Broadcast_mode([message[0],message[1],message[2],Lir])  #Retrasmite un mensaje con la
                                                                                             #información del ancla
                            else:     #Si conoce el ancla
                                value = anchorList[message[0]]    #Almacena los valores del mensaje referentes al ancla
                                if(Div+message[3]<value[2]):      #Comprueba si la distancia que ha recorrido es más larga
                                    Lir = message[3]+Div          #Si es una distancia más corta lo actualiza
                                    anchorList.update({message[0]:[message[1],message[2],Lir]})
                                    self.Broadcast_mode([message[0],message[1],message[2],Lir]) #Retransmite otra vez el ancla
                self.Listen_mode()            #Lo pone en escucha
                self.AnchorList = anchorList  #Almacena en memoria el nuevo diccionario
                return False                  #Devuelve false si no ha terminado el tiempo
            else:
                self.Delete_memory()     #Borra los mensajes que le han llegado cuando termina el algoritmo
                return True              #Devuelve True cuando ha pasado el tiempo

    #Algoritmo Min-Max para la obtención de las coordenadas Ui que son las coordenadas del Kilobot(nodo) en bruto (STAGE-II)
    def Min_Max(self):
        anchorList = self.AnchorList      #Carga la lista de anclas
        values = []                       #Se tiene que extraer con un bucle for porque la función values()
        for i in anchorList.values():     #da una direccion y no una lista
            values.append(i)              #Extrae de la dirección de todos las anclas la distancia Lir,xr e yr
        Bir1x = []                        #Crea una lista donde se almacenará todos los valores xr-Lir
        Bir1y = []                        #de la misma forma con yr-Lir
        Bir2x = []                        #y con xr+Lir e yr+Lir
        Bir2y = []                        #Estas matrices son las coordenadas del área cuadrada donde estará el nodo
        for i in range(len(values)):
            Bir1x.append(values[i][0]-values[i][2])   #Almacena los valores de todos los nodos
            Bir1y.append(values[i][1]-values[i][2])
            Bir2x.append(values[i][0]+values[i][2])
            Bir2y.append(values[i][1]+values[i][2])
        Si = [[max(Bir1x),max(Bir1y)],[min(Bir2x),min(Bir2y)]]  #Se obtiene el máximo de BR1x y bir1y y
                                                                #Los mínimos Br2x y Bir2y
        Deltax = Si[1][0]-Si[0][0]                              #Deltax y Deltay son los valores de las dimensiones de la
                                                                #caja que contiene Ui estos dos datos son necesarios para
        Deltay = Si[1][1]-Si[0][1]                              #ejecutar el algoritmo MCMM
        Ui = [(Si[0][0]+Si[1][0])/2,(Si[0][1]+Si[1][1])/2]      #Ui son las coordenadas en bruto que son la media de la suma
        return [Ui,[Deltax,Deltay]]  #de los valores x y la suma de los valores y, además de devolver también Deltax y Deltay

    #Algoritmo MCMMM(Multi-hop collaborative Min-Max) este algoritmo sirve cuando se tienen insuficientes
    #anclas en el STAGE-I (STAGE-III)
    def MCMM(self,Ts):
        if(not(self.FlagEx_me)):                    #Revisa si la bandera de la función Exchange_messages es True
            dates = [self.Coordinates,self.Delta]   #Almacena los datos que va a compartir con los nodos
            if(self.Exchange_messages(dates)==True):#Activa la función en la que los Kilobots intercambian
                self.FlagEx_me = True               #información de los nodos
                self.Delete_memory()                #Pone la bandera en True para que no vuelva a entrar
        elif(not(self.ElectEx_Ey)):       #Comprueba con la bandera ElctEx_Ey si se ha elegido un Ex y Ey
            self.ElectEx_Ey = True        #Pone en True la bandera que indica si se ha elegido uno
            Ex = self.Elect_ex()          #Activa las funciones que eligen Ex y Ey
            Ey = self.Elect_ey()
            if(Ex==self.nameKilobot or Ey==self.nameKilobot): #Si el nombre elegido coincide con el propio
                self.AnchorAux = True                         #se activa como nodo auxiliar
            elif((Ex==True) or (Ey==True)): #Si Ex o Ey son True quiere decir q la lista esta vacía por lo que
                self.FlagEx_me = False      #ya se han evaluado todos los nodos
                self.Flagx = False          #Inicializa todas las variables utilizadas y devuelve True indicando
                self.Flagy = False          # que ha terminado
                self.ListEx = []
                self.ListEy = []
                self.Delete_memory()
                self.State = False
                self.ElectEx_Ey = False
                return True
            else:
                self.AnchorAux = False          #Si no es de ningún otro valor se inicializa el ancla auxiliar
```

```
        if(self.ElectEx_Ey and self.FlagEx_me): #Si ya se ha compartido la información y elegido los anclas auxiliar
            if(self.Sum_Dist(Ts)==True):        #Llama a la funcion Sum_dist y si termina (devuelve True)
                self.ElectEx_Ey = False         #Inicializa la bandera para elegir un nuevo ancla auxiliar
                self.AnchorAux = False
                if(self.Anchor!=True):           #Si no es ancla
                    self.Clear_AnchorListAux()   #Limpia la lista de anclas de las auxiliares
                    if(len(self.AnchorList)<3):  #revisa la primera condición del algoritmo(si tiene menos de tres anclas)
                        values=[]
                        coordinates = self.Min_Max() #Llama a la función min_max para almacenar las nuevas x,y e delta
                        for i in self.AnchorList.values():
                            values.append(i)
                        if(len(self.AnchorList)==1):  #Y calcula la beta dependiendo si tiene 1 o dos anclas
                            lir = values[0][2]
                            xr = values[0][0]
                            yr = values[0][1]
                            beta = (lir/(((xr**2)+(yr**2))**(1/2)))-1
                        elif(len(self.AnchorList)==2):
                            lir1 = values[0][2]
                            lir2 = values[1][2]
                            xr1 = values[0][0]
                            xr2 = values[1][0]
                            yr1 = values[0][1]
                            yr2 = values[1][1]
                            beta = ((lir1+lir2)/((((xr1-xr2)**2)+(yr1-yr2)**2)**(1/2)))-1
                        else:
                            beta = -1          #Si no encuentra ningún ancla beta es -1
                        if(beta<=self.__Beta): #Si cumple la condición de la beta calculada es menor que la constante beta
                            self.Coordinates = coordinates[0]#Actualiza los valores de las coordenadas y deltas
                            self.Delta = coordinates[1]
            self.Delete_memory()
        return False

    #Función para limpiar las anclas auxiliares de la lista de la lista de anclas
    def Clear_AnchorListAux(self):
        anchorList = []
        for i in self.AnchorList.keys():
            anchorList.append(i)   #Crea una lista auxiliar con las que hay actualmente
        for i in anchorList:       #Recorre la lista auxiliar
            if(not((i in self.__AnchorsName))):  #Si no está en la lista de anclas referencia(constante)
                self.AnchorList.pop(i)           #la borra de la lista de anclas
        return True

    #Función para elegir el menor delta x que no haya sido elegido si se han elegido todos devuelve True
    def Elect_ex(self):
        if(self.Flagx !=True or len(self.ListEx)!=0):  #Con la variable Flagx revisa si es la primera vez que
            if(self.Flagx !=True):                     #entra en la función si es la primera vez lo señaliza
                self.Flagx = True
                for i in self.UnknowNodeList.items():    #Y crea una lista con los nombres y los delta x de cada nodo
                    self.ListEx.append([i[0],i[1][1][0]])
            deltasX = []
            for i in self.ListEx:
                deltasX.append(i[1])  #Hace una lista de todos los valores deltaX
            minValue = min(deltasX)   #saca el mínimo valor
            index = deltasX.index(minValue) #Y su índice en la lista
            elect = self.ListEx[index][0]  #El índice del valor es el kilobot que será el ancla auxiliar
            self.ListEx.pop(index)         #Lo borra de la lista para que no pueda volver a ser elegido
            return elect                   #Devuelve el valor del kilobot que será el nuevo ancla
        else:            #Si la bandera Flagx y la lista de Ex no tiene elementos quiere decir que ya se
            return True #han evaluado todos los kilobots por lo que devuelve True

    #Función para elegir el menor deltay que no haya sido elegido
    def Elect_ey(self):
        if(self.Flagy !=True or len(self.ListEy)!=0):
            if(self.Flagy !=True):#Esta función es análoga a la anterior pero cogiendo los valores deltay
                self.Flagy = True
                for i in self.UnknowNodeList.items():
                    self.ListEy.append([i[0],i[1][1][1]])
            deltasY = []
            for i in self.ListEy:
                deltasY.append(i[1])
            minValue = min(deltasY)
            index = deltasY.index(minValue)
            elect = self.ListEy[index][0]
            self.ListEy.pop(index)
            return elect
        else:
            return True
```

```
    #Función para compartir la información de los nodos desconocidos
    def Exchange_messages(self,dates):
        if(self.State==False):          #Comprueba que si tiene la lista
            self.Clear_same_message()   #Limpia los mensajes repetidos
            for i in self.Memory:
                messageId = i[0][0]
                messageOrigin = i[0][1]    #Revisa los mensajes que le han llegado
                message = i[0][2]
                if(messageId=='Ex_me'):    #Comprueba que pertenecen a la función
                    self.UnknowNodeList.update(message) #Si pertenece actualiza la información
                else:
                    self.Broadcast_mode(i[0]) #Sino la comparte con el enjambre por si alguno lo necesita
            if(self.Anchor!=True):            #Si no es ancla se mete en la lista de nodos desconocidos
                self.UnknowNodeList.update({self.nameKilobot:dates})
            self.Broadcast_mode(['Ex_me',self.nameKilobot,self.UnknowNodeList]) #retransmite la lista de nodos
            self.Listen_mode()
            self.Delete_memory()
            if(len(self.UnknowNodeList)==self.__UnknowNodeNumber): #Si la longitud de la  lista de nodos desconocidos
                                                                   #coincide con el número de nodos desconocidos
                self.State = True   #Pone que está listo para pasar a la siguiente parte
            else:
                return False
        elif(self.Swarm_state()): #Comprueba si el grupo está listo
            return True           #Si lo está devuelve True
        else:
            return False

    #Cálculo del factor de confianza para su uso en el BSA y lo comparte con el enjambre
    def Confidance_factor(self):
        if(len(self.UnknowNodeList)<self.__UnknowNodeNumber): #Comprueba que la lista de nodos tiene el mismo
            if(self.Anchor!=True):                            #número de elementos que nodos desconocidos hay
                confidanceFactor = 1+(self.Delta[0]*self.Delta[1])     #Calcula el factor de confianza
                self.ConfidenceFactor = confidanceFactor
                self.UnknowNodeList.update({self.nameKilobot:[self.Coordinates,confidanceFactor]}) #Se incluye en la lista
            for i in self.Memory:      #Comprueba la memoria los mensajes que le han llegado
                if not('Sw_st'in i[0]):#Si no es un mensaje de la función 'Swarm_state'('Sw_st') quiere
                                       #decir que si es de esta función
                    self.UnknowNodeList.update(i[0]) #Actualiza la lista con el mensaje
            self.Broadcast_mode(self.UnknowNodeList) #y comparte la lista
            self.Listen_mode()
            self.Delete_memory()
            return False
        else:   #Si ya tiene el mismo número de elementos en el diccionario de
                #'UnknowNodeList' que número de nodos desconocidos
            self.Delete_memory()  #Borra la memoria
            return True           #y devuelve true para indicar que ha terminado

    #Algoritmo para refinar los resultados con BSA (STAGE-IV)
    def Refinament_with_BSA(self):
        if(self.Anchor==True):        #Comprueba si es ancla
            self.Clear_same_message() #Borra mensajes repetidos
            for i in self.Memory:     #y los retrasmite por si los necesita alguien
                self.Broadcast_mode(i[0])
            self.Delete_memory()
            self.Listen_mode()
            return True               #Devuelve true
        else:                         #Si no es un ancla
            if not(self.FlagInitP):   #Revisa si esta inicializada la matriz P comprobando la bandera 'FlagInitP'
                self.P,self.Phist = self.Initialize_P()  #Inicializa la matriz P y Phist llamando a la función 'Initialize_P'
                self.FitnessP = self.Compute_fitness(self.P) #Y calcula el error de todas las coordenadas
                self.FlagInitP = True   #Almacena en la bandera que ya se han inicializado
            else:
                self.Clear_same_message()   #Limpia los mensajes iguales en memoria
                for message in self.Memory: #y los revisa
                    if ('BSA' in message[0]):  #Si tienen la etiqueta 'BSA' pertenecen a la función por lo que los lee
                        for kilobot in message[0][2]: #Revisa el diccionario
                            name = kilobot
                            data = message[0][2][name]
                            if(len(self.UnknowNodeList[name])<3):       #Actualizando los datos del diccionario
                                self.UnknowNodeList.update({name:data}) #Lo actualiza si no lo tiene
                            elif(len(data)>2 and data[2]<self.UnknowNodeList[name][2]):#o si el valor del error
                                self.UnknowNodeList.update({name:data})               #es mejor que el que tiene almacenado
                self.Delete_memory() #Borra memoria
                if(self.Counter<=self.__NumberCycles):  #Comprueba el número de ciclos que lleva
                    counter = self.Counter
                    for r in range(self.__IterationsBSA): #Este ciclo es para repetir el BSA
                        self.Selection_I()     #Llama a la primera función 'Selection_I'
                        PMod = self.Mutation() #ahora a la función de mutación y almacena el valor de la matriz Pmod
```

```
                    PNew = self.Crossover_operation(PMod) #Con la matriz Pmod calculada llama a la función
                                                          #de cruce devolviendo Pnew
                    minimal = self.Selection_II(PNew) #Con la matriz Pnew se llama a la función 'Selection_II'
                                                      #devolviendo el mínimo valor de error
                indexMinimal = self.FitnessP.index(minimal) #Con ese valor mínimo se busca su índice en la
                                                            #matriz de error 'FitnessP'
                coordinatesMinimal = self.P[indexMinimal]   #y el valor de sus coordenadas se encuentra con ese índice
                if(coordinatesMinimal!=self.Coordinates):   #Si son diferentes a las actuales se actualizan los valores
                    self.Coordinates = self.P[indexMinimal] #Las coordenadas
                    Min = min(self.FitnessP)
                    self.UnknowNodeList.update({self.nameKilobot:[self.Coordinates,self.ConfidenceFactor,Min]})
                                                            #y la matriz de los nodos con su nuevo valor calculado
                    self.Broadcast_mode(['BSA',self.nameKilobot,self.UnknowNodeList])#Se retransmite el diccionario de nodos
                    self.Listen_mode()  #Y lo pone en escucha
                counter+=1
                self.Counter = counter  #Se almacena que ha hecho un ciclo
            else:
                self.Delete_memory()   #Si termina todos los ciclos devuelve True y borra la memoria
                return True
        return False   #Para cualquier otro resultado devuelve False

    #Inicialización de la población P y Phist
    def Initialize_P(self):      #Genera una matriz P que contiene valores aleatorios que siguen una distribución uniforme
        Px = np.random.uniform(self.Coordinates[0]-abs(self.Delta[0]/2),self.Coordinates[0]+abs(self.Delta[0]/2),4)
        Py = np.random.uniform(self.Coordinates[1]-abs(self.Delta[1]/2),self.Coordinates[1]+abs(self.Delta[1]/2),4)
        P = []                   #El rango de la distribución será del área S que viene definida por delta
        count = 0
        for i in Px:
            P.append([i,Py[count]])  #Crea la matriz P con las coordenadas x e y
            count+=1
        P.append(self.Coordinates) #Agrega los valores de las coordenadas actuales
        PxHist = np.random.uniform(self.Coordinates[0]-abs(self.Delta[0]/2),self.Coordinates[0]+abs(self.Delta[0]/2),5)
        PyHist = np.random.uniform(self.Coordinates[1]-abs(self.Delta[1]/2),self.Coordinates[1]+abs(self.Delta[1]/2),5)
        Phist = []     #Crea la matriz Phist análogamente a la anterior
        count = 0
        for i in PxHist:
            Phist.append([i,PyHist[count]])
            count+=1
        return [P,Phist]  #La problación que se ha escogido es de 5 pero se habría podido coger otra

    #Calcula el error de unas coordenadas introducidas
    def Compute_error(self,inCoordinates):
        errorG = 0
        for i in self.NeighborsList:       #Recorre la lista de vecinos
            if(i in self.__AnchorsName):   #Si es un ancla su factor de confianza y sus coordenadas son fijas
                confidance = 1
                coordinates = self.__AnchorsName[i]
            else:
                dates = self.UnknowNodeList[i]  #Si no lo es los datos del vecino los tiene en el diccionario
                confidance = dates[1]           #de nodos desconocidos
                coordinates = dates[0]
            pv_pi =[inCoordinates[0]-coordinates[0],inCoordinates[1]-coordinates[1]] #Calcula el vector pv-pi
                                                    #(pv=posicion vecino, pi=posición introducida)
            module=(((pv_pi[0])**2)+(pv_pi[1])**2)**(1/2) #Calcula el módulo del vector
            error = self.NeighborsList[i]-module    #Calcula el error con la distancia a este vecino
            errorG = errorG+((error**2)/confidance) #y lo almacena en errorG dividiendo el error por el factor de confianza
        errorH = 0
        for i in self.NodeTwoHope:
            if(i in self.__AnchorsName):    #El cálculo de este error es análogo al anterior
                confidance = 1              #pero usando la lista de nodos a dos saltos
                coordinates = self.__AnchorsName[i]
            else:
                dates = self.UnknowNodeList[i]
                confidance = dates[1]
                coordinates = dates[0]
            pw_pi = [inCoordinates[0]-coordinates[0],inCoordinates[1]-coordinates[1]]
            module = ((pw_pi[0]**2)+pw_pi[1]**2)**(1/2)
            error = (max([0,self.__ComunicationScope-module,module-2*self.__ComunicationScope]))**2 #El error es el maximo
                                                                                                    #de estos tres valores
            errorH = errorH+(error/confidance)
        errorT = errorG+errorH
        return errorT

    #Función para evaluar una matriz P introducida y calcular el error de todos los valores de esta y devuelve fitnessP
    def Compute_fitness(self,P):
        fitnessP = []
        for coordinates in P:                       #Recorre la lista de coordenadas
            fitness = self.Compute_error(coordinates) #Calcula el error
            fitnessP.append(fitness)                #Y lo almacena en la variable fitnessP
```

```
        return fitnessP   #Devuelve una matriz con los valores calculados

    #Fase Selection-I del algoritmo BSA en el que se elige aleatoriamente P o Phist y devuelve
    #Phist con los valores barajados
    def Selection_I(self):
        a = np.random.uniform(0, 1)
        b = np.random.uniform(0, 1)
        if(a<b):  #Aqui es donde aleatoriamente elige actualizar Phist
            self.Phist = self.P
        random.shuffle(self.Phist)  #Con esta función se barajan las coordenadas
        return True

    #Cálculo de la matriz de mutación del BSA
    def Mutation(self):
        count = 0
        T = np.random.normal(0,1)   #Calcula el valor T con una distribución normal con centro en 0 y sigma 1
        Pmod = []
        for coordinates in self.P:
            px = coordinates[0]+(self.__Eta)*T*(self.Phist[count][0]-coordinates[0])
            py = coordinates[1]+(self.__Eta)*T*(self.Phist[count][1]-coordinates[1])
            Pmod.append([px,py])   # Y de acuerdo a la función del cálculo de la matriz
            count +=1              #de mutacion va almacenando los valores de las coordenadas en Pmod
        return Pmod

    #Acción de cruce en el algoritmo BSA
    def Crossover_operation(self,Pmod):
        a = np.random.uniform(0, 1)  #Genera dos números a y b de forma aleatoria siguiendo una distribución uniforme
        b = np.random.uniform(0, 1)  #entre 0 y 1
        X = []                       #Genera la matriz de cruce X
        for i in self.P:
            X.append([1,1]) #Inicializa la matriz X con filas(1,1) con la misma dimensión que P
        if (a<b):  #Aleatoriamente con los valores de a y b se decide el método a utilizar en esta iteración
            for i in range(len(X)): #Por cada fila de X se decide si un valor se cambia (0) o no (1)
                u=int(abs(round(np.random.normal(0,1)*2*self.__Q,1))) #Siguiendo la formula del algoritmo
                if(u>=2):    #Si u es mayor q 2
                    X[i] = [0,0] #La fila se pone a [0,0]
                else:
                    X[i][u] = 0 #Sino solo cambia el valor de u q es 0 o 1
        else: #Este método es parecido al anterior pero solo cambia uno de los valores de cada fila
            for i in range(len(X)):
                rndi=int(abs(round(np.random.normal(0,2),1))) #Usando una distribución normal entre 0 y 2 con centro en 1
                if(rndi>=2):
                    rndi = 1
                elif(rndi<0):
                    rndi = 0
                X[i][rndi] = 0  #Cambia uno de los valores o 0 o 1
        Pnew = Pmod             #Genera la nueva matriz Pnew
        counti = 0
        countj = 0
        for coordinates in self.P:
            for i in coordinates:
                if(X[counti][countj]==1):    #Usando la matriz X que si tiene alguna coordenada 1 se guarda el valor de P
                    Pnew[counti][countj] = i #en esa misma posición de X en Pnew
                countj+=1
            countj = 0
            counti+=1
        return Pnew #Al final se devuelve una matriz Pnew que es un cruce de Pmod y P con valores de P aleatorios definidos
                    #por la matriz X

    #Selection-II del algoritmo BSA
    def Selection_II(self,Pnew):
        for coordinates in Pnew:                            #Aqui revisa las coordenadas de Pnew
            for count in range(len(self.FitnessP)):         #Y en este bucle el error
                fitness = self.Compute_error(coordinates)   #Calcula el error con esas coordenadas
                if(fitness<self.FitnessP[count]):           #Y si tiene menor error
                    self.P[count] = coordinates             #Guarda las nuevas coordenadas
                    self.FitnessP[count] = fitness          #y actualiza el error en la matriz FitnessP
        return min(self.FitnessP)                           #Devuelve el mínimo error de la matriz FitnessP

    #Función en la que los Kilobots se informan entre ellos para saber si están listos
    #(sirve para saltar de una tarea a otra todos los kilobots de forma conjunta)
    def Swarm_state(self):
        self.Clear_same_message()
        for i in self.Memory:                                   #Revisa la memoria de mensajes
            if('Sw_st' in i[0]):
                message = i[0][2]                               #Guarda el mensaje
                for kilobotName in message.keys():              #Revisa todas las claves a ver si tiene algún True
                    if(message[kilobotName]==True):             #Si algún kilobot tiene True(está listo)
```

```
                    self.SwarmState.update({kilobotName:True}) #Lo actualiza
        elif('BSA'in i[0]):
            self.Broadcast_mode(i[0])
        elif(i[0][1] in self.SwarmState):
            if(not(self.SwarmState[i[0][1]])):#Solo lo comparte si el remitente ha dejado de transmitir ese mensaje
                self.Broadcast_mode(i[0])     #Si es un dato que no es de la función lo comparte por si lo necesita
                                              #otro kilobot
    self.SwarmState.update({self.nameKilobot:self.State})           #Actualiza el estado actual del propio kilobot
    self.Broadcast_mode(['Sw_st',self.nameKilobot,self.SwarmState]) #Retransmite el estado del enjambre
    self.Listen_mode()                                  #y lo pone en modo escucha
    self.Delete_memory()                                #borra la memoria de mensajes
    Swarmstate=[]
    for i in self.SwarmState.values():           #En este bucle almacena los estados de todos los kilobots
        Swarmstate.append(i)
    if(False in Swarmstate or len(Swarmstate)<7): #Si hay algún kilobot que no está preparado devuelve False
        return False
    else:                                        #sino devuelve True y reinicia el estado del enjambre
        self.State = False                          #Inicializa el estado
        self.SwarmState ={self.nameKilobot:self.State}      #y el diccionario de estados de los kilobots
        return True        #(este paso hay que asegurarse llamando fuera del bucle a la función reset_Swarm())

#Función que sirve para borrar que provienen del mismo origen y continen lo mismo
#(sirve para que no se sature la memoria con mensajes repetidos)
def Clear_same_message(self):
    messageList = []
    count = 0
    for i in self.Memory:
        if(not(i[0] in messageList)):
            messageList.append(i[0])  #Va creando una lista con los mensajes que no tiene en esta
        else:                         #Si lo tiene quiere decir que esta repetido por lo que lo quita
            self.Memory.pop(count)
        count+=1

#Algoritmo completo de localización de los kilobots
def Complete_algorithm(self,Ts):  #El algoritmo completo se ejecutara a partir de banderas que indiquen que
                                  #han ido haciendo cada etapa
    if(self.FlagStageI):          #Comprueba la bandera de la primera etapa
        if(self.Sum_Dist(Ts)):    #Una vez la primera función del algoritmo termina devuelve True
            self.FlagStageI = False #Desactiva la bandera de la primera etapa
            self.FlagStageIII = True #Activa la de la tercera ya que la segunda solo la hacen los nodos y
            if(self.Anchor!=True):   #es un cálculo rápido
                self.Coordinates,self.Delta=self.Min_Max()  #Aqui calcula las coordenadas los nodos desconocidos
            print('StageII; '+self.nameKilobot +':'+str(self.Coordinates)+'  Number of Anchors:'+str(len(self.AnchorList)))
            #Muestra en pantalla las coordenadas y el número de anclas detectados
            self.Delete_memory()     #Borra memoria
            return False
    elif(self.FlagStageIII):  #De forma análoga a la anterior se entra en la etapa 3
        if(self.MCMM(Ts)):
            self.FlagStageIII = False    #Una vez termina se almacena
            self.FlagConfidence = True   #Se activa la etapa en la que se comparte el factor de confidencia
            self.UnknowNodeList = {}     #Se reinicia el diccionario de información de los nodos
            self.State = True            #Y pone el estado que está listo para saltar a la siguiente etapa
            self.Delete_memory()
            print('StageIII; '+self.nameKilobot +':'+str(self.Coordinates))
            return False
    elif(self.FlagConfidence):
        if(self.State==True):      #Comprueba si está listo para la siguiente etapa y activa la función 'Swarm_state'
            if(self.Swarm_state()):#para que todos los kilobots pasen al mismo tiempo a esta etapa y no se mezclen datos
                self.State = False #Cuando devuelve true pone el estado en false para pasar a la siguiente etapa
                self.Delete_memory()
        elif(self.Confidance_factor()): #Llama a la función que calcula y comparte con todo el enjambre el factor
            self.FlagNeigborhood = True #de confianza
            self.FlagConfidence = False #Activa la bandera de la siguiente etapa y desactiva la del factor de confianza
            self.State = True
            self.Delete_memory()
        return False
    elif(self.FlagNeigborhood):   #Este algoritmo es análogo al anterior, se llama a la función 'Swarm_state' para
        if(self.State==True):     #que todos los kilobots estén en la misma etapa
            if(self.Swarm_state()):
                self.State = False
                self.Delete_memory()
        elif(self.Node_two_hope_discover(Ts)): #Aquí activa la función que hace descubrir los nodos a 1 y 2 saltos
            self.FlagNeigborhood = False  #Desactiva la bandera de descubrir el vecindario y lo pone listo para saltar
            self.State = True             #a la siguiente etapa
            self.Delete_memory()
            return False
    else:
        if(self.State==True):
```

```
            if(self.Swarm_state()): #Esta parte es análoga a la anterior que sirve para que todos los kilobots estén en la
                self.State = False  #la misma etapa
                self.Delete_memory()
                if(self.FlagStageIV==True): #Aquí la diferencia es que la bandera de la fase IV se utiliza para indicar
                    return True             #que se ha terminando el algoritmo y devuelve True
                else:                       #Con estos condicionales obliga a todos los kilobots a terminar al mismo tiempo
                    return False
        elif(self.Refinament_with_BSA()):  #Se activa la función BSA para refinar la posición
            self.State = True              #Cuando termina lo pone en listo para saltar
            self.FlagStageIV = True        #Y activa la bandera indicando que ya ha terminado
            self.Delete_memory()
            return False

#Función para guiar el enjambre hacia la luz como los Kilobots reales
def Leader_replicate(self):
    self.EnableReceived = 0                #Desactiva la escucha de mensajes
    readings = self.Light_sensor()[1]      #Obtiene las lecturas del sensor de luz(coordenadas x e y relativas a el)
    angle = alpha(readings[0],readings[1]) #Calcula el ángulo de apertura respecto al foco
    if(self.Flag==False):                  #Inicialización del Kilobot
        self.Flag = True                   #Señaliza en la bandera que ya está inicializado
        self.Broadcast_mode('Left')        #Empieza moviéndose a izquierda y lo retransmite al enjambre
        self.Turn_left()                   #Llama a la función de girar a izquierdas
    else:                                  #Si ya está inicializado
        if(angle<300 and angle>180):       #Comprueba que el ángulo está en el rango de giro a derechas
            self.Broadcast_mode('Right')   #Si lo está lo retransmite
            self.Turn_right()              #y empieza a girar a derechas
        elif(angle>60 and angle<180):      #Si está en el rango de giro a izquierdas
            self.Broadcast_mode('Left')    #Lo retransmite y gira a izquierdas
            self.Turn_left()

#Función en la que los kilobots siguen a un líder
def Follower_replicate(self):
    if(self.MemoryCount!=0):                    #Comprueba si ha llegado algún mensajes
        if(self.Father==False):                 #Si no tiene padre
            self.Father = self.Memory[0][2]     #Asigna al remitente del primer mensaje como padre
        for message in self.Memory:             #Revisa todos los mensajes que le han llegado
            if(message[2]==self.Father):        #Si el remitente del mensaje es el padre
                self.Broadcast_mode(message[0]) #Retransimte la orden de giro del padre
                if(message[0]=='Left'):         #Si la orden es girar a izquierdas
                    self.Turn_left()            #Gira a izquierdas
                else:                           #Sino quiere decir que gire a derechas
                    self.Turn_right()
    self.Listen_mode()                          #Pone en modo escucha
    self.Delete_memory()                        #Borra la memoria de mensajes

#Función para guiar el enjambre hacia la luz con realimentación
def Leader_guide(self):
    if(self.MemoryCount==0):                        #Comprueba si hay algún mensaje
        self.Move_to_light()                        #Si no hay mensajes mueve hacia la luz
        self.Broadcast_mode([True,self.nameKilobot]) #y retransmite al enjambre que se muevan
    else:                                           #Si hay algún mensaje
        order = self.Revise_messages()              #Revisa los mensajes para saber qué orden hacer
        if(order==True):                            #Si la orden=True quiere decir que el enjambre está listo
            self.Move_to_light()                    #como está listo mueve el enjambre hacia la luz
        else:                                       #Si no está listo el enjambre se detiene para esperar al
            self.Straight(0)                        #al kilobot rezagado
        self.Broadcast_mode([order,self.nameKilobot])#retransmite la orden o el kilobot que se tiene que mover
    self.Listen_mode()        #Lo pone en modo escucha
    self.Delete_memory()      #y borra la memoria

#Función para que el robot líder se mueva hacia la luz
def Move_to_light(self):
    readings = self.Light_sensor()[1]      #Lee el sensor
    angle = alpha(readings[0],readings[1]) #Calcula el ángulo
    if(angle<300 and angle>180):           #Comprueba si se encuentra en el rango de cambio de giro a derechas
        self.Turn_right()                  #Gira a derechas
        self.Motor='Right'                 #Guarda el actual sentido de giro
    elif(angle>60 and angle<180):          #Comprueba si se encuentra en el rango de cambio de giro a izquierda
        self.Turn_left()                   #Gira a derechas
        self.Motor='Left'                  #Guarda el actual sentido de giro
    elif(self.Motor=='Right'):   #Si no está en ningún rango de giro pero está girando a derechas
        self.Turn_right()        #Lo pone a girar a derecha(Esto se hace por si se ha parado para esperar a un Kilobot)
    else:                        #Sino gira a izquierdas
        self.Turn_left()
        self.Motor='Left'
```

```
    #Función para revisar los mensajes y decidir qué hacer dependiendo de las circunstancias
    def Revise_messages(self):
        if(self.Reading!=0):           #Revisa si hay algún Kilobot rezagado
            for message in self.Memory:#Si lo hay revisa en la memoria los mensajes de realimentación
                if(len(message[0])>2): #y si hay alguno que indique que el kilobot rezagado ya está cerca
                    if(message[0][3]==self.Reading and message[0][1]<self.__ComunicationScope-0.02):
                        self.Reading = 0 #Lo vuelve a 0
                        return True      #devuelve True y comienza a moverse
            return self.Reading  #si no devuelve el Kilobot rezagado para que se acerque
        else:
            for message in self.Memory:#Revisa la memoria para comprobar si algún Kilobot está rezagado respecto a su padre
                if(len(message[0])>2):      #Comprueba que es un mensaje de realimentación con la longitud de este
                    if(message[0][1]>self.__ComunicationScope-0.005): #Si algún kilobot esta más lejos que el
                        self.Reading = message[0][3]                  # rango de comunicación
                        return message[0][3] #Retransmite la orden de movimiento a ese kilobot devolviendo el nombre de este
            return True                      #Si no hay ningún Kilobot rezagado devuelve True para indicar que se mueva
        return self.Reading

    #Función en la que los kilobots siguen a un lider como en un caso real
    def Follow_leader(self):
        if(self.MemoryCount!=0):      #Comprueba si hay algún mensaje
            if(self.Father==False):   #Si no tiene padre se lo asigna como el remitente del primer mensaje
                self.Father = self.Memory[0][2]
            for message in self.Memory:  #Bucle que revisa los mensajes de la memoria
                if(message[2]==self.Father and len(message[0])<3):#Si algún mensaje es del padre y no es de realimentación
                    if(message[0][0]==True):      #Comprueba si la orden es True que indica que se mueva
                        self.Aproach(message[1])  #Se mueve en dirección del que le ordena la orden basándose en
                                                  #la distancia de este mensaje
                        self.Broadcast_mode(message[0])#Retransmite la orden para los hijos
                        self.Broadcast_mode([True,message[1],self.Father,self.nameKilobot]) #Y la realimentación para
                                                                                            #el padre
                    elif(message[0][0]==self.nameKilobot):#Si el mensaje contiene su nombre por orden quiere decir
                        self.Aproach(message[1])          #que está rezagado por lo que le indica que se acerque
                        self.Broadcast_mode([True,self.nameKilobot]) #Retransmite la orden para los hijos
                        self.Broadcast_mode([True,message[1],self.Father,self.nameKilobot])#Y la realimentación para
                                                                                           #el padre
                    else:                #Si le llega otra orden diferente se para (indica que la orden no es para él)
                        self.Straight(0) #entonces se para esperando a que se mueva el padre otra vez
                        self.Broadcast_mode(message[0])#Retransmite la orden para los hijos
                if(len(message[0])>3):     #Comprueba los mensajes de realimentación
                    if(message[0][2]==self.nameKilobot): #si algún mensaje indica que el padre del remitente es el mismo
                        self.Broadcast_mode(message[0])  #Lo retransmite (es el mensaje de un hijo)
            self.Listen_mode()    #Lo pone en modo escucha y borra memoria
            self.Delete_memory()

    #Función para acercarse a un objetivo remitente del valor introducido "distance"
    def Aproach(self,distance):
        if(distance<=0.036):          #Si la distancia es menor a 0.036m no se acerca mas
            self.Straight(0)          #Lo para
        else:
            if(self.Variable_timer(0.5)): #El muestreo se realiza cada 0.5s
                if(distance>=self.Reading):   #Si la distancia introducida es mayor que hace 0.5s realiza el cambio
                    if(self.Motor=='Left'):   #lo que hay que alternar de motor, si el motor que estaba
                        self.Turn_right()     #moviendose es el izquierdo cambia el sentido de giro a derecha
                        self.Motor='Right'    #Anota el actual sentido de giro
                    else:                     #En otro caso
                        self.Turn_left()      #Cambia el sentido de giro a izquierda
                        self.Motor='Left'     #y anota el sentido
                self.Reading = distance       #Actualiza la nueva lectura


#----------------------------------------------------------------------------------------------------------------------
#-------------------------------------------------Definición de todos los Kilobots-------------------------------------
#----------------------------------------------------------------------------------------------------------------------

#KilobotD_0
Kilobot0 = Kilobot('Kilobot0','RightLeg_0','LeftLeg_0','Proximity_sensor_0','MsgSensor_0','Ambient_light_sensor_0')

#KilobotD_1
Kilobot1 = Kilobot('Kilobot1','RightLeg_1','LeftLeg_1','Proximity_sensor_1','MsgSensor_1','Ambient_light_sensor_1')

#KilobotD_2
Kilobot2 = Kilobot('Kilobot2','RightLeg_2','LeftLeg_2','Proximity_sensor_2','MsgSensor_2','Ambient_light_sensor_2')

#KilobotD_3
Kilobot3 = Kilobot('Kilobot3','RightLeg_3','LeftLeg_3','Proximity_sensor_3','MsgSensor_3','Ambient_light_sensor_3')

#KilobotD_4
Kilobot4 = Kilobot('Kilobot4','RightLeg_4','LeftLeg_4','Proximity_sensor_4','MsgSensor_4','Ambient_light_sensor_4')
```

```
#KilobotD_4
Kilobot4 = Kilobot('Kilobot4','RightLeg_4','LeftLeg_4','Proximity_sensor_4','MsgSensor_4','Ambient_light_sensor_4')

#KilobotD_5
Kilobot5 = Kilobot('Kilobot5','RightLeg_5','LeftLeg_5','Proximity_sensor_5','MsgSensor_5','Ambient_light_sensor_5')

#KilobotD_6
Kilobot6 = Kilobot('Kilobot6','RightLeg_6','LeftLeg_6','Proximity_sensor_6','MsgSensor_6','Ambient_light_sensor_6')


#-------------------------------------------------------------------------------------------------------------------------
#-----------------------------------------------------Otras funciones-----------------------------------------------------
#-------------------------------------------------------------------------------------------------------------------------

#Función para dar el angulo de un vector(siempre valor positivo entre 0 y 360º)
def alpha(coordinate_x,coordinate_y):
    angle = (math.atan2(coordinate_y,coordinate_x))*(180/math.pi)
    if(angle<0):
        angle = angle+360
    return angle

#Función que representa cuando se esta enviando un mensaje por el area de trabajo
def broadcast_message(Kilobot,message):
    _,ProximitySensorHandle = Kilobot.ProximitySensorHandle
    for i in range(7):                    #El bucle 'for' comprueba todos los Kilobots para ver si hay alguno cerca
        kilobot = eval('Kilobot'+str(i))  #almacena el nombre del Kilobot que va a evaluar
        msgHandle = eval('Kilobot'+str(i)+'.MsgSensorHandle')#el handle del recibidor de infrarrojos
        listenerEnabled = eval('Kilobot'+str(i)+'.EnableReceived')# y la bandera que indica si esta escuchando
        readings = client.simxCheckProximitySensor(ProximitySensorHandle,msgHandle[1],client.simxServiceCall())
        #Lee si esta cerca y almacena las lecturas
        if(readings[1]==1 and listenerEnabled==1): #Comprueba si esta en rango y si está en escucha el kilobot
            distance = round(readings[2],3)        #almacena la distancia redondeada al tercer decimal
            message_on_buffer(kilobot,message,distance,Kilobot.nameKilobot) #y llama a la función que representa que le
                                                                            #ha llegado al buffer
    return True

#Función para que representa cuando a un kilobot le llega un mensaje por infrarrojos al buffer
def message_on_buffer(Kilobot,message,distance,sender):
    Kilobot.Store_data([message,distance,sender])   #lo almacena en memoria
    return True

#Función para poner en False en el estado del Kilobot y reiniciar el SwarmState de todos los kilobots
def reset_SwarmState():
    for i in range(7):                    #El bucle 'for' para recorrer la lista de kilobots
        kilobot = eval('Kilobot'+str(i))  #almacena el nombre del Kilobot que va a evaluar
        clean_SwarmState(kilobot)         #y llama a la funcion que limpia el Kilobot
    return True

#Función que limpiar el kilobot evaluado
def clean_SwarmState(Kilobot):
    Kilobot.State = False                                   #Inicializa el estado
    Kilobot.SwarmState = {Kilobot.nameKilobot:Kilobot.State}    #y la lista de todos los kilobots
    return True
```

Para la gestión de la ejecución de los diferentes métodos del código se han realizado mediante bloques separados:

Inicio/Parada simulación:

```
#-------------------------------------------------------------------------------------------------------------------------
#------------------Inicialización de los Kilobots y llamada que hace que inicie Coppelia--------------------------------
#-------------------------------------------------------------------------------------------------------------------------
client.simxStartSimulation(client.simxServiceCall())
Kilobot0.Initialization(False)
Kilobot1.Initialization(True)
Kilobot2.Initialization(True)
Kilobot3.Initialization(True)
Kilobot4.Initialization(True)
Kilobot5.Initialization(True)
Kilobot6.Initialization(True)
```

```
#-------------------------------------------------------------------------------------------------------------------------
#--------------------------Detiene la simulación en Coppelia----------------------------------------------------------------
#-------------------------------------------------------------------------------------------------------------------------
client.simxStopSimulation(client.simxDefaultPublisher())
```

Método de réplica de movimientos:

```
#-------------------------------------------------------------------------------------------------------------------
#-----------------------------Enjambre hacia la luz método de replicación de ordenes ---------------------------------
#-------------------------------------------------------------------------------------------------------------------
while(Kilobot0.Light_sensor()[0]>0.8):                    #Comprueba a la distancia que se encuentra
    Kilobot0.Leader_replicate()                           #Ejecuta el método de guiar hacia la luz el Kilobot 0
    number=1                                              #Contador de ID del Kilobot
    while(number<=6):                                     #Bucle que recorre todos los kilobots
        eval('Kilobot'+str(number)+'.Follower_replicate()') #Activa al Kilobot"number" el método para seguir al líder
        number = number+1                                 #Incrementea en uno el contador
```

Método realimentado con la distancia:

```
#-------------------------------------------------------------------------------------------------------------------
#-----------------------------Enjambre hacia la luz método basado en la distancia -------------------------------------
#-------------------------------------------------------------------------------------------------------------------
while(Kilobot0.Light_sensor()[0]>0.8):                    #Comprueba a la distancia que se encuentra
    Kilobot0.Leader_guide()                               #Ejecuta el método de guiar hacia la luz el Kilobot 07
    number=1                                              #Contador de ID del Kilobot
    while(number<=6):                                     #Bucle que recorre todos los kilobots
        eval('Kilobot'+str(number)+'.Follow_leader()')    #Activa al Kilobot"number" el método para seguir al lider
        number = number+1                                 #Incrementea en uno el contador
```

Algoritmo “distributed and resilient localization”:

```
#-------------------------------------------------------------------------------------------------------------------
#----------------------Inicia la simulación del algoritmo de localización de un enjambre ----------------------------
#-------------------------------------------------------------------------------------------------------------------
while(not(Kilobot0.Complete_algorithm(4))):
    number=1
    while(number<=6):
        eval('Kilobot'+str(number)+'.Complete_algorithm(7)')
        number = number+1

#Muestra en pantalla los resultados de todos los Kilobots
print('\n ')
print('Stage IV(Coordenadas finales): ')
print('\n ')
for n in range(7):
    print('kilobot'+str(n)+':')
    print(eval('Kilobot'+str(n)+'.Coordinates'))
```

Método réplica de movimientos:

Script Kilobot líder:

```
#include "kilolib.h"
//-------------------------------------------------------------------------------------------------------------------
//Código en el que el Kilobot va hacia la luz y retransmite al enjambre los movimientos que hace para que le puedan seguir
//-------------------------------------------------------------------------------------------------------------------

//Definición de constantes y variables a utilizar
//Aquí se definen las palabras Forward,left y right como l,r respectivamente
//esto sirve para que a la hora de hacer el código sea menos lioso, y se le dan valores
#define Left 'l'                  //de un solo caracter para que cuando se envíe un mensaje no ocupen mucho espacio
#define Right 'r'                 //ya que enviar una cadena de caracteres ocupa mucho espacio en órdenes.
message_t MessageToSend;          //Variable donde se almacena el mensaje a enviar
char CurrentMotion = Left;        //Variable donde se almacena el movimiento actual de kilobot
uint16_t CurrentLight = 0;        //Variable donde se almacena el valor de la luminosidad
uint8_t MessageSentFlag = 0;      //Bandera para señalar que se ha enviado un mensaje con éxito

//-------------------------------------------------------------------------------------------------------------------
//----------------------------------------Funciones que se van a utilizar--------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//Función necesaria para que se envíe el mensaje, esta función crea un puntero que apunta al mensaje a enviar
message_t *Message_Pointer(){
    return &MessageToSend;  //devuelve el mensaje a enviar
}

//Función para comprobar que un mensaje se ha enviado con éxito
void Tx_Message_Success(){
    MessageSentFlag = 1;
}

//Función para retransmitir el movimiento que se va a hacer
void Broadcast_Movement(char motion){
    MessageToSend.type = NORMAL;                    //Inicialización del mensaje a enviar
    MessageToSend.data[1] = motion;                 //y el motor que ha activado
    MessageToSend.crc = message_crc(&MessageToSend); //La función message_crc se tiene que ejecutar y guardar en el mensaje
}

//Función que cambia los motores para moverse hacia un lado u otro dependiendo de la entrada "NewMotion"
void Set_Motion(char NewMotion){
    if (CurrentMotion != NewMotion){   //Solo se mueve si cambia de movimiento
        CurrentMotion = NewMotion;     //Si es diferente la actualiza
        Broadcast_Movement(CurrentMotion);  //retransmite al enjambre el movimiento
        MessageSentFlag=0; //Reinicia la bandera
        delay(600);        //Espera 600 ms para que le dé tiempo a los demás kilobots a procesar el mensaje enviado
        spinup_motors();//Función para que el Kilobot se mueva durante un breve tiempo a máxima potencia para vencer el rozamiento del suelo
        if (CurrentMotion == Left){  //Si el movimiento es Left gira a la izquierda
            set_motors(kilo_turn_left, 0);//para el motor derecho y lo hace girar con el valor almacenado en la calibración
        }
        else if (CurrentMotion == Right){  //Si el movimiento es Right gira a la derecha
            set_motors(0, kilo_turn_right);//Análogo a la anterior
        }

    }
}

//Función para recoger muestras de luz y hacer la media ya que con una sola muestra no es suficiente debido al ruido
void Sample_Light(){
    uint8_t numberSamples = 0;              //Contador del número de muestras(variable de 1byte)
    uint16_t sum = 0;                       //Aquí se almacena el total de muestras(variable de 4bytes por nº grandes)
    while (numberSamples < 60){             //Coge 60 muestras que son el máximo número que se puede representar en la variable sum es
                                            //de 2^16=65536, como el máximo valor de las muestras son 1020 ->65536/1020=64
        int16_t sample = get_ambientlight();//Llama la función que lee el sensor de luz y lo almacena en "sample"
        if (sample != -1){                  //Si es una muestra diferente a -1(lectura incorrecta) la suma
            sum = sum + sample;             //Almacena la muestra en el total "sum"
            numberSamples++;                //Sube en uno el número de muestras
        }
    }
    CurrentLight = sum / numberSamples;           //Calcula la media y lo almacena en la variable "CurrentLight"
}

//-------------------------------------------------------------------------------------------------------------------
//----------------------------------------Funciones Principales------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//Función de inicialización del código
void setup(){
    spinup_motors();//Esta función hace que el Kilobot se mueva durante un breve tiempo a maxima potencia para vencer el rozamiento del suelo
    set_motors(kilo_turn_left, 0);//el rozamiento del suelo, después se pone a girar a la izquierda para asegurar movimiento
}

//Función principal donde se pone el código que se va a ejecutar
void loop(){
    Sample_Light();                 //Empieza calculando la luz y almacenando en la variable "CurrentLight"
    if (CurrentLight < 300){        //Si es más bajo que el umbral menor quiere decir que no está bien orientado
        Set_Motion(Right);          //y lo hace girar a la derecha
    }
    else if (CurrentLight > 600){   //Si supera el umbral superior lo gira a la izquierda
        Set_Motion(Left);
    }
}
```

```
 //Esta es la función main donde se ejecuta la inicialización del Kilobot y la función que llama al código que se ejecutara
int main(){
    kilo_init();                                    //Función de inicialización del hardware del Kilobot
    kilo_message_tx = Message_Pointer;              //registra la función de llamada
    kilo_message_tx_success = Tx_Message_Success;   //registra la función de éxito de llamada
    kilo_start(setup, loop);                        //Función para iniciar el bucle principal y la inicialización
    return 0;
}
```

Script Kilobot Seguidor:

```
#include "kilolib.h"
//-------------------------------------------------------------------------------------------------------------------
//Código en el que el Kilobot sigue las ordenes que le llegan de un Kilobot lider definido de acuerdo al algoritmo "wave"
//-------------------------------------------------------------------------------------------------------------------


//Definición de constantes y variables a utilizar
//Aquí se definen las palabras left y right como 'l' y 'r' respectivamente
//esto sirve para que a la hora de hacer el código sea menos lioso, y se le dan valores
#define Left 'l'                //de un solo carácter para que cuando se envíe un mensaje no ocupen mucho espacio
#define Right 'r'               //ya que enviar una cadena de caracteres ocupa mucho espacio en ordenes.
char CurrentMotion = 'l';       //Variable donde se almacena el movimiento actual de kilobot
message_t MessageToSend;        //Variable donde se almacena el mensaje a enviar
uint8_t RecivedMessage[2];      //Variable donde se almacena el mensaje recibido
int8_t Father = -1;             //Aquí se almacena la ID del Kilobot padre
uint8_t NewMessageFlag = 0;     //Bandera que indica si ha llegado algún mensaje
uint8_t MessageSentFlag = 0;    //Bandera para señalar que se ha enviado un mensaje con éxito

//-------------------------------------------------------------------------------------------------------------------
//-----------------------------------Funciones que se van a utilizar-------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//Función necesaria para que se envíe el mensaje, esta función crea un puntero que apunta al mensaje a enviar
message_t *Message_Pointer(){
    return &MessageToSend;  //devuelve el mensaje a enviar
}

//Función para comprobar que un mensaje se ha enviado con éxito
void Tx_Message_Success(){
    MessageSentFlag = 1;
}

//Función para retransmitir el movimiento que se va a hacer
void Broadcast_Movement(char motion){
    MessageToSend.type = NORMAL;                    //Inicialización del mensaje a enviar
    MessageToSend.data[0] = kilo_uid;               //Se almacena su ID en el mensaje a enviar
    MessageToSend.data[1] = motion;                 //y el motor que ha activado
    MessageToSend.crc = message_crc(&MessageToSend); //La función message_crc se tiene que ejecutar y guardar en el mensaje
}

//Función para almacenar el mensaje que le ha llegado en la variable "RecivedMessage"
void Message_Store(message_t *message, distance_measurement_t *distance_measurement){
    NewMessageFlag = 1;                     //Señaliza que hay un nuevo mensaje
    RecivedMessage[0] = message->data[0];  //Lo almacena en la variable "RecivedMessage"(Aquí la ID del Kilobot)
    RecivedMessage[1] = message->data[1];  //Aquí almacena el movimiento que está haciendo el remitente del mensaje
}

//Función que cambia los motores para moverse hacia un lado u otro dependiendo de la entrada "NewMotion"
void Set_Motion(char NewMotion){
    if (CurrentMotion != NewMotion){   //Solo se mueve si cambia de movimiento
        CurrentMotion = NewMotion;     //Si es diferente la actualiza
        spinup_motors();//Función para que el Kilobot se mueva durante un breve tiempo a máxima potencia para vencer el rozamiento del suelo
        if (CurrentMotion == Left){  //Si el movimiento es Left gira a la izquierda
            set_motors(kilo_turn_left, 0);//para el motor derecho y lo hace girar con el valor almacenado en la calibración
        }
        else if (CurrentMotion == Right){  //Si el movimiento es Right gira a la derecha
            set_motors(0, kilo_turn_right);//Análogo a la anterior
        }
        Broadcast_Movement(CurrentMotion);//Una vez actualizado el movimiento lo retransmite al enjambre
        MessageSentFlag=0;
    }
}


//-------------------------------------------------------------------------------------------------------------------
//-----------------------------------Funciones Principales-----------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//Función de inicialización del código
void setup(){
    spinup_motors();//Función para que el Kilobot se mueva durante un breve tiempo a maxima potencia para vencer el rozamiento del suelo
    set_motors(kilo_turn_left, 0);//el rozamiento del suelo, después se pone a girar a la izquierda para asegurarse movimiento
}

//Función principal donde se pone el código que se va a ejecutar
void loop(){
    if(NewMessageFlag==1){                  //Si hay mensaje comprobando el estado de la bandera
        NewMessageFlag = 0;                 //Reinicia la bandera
        if(Father==-1){                     //Si no tiene padre
            Father = RecivedMessage[0];     //lo almacena al primero que le envíe el mensaje
            }
        if(RecivedMessage[0]==Father){      //Si el mensaje es del padre
            Set_Motion(RecivedMessage[1]); //Lo lee y modifica el movimiento de acuerdo con el asignado
            }
        }
}
```

```
//Esta es la función main donde se ejecuta la inicialización del Kilobot y la función que llama al código que se ejecutara
int main(){
    kilo_init();                                //Función de inicialización del hardware del Kilobot
    kilo_message_tx = Message_Pointer;          //registra la función de llamada
    kilo_message_tx_success = Tx_Message_Success;    //registra la función de éxito de llamada
    kilo_message_rx = Message_Store;            //registra la función de llegada de un mensaje
    kilo_start(setup, loop);                    //Función para iniciar el bucle principal y la inicialización
    return 0;
}
```

Algoritmo “ditributed and resilient localization”:

Script Kilobot ancla:

```
#include "kilolib.h"
//-------------------------------------------------------------------------------------------------------------------
//Código con el que los kilobots asignados como anclas retransmiten su posición para que puedan usarlo los nodos
//-------------------------------------------------------------------------------------------------------------------

//Definición de variables a utilizar
message_t MessageToSend;   //Variable donde se almacena el mensaje a enviar
uint8_t Coordinates[2];    //Aquí almacena la posición que tiene el ancla que será la que compartirá
MessageSentFlag = 1;

//-------------------------------------------------------------------------------------------------------------------
//---------------------------------------Funciones que se van a utilizar---------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//Función necesaria para que se envíe el mensaje, esta función crea un puntero que apunta al mensaje a enviar
message_t *Message_Pointer(){
    return &MessageToSend;  //devuelve el mensaje a enviar
}

//Función para comprobar que un mensaje se ha enviado con éxito
void Tx_Message_Success(){
    MessageSentFlag = 1;
}

//Función para retransmitir las coordenadas
void Broadcast_Mode(uint8_t data[4]){
    MessageToSend.type = NORMAL;                    //Inicialización del mensaje a enviar
    uint8_t count=0;                                //Este contador se utiliza en el bucle while
    while(count<4){                                 //Con este bucle recorre el mensaje a enviar para almacenarlo
        MessageToSend.data[count] = data[count];    //Lo almacena en la variable "MessageToSend"
        count++;
    }                                               //La función message_crc se tiene que ejecutar y guardar en
    MessageToSend.crc = message_crc(&MessageToSend); //el mensaje a enviar para que el receptor no lo descarte
}

//-------------------------------------------------------------------------------------------------------------------
//---------------------------------------Funciones Principales-------------------------------------------------------
//-------------------------------------------------------------------------------------------------------------------

//Función de inicialización del código
void setup(){               //Inicializa las coordenadas de los anclas dependiendo de su número de ID
    if(kilo_uid==0){
        Coordinates[0] = 0; //Si es el Kilobot 0 se pone en la posición (0,0)mm
        Coordinates[1] = 0;
    }
    else if(kilo_uid==1){
        Coordinates[0] = 70; //Si es el Kilobot 1 se pone en la posición (70,0)mm
        Coordinates[1] = 0;
    }
    else if(kilo_uid==2){
        Coordinates[0] = 0; //Si es el Kilobot 2 se pone en la posición (0,70)mm
        Coordinates[1] = 70;
    }
}

//Función principal donde se pone el código que se va a ejecutar
void loop(){                            //Esta estructura de mensajes es la que se hace en el algoritmo
    uint8_t v[4] = {kilo_uid,Coordinates[0],Coordinates[1],0}; //Almacena su ID y sus coordenadas y la distancia Lr en la variable v
    Broadcast_Mode(v);                                         //Llama a la función que lo retransmite
    MessageSentFlag = 0;
    delay(rand_soft()*2);       //Espera un tiempo aleatorio hasta enviar un nuevo mensaje (se hace para que no se solapen)
}

//Esta es la función main donde se ejecuta la inicialización del Kilobot y la función que llama al codigo que se ejecutara
int main(){
    kilo_init();                                //Función de inicialización del hardware del Kilobot
    kilo_message_tx = Message_Pointer;          //registra la función de llamada
    kilo_message_tx_success = Tx_Message_Success;  //registra la función de éxito de llamada
    kilo_start(setup, loop);                    //Función para iniciar el bucle principal y la inicialización
    return 0;
}
```

Script Kilobot nodo:

```
#include "kilolib.h"
#define DEBUG
#include "debug.h"
//-------------------------------------------------------------------------------------------------------------------
//Este código se utilizará para que un Kilobot calcule su posición con el algoritmo Min_Max y lo retransmita por el cable
//serial, será necesaria la librería debug para poder usar el cable serial
//-------------------------------------------------------------------------------------------------------------------

//Definición de variables a utilizar
uint8_t RecivedMessage[3];   //Variable donde se almacena el mensaje recibido
uint8_t NewMessageFlag = 0;  //Bandera que indica si ha llegado algún mensaje
uint8_t Distance = 0;        //Aquí se almacena la distancia a la que se encuentra el remitente en mm
uint8_t Anchors[3][4];       //En esta matriz se almacena la información de los mensajes que le llegan
uint8_t Coordinates[2];      //Variable donde se guarda su propia posición
uint8_t FlagMin_Max = 0;     //Esta bandera sirve para indicar que se ha calculado ya la posición y no repita el proceso

//Función para almacenar el mensaje que le ha llegado en la variable "RecivedMessage"
void Message_Store(message_t *message, distance_measurement_t *distance_measurement){
    NewMessageFlag = 1;                   //Señaliza que hay un nuevo mensaje
    RecivedMessage[0] = message->data[0];  //Lo almacena en la variable "RecivedMessage"(Aqui la ID del Ancla)
    RecivedMessage[1] = message->data[1];  //Aquí la posición x en mm
    RecivedMessage[2] = message->data[2];  //Aquí la posición y en mm
    Distance = estimate_distance(distance_measurement); // Aquí la distancia a la que se encuentra el remitente
}

//Función que devuelve el máximo de un vector de dimensión 3
int8_t Max(int8_t vector[3]){
    int8_t max = vector[0]; //Anota el primer elemento como el máximo
    if(vector[1]>max){      //Comprueba si el segundo elemento es más grande que la segunda
        max = vector[1];    //Si lo es la actualiza como el máximo
    }
    else if(vector[2]>max){ //De forma análoga se hace con el tercer elemento
        max = vector[2];
    }
    return max;             //Devuelve el máximo
}

//Función que devuelve el mínimo de un vector de dimensión 3
int8_t Min(int8_t vector[3]){
    int8_t min = vector[0];  //Sigue el mismo procedimiento que con la función anterior
    if(vector[1]<min){       //pero comprobando el mínimo
        min = vector[1];
    }
    else if(vector[2]<min){
        min = vector[2];
    }
    return min;
}

//Función que calcula la posición con los datos de los anclas usando el método Max_min
void Min_Max(){
    //Aquí almacena los valores de los cuadros delimitadores Bi de un nodo
    int8_t Bir1x[3] = {Anchors[0][1]-Anchors[0][3],Anchors[1][1]-Anchors[1][3],Anchors[2][1]-Anchors[2][3]};
    int8_t Bir1y[3] = {Anchors[0][2]-Anchors[0][3],Anchors[1][2]-Anchors[1][3],Anchors[2][2]-Anchors[2][3]};
    int8_t Bir2x[3] = {Anchors[0][1]+Anchors[0][3],Anchors[1][1]+Anchors[1][3],Anchors[2][1]+Anchors[2][3]};
    int8_t Bir2y[3] = {Anchors[0][2]+Anchors[0][3],Anchors[1][2]+Anchors[1][3],Anchors[2][2]+Anchors[2][3]};
    int8_t Si[2][2];       //Aquí se almacena la intersección de los cuadros
    Si[0][0] = Max(Bir1x); //Calculando su máximo
    Si[0][1] = Max(Bir1y);
    Si[1][0] = Min(Bir2x); //y su mínimo
    Si[1][1] = Min(Bir2y);
    Coordinates[0] = (Si[0][0]+Si[1][0])/2; //calcula el centro de dicho cuadrado que será la posición estimada
    Coordinates[1] = (Si[0][1]+Si[1][1])/2;
}

//-------------------------------------------------------------------------------------------------------------------

//Función de inicialización del código
void setup(){
}

//Función principal donde se pone el código que se va a ejecutar
void loop(){
    //Stage I del algoritmo
    if(Anchors[0][3]==0 || Anchors[1][3]==0 || Anchors[2][3]==0){//Si no tiene los datos de algun ancla no para de repetir el proceso
        if(NewMessageFlag==1){                                   //Comprueba si hay un nuevo mensaje
            NewMessageFlag = 0;                                  //Reinicia la bandera
            Anchors[RecivedMessage[0]][0]=RecivedMessage[0]; //Almacena la ID del remitente
            Anchors[RecivedMessage[0]][1]=RecivedMessage[1]; //Su posición en x
            Anchors[RecivedMessage[0]][2]=RecivedMessage[2]; //Su posición en y
            Anchors[RecivedMessage[0]][3]=Distance;          //Y la distancia recorrida por el mensaje
        }
    }
    //Stage II del algoritmo
    else if(FlagMin_Max==0){                    //Una vez terminada la fase de recopilación de datos de los anclas
        Min_Max();                              //Se inicia el metodo Min_Max para la obtención de las coordenadas
        printf ( "Coordenadas nodo: (%d,%d) " , Coordinates[0],Coordinates[1]); //Envía por el serial las coordenadas para que se puedan ver
        FlagMin_Max = 1;                //Pone la bandera en 1 para que pare de realizar el código indicando que ha terminado
    }
}
```

```
//Esta es la función main donde se ejecuta la inicialización del Kilobot y la funcion que llama al codigo que se ejecutara
int main(){
    kilo_init();                    //Función de inicialización del hardware del Kilobot
    kilo_message_rx = Message_Store;  //registra la función de llegada de un mensaje
    debug_init();//Hay que llamar a la función "debug_init" para poder usar printf y que se pueda ver en la consola de KiloGUI
    kilo_start(setup, loop);          //Función para inciar el bucle principal y la inicialización
    return 0;
}
```

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

La robótica móvil está teniendo un papel importante en la actualidad tanto a nivel industrial como social. Más concretamente, una de las áreas que está emergiendo actualmente es la robótica de enjambre o control de flotas de robots. Este Trabajo de Fin de Grado (TFG) trata sobre la implementación de algoritmos de tecnología de enjambre dirigidos al control de flotas de minirobots, concretamente Kilobots, tanto a nivel de simulación como de robots reales.

En este TFG se realiza un estudio bibliográfico sobre algoritmos que usan tecnología de enjambre, se implementa el algoritmo "distributed and resilient localization" para una localización propia en base a los robots adyacentes, y el algoritmo "wave" para una navegación conjunta de una flota de robots. La simulación se lleva a cabo con el software CoppeliaSim Edu haciendo uso de robots Kilobots y a partir de esta simulación se hace un ensayo con Kilobots reales de los algoritmos mencionados.

La programación de los Kilobots en CoppeliaSim se desarrolla en lenguaje Python puesto que ofre-ce una serie de herramientas para hacer una programación más sencilla. Para el uso de Python en CoppeliaSim se hace uso de Anaconda, un software para administrar las diferentes librerías utiliza-das en Python, que además ofrece varios editores de programación, como por ejemplo Jupyter.

Palabras clave: Tecnología de enjambre, swarm robotic, control de flotas, Kilobots, CoppeliaSim, sistemas multi robot.

Nowadays mobile robotics is having an important role, at industrial and social level. More specifical-ly, one of the fields that is currently emerging is swarm robotics. The main objective of this diploma thesis is the development of a swarm robotics algorithm to control minirobots fleets, specifically Kilobots, both at the simulation and real levels.

In this diploma thesis a bibliographic study about swarm robotics algorithms has been developed, in addition, the algorithm "distributed and resilient localization" for a self-location based on the adja-cent robots and the "wave" algorithm is implemented for a joint navigation of a robot's swarm. The simulation is carried out with CoppeliaSim Edu software making use of Kilobots and, afterwards, real tests have been performed using real Kilobots.

The programming of Kilobots have been developed in CoppeliaSim with Python since it offers sev-eral tools for a simple programming. To use Python in CoppeliaSim, Anaconda software has been used. It is a software to manage several libraries from Python besides offering several programming environments like for example Jupyter.

Keywords: Swarm technology, swarm robotics, fleet control, Kilobots, CoppeliaSim, multirobot systems.