UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"SIMULACIÓN DE ROBOT MÓVIL YOUBOT KUKA PARA APLICACIÓN DE REPARTO"

TRABAJO FIN DE GRADO

Junio-2024

AUTOR: Pau Vañó Agulló

DIRECTOR/ES: Arturo Gil Aparicio

Luis Miguel Jiménez García

Índice

1. INTRODUCCIÓN

1.1. ANTECEDENTES

Desde mediados del siglo XX, la robótica ha experimentado una evolución significativa revolucionando la manera de abordar los desafíos industriales. Estos avances han permitido una mejora de la eficiencia, precisión y seguridad en diversos sectores, especialmente en el ámbito industrial. Precisamente, la creciente demanda de aplicaciones flexibles y adaptables en el sector industrial ha obligado a impulsar la investigación y el desarrollo de sistemas robóticos móviles. Por ello, desde hace varias décadas, la robótica móvil se ha convertido en uno de los principales campos de estudio de la robótica moderna.

El punto de inflexión, en robótica móvil, apareció con el desarrollo de diversos algoritmos que permitieron transferir al robot cierta inteligencia y autonomía para enfrentarse a entornos complejos y dinámicos, como: la capacidad de realizar planificación de la trayectoria, de localizarse en un entorno de trabajo o la capacidad de percibir cambios en el entorno. Desde entonces, la robótica móvil ha seguido evolucionando durante el siglo XXI y en gran medida por la aparición de la inteligencia artificial (IA).

El presente trabajo final de grado se centra en el desarrollo de los algoritmos necesarios para dotar a el robot de la ***Figura 1-1*** de la autonomía suficiente para poder realizar un trabajo de reparto en un escenario concreto en un entorno de simulación, con el objetivo de presentar una de las posibles aplicaciones de esta rama de la robótica móvil.

Figura 1-1. Kuka Youbot

1.2. OBJETIVOS DEL PROYECTO

El objetivo general de este trabajo es desarrollar un software de control para el robot móvil Kuka Youbot, capacitándolo para realizar tareas de reparto de manera autónoma en un entorno predefinido. Para alcanzar este objetivo principal, se han identificado distintos subobjetivos que se dividen en dos categorías: objetivos de aprendizaje y objetivos técnicos.

1.2.1. OBJETIVOS DE APRENDIZAJE

Los objetivos de aprendizaje se centran en la comprensión teórica de los elementos clave del trabajo. Estos objetivos incluyen:

- **Investigar las capacidades limitaciones del robot Kuka Youbot**: Es crucial comprender las capacidades de movilidad y manipulación del robot para hacer un diseño efectivo del algoritmo de control. Esto garantizará que el robot sea capaz de adaptarse de manera eficiente al entorno.

- **Estudiar y comprender el entorno de simulación CoppeliSim y su API para Python:** Entender el entorno de simulación y su integración con Python es esencial para poder realizar pruebas y desarrollar el algoritmo.

- **Familiarizarse con los fundamentos del algoritmo de localización Monte Carlo Localization**: La capacidad del robot para evitar obstáculos es fundamental. Por ello, tener un conocimiento sólido del funcionamiento del algoritmo de localización es importante para desarrollar un algoritmo competente.

- **Comprender el funcionamiento del algoritmo de planificación de rutas A***: Implementar un algoritmo de planificación de rutas es esencial para el éxito de la navegación autónoma del robot. Comprender el algoritmo A* permitirá hacer una implementación efectiva y capaz de adaptarse a diferentes escenarios de reparto.

1.2.2. OBJETIVOS TÉCNICOS

Los objetivos técnicos se centran en la implementación y verificación de los conceptos teóricos previamente estudiados. Estos objetivos incluyen:

- **Definir el escenario en CoppeliaSim y establecer la conexión con el entorno de simulación**: Después de comprender la API y el entorno de simulación, se definirá el escenario de trabajo y se asegurará la conexión con el robot.

- **Dotar al robot de un sistema de percepción y desarrollar el algoritmo de localización**: Se dotará al robot de los sensores necesarios y se desarrollará el algoritmo de localización.

- **Instaurar el algoritmo de planificación de rutas A***: Para dotar al robot de autonomía, se implementará el algoritmo que determinará las rutas óptimas para alcanzar los objetivo.

- **Integrar un sistema de manipulación para aplicación 'Pick and Place':** El robot necesitará manipular objetos con su brazo robótico, lo que se logrará mediante la integración de un sistema de manipulación.

- **Realizar diferentes pruebas en el entorno simulado para evaluar la confiabilidad del software de control:** Se someterá el algoritmo a pruebas exhaustivas para mejorar su eficiencia y fiabilidad.

Cada uno de los objetivos específicos mencionados anteriormente están interconectados de tal forma que cada uno de ellos contribuye al logro de los demás. Esta interrelación entre objetivos será imprescindible para el éxito del proyecto y la consecución del objetivo general.

2. ESTADO DEL ARTE

La robótica, una disciplina arraigada en la historia de la humanidad, ha evolucionado junto con nosotros, integrándose en nuestra vida diaria y experimentando un rápido progreso en diversos ámbitos de aplicación. Desde robots manipuladores y de entretenimiento hasta robots de servicio y móviles, la robótica ha diversificado su alcance para adaptarse a nuestras necesidades cambiantes.

En particular, la robótica móvil ha experimentado un notable avance en las últimas décadas, impulsada por la creciente demanda de soluciones autónomas para una amplia gama de aplicaciones en entornos cerrados. Este campo multidisciplinario fusiona conocimientos de ingeniería mecánica, informática, inteligencia artificial y percepción sensorial para desarrollar sistemas robóticos capaces de moverse, interactuar y operar de forma autónoma en espacios interiores complejos y dinámicos.

2.1. EVOLUCIÓN ROBÓTICA MÓVIL EN INDUSTRIA

Desde los primeros días de la robótica móvil, el desarrollo de algoritmos de navegación y control ha sido uno de los principales desafíos. Un hito temprano en esta búsqueda fue la creación de la "Máquina Speculatrix" entre 1948 y 1949 por W. Walter Grey. Este robot pionero demostró una autonomía rudimentaria al realizar tres comportamientos clave: superar obstáculos de manera tosca, regresar a su posición inicial y recargar sus baterías antes de agotarse.

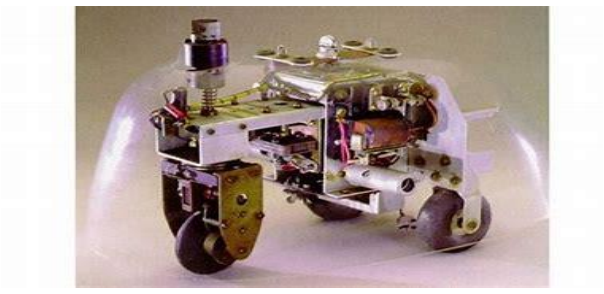


Figura 2-1. Máquina Speculatrix de W. Walter Grey

No obstante, un punto de inflexión verdadero en la historia de la robótica móvil fue la creación del robot SHAKEY en la década de 1970 en la Universidad de Stanford.

SHAKEY fue el primero en estar equipado con inteligencia artificial para controlar sus movimientos. Este robot estaba equipado con varios sensores, como sensores de contacto tipo bigotes de gato, una cámara de televisión y una antena para comunicación por radio. Todo esto era controlado por una computadora central y una unidad de control de cámara ubicada en el centro del robot.

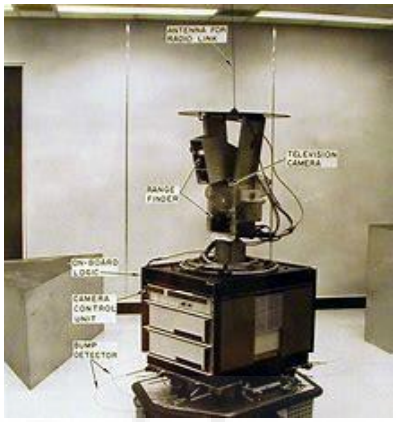


Figura 2-2. Robot SHAKEY

A lo largo de los años, los avances en el desarrollo de algoritmos de navegación y control, junto con la miniaturización de sensores y actuadores, así como el desarrollo de la inteligencia artificial, han impulsado la creación de robots móviles cada vez más sofisticados y capaces. Estos avances han permitido una variedad de aplicaciones innovadoras, desde robots de exploración en entornos hostiles hasta robots de servicio en el hogar y en la industria.

En la industria, este avance se ha vuelto especialmente significativo. La automatización ha revolucionado los sistemas de transporte interno, pasando de los tradicionales vehículos de guiado automático (AGV) a los robots móviles autónomos (AMR).

Los AGV, o Vehículos Guiados Automáticamente, son sistemas robóticos diseñados originalmente para el transporte de materiales y productos en entornos industriales. Surgieron en la década de 1950 como vehículos simples que seguían rutas predefinidas mediante sistemas de guía física como cables magnéticos o cintas reflectantes en el suelo. A medida que avanzaba la tecnología, los AGVs se volvieron más sofisticados y

versátiles, integrando tecnologías avanzadas de navegación como la navegación láser y la navegación por visión.

Figura 2-3. Robot AGV

Con el continuo desarrollo de la robótica móvil, surgieron los AMR, o Robots Móviles Autónomos, como una evolución natural de los AGVs. Estos sistemas están diseñados para moverse de manera autónoma en entornos dinámicos y variables, sin necesidad de guía física específica. Equipados con una variedad de sensores, como cámaras y lidars, y utilizando algoritmos avanzados de navegación, los AMR pueden adaptarse fácilmente a cambios en su entorno y en los procesos de trabajo. Esta flexibilidad los hace ideales para entornos industriales y logísticos en constante cambio. Lo que los convierte en una solución versátil para una variedad de industrias, desde la manufactura hasta la distribución.

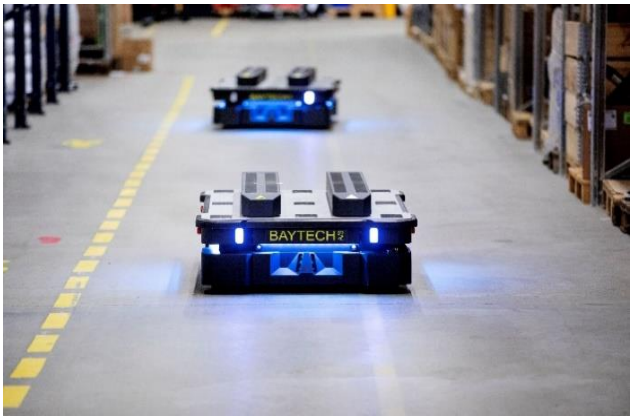


Figura 2-4. Robot AMR

Además, uno de los últimos avances destacado en la industria es la robótica híbrida, donde los robots no solo se limitan a tareas de logística o transporte, sino que también son capaces de mover y manipular objetos de forma autónoma. Esto amplía significativamente las posibilidades de aplicaciones industriales en las que estos robots pueden participar, brindando una mayor flexibilidad y eficiencia en los procesos de trabajo.

Figura 2-5. Robot Híbrido

2.2. TIPOS DE SENSORES

En el desarrollo de robots móviles, uno de los aspectos cruciales ha sido el avance en los sensores que proporcionan información vital para su funcionamiento y toma de decisiones. Estos sensores, fundamentales para la robótica móvil, han evolucionado significativamente a lo largo del tiempo. Aquí presentamos los sensores más comunes utilizados en este campo:

2.2.1. SENSORES DE DISTANCIA

2.2.1.1. SENSOR ULTRASÓNICO

El sensor ultrasónico se basa en la medición del tiempo que tardan las ondas de sonido de alta frecuencia en viajar desde el sensor hasta un objeto y regresar. Este tiempo de vuelo se utiliza para calcular la distancia entre el sensor y el objeto. Este dispositivo se distingue por diversas características, como su rango de detección, ángulo de detección y precisión.

Hoy en día, este dispositivo electrónico se utiliza ampliamente como complemento a sistemas de navegación junto con otros sensores debido a sus ventajas. Estas incluyen su facilidad de uso, su capacidad para funcionar en una variedad de condiciones ambientales y su costo relativamente bajo. Su versatilidad los hace ideales para aplicaciones como sistemas de estacionamiento automático en vehículos, detección de obstáculos en robots móviles y sistemas de alarma anticolisión en entornos industriales.

Figura 2-6. Sensor Ultrasonido

2.2.1.2. LiDAR

El sensor LIDAR, abreviatura de "Light Detection and Ranging", es un dispositivo que emplea pulsos láser para medir la distancia entre el sensor y los objetos en su entorno. Este sistema emite pulsos láser de alta precisión y miden el tiempo que tarda cada pulso en rebotar en los objetos circundantes y regresar al sensor. La información recopilada se utiliza para calcular la distancia entre el sensor y los objetos. Dependiendo del tipo de medición que se requiera, existen dos tipos de LiDAR:

- LiDAR 2D: Emite pulsos láser en un único plano horizontal, midiendo distancia en ese único plano. Ideal para aplicaciones donde solo se requiere un perfil horizontal del entorno, como la detección de obstáculos en robots móviles o la navegación en interiores.

Figura 2-7. Sensor LiDAR 2D

- LiDAR 3D: Emite pulsos láser en múltiples planos, tanto horizontales como verticales, proporcionando una visión tridimensional del entorno. Crucial para aplicaciones que requieren un mapeo detallado en 3D.

Figura 2-8. Sensor LiDAR 3D

Su capacidad para generar mapas detallados del entorno, detectar objetos en movimiento y calcular distancias con precisión los hace indispensables en la robótica móvil, entre otras áreas.

2.2.2. SENSORES DE VISIÓN

2.2.2.1. CÁMARA ESTEREOSCÓPICA

Este tipo de dispositivo, también conocido como cámara 3D, aprovecha dos o más cámaras separadas a una distancia conocida para capturar imágenes desde distintos ángulos de un mismo objeto. Esta configuración permite percibir la profundidad y la distancia a los objetos en una escena tridimensional, siguiendo el mismo principio que el sistema visual humano.

Este sensor es sumamente versátil y es empleado en diversos campos, destacando especialmente en robótica móvil. Su capacidad para generar mapas de puntos en 3D les permite, en conjunción con otros sensores como el LiDAR, llevar a cabo tareas de localización y mapeo simultáneo (SLAM). Además, la información tridimensional obtenida por estos sensores es también útil para la detección de objetos, permitiendo al robot discernir su entorno de manera precisa. Esto habilita al robot para realizar una amplia gama de tareas, desde manipulación de objetos hasta visión artificial, incluido el reconocimiento de objetos y mucho más.

Figura 2-9. Cámara Estéreo

2.2.2.2. CÁMARA RGB-D

Este dispositivo, comúnmente conocido como cámara RGB-D o cámara de profundidad, difiere en su tecnología de las cámaras estéreo, pero comparte el objetivo de capturar información tridimensional de una escena. A diferencia de las cámaras estéreo, que utilizan dos o más cámaras separadas para calcular la profundidad mediante disparidades en las imágenes, la cámara RGB-D combina la captura de imágenes en color (RGB) con la capacidad de medir la profundidad (D) de los objetos en una escena, utilizando un sensor especializado para ello.

Aunque la tecnología subyacente es distinta, las cámaras RGB-D ofrecen prácticamente las mismas posibilidades que las cámaras estéreo, con la ventaja adicional de proporcionar información de color sobre los objetos en la escena. Esta información de

color puede ser crucial en aplicaciones donde la identificación de objetos basada en color es fundamental.

Figura 2-10. Cámara RGB-D

2.2.3. SENSORES PARA ODOMETRÍA

2.2.3.1. IMU

El sensor IMU (Unidad de Medición Inercial) es un dispositivo electrónico esencial en robótica móvil, diseñado para medir una variedad de parámetros físicos clave, como la velocidad angular, la aceleración lineal y, en algunos casos, el campo magnético. Este tipo de sensor se compone de varios componentes, incluyendo acelerómetros, giroscopios y magnetómetros.

- **Acelerómetro**: Este sensor mide la aceleración lineal experimentada por el dispositivo en las tres direcciones del espacio (x, y, z). Proporciona información sobre los cambios de velocidad y movimiento lineal del robot.

- **Giroscopio**: Encargado de medir la velocidad angular del dispositivo en las tres direcciones del espacio. Permite detectar y medir los cambios en la orientación y la rotación del robot.

- **Magnetómetro**: Mide el campo magnético terrestre en las tres direcciones del espacio, lo que ayuda a determinar la orientación del dispositivo con respecto al campo magnético de la Tierra.

La combinación de estas características convierte a este dispositivo en una herramienta extremadamente útil en la robótica móvil. Con la ayuda de algoritmos de filtrado y fusión de datos, el sensor IMU puede ayudar a obtener información sobre la posición y la orientación del robot en tiempo real, sin depender de señales externas como el GPS.

Figura 2-11. Sensor IMU

2.2.3.2. ENCODERS

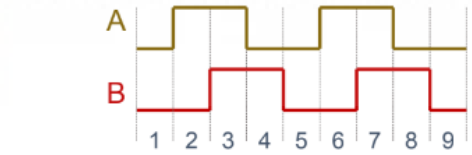
Los encoders son dispositivos electromecánicos utilizados para medir la velocidad, dirección y posición angular de un eje giratorio, como el de un motor o un eje de una rueda. Son componentes esenciales en sistemas de control de movimiento y robótica, ya que proporcionan retroalimentación precisa sobre el movimiento de los actuadores.

Los encoders funcionan generando pulsos eléctricos o señales digitales en respuesta al movimiento del eje al que están acoplados. Estos pulsos se pueden contar y medir para determinar con precisión la posición y la velocidad del eje.

Hay dos tipos principales de encoders:

- **Encoder Incremental:** Estos encoders generan pulsos eléctricos en respuesta al movimiento del eje, proporcionando información sobre el movimiento relativo, es decir, cuánto ha girado el eje desde su posición inicial. Son especialmente útiles para medir la velocidad y la dirección del movimiento. Sin embargo, para determinar la posición absoluta, se requiere una referencia conocida, así como un módulo contador. Los encoders incrementales son simples y económicos, pero necesitan un procedimiento de inicialización para establecer la posición absoluta.

  Además, existen encoders de diferentes tipos según la cantidad de canales que poseen. Los de un canal proporcionan información sobre el movimiento relativo. Los de dos canales permiten determinar el sentido del giro con mayor precisión. Y finalmente, los de tres canales, donde el tercer canal, conocido como señal de índice, proporciona una referencia absoluta que se utiliza para reiniciar el módulo contador y establecer la posición inicial de forma precisa.



Giro horario

Fase	A	B
1	0	0
2	1	0
3	1	1
4	0	1

Giro antihorario

Fase	A	B
5	0	0
4	0	1
3	1	1
2	1	0

Figura 2-12. Encoder Incremental de 2 canales

- **Encoders Absoluto:** Estos encoders proporcionan información sobre la posición absoluta del eje en cualquier momento, sin necesidad de una referencia inicial. Cada posición angular del eje se corresponde con un valor único de salida del encoder. Esto permite conocer la posición exacta del eje en cualquier momento, incluso después de un apagado y encendido del sistema. Los encoders absolutos son más complejos y costosos que los incrementales,

pero ofrecen una mayor precisión y fiabilidad en aplicaciones donde la precisión absoluta es crucial.

Dentro de los encoders absolutos, existen dos tipos según su capacidad para proporcionar información de la posición absoluta: los de vuelta simple, que ofrecen información de posición absoluta en una sola vuelta del eje, y los de vuelta múltiple, que son capaces de proporcionar información de posición absoluta en varias vueltas del eje, lo que brinda una mayor precisión.

Figura 2-13. Encoder absoluto

Además, según las características físicas utilizadas para detectar y medir el movimiento del eje, existen tres tipos de encoders:

- **Óptico**: Los encoders ópticos utilizan una fuente de luz (generalmente un LED) y un conjunto de sensores fotodetectores para medir el movimiento. Un disco de código óptico con patrones de franjas o ranuras giratorias altera la cantidad de luz que llega a los sensores, lo que permite detectar el movimiento y determinar la posición del eje.

- **Magnético**: Los encoders magnéticos utilizan campos magnéticos para detectar el movimiento del eje. Un disco de código magnético con áreas magnéticas y no magnéticas genera cambios en el campo magnético que son detectados por sensores de efecto Hall u otros sensores magnéticos.

- **Capacitivo**: Los encoders capacitivos utilizan cambios en la capacitancia entre un electrodo fijo y un electrodo móvil para detectar el movimiento del eje. A medida que el eje gira, la distancia entre los electrodos cambia, lo que provoca cambios en la capacitancia que se pueden medir para determinar la posición del eje.

3. ALGORITMOS EN ROBÓTICA MÓVIL

Como se ha destacado anteriormente, uno de los aspectos centrales en el campo de la robótica móvil ha sido el desarrollo e implementación de algoritmos destinados a otorgar autonomía a los robots, permitiéndoles navegar de forma autónoma y superar obstáculos imprevistos, adaptándose a diversas circunstancias. Estos algoritmos desempeñan un papel crucial en la capacidad de los robots para operar de manera eficiente y segura en entornos dinámicos y cambiantes.

Dentro del amplio espectro de algoritmos utilizados en robótica móvil, se encuentran los algoritmos de localización, de planificación de rutas (path planning) y de mapeo. Cada uno de estos tipos de algoritmos tiene como objetivo principal proporcionar al robot la capacidad de moverse de manera autónoma y realizar tareas específicas en su entorno.

Figura 3-1. Algoritmos utilizados en robótica móvil

3.1. ALGORITMOS DE LOCALIZACIÓN

Cuando se aborda el problema de la localización en el ámbito de la robótica móvil, es importante entender que no se trata de un solo desafío, sino de un conjunto de problemas de diversos tipos. Entre estos problemas, se encuentran:

- **Problema de localización local o seguimiento de posición**: En este escenario, se parte de una posición inicial conocida del robot, y el objetivo

principal es mitigar los errores incrementales que surgen durante el cálculo de la odometría del robot. Los algoritmos diseñados para este problema a menudo hacen suposiciones sobre el tamaño del error y la incertidumbre del robot.

- **Problema de localización global**: A diferencia del problema anterior, en este caso la posición inicial del robot no es conocida y debe ser estimada, lo que añade una complejidad adicional al proceso de seguimiento. Aquí, no se puede asumir que el error en la estimación de la posición del robot sea pequeño, lo que requiere enfoques más sofisticados para determinar la posición del robot.

- **Problema del robot secuestrado**: Este tipo de problema surge cuando el robot es trasladado repentinamente a una ubicación diferente sin previo aviso. Esto puede ocurrir, por ejemplo, si el robot es recogido y movido por una persona un vehículo. La principal dificultad en este caso es que el robot pierde temporalmente la capacidad de obtener datos de sus sensores para corregir su posición, lo que requiere estrategias especiales para detectar y recuperarse de esta situación.

3.1.1. FILTRO DE KALMAN

El filtro de Kalman es uno de los primeros algoritmos que se utilizaron en el campo de la estimación estadística, basado en una sólida base matemática que incluye conceptos de probabilidad, modelos de espacio de estados, matrices de covarianza y álgebra lineal. Fue desarrollado por Rudolf E. Kalman en la década de 1960 y desde entonces ha tenido una amplia aplicación en una variedad de áreas, desde la navegación y la ingeniería hasta la inteligencia artificial.

En esencia, el filtro de Kalman se basa en la teoría de la estadística bayesiana para actualizar creencias sobre el estado de un sistema a medida que se obtienen nuevas observaciones. Modela el sistema dinámico como un modelo de espacio de estados, utilizando distribuciones de probabilidad gaussiana para representar tanto el proceso de estado como el proceso de observación.

El proceso del filtro de Kalman se puede dividir en dos etapas principales: la predicción y la actualización. En la etapa de predicción, se utiliza el modelo dinámico del sistema para predecir el estado futuro del sistema, junto con su covarianza. En la etapa de actualización, se incorporan las nuevas mediciones para corregir la predicción anterior y mejorar la estimación del estado actual del sistema.

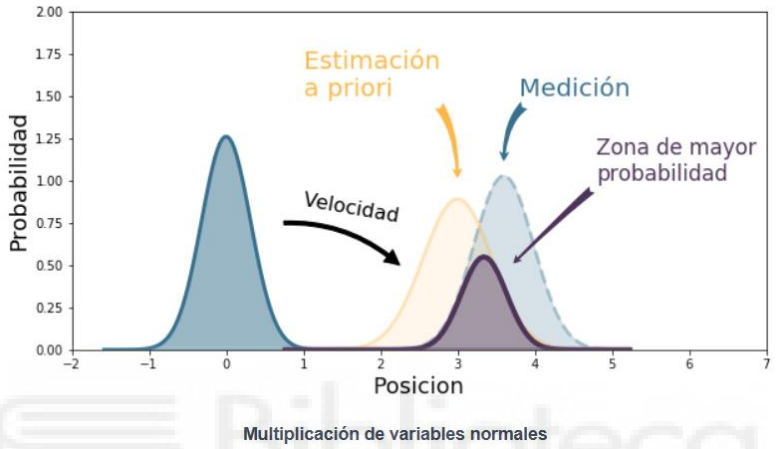


Figura 3-2. Representación etapas Filtro de Kalman

Sin embargo, a pesar de su utilidad generalizada, el filtro de Kalman presenta limitaciones en problemas de localización global y en la solución de problemas de robot secuestrado. Estas limitaciones surgen de su dependencia en ecuaciones de movimiento lineales y la restricción inherente de las distribuciones gaussianas para representar una sola hipótesis, lo que puede no ser suficiente para abordar la complejidad y la incertidumbre en estos escenarios.

3.1.2. FILTRO DE KALMAN EXTENDIDO

El Filtro de Kalman Extendido (EKF) es una solución innovadora que amplía las capacidades del filtro de Kalman clásico para abordar sistemas dinámicos no lineales. Su desarrollo surgió de la necesidad de aplicar técnicas de estimación en situaciones donde los modelos y mediciones no cumplen estrictamente con la linealidad.

Al igual que su predecesor, el EKF se fundamenta en los principios de la estadística bayesiana y el modelo de espacio de estados. Sin embargo, en lugar de limitarse a modelos

lineales, el EKF emplea una aproximación lineal local alrededor de la estimación actual del estado del sistema. Esto permite que el EKF sea flexible y capaz de manejar sistemas no lineales.

Si bien el EKF conserva su utilidad en la resolución de problemas de localización local, donde el robot debe estimar su posición y orientación dentro de un área circunscrita, su eficacia puede disminuir en problemas de localización global, donde la precisión y la robustez son cruciales en entornos más amplios y complejos. En tales casos, se pueden requerir enfoques complementarios para lograr una solución satisfactoria.

3.1.3. LOCALIZACIÓN MONTE CARLO

El algoritmo de Localización Monte Carlo, también conocido como filtro de partículas, fue introducido en la década de 1990 como una técnica para abordar el problema de la localización en robótica móvil. La idea básica es utilizar muestras aleatorias (como en los métodos de Monte Carlo) para estimar la posición de un robot en función de información sensorial y de movimiento.

Se trata de un algoritmo estadístico no paramétrico que utiliza un conjunto de partículas para representar el estado del sistema en un mapa de un entorno conocido. En el contexto de la robótica móvil, cada partícula representa una hipótesis sobre la posición y orientación del robot y tiene un peso asociado que indica la probabilidad de que esa hipótesis sea correcta. A medida que el robot se mueve y recibe nuevas observaciones, estos pesos se actualizan para reflejar mejor la probabilidad de cada hipótesis.

El funcionamiento del filtro de partículas se basa en cuatro etapas clave:

3.1.3.1. INICIALIZACIÓN

La primera etapa de inicialización se realiza solo una vez al inicio del código para generar un conjunto de N partículas repartidas por el mapa del entorno conocido. Dependiendo del conocimiento que se tenga sobre el estado del sistema en el momento de la inicialización, se pueden presentar dos situaciones:

- Desconocimiento del estado inicial: Esto se conoce como un problema de localización global. Se desconoce la posición inicial del robot, por lo que se necesita

una gran cantidad de partículas distribuidas por todo el mapa. Cuanto mayor sea el conjunto de partículas, mayor será la probabilidad de una convergencia exitosa al estado real del robot. A medida que las partículas comienzan a converger, es posible reducir el número de partículas.

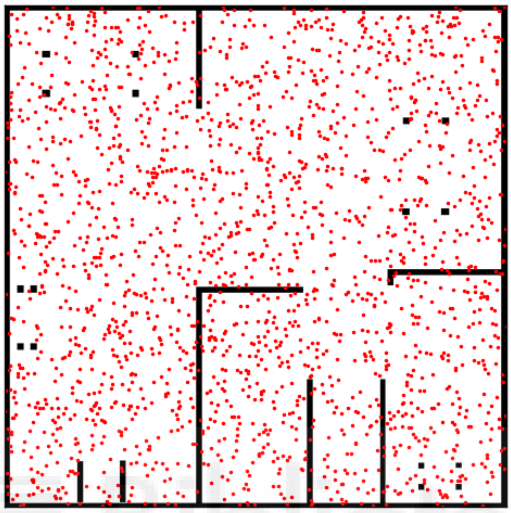


Figura 3-3. Situación de desconocimiento del estado inicial del sistema en la etapa de inicialización del algoritmo de localización Monte Carlo.

- Conocimiento del estado inicial: En este caso, se necesita una menor cantidad de partículas, agrupando este conjunto de N partículas alrededor del estado estimado.

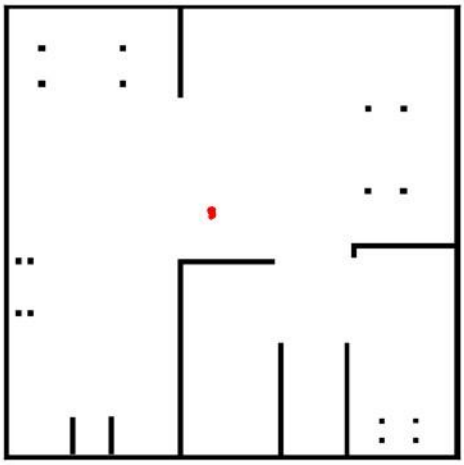


Figura 3-4. Situación de conocimiento del estado inicial del sistema en la etapa de inicialización del algoritmo de localización Monte Carlo

3.1.3.2. PREDICCIÓN

En la etapa de predicción, se emplea el modelo de movimiento del robot para calcular la distancia de desplazamiento y aplicar dicha distancia y dirección a todas y cada una de las partículas del estado anterior. Este modelo debe simular las posibles incertidumbres y errores inherentes al movimiento en forma de ruido.

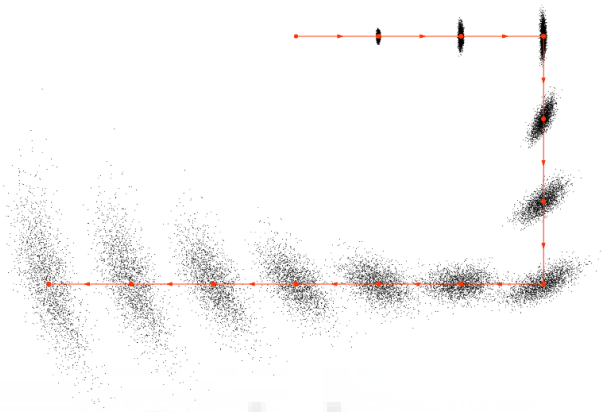


Figura 3-5. Comportamiento de las partículas después de varios pasos utilizando solo la etapa de predicción

3.1.3.3. ACTUALIZACIÓN

En esta etapa, se utilizan los datos recogidos por los sensores para ajustar los pesos de cada partícula. Se comparan las observaciones de cada partícula en la posición actualizada en la etapa anterior con las observaciones del robot real, asignando mayor peso a aquellas partículas cuyas observaciones se asemejan más a las del robot real. Además, una vez actualizados los pesos, estos se normalizan para que su suma sea igual a uno, manteniendo la coherencia probabilística.

3.1.3.4. RE-MUESTREO

En esta última etapa, se busca eliminar aquellas partículas con pesos bajos, es decir, que se han alejado demasiado de la posición estimada del robot, y duplicar aquellas con mayor peso, es decir, que están muy cerca de la posición estimada del robot. Este paso asegura que las partículas con mayor probabilidad tengan una representación adecuada en la próxima iteración, mejorando la precisión de la estimación.

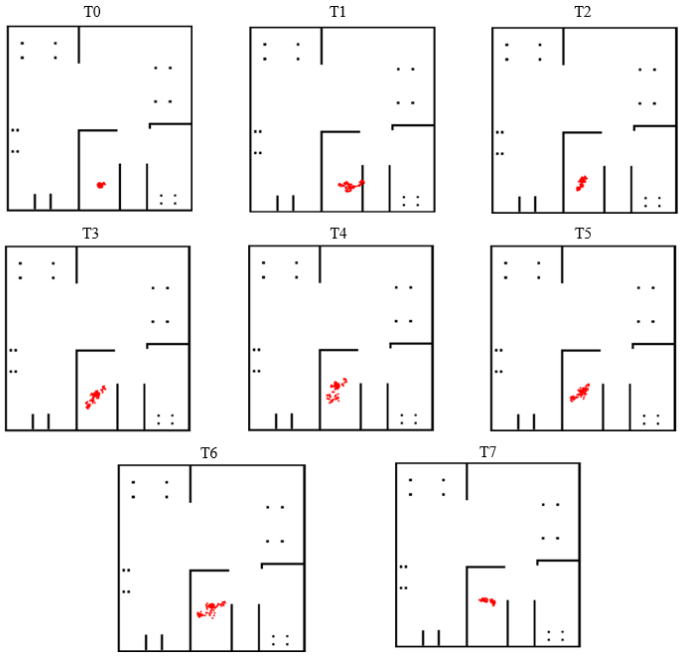


Figura 3-6. Secuencia del algoritmo de Localización Monte Carlo

La estructura del filtro de partículas, junto con los principios de los métodos Monte Carlo, confieren al algoritmo de Localización Monte Carlo su robustez y flexibilidad para enfrentar la localización de robots en entornos dinámicos y no lineales. Esta técnica demuestra su capacidad de adaptación a una amplia gama de problemas de localización, pudiendo resolver el problema con diversos tipos de sensores y modelos de movimiento. Asimismo, su capacidad para re-localizar eficientemente al robot en casos de pérdida de posición, gracias a su habilidad para representar múltiples hipótesis simultáneamente, resalta su potencial en la resolución tanto de problemas de localización global como local.

3.2. ALGORITMOS DE PLANIFICACIÓN DE TRAYECTORIA

En el contexto de la planificación de trayectorias en robótica móvil, es fundamental comprender que existen dos enfoques principales según el nivel de conocimiento del entorno en el que el robot operará:

- **Planificación de trayectorias global**: Este enfoque implica la generación de una ruta completa desde la posición inicial del robot hasta su destino final, basándose en la información disponible del entorno. La planificación global busca

encontrar la ruta óptima o subóptima que minimice algún criterio, como la distancia recorrida, el tiempo de llegada o el consumo de energía, entre otros.

- **Planificación de trayectorias local**: En contraste, la planificación de trayectorias local se enfoca en generar movimientos de corto alcance para el robot mientras navega en el entorno. Este enfoque se fundamenta en información sensorial en tiempo real y resulta especialmente útil en entornos dinámicos o desconocidos, donde la información disponible sobre el entorno puede ser parcial o totalmente desconocida. Los algoritmos de planificación local crean secuencias de acciones que permiten al robot evitar obstáculos y alcanzar su objetivo de manera segura y eficiente.

Sin embargo, en las estrategias prácticas de planificación de trayectorias, la combinación de ambas planificaciones puede resultar en un sistema de navegación robusto y adaptable, capaz de enfrentarse a una gran variedad de escenarios en el mundo real.

3.2.1. PLANIFICACIÓN DE TRAYECTORIA DE DIJKSTRA

El planificador de trayectorias de Dijkstra es una implementación del algoritmo de Dijkstra, nombrado así en honor a su autor Edsger Dijkstra, quien lo publicó por primera vez en 1959. Este algoritmo se emplea para resolver el problema del camino más corto desde un nodo origen hasta todos los demás nodos en un grafo ponderado con pesos no negativos. El algoritmo funciona manteniendo una lista de nodos visitados y una lista de nodos por visitar. En cada paso, selecciona el nodo no visitado con la distancia mínima conocida desde el nodo de origen y lo marca como visitado. Luego, actualiza las distancias a los nodos adyacentes al nodo recién visitado, si la distancia a través del nodo recién visitado es menor que la distancia previamente conocida. Este proceso continúa hasta que todos los nodos hayan sido visitados.

En robótica, este algoritmo se utiliza para encontrar la ruta más corta entre un punto inicial y un punto final en un entorno conocido. Este entorno se representa como un grafo donde los nodos representan posiciones posibles del robot dentro del entorno y las aristas representan posibles movimientos del robot con sus respectivos costos (distancias). El

algoritmo permite evitar obstáculos y obtener los caminos más cortos, pero el costo computacional puede ser elevado si el entorno discretizado es muy grande.

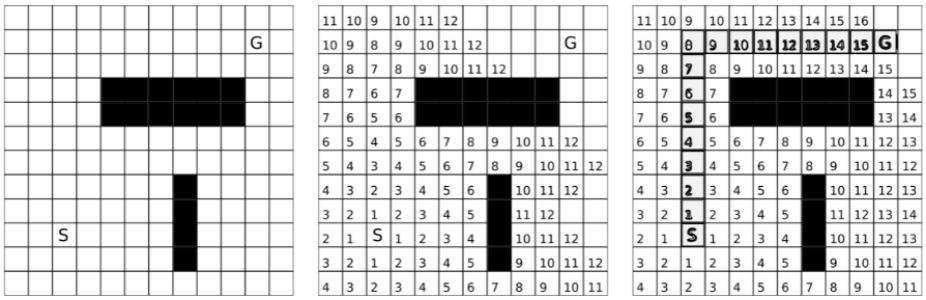


Figura 3-7. Algoritmo de Dijkstra encontrando una ruta más corta de 'S' a 'G' suponiendo que el robot solo puede viajar lateralmente (no diagonalmente) con un costo de uno por celda de cuadrícula.

El algoritmo se utiliza principalmente para la planificación global, ya que asume que el entorno es estático y no puede enfrentarse a posibles nuevos obstáculos que puedan aparecer durante la ejecución del plan.

3.2.2. ALGORITMO RRT

El planificador de trayectorias RRT, empleado principalmente para la planificación de trayectorias global, fue propuesto por Steven M. LaValle en 1998 y desde entonces ha ganado gran popularidad gracias a su capacidad para manejar entornos de alta dimensionalidad y complejidad. A diferencia de otros algoritmos que exploran exhaustivamente todo el espacio de configuración, el RRT se basa en una exploración aleatoria y dirigida del espacio de configuración.

El funcionamiento de este algoritmo comienza con un solo nodo, que representa la posición inicial del robot en el espacio de configuración. En cada iteración del algoritmo, se genera aleatoriamente un punto en el espacio de configuración. Luego, se busca el nodo más cercano en el árbol actual al punto generado para unirlo, siempre y cuando el punto no esté situado en un obstáculo del entorno. Una vez que se ha explorado suficientemente el espacio de configuración y se ha alcanzado el nodo objetivo (o una proximidad aceptable al mismo), se puede construir la trayectoria final retrocediendo desde el nodo objetivo hasta el nodo raíz siguiendo los enlaces en el árbol.

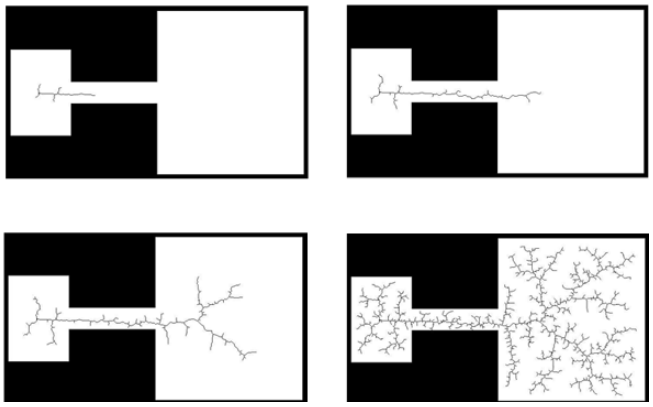


Figura 3-8. Evolución algoritmo RRT. Fuente:

3.2.3. ALGORITMO DE CAMPOS POTENCIALES

El algoritmo de campos potenciales es un planificador de trayectorias local que destaca por su capacidad para generar movimientos en tiempo real, lo que permite al robot esquivar obstáculos de forma dinámica. Su funcionamiento se basa en la analogía de modelar al robot como una partícula cargada y el entorno como un campo de potencial.

Este algoritmo utiliza dos funciones fundamentales:

- **Función Potencial Atractiva**: Esta función crea el potencial asociado al objetivo deseado. Se comporta como una carga de signo opuesto, atrayendo al robot hacia el objetivo. Normalmente, aumenta mientras el robot se aleja del objetivo y disminuye a medida que se acerca.
- **Función Potencial Repulsiva**: Genera el potencial asociado a los obstáculos, actuando como cargas del mismo signo que el robot y provocando una fuerza de repulsión. Su magnitud aumenta a medida que el robot se acerca al obstáculo y disminuye al alejarse.

Para moverse, el robot calcula el gradiente del campo potencial total, que es la suma de los potenciales atractivos y repulsivos. Luego, se desplaza en la dirección opuesta a este gradiente, siguiendo la ruta de mayor descenso del campo potencial.

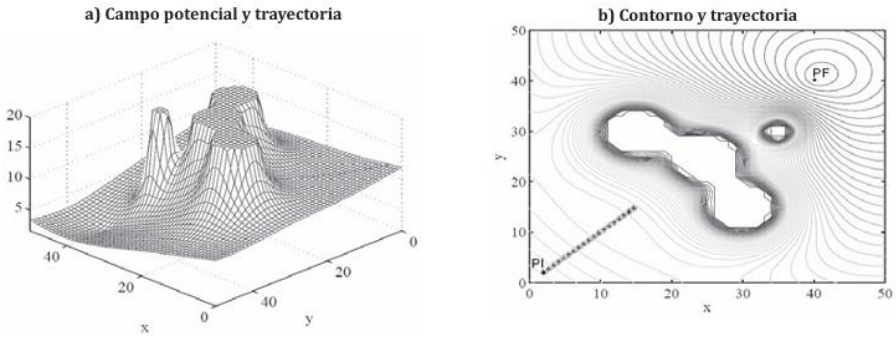


Figura 3-9. a) Campo potencial con la presencia de una barrera, b) Contorno del campo potencial con la presencia de una barrera. Fuente:

Sin embargo, el algoritmo tiene una limitación: el robot puede quedar atrapado en mínimos locales debido a la naturaleza de la función de potencial, que puede presentar equilibrios. Esto significa que el robot podría detenerse en lugares que no son el objetivo final. Para abordar este inconveniente, se pueden emplear estrategias adicionales. Una de ellas es la generación de movimientos aleatorios cuando se detecta un mínimo local, lo que permite que el robot escape de esa situación. Otra opción es aplicar fuerzas virtuales para empujar al robot fuera de mínimos locales y continuar su progreso hacia el objetivo.

3.2.4. ALGORITMO A*

El algoritmo de planificación de trayectorias conocido como A* fue desarrollado por Peter Hart, Nils Nilsson y Bertram Raphael en 1968, y desde entonces ha sido ampliamente utilizado en una variedad de aplicaciones. Al igual que el algoritmo de Dijkstra, A* se utiliza para encontrar el camino más corto desde un nodo inicial hasta un nodo objetivo en un grafo ponderado.

Sin embargo, la diferencia principal entre A* y el planificador Dijkstra a la hora de realizar planificaciones de trayectoria radica en la introducción de una función adicional conocida como función heurística. Esta función proporciona una estimación del costo restante para llegar al nodo objetivo desde cualquier nodo dado en el grafo. La función heurística guía la búsqueda de A* al proporcionar una estimación del costo futuro, lo que permite al algoritmo priorizar los nodos que probablemente conduzcan a la solución más rápidamente.

Es importante destacar que la elección de una función heurística adecuada puede influir en el rendimiento del algoritmo A*. Algunas de las funciones heurísticas más comunes usadas en robótica son:

- Distancia Manhattan: Esta heurística calcula la distancia horizontal y vertical entre el nodo actual y el nodo objetivo en una cuadrícula, ignorando los obstáculos.

- Distancia Euclídea: Esta heurística calcula la distancia en línea recta entre el nodo actual y el nodo objetivo en un espacio euclidiano.

La evaluación de los nodos se realiza mediante una combinación del costo acumulado desde el nodo inicial hasta el nodo actual (conocido como "costo g") y la estimación heurística del costo restante hasta el nodo objetivo desde el nodo actual (denominado "costo h"). Sumando estos costos se obtiene el costo total de un nodo. A* selecciona el nodo con el costo total más bajo para expandir y continuar la búsqueda en la dirección que parece ser más prometedora.

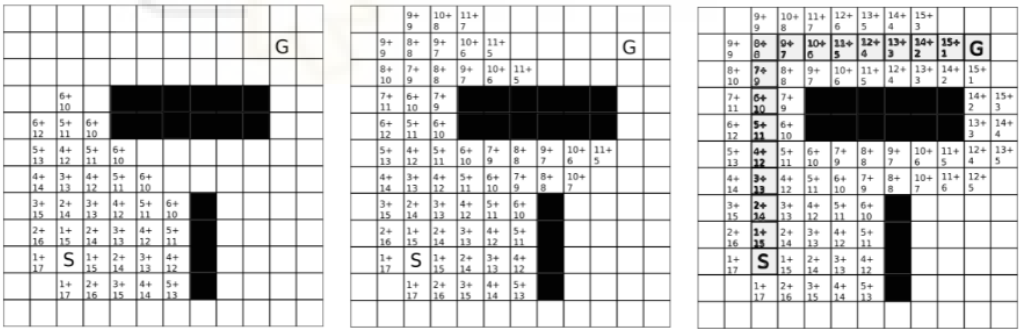


Figura 3-10. Algoritmo de A* encontrando una ruta más corta de 'S' a 'G' suponiendo que el robot solo puede viajar lateralmente (no diagonalmente) con un costo de uno por celda de cuadrícula más el costo heurístico.

4. HERRAMIENTA SOFTWARE

4.1. COPPELIASIM

CoppeliaSim, anteriormente conocido como V-REP (Virtual Robot Experimentation Platform), es un software de simulación de robots y sistemas robóticos desarrollado por Coppelia Robotics. Es una herramienta versátil y potente que permite a los usuarios simular y visualizar el comportamiento de robots en un entorno virtual tridimensional, ya que ofrece una interfaz gráfica intuitiva que permite a los usuarios diseñar entornos de simulación personalizados, importar modelos de robots en 3D y programar el comportamiento de los robots utilizando diversos lenguajes de programación (como Python, C/C++, MATLAB, etc.).

4.1.1. ENTORNO DE SIMULACIÓN

A la hora de diseñar un entorno de simulación, es crucial familiarizarse con las diversas herramientas que ofrece el software de simulación, en este caso CoppeliaSim, y comprender cómo utilizarlo eficazmente.

Al iniciar CoppeliaSim, lo primero que se encuentra es una ventana principal que presenta varios elementos clave para navegar por el programa y poder empezar con el diseño.

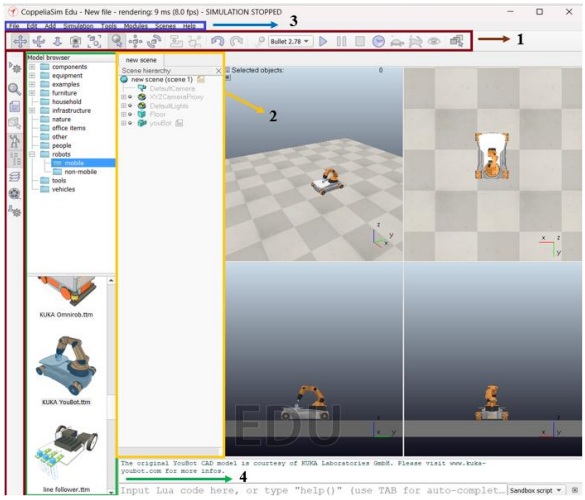


Figura 4-1. Interfaz de usuario CoppeliaSim

1. **Barra de herramientas**: En ella se encuentran una serie de funciones de uso frecuente, a las cuales puedes acceder directamente o a través de la barra de menú.

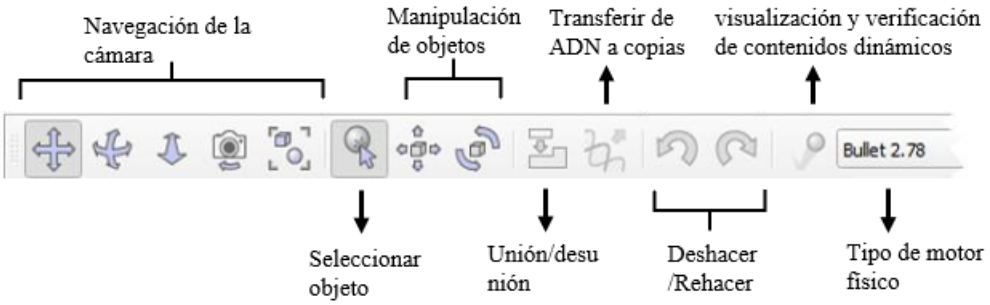


Figura 4-2. Funciones de CoppeliaSim en la barra de herramientas horizontal

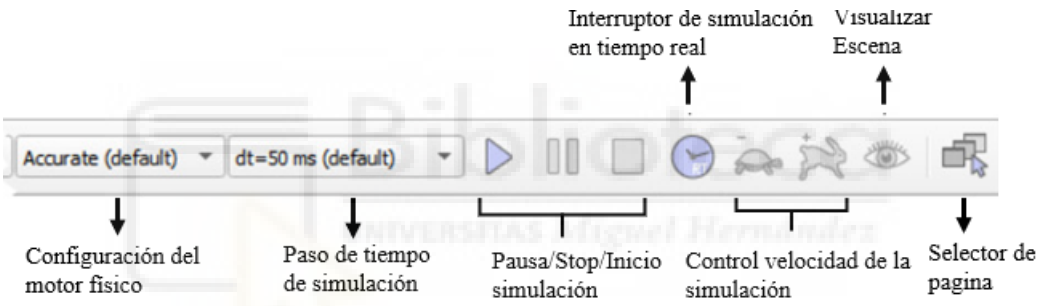


Figura 4-3. Funciones de CoppeliaSim en la barra de herramientas horizontal

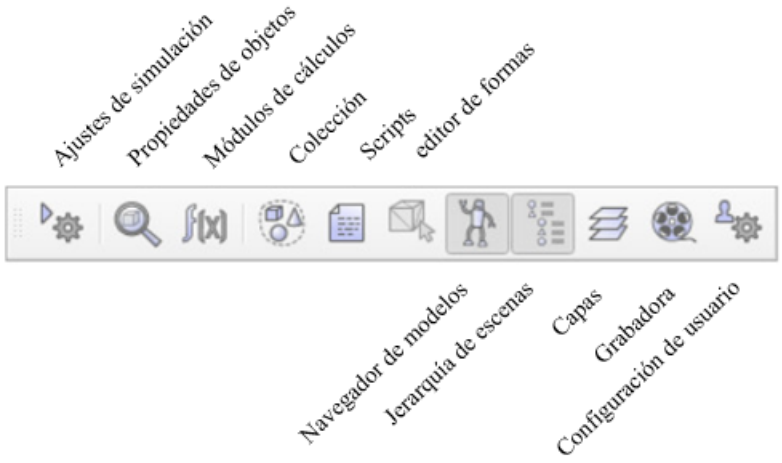


Figura 4-4. Funciones de CoppeliaSim en la barra de herramientas vertical

2. **Jerarquía de escenas**: Muestra todos los objetos que componen la escena de manera jerárquica, agrupando todos los objetos que conforman un modelo en un árbol jerárquico independiente. Puedes editar el nombre o características de los objetos dando doble clic sobre el icono correspondiente. Además, puedes cambiar el orden jerárquico de los objetos arrastrándolos y soltándolos en la posición deseada.

3. **Menú**: Permite acceder a casi todas las funciones del simulador. Sin embargo, es importante tener en cuenta que el contenido que se muestra puede variar dependiendo del estado actual del simulador. Por ejemplo, algunas funciones pueden deshabilitarse si hay una simulación en marcha para evitar interferencias o cambios no deseados en el entorno de simulación.

4. **Navegador de modelos**: El navegador se divide en dos secciones principales: la parte superior presenta una estructura de carpetas que contiene todos los modelos predeterminados ofrecidos por CoppeliaSim. Mientras tanto, en la sección inferior, se muestra una vista en miniatura de los modelos contenidos en la carpeta seleccionada, permitiendo una visualización rápida y fácil a escala reducida.

4.1.2. ELEMENTOS DE COPPELIASIM

Una vez que se ha explicado la estructura de la ventana de CoppeliaSim, es importante entender cómo funciona la estructura de la escena y qué elementos la componen. Es decir, entender qué significa que un elemento sea un modelo, qué características se le atribuyen y cuál es la diferencia con lo que se conoce como objetos de escena.

4.1.2.1. ESCENAS

Una escena es la columna vertebral del entorno de simulación en CoppeliaSim. En ella, se integran una variedad de elementos específicos que facilitan la interacción y el desarrollo del proyecto. Estos elementos incluyen:

- **Entorno**: Proporciona un conjunto de propiedades y parámetros que influyen en la caracterización general de la escena, sin necesidad de estar vinculados a objetos específicos. Entre estas configuraciones se encuentran el color del fondo, los parámetros que controlan la niebla y la luz ambiente. Estas opciones

permiten personalizar cada aspecto de la escena según las necesidades y preferencias específicas, sin afectar directamente a los elementos individuales presentes en la escena.

- **Script principal**: Se trata de un script de simulación encargado de dirigir el comportamiento y la lógica de esta. Por lo general, este script incluye instrucciones que regulan la interacción entre los diversos objetos presentes en la escena y el entorno en sí. Es importante destacar que se aconseja no modificar este script ni las funciones callback que se incluyen por defecto: '**sysCall_init**', '**sysCall_actuation**', '**sysCall_sensing**' y '**sysCall_cleanup**'. Estas funciones son fundamentales para el correcto funcionamiento de la simulación y cualquier modificación podría afectar su comportamiento esperado.

- **Página y vistas**: Estas herramientas son esenciales para la visualización de la escena, pero cada una tiene sus propias características distintivas. Una página es el espacio donde se puede observar la escena en conjunto, incluidas todas las vistas asociadas a objetos de escena como cámaras o sensores de visión. Por otro lado, una vista se centra en mostrar el contenido visual de un objeto de escena específico, ya sea una cámara o un sensor de visión. Esta distinción permite una visualización detallada y específica de los elementos importantes dentro de la simulación.

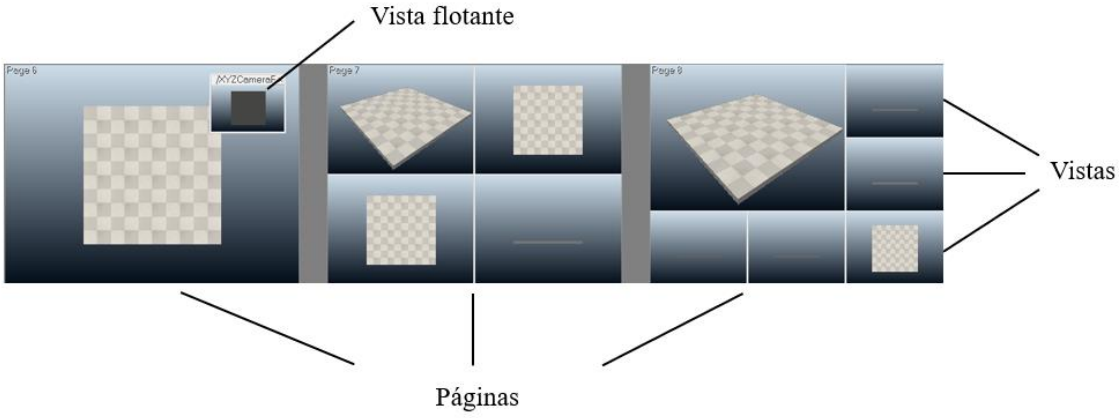


Figura 4-5. Ejemplos de diferentes configuraciones de página y vistas asociadas

Además de estos elementos esenciales, CoppeliaSim incluye de manera predeterminada una serie de modelos. Estos modelos están compuestos por objetos de escena que amplían las capacidades de la simulación al ofrecer funcionalidades como:

- Modificación de la iluminación: Permitiendo ajustar la luminosidad y el ambiente visual de la escena para recrear condiciones realistas o específicas del proyecto.

- Observación desde diferentes perspectivas: Ofreciendo la posibilidad de visualizar la escena desde múltiples ángulos y puntos de vista, lo que facilita la comprensión y el análisis del comportamiento de los objetos en la simulación.

- Modificación del suelo: Posibilitando cambiar la apariencia y las propiedades físicas del suelo de la escena, lo que permite simular diversos tipos de terrenos y superficies con precisión.

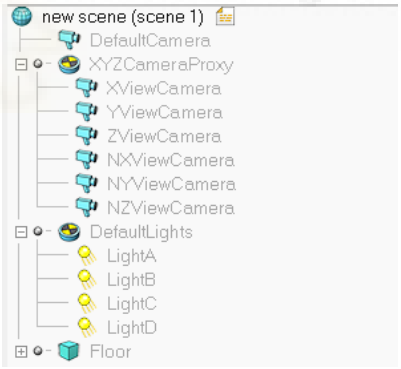


Figura 4-6. Modelos predefinidos en una escena

4.1.2.2. MODELOS

Los modelos son elementos secundarios de la escena, conformados por un conjunto de objetos de escena organizados en una jerarquía específica. Los modelos no pueden ser simulados de forma independiente y para identificarlos, son marcados con una etiqueta distintiva, lo que permite su reconocimiento dentro del entorno de simulación.

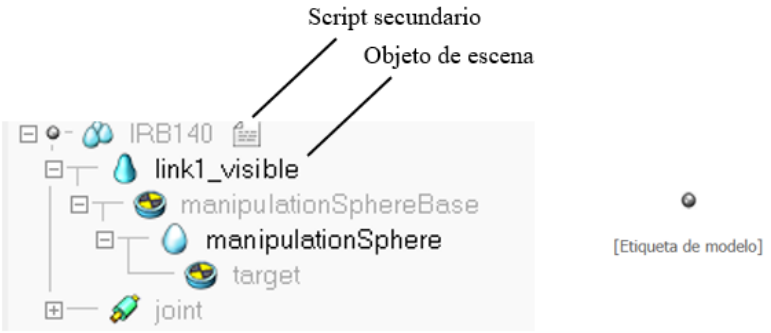


Figura 4-7. Estructura de un modelo en CoppeliaSim

4.1.2.3. OBJETOS DE ESCENA

Los objetos de escenas u objetos son los elementos fundamentales en el desarrollo de una escena y en la creación de cualquier modelo que se quiera simular. Algunos de estos objetos cuentan con propiedades especiales configurables que les permiten interactuar con otros objetos:

- **Colisionable**: Permite detectar si ocurre una colisión entre dos objetos colisionables, lo que facilita la realización de pruebas y simulaciones.
- **Medible**: Los objetos medibles pueden calcular la distancia mínima entre ellos y otros objetos medibles.
- **Detectable**: Pueden ser detectados por sensores de proximidad.
- **Visible**: Permiten ser vistos en la escena y por los diferentes objetos de visión.

Existen diversas categorías de objetos de escena, cada una diseñada para diferentes propósitos dentro de la simulación. En este trabajo, nos centraremos en presentar las categorías que han sido empleadas:

- Formas: Son objetos de malla rígida compuestos por caras triangulares, con capacidad para ser importados, exportados y editados. Se clasifican en cuatro subtipos distintos: simple o compuesta, convexa o compuesta convexa, primitiva o compuesta primitiva, y campo de altura.

Figura 4-8. Tipo de forma, de izquierda a derecha: simple, compuesta, convexa, convexa compuesta, primitiva, primitiva compuesta y campo de altura

- Articulaciones: Estos elementos funcionan como articulaciones o actuadores y se dividen en cuatro tipos principales: giratorias, prismáticas, esféricas y tornillos, este último siendo una combinación de una articulación giratoria y prismática.

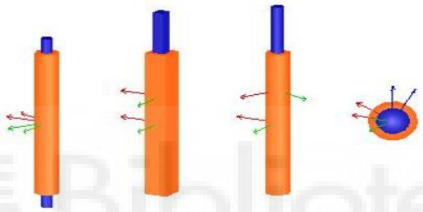


Figura 4-9. Tipo de articulación, de derecha a izquierda: giratoria, prismática, tornillo y esférica

- Sensores de proximidad: Estos elementos detectan objetos habilitados para ser detectados, a excepción de los dummies, que no pueden ser detectados por los sensores de tipo rayo. Hay diferentes tipos de sensores según la forma de los haces: piramidales, cilíndricos, de disco, cónicos, de rayos y rayos aleatorios.

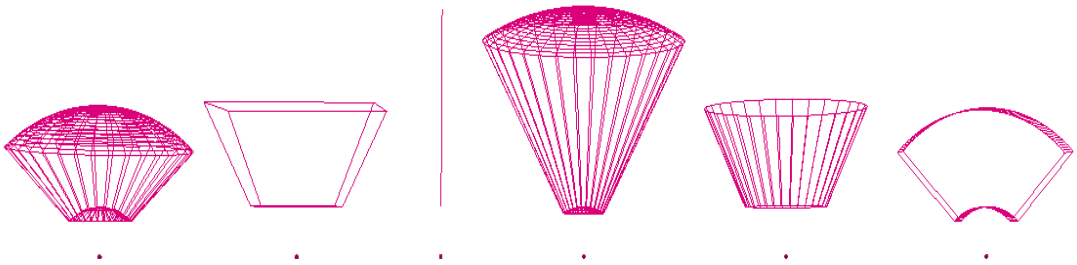


Figura 4-10. Tipo de sensor de proximidad según la forma de los haces, de izquierda a derecha: cónico, piramidal, rayo, rayo aleatorio, cilíndrico y de dico

- Sensores de visión: Estos elementos renderizarán los objetos que están en su campo de visión y tiene una resolución fija. Se recomiendan su uso en

aplicaciones donde el color, la luz o la estructura desempeñan un papel en el proceso de detección. Hay dos tipos de sensores de visión: sensor ortográfico y sensor de perspectiva.

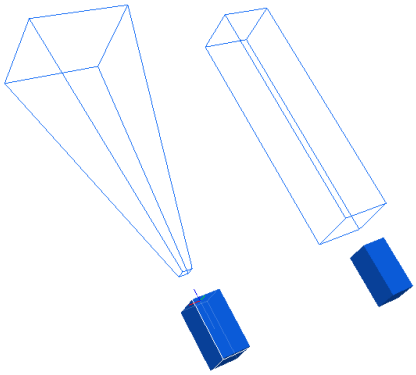


Figura 4-11. Tipo de sensor de visión, de derecha a izquierda: sensor de perspectiva y sensor ortográfico

4.2. ROBOT YOUBOT

El robot utilizado en este trabajo ha sido creado por KUKA Robotics, una destacada empresa en el campo de la robótica industrial. Diseñado específicamente para aplicaciones en investigación y educación en robótica, este robot ha ganado reconocimiento gracias a su excepcional movilidad y capacidades de manipulación.

El robot Kuka Youbot se compone principalmente de dos elementos clave: una base móvil altamente dinámica, diseñada para facilitar una amplia gama de movimientos gracias a su configuración especial, y un brazo robótico que añade aún más versatilidad al conjunto.

Figura 4-12. Elementos del robot Youbot

4.2.1. PLATAFORMA MÓVIL

La plataforma del robot Youbot puede moverse de manera omnidireccional gracias a las cuatro ruedas mecanum que la componen. Estas ruedas tienen la capacidad única de permitir desplazamientos en línea recta y realizar giros simultáneamente, lo que posibilita ajustar la orientación del robot mientras se desplaza.

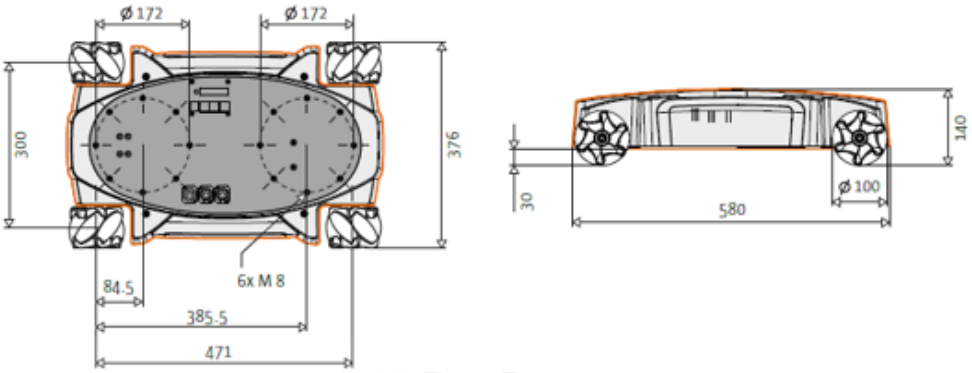


Figura 4-13. Dimensiones y configuración plataforma móvil del robot Youbot

Cada rueda mecanum está equipada con una serie de rodillos dispuestos oblicuamente a lo largo de la circunferencia de la llanta. En cuanto a la configuración de las ruedas, se puede observar en la Figura *4-14* que se está ante una estructura AB, donde el ángulo de los rodillos de las llantas "B" llamado β tiene un valor de 45°, mientras que el ángulo α de las llantas "A" tiene un valor de 135°, ambos respecto al sistema de referencia que se muestra sobre la plataforma.

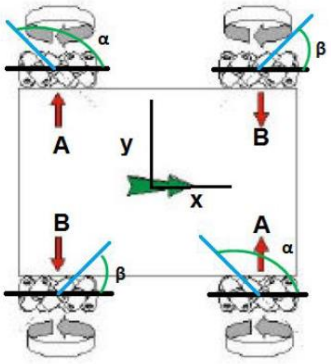


Figura 4-14. Configuración de las ruedas mecanum en la plataforma móvil del robot Youbot

Además, cada rueda es independiente en cuanto a su capacidad de movimiento y está equipada con su propio sistema motriz, lo que le permite al robot ejecutar una amplia variedad de movimientos en cualquier dirección.

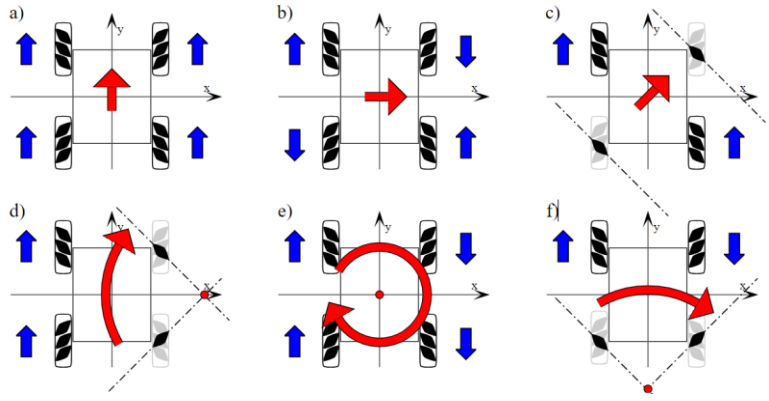


Figura 4-15. Posibles movimientos de la plataforma móvil debido a las ruedas mecanum

4.2.1.1. ESTUDIO CINEMÁTICO

Para controlar la velocidad del punto central de la plataforma en términos lineales (v_x, v_y, w), se llevó a cabo un estudio cinemático detallado de la base. Este análisis permitió determinar las velocidades angulares de cada rueda ($\phi_1, \phi_2, \phi_3, \phi_4$).

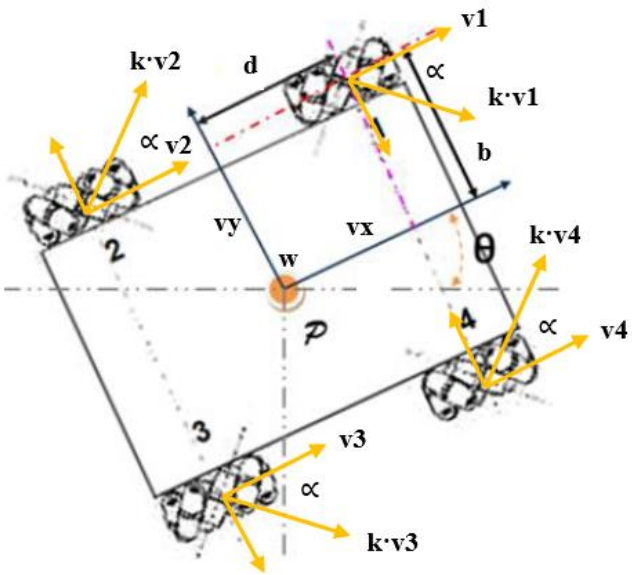


Figura 4-16. Modelo Cinemático

De acuerdo con el modelo cinemático y lo que se ha comentado en párrafo anterior, la velocidad del punto central se descompone en velocidad en el eje X, velocidad en el eje Y y velocidad angular de rotación.

$$(v_x, v_y, \mathrm{w})$$

Al plantear el estudio cinemático para las ruedas mecanum, es esencial considerar que el movimiento de una rueda hacia adelante también genera un desplazamiento lateral. Por lo tanto, para capturar todas las influencias de las cuatro ruedas en el punto central de la plataforma, se obtiene la siguiente relación:

$$\mathrm{v_i} + \mathrm{kv_i}\cos \propto_\mathrm{i} = \mathrm{v_x} + \mathrm{b_i w} \qquad (1)$$

$$\mathrm{kv_i}\sin \propto_\mathrm{i} = \mathrm{v_y} + \mathrm{d_i w} \qquad (2)$$

Donde 'i' representa el número de la rueda y $\propto_i$ es la inclinación de los rodillos. Al obtener esta relación entre las ruedas y el punto central de la plataforma, se puede aplicar una transformación, resultando en:

$$\tan \propto_\mathrm{i} = \frac{\mathrm{kv_i}\cos\propto_\mathrm{i}}{\mathrm{kv_i}\sin\propto_\mathrm{i}} = \frac{\mathrm{v_y}+\mathrm{d_i w}}{-\mathrm{v_i}+\mathrm{v_x}+\mathrm{b_i w}} \qquad (3)$$

$$\mathrm{v_i} = \mathrm{v_x} + \mathrm{b_i w} - \frac{\mathrm{v_y}+\mathrm{d_i w}}{\tan\propto_\mathrm{i}} \qquad (4)$$

Al particularizar esta relación para cada rueda y considerar las inclinaciones de los rodillos $\propto_i$, donde $\propto_{2,4}$=135º y $\propto_{1,3}$=45º, obtenemos:

$$\mathrm{v_1} = \mathrm{v_x} - \mathrm{v_y} - \mathrm{b_i w} - \mathrm{d_i w} \qquad (5)$$

$$\mathrm{v_2} = \mathrm{v_x} + \mathrm{v_y} - \mathrm{b_i w} - \mathrm{d_i w} \qquad (6)$$

$$\mathrm{v_3} = \mathrm{v_x} - \mathrm{v_y} + \mathrm{b_i w} + \mathrm{d_i w} \qquad (7)$$

$$\mathrm{v_4} = \mathrm{v_x} + \mathrm{v_y} + \mathrm{b_i w} + \mathrm{d_i w} \qquad (8)$$

Considerando además que:

$$\mathrm{v_i} = R * \mathrm{w_i} \qquad (9)$$

Finalmente, se puede obtener las ecuaciones que relacionan la velocidad del punto central de la plataforma con las velocidades angulares de las ruedas.

$$\begin{bmatrix}\phi_1\\ \phi_2\\ \phi_3\\ \phi_4\end{bmatrix} = \begin{bmatrix}\frac{1}{R} & -\frac{1}{R} & -\frac{b+d}{R}\\ \frac{1}{R} & \frac{1}{R} & -\frac{b+d}{R}\\ \frac{1}{R} & -\frac{1}{R} & \frac{b+d}{R}\\ \frac{1}{R} & \frac{1}{R} & \frac{b+d}{R}\end{bmatrix} * \begin{bmatrix}V_x\\ V_y\\ W\end{bmatrix} \quad (10)$$

4.2.2. BRAZO ROBÓTICO

El brazo robótico del robot KUKA YouBot utilizado en este proyecto consta de cinco grados de libertad, los cuales están definidos por cinco articulaciones rotacionales y sus respectivos eslabones. Además, está equipado con una pinza en su extremo que le permite manipular objetos.

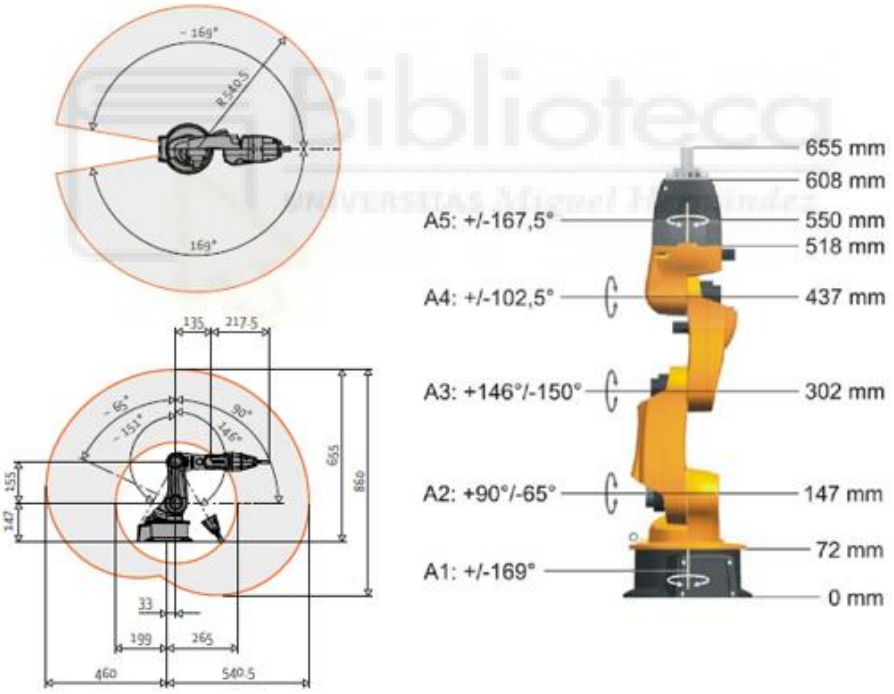


Figura 4-17. Dimensiones y área de trabajo del brazo del Kuka Youbot

4.2.2.1. ESTUDIO CINEMÁTICO

En este caso, al tratarse de un brazo robótico, se ha llevado a cabo tanto el análisis de la cinemática directa como de la cinemática inversa.

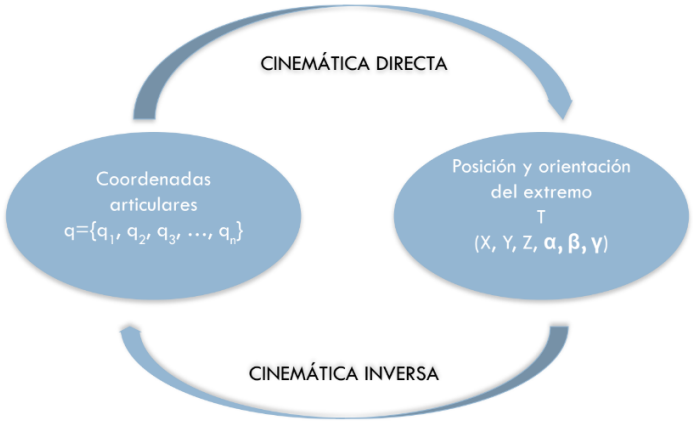


Figura 4-18. Esquema de la cinemática de un brazo robótico

4.2.2.1.1. CINEMÁTICA DIRECTA

Al explorar la cinemática directa de un brazo robótico, el objetivo principal es determinar las coordenadas cartesianas y la orientación del efector final del brazo a partir de las coordenadas articulares de sus articulaciones.

En este trabajo, se ha empleado una metodología basada en las matrices de Denavit-Hartenberg para llevar a cabo el estudio cinemático directo. Esta técnica, desarrollada para describir la geometría de los sistemas robóticos articulados, se fundamenta en una serie de transformaciones homogéneas que relacionan los sistemas de referencia de cada articulación, permitiendo así calcular la posición y orientación del extremo del brazo respecto al sistema de referencia de la base del robot. El análisis se ha realizado utilizando las dimensiones y los sistemas de referencia que se observan en la Figura 4-19.

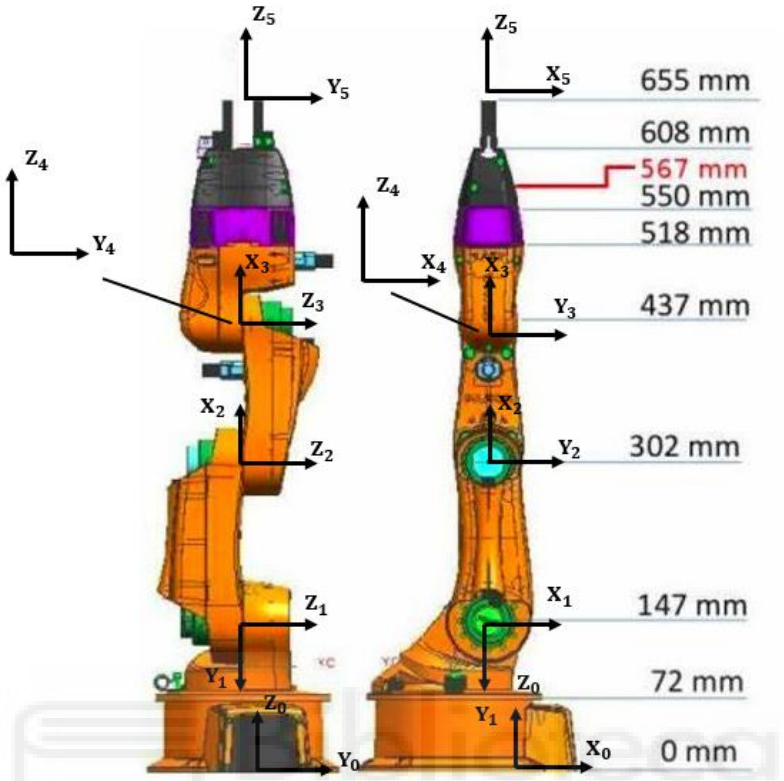


Figura 4-19. Cinemática directa brazo Kuka Youbot

Una vez se conocen los sistemas de referencia, es posible obtener los valores de la tabla de Denavit-Hartenberg, que en este caso se presentan en la *Tabla 1*.

	Θ (º)	**d (m)**	**a (m)**	**α(º)**
$\mathbf{A_1^0}$	q_1	0.147	-0.033	$-\frac{\pi}{2}$
$\mathbf{A_2^1}$	$q_2 + \frac{\pi}{2}$	0	0.155	0
$\mathbf{A_3^2}$	q_3	0	0.135	0
$\mathbf{A_4^3}$	$q_4 + \frac{\pi}{2}$	0	0	$\frac{\pi}{2}$
$\mathbf{A_5^4}$	q_5	0.218	0	0

Tabla 1. Parámetros de Denavit-Hartenberg

Con estos valores, se pueden derivar las matrices homogéneas que definen la relación entre los sistemas de referencia.

- $\mathbf{A_1^0}$:

$$\mathbf{A_1^0} = \begin{pmatrix} \cos q_1 & 0 & -\sin q_1 & -0.033\cos q_1 \\ \sin q_1 & 0 & \cos q_1 & -0.033\sin q_1 \\ 0 & -1 & 0 & 0.147 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{A_2^1}$:

$$\mathbf{A_2^1} = \begin{pmatrix} \cos q_2 + \frac{\pi}{2} & -\sin q_2 + \frac{\pi}{2} & 0 & 0.155\cos q_2 + \frac{\pi}{2} \\ \sin q_2 + \frac{\pi}{2} & \cos q_2 + \frac{\pi}{2} & 0 & 0.155\sin q_2 + \frac{\pi}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Pero sabiendo que:

$$\cos q_2 + \frac{\pi}{2} = -\sin q_2 \;\; \text{y} \;\; \sin q_2 + \frac{\pi}{2} = \cos q_2$$

La matriz A_1^0 se queda:

$$\mathbf{A_2^1} = \begin{pmatrix} -\sin q_2 & -\cos q_2 & 0 & -0.155\sin q_2 \\ \cos q_2 & -\sin q_2 & 0 & 0.155\cos q_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{A_3^2}$:

$$\mathbf{A_3^2} = \begin{pmatrix} \cos q_3 & -\sin q_3 & 0 & 0.135\cos q_3 \\ \sin q_3 & \cos q_3 & 0 & 0.135\sin q_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{A_4^3}$:

$$\mathbf{A_4^3} = \begin{pmatrix} \cos q_4 + \frac{\pi}{2} & 0 & \sin q_2 + \frac{\pi}{2} & 0 \\ \sin q_4 + \frac{\pi}{2} & 0 & -\cos q_2 + \frac{\pi}{2} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Pero sabiendo que:

$$\cos q_4 + \frac{\pi}{2} = -\sin q_4 \;\; \text{y} \;\; \sin q_4 + \frac{\pi}{2} = \cos q_4$$

La matriz $\mathbf{A_1^0}$ se queda:

$$\mathbf{A_4^3} = \begin{pmatrix} -\sin q_4 & 0 & \cos q_4 & 0 \\ \cos q_4 & 0 & \sin q_4 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{A_5^4}$

$$\mathbf{A_5^4} = \begin{pmatrix} \cos q_5 & -\sin q_5 & 0 & 0 \\ \sin q_5 & \cos q_5 & 0 & 0 \\ 0 & 0 & 1 & 0.218 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Finalmente, la transformación total que relaciona el sistema de referencia de la base con el del extremo se obtiene multiplicando todas las matrices de transformación.

$$\mathbf{A_5^0} = \mathbf{A_1^0} * \mathbf{A_2^1} * \mathbf{A_3^2} * \mathbf{A_4^3} * \mathbf{A_5^4}$$

Esta matriz homogénea proporciona información tanto de la posición como de la orientación del extremo del brazo en relación con el sistema de referencia de la base.

$$\mathbf{A_5^0} = \begin{bmatrix} R_5^0 & p_5^0 \\ 0 & 1 \end{bmatrix}$$

4.2.2.1.2. CINEMÁTICA INVERSA

En el caso de la cinemática inversa, el objetivo es determinar las coordenadas articulares a partir de las coordenadas cartesianas y la orientación del efector final con respecto al sistema de referencia de la base.

En este análisis, se puede obtener la información mediante dos métodos: el método algebraico y el método geométrico. En este trabajo, se ha optado por el método geométrico, por lo que se buscará relaciones geométricas entre los segmentos del brazo para obtener la solución. En este trabajo se ha tenido en cuenta una simplificación, considerando la coordenada articular $\mathbf{q_1}$ fija. Es decir:

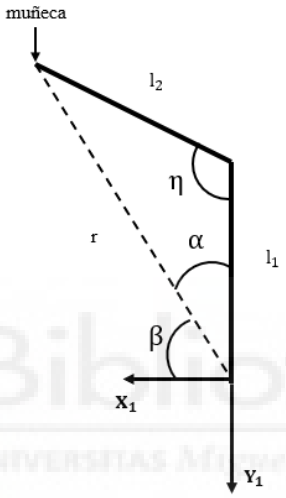
$$\mathbf{q_1} = 0 \qquad (12)$$

Teniendo esto en cuenta, en primer lugar, se ha calculado las coordenadas del punto definido como punto muñeca $\mathbf{p_m}$ utilizando la información proporcionada por la matriz

homogénea que relaciona el sistema de referencia de la base con el sistema de referencia del extremo del robot.

$$\mathbf{A_5^0} = \begin{bmatrix} R_5^0 & p_5^0 \\ 0 & 1 \end{bmatrix} \rightarrow p_m^0 = p_5^0 - (0.218 * z_5^0) \qquad (11)$$

En segundo lugar, y con la información que se ha obtenido de p_m^0 , se pasa a calcular las coordenadas articulares $\mathbf{q_2}$ y $\mathbf{q_3}$. En esta ocasión, habrá dos resultados posibles.



- Codo arriba:

$$\mathbf{q_2} = \frac{\pi}{2} - \beta - \alpha \qquad (13)$$

$$\mathbf{q_3} = \pi - \eta \qquad (14)$$

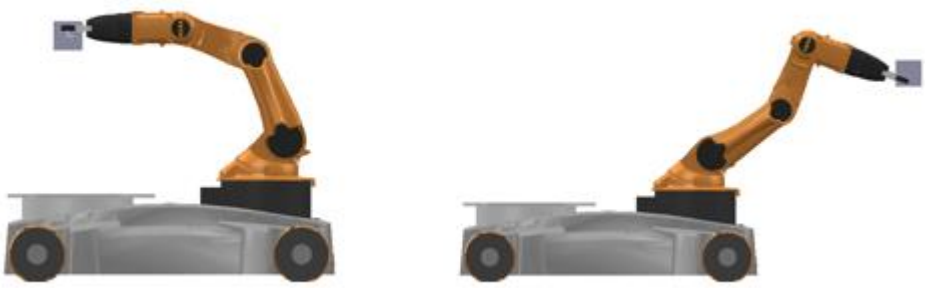


Figura 4-20. Ejemplos del brazo robótico del Kuka Youbot con la configuración 'Codo arriba'

- Codo abajo:

$$\mathbf{q_2} = \frac{\pi}{2} - \beta - \alpha \quad (15)$$

$$\mathbf{q_3} = -\pi + \eta \quad (16)$$

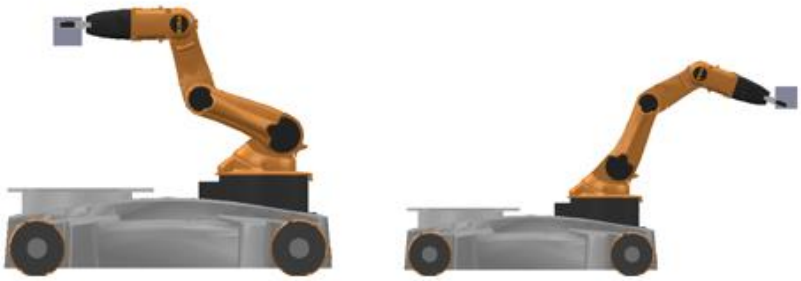


Figura 4-21. Ejemplos del brazo robótico del Kuka Youbot con la configuración 'Codo abajo'

Donde:

$$\alpha = \arccos\left(\frac{-{l_2}^2+{l_1}^2+r^2}{2l_1 r}\right) \quad (17)$$

$$\eta = \arccos\left(\frac{{l_2}^2+{l_1}^2-r^2}{2l_1 l_2}\right) \quad (18)$$

Y a través de una transformación del p_m:

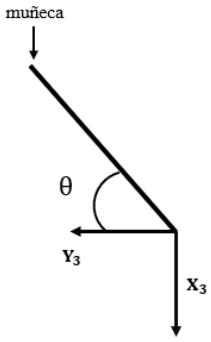
$$p_m^1 = \text{inv}\left(\mathbf{A_1^0}\right) * p_m^0 \quad (19)$$

Se obtiene:

$$r = \sqrt{{p_{m_x}^1}^2 + {p_{m_y}^1}^2} \quad (20)$$

$$\beta = \arctan2(-p_{m_y}^1, p_{m_x}^1) \quad (21)$$

Seguidamente, a través de la posición del extremo del robot, se calcula la coordenada articular $\mathbf{q_4}$.



$$\mathbf{A_3^0} = \mathbf{A_1^0} * \mathbf{A_2^1} * \mathbf{A_3^2}$$

$$p_5^3 = \text{inv}\left(\mathbf{A_3^0}\right) * p_5^0 \qquad (22)$$

$$\mathbf{q_4} = \arctan2(p_{5_y}^3, p_{5_x}^3) \qquad (23)$$

Finalmente, se obtiene la última coordenada articular $\mathbf{q_5}$ a partir de las matrices de transformación D-H.

$$Q = \left(\mathbf{A_4^3}\right)^{-1} * \left(\mathbf{A_3^2}\right)^{-1} * \left(\mathbf{A_2^1}\right)^{-1}\left(\mathbf{A_1^0}\right)^{-1} * \mathbf{A_5^0} = \mathbf{A_5^4}$$

$$\mathbf{q_5} = \arctan2(\sin q_5, \cos q_5) \rightarrow \mathbf{q_5} = \arctan2(Q(0,1), Q(0,0)) \qquad (24)$$

4.2.3. SENSORES INCORPORADOS

El modelo del Kuka Youbot utilizado en CoppeliaSim no incluye sensores en su modelo. Por este motivo, y para recolectar la información necesaria para el trabajo, se han añadido dos sensores al modelo de CoppeliaSim del Kuka Youbot.

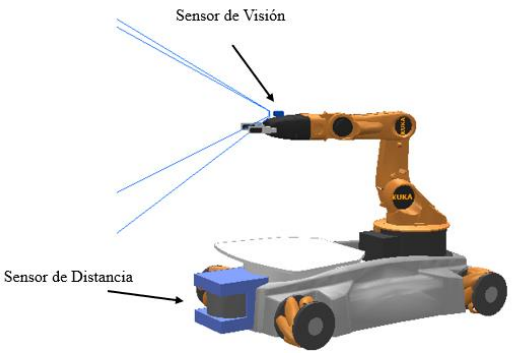


Figura 4-22. Sensores añadidos al Youbot

El primer sensor añadido es el objeto de escena "2D laser scanner", que simula la función de un LiDAR 2D. Este sensor, detallado en el capítulo 2.2.1.2, es fundamental para implementar el algoritmo de localización.

El segundo sensor, se trata de un sensor de visión ubicado en el extremo del brazo para poder realizar la aplicación de visión.

4.3. DESCRIPCIÓN DEL CÓDIGO REALIZADO

Este trabajo se ha llevado a cabo con la intención de simular una aplicación industrial, como el transporte y reparto de paquetes. Para el desarrollo del trabajo se ha utilizado el entorno de simulación CoppeliaSim, en el cual se creó un escenario para replicar una situación de reparto de paquetes en oficinas.

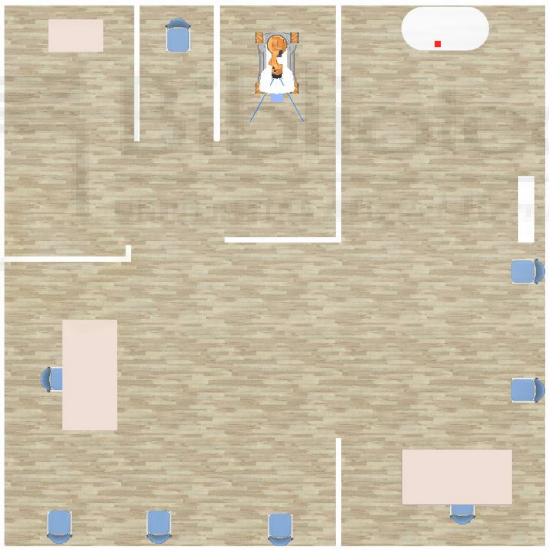


Figura 4-23. Escenario creado en CoppeliaSim para el desarrollo del trabajo

El código desarrollado se ha escrito en el lenguaje de programación Python, utilizando la API remota de CoppeliaSim. Esta API permite controlar una simulación (o el propio simulador) desde una aplicación externa o un hardware remoto (por ejemplo, un robot real, un ordenador remoto, etc.). La API remota de CoppeliaSim se compone de un centenar de funciones específicas y una función genérica.

Figura 4-24. Entorno de simulación y lenguaje empleados en este trabajo

En cuanto al aspecto técnico del código desarrollado para este trabajo, consta de tres partes clave: el algoritmo de localización Monte Carlo, el algoritmo de planificación de trayectorias A*, y la aplicación de visión. Estas tres partes esenciales del código se han integrado, creando un programa robusto y eficiente que permite al robot moverse de manera autónoma a cualquier punto del escenario con cualquier orientación y con la capacidad de recoger y entregar paquetes.

Para comprender el código, es importante conocer la estructura del programa, ya que se ha utilizado la programación orientada a objetos. Este enfoque organiza el código en una serie de clases, cada una de las cuales encapsula una parte crucial del funcionamiento del sistema. A continuación, se detallan algunos aspectos clave de esta estructura:

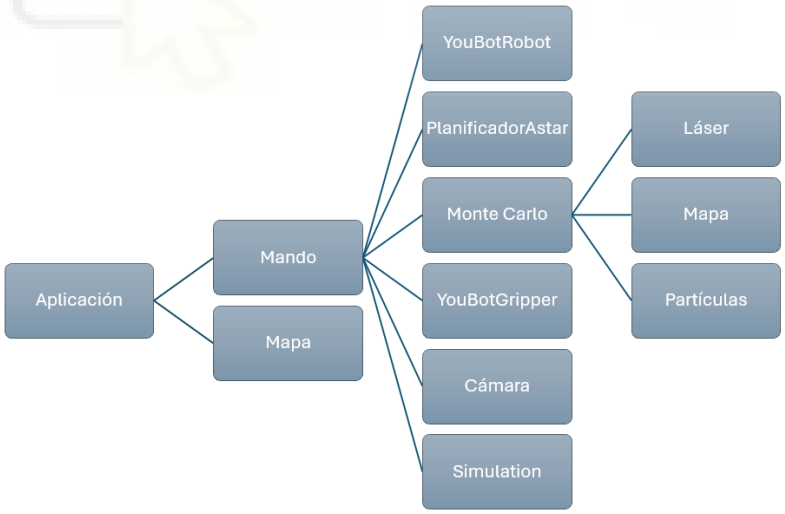


Figura 4-25. Dependencia entre clases que componen el programa

4.3.1. APLICACIÓN

Este primer escalón es la base desde donde se controla todo el funcionamiento del programa, es decir, es donde se indica aquello que queremos que realice el robot en la

simulación, se inicializa la interfaz del programa y se realiza la actualización de los datos que se representan en la interfaz.

Para permitir que el fichero realice diferentes tareas de manera simultánea, se ha empleado un módulo de Python que facilita la creación y gestión de hilos dentro del mismo programa. Este módulo se conoce como:

```
import threading
```

Figura 4-26. Módulo de Python

A partir de este módulo, en el programa se crea un hilo, para la función que lanza la simulación y controla el orden de ejecución del programa.

```
def control():
    global objeto_control
    objeto_control.robot.set_base_speed(0, 0, 0)
    objeto_control.gripper.open(precision=True)
    objeto_control.robot.moveAbsJ(np.array([0, 0, 1.57, 0, 0]))
    objeto_control.control(inicial=[-0.02, -1.925, np.pi / 2], final=[-1.5211, -1.25, np.pi / 2])
    objeto_control.robot.moveAbsJ(np.array([0, -0.6, -1.9, 0.7854, 0]))
    recogida = objeto_control.recolocazion()
    objeto_control.pick(punto=recogida[:3], euler=[3.14159265, -0.78659265, -3.14159265])
    objeto_control.place(punto=[3.24742727e-01, 1.84390698e-17, 8.38671377e-02],
                         euler=[3.14159265, 0.61159265, - 3.14159265])
    objeto_control.robot.moveAbsJ(np.array([0, 0, 1.57, 0, 0]))
    objeto_control.control(inicial=objeto_control.pos, final=[-1.595, 1.225, -np.pi / 2])
    objeto_control.pick(punto=[3.24742727e-01, 1.84390698e-17, 8.38671377e-02], euler=[3.14159265, 0.61159265, -3.14159265])
    objeto_control.place(punto=[-3.41269498e-01,  3.10944503e-18,  3.14218909e-01],
                         euler=[-3.14159265, -1.22174265, -3.14159265])
```

Figura 4-27. Función encargada de la simulación

4.3.2. CLASE MAPA

En esta clase, se encuentran los métodos responsables de la inicialización y actualización de la interfaz. La interfaz creada para este programa consta de cuatro ventanas que muestran diferentes tipos de datos.

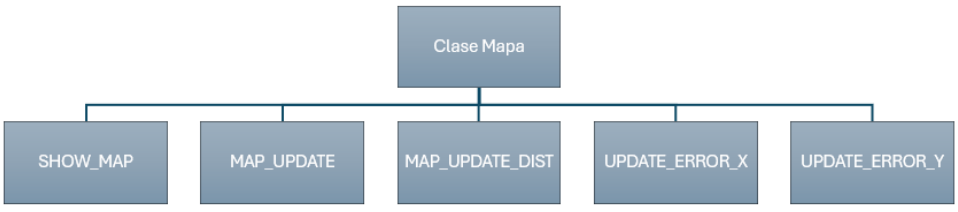


Figura 4-28. Métodos de la clase Mapa

El método **'mostrar_mapa'** se encarga de inicializar la interfaz como se observa en la Figura 4-29.

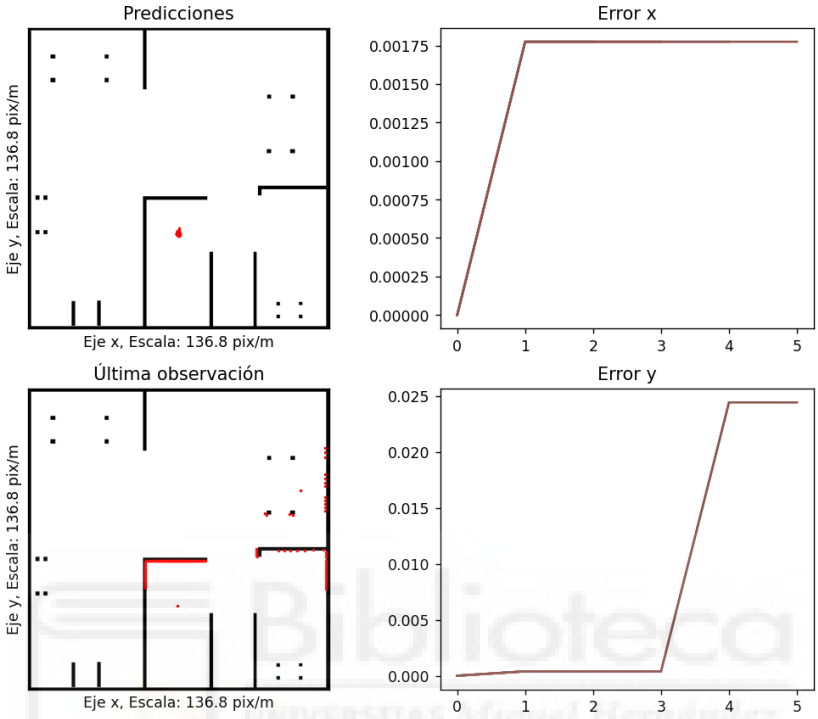


Figura 4-29. Interfaz del programa

El método '**map_update**' se encarga de actualizar los datos que se observan en la ventana de **'predicciones'**. En esta ventana se muestran todas las estimaciones del programa sobre la ubicación del robot, representadas a través de una nube de partículas.

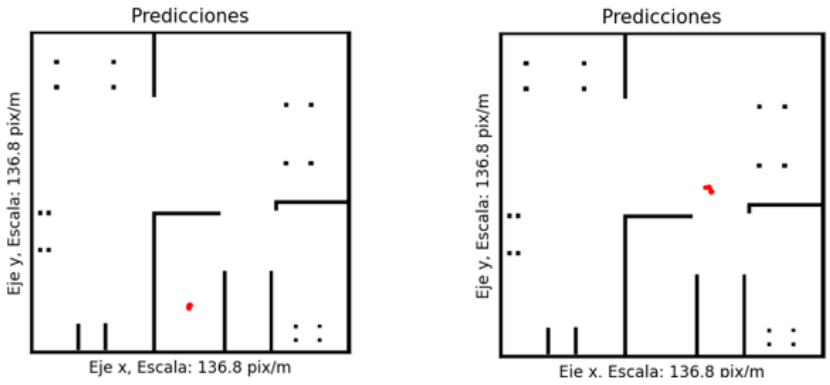


Figura 4-30. Ejemplos de la ventana 'predicciones' de la interfaz

El método **'map_update_dist'** es el responsable de actualizar los datos que se muestran en la ventana **'Última observación'**. En esta ventana se visualizan las distancias obtenidas del LiDAR 2D en función de la última estimación de la posición del robot. Esta vista permite evaluar la precisión del algoritmo de localización al comparar las mediciones con la ubicación estimada del robot.

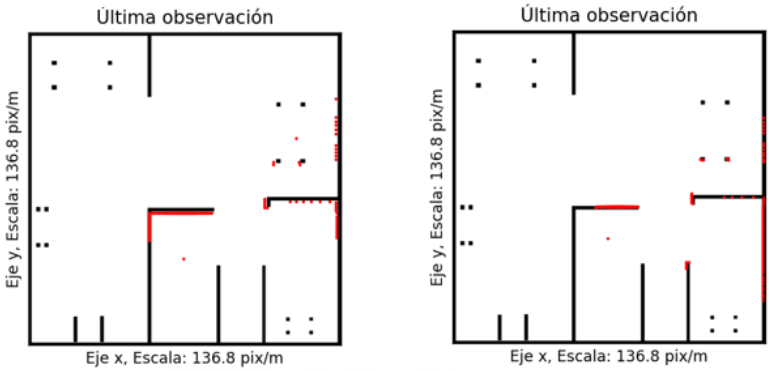


Figura 4-31. Ejemplos de la ventana 'última observación' de la interfaz

Por último, los dos últimos métodos se encargan de actualizar el error de posición que nuestro algoritmo de localización comete en cada instante tanto en el eje X como en el eje Y. Para ello, calculamos la diferencia entre la posición que el entorno de simulación CoppeliaSim indica para el robot y la ubicación predicha por nuestro algoritmo de localización.

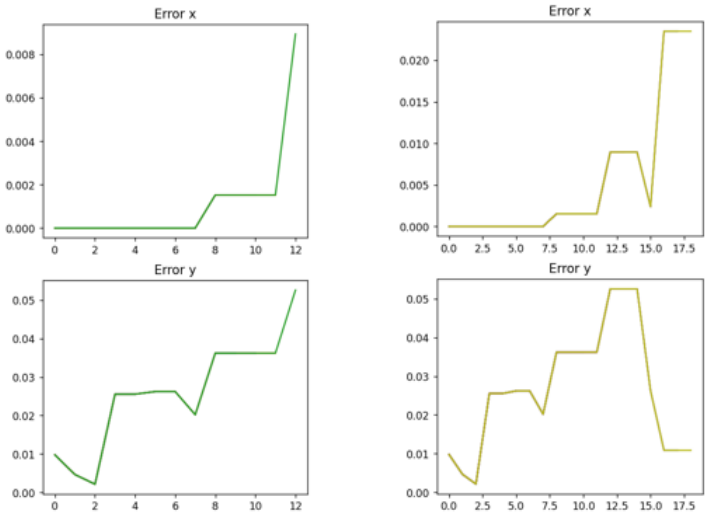


Figura 4-32. Ejemplos de las ventanas de error

4.3.3. CLASE MANDO

Esta clase unifica la mayoría de las clases del programa y sus métodos están pensados para realizar funciones específicas y que puedan ser solicitadas desde el fichero **'aplicación'**.

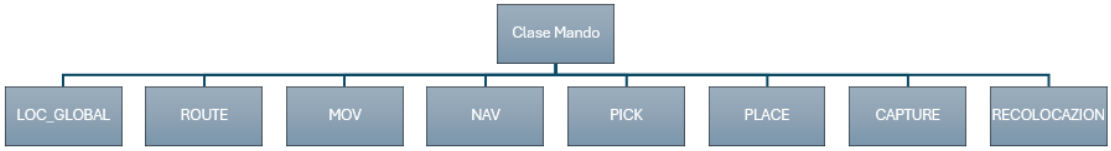


Figura 4-33. Métodos de la clase de 'Mando'

Los cuatro primeros métodos están centrados en la navegación del robot. El primero, **'loc_global'**, realiza la localización global del robot. El segundo método, **'route'**, calcula la ruta óptima en el mapa. El tercero, **'mov'**, gestiona el control de la base durante cada desplazamiento. Finalmente, el método **'nav'** integra los dos anteriores, coordinando la planificación de la ruta y el control de la base en un flujo de ejecución coherente.

```
def route(self, p_inicial, p_final):
    self.a.get_path(start=p_inicial, goal=p_final)
    lista_puntos = self.a.get_path_meters()
    return lista_puntos
```

Figura 4-34. Método que emplea el algoritmo A* para la planificación de la ruta

```
def nav(self, inicial, final, diferencia_error=0.1):
    i = 0
    self.monte.start_particulas(inicial)
    path_global = self.route(inicial[:2], final[:2])
    path_global.append([path_global[-1][0], path_global[-1][1], final[2]])
    n_punt = len(path_global)
    self.pos = self.monte.posicion(motion_inputs=[0, 0, 0])
    self.hab_map_abs = True
    for p in path_global:
        if i < n_punt-1:
            print('Datos PARTICULA')
            self.mov(p, diferencia=diferencia_error)
        else:
            print('Datos PARTICULA')
            self.mov(p, diferencia=0.01)
        i += 1
    self.robot.set_base_speed(0, 0, 0)
```

Figura 4-35, Método de la clase 'Mando' encargado de coordinar la planificación de la ruta y el control del robot

```python
def mov(self, p, diferencia):
    p.append(1)
    i = 0
    while True:
        # pasar
        error = self.a.punto_coordenadas_robot(self.pos, p)
        # controlador P del robot
        vx, vy, wz = self.controlador(error=error)
        self.robot.wait()
        if i <= 30:
            self.pos = self.monte.prediction([vx, vy, wz], 0.05)
            self.hab_map = True
        else:
            self.pos = self.monte.posicion([vx, vy, wz], 0.05)
            self.hab_map_abs = True
            i = 0
        dif = np.array(self.pos) - np.array(p[:3])
        dif = np.linalg.norm(dif)
        i += 1
        if dif <= diferencia:
            break
```

Figura 4-36. Método de control de la base del robot

Los dos métodos siguientes se centran en el movimiento del brazo robótico y el control de la pinza.

```python
def pick(self, punto, euler):
    self.robot.set_base_speed(0, 0, 0)
    # Los valores en el sistema de referencia del robot
    tp1 = Vector(punto)
    to = Euler(euler)  # convención de los angulos de euler es XYZ
    self.gripper.open(precision=True)
    self.robot.moveJ(target_position=tp1, target_orientation=to)
    self.gripper.close(precision=True)
```

Figura 4-37. Método de control de la acción pick del brazo robótico

```python
def place(self, punto, euler,codo=2):
    self.robot.set_base_speed(0, 0, 0)
    # Los valores en el sistema de referencia del robot
    tp1 = Vector(punto)
    to = Euler(euler)  # convención de los angulos de euler es XYZ
    self.robot.moveJ(target_position=tp1, target_orientation=to,codo=codo)
    self.gripper.open(precision=True)
```

Figura 4-38. Método de control de la acción place del brazo robótico

Por último, los métodos **'capture'**, **'recolocación'** están centrados en el manejo de la aplicación de visión.

```
def capture(self):
    self.robot.set_base_speed(0, 0, 0)
    self.robot.wait()
    self.camara.get_image()
    self.camara.get_position()
    return self.camara.pos_aruco
```

Figura 4-39. Método de la clase 'Mando' encargado de la captura de imágenes

4.3.4. CLASE SIMULATION

Esta clase es la encargada de recoger aquellos métodos que establecen algún tipo de interacción con el tiempo de simulación, ya sea la conexión entre nuestros scripts de Python y el entorno de simulación CoppeliaSim, un parada en la comunicación o el final de esta.

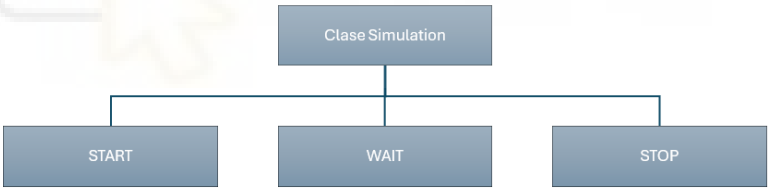


Figura 4-40. Métodos de la clase 'Simulation'

El método **'start'**, encargado de establecer la conexión con CoppeliaSim, utiliza la comunicación vía socket. Esta conexión devuelve un ID, el cual se utiliza en todas las funciones de la API para identificar y manejar la sesión de comunicación activa con el simulador. Este ID es crucial ya que asegura que todas las operaciones y comandos se ejecuten en el contexto correcto de la simulación.

```python
def start(self):
    sim.simxFinish(-1)
    clientID = sim.simxStart('127.0.0.1', 19997, True, True, 5000, 5)
    self.clientID = clientID
    if clientID != -1:
        print("Connected to remote API server")
        # stop previous simiulation
        sim.simxStopSimulation(clientID=clientID, operationMode=sim.simx_opmode_blocking)
        time.sleep(3)
        sim.simxStartSimulation(clientID=clientID, operationMode=sim.simx_opmode_blocking)
        # enable the synchronous mode
        sim.simxSynchronous(clientID=clientID, enable=True)
    else:
        print("Connection not successful")
        sys.exit("Connection failed,program ended!")
    return clientID
```

Figura 4-41. Código de conexión con el entorno de simulación CoppeliaSim

4.3.5. CLASE YOUBOTGRIPPER

Esta clase recoge los métodos encargados de controlar el movimiento de la pinza del brazo robótico.

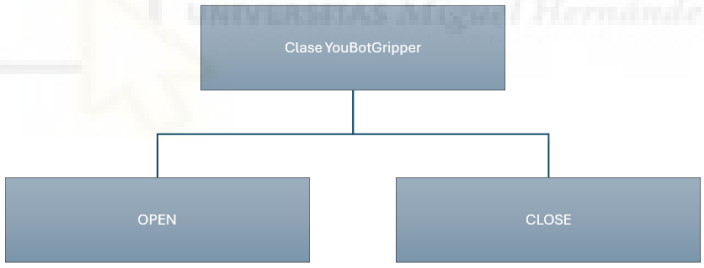


Figura 4-42. Métodos de la clase 'YouBotGripper'

```python
def open(self, precision=False):
    """sim.simxSetJointTargetPosition(clientID=self.clientID, jointHandle=self.joints[0],
                                      targetPosition=-0.05, operationMode=sim.simx_opmode_oneshot)"""
    sim.simxSetJointTargetPosition(clientID=self.clientID, jointHandle=self.joints[1],
                                   targetPosition=-0.05, operationMode=sim.simx_opmode_oneshot)
    if precision:
        for i in range(10):
            sim.simxSynchronousTrigger(clientID=self.clientID)
```

Figura 4-43. Método encargado de abrir la pinza

4.3.6. CLASE CAMERA

Esta clase se encarga de inicializar, capturar y analizar las imágenes capturadas por el sensor de visión instalado en el brazo robótico, como se observa en la Figura *4-22*. Durante el análisis de las imágenes, el método correspondiente utiliza los marcadores ArUco para determinar la posición de los objetos en relación con el sistema de referencia de la cámara. Posteriormente, esta información se utiliza para calcular la posición de los objetos en relación con el sistema de referencia del robot, permitiendo que el robot pueda recoger el objeto con éxito.

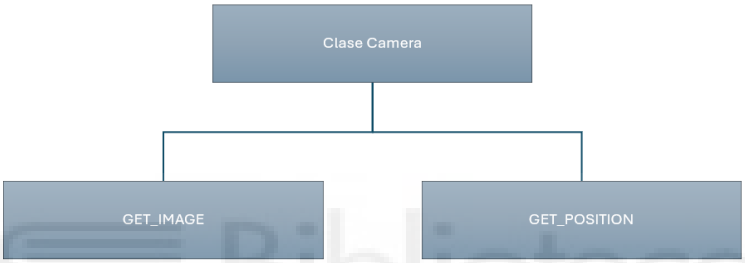


Figura 4-44. Métodos de la clase 'Camera'

Las marcas Aruco son marcadores visuales diseñados específicamente para ser fácilmente detectados por cámaras, lo que las hace ideales para tareas de localización precisa en entornos robóticos. Este enfoque integrado de visión y navegación proporcionará una solución más robusta y precisa para las tareas de localización y manipulación del robot.

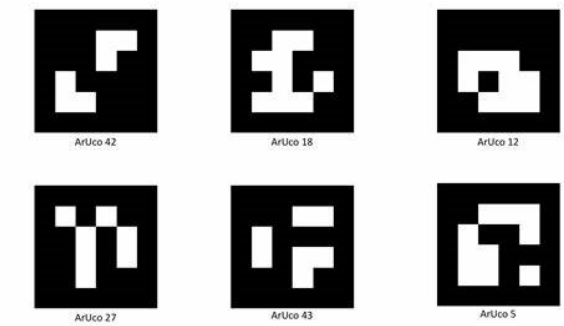


Figura 4-45. Ejemplos de marcadore ArUco

```python
def get_position(self):
    self.pos_aruco = []  # borrar posiciones anteriores
    self.tvecs = []
    gray_image = cv.cvtColor(self.image, cv.COLOR_BGR2GRAY)  # transforms to gray level
    corners, ids, rejectedImgPoints = cv.aruco.detectMarkers(gray_image, self.dictionary)
    # dispimage = cv.aruco.drawDetectedMarkers(self.image, corners, ids, borderColor=(0, 0, 255))

    """# display corner order (Board file)
    for item in corners:
        for i in range(4):
            cv.putText(dispimage, str(i), item[0, i].astype(int), cv.FONT_HERSHEY_DUPLEX, 0.4, (255, 0, 0), 1,
                       cv.LINE_AA)
    """
    # Calculate POSE
    if len(corners) > 0:
        # 75 tamaño del aruco de un lado en mm
        rvecs, tvecs, _objPoints = cv.aruco.estimatePoseSingleMarkers(corners, 50, self.cameraMatrix,
                                                                      self.distCoeffs)
        for tvec in tvecs:
            for t in tvec:
                self.tvecs.append(np.round(t, 3))
        for i,tvec in enumerate(self.tvecs):
            self.tvecs[i] = np.append(tvec * 0.001, 1)
            # print(self.tvecs)
        for tvec in self.tvecs:
            self.pos_aruco.append(tvec)
```

Figura 4-46. Método de la clase 'Camera' para obtener la localización del objeto en relación con el sistema de referencia del sensor de visión

4.3.7. CLASE YOUBOTROBOT

En esta clase se implementan todos los métodos relacionados con el control del robot, abarcando desde el movimiento de la base hasta el cálculo cinemático directo e inverso del brazo. También se incluyen aquellas transformaciones homogéneas que permiten determinar la posición y orientación de los componentes en el espacio. Estas transformaciones homogéneas son matrices que combinan rotaciones y traslaciones, facilitando el cálculo preciso de las coordenadas y ángulos en el sistema de referencia del robot, esencial para tareas como la recogida y colocación de objetos.

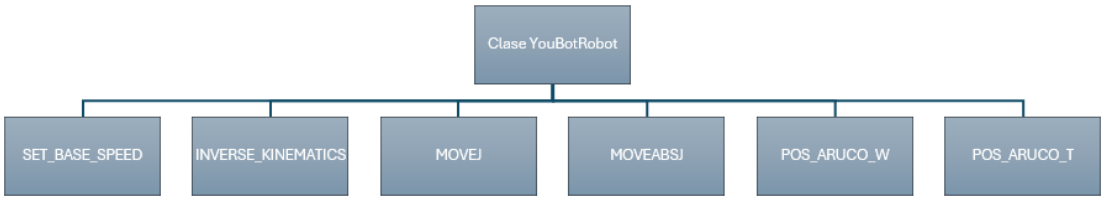


Figura 4-47. Métodos de la clase 'YouBotRobot'

El método **'movej'** es el encargado de recibir la posición y orientación deseadas para el extremo del brazo. A través del método **'inverse_kinematics'**, calcula las coordenadas articulares necesarias y comanda al robot para que alcance las especificaciones indicadas.

```
def moveJ(self, target_position, target_orientation, precision=True):
    """
    Commands the robot to a target position and orientation.
    All solutions to the inverse kinematic problem are computed. The closest solution to the
    current position of the robot q0 is used
        """
    q0 = self.q_current

    self.qs = self.inverse_kinematics(t_position=target_position,
                                      t_orientation=target_orientation)  # almacenamos último movimiento del brazo
    # remove joints out of range and get the closest joint
    self.qs = filter_path(self, q0, [self.qs])
    # comandar al robot hasta
    self.set_joint_target_positions(self.qs, precision=precision)
```

Figura 4-48. Método de la clase 'YouBotRobot' encargado de comandar al brazo robótico a una posición con una orientación específica

El método **'moveabsj'** tiene la misma finalidad que el método **'movej'**. Sin embargo, en este caso, se indican directamente las coordenadas articulares que se desean mandar al brazo robótico.

```
def moveAbsJ(self, q_target, precision=True):
    """
    Commands the robot to the specified joint target positions.
    The targets are filtered and the robot is not commanded whenever a single joint is out of range
    """
    self.qs = q_target  # almacenar último movimineto del brazo
    # remove joints out of range and get the closest joint
    total, partial = self.check_joints(q_target)
    if total:
        self.set_joint_target_positions(q_target, precision=precision)
    else:
        print('moveABSJ ERROR: joints out of range')
```

Figura 4-49. Método de la clase 'YouBotRobot' encargado de posicionar al brazo robótico en unas coordenadas articulares específicas

El método **'set_base_speed'** utiliza la ecuación 20, obtenida del estudio cinemático de la base, para determinar las velocidades correspondientes a cada rueda. Este método

permite comandar las velocidades lineales requeridas para el movimiento deseado del robot.

```
def set_base_speed(self, vx, vy, wz):

    direccion = np.array([vx, vy, wz])  # la velocidad está en m/s
    cinematica = (1 / self.r) * np.array([[1, -1, -(self.b + self.d)],
                                          [1, 1, -(self.b + self.d)],
                                          [1, -1, (self.b + self.d)],
                                          [1, 1, (self.b + self.d)]])
    w = np.dot(cinematica, direccion)

    error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[0],
                                           targetVelocity=w[0],
                                           operationMode=sim.simx_opmode_oneshot)

    error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[1],
                                           targetVelocity=w[1],
                                           operationMode=sim.simx_opmode_oneshot)

    error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[2],
                                           targetVelocity=w[2],
                                           operationMode=sim.simx_opmode_oneshot)

    error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[3],
                                           targetVelocity=w[3],
                                           operationMode=sim.simx_opmode_oneshot)
```

Figura 4-50. Método de la clase 'YouBotRobot' encargado de comandar las velocidades de las ruedas de la base del robot

Por último, los dos métodos restantes son los encargados de realizar las transformaciones homogéneas. El método **'pos_aruco_w'** convierte las coordenadas de un punto en el sistema de referencia de la cámara a las coordenadas en el sistema de referencia del entorno de simulación y el método **'pos_aruco_t' c**onvierte las coordenadas de un punto en el sistema de referencia del entorno de simulación a las coordenadas en el sistema de referencia de la base del brazo robótico.

```
def pos_aruco_t(self, punto_w, pos_robot):
    w_a_r = np.array([[np.cos(pos_robot[2]), -np.sin(pos_robot[2]), 0, pos_robot[0]],
                      [np.sin(pos_robot[2]), np.cos(pos_robot[2]), 0, pos_robot[1]],
                      [0, 0, 1, 0],
                      [0, 0, 0, 1]])

    cero_a_w = np.linalg.inv(self.r_A_cero) @ np.linalg.inv(w_a_r)

    pieza = cero_a_w @ punto_w

    return pieza
```

Figura 4-51. Método de la clase 'YouBotRobot' encargado de realizar una de las transformaciones homogéneas del programa.

4.3.8. CLASE PLANIFICADORASTAR

Esta clase es una de las partes clave de este trabajo, ya que es la que implementa el algoritmo de planificación de trayectorias A*. Los métodos de esta clase tienen diversas funciones, todas ellas dirigidas al funcionamiento eficiente del algoritmo A*.

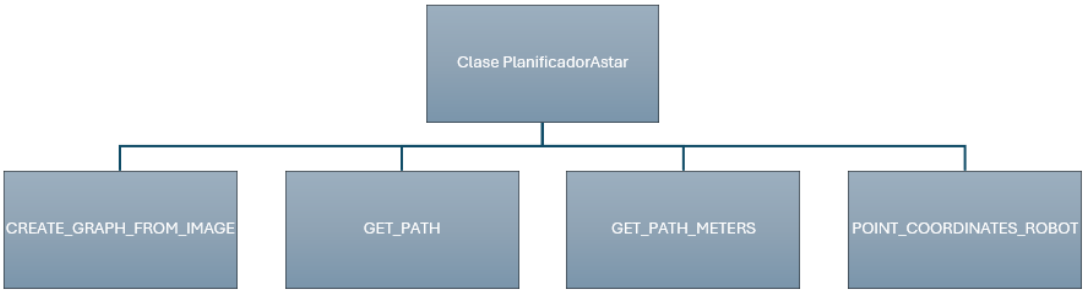


Figura 4-52. Métodos de la clase 'PlanificadorAstar'

El método **'create_graph_from_image'** se encarga de transformar la imagen en un grafo con nodos, facilitando así la aplicación del algoritmo A*. Utiliza la librería **'networkx'** para generar el grafo, la cual simplifica la creación, manipulación y análisis de estructuras complejas de grafos.

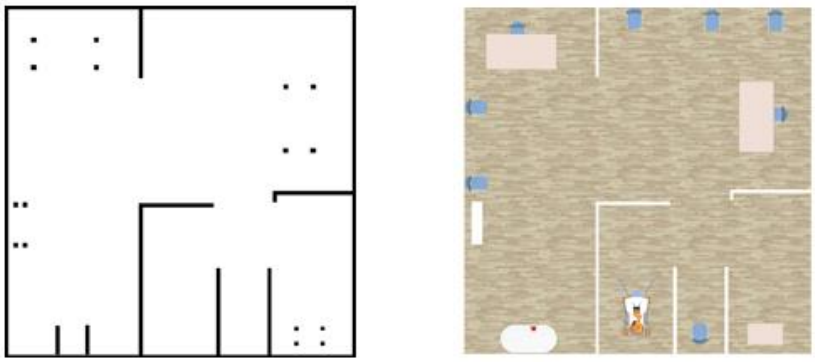


Figura 4-53. Binarización del escenario de CoppeliaSim para el funcionamiento de los algoritmos A* y Localización Monte Carlo

Además, en este método, se emplean operaciones morfológicas en la imagen que se ve en la Figura *4-51*, tales como la dilatación. Estas operaciones aseguran que la ruta creada

sea segura, evitando que el robot choque con los muros, ya que cada píxel blanco en la imagen se considera un nodo accesible para el algoritmo.

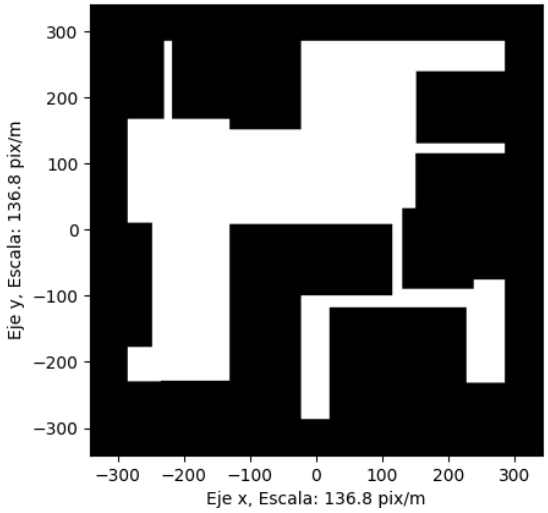


Figura 4-54. Imagen binaria después de aplicar la operación de dilatación morfológica

Una vez obtenido el grafo, se invoca el método **'get_path'** especificando el punto de partida y llegada. Este método devuelve el camino óptimo a través del mapa. Luego, el camino obtenido se pasa al método **'get_path_meters'** para convertirlo a coordenadas del sistema de referencia del entorno de simulación.

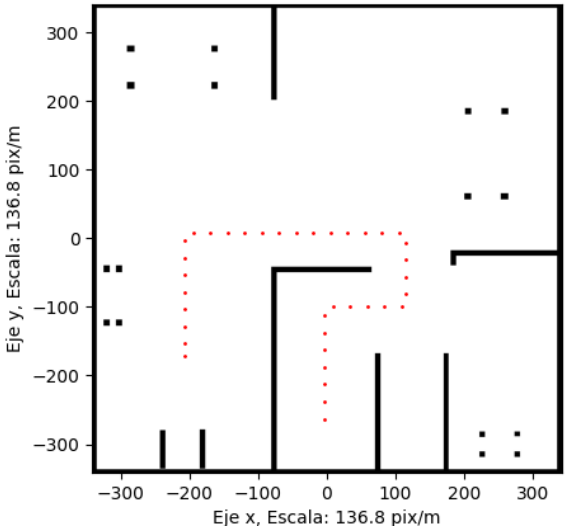


Figura 4-55. Ejemplo de ruta creada por la clase 'Planificador'

```python
def get_path(self, start, goal):
    p_ini = [0, 0]
    p_fin = [0, 0]
    mapa = cv.imread(self.imagen)
    mapa = ndimage.rotate(mapa, 180)
    pix_height, pix_width, _ = mapa.shape
    self.rel_mp_ancho = pix_width / 5.0
    self.rel_mp_alto = pix_height / 5.0
    # Utilizar A* para encontrar la ruta óptima desde
    # Transformo las coord. de coppelia en las del ma
    p_ini[0] = start[0] + 2.5
    p_ini[1] = start[1] + -2.5
    p_ini[0] = p_ini[0] * self.rel_mp_ancho
    p_ini[1] = p_ini[1] * self.rel_mp_alto
    p_ini[0] = abs(round(p_ini[0]))
    p_ini[1] = abs(round(p_ini[1]))
    p_ini = (p_ini[0], p_ini[1])

    p_fin[0] = goal[0] + 2.5
    p_fin[1] = goal[1] + -2.5
    p_fin[0] = p_fin[0] * self.rel_mp_ancho
    p_fin[1] =p_fin[1] * self.rel_mp_alto
    p_fin[0] = abs(round(p_fin[0]))
    p_fin[1] = abs(round(p_fin[1]))
    p_fin = (p_fin[0], p_fin[1])
    self.path = nx.astar_path(self.g, p_ini, p_fin)
```

Figura 4-56. Método de la clase 'PlanificadorAstar' encargado de buscar la ruta óptima

Finalmente, el método **'point_coordinates_robot'** se utiliza para trasladar cada punto del sistema de referencia del entorno de simulación al sistema de referencia del robot. Esto permite controlar el robot utilizando el método **'mov'** de la clase 'Mando', completando así el proceso de planificación y ejecución del movimiento del robot.

4.3.9. CLASE MONTECARLO

En esta clase se encuentran los métodos que implementan las distintas etapas del algoritmo de localización Monte Carlo. Esta clase se encarga de manejar la localización probabilística del robot, permitiéndole determinar su posición dentro del entorno. Estos métodos trabajan en conjunto con la clase **'Laser'** y la clase **'Particulas'**.

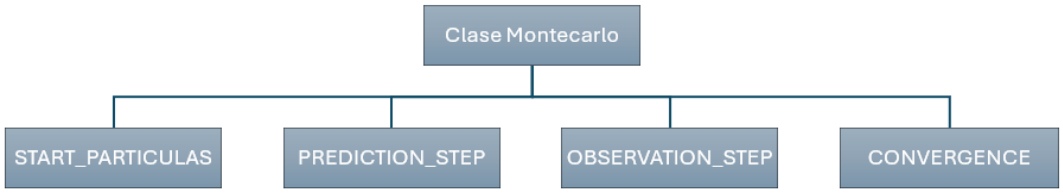


Figura 4-57. Métodos de la clase 'Montecarlo'

En cuanto al método **'convergence'**, no depende de ningún otro método y se utiliza durante la localización global. Este método calcula la convergencia entre los parámetros de las partículas (x, y, theta) para determinar cuándo el algoritmo ha encontrado una posición segura del robot. El método evalúa si la desviación estándar de las coordenadas x, y y theta de las partículas está por debajo de un umbral predefinido, lo que indica que las partículas se han agrupado alrededor de una posición específica.

```
def convergence(self,threshold):
    x_coords = [p[0] for p in self.particulas]
    y_coords = [p[1] for p in self.particulas]
    theta_coords = [p[2] for p in self.particulas]
    std_x = np.std(x_coords)
    std_y = np.std(y_coords)
    std_theta = np.std(theta_coords)
    print(std_y,std_theta,std_x)
    return std_x < threshold and std_y < threshold and std_theta < threshold
```

Figura 4-58. Método de la clase 'Montecarlo' para calcular la convergencia de las diferentes partículas

4.3.10. CLASE LASER

La clase **'Laser'** se utiliza para obtener las lecturas de distancia del entorno, las cuales son cruciales para la actualización de las creencias sobre la posición del robot.

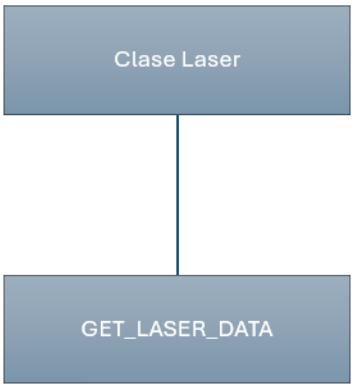


Figura 4-59. Método de la clase 'Laser'

4.3.11. CLASE PARTICULAS

La clase **‘Particulas’** es fundamental en este trabajo, ya que implementa los métodos que permiten el funcionamiento del algoritmo de localización Monte Carlo. Los métodos de esta clase incluyen la inicialización de partículas, la actualización de la creencia sobre la posición del robot basándose en las observaciones y movimientos, y la estimación de la posición más probable. Además, incorpora la técnica de resampling para mantener la diversidad de partículas y evitar la degeneración, asegurando así la precisión y robustez del algoritmo. Con estas funcionalidades, la clase **‘Monetcarlo’** proporciona una solución eficiente y confiable para la localización en entornos dinámicos y desconocidos.

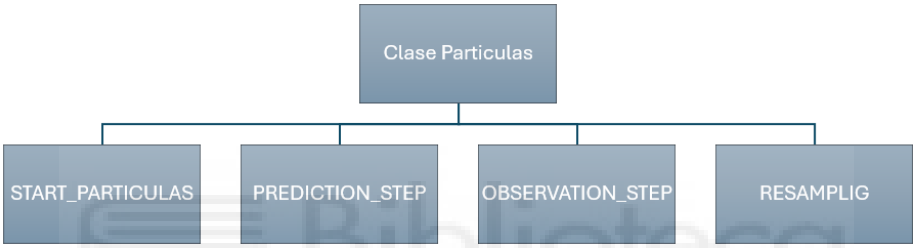


Figura 4-60. Métodos de la clase 'Particulas'

El método **‘start_particulas’**, como su nombre indica, se encarga de inicializar el conjunto de partículas que se usarán para estimar la posición del robot. Dependiendo de cómo se haya configurado el objeto de la clase, las partículas pueden inicializarse de manera global, distribuyéndose uniformemente por todo el espacio, o, si se conoce la ubicación inicial del robot, pueden concentrarse alrededor de ese punto específico. Esta flexibilidad permite adaptar el algoritmo tanto a situaciones en las que no se tiene información previa sobre la posición del robot como a aquellas en las que se dispone de una estimación inicial precisa.

```python
def start_particulas(self, p_inicical, numero,loc):
    # numero de particulas
    # limites de la escena
    particulas = []
    for i in range(numero):
        if loc==True:
            # Si se quisiera realizar una prueba de localización sin conocimiento de donde se encuentra el robot
            x_max, x_min = 2.5, -2.5
            y_max, y_min = 2.5, -2.5
            theta_min, theta_max = 0, 2*np.pi

            x = random.uniform(x_min, x_max)
            y = random.uniform(y_min, y_max)
            theta = random.uniform(theta_min, theta_max)
            particulas.append([x, y, theta])
        else:
            particulas.append(p_inicical)
    return particulas
```

Figura 4-61. Método de la clase 'Particulas' para iniciar conjunto de partículas

El método **‘prediction_step’** es el encargado de realizar el paso de predicción en el algoritmo de localización Monte Carlo. Este método actualiza las partículas basándose en el movimiento del robot, que puede incluir traslaciones y rotaciones. Utilizando el modelo de movimiento del robot, **’prediction_step’** calcula las nuevas posiciones de las partículas, incorporando tanto el control de movimiento como la incertidumbre asociada a dicho movimiento.

```python
def prediction_step(self, motion_inputs, particulas, segundos):

    for i in range(len(particulas)):
        for m in range(len(motion_inputs)-1):
            if self.contador < 10:
                motion_inputs[m] = motion_inputs[m] + random.gauss(0, 0.015)  # mejor 0.015
                self.contador+=1
            else:
                motion_inputs[m] = motion_inputs[m] + random.gauss(0, 0.025)
        # Para caudrar tanto
        coordenadas = particulas[i]
        x_i = coordenadas[0] + segundos * (
                    motion_inputs[0] * np.cos(coordenadas[2]) - np.sin(coordenadas[2]) * motion_inputs[1])
        y_i = coordenadas[1] + segundos*(
                    motion_inputs[1] * np.cos(coordenadas[2]) + np.sin(coordenadas[2]) * motion_inputs[0])
        if self.contador < 10:
            motion_inputs[2] = motion_inputs[2] + random.gauss(0, 0.01)
            self.contador += 1
        else:
            motion_inputs[2] = motion_inputs[2] + random.gauss(0, 0.015)
            self.contador = 0
        theta = coordenadas[2] + segundos * motion_inputs[2]
        particulas[i] = [x_i, y_i, theta]
    return particulas
```

Figura 4-62. Método de la clase 'Particulas' que realiza la actualización de las partículas

El método **'observation_step'** es el encargado de realizar el paso de actualización en el algoritmo de localización Monte Carlo, ya que se encarga de actualizar los pesos de las partículas basándose en las observaciones del entorno. Este método evalúa la probabilidad de que cada partícula represente la posición real del robot al comparar las observaciones actuales con el mapa del entorno.

En detalle, la función **'observation_step'** opera de la siguiente manera: primero, el robot realiza una observación del entorno utilizando un sensor ubicado en su base. Luego, para cada partícula, se calcula qué observaciones se recibirían si el robot estuviera realmente en la posición de esa partícula. En este caso, se simplifica el proceso: se verifica si hay un muro a una distancia de hasta 10 mm del valor real. Si no se detecta nada, se añade una distancia fija de 5 metros.

```
def cal_recta(angle, particula, mapa, obs, conversion):

    d = np.linspace(obs-0.05, obs + 0.05, 7)

    for di in d:
        i = particula[0] + (di * np.cos(particula[2] + angle))
        j = particula[1] + (di * np.sin(particula[2] + angle))
        # Per a canviar la posició del punt (0,0) a la esquina
        i = i + 2.5
        j = j - 2.5
        i = i * conversion[0]
        j = j * conversion[1]
        i = abs(round(i))
        j = abs(round(j))
        # primero indicamos el alto y luego el ancho del pixel
        try:
            if mapa[j, i] == 0:
                return di
        except IndexError:
            pass
    return 5.0
```

Figura 4-63. Función empleada para el cálculo de las observaciones para cada partícula

Por último, estas observaciones se comparan con las observaciones reales para calcular un peso de probabilidad para cada partícula. Aquellas cuyas observaciones simuladas se

ajustan más cercanamente a las observaciones reales reciben mayores pesos, lo cual indica una mayor probabilidad de ser la posición correcta del robot. Este cálculo se realiza utilizando una expresión que evalúa la diferencia cuadrática entre la distancia observada, **'dist'**, y la distancia simulada, **'dist_simu'**. Esta diferencia se ajusta mediante una distribución exponencial y un término constante, que suaviza y penaliza las discrepancias significativas, asegurando así una estimación de la probabilidad de cada partícula.

```
def observation_step(self, obs_dist, particulas, mapa, conversion):

    part = []

    list_w = []
    # mover las particulas a la posición del sensor teniendo en cuenta el sist. de referencia de coppelia
    # ya que las medidas estan tomadas en la posicion del sensor
    for particula in particulas:
        x, y, z = particula
        x = x + (0.3 * np.cos(z))
        y = y + (np.sin(z) * 0.3)
        coord = [x, y, z]
        part.append(coord)
    # indicar la cantidad de rayos que tiene el sensor de proximidad
    angles = np.linspace(-np.pi/2, np.pi/2, 181)
    for particula in part:
        w_i = 1.0
        for j in range(len(angles)):
            dist = obs_dist[j]
            angle = angles[j]
            dist_simu = cal_recta(angle=angle, particula=particula, mapa=mapa, obs=dist, conversion=conversion)
            p_j = np.exp(-(np.power(dist - dist_simu, 2) / 0.1)) + 0.15
            w_i *= p_j
        list_w.append(w_i)
    norm_w = np.array(list_w)
    # Para conecer el peso máx en cada instante y observar como funciona el algoritmo de localización
    print('peso más grande: ', norm_w[np.argmax(norm_w)])
    suma = np.sum(norm_w)
    if suma > 0.0:
        norm_w = norm_w/suma
    return norm_w
```

Figura 4-64.Método de la clase 'Particulas' encargado de realizar el paso de actualización de pesos

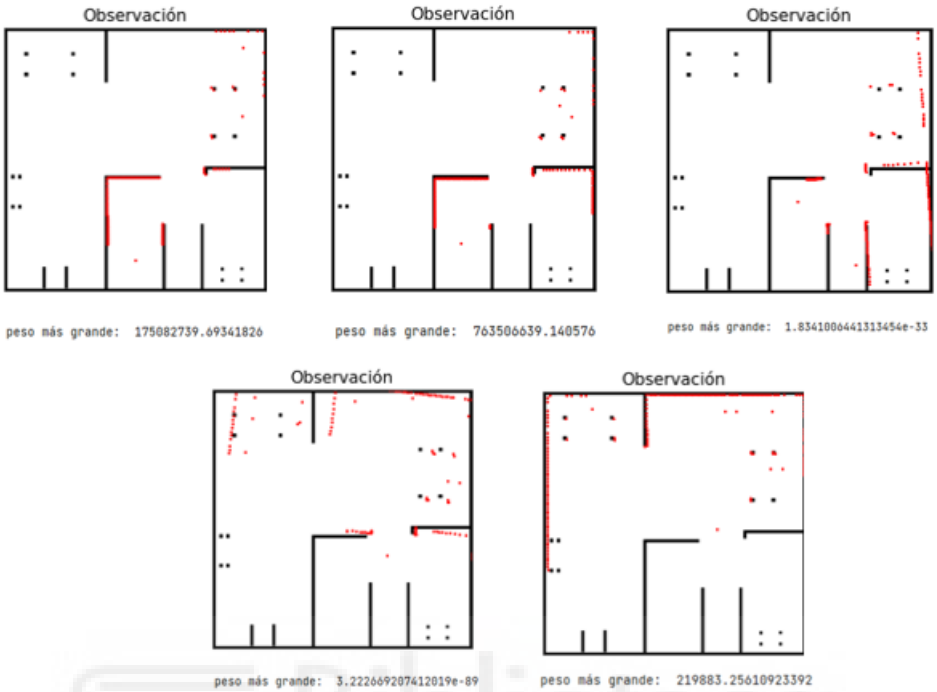


Figura 4-65. Ejemplo del método 'observation_step'

Por último, el método **'resampling'** es fundamental en el algoritmo de localización Monte Carlo para mantener la diversidad y la precisión del conjunto de partículas. Este método selecciona partículas basándose en sus pesos normalizados, permitiendo que las partículas con mayores pesos (más probables) se dupliquen mientras que las de menor peso se descarten.

```
def resampling(self, norm_w, particulas):
    nuevas_part = []
    while True:
        for i in range(len(particulas)):
            w_random = random.uniform(0, 1)
            if norm_w[i] >= w_random:
                nuevas_part.append(particulas[i])
            if len(nuevas_part) == len(particulas):
                return nuevas_part
```

Figura 4-66. Método de la clase 'Partículas' encargado de realizar el método 'resampling'

5. PRUEBAS REALIZADAS

Una vez presentada la estructura del código y explicados los distintos algoritmos empleados, es el momento de revisar los resultados obtenidos. En términos de implementación de los algoritmos de planificación, como A* y el de localización Monte Carlo, se observan dos versiones principales: localización local y localización global. Ambas versiones comparten un objetivo común, pero difieren en el método de inicio del proceso de localización.

Además, se presentará una implementación adicional que incluirá una aplicación de visión utilizando marcas Aruco. Esta implementación mostrará la utilidad de los Aruco para controlar la aproximación del robot al objeto, así como la recogida de este. Este enfoque integrado de visión y navegación proporcionará una solución para las tareas de localización y manipulación del robot.

5.1. LOCALIZACIÓN LOCAL

En esta versión del programa, la posición inicial es conocida, lo que simplifica el problema de la localización. Al partir de un punto conocido, el programa puede mantener al robot controlado de manera efectiva.

Para implementar esta versión, se buscó el conjunto de partículas óptimo que permitiera tener un control real del robot. Es decir, se buscó un conjunto que permitiera llevar el robot a cualquier punto del mapa y que este llegara al destino con cierta aproximación. Para ello, se realizaron una serie de pruebas con 1, 10, 100 y 1000 partículas, comprobando la posición final del robot en CoppeliaSim. De esta manera, se pudo tener en cuenta tanto el error del algoritmo como el tiempo que este tardaba en realizar la misma ruta indicada en la Figura 5-1.

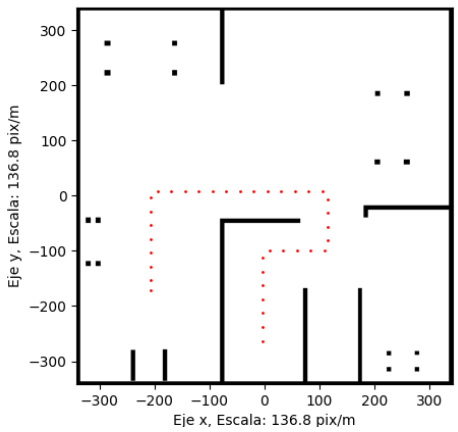
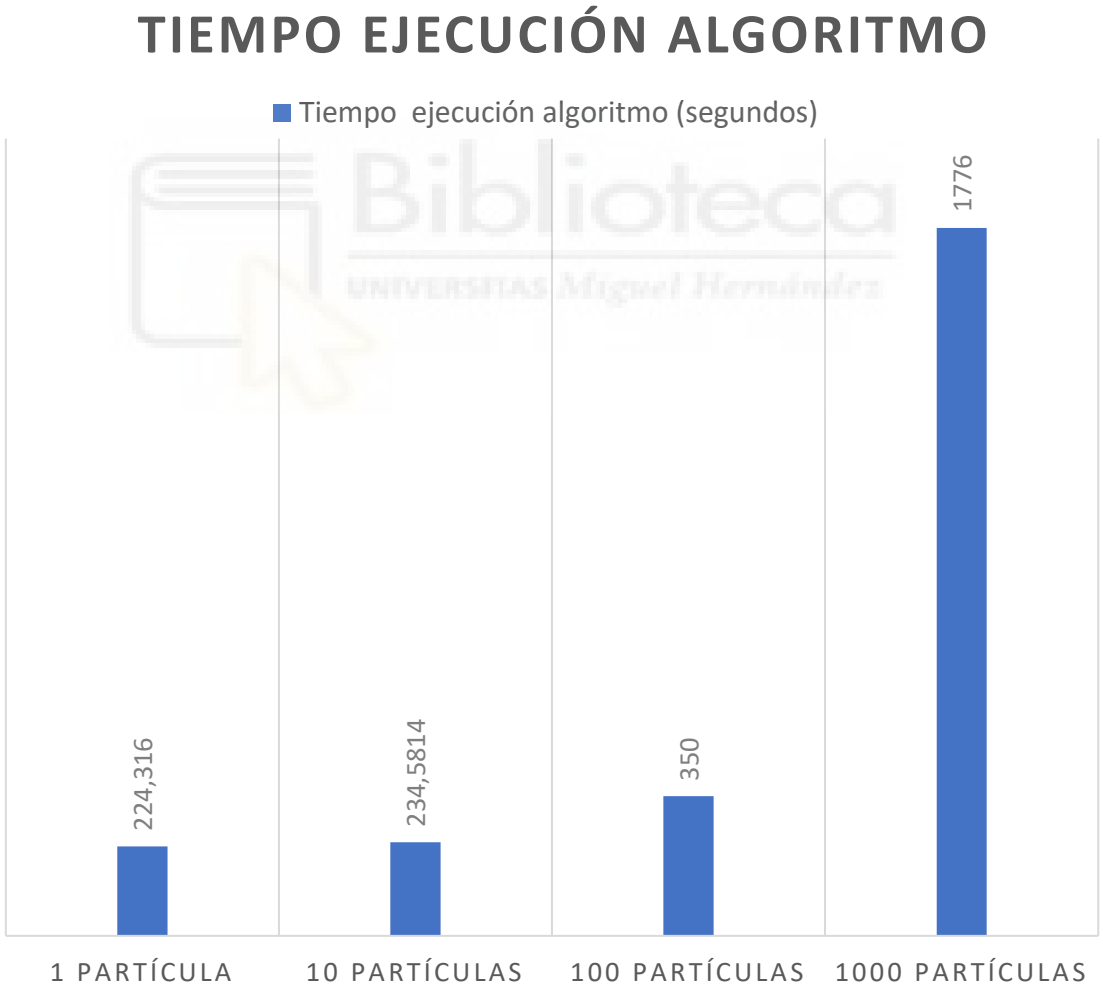
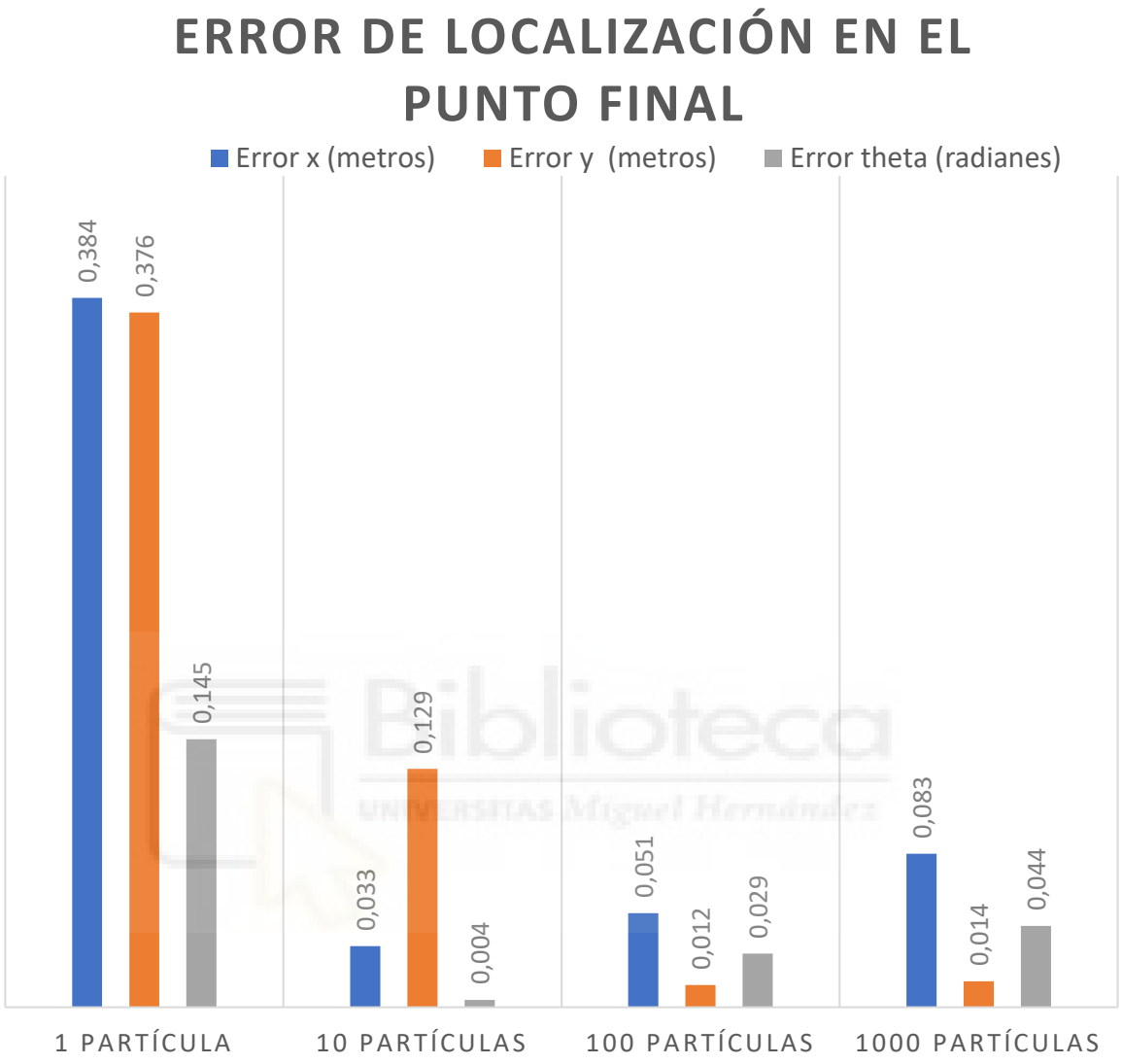


Figura 5-1. Ruta empleada para buscar conjunto de partículas óptimo



Gráfica 1. Tiempo que tarda el programa en realizar la aplicación



Gráfica 2. Error del robot en la posición final de la ruta

Los resultados obtenidos, como se muestran en los gráficos adjuntos, indican que, al aumentar la cantidad de partículas, se mejora la precisión en la localización del robot, pero a costa de un mayor tiempo de ejecución del algoritmo. Con 1 y 10 partículas, el error de localización es considerablemente alto, lo que demuestra una baja precisión. En cambio, al usar 100 y 1000 partículas, el error disminuye significativamente.

Sin embargo, el tiempo de ejecución del algoritmo se incrementa notablemente al aumentar la cantidad de partículas. Con 1000 partículas, el tiempo de ejecución alcanza

los 1776 segundos, comparado con los 234.5814 segundos para 10 partículas y 350 segundos para 100 partículas.

En conclusión, el uso de 100 partículas ofrece un balance óptimo entre precisión y tiempo de ejecución, permitiendo un control efectivo del robot sin un aumento desproporcionado del tiempo de procesamiento.

Es por ello por lo que, a la hora de inicializar el programa, en este trabajo se ha decidido usar un conjunto de 200 partículas. Para ello, en primer lugar, se debe crear un objeto de la clase Mando, especificando la cantidad de partículas que utilizará el filtro de partículas y el tipo de localización deseada (en este caso, local). Además, se debe crear un objeto de la clase Mapa para inicializar la interfaz.

```
if __name__ == "__main__":
    objeto_control = Mando(type_loc=False,n_part=200)
    objeto_control.start()
    update_mapa = Mapa(imagen='Mapa.jpg')
    update_mapa.start()
    mostrar_figura()
```

Figura 5-2. Configuración para iniciar el programa con una localización local

Una vez inicializado el programa, se lanza el hilo que pondrá en funcionamiento la función **'control'**, encargada de dirigir el funcionamiento del robot. Con la simulación en marcha y la interfaz creada, se debe indicar el punto inicial, que ya conocemos, y el punto final al que queremos trasladar al robot. Es fundamental asegurarse de que ambos puntos sean accesibles para el robot, ya que el algoritmo A* tomará en cuenta la Figura 4-54 para trazar la ruta.

```
r1 = threading.Thread(target=control_glob,args=())
r1.start()
```

Figura 5-3

```
def control():
    global objeto_control
    objeto_control.robot.set_base_speed(0, 0, 0)
    objeto_control.gripper.open(precision=True)
    objeto_control.robot.moveAbsJ(np.array([0, 0, 1.57, 0, 0]))
    objeto_control.nav(inicial=[-1.52, -1.5, np.pi / 2], final=[1, 0.9, np.pi / 2])
```

Figura 5-4

Establecida la ruta a seguir, en la interfaz se puede observar cómo las partículas se agrupan en un mismo punto, coincidiendo con el punto inicial indicado. Adicionalmente, se muestran las observaciones recogidas por el sensor en la parte inferior de la pantalla.

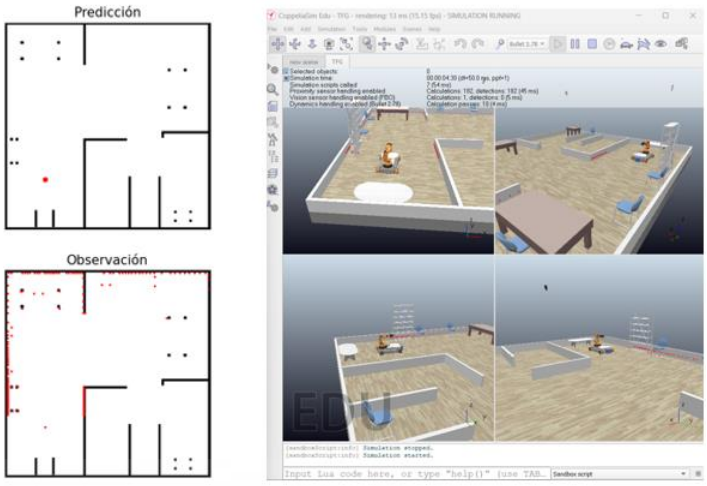


Figura 5-5

A medida que el robot se desplaza por el escenario, el programa actualiza la posición de la nube de partículas de acuerdo con el modelo de movimiento, reagrupándolas cada vez que se realiza una actualización.

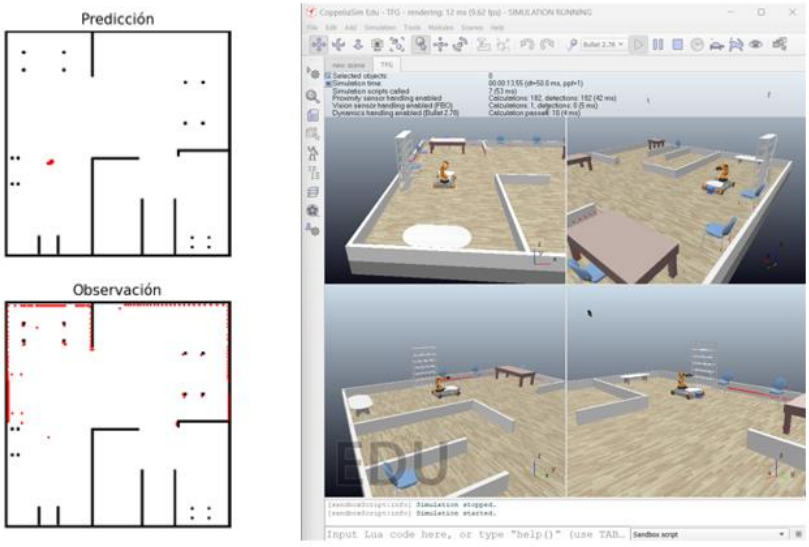


Figura 5-6

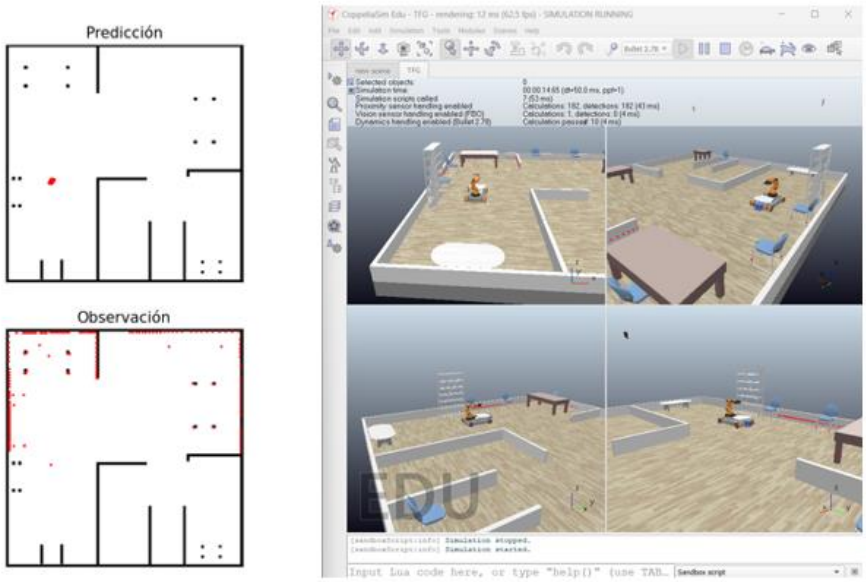


Figura 5-7

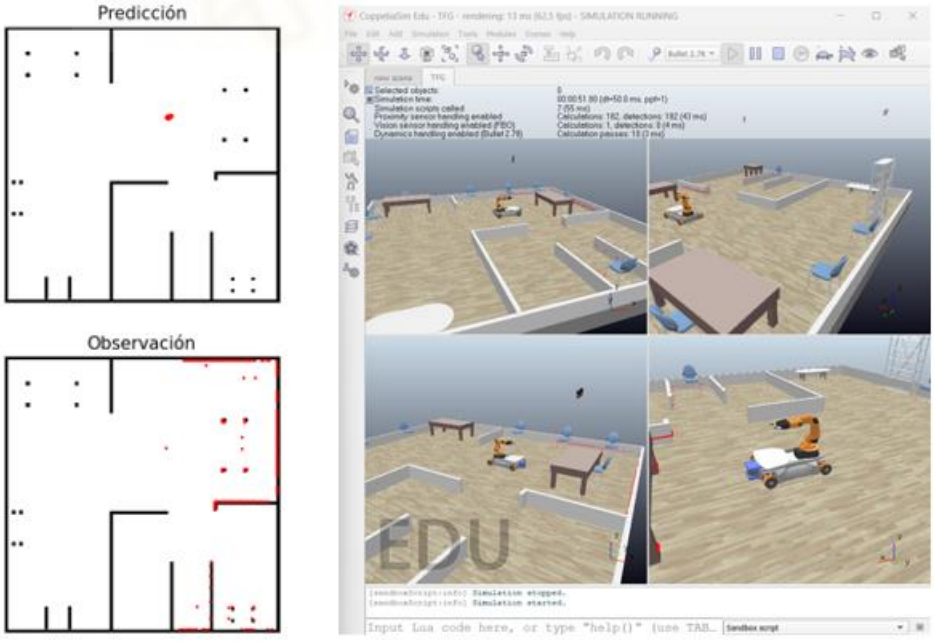


Figura 5-8

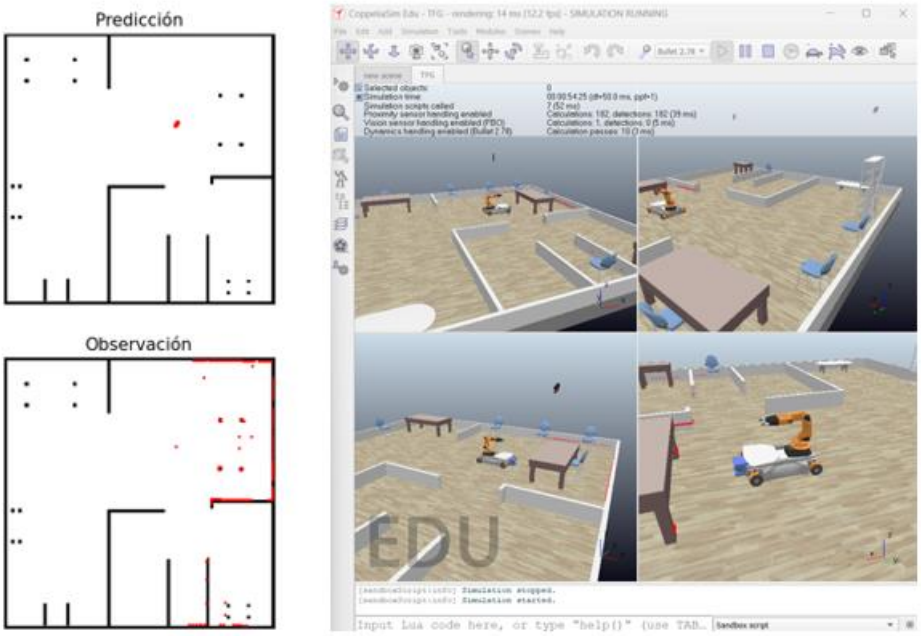


Figura 5-9

Además, durante la simulación, es posible monitorear los errores en la localización del robot. Esto se logra comparando la posición estimada por el algoritmo con la posición real obtenida desde CoppeliaSim.

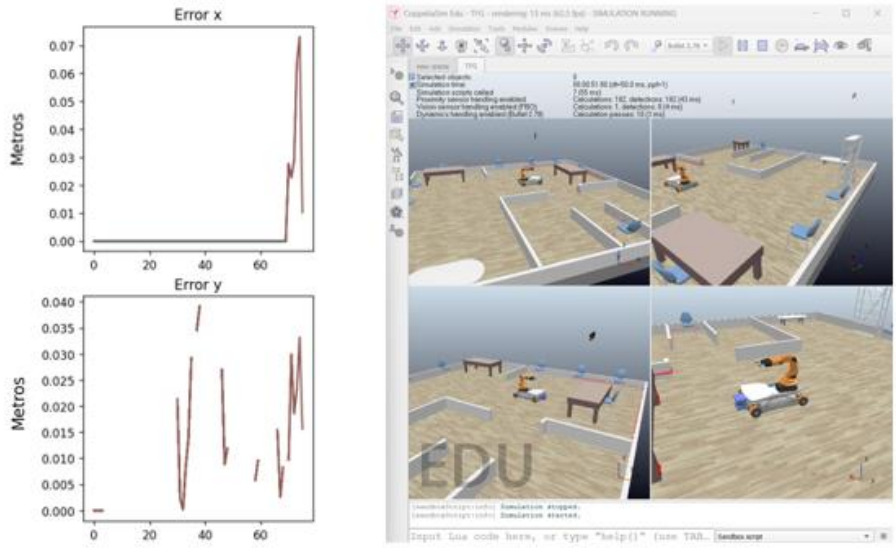


Figura 5-10

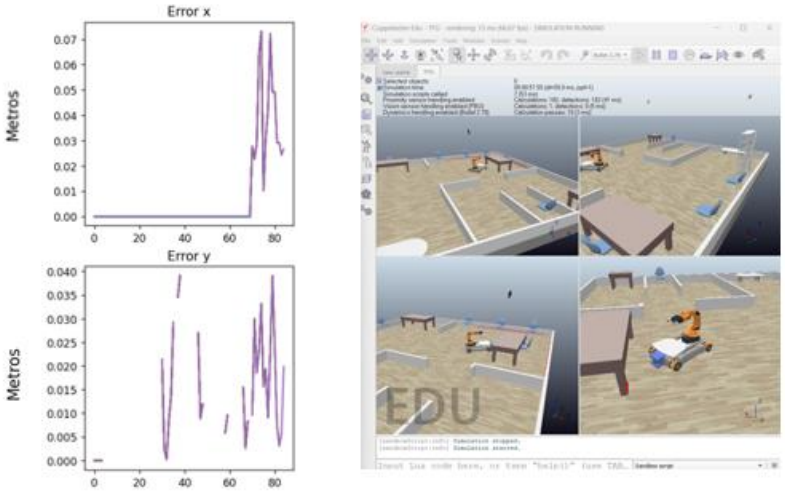


Figura 5-11

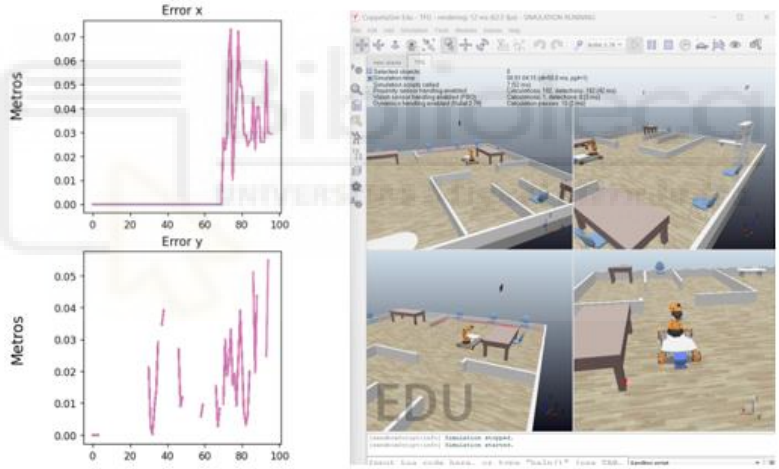


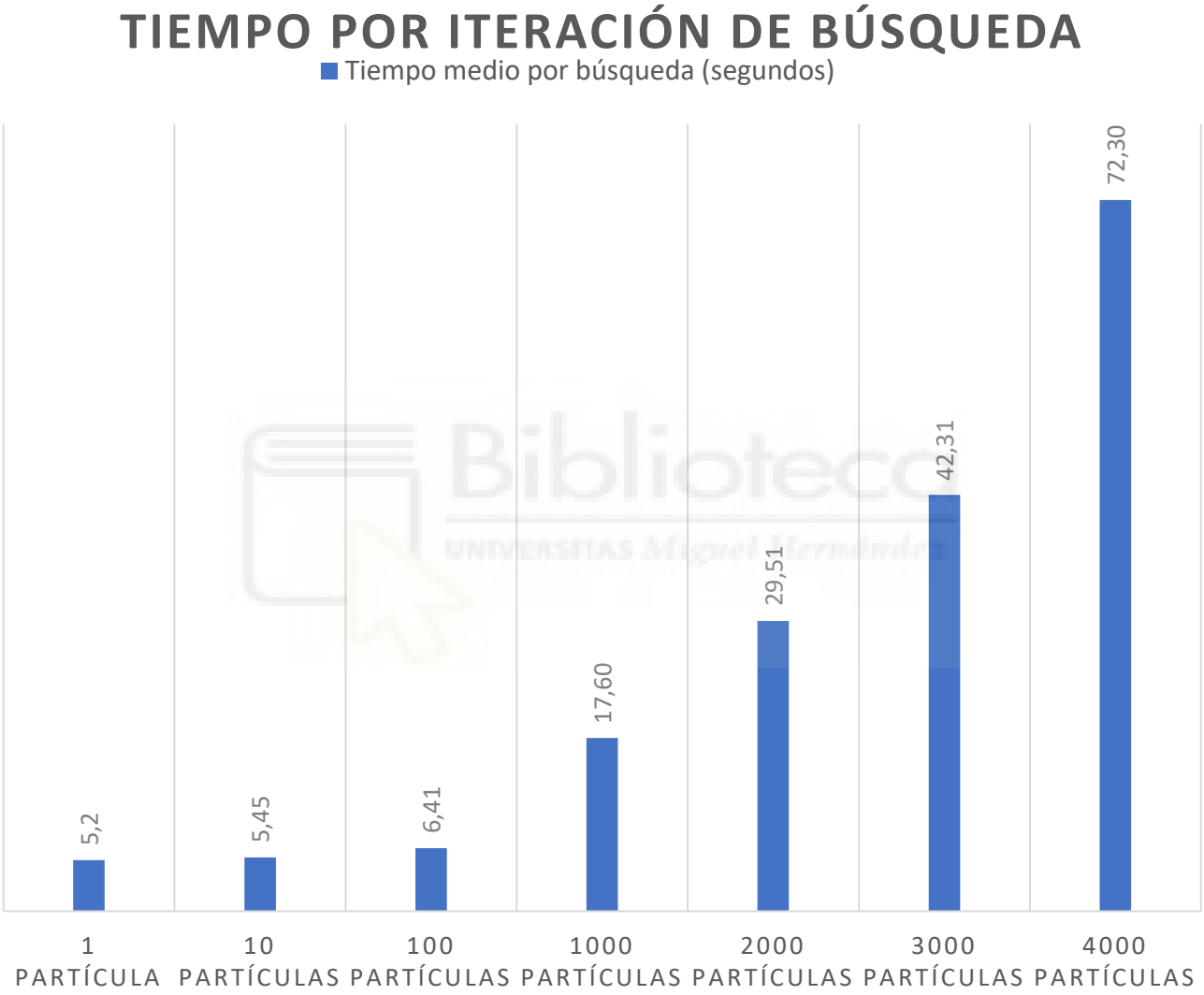
Figura 5-12

5.2. LOCALIZACIÓN GLOBAL

En esta versión del programa, la posición del robot al inicio no es conocida. Por lo tanto, al iniciar el programa, se debe realizar un proceso de localización para determinar la posición del robot. Para ello, al crear el objeto de la clase Mando, se especifica una cantidad mayor de partículas y se cambia el tipo de localización a global.

En esta ocasión, decidir el número de partículas con las que se inicia el algoritmo de localización es crucial, ya que de esto dependerá la precisión en la detección de la

posición del robot. En este trabajo, se han realizado pruebas con diversos números de partículas para determinar la configuración óptima. Además, se evaluaron dos ubicaciones distintas para analizar cómo el entorno influye en la cantidad de partículas necesarias para una localización precisa.



Gráfica 3. Tiempo medio que tarda el algoritmo en realizar una iteración de búsqueda

En primer lugar, este gráfico muestra un incremento significativo en el tiempo de búsqueda a medida que aumenta el número de partículas. Este incremento muestra un compromiso esperado entre precisión y velocidad: más partículas permiten una localización más precisa, pero también incrementan el tiempo necesario para completar

cada búsqueda. Por ello, al elegir un conjunto de partículas, es crucial considerar el tiempo, ya que un número excesivo de partículas puede ralentizar considerablemente el algoritmo.

Una vez atendido este aspecto, nos centramos en el funcionamiento del algoritmo en las dos ubicaciones y cómo varía la tasa de éxito.

- Primera ubicación:

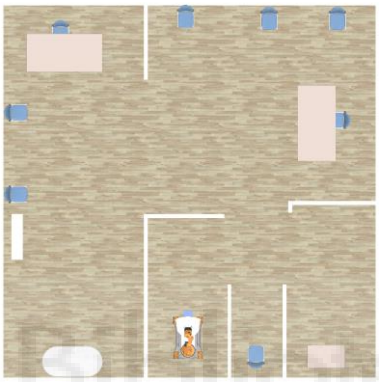
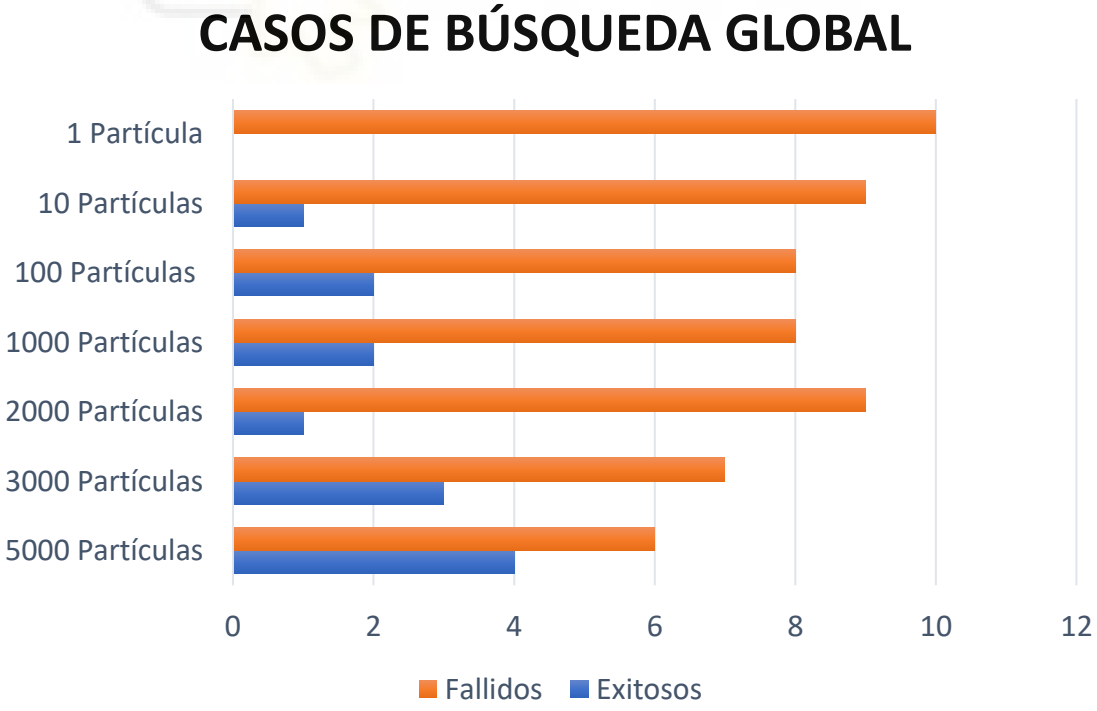


Figura 5-13. Ubicación del robot en la primera prueba de localización global



Gráfica 4. Tasa de éxito en la primera ubicación

En esta primera ubicación, el algoritmo presenta una tasa de éxito baja en los casos de búsqueda global, especialmente cuando el número de partículas es reducido. A medida que se incrementa el número de partículas, la tasa de éxito mejora, pero no de manera significativa, lo que indica que la complejidad de la ubicación desafía al algoritmo incluso con un mayor número de partículas.

- Segunda ubicación:

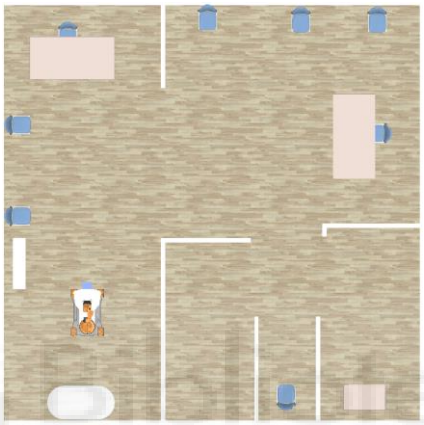
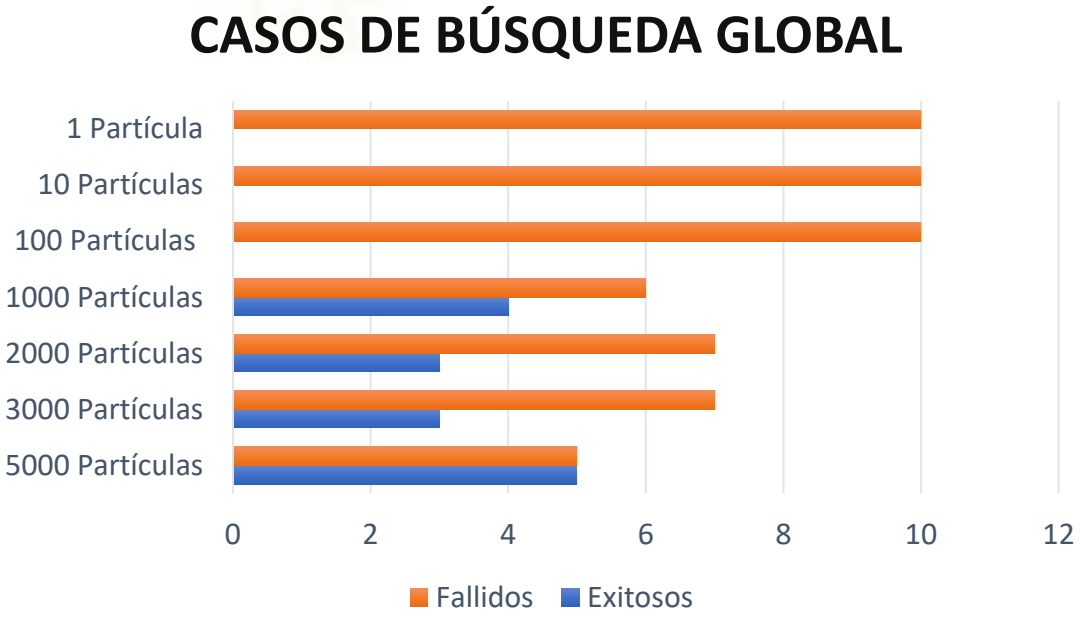


Figura 5-14. Ubicación del robot en la segunda prueba de localización global



Gráfica 5. Tasa de éxito en la segunda ubicación

En comparación, en esta segunda ubicación, el algoritmo muestra un mejor rendimiento en los casos con mayor número de partículas. Sin embargo, los resultados, aunque mejores, todavía dejan margen para mejoras significativas. Aumentar el número de partículas en este entorno podría optimizar aún más la tasa de éxito.

Estas observaciones subrayan la importancia de la complejidad del entorno en la eficiencia del algoritmo de localización. Es por ello que al seleccionar el número óptimo de partículas se debe considerar tanto el entorno específico en el que operará el robot como el balance entre precisión y tiempo de ejecución. En ambos casos, los resultados indican que aumentar el número de partículas sería una medida necesaria para lograr una localización más precisa y confiable.

Para implementar esta versión del programa y llevar a cabo estas pruebas, como se mencionó anteriormente, al crear el objeto de control se debe especificar un mayor número de partículas y cambiar el tipo de localización a global.

```
objeto_control = Mando(type_loc=True,n_part=3000)
objeto_control.start()
update_mapa = Mapa(imagen='Mapa.jpg')
update_mapa.start()
mostrar_figura()

r1 = threading.Thread(target=control())


r1.start()
```

Figura 5-15

```
def control():
    objeto_control.robot.set_base_speed(0, 0, 0)
    objeto_control.robot.moveAbsJ(np.array([0, 0, 1.57, 0, 0]))
    objeto_control.loc_glob(threshold=0.01)
    objeto_control.nav(inicial=objeto_control.pos, final=[1, 0.9, np.pi / 2])
```

Figura 5-16

Durante el proceso de localización, se pueden observar como las partículas distribuidas por el mapa se desplazan en sincronía con el movimiento del robot. A continuación, se presentan ejemplos de situaciones observadas durante las pruebas.

- **1000 partículas**:

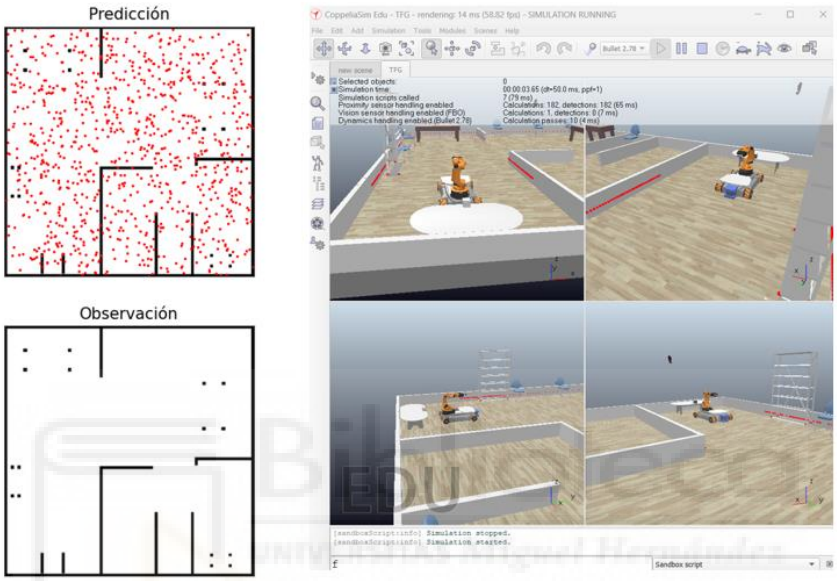


Figura 5-17

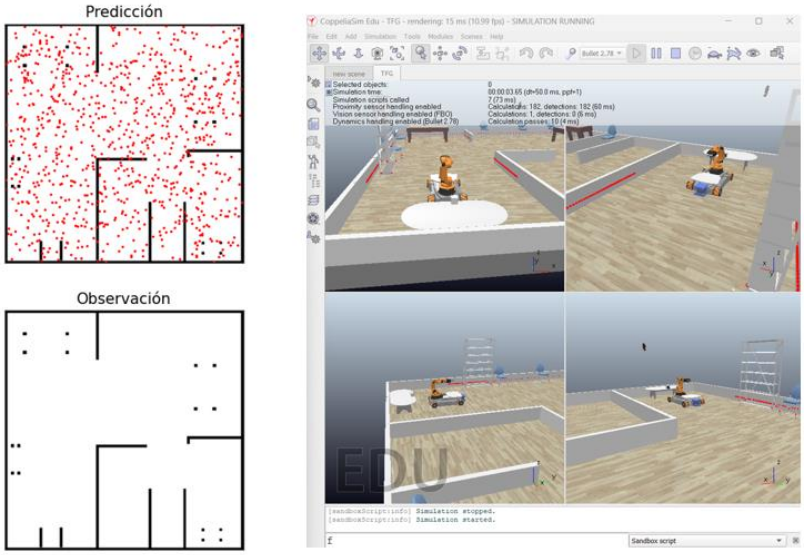


Figura 5-18

En este primer intento, se aprecia que la cantidad de partículas ha sido insuficiente, ya que el algoritmo ha convergido en un punto erróneo, como se puede observar en las siguientes imágenes:

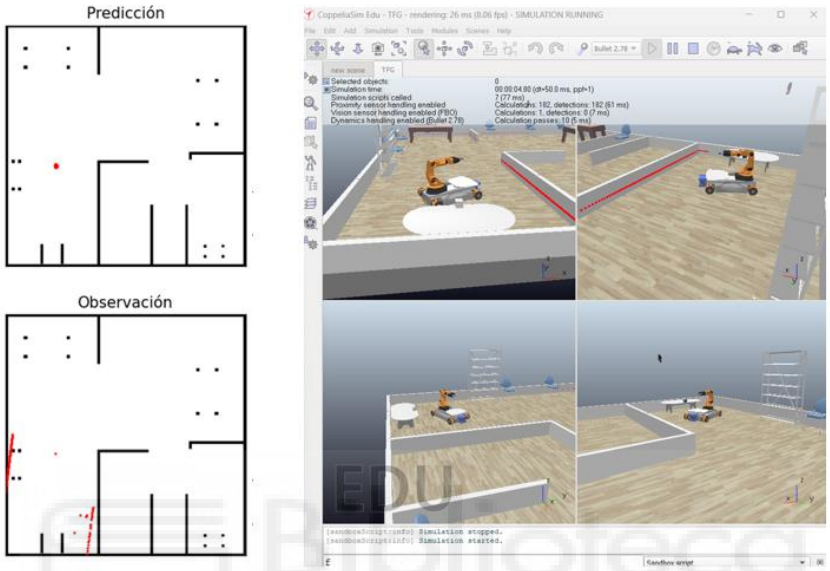


Figura 5-19

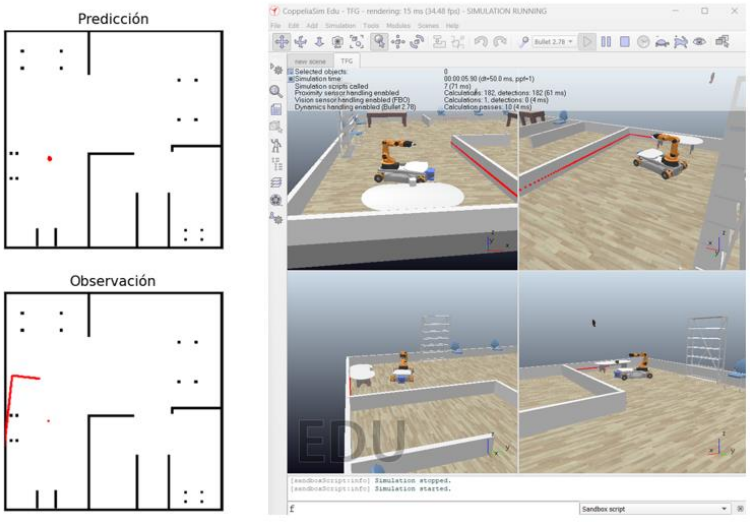


Figura 5-20

- **2000 partículas**:

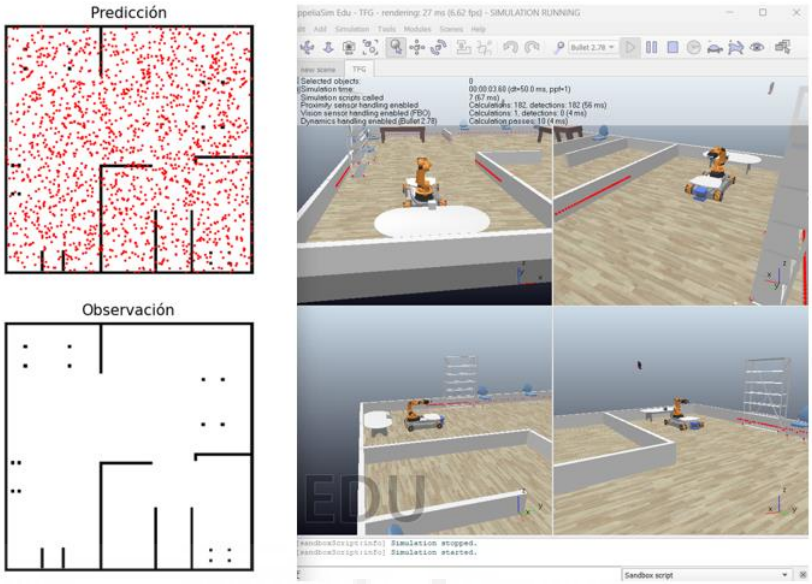


Figura 5-21

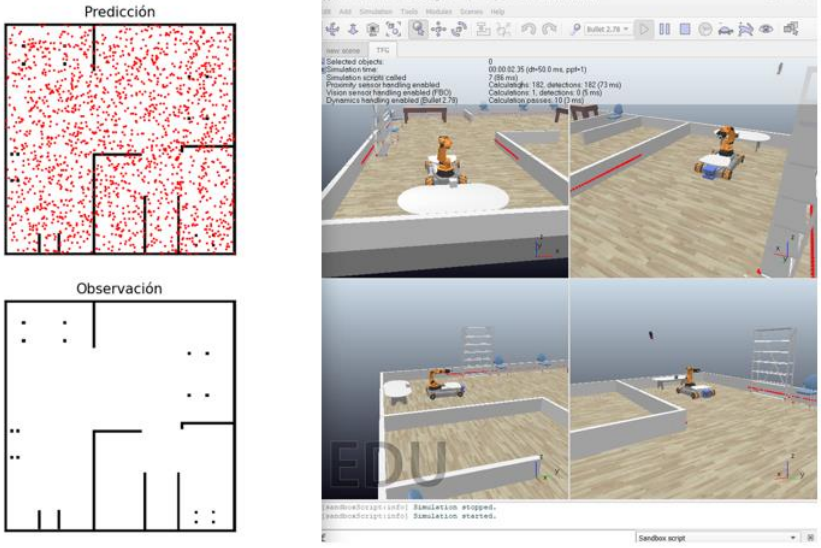


Figura 5-22

En este segundo intento, el número de partículas sigue siendo insuficiente. Como se muestra en la Figura 5-23, el algoritmo continúa divergiendo hacia una localización incorrecta del robot, lo que hace imposible continuar con el programa.

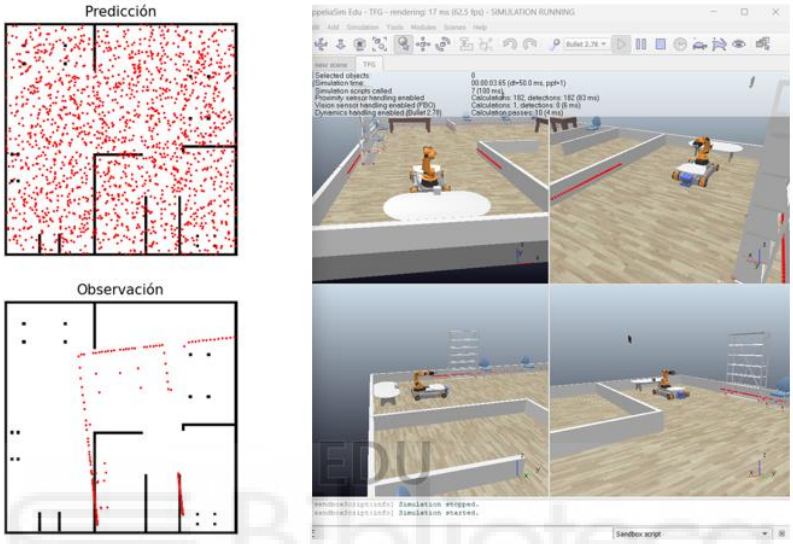


Figura 5-23

- **3000 partículas**:

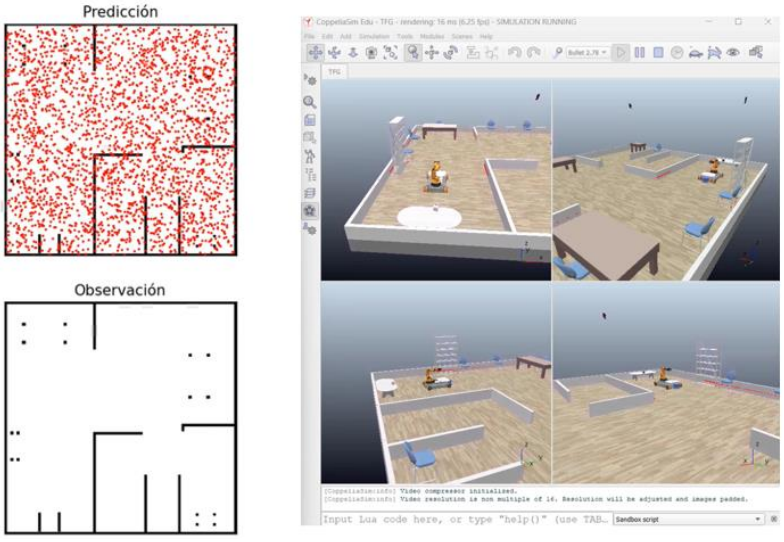


Figura 5-24

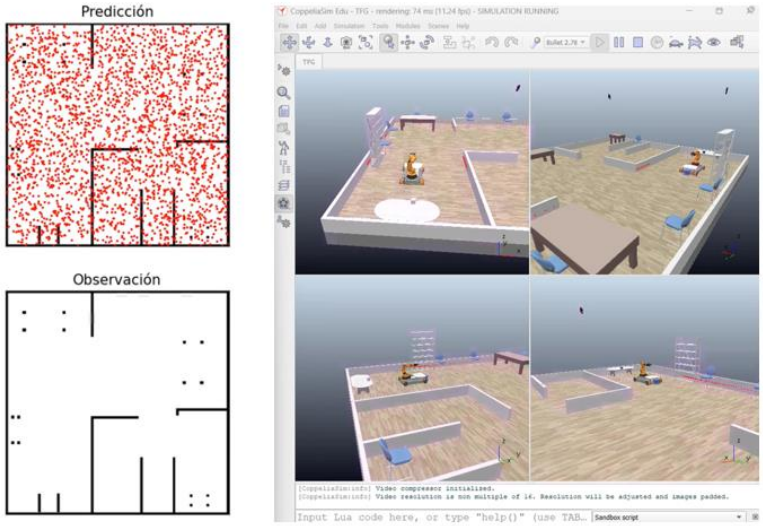


Figura 5-25

En esta ocasión, y después de algunas iteraciones, el proceso de localización logra determinar la posición del robot, permitiendo iniciar el mismo procedimiento que en la versión de localización local.

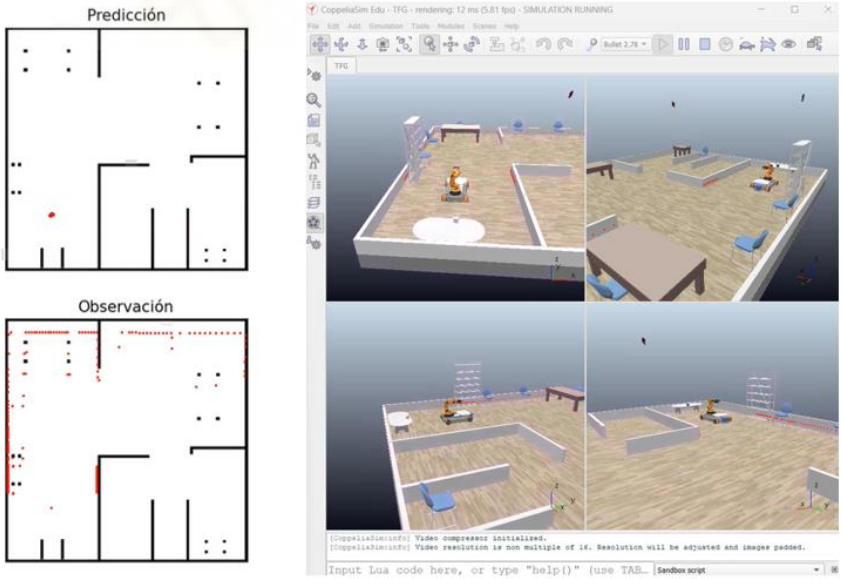


Figura 5-26

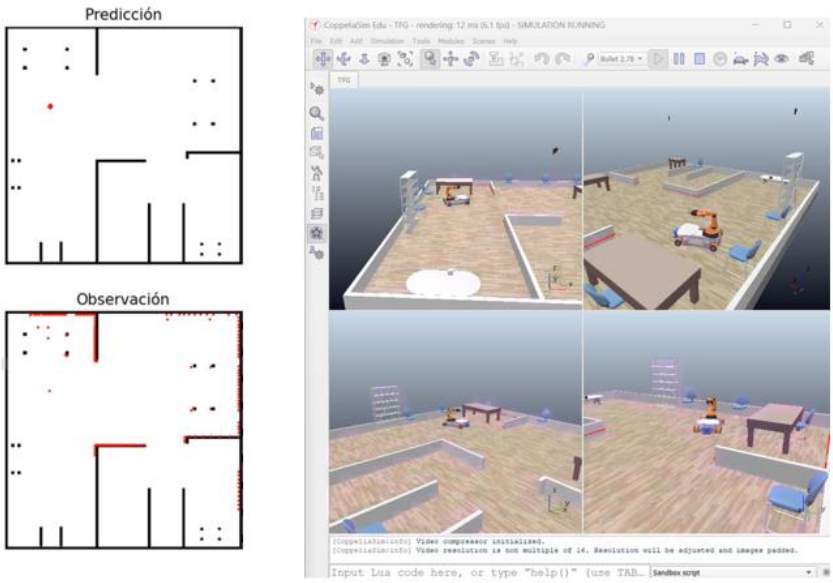


Figura 5-27

5.3. APLICACIÓN DE VISIÓN

Por último, se presenta la aplicación de visión que se ha integrado en el programa. Esta aplicación utiliza marcas Aruco para mejorar la precisión del robot al recoger objetos. Gracias a estas marcas, el robot puede recalcular continuamente su posición y reajustarse de manera adecuada, asegurando una correcta alineación para recoger el objeto deseado. Este proceso de recalibración y ajuste continuo permite que el robot mantenga una alta precisión en sus movimientos, facilitando la tarea de manipulación de objetos con mayor eficacia.

En primer lugar, el robot parte desde un punto inicial conocido y se dirige hacia una de las mesas ubicadas en el escenario donde se encuentra el objeto a recoger.

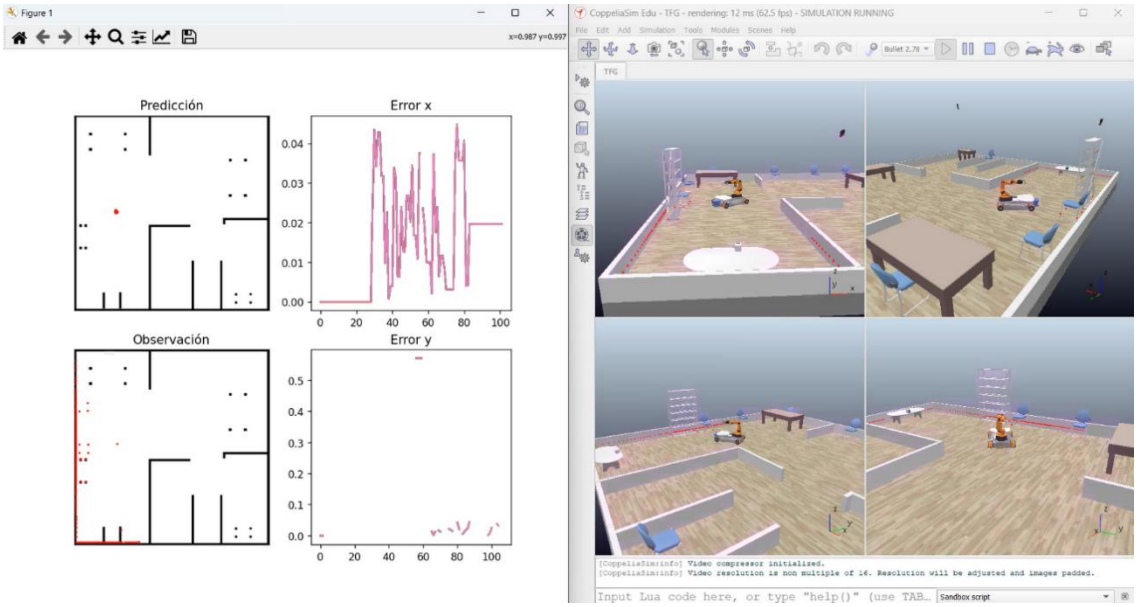


Figura 5-28

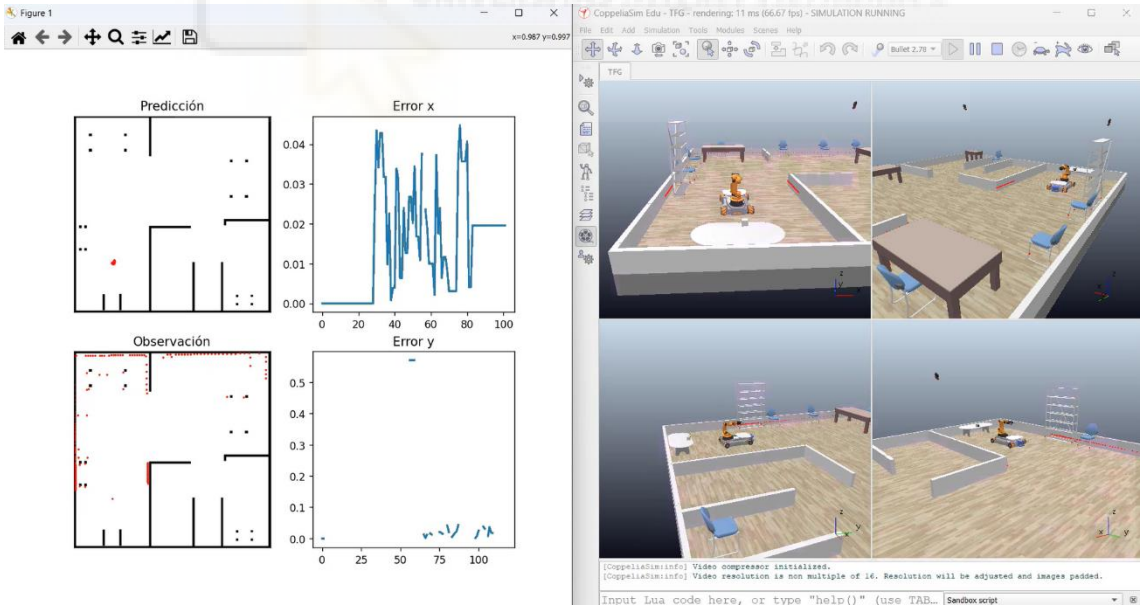


Figura 5-29

Una vez que el robot llega al punto indicado, utiliza el sensor ubicado en el extremo de su brazo para iniciar el proceso de ajuste de su posición, asegurándose de colocarse de manera que pueda recoger el objeto.

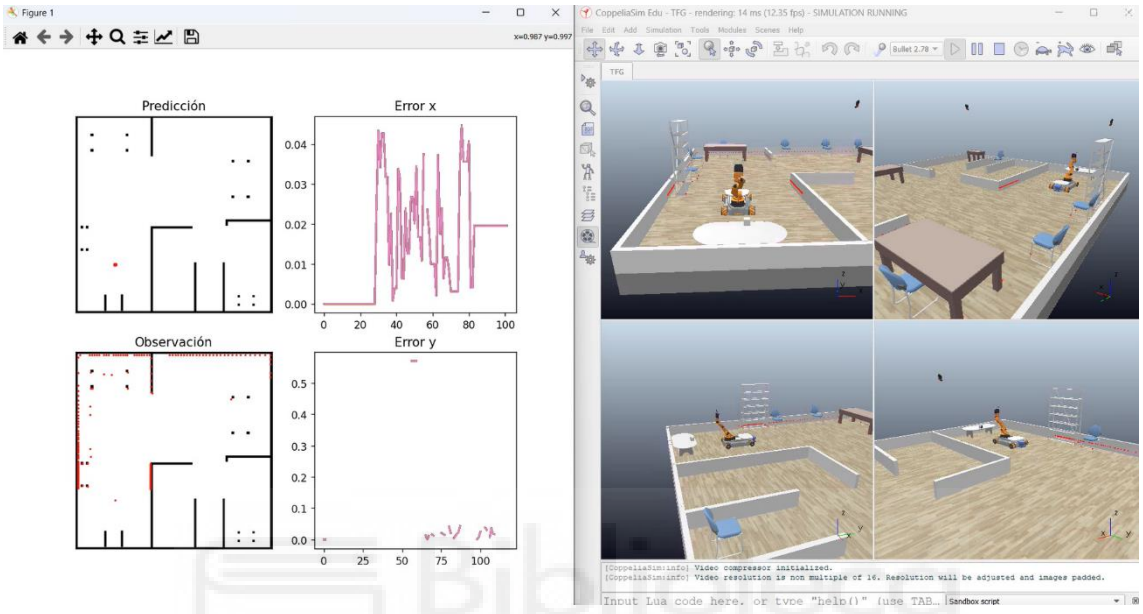


Figura 5-30

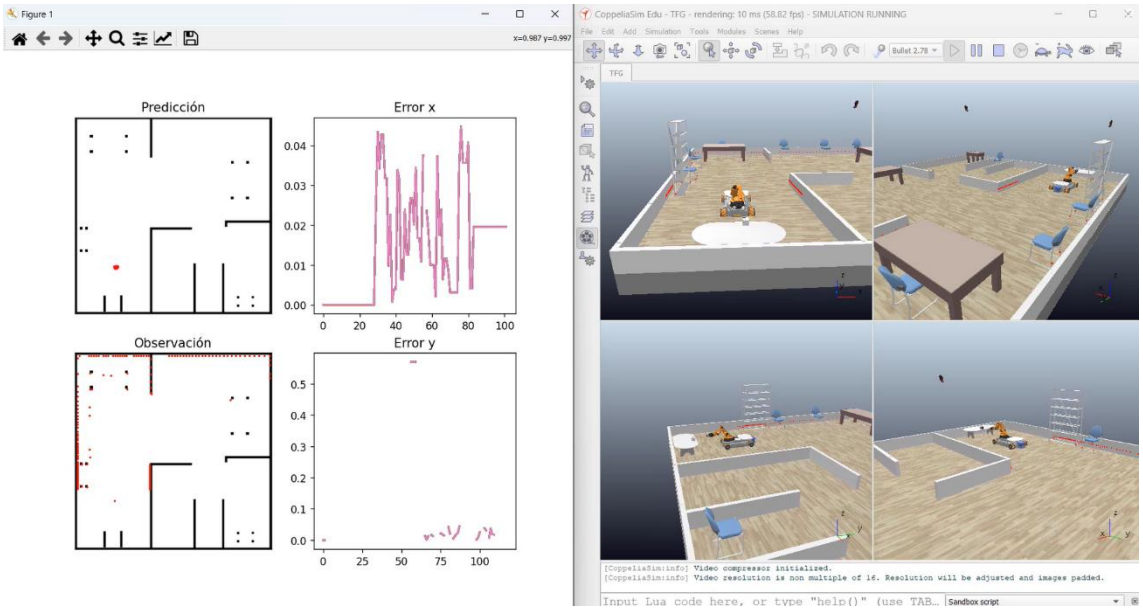


Figura 5-31

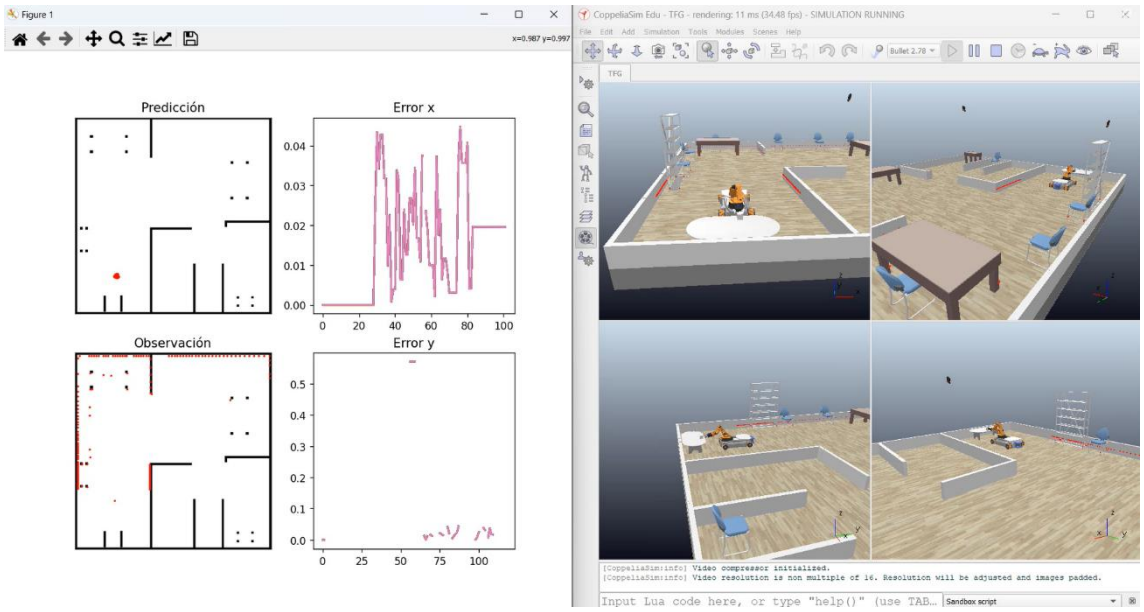


Figura 5-32

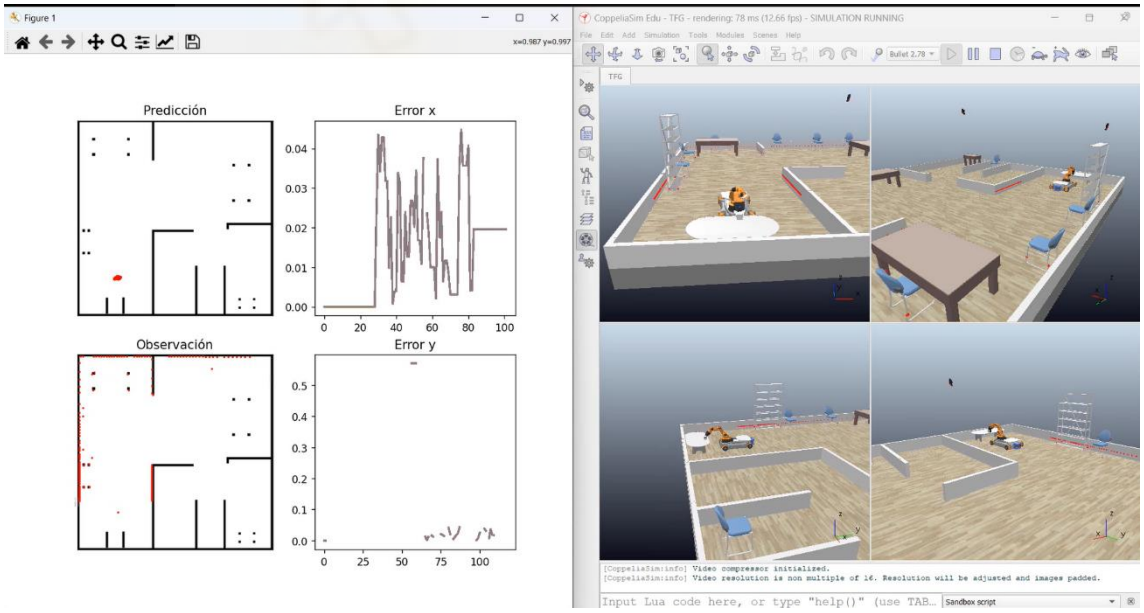


Figura 5-33

Después de verificar que puede recoger el objeto, procede a realizar el proceso de “pick and place”, recogiéndolo de la mesa y colocándolo en la plataforma que tiene en su base.

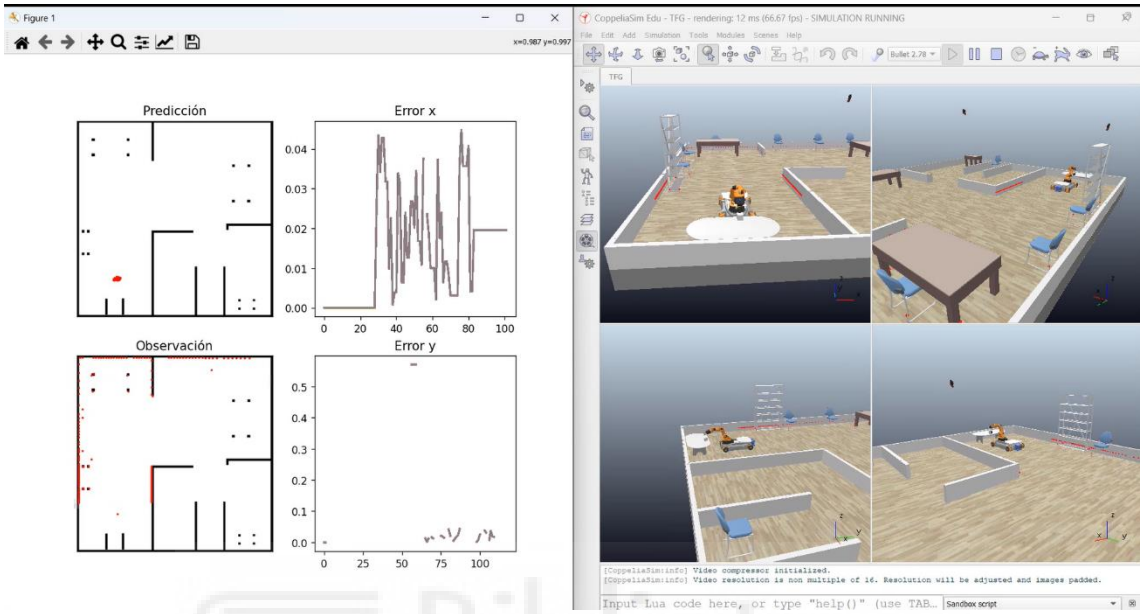


Figura 5-34

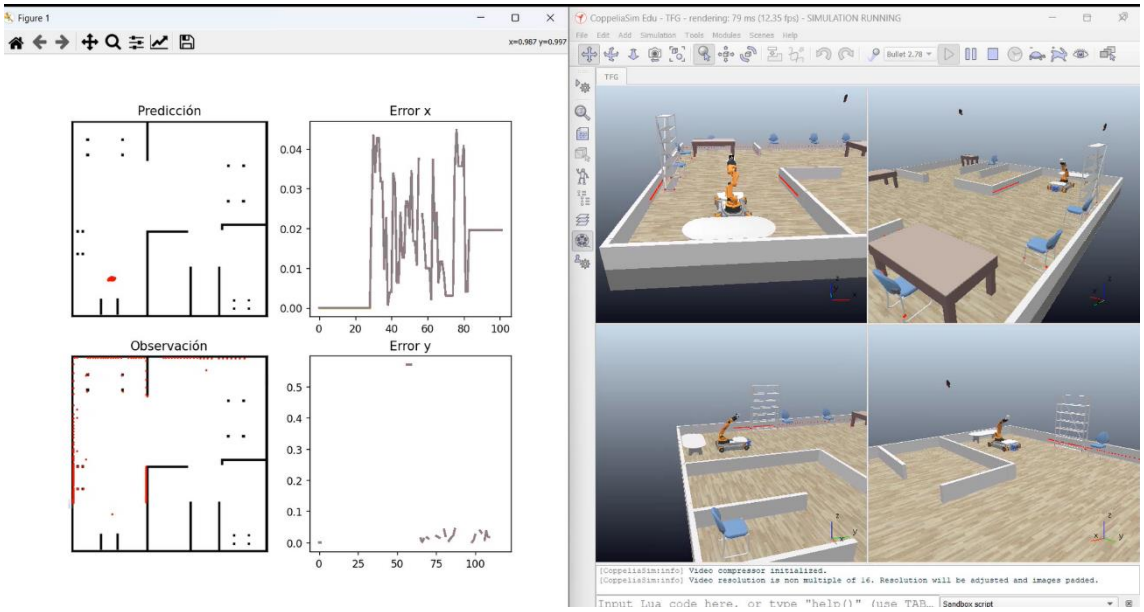


Figura 5-35

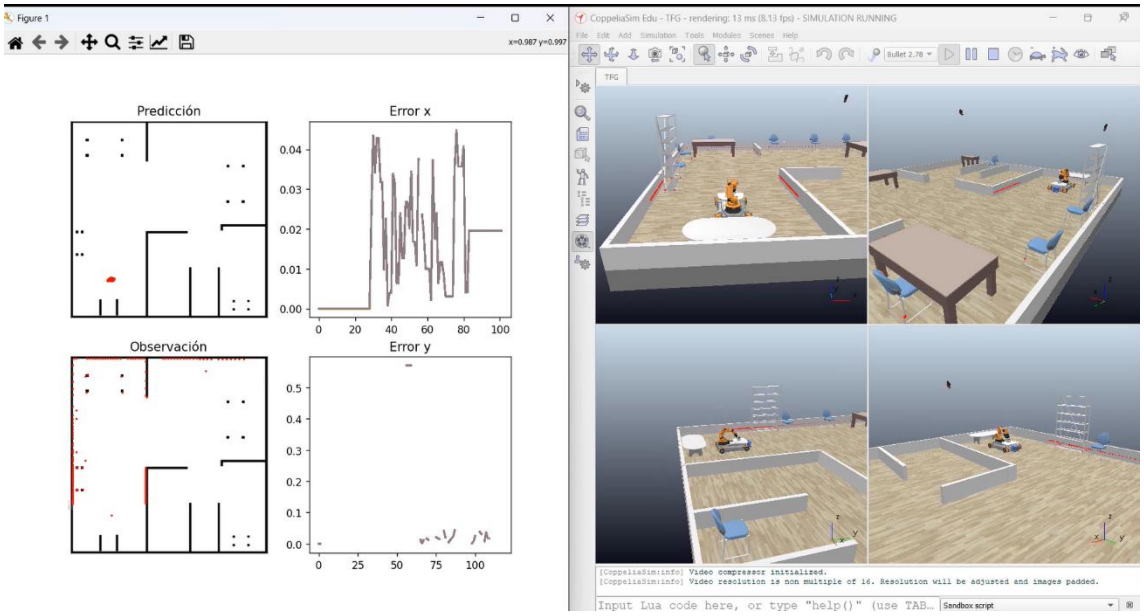


Figura 5-36

Finalmente, se dirige hacia una de las mesas en el escenario de CoppeliaSim para depositar el objeto sobre ella, simulando una aplicación de reparto.

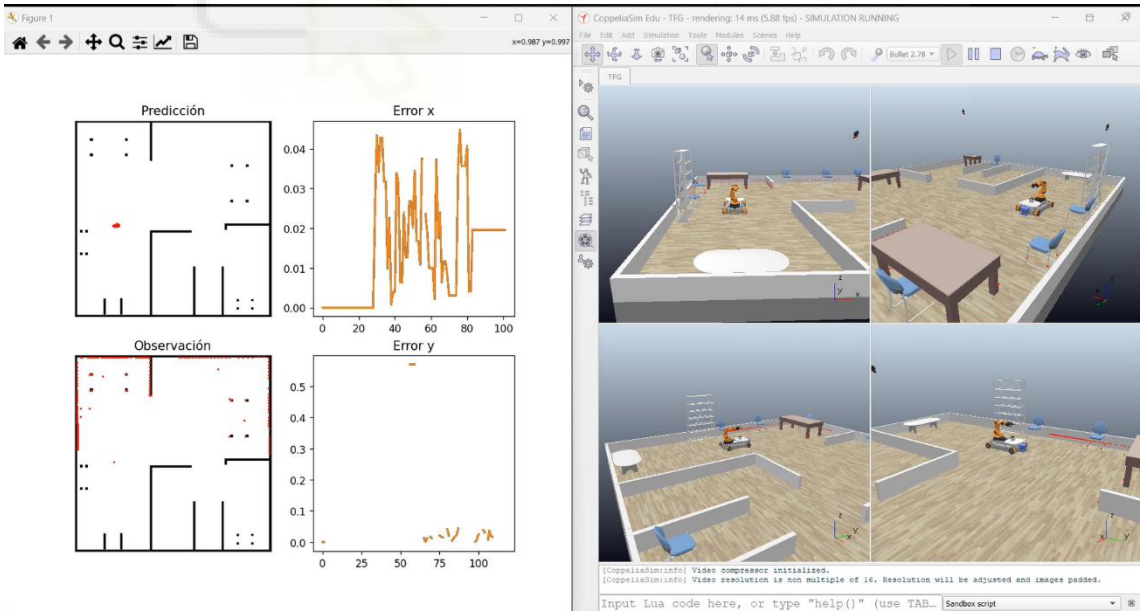


Figura 5-37

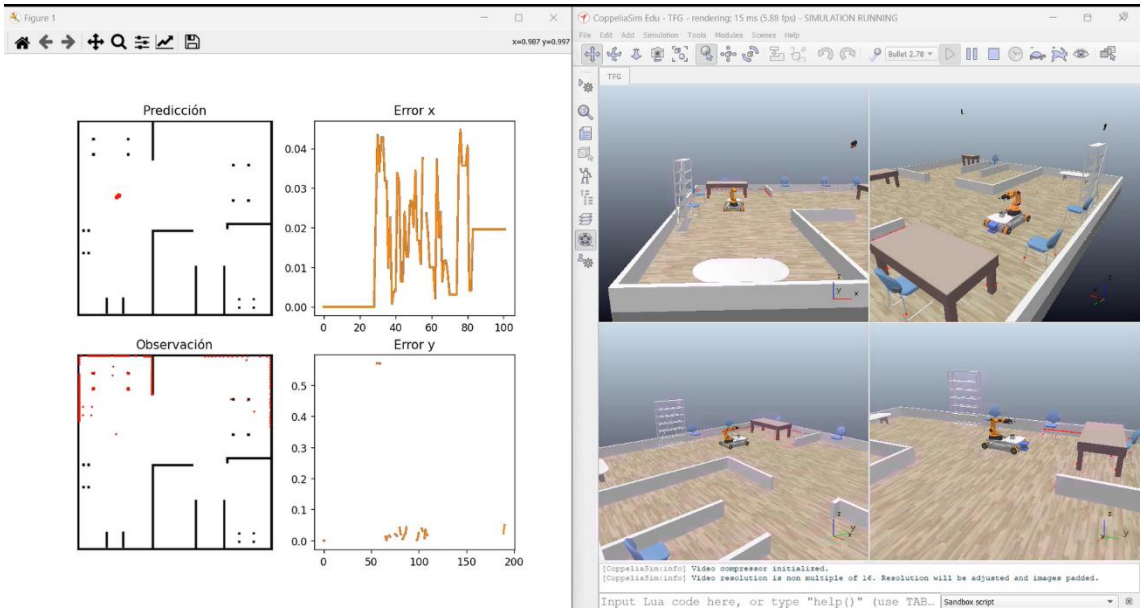


Figura 5-38

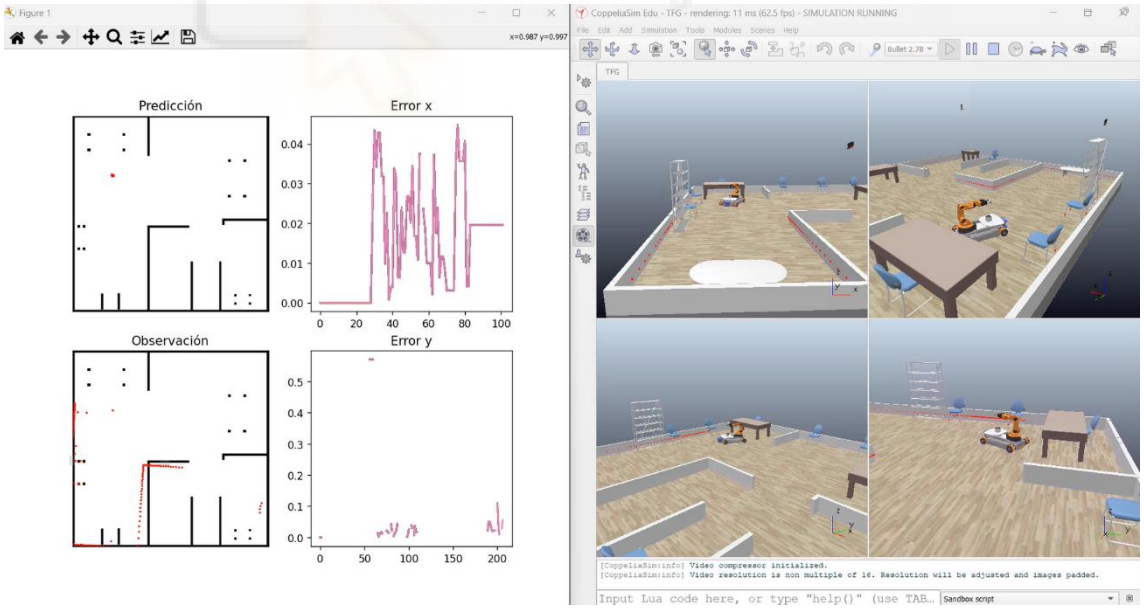


Figura 5-39

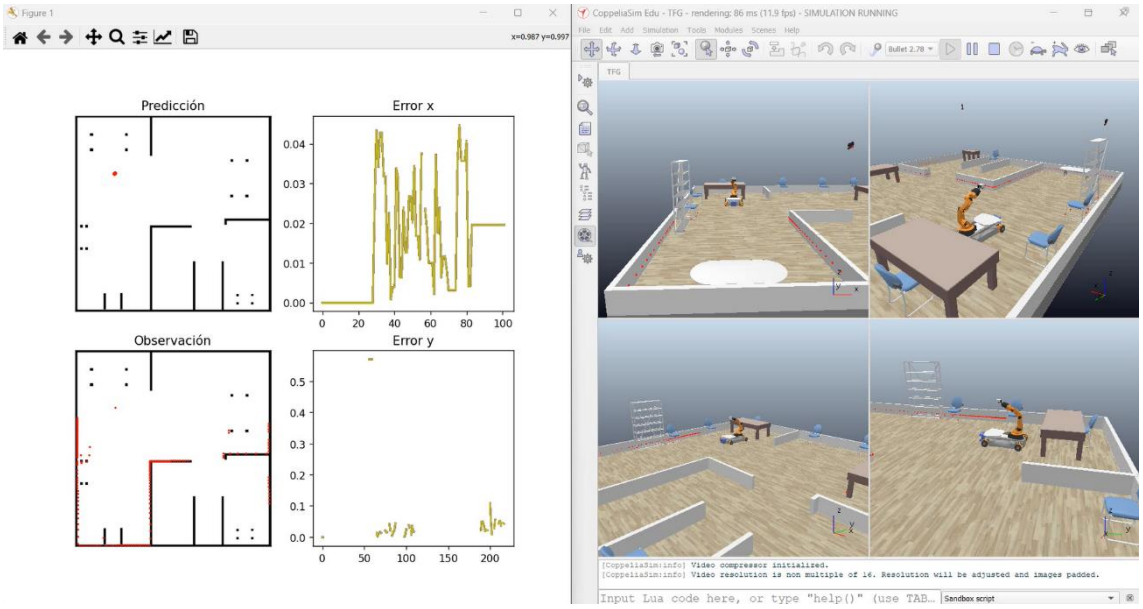


Figura 5-40

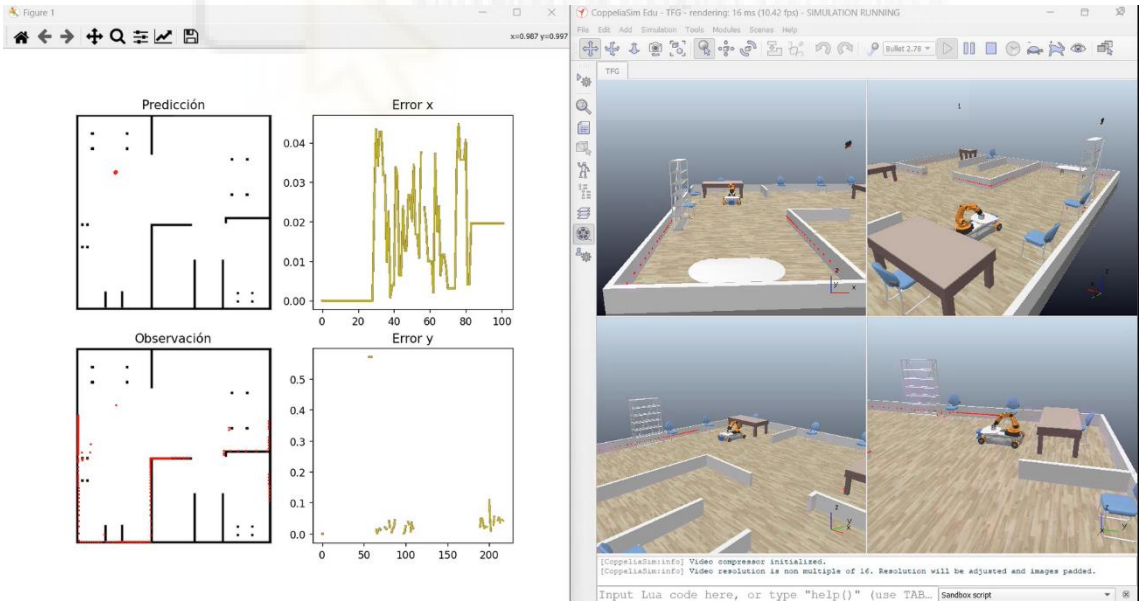


Figura 5-41

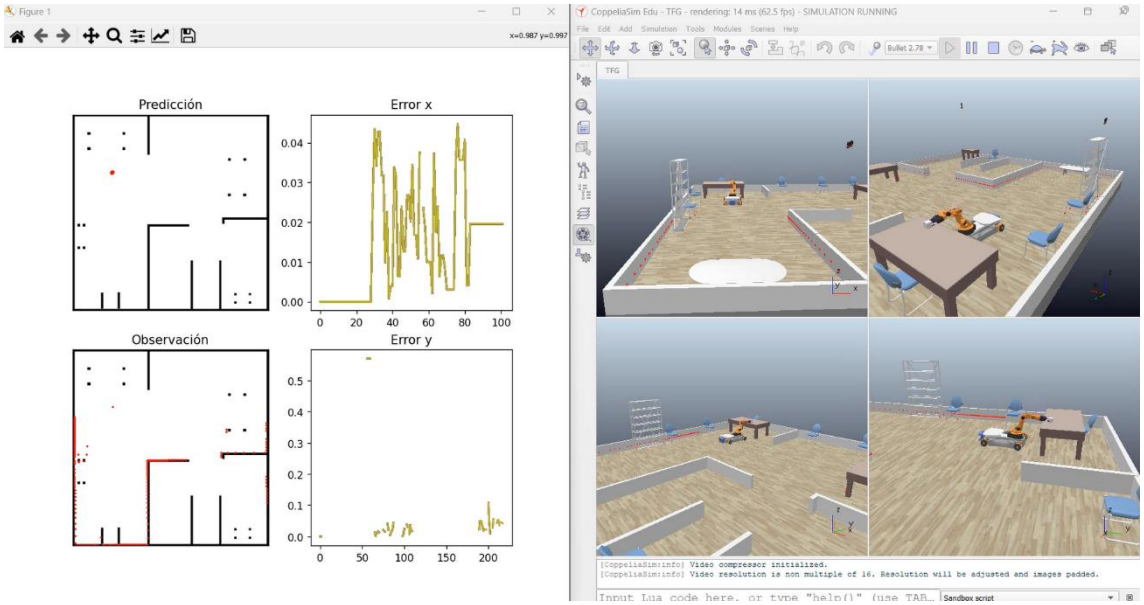


Figura 5-42

6. CONCLUSIONES

En este Trabajo de Fin de Grado (TFG) se ha desarrollado una aplicación en el ámbito de la robótica móvil. Se han estudiado, empleado y desarrollado diversos algoritmos relacionados con la robótica móvil, los cuales han dotado al robot Kuka Youbot de la autonomía necesaria para navegar y sortear obstáculos en un entorno conocido. Además, se ha implementado una aplicación de visión utilizando marcas Aruco, lo que permite al robot realizar tareas de manipulación como el 'pick and place' y simular aplicaciones de reparto. Todo este desarrollo se ha llevado a cabo en el entorno de simulación CoppeliaSim, anteriormente conocido como V-REP.

Desde un punto de vista de aprendizaje, he tenido la oportunidad de investigar y estudiar algoritmos que resultan especialmente relevantes en el ámbito de la robótica móvil, ya que ofrecen soluciones avanzadas para la navegación autónoma de robots. Esta investigación me ha permitido comprender en profundidad cómo los algoritmos de localización y los métodos de planificación de trayectorias, juegan un papel crucial en dotar de autonomía a los robots. Este proceso de aprendizaje ha sido respaldado por la implementación y simulación de estos algoritmos y aplicaciones en entornos como CoppeliaSim, proporcionándome una comprensión práctica y profunda de las oportunidades en el campo de la robótica móvil.

Desde el punto de vista de la implementación, todo el trabajo se ha realizado utilizando el entorno de simulación CoppeliaSim, mencionado previamente, y el lenguaje de programación Python. Esto me ha permitido avanzar significativamente en el ámbito de la programación, especialmente en la programación orientada a objetos. Además, este proyecto me ha brindado la oportunidad de aplicar conocimientos teóricos adquiridos durante la carrera, como el modelado cinemático inverso y directo de un brazo robótico, el uso de operaciones homogéneas para la transformación de información entre diferentes sistemas de referencia, así como la utilización de marcas Aruco para la detección y posicionamiento de objetos en el entorno, los cuales el robot debía manipular.

6.1. TRABAJOS FUTUROS

En cuanto a futuras mejoras y adiciones, se propone desarrollar algoritmos de planificación de trayectorias que permitan al robot sortear obstáculos nuevos mediante la integración de sensores adicionales. Además, se contempla implementar mejoras en la navegación para permitir al robot re-localizarse en caso de perderse durante la ruta. También se considera la posibilidad de integrar esta aplicación en un robot real, aprovechando las capacidades de gestión de hardware y comunicación entre componentes del entorno de programación ROS (Robot Operating System). Asimismo, se planea mejorar la interfaz de usuario para permitir indicar de manera intuitiva y eficiente la ruta deseada del robot, así como especificar el número de partículas requeridas y el tipo de localización directamente desde la interfaz.

Además, como visión más enfocada en las investigaciones actuales, se podría explorar mejoras para la optimización del rendimiento del algoritmo de localización en entornos dinámicos, aprovechando técnicas avanzadas de Deep Learning.

7. BIBLIOGRAFÍA

[1] Fox, D. a. (01 de 1999). Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. *Proceedings of the National Conference on Artificial Intelligence*, 343-349.

[2] Gil, A. (2023). *coppelia_API*. Obtenido de Github.com: https://github.com/4rtur1t0/coppelia_API

[3] Gil, A. (2023). *pyARTE*. Obtenido de Github.com: https://github.com/4rtur1t0/pyARTE/pulse

[4] Gonzalez-Villela, V. a.-B.-A. (09 de 2015). Cinemática de una plataforma móvil omnidireccional con llantas Mecanum en configuración AB.

[5] Julier, S. J. (1997). *New extension of the Kalman filter to nonlinear systems.* (I. Kadar, Ed.) doi:10.1117/12.280797

[6] López, D. G.-B. (2010). Planificación de Trayectorias con el Algoritmo RRT. Aplicación a Robots No Holónomos. *Revista Iberoamericana De Automática E Informática Industrial*, 56–67. Obtenido de https://polipapers.upv.es/index.php/RIAI/article/view/8144

[7] Sebastian Thrun, D. F. (2001). Robust Monte Carlo localization for mobile robots. *Artificial Intelligence*, 99-141. doi:https://doi.org/10.1016/S0004-3702(01)00069-8

[8] Tang, Z. a. (07 de 2021). An overview of path planning algorithms. *IOP Conference Series: Earth and Environmental Science*. doi:10.1088/1755-1315/804/2/022024

[9] Thrun, S. (2000). Probabilistic Algorithms in Robotics. *AI Magazine*, 93-109. doi:https://doi.org/10.1609/aimag.v21i4.1534

[10] Wang, L. L. (2023). Path planning techniques for mobile robots: Review and prospect. *Expert Systems with Applications*, 120-254. doi:https://doi.org/10.1016/j.eswa.2023.120254

[11] Wang, L. L. (2023). Path planning techniques for mobile robots: Review and prospect. *Expert Systems with Applications*, 120-254. doi:https://doi.org/10.1016/j.eswa.2023

8. ANEJOS

8.1. ANEJO I: CÓDIGO PYTHON DE LAS CLASES DEL PROGRAMA

```python
from Clase_Montecarlo import Montecarlo
from robots.youbot import YouBotRobot
from robots.simulation import Simulation
from planificador import PlanificadorAstar
from robots.grippers import YouBotGripper
from camera import Camera
from artelib.euler import Euler
from artelib.vector import Vector
import random

import numpy as np
# import time


class Mando:

    def __init__(self,type_loc=False,n_part=200):

        self.simulation = Simulation()
        self.clientid = self.simulation.start()
        self.a = PlanificadorAstar(image='Mapa.jpg')
        self.robot = YouBotRobot(clientid=self.clientid)
        self.monte = Montecarlo(clientid=self.clientid,start_loc=type_loc,n_particulas=n_part)
        self.gripper = YouBotGripper(clientID=self.clientid)
        self.camara = Camera(clientID=self.clientid)
        self.pos = None
        # Sirven para habilitar la actualización del mapa que se realiza en el archivo aplicación
        self.hab_map = False
        self.hab_map_abs = False
        self.angulo = np.linspace(0.7854 - 0.03490658504, 0.7854 + 0.0872664626, 8)

    def start(self):
        self.a.create_graph_from_image()
        self.robot.start()
        self.monte.start()
        self.gripper.start()
        self.camara.start(name='/youBot/Vision_sensor')
```

```
38
39      def loc_glob(self,threshold):
40          self.monte.start_particulas()
41          i = 0
42          attempts = 0
43          while True:
44              if i <= 4:
45                  vx = random.uniform(0.1, 0.3)
46                  vy = random.uniform(0.1, 0.3)
47                  self.robot.set_base_speed(vx=vx, vy=vy, wz=0)
48                  self.robot.wait(5)
49                  self.pos = self.monte.prediction([vx, vy, 0], 0.25)
50
51                  self.hab_map = True
52                  self.robot.set_base_speed(vx=-vx, vy=-vy, wz=0)
53                  self.robot.wait(5)
54                  self.pos = self.monte.prediction([-vx, -vy, 0], 0.25)
55                  self.hab_map = True
56              else:
57                  self.robot.set_base_speed(vx=-0.2, vy=0, wz=0)
58                  self.robot.wait()
59                  self.pos = self.monte.posicion([-0.2, 0, 0], 0.05)
60                  self.hab_map_abs = True
61                  i = 0
62                  loc = self.monte.convergence(threshold=threshold)
63                  if loc:
64                      self.monte.n_part = 200
65                      self.monte.local_global = False
66                      print('Localización global terminada')
67                      break
68                  else:
69                      if attempts == 4:
70                          self.monte.start_particulas()
71                          print('Localización global reiniciada')
72                          attempts = 0
73                      attempts += 1
74              i += 1
75
76
77      def route(self, p_inicial, p_final):
78          self.a.get_path(start=p_inicial, goal=p_final)
79          lista_puntos = self.a.get_path_meters()
80          return lista_puntos
```

```python

    def mov(self, p, diferencia):
        p.append(1)
        i = 0
        while True:
            # pasar
            error = self.a.point_coordinates_robot(self.pos, p)# orientación respecto al sis. ref. del mapa
            # controlador P del robot
            vx, vy, wz = self.controlador(error=error)
            self.robot.wait()
            if i <= 10:
                self.pos = self.monte.prediction([vx, vy, wz], 0.05)
                self.hab_map = True
            else:
                self.pos = self.monte.posicion([vx, vy, wz], 0.05)
                self.hab_map_abs = True
                i = 0
            dif = np.array(self.pos) - np.array(p[:3])
            dif = np.linalg.norm(dif)
            i += 1
            if dif <= diferencia:
                break
```

```python

    def nav(self, inicial, final, diferencia_error=0.1):
        i = 0
        self.monte.start_particulas(inicial)
        path_global = self.route(inicial[:2], final[:2])
        path_global.append([path_global[-1][0], path_global[-1][1], final[2]])
        n_punt = len(path_global)
        self.pos = self.monte.posicion(motion_inputs=[0, 0, 0])
        self.hab_map_abs = True
        for p in path_global:
            if i < n_punt-1:
                print('Datos PARTICULA')
                self.mov(p, diferencia=diferencia_error)
            else:
                print('Datos PARTICULA')
                self.mov(p, diferencia=0.01)
            i += 1
        self.robot.set_base_speed(0, 0, 0)

    def pick(self, punto, euler, codo=1):
        self.robot.set_base_speed(0, 0, 0)
        # Los valores en el sistema de referencia del robot
        tp1 = Vector(punto)
        to = Euler(euler)  # convención de los angulos de euler es XYZ
        self.gripper.open(precision=True)
        self.robot.moveJ(target_position=tp1, target_orientation=to,codo=codo)
```

```
        self.gripper.close(precision=True)

    def place(self, punto, euler,codo=2):
        self.robot.set_base_speed(0, 0, 0)
        # Los valores en el sistema de referencia del robot
        tp1 = Vector(punto)
        to = Euler(euler)  # convención de los angulos de euler es XYZ
        self.robot.moveJ(target_position=tp1, target_orientation=to,codo=codo)
        self.gripper.open(precision=True)

    def capture(self):
        self.robot.set_base_speed(0, 0, 0)
        self.robot.wait()
        self.camara.get_image()
        self.camara.get_position()
        return self.camara.pos_aruco

    def recolocazion(self):
        r = 0
        while True:
            try:
                pieza = self.capture()
                # print(pieza)
                pos_w = self.robot.pos_aruco_w(pos_robot=self.pos, punto_c=pieza[0])
                pos_t = self.robot.pos_aruco_t(punto_w=pos_w, pos_robot=self.pos)
                # print(pos_t)
                error_x = pos_t[0] + 0.4
                error_y = 0 + pos_t[1]
                if abs(pos_t[1]) < 0.01 and abs(error_x) < 0.01:
                    self.robot.moveAbsJ(np.array([0, 0, -1.57, 0, 0]))
                    # acercar el robot un poco ya que la camara no es capaz de obtener imagenes tan cercanas
                    self.robot.set_base_speed(vx=-0.2, vy=0, wz=0)
                    self.robot.wait(5)
                    self.pos = self.monte.posicion([-0.2, 0, 0], 0.05)
                    self.hab_map_abs = True
                    break
                else:
                    vx, vy, wz = self.controlador(error=[error_x, error_y, self.pos[2]])
                    self.robot.wait()
                    self.pos = self.monte.prediction([vx, vy, wz], 0.05)
                    self.hab_map = True
            except:
                if r < 8:
                    q = self.robot.qs
                    q[3] = self.angulo[r]
                    self.robot.moveAbsJ(q)
                    r += 1
                else:
                    r = 0
                    q = self.robot.qs
                    q[3] = self.angulo[r]
                    q[2] = q[2] - 0.05235987756
                    q[1] = q[1] + 0.01745329252
                    self.robot.moveAbsJ(q)

```

```
185         return pos_t
186
187     def controlador(self, error):
188         vx = 0.5 * error[0]
189
190         if 0 < vx < 0.02:
191             vx = 0.02
192         vy = 0.5 * error[1]
193         if error[2] < 0:
194             error[2] += 2*np.pi
195         wz = 0.4 * (error[2] - self.pos[2])
196         self.robot.set_base_speed(vx=vx, vy=vy, wz=wz)
197         return vx, vy, wz
198
199     def desplazamiento(self):
200         self.robot.set_base_speed(vx=0.2, vy=0, wz=0)
201         self.robot.wait()
202         self.robot.set_base_speed(vx=0, vy=0, wz=0)
203         self.pos = self.monte.posicion([0.2, 0, 0], 0.05)
204         self.hab_map_abs = True
```

```python

import scipy.ndimage as ndimage
import cv2 as cv


class Mapa:
    def __init__(self, imagen, ancho=5, alto=5):
        self.imagen = imagen
        self.mapa = None
        self.ancho_esc = ancho
        self.alto_esc = alto
        self.pix_height = None
        self.pix_width = None
        self.rel_mp_ancho = None
        self.rel_mp_alto = None
        self.color = 'red'
        self.dimension = 1

    def start(self):
        self.mapa = cv.imread(self.imagen)
        # rotamos la imagen para que cuadre con la orientación de los ejes x,y de coppeli
        # girada
        self.mapa = ndimage.rotate(self.mapa, 180)
        self.pix_height, self.pix_width, _ = self.mapa.shape
        self.rel_mp_ancho = self.pix_width / self.ancho_esc
        self.rel_mp_alto = self.pix_height / self.alto_esc

    def mapa_update(self, particulas, ax1):
        try:
            particulas_x = []
            particulas_y = []
            for i in particulas:
                particulas_x.append(i[0])
            for i in particulas:
                particulas_y.append(i[1])
            # Pasar la imagen de metros a pixeles
            mx = []
            my = []
            for i in particulas_x:
                x = i * self.rel_mp_ancho
                mx.append(x)
            for i in particulas_y:
                y = i * self.rel_mp_alto
```

```
            my.append(y)
        # plt.clf()  # borra la figura que se está mostrando
        for coll in ax1.collections:
            coll.remove()
        # Añadir puntos a la imagen

        ax1.scatter(mx, my, c=self.color, marker='o', s=self.dimension)

    except:
        pass

def mapa_update_dist(self, array, ax2):
    try:
        particulas_x = []
        particulas_y = []
        for i in array:
            particulas_x.append(i[0])
        for i in array:
            particulas_y.append(i[1])
        # Pasar la imagen de metros a pixeles
        mx = []
        my = []
        for i in particulas_x:
            x = i * self.rel_mp_ancho
            mx.append(x)
        for i in particulas_y:
            y = i * self.rel_mp_alto
            my.append(y)
        # plt.clf()  # borra la figura que se está mostrando
        for coll in ax2.collections:
            coll.remove()
        # Añadir puntos a la imagen

        ax2.scatter(mx, my, c=self.color, marker='o', s=self.dimension)


    except:
        pass

def update_error_x(self, vector_x, ax3):
    for coll_x in ax3.collections:
        coll_x.remove()
    ax3.plot(vector_x[1], vector_x[0],'r-')
```

```
89      def update_error_y(self, vector_y, ax4):
90          for coll_y in ax4.collections:
91              coll_y.remove()
92          ax4.plot(vector_y[1], vector_y[0],'r-')
93
94
95      def show_map(self, ax2, ax1, ax3, ax4):
96          # Pasar la imagen de metros a pixeles
97          limite_inferior_x = -self.pix_width / 2
98          limite_superior_x = self.pix_width / 2
99
100         limite_inferior_y = -self.pix_height / 2
101         limite_superior_y = self.pix_height / 2
102
103         ax1.imshow(self.mapa, extent=[limite_inferior_x, limite_superior_x, limite_inferior_y, limite_superior_y])
104         ax1.set_xlim(limite_inferior_x, limite_superior_x)
105         ax1.set_ylim(limite_inferior_y, limite_superior_y)
106         ax1.set_xticks([])
107         ax1.set_yticks([])
108         # ax1.set_xlabel(f'Eje x, Escala: {self.rel_mp_ancho} pix/m')
109         # ax1.set_ylabel(f'Eje y, Escala: {self.rel_mp_alto} pix/m')
110         ax1.set_title('Predicción')
111
112         ax2.imshow(self.mapa, extent=[limite_inferior_x, limite_superior_x, limite_inferior_y, limite_superior_y])
113         ax2.set_xlim(limite_inferior_x, limite_superior_x)
114         ax2.set_ylim(limite_inferior_y, limite_superior_y)
115         ax2.set_xticks([])
116         ax2.set_yticks([])
117         # ax2.set_xlabel(f'Eje x, Escala: {self.rel_mp_ancho} pix/m')
118         # ax2.set_ylabel(f'Eje y, Escala: {self.rel_mp_alto} pix/m')
119         ax2.set_title('Observación')
120
121         ax3.set_title('Error x')
122         ax3.set_ylabel('Metros')
123         ax4.set_title('Error y')
124         ax4.set_ylabel('Metros')
125
```

```python
import cv2
import numpy as np
import sim
import cv2 as cv
from PIL import Image
import os
import json


class Camera:
    def __init__(self, clientID):
        self.clientID = clientID
        self.camera = None
        self.image = None
        self.position = None
        # La matriz de rotació la marca la función de los arucos conforme da los valores de las coordenad


        try:
            with open("camera_calib.json") as file:
                data = json.load(file)
        except:
            print('Camera Calibration File not valid')
            exit()

        self.cameraMatrix = np.array(data['camera_matrix'])
        self.distCoeffs = np.array(data['distortion_coefficients'])
        self.dictionary = cv.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_1000)
        self.pos_aruco = []
        self.tvecs = []

    def start(self, name='camera'):
        _, camera = sim.simxGetObjectHandle(self.clientID, name, sim.simx_opmode_oneshot_wait)
        self.camera = camera

    def get_image(self):
        print('Capturing image of vision sensor ')
        _, resolution, image = sim.simxGetVisionSensorImage(self.clientID, self.camera, 0,
                                                            sim.simx_opmode_oneshot_wait)
        # return image in openCV format.
        image = np.array(image, dtype=np.uint8)
        image.resize([resolution[1], resolution[0], 3])
        # cambiar el orden de los canales y pasar la imagen de RGB a BGR
        image = image[:, :, ::-1]
        image = np.array(image, dtype=np.uint8)
        # image = np.rot90(image, 2)
        image = cv2.flip(image, 0)
        self.image = np.array(image, dtype=np.uint8)
        """cv.imshow('hola', self.image)
        cv.waitKey(0)"""


    def get_position(self):
        self.pos_aruco = []  # borrar posiciones anteriores
        self.tvecs = []
        gray_image = cv.cvtColor(self.image, cv.COLOR_BGR2GRAY)  # transforms to gray level
        corners, ids, rejectedImgPoints = cv.aruco.detectMarkers(gray_image, self.dictionary)
        # dispimage = cv.aruco.drawDetectedMarkers(self.image, corners, ids, borderColor=(0, 0, 255))
```

```
70
71          # Calculate POSE
72          if len(corners) > 0:
73              # 75 tamaño del aruco de un lado en mm
74              rvecs, tvecs, _objPoints = cv.aruco.estimatePoseSingleMarkers(corners, 50, self.cameraMatrix,
75                                                                            self.distCoeffs)
76              for tvec in tvecs:
77                  for t in tvec:
78                      self.tvecs.append(np.round(t, 3))
79              for i,tvec in enumerate(self.tvecs):
80                  self.tvecs[i] = np.append(tvec * 0.001, 1)
81                  # print(self.tvecs)
82              for tvec in self.tvecs:
83                  self.pos_aruco.append(tvec)
```



```
1   import random
2   import numpy as np
3   import heapq
4
5   class Particulas:
6
7       def __init__(self):
8           self.contador = 0 # variar valor sigma cada ciertos pasos de simulación
9       def start_particulas(self, p_inicical, numero,loc):
10          # numero de particulas
11          # limites de la escena
12          particulas = []
13          for i in range(numero):
14              if loc==True:
15                  # Si se quisiera realizar una prueba de localización sin conocimiento de donde se encuentra el robot
16                  x_max, x_min = 2.5, -2.5
17                  y_max, y_min = 2.5, -2.5
18                  theta_min, theta_max = 0, 2*np.pi
19
20                  x = random.uniform(x_min, x_max)
21                  y = random.uniform(y_min, y_max)
22                  theta = random.uniform(theta_min, theta_max)
23                  particulas.append([x, y, theta])
24              else:
25                  particulas.append(p_inicical)
26          return particulas
27
```

```python

    def prediction_step(self, motion_inputs, particulas, segundos):

        for i in range(len(particulas)):
            for m in range(len(motion_inputs)-1):
                if self.contador < 10:
                    motion_inputs[m] = motion_inputs[m] + random.gauss(0, 0.015)  # mejor 0.015
                    self.contador+=1
                else:
                    motion_inputs[m] = motion_inputs[m] + random.gauss(0, 0.025)
            # Para caudrar tanto
            coordenadas = particulas[i]
            x_i = coordenadas[0] + segundos * (
                        motion_inputs[0] * np.cos(coordenadas[2]) - np.sin(coordenadas[2]) * motion_inputs[1])
            y_i = coordenadas[1] + segundos*(
                        motion_inputs[1] * np.cos(coordenadas[2]) + np.sin(coordenadas[2]) * motion_inputs[0])
            if self.contador < 10:
                motion_inputs[2] = motion_inputs[2] + random.gauss(0, 0.01)
                self.contador += 1
            else:
                motion_inputs[2] = motion_inputs[2] + random.gauss(0, 0.015)
                self.contador = 0
            theta = coordenadas[2] + segundos * motion_inputs[2]
            particulas[i] = [x_i, y_i, theta]
        return particulas
```



```python

    def observation_step(self, obs_dist, particulas, mapa, conversion):

        part = []

        list_w = []
        # mover las particulas a la posición del sensor teniendo en cuenta el sist. de referencia de coppelia
        # ya que las medidas estan tomadas en la posicion del sensor
        for particula in particulas:
            x, y, z = particula
            x = x + (0.3 * np.cos(z))
            y = y + (np.sin(z) * 0.3)
            coord = [x, y, z]
            part.append(coord)
        # indicar la cantidad de rayos que tiene el sensor de proximidad
        angles = np.linspace(-np.pi/2, np.pi/2, 181)
        for particula in part:
            w_i = 1.0
            for j in range(len(angles)):
                dist = obs_dist[j]
                angle = angles[j]
                dist_simu = cal_recta(angle=angle, particula=particula, mapa=mapa, obs=dist, conversion=conversion)
                p_j = np.exp(-(np.power(dist - dist_simu, 2) / 0.1)) + 0.15
                w_i *= p_j
            list_w.append(w_i)
        norm_w = np.array(list_w)
        # Para conecer el peso máx en cada instante y observar como funciona el algoritmo de localización
        print('peso más grande: ', norm_w[np.argmax(norm_w)])
```

```
        suma = np.sum(norm_w)
        if suma > 0.0:
            norm_w = norm_w/suma
        return norm_w

    def resampling(self, norm_w, particulas):
        nuevas_part = []
        while True:
            for i in range(len(particulas)):
                w_random = random.uniform(0, 1)
                if norm_w[i] >= w_random:
                    nuevas_part.append(particulas[i])
                if len(nuevas_part) == len(particulas):
                    return nuevas_part



def cal_recta(angle, particula, mapa, obs, conversion):

    d = np.linspace(obs-0.05, obs + 0.05, 7)

    for di in d:
        i = particula[0] + (di * np.cos(particula[2] + angle))
        j = particula[1] + (di * np.sin(particula[2] + angle))
        # Per a canviar la posició del punt (0,0) a la esquina que es desde donde empieza la imagen a contr
        i = i + 2.5
```



```
        j = j - 2.5
        i = i * conversion[0]
        j = j * conversion[1]
        i = abs(round(i))
        j = abs(round(j))
        # primero indicamos el alto y luego el ancho del pixel
        try:
            if mapa[j, i] == 0:
                return di
        except IndexError:
            pass
    return 5.0
```

```python
import numpy as np
from Clase_Mapa import Mapa
from Clase_Particulas import Particulas
from Clase_laser import LaserScanner2D


class Montecarlo:
    def __init__(self, clientid, n_particulas=200,start_loc = False ):
        self.n_part = n_particulas  # El número de particulas que queremos dibujar
        self.particulas = []  # Donde se guardan todas las coordenadas de las partículas
        self.mapa = None  # objeto de la clase Mapa
        self.clientid = clientid  # atributo que recoge el canal de conexion con CoppeliaSim
        self.local_global = start_loc # indicar que tipo de localización quieren al inico del algoritmo (global:true,
        self.p = None  # Objeto de la clase particulas
        self.laser = None  # objeto de la clase Laserscanner 2D
        self.pos_laser = None
        self.dist = None
        self.pos = None
        self.array = []
        self.pesos = []

```

```
21
22      def start(self):
23          # Crear objetos
24          self.p = Particulas()
25          self.laser = LaserScanner2D(self.clientid)
26          self.mapa = Mapa(imagen='Mapa.jpg')
27          # inicializar objetos
28          self.mapa.start()
29          self.laser.start()
30
31      def start_particulas(self, inicio=0):
32          self.particulas = self.p.start_particulas(inicio, self.n_part, loc=self.local_global)
33
34      def prediction_step(self, motion_inputs, segundos):
35          self.particulas = self.p.prediction_step(motion_inputs=motion_inputs,
36                                                   particulas=self.particulas, segundos=segundos)
37
38      def observation_step(self):
39          self.dist = self.laser.get_laser_data()
40          norm_w = self.p.observation_step(obs_dist=self.dist, particulas=self.particulas, mapa=self.mapa.mapa[:, :, 0],
41                                           conversion=[self.mapa.rel_mp_ancho, self.mapa.rel_mp_alto])
42          self.particulas = self.p.resampling(norm_w=norm_w, particulas=self.particulas)
43          return norm_w
```

```
45      def posicion(self, motion_inputs, segundos=0.05):
46          self.prediction_step(motion_inputs=motion_inputs, segundos=segundos)
47          self.pesos = self.observation_step()
48          self.pos = self.particulas[np.argmax(self.pesos)]
49          #self.pos = self.media()
50          self.calc_dist()
51          self.array.append(self.pos[:2])
52          return self.pos
53
54      def prediction(self, motion, segundos=0.05):
55          self.particulas = self.p.prediction_step(motion_inputs=motion,
56                                                   particulas=self.particulas, segundos=segundos)
57          self.pos = self.media()
58          return self.pos
```

```
59
60      def media(self):
61          media = np.array(self.particulas)
62          return np.max(media, axis=0)
63
64      def convergence(self,threshold):
65          x_coords = [p[0] for p in self.particulas]
66          y_coords = [p[1] for p in self.particulas]
67          theta_coords = [p[2] for p in self.particulas]
68          std_x = np.std(x_coords)
69          std_y = np.std(y_coords)
70          std_theta = np.std(theta_coords)
71          print(std_y,std_theta,std_x)
72          return std_x < threshold and std_y < threshold and std_theta < threshold
73
74      # calcular los puntos finales de los rayos
75      def calc_dist(self):
76          self.array = []
77          angles = np.linspace(-np.pi / 2, np.pi / 2, 181)
78          # a la hora de calcular los finales de los rayos hay que tener en cuenta que el sensor se encuentra en
79          # otra posción a 0.3 del centro del robot que es lo que indicamos y obtenemos con el algoritmo.
80          x, y, z = self.pos
81          x = x + (0.3 * np.cos(z))
82          y = y + (np.sin(z) * 0.3)
83          self.pos_laser = [x, y, z]
```

```
84
85          for i, di in enumerate(self.dist):
86              angle = angles[i]
87              i = self.pos_laser[0] + di * np.cos(self.pos_laser[2] + angle)
88              j = self.pos_laser[1] + di * np.sin(self.pos_laser[2] + angle)
89              self.array.append([i, j])
```

```python
import matplotlib.pyplot as plt
import networkx as nx
import scipy.ndimage as ndimage
from scipy.ndimage import binary_dilation
import numpy as np
import cv2 as cv


class PlanificadorAstar:

    def __init__(self, image):
        self.imagen = image
        self.mapa = None
        self.g = None
        self.path = None
        self.rel_mp_ancho = None
        self.rel_mp_alto = None
        self.pix_height = None
        self.pix_width = None

    def create_graph_from_image(self):
        # Paso 1: Leer la imagen binaria
        self.mapa = cv.imread(self.imagen)
        self.pix_height, self.pix_width, _ = self.mapa.shape
        self.mapa = ndimage.rotate(self.mapa, 180)
        self.mapa = self.mapa[:, :, 0]

        # aseguramos que la imagen sea binaria.(para buscar en imagen se indica primero altura(y) y luego anchura (x)
        for y in range(self.pix_height):
            for x in range(self.pix_width):
                if 250 < self.mapa[y, x] < 255:
                    self.mapa[y, x] = 255
                elif 0 < self.mapa[y, x] < 5:
                    self.mapa[y, x] = 0
        # invertimos la imagen porque las dilataciones son sobre el pixel en blanco
        inverted_image = 255 - self.mapa
        """plt.imshow(self.mapa)
        plt.show()"""
        kernel = np.array([[True, True, True], [0, 0, 0], [True, True, True]])
        dilated_image = binary_dilation(inverted_image, structure=kernel, iterations=35)
        kernel = np.array([[True, 0, True], [True, 0, True], [True, 0, True]])
        dilated_image = binary_dilation(dilated_image, structure=kernel, iterations=15)
        # Volvemos a invertir para obtener los muros en negro.
        self.mapa = dilated_image - 1
        self.mapa = abs(self.mapa)

        # Paso 2: Crear un grafo vacío para el algoritmo A*
        self.g = nx.Graph()
        # Paso 3 y 4: Agregar nodos y aristas al grafo
        for y in range(self.pix_height):
            for x in range(self.pix_width):
                pixel_value = self.mapa[y, x]
                if pixel_value == 1:  # Píxel blanco (celda transitable)
                    self.g.add_node((x, y))
                    # Agregar aristas con celdas transitables adyacentes
                    for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                        new_x, new_y = x + dx, y + dy
                        if 0 <= new_x < self.pix_width and 0 <= new_y < self.pix_height and self.mapa[new_y, new_x] == 1
                            self.g.add_edge((x, y), (new_x, new_y))

```

```
    def get_path(self, start, goal):
        p_ini = [0, 0]
        p_fin = [0, 0]
        mapa = cv.imread(self.imagen)
        mapa = ndimage.rotate(mapa, 180)
        pix_height, pix_width, _ = mapa.shape
        self.rel_mp_ancho = pix_width / 5.0
        self.rel_mp_alto = pix_height / 5.0
        # Utilizar A* para encontrar la ruta óptima desde el punto de inicio al objetivo
        # Transformo las coord. de coppelia en las del mapa (pixel) y modifico las coorden
        p_ini[0] = start[0] + 2.5
        p_ini[1] = start[1] + -2.5
        p_ini[0] = p_ini[0] * self.rel_mp_ancho
        p_ini[1] = p_ini[1] * self.rel_mp_alto
        p_ini[0] = abs(round(p_ini[0]))
        p_ini[1] = abs(round(p_ini[1]))
        p_ini = (p_ini[0], p_ini[1])

        p_fin[0] = goal[0] + 2.5
        p_fin[1] = goal[1] + -2.5
        p_fin[0] = p_fin[0] * self.rel_mp_ancho
        p_fin[1] =p_fin[1] * self.rel_mp_alto
        p_fin[0] = abs(round(p_fin[0]))
        p_fin[1] = abs(round(p_fin[1]))
        p_fin = (p_fin[0], p_fin[1])
        self.path = nx.astar_path(self.g, p_ini, p_fin)


def get_path_meters(self):
    path_redu = []  # guardaremos los valores de pixel del path pero ya reducido este
    n_path = []  # guardaremos los valores del path reducido en coordenadas del SG
    angle_rad = 0
    i = 25

    # Pasar los puntos a metros y cambiar la posició del punt (0,0)
    for p in self.path:
        p = list(p)
        p[0] = p[0] / self.rel_mp_ancho
        p[1] = p[1] / self.rel_mp_alto
        # Per a cambiar la posició del punt (0,0)
        p[0] = p[0] + -2.5
        p[1] = -p[1] + 2.5 # tener en cuenta que la 'y' en coppelia está girada 180° relativo a la 'y' del
        n_path.append(p)
    # Para añadir a cada punto el angulo theta del robot
    for p in range(len(n_path)-1):
        delta_x = n_path[p+1][0] - n_path[p][0]
        delta_y = n_path[p+1][1] - n_path[p][1]
        angle_rad = np.arctan2(delta_y, delta_x)
        if angle_rad < 0:
            angle_rad += 2*np.pi
        n_path[p].append(angle_rad)
    n_path[-1].append(angle_rad)
    m_ant = []
```

```
        # Reduir el numero de punts
        for m in n_path:
            if i == 25:
                m = list(m)
                path_redu.append(m)
                i = 0
            elif m_ant[2]!=m[2]:
                m = list(m)
                path_redu.append(m)
            m_ant = m
            i += 1
        path_redu.append(list(n_path[(len(n_path) - 1)]))

        """# Ajustar cada coordenada para tener en cuenta los giros del robot
        i = 0
        for p in path_robot:
            if p[2] != start[2]:
                dif = p[2] - start[2] # per a calcular la rotacio segons el angle que gira el robot
                rotation_matrix = np.array([[np.cos(dif), -np.sin(dif)],
                                            [np.sin(dif), np.cos(dif)]])
                transformed_point = np.dot(rotation_matrix, p[:2])
                path_robot[i] = np.array([transformed_point[0], 0, p[2]])
            i += 1"""
        return path_redu


    # Pasar todos los puntos al sistema de coordenadas del robot
    def point_coordinates_robot(self, ubi_r, p_camino):

        t_wr = np.array([[np.cos(ubi_r[2]), -np.sin(ubi_r[2]), 0, ubi_r[0]],
                         [np.sin(ubi_r[2]), np.cos(ubi_r[2]), 0, ubi_r[1]],
                         [0, 0, 1, 0],
                         [0, 0, 0, 1]])
        t_wr_inv = np.linalg.inv(t_wr)
        # Pasar el punto al sistema de referencia del robot y que está pasado con el del coppeliasim
        p_ri = np.dot(t_wr_inv, p_camino)
        return p_ri[:3]
```

```python
import sim

class LaserScanner2D:
    def __init__(self, clientid):
        self.clientID = clientid
        self.handle = None

    def start(self, name='/youBot/LaserScanner2D'):
        errorcode, handle = sim.simxGetObjectHandle(self.clientID, name, sim.simx_opmode_oneshot_wait)
        self.handle = handle

    def get_laser_data(self):
        """
        This reads the laserdata signal in Coppelia and returns it.
        The laserdata signal must be defined as in the Youbot2.ttt environment.
        """
        # Datos recgidos desde coppeliaasim con una serie de modificaciones en el script del laser
        error, dist = sim.simxGetStringSignal(self.clientID, 'laserdata', sim.simx_opmode_oneshot_wait)
        # TODO: after unpacking the floats, some more-readable data structure should be built.
        # Los datos se recogen de derecha a izquierda mirando el laser por detrás
        dist = sim.simxUnpackFloats(dist)
        return dist
```

```python
import sim
from robots.robot import Robot
import numpy as np
from artelib.seriallink import SerialRobot
from artelib.homogeneousmatrix import HomogeneousMatrix
from artelib.path_planning import filter_path
# standard delta time for Coppelia, please modify if necessary
DELTA_TIME = 50.0/1000.0


class YouBotRobot(Robot):
    def __init__(self, clientid):
        Robot.__init__(self)
        self.clientID = clientid
        self.wheeljoints = None
        self.joints = None
        self.dummy = None
        self.gripper = None
        self.joint_directions = None
        self.b = 0.1598  # En metros recogidos en coppeliasim 0.1598
        self.d = 0.228  # En metros recogidos en coppeliasim 0.228
        self.r = 0.045  # En metros recogidos en coppeliasim 0.045
        # complete all data from base class
        self.epsilonq = 0.005
        self.coordenades = None
        self.posiciones = []

        # para la filter_path
        self.DOF = 5
        self.q_current = np.zeros((1, self.DOF))
        # Brazo robotico
        joint_ranges = np.array([[-169, -90, -131, -102, -90],
                                 [169, 75, 131, 102, 90]])
        self.joint_ranges = joint_ranges * np.pi / 180.0
        self.q_current = np.zeros((1, 5))
        # DH parametres
        self.serialrobot = SerialRobot(n=5, T0=np.eye(4), name='Arm_Youbot')
        self.serialrobot.append(th=0, d=0.147, a=-0.033, alpha=-np.pi / 2, link_type='R')
        self.serialrobot.append(th=-np.pi / 2, d=0, a=0.155, alpha=0, link_type='R')
        self.serialrobot.append(th=0, d=0, a=0.135, alpha=0, link_type='R')
        self.serialrobot.append(th=np.pi / 2, d=0, a=0, alpha=np.pi / 2, link_type='R')
        self.serialrobot.append(th=0, d=0.218, a=0, alpha=0, link_type='R')

        # CONTROL DEL BRAZO
        self.DOF = 5
        self.q_current = np.zeros((1, self.DOF))

        # maximum joint speeds (rad/s)
        max_joint_speeds = np.array([90, 90, 90, 90, 90, 90])
        self.max_joint_speeds = max_joint_speeds * np.pi / 180.0

        self.max_iterations_inverse_kinematics = 15000
```

```
        self.max_error_dist_inversekinematics = 0.01
        self.max_error_orient_inversekinematics = 0.01
        # Matrices que define como ve el sistema de referencia del robot al brazo
        self.r_A_cero = np.array([[1, 0, 0, -0.16],
                                  [0, 1, 0, 0],
                                  [0, 0, 1, 0.1],
                                  [0, 0, 0, 1]])
        self.cinco_A_c = np.array([[0, 1, 0, 0.02748],
                                   [-1, 0, 0, -0.00114],
                                   [0, 0, 1, -0.06622],
                                   [0, 0, 0, 1]])
        self.qs = None  # almacenar posiciones del brazo

    def start(self, base_name='/youBot', joint_name='youBotArmJoint'):
        armjoints = []
        # Get the handles of the relevant objects
        errorCode, robotbase = sim.simxGetObjectHandle(self.clientID, '/youBot/youBot_ref', sim.simx_opmode_oneshot_wait)
        # handles to the wheels
        errorCode, fl = sim.simxGetObjectHandle(self.clientID, base_name + '/rollingJoint_fl',
                                                sim.simx_opmode_oneshot_wait)
        errorCode, rl = sim.simxGetObjectHandle(self.clientID, base_name + '/rollingJoint_rl',
                                                sim.simx_opmode_oneshot_wait)
        errorCode, rr = sim.simxGetObjectHandle(self.clientID, base_name + '/rollingJoint_rr',
                                                sim.simx_opmode_oneshot_wait)
        errorCode, fr = sim.simxGetObjectHandle(self.clientID, base_name + '/rollingJoint_fr',
                                                sim.simx_opmode_oneshot_wait)
        wheeljoints = [rr, fr, fl, rl]

        errorCode, q1 = sim.simxGetObjectHandle(self.clientID, base_name + '/' + joint_name + '0',
                                                sim.simx_opmode_oneshot_wait)
        errorCode, q2 = sim.simxGetObjectHandle(self.clientID, base_name + '/' + joint_name + '1',
                                                sim.simx_opmode_oneshot_wait)
        errorCode, q3 = sim.simxGetObjectHandle(self.clientID, base_name + '/' + joint_name + '2',
                                                sim.simx_opmode_oneshot_wait)
        errorCode, q4 = sim.simxGetObjectHandle(self.clientID, base_name + '/' + joint_name + '3',
                                                sim.simx_opmode_oneshot_wait)
        errorCode, q5 = sim.simxGetObjectHandle(self.clientID, base_name + '/' + joint_name + '4',
                                                sim.simx_opmode_oneshot_wait)
        # handles to the armjoints
        armjoints.append(q1)
        armjoints.append(q2)
        armjoints.append(q3)
        armjoints.append(q4)
        armjoints.append(q5)

        # must store the joints
        self.wheeljoints = wheeljoints
        self.joints = armjoints
        self.dummy = robotbase
        # self.gripper = gripper
        self.joint_directions = [1, -1, -1, -1, 1]
```

```
112
113     def set_base_speed(self, vx, vy, wz):
114         """
115         Given a speed in forward/backward direction
116         - left/right directino
117         and - rotation speed
118         Computes the speeds of each wheel so a to apply the commanded speed to the robot base.
119         """
120
121         """wz = wz * ((2*np.pi)/360)"""
122         direccion = np.array([vx, vy, wz])  # la velocidad está en m/s
123         cinematica = (1 / self.r) * np.array([[1, -1, -(self.b + self.d)],
124                                               [1, 1, -(self.b + self.d)],
125                                               [1, -1, (self.b + self.d)],
126                                               [1, 1, (self.b + self.d)]])
127         w = np.dot(cinematica, direccion)
128
129         error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[0],
130                                                targetVelocity=w[0],
131                                                operationMode=sim.simx_opmode_oneshot)
132
133         error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[1],
134                                                targetVelocity=w[1],
135                                                operationMode=sim.simx_opmode_oneshot)
136
137         error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[2],
138                                                targetVelocity=w[2],
139                                                operationMode=sim.simx_opmode_oneshot)
140
141         error = sim.simxSetJointTargetVelocity(clientID=self.clientID, jointHandle=self.wheeljoints[3],
142                                                targetVelocity=w[3],
143                                                operationMode=sim.simx_opmode_oneshot)
144
145     def get_pos_ori(self):
146         errorcode, pos = sim.simxGetObjectPosition(clientID=self.clientID, objectHandle=self.dummy,
147                                                    relativeToObjectHandle=-1, operationMode=sim.simx_opmode_oneshot_wait)
148         errorcode, ori = sim.simxGetObjectOrientation(clientID=self.clientID, objectHandle=self.dummy,
149                                                       relativeToObjectHandle=-1, operationMode=sim.simx_opmode_oneshot_wait)
150         # devuelve posición en metros y ángulo en radianes
151         coordenades = np.array([pos[0], pos[1], ori[2]])
152         return coordenades

155     def inverse_kinematics(self, t_position, t_orientation,codo):
156         gamma = 0
157         theta = 0
158         Ttarget = HomogeneousMatrix(t_position, t_orientation)
159         o_p_m = Ttarget[:, 3] - (0.218 * Ttarget[:, 2])
160         #  calculo q
161         q_1 = 0
162
163         A01 = self.serialrobot.transformations[0].dh(q_1)
164         # calculo q2 y q3
165         uno_p_m = np.dot(A01.inv().toarray(), o_p_m)
166         # print(uno_p_m)
167         betha = np.arctan2(-uno_p_m[1], uno_p_m[0])
168         r = (uno_p_m[0] ** 2 + uno_p_m[1] ** 2) ** 0.5
169         r = np.round(r, 12)
```

```
        # print(r)
        L1 = self.serialrobot.transformations[1].a
        L2 = self.serialrobot.transformations[2].a
        gamma_v = (-L2 ** 2 + L1 ** 2 + r ** 2) / (2 * L1 * r)

        theta_v = (-r ** 2 + L1 ** 2 + L2 ** 2) / (2 * L1 * L2)

        if np.abs(gamma_v) <= 1.0:
            gamma = np.arccos(gamma_v)
        else:
            print('WARNING_GAMMA: ONE OF THE INVERSE KINEMATIC SOLUTIONS IS NOT FEASIBLE ( ROBOT). '
                  'The point is out of the workspace')
        if np.abs(theta_v) <= 1:
            theta = np.arccos(theta_v)
        else:
            print('WARNING_THETA: ONE OF THE INVERSE KINEMATIC SOLUTIONS IS NOT FEASIBLE ( ROBOT). '
                  'The point is out of the workspace')
        if codo==1:
            q_2 = np.pi / 2 - betha - gamma
            q_3 = np.pi - theta
        elif codo==2:
            q_2 = np.pi / 2 - betha + gamma
            q_3 = -np.pi + theta
```

```
        A12 = self.serialrobot.transformations[1].dh(q_2)
        A23 = self.serialrobot.transformations[2].dh(q_3)
        # calculo de q_4
        A03 = np.dot(A01.toarray(), np.dot(A12.toarray(), A23.toarray()))

        tres_p_m = np.dot(np.linalg.inv(A03), Ttarget[:4, 3])

        q_4 = np.arctan2(tres_p_m[1], tres_p_m[0])

        A34 = self.serialrobot.transformations[3].dh(q_4)
        # calculo q_5
        Q = np.dot(np.dot(np.dot(np.dot(A34.inv(), A23.inv()), A12.inv()), A01.inv()), Ttarget)

        q_5 = np.arctan2(Q[0, 1], Q[0, 0])
        q_t = np.array([q_1, q_2, q_3, q_4, q_5])
        return q_t
```

```
def moveAbsJ(self, q_target, precision=True):
    """
    Commands the robot to the specified joint target positions.
    The targets are filtered and the robot is not commanded whenever a single joint is out of range.
    """
    self.qs = q_target  # almacenar último movimineto del brazo
    # remove joints out of range and get the closest joint
    total, partial = self.check_joints(q_target)
    if total:
        self.set_joint_target_positions(q_target, precision=precision)
    else:
        print('moveABSJ ERROR: joints out of range')
```

```python

    def moveJ(self, target_position, target_orientation,codo , precision=True):
        """
        Commands the robot to a target position and orientation.
        All solutions to the inverse kinematic problem are computed. The closest solution to the
        current position of the robot q0 is used
            """
        q0 = self.q_current

        self.qs = self.inverse_kinematics(t_position=target_position,
                                          t_orientation=target_orientation,codo=codo)  # almacenamos último movim
        # remove joints out of range and get the closest joint
        self.qs = filter_path(self, q0, [self.qs])
        # comandar al robot hasta
        self.set_joint_target_positions(self.qs, precision=precision)

    def pos_aruco_w(self, punto_c, pos_robot):
        w_a_r = np.array([[np.cos(pos_robot[2]), -np.sin(pos_robot[2]), 0, pos_robot[0]],
                          [np.sin(pos_robot[2]), np.cos(pos_robot[2]), 0, pos_robot[1]],
                          [0, 0, 1, 0],
                          [0, 0, 0, 1]])

        A_T = self.matriz_homogenea(self.qs)
        w_a_c = np.dot(w_a_r, np.dot(self.r_A_cero, np.dot(A_T.toarray(), self.cinco_A_c)))
        pos_pieza = w_a_c @ punto_c
        return pos_pieza


    def pos_aruco_t(self, punto_w, pos_robot):
        w_a_r = np.array([[np.cos(pos_robot[2]), -np.sin(pos_robot[2]), 0, pos_robot[0]],
                          [np.sin(pos_robot[2]), np.cos(pos_robot[2]), 0, pos_robot[1]],
                          [0, 0, 1, 0],
                          [0, 0, 0, 1]])

        cero_a_w = np.linalg.inv(self.r_A_cero) @ np.linalg.inv(w_a_r)

        pieza = cero_a_w @ punto_w

        return pieza

    def matriz_homogenea(self, q):
        # Matriz de transformación del robot
        A_T = self.serialrobot.directkinematics(q)
        """rot = A_T[:3, :3]
        rotation_obj = R.from_matrix(rot)
        euler_angles = rotation_obj.as_euler('XYZ', degrees=False)"""
        return A_T
```

```
113 class YouBotGripper():
114     def __init__(self, clientID):
115         self.clientID = clientID
116         self.joints = None
117 
118     def start(self, name='youBotGripperJoint'):
119         errorCode, gripper_joint1 = sim.simxGetObjectHandle(self.clientID, name+'1',
120                                                             sim.simx_opmode_oneshot_wait)
121         errorCode, gripper_joint2 = sim.simxGetObjectHandle(self.clientID, name + '2',
122                                                             sim.simx_opmode_oneshot_wait)
123 
124         self.joints = [gripper_joint1, gripper_joint2]
125 
126     def open(self, precision=False):
127         """sim.simxSetJointTargetPosition(clientID=self.clientID, jointHandle=self.joints[0],
128                                           targetPosition=-0.05, operationMode=sim.simx_opmode_oneshot)"""
129         sim.simxSetJointTargetPosition(clientID=self.clientID, jointHandle=self.joints[1],
130                                        targetPosition=-0.05, operationMode=sim.simx_opmode_oneshot)
131         if precision:
132             for i in range(10):
133                 sim.simxSynchronousTrigger(clientID=self.clientID)
134 
135     def close(self, precision=False):
136         """sim.simxSetJointTargetPosition(clientID=self.clientID, jointHandle=self.joints[0],
137                                           targetPosition=0.0, operationMode=sim.simx_opmode_oneshot)"""
138         errocode = sim.simxSetJointTargetPosition(clientID=self.clientID, jointHandle=self.joints[1], targetPo
139 
140         if precision:
141             for i in range(10):
142                 sim.simxSynchronousTrigger(clientID=self.clientID)
```

```python
import sys
import time

import sim


class Simulation():
    def __init__(self):
        self.clientID = None

    def start(self):
        sim.simxFinish(-1)
        clientID = sim.simxStart('127.0.0.1', 19997, True, True, 5000, 5)
        self.clientID = clientID
        if clientID != -1:
            print("Connected to remote API server")
            # stop previous simiulation
            sim.simxStopSimulation(clientID=clientID, operationMode=sim.simx_opmode_blocking)
            time.sleep(3)
            sim.simxStartSimulation(clientID=clientID, operationMode=sim.simx_opmode_blocking)
            # enable the synchronous mode
            sim.simxSynchronous(clientID=clientID, enable=True)
        else:
            print("Connection not successful")
            sys.exit("Connection failed,program ended!")
        return clientID


    def stop(self):
        sim.simxStopSimulation(self.clientID, sim.simx_opmode_oneshot_wait)
        sim.simxFinish(self.clientID)

    def wait(self, steps=1):
        """
        Wait n simulation steps.
        """
        for i in range(0, steps):
            sim.simxSynchronousTrigger(clientID=self.clientID)


```