UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Planificación de movimiento de robots mediante la descomposición del entorno en celdas

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

AUTOR/A: Esteve Gonzálvez, Iván

Tutor/a: Zotovic Stanisic, Ranko

CURSO ACADÉMICO: 2021/2022

Agradecimientos

Quiero agradecer a toda mi familia por su apoyo cuando decidí dejar mi tierra para venir a Valencia para realizar estos estudios, que han llegado hasta el presente proyecto. Mis más sinceros agradecimientos a la Univèrsitat Politècnica de Valencia por brindarme la oportunidad de llevar a cabo mi formación profesional en sus instalaciones, así como a la Escuela Técnica Superior de Ingeniería del Diseño por los cuatro años en los que he podido asistir a sus clases.

Gracias también a mis amigos y todos los que me han acompañado durante el transcurso de estos años.

Por último, gracias al profesor Ranko Zotovic Stanisic por haber accedido a ser mi tutor y abrirme las puertas a esta disciplina tan interesante.

Resumen

El presente proyecto aborda un campo concreto de la robótica móvil y la inteligencia artificial: la planificación de movimiento. Se trata de una rama que explora nuevas técnicas y algoritmos que permitan generar trayectorias para ser recorridas por un robot móvil, procurando evitar la colisión con los obstáculos del entorno. La planificación de movimiento es un campo amplio que presenta diversidad de metodologías en sus algoritmos de cálculo de rutas. El presente trabajo de fin de grado ha pretendido explorar la senda de la 'descomposición en celdas' del entorno, a partir de un trabajo previo de esta filosofía (autores: antiguos alumnos JL. Guardiola y R. Sebastiá).

Mediante el uso del software de programación Matlab y de la toolbox facilitada por el tutor de este trabajo (Ranko Zotovic Stanisic), se ha implementado nuevos algoritmos que permiten realizar tratamientos sobre el entorno, descomponer en celdas y hallar rutas óptimas.

Abstract

This project explores a specific field of study related to mobile robotics and artificial intelligence: motion planning. This discipline explores new techniques and algorithms that generate trajectories to be traveled by a mobile robot, trying to avoid the collision with the obstacles in the environment. Motion planning is a broad field that includes a diversity of methodologies in its route calculation algorithms. This end-of-degree project has attempted to explore the technique of 'decomposition into cells' of the environment, based on a previous work on this philosophy (authors: former students JL. Guardiola and R. Sebastiá).

Through the use of the Matlab programming software and the toolbox provided by the professor of this work (Ranko Zotovic Stanisic), new algorithms have been implemented. They allow treatments to be carried out on the environment, decompositions into cells and searches of optimal routes.

Índice General

Índice de Figuras

Índice de Tablas

1. Introducción

El proyecto desarrollado se enmarca en el ámbito de la robótica móvil, campo de estudio que observa, analiza y busca solución a problemas mecánicos y cinemáticos para robots con capacidades móviles. En los últimos años, los robots móviles se han incorporado en la industria 4.0, permitiendo la automatización de procesos logísticos, de almacenamiento, asistiendo como elementos colaborativos… Sus capacidades han sobresalido de tal forma que, incluso algunos servicios del sector terciario (restaurantes, hospitales, farmacias, oficinas…) están comenzando a introducirlos para desempeñar trabajos.

La principal ventaja que aportan los robots móviles es la flexibilidad o versatilidad. Permiten la automatización de procesos intralogísticos casi de cualquier índole, con la capacidad de adaptarse a problemas no anticipados de detección de obstáculos y manteniendo su productividad durante largas jornadas. El aumento de la eficiencia del proceso en el que participa, confiere a este nuevo empleado una atractiva característica competitiva.

***Figura 1**. Robot móvil industrial en un proceso interno logístico.*

Por otro lado, los robots móviles presentan ciertas características accesorias de gran utilidad: conectividad entre sí, aportación de datos del proceso en tiempo real (que permite identificar inconvenientes como cuellos de botella para abordarlos rápidamente), recopilación de datos para uso estadístico, etc.

Debido a los factores mencionados, la disciplina de la robótica móvil se encuentra en actual desarrollo y auge, y casi cada institución universitaria dedica una parte de su investigación a esta materia, en pos de encontrar soluciones más eficientes al sensorizado, modelado y control de los robots móviles. En este contexto, sobresale el tema de estudio particular del presente proyecto: la planificación de movimiento de robots (Motion Planning en inglés). Esta disciplina, que parte a raíz de la robótica móvil, se encarga de encontrar soluciones eficientes al problema del cálculo de la ruta, desde un punto inicial hasta un punto final, que debe trazar un robot móvil evitando la colisión con los obstáculos del entorno.

Dentro del diagrama del control de un robot móvil, corresponde a la etapa de ‘Generación de trayectoria’, parte del control sobre el modelo cinemático del robot. Para abordar esta etapa se debe disponer de los datos de entrada, entre ellos, el objetivo de destino e información del entorno que rodea al robot para evitar la colisión con los obstáculos. Esta información puede ser reconocida por sensores, precargada en formato de planos del área de trabajo, u otros métodos auxiliares.

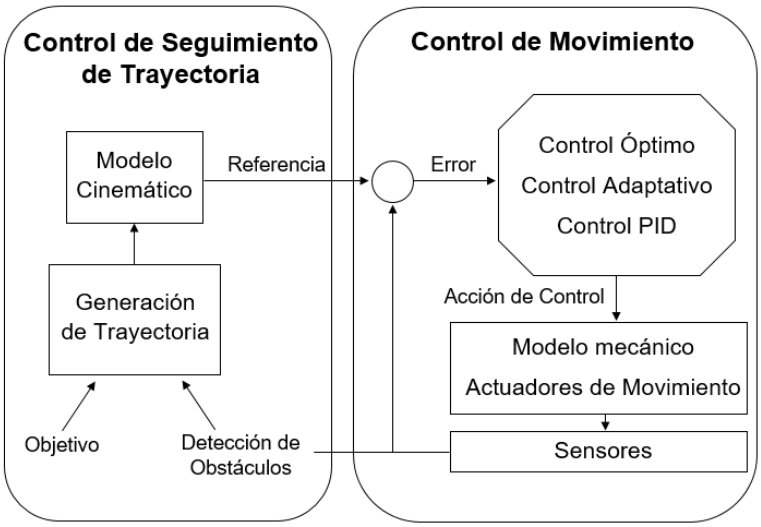

***Figura 2**. Diagrama del control de un robot móvil.*

La planificación de movimiento es un campo de estudio muy amplio, con diversas perspectivas y extensión. Aunque es un campo extensible a las tres dimensiones del entorno real para procesos más complejos, para el proyecto desarrollado la planificación se reduce al movimiento en dos dimensiones sobre la superficie de un plano, simulando el movimiento de un robot móvil terrestre por el suelo.

La metodología de análisis suele consistir en la representación, sobre un espacio euclídeo de dos dimensiones (simulando la superficie de movimiento), de un robot de características físicas conocidas (dimensiones, capacidad de giro, etc.) y un entorno representado mediante obstáculos poligonales. A partir de diversos algoritmos basados en geometría computacional y redes de nodos, se debe obtener la ruta óptima entre el punto inicial y final. Estos algoritmos tratan de analizar el coste (entendido en términos de distancia, capacidad de giro del robot móvil, seguridad y trazabilidad de la ruta, energía, tiempo, y aquellos otros criterios propios del ámbito de aplicación) de las diversas rutas posibles, encontrando la que posee menor coste en su recorrido.

El proceso de búsqueda de la ruta óptima puede seguir distintos razonamientos. Como se expondrá más adelante en la sección '*2.3.2. Configuración del espacio de trabajo y algoritmos de cálculo de ruta*' (pág. 18, véase *Índice General*), el cálculo del camino a recorrer por el robot se automatiza a partir de algoritmos de muy diversa índole, desde exploración de grafos, tratamientos geométricos como descomposición en celdas, árboles probabilísticos, etc.

La planificación de movimientos es utilizada en numerosos ámbitos en los que son necesarias soluciones adaptativas al problema del movimiento del robot, situaciones en las que premia la eficiencia y la seguridad, y en las que no existe una solución única, por lo que alcanzar la solución óptima se vuelve una tarea compleja. Entre los diversos campos de aplicación de la planificación de movimiento sobresalen:

- Drones, UAVs, aeronaves no tripuladas…
- Coches, coches autónomos, vehículos guiados (AGVs), vehículos no guiados (AMRs)…
- Monociclos y biciclos automáticos.
- Aeronaves con navegación asistida: aviones, helicópteros, etc.

- Robots acuáticos autónomos como AUVs.

1.1. Toolbox de Matlab

El proyecto surge a raíz de una toolbox desarrollada a través del software de Matlab, cuyo propietario es el actual profesor de la *Universitat Politècnica de Valencia*, *Escuela Técnica Superior de Ingeniería Industrial*, *Departamento de Ingeniería de Sistemas y Automática*: *Ranko Zotovic Stanisic*.

Matlab (abreviatura de MATrix LABoratory) es un software privativo desarrollado por MathWorks (*© 1994-2022 The MathWorks, Inc.*), cuyo autor original es Cleve Moler. Se trata de una plataforma de programación y cómputo numérico que permite la creación de modelos y algoritmos orientados al análisis científico y de ingeniería.

El paquete de Matlab incluye dos herramientas adicionales que expanden sus prestaciones: Simulink y GUIDE. En el proyecto presente, no se ha hecho uso de estas herramientas. No obstante, Matlab también ofrece las denominadas toolboxes (traducido literalmente como *cajas de herramientas*). Se trata de paquetes de archivos de Matlab que pueden incluir código, datos, apps, ejemplos y documentación. Oficialmente, Matlab ofrece sus propias toolboxes, tales como '*Parallel Computing*' (permite resolver problemas con un uso intensivo de cálculos y datos mediante procesadores multinúcleo, GPUs y clusters de ordenadores), '*Control Systems*' (proporciona algoritmos y apps para analizar, diseñar y ajustar sistemas de control lineales de forma metódica), '*Image Processing and Computer Vision*' (provee un entorno completo para la visualización, exploración y análisis geoespacial de datos como mapas de vectores, imágenes georreferenciadas y datos de terreno)…

Los módulos mencionados se pueden obtener a partir de la licencia de Matlab y permiten incorporar nuevas funcionalidades de uso académico e industrial. Además, este tipo de paquetes no son exclusivos de desarrollo oficial, sino que los usuarios del programa poseen la capacidad de crear y compartir sus propios paquetes. Bajo esta línea de trabajo, se enmarca la toolbox de la que nace el proyecto de fin de grado.

El profesor *Ranko Zotovic Stanisic* ha facilitado su toolbox de Matlab sobre planificación de movimiento de robots y el objeto del trabajo ha sido ampliar sus funcionalidades, haciendo uso de los archivos, scripts y funciones existentes.

Específicamente, se ha pretendido ampliar en la senda de la descomposición en celdas del entorno, para su posterior tratamiento en rutas. Asimismo, se ha implementado dos herramientas auxiliares de análisis del entorno: expansión del entorno mediante suma de Minkowski y el diagrama de Voronoi. Ambas herramientas se detallarán en sus correspondientes secciones más adelante en el documento (véase*: Índice General*).

1.2. Planificación de movimiento de robots mediante descomposición en celdas del entorno

Existen numerosos métodos de planificación de movimientos, métodos que abordan el problema desde perspectivas muy diversas. Además, el problema de 'planificar un movimiento' puede dividirse en distintas etapas en función del método escogido. Estas etapas pueden consistir en modelizar el entorno de trabajo, realizar pretratamientos sobre el entorno (ejemplo: expansión del entorno), mapas de potencial en función del coste, etc. Como se ha mencionado previamente, de entre todos los itinerarios de estudio en esta disciplina, se procede a abordar la 'descomposición en celdas del entorno'.

Este procedimiento de trabajo consiste en la diferenciación de las regiones del entorno en dos posibles clases generales: "libre", es decir, transitable por el robot, y "ocupada", en otras palabras, la existencia de un obstáculo, límite del entorno o margen de seguridad en esa región del espacio, impide formular una ruta segura para que el robot la transite. Estas dos clases generales se adecúan posteriormente al método específico desarrollado de entre los posibles procedimientos para la descomposición en celdas, pero permiten una visión intuitiva del funcionamiento general. Tras la diferenciación, el algoritmo deberá calcular la ruta óptima que conecta el punto inicial y final teniendo en cuenta que solamente al área designada como "libre" puede ser recorrida.

Esta senda de trabajo de descomposición en celdas para la toolbox de Matlab mecionada tiene su origen en un trabajo previo realizado por los autores, estudiantes de Máster en su momento: *JL. Guardiola y R. Sebastiá*. Su trabajo consistió en la descomposición en celdas uniformes del entorno y, tras la aplicación de un campo potencial de coste (en función de la distancia), se obtenía la ruta óptima.

El trabajo mencionado puso la semilla que ha retomado el presente proyecto para ampliar la toolbox de Matlab. De entre las diversas técnicas disponibles, este proyecto ha revisado el trabajo previo de celdas uniformes y se ha desarrollado e implementado cuatro nuevos métodos: celdas verticales, a bandas verticales y mediante configuración trapezoidal, una versión propia de las celdas uniformes y Quadtree.

Estos procedimientos son entendidos como tratamientos únicamente sobre el entorno. Además de ellos, se ha desarrollado algoritmos de búsqueda de rutas. Por un lado, se ha hecho uso de la función ya descrita en la toolbox que contiene el algoritmo de Dijkstra, adecuándola a la descomposición en celdas. Por otro lado, se ha revisado los campos potenciales y su posible aplicación en otros problemas, además de una versión propia.

Todos los procedimientos mencionados y demás pretratamientos del entorno que se han podido abordar durante la realización de este proyecto serán estudiados en sus respectivas secciones (véase: *Índice General*).

2. Estado del Arte

El origen de la robótica como es entendida en la actualidad es bastante reciente, concretamente del siglo XX, ya que los primeros ejemplos y alusiones al término se localizan durante este siglo.

No obstante, su concepto más genérico, "autómata", es una idea que nace incluso en la prehistoria, cuando las estatuas de algunos dioses o reyes despedían fuego de los ojos, o poseían brazos mecánicos operados por sacerdotes. En este contexto, el primero en recopilar por escrito los conocimientos sobre estos sistemas mecánicos fue Herón de Alejandría en su obra *Pneumática y Autómata*, durante el siglo I d. C. El mismo Herón fue el ideador de las llamadas 'Puertas de Herón' que pueden ser entendidas como las primeras puertas automáticas de la historia.

Más tarde, destacados ingenieros e inventores de la historia crearon nuevas máquinas que protagonizaron la evolución de los autómatas en la historia, hasta llegar al concepto actual de robot. Entre los referentes más importantes y sus aportaciones, se puede mencionar:

- Leonardo da Vinci, quien fue el autor de dos autómatas: 'el caballero mecánico' y 'el león mecánico'.
- Jacques de Vaucanson, ingeniero francés que creó el pato de Vaucanson.
- Pierre Jaquet-Droz, quien construyó los tres autómatas de Droz: el dibujante, el músico y el escritor, los cuales realizaban sus tareas automáticamente.
- Los Karakuri japoneses, muñecos autómatas capaces de llevar a cabo tareas simples como servir el té.

***Figura 3**. Autómatas más relevantes de la historia.*
A la izquierda, león mecánico de Leonardo. En el centro, pato de Vaucanson. A la de derecha, muñecos Karakuri japoneses.

Con la llegada del siglo XX, el campo de la automatización protagonizó un gran avance. Aunque no fue hasta finales de la década de los 40 y principios de los 50 cuando aparecieron los que pueden ser denominados como 'los primeros robots', el término tiene su origen en el año 1920, cuando el escritor Karel Kapek publicó su novela Rossum's Universal Robot. Fue este escritor checo quien acuñó el término "robot" a partir de la palabra checa "robota", que significa esclavo, servidumbre o trabajo forzado. En su novela, el autor utiliza el término para referirse a unas máquinas de aspecto humano que realizaban los trabajos que las personas no deseaban realizar.

En los años venideros, las alusiones al término continuaron sucediendo. En 1927, en la película Metrópolis aparece un robot de aspecto antropomórfico que es utilizado para suplantar a otras personas. Más tarde, entre 1939 y 1940 la Exposición Universal (exposiciones de gran envergadura celebradas por todo el mundo desde la segunda mitad del siglo XIX), alojó la exhibición de un robot humanoide. Su nombre era *ELEKTRO*. Concebido por Westinghouse Electric Corporation, podía caminar por comando de voz y decir 700 palabras. Además, fumaba cigarrillos, inflaba globos y era capaz de mover la cabeza y los brazos.

Pasaron los años y, en 1942, fue acuñado oficialmente el término 'Robótica'. Ello se debe a Isaac Asimov, destacado profesor de la universidad de Boston que sobresalió como novelista de ciencia ficción. En sus obras, inicialmente en *Círculo vicioso* (1942) y, posteriormente logrando gran popularidad, en *Yo, robot* (1950), desarrolló las llamadas 'tres leyes de la robótica', una especie de código moral que limitaba las capacidades de los robots para poder dañar a la humanidad.

2.1. Los primeros robots

El primer robot autónomo electrónico de comportamiento complejo aparece entre 1948 y 1949, de mano de William Gray Walter. El objetivo de este neurólogo norteamericano experto en robótica era demostrar que las conexiones adecuadas entre un número pequeño de células del cerebro podrían desarrollar comportamientos muy complejos. Sus primeros robots, llamados *Elmer* y *Elsie*, aunque también conocidos como 'las tortugas de Bristol', eran capaces de realizar fototaxia (habilidad que poseen ciertas células de orientarse hacia el foco de intensidad lumínica), por lo que podían buscar la manera de recargarse cuando la batería era baja.

En 1951, Walter publicó *Una Máquina que aprende*, en el que exponía cómo sus robots mecánicos más adelantados demostraban comportamientos inteligentes mediante el aprendizaje por reflejos condicionados.

***Figura 4**. Tortuga de Bristol.*
Primera máquina robótica autónoma de la historia.

Posteriormente, en el año 1954, se creó el primer robot industrial digitalmente operativo y programable, de nombre *Unimate*. Su inventor fue George Devol, quien, junto al ingeniero y emprendedor Joseph F. Engelberger, fundaron en 1953 la primera empresa de robótica de la historia: Unimation.

La primera instalación de este robot industrial se produjo en 1961, en una cadena de montaje de General Motors, para transportar piezas calientes de metal de una máquina de fundición a presión y colocarlas en un líquido refrigerante. También realizaba tareas de soldadura sobre el chasis del vehículo, una tarea hasta entonces peligrosa para los trabajadores puesto que corrían el riesgo de inhalar peligrosos gases de combustión. Todo ello era posible debido a su morfología, un brazo articulado que, en versiones posteriores, lograría alcanzar hasta los seis grados de libertad. Este primer robot industrial dio origen a la industria robótica moderna, popularizando los denominados brazos robóticos.

***Figura 5**. Unimate, primer robot industrial programable.*

Durante los años venideros se llevaron a cabo nuevos avances y aparecieron en el mercado nuevos robots industriales:

- En 1954, Barrett Electronics Corporation saca a la luz el primer vehículo eléctrico que no necesitaba conductor humano, lo que se conoce actualmente como el primer AGV (Vehículo Autónomo Guiado).
- Robot paletizador de Okura Yusoki, compañía japonesa que comenzó su comercialización en 1963.
- Robot Versatran en 1967, construido por 'The American Machine and Foundry'.
- 'Robot Tentáculo' creado por Marvin Minsky (notorio exponente en el campo de la inteligencia artificial) en 1968.
- Robot Shakey, creado por SRI International en 1968, primer robot móvil capacitado para razonar en sus acciones. Shakey fue el primer robot móvil del mundo que, gracias al software y hardware, le permitía percibir y comprender el entorno, aunque de forma limitada.
- 'Brazo Stanford', cuyo inventor fue Victor Scheinman (pionero estadounidense en el campo de la robótica) en 1972 (aunque algunas fuentes citan 1969).
- En 1970 la NASA, con la colaboración del Jet Propulsion Laboratory, creó el Mars Rover (primera versión de los Rovers utilizados actualmente) con la finalidad de explorar Marte. Integraba un brazo mecánico, sensores de proximidad, un dispositivo telemétrico láser y cámaras estéreo.

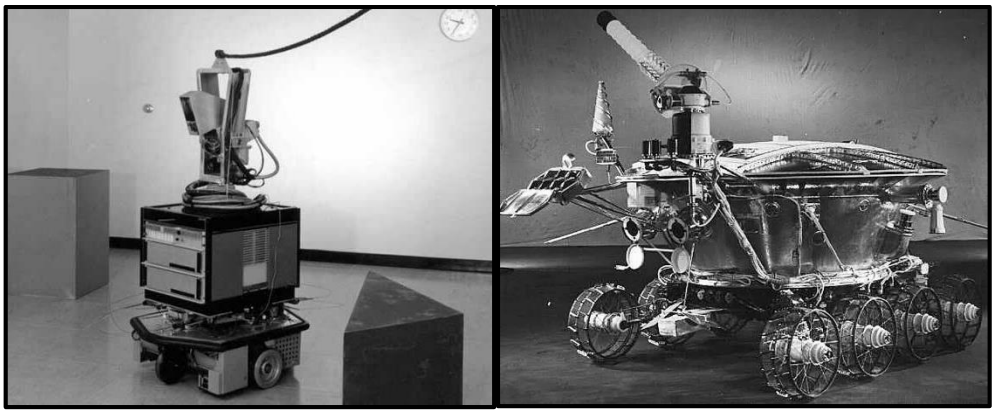

***Figura 6**. Exponentes de los primeros robots móviles.*
A la izquierda, robot Shakey. A la derecha, el Mars rover desarrollado en 1970 para la misión Mars de 1979.

El siguiente salto en la robótica se produjo en 1971, cuando la Unión Soviética logró aterrizar exitosamente una nave, la *Mars 3*, por primera vez en la superficie de Marte. Como parte del programa Mars, el objetivo era la obtención de imágenes de la superficie marciana y de las nubes, determinar la temperatura, estudiar la topografía, composición y propiedades físicas de la superficie, medir las propiedades de la atmósfera… Sin embargo, se perdió el contacto con la nave pocos segundos después del amartizaje.

Dos años más tarde, en 1973, la compañía alemana KUKA Robot Group introdujo en el mercado un nuevo robot industrial, el *Famulus*. Se trata del primer robot con seis ejes de accionamiento electromecánico, lo que produjo una revolución industrial en su momento. Incluso a día de hoy, casi 50 años más tarde, KUKA se sitúa como uno de los principales fabricantes mundiales de robots industriales.

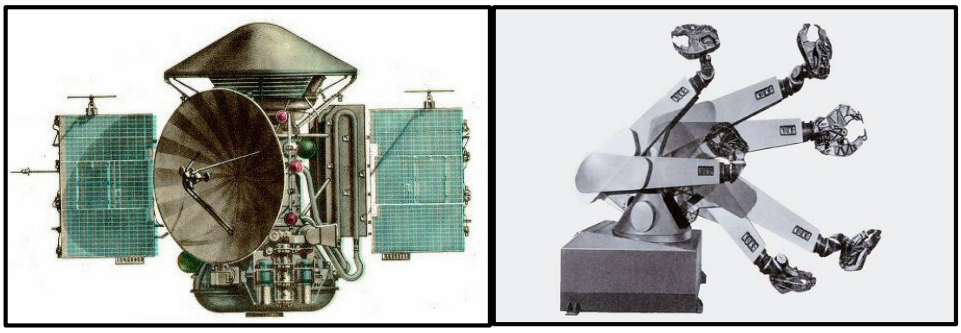

***Figura 7**. Exponentes de la robótica de la década de los 70.*
A la izquierda, orbitador soviético Mars 3. A la derecha, robot industrial alemán Famulus.

El siguiente avance más notorio en la robótica se produjo en 1975, con la llegada del robot industrial *PUMA* (Programmable Universal Machine for Assembly) desarrollado por Victor Scheinman en la empresa pionera en robótica Unimation. El modelo *PUMA-560* (manufacturado por Nokia Robotics tras diversas compras y traspasos de la empresa original Unimation) se convertiría posteriormente como uno de los brazos robot más populares de la década de los 80.

Más tarde, en 1976, los Estados Unidos de América lograron la hazaña rusa de aterrizar una nave en Marte. Ello fue protagonizado por el robot estadounidense *Viking I* desarrollado por la NASA, que pudo tomar la primera fotografía de Marte de la historia. Este logro no fue la

única de sus tareas, también realizó importantes experimentos meteorológicos, geológicos y biológicos (búsqueda de vida en forma de microorganismos).

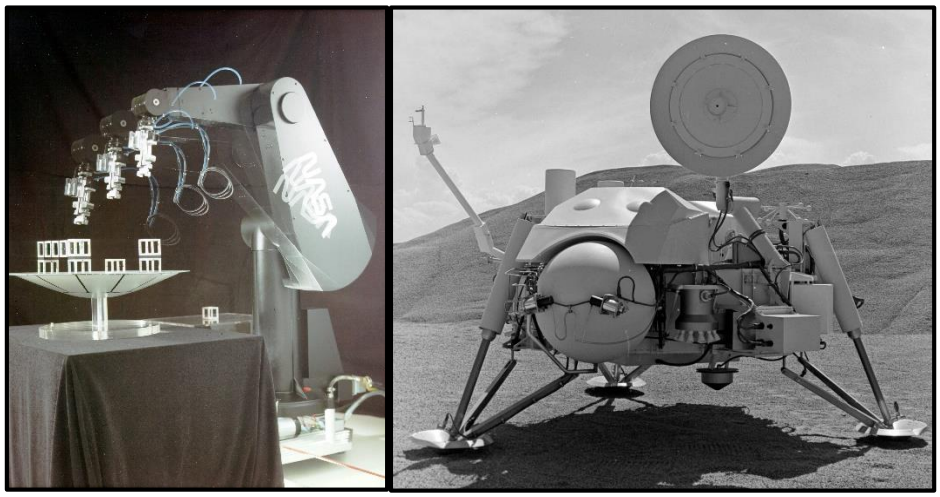

***Figura 8**. Brazo robot industrial PUMA (izquierda) y nave Viking I (derecha).*

Unos años después, en 1982, la robótica se trasladó de nuevo al dominio de la opinión pública al popularizarse la obra de ficción *El robot completo*, escrito por Isaac Asimov (autor de las tres leyes de la robótica). La obra, una colección de sus escritos entre 1940 y 1976, profundiza sobre las leyes de la robótica y su complejidad moral y ética, incluyendo una cuarta ley (ley cero).

Los últimos años del siglo XX se caracterizaron por un progreso multidireccional en la robótica. Los años 80 supusieron una explosión en el desarrollo de la robótica y su década es considerada como el inicio de la 'Era Robótica', y es que su fabricación y venta aumentaron en un 80%. Se considera que empezó a sembrarse en esta época la semilla que luego haría surgir la robótica inteligente que hoy en día se conoce. Fue en esta década que Honda Motor Company comenzó a interesarse en este campo y desarrolló la Serie E Robots, modelos experimentales de robots de aspecto humanoide. Fueron siete modelos, desarrollados entre 1986 y 1993.

Asimismo, fue en esta época cuando surgieron los primeros exponentes notorios de la robótica móvil. Aunque ya se habían creado robots con capacidades móviles, sin base fija (como el primer AGV, el robot *Shakey* o el *Mars rover*), la llegada del CART de SRI International introdujo un modelo más asequible para la época. Consistía en una plataforma que modelaba obstáculos gracias a coordenadas cartesianas en sus vértices. Se caracterizaba por trabajar con un procesador de imagen estéreo más una cámara adicional. Este robot puede ser entendido como uno de los primeros que abordaron la disciplina de la 'planificación de movimiento'. El CART fue posteriormente sobrepasado por el CMU Rover en 1983.

Durante los años 90, la robótica en la industria continuó creciendo a pasos agigantados. La globalización, ocurrida en parte por la internacionalización de muchas empresas, contribuyó a que se demanden y produzcan muchos más bienes de consumo que décadas anteriores. Además, el desarrollo de disciplinas auxiliares como la inteligencia artificial, los microprocesadores o la visión artificial multiplicaron las posibilidades de los robots de cara al nuevo siglo. De estos años, destacan las siguientes aportaciones:

- El robot Genghis, revelado por el MIT en 1989. Genghis era famoso por ser manufacturado rápida y baratamente debido a sus métodos de construcción. Integraba 4 microprocesadores, 22 sensores, y 12 servo motores.

- Honda P Series. En la década de los 90, Honda desarrolló cuatro robots de aspecto humanoide. Sucesores de la Serie E y predecesores de la serie ASIMO, estos robots eran capaces de caminar lentamente y cargar peso.
- En 1999, Sony introdujo el AIBO, un perro robótico capaz de interaccionar con humanos. Eran mascotas robóticas con fines de entretenimiento; sin embargo, fueron ampliamente adoptados por las universidades con fines educativos.
- Aparición de robots móviles educativos, como los Pioneer, y brazos robóticos industriales con seis grados de libertad, destacando fabricantes como ABB.

2.2. Robots del siglo XXI

La llegada del siglo XXI fue protagonizada en la robótica por grandes avances que revolucionaron los límites conocidos de los robots. Las dos décadas de este siglo han sido testigos de cómo la robótica se ha expandido hacia una gran variedad de campos de aplicación, más allá de su inicial concepción industrial. Destacan nuevas áreas como la domótica, la robótica autónoma, la ingeniería mecatrónica, la robótica educativa, robots de limpieza, asistenciales, colaborativos o prótesis robóticas.

A poco de comenzar el nuevo siglo, el 31 de octubre del año 2000, se dio a conocer el primer avance de la época. Este fue el robot ASIMO, presentado por la compañía japonesa Honda. Este robot de aspecto humanoide podía desplazarse e interactuar con las personas. Era capaz de caminar, correr, correr en reversa, saltar en una o dos piernas de manera continua, además de mantener el equilibrio en superficies irregulares. Asimismo, podía controlar cada uno de sus dedos de manera independiente y llevar a cabo tareas con una precisión. Por último, sus sensores visuales y auditivos le permitían reconocer simultáneamente los rostros y la voz de las personas, incluso si hablaban al mismo tiempo.

Sin embargo, el pasar de los años, aunque incorporando actualizaciones en sus modelos, no lograron hacer del robot el producto atractivo por el que fue concebido en primer lugar: un asistente en las tareas de personas con movilidad reducida. Además, su alto coste por unidad (la última versión de ASIMO en 2011 costaba dos millones de euros) no era asumible por la gran mayoría de familias, por lo que el proyecto ASIMO finalmente se terminó en 2018.

***Figura 9**. Robot de aspecto humanoide ASIMO.*

El siguiente avance robótico más destacable de la década fue el robot iCub, robot humanoide de código abierto diseñado por el Consorcio RobotCub de varias universidades europeas y construido por el Instituto italiano de Tecnología entre los años 2004 y 2010. El robot se enfocaba en el estudio e investigación de la cognición social y la inteligencia artificial. Entre sus capacidades destacan: gatear, expresiones faciales, tiro con arco, control de fuerza, evitar colisiones en entornos estáticos y no estáticos, etc.

Otro robot humanoide muy relevante de la época es el robot InMoov. Originalmente creado por el escultor francés Gaël Langevin en septiembre de 2011 con fines artísticos, fue en enero de 2012 que sus planos fueron publicados permitiendo la impresión 3D de sus piezas. Además de ello, su software es de código de abierto y basado en microcontroladores de Arduino (para algunas herramientas de control se recomienda otros softwares como MyRobotLab). Ello hace de InMoov un robot que puede ser construido en casa, con un precio que oscila entre los 800 y los 2000 euros, dependiendo de las funcionalidades previstas.

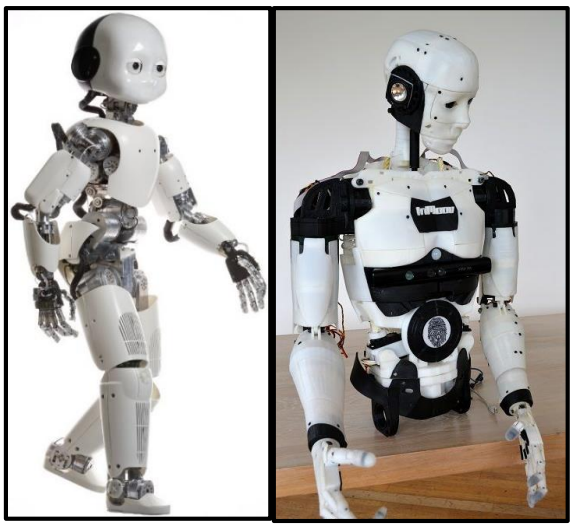

***Figura 10**. Robot humanoide iCub (izquierda) e InMoov (derecha).*

En los últimos años, ha surgido un robot que ha implementado las últimas técnicas de inteligencia artificial. Este es el robot humanoide Sophia (ginoide, simula a una mujer), creado por David Hanson en la compañía con sede en Hong Kong, Hanson Robotics, en 2015. Ha sido diseñada para aprender, adaptarse al comportamiento humano y trabajar con estos satisfactoriamente. Sophia destaca por su inteligencia artificial (IA), procesamiento de datos visuales, reconocimiento facial y tecnología de reconocimiento de voz de Alphabet Inc., (compañía matriz de Google), diseñada con capacidad de aprendizaje. En 2017, Sophia se convirtió en el primer robot con nacionalidad (saudí).

***Figura 11**. Robot humanoide Sophia (2017).*

2.3. Robótica móvil

Aunque el desarrollo histórico de la robótica en su conjunto ha influenciado la disciplina de la planificación de movimiento, es especial el caso de la robótica móvil, pues es la materia objeto en la que se enfoca el área de la planificación, disciplina que se aborda en el presente proyecto.

La robótica móvil es la rama de la robótica que incluye todos aquellos robots (modelización, diseño, construcción y aplicabilidad) que poseen capacidades móviles, es decir, son sistemas robóticos que pueden desplazarse en distintos entornos y cuentan con distintas capacidades que les permiten ejecutar tareas complejas, ya sea de forma autónoma o controlados por un operador humano. Estos sistemas cuentan con sensores y actuadores que los dotan de capacidades para conocer el entorno y modificarlo.

En esta definición los robots manipuladores industriales, aunque con capacidades móviles, no son capaces de desplazar su sistema de coordenadas debido a que poseen base fija, por lo que no se incluyen dentro de los robots móviles.

Los robots móviles pueden clasificarse en función de diferentes criterios:

1. De acuerdo a su tipo de locomoción. Se distinguen:

   1.1. Robots terrestres. Son aquellos que se desplazan a través de la superficie terrestre, es decir, por terrenos sólidos. Para ello existen diversas configuraciones:

      1.1.1. Robots móviles terrestres con ruedas. Las ruedas desde hace siglos han representado el mecanismo de tracción y desplazamiento por excelencia, desde que se comenzaron a utilizar en la antigüedad para permitir el desplazamiento de elementos como las carretas, y más reciente de los vehículos que utilizamos en nuestra vida diaria. Este sistema de locomoción permite al robot desplazarse por distintas superficies, es simple de controlar e implementar, y permite alcanzar distintas velocidades, pero también presenta desventajas cuando se enfrenta a terrenos muy irregulares o de mucha complejidad.

      1.1.2. Robots móviles terrestres con orugas. Las orugas permiten a los robots desplazarse en terrenos blandos y les da una capacidad de tracción mayor a las ruedas debido a sus múltiples puntos de contacto, pero de igual forma existen algunas desventajas, en cuanto a su tracción, velocidad, control.

1.1.3. Robots móviles terrestres con extremidades. La robótica se ha inspirado comúnmente en los comportamientos y habilidades de los seres vivos. La locomoción a través de extremidades es común en la naturaleza, los seres humanos, los insectos, los animales, se desplazan a través de configuraciones de extremidades. En robótica móvil este tipo de configuración permite realizar maniobras que las ruedas o las orugas no permiten. Este tipo de mecanismo presenta una alta complejidad en cuanto al diseño y control de los robots.

***Figura 12**. Vehículo terrestre autónomo en el DARPA Grand Challenge.*

1.2. Robots aéreos. Son robots con la capacidad de volar o mantenerse suspendidos en el aire. Los drones son un ejemplo de este tipo de robots. Se trata de aeronaves no tripuladas que han alcanzado gran popularidad entre el público general en los últimos años, lo que ha llevado a que emerjan cada vez más empresas que aprovechan este nicho de mercado.

No obstante, los drones históricamente surgen como herramientas militares de reconocimiento u ofensiva. Su primera aparición histórica, como un predecesor de cómo son entendidos en la actualidad, data de 1849, como globos incendiarios que las fuerzas austriacas lanzaron sobre Venencia. Posteriormente, durante todo el siglo XX, las aeronaves no tripuladas incrementaron su protagonismo en las guerras mundiales.

Durante la Primera Guerra Mundial, comenzó el primer intento serio de desarrollar el UAV. Para finales de la Primera Guerra Mundial y durante la Segunda Guerra Mundial, las aeronaves no tripuladas servían para el entrenamiento de los operarios de cañones antiaéreos.

Fue en el período de post-guerra cuando comenzó el desarrollo de estas aeronaves hasta como son conocidas actualmente. Mientras que hasta entonces se habían concebido como vehículos operados remotamente, a finales de los 90 y principios de nuevo siglo se logró que las naves tuvieran vuelo autónomo. Además, incorporaron nuevas funcionalidades tácticas, de reconocimiento y ofensivas.

***Figura 13**. UAV americano modelo MQ-9 Reaper de General Atomics.*

1.3. Robots acuáticos. Explorar fuentes acuáticas es una tarea que puede ser peligrosa para los seres humanos, por lo que se ha estudiado robots con capacidades de movilidad acuáticas, para operaciones de exploración, recolección, rescate, y que pueden actuar a distintas profundidades.

1.3.1. Robots acuáticos flotantes. Son robots acuáticos que permanecen a nivel del mar sin sumergirse. En los últimos años han surgido importantes prototipos.

Entre los últimos avances, destaca recientemente el programa 'Roboat', un programa de investigación de barcos autónomos con sede en Amsterdam que se ha desarrollado entre los años 2016 y 2021. El Instituto de Ámsterdam para Soluciones Metropolitanas Avanzadas (Instituto AMS) junto al Instituto de Tecnología de Massachusetts (MIT) son los autores de estos barcos autónomos.

Los roboats pueden evitar obstáculos y desviarse si perciben algo en su camino. Tienen la capacidad de formar estructuras de gran envergadura como puentes peatonales e incluso las bases para un mercado de alimentos, al desmontarse y conectarse de forma totalmente autónoma. Además, los roboats pueden usarse como sensores móviles para recopilar datos sobre la infraestructura de la ciudad y la calidad del aire y el agua, entre otras cosas.

1.3.2. Robots acuáticos submarinos. Son aquellos robots que desempeñan sus funciones por debajo del nivel del mar, por lo que poseen capacidades de buceo y pueden modificar su flotabilidad. Son conocidos como AUV por sus siglas en inglés, que hacen alusión a su capacidad autónoma.

Desde la creación del primer AUV en el Laboratorio de Física Aplicada de la Universidad de Washington en 1957, sus capacidades y aplicaciones se han incrementado enormemente. Actualmente, los AUV son utilizados para rutas comerciales, investigación marina, operaciones militares, rescate en accidentes aéreos (que terminan en el mar) e incluso como pasatiempo o tráfico ilegal.

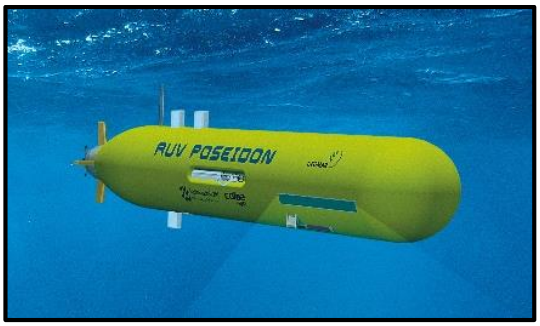

***Figura 14**. AUV modelo Poseidon, tecnología de torpedo nuclear ruso.*

2. En función de cómo se guían por el entorno que los rodea. Se distinguen:

2.1. Robots guiados. Son aquellos que siguen un camino predeterminado en su acción, como es el caso de los robots que siguen líneas sobre el suelo para desplazarse. Un claro ejemplo de este tipo de robots son los llamados AGV (Automatic Guided Vehicle). Los sistemas AGV tienen su comienzo en 1954 cuando Barrett Electronics Corporation dio a conocer el primer vehículo eléctrico que no necesitaba conductor humano. En un principio, un cable enterrado en el suelo generaba un campo magnético que era seguido por el vehículo. A día de hoy este mecanismo ha evolucionado a guías, cables, bandas magnéticas y sensores (escáneres láser, sensores de proximidad, sensores de ultrasonidos…) que le permiten desplazarse por su circuito. Además, se les puede incorporar sistemas de comunicaciones y diagnóstico para analizar el comportamiento en fábrica.

La desventaja de este tipo de robots móviles es que solo son capaces de moverse a través de su circuito cerrado. Pueden poseer ciertas capacidades de detección de obstáculo, para detenerse si es preciso, pero no pueden calcular nuevas rutas que eviten el obstáculo y a la vez le permitan continuar su trayectoria.

Cabe mencionar que los AGV no son el único sistema automático de transporte de mercancías en industria. Como sistemas guiados, también destacan sistemas como los transportadores (sistemas basados en cintas de transporte y elevadores que permiten trasladar la carga) o las electrovías (sistemas compuestos de carros automáticos, a nivel de suelo o aéreos, que transportan la carga a través de raíles electrificados).

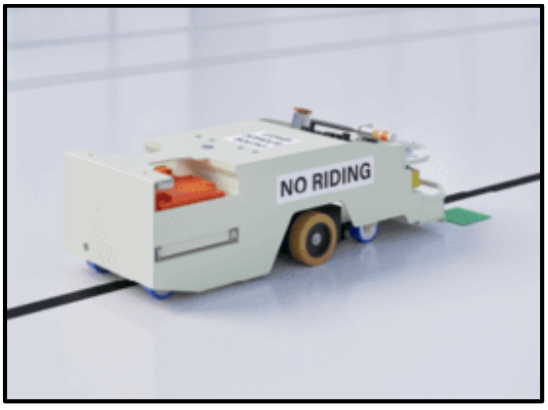

***Figura 15**. AGV industrial guiado por un circuito en el suelo.*

2.2. Robots no guiados. Son los que no siguen ningún patrón predeterminado de movimiento y se desplazan de acuerdo a la información que captan del entorno y las características del mismo. Un ejemplo de esta clase de robots móviles son los AMR (Autonomous Mobile Robot). La introducción de los AMR en la industria es un cambio relativamente reciente, ya que los AGV habían sido los protagonistas en las últimas décadas para tareas de logística, transporte y producción. No obstante, los AMR incorporan una flexibilidad nueva hasta la fecha, lo que ha permitido su rentabilidad para ciertas aplicaciones.

Respecto a su funcionamiento, los AMR se desplazan gracias a un software que les proporciona mapas en tiempo real y no necesitan un circuito por el que guiarse. La flexibilidad ya mencionada y la capacidad para evitar obstáculos al desplazarse también les distingue. Los AMR, ante la detección de un obstáculo en el entorno, tratará de encontrar un nuevo camino alternativo y más eficiente. Para ello suele necesitar tener precargados los planos del entorno, así como un sistema software de planificación de movimiento.

Aunque parece que las funcionalidades de los AMR rebasan las de los tradicionales AGV, no se puede concluir que los AMR son siempre mejor opción que los AGV. La elección de uno u otro dependerá del campo y ámbito de aplicación industrial. Entre los criterios para escoger uno u otro, se destaca la variabilidad en el proceso. Si se trata de una producción en serie en la que los procesos siempre son los mismos y los productos que se transportan tienen siempre un método fijo sin obstáculos en las rutas, probablemente los AGV darán mejores resultados que los AMR. Sin embargo, si se trata de un proceso cambiante con trayectos largos que incluyan muchos obstáculos y constante movilidad, los AMR son mejor opción. Otros criterios a valorar será el coste de producto (los AMR suelen poseer un coste unitario más alto que los AGV), el servicio técnico, el entorno de trabajo, el acondicionamiento inicial (el AGV requiere preparar el circuito y el AMR un diseño sobre el proceso en cuestión), el presupuesto de la empresa, etc.

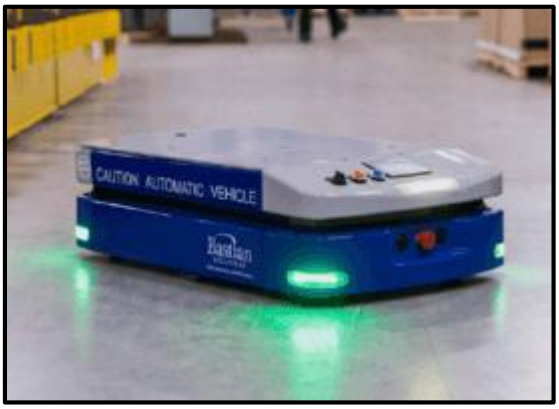

***Figura 16**. AMR industrial sensorizado.*

2.3. Planificación de movimiento

La planificación de movimiento es la disciplina que surge a raíz de la robótica móvil que estudia y busca solución al cálculo de la ruta óptima que debe seguir un robot móvil desde un punto inicial hasta un punto final a través de un entorno, en el que se presentan obstáculos con los que se debe evitar la colisión.

Para desempeñar sus funciones en las rutas previstas, los robots basan su funcionamiento en dos pilares de la robótica móvil: sensores y actuadores. Mediante un organismo de control que gestiona estos dos elementos y que integra el software para 'planificar el movimiento' se logra este cometido.

Este campo de investigación y desarrollo se encuentra muy ligado a los robots móviles terrestres, lo que reduce las dimensiones del problema al plano de la superficie por la que el robot puede desplazarse. No obstante, cabe recalcar que existen diversos métodos que extienden su aplicación al espacio de tres dimensiones. Tanto para el problema de dos como de tres dimensiones, existen diversas técnicas y perspectivas de trabajo que suelen abordarse en función del caso concreto y su aplicabilidad.

2.3.1. Condiciones iniciales

Un criterio muy importante para el diseño del software del robot es la información inicial de la que se parte. Entre otros aspectos, será necesario conocer:

- El dimensionamiento del robot y sus capacidades cinemáticas.

  A la hora de calcular sobre el entorno la ruta que debe seguir el robot es necesario tomar en consideración las limitaciones que el propio robot presenta. En función de sus dimensiones, puede haber caminos por los que el robot pueda o no circular.

  Por otro lado, los algoritmos propios de la planificación de movimientos no se encuentran adaptados con independencia de la naturaleza del robot, pues se debe tener en cuenta todas las limitaciones cinemáticas que poseen los robots, ya que cada cual posee distinta velocidad, capacidad de giro, capacidad de frenado… Debido a esta razón, es posible que un algoritmo encuentre una ruta óptima que mediante cálculo es trazable, pero puede que posea giros bruscos que no son capaces de realizar por el robot.

- Conocimiento sobre el entorno.

  Entendiendo el entorno como todo aquello que rodea y alberga al robot en su aplicación, se trata de un espacio que puede presentar incertidumbre.

  En primer lugar, se debe conocer cómo interactúa e identifica el robot su entorno y si esta información es completa o incompleta. En la mayoría de ocasiones, los robots móviles operan con los planos precargados del área de trabajo en la que se encuentran; no obstante, ello no garantiza un conocimiento total del entorno. Es por ello que integran sensores auxiliares para completar el conocimiento que posee de sus alrededores. Mediante estos sensores (basados en ultrasonidos, infrarrojos, etc.) se amplía la información de los obstáculos no anticipados, pero estos no pueden ser completamente modelados puesto que los sensores tienen la limitación de estar incorporados en el propio robot móvil.

En segundo lugar, los entornos pueden ser no estáticos, cambiantes, lo que requiere actualizar la información de la que dispone. Los sensores ya mencionados son los encargados de actualizar el conocimiento del robot sobre el entorno, pero la naturaleza cambiante de algunos obstáculos como operarios y vehículos requiere el diseño de algoritmos altamente flexibles.

En tercer lugar, algunos casos pueden presentar un problema añadido: la superficie del propio terreno. Enfocado a la planificación en dos dimensiones, el terreno por el que circula el robot puede ser irregular. Es decir, la presencia de desniveles, baches o la inclinación del terreno son limitaciones adicionales para el cálculo de la ruta. Por otro lado, en el problema de tres dimensiones, la dificultad puede incrementarse altamente, como es el caso de pisos comunicados por rampas o elevadores.

- La tarea encomendada al robot.

  En la industria la mayoría de robots móviles se encargan de procesos logísticos, es decir, permiten la automatización del transporte interno de mercancía. Por tanto y relacionado con el problema del dimensionamiento, se debe tener en cuenta las especificaciones de la carga: peso, volumen, fragilidad, etc. La etapa de diseño del software que planificará el movimiento debe tener en cuenta estos factores. Es por ello que los fabricantes estandarizan las capacidades de carga del robot e informan de las especificaciones máximas que no comprometen su funcionamiento.

2.3.2. Configuración del espacio de trabajo y algoritmos de cálculo de ruta

Una vez abordadas las limitaciones que pueden surgir, se procede a tratar las dos etapas básicas en las que se divide la planificación de movimientos:

- Configuración del espacio de trabajo.

  Antes de proceder a calcular sobre el entorno técnicas que permitan obtener la ruta óptima, se debe modelar el espacio de trabajo del robot móvil, es decir, determinar cómo es concebido el robot y sus configuraciones posibles en el área de trabajo. Existen diversos puntos de partida para modelar el robot:

  - Como un único punto en el plano, con la capacidad de desplazarse sobre dos dimensiones. Dos grados de libertad (x, y).
  - Como una figura de dos dimensiones en el plano, con la capacidad de trasladarse y rotar. Tres grados de libertad (x, y, θ).
  - Como un cuerpo modelado en el espacio de trabajo de tres dimensiones, con la capacidad de trasladarse y rotar. Seis (tres debidos a la traslación y tres a la orientación) grados de libertad $(x, y, z, \alpha, \beta, \gamma)$.
  - Otras posibilidades, como un brazo articulado conectado a una base móvil. Grados de libertad en función de las articulaciones.

  El presente proyecto ha buscado abordar las dos primeras opciones. Puesto que la primera opción es más simple que la segunda, existen pretratamientos del entorno que permiten su conversión. Se ha desarrollado la denominada ‘expansión del entorno’, técnica que se encuentra descrita en esta propia sección (pág. 24).

- Algoritmos de clasificación del entorno y de cálculo de ruta.

  Una vez queda definida la configuración del espacio de trabajo y el entorno de aplicación, se puede proceder a calcular la ruta que describa el robot desde el punto inicial hasta el punto final. Este proceso es calculado mediante algoritmos que plantean el problema desde distintas perspectivas, pero en todos ellos se clasifica el entorno (no ocupado por el propio robot) en tres posibilidades generales:

  · Espacio libre. Es aquel por el que el robot puede trasladarse. En función del grado de complejidad que se pretenda abordar, se trata de un espacio cambiante, que depende de la posición y orientación del robot en cada momento, ello considerando un entorno estático. Si se añade la posibilidad de un entorno no estático, el área libre se modificará constantemente y el algoritmo deberá recalcular con frecuencia la ruta a seguir.

  · Espacio objetivo. Es el espacio que se pretende alcanzar, aquel en el que el robot finaliza la ruta planificada.

  · Espacio ocupado por obstáculo. Denota aquellas regiones en las que el robot no puede situarse y, por lo tanto, la ruta calculada no puede atravesarlas.

  En función del itinerario que sigue el algoritmo, calculará simultáneamente o previamente la clasificación del entorno respecto a la ruta a tomar. Los algoritmos plantean las siguientes perspectivas:

  - Algoritmos de cálculo de rutas en grafos.

    Se trata de algoritmos que permiten obtener la trayectoria que debe describir un robot móvil desde un punto inicial hasta un punto final a partir de un espacio descrito como un grafo, en el que los puntos accesibles se denominan nodos, los cuales se conectan entre sí (arcos) formando la red.

    Se trata de algoritmos muy útiles, pero su uso exclusivo no suele ser suficiente para la planificación de movimientos. En esta línea, la principal desventaja de estos algoritmos es que es necesario la creación previa de la red de nodos para el cálculo de la ruta. La obtención de este grafo puede seguir enfoques muy diferentes, como se abordará en el resto de perspectivas de algoritmos.

    Dentro de esta clasificación de algoritmos, cabe destacar:

    - Algoritmo de Dijkstra. El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo exhaustivo (su coste computacional se incrementa exponencialmente conforme crezca la red) que consiste en la determinación del camino mínimo en coste entre un nodo inicial y final, a partir de un grafo cuyas conexiones entre nodos, denominadas arcos, llevan asociadas un coste (que usualmente es entendido en términos de distancia).
      Inicialmente, se construyen dos elementos:

      · Una lista con todos los nodos del entorno.

      · Una matriz de arcos. Estos representan el coste (distancia, energía, tiempo, etc.) entre todos los nodos. Si dos nodos no están conectados, este coste es infinito. Respecto al coste, un arco puede ser unidireccional o bidireccional, es decir, puede ser posible o no que el arco sea recorrido en sentido directo e inverso. En caso de no ser posible, el coste asociado al sentido no posible también es considerado infinito.

Los pasos que sigue el algoritmo de Dijkstra para determinar el camino de menor coste se pueden secuenciar en:

1. Comenzando desde el nodo inicial de la ruta, el algoritmo de Dijkstra analiza el grafo para encontrar el camino de menor coste entre el nodo de origen y el resto de nodos accesibles (cuyo coste no sea infinito).
2. El algoritmo repite esta búsqueda a partir de los nodos accesibles de los siguientes nodos, actualizando la ruta de menor coste calculada hasta cada nodo de la red que ha sido explorado hasta el momento.
3. Una vez el algoritmo encuentra el camino de menor coste entre el nodo de inicio y cualquier otro nodo, este último es marcado como "visitado" y es añadido a la matriz de rutas.
4. El proceso continúa hasta que se encuentra la ruta de menor coste hasta todos los nodos del grafo, es decir, hasta que todos los nodos son visitados. De esta forma, se constituye una matriz de rutas que aloja los caminos (mínimos en coste) que permiten unir el nodo inicial con cualquier nodo, incluyendo el nodo final.

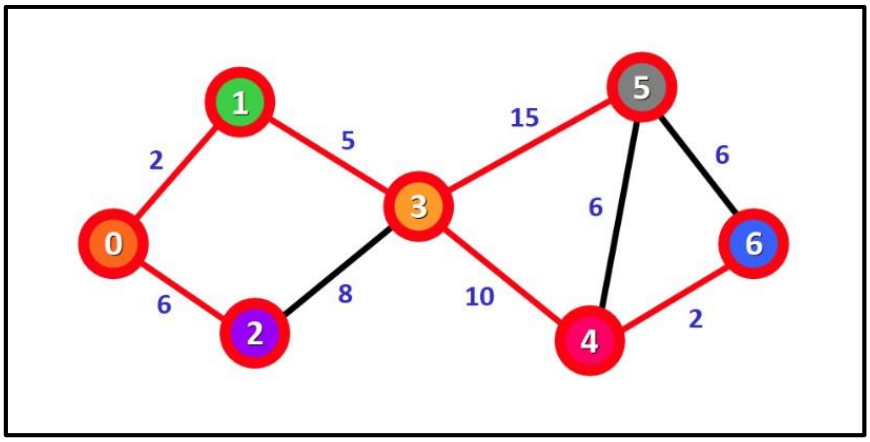

***Figura 17**. Implementación del algoritmo de Dijkstra sobre un grafo.*

El algoritmo de Dijkstra es un algoritmo completo (explora toda la red) y, si existe, encuentra la solución al cálculo de la ruta.

- Algoritmo A*. Pronunciado como algoritmo A asterisco, A estrella o A star (en inglés), se trata de un algoritmo de exploración de grafos heurístico que permite encontrar el camino de mínimo en coste que une un punto de origen hasta un punto de destino, a partir de una red constituida por nodos cuyas conexiones poseen costes asociados.

  Se ha mencionado el término heurístico y es lo que diferencia a este algoritmo del ya descrito algoritmo de Dijkstra. Para la exploración de los nodos en la búsqueda del camino de menor coste, se debe tener en cuenta la distancia heurística entre el nodo de estudio y el nodo objetivo. De esta forma, la función de coste de cualquier nodo n al destino ($f(n)$) es la suma del coste real desde el origen al nodo n ($g(n)$), y del coste heurístico del nodo n al destino ($h(n)$):

$$f(n) = g(n) + h(n)$$

Al igual que el algoritmo de Dijkstra, se requiere el conocimiento completo del grafo. Ello equivale a dos elementos de partida:

· Una lista con todos los nodos del grafo.

· Una matriz de arcos. Estos representan el coste (distancia, energía, tiempo, etc.) entre todos los nodos. Si dos nodos no están conectados, este coste es infinito. Respecto al coste, un arco puede ser unidireccional o bidireccional, es decir, puede ser posible o no que el arco sea recorrido en sentido directo e inverso. En caso de no ser posible, el coste asociado al sentido no posible también es considerado infinito.

Adicionalmente, mediante cálculo o como argumento de entrada, debe poder hallarse el coste heurístico de cada nodo al nodo objetivo.

Una vez se dispone de la información del grafo, el proceso que secuenciará el algoritmo A* será el siguiente:

1. Desde el nodo inicial de la ruta, se abre una lista con los nodos adyacentes, comprobando que no se conecte ya con el nodo objetivo, y asociando a cada nodo su coste (coste real más coste heurístico).
2. Sobre la lista de nodos actualmente explorados, se escoge el nodo de menor coste (coste real más coste heurístico) y se explora al igual que en el primer paso, es decir, se añade a la lista sus nodos accesibles y se asocian sus respectivos costes.
3. Este proceso, de búsqueda en la lista de nodos del coste mínimo e incorporación (conforme se explora el grafo) de nuevos nodos, continúa hasta que se logra conectar con el nodo de destino.

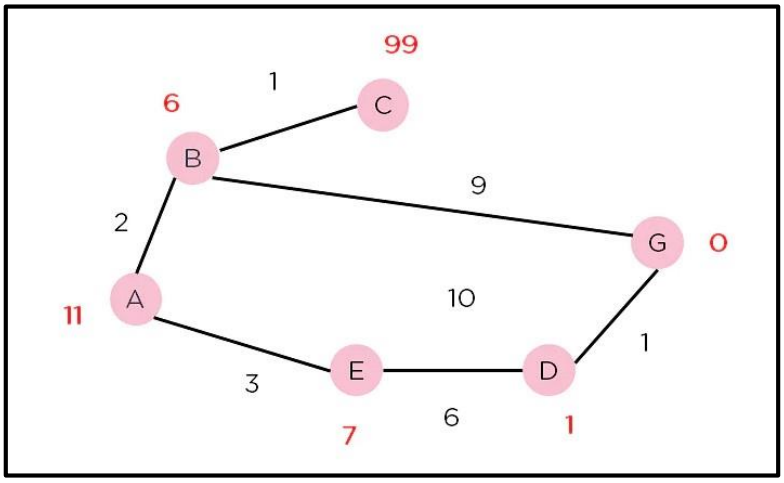

***Figura 18**. Grafo sobre el que implementar el algoritmo A*.*
El nodo de origen es el identificado por la letra A, mientras que el nodo objetivo es la G. Los números en negro informan del coste en el arco que permite la conexión entre dos nodos. Los números en rojo representan el coste heurístico desde el nodo en cuestión hasta el nodo objetivo (G).

El algoritmo de A* también es un algoritmo completo (explora toda la red) y, si existe, encuentra la solución al cálculo de la ruta.

- Algoritmo D*. El algoritmo D* (D estrella o D star) surge como una herramienta más avanzada que A*, utilizada para el cálculo de rutas sobre entornos no conocidos y cuya información se capta periódicamente a partir de los sensores del robot.

  Existen tres versiones de este algoritmo, la versión 'Original' creada por Anthony Stentz en 1994, la versión 'Enfocado' desarrollada también por Anthony Stentz como un avance en su modelo original, y la versión 'Lite' creada por Sven Koenig y Maxim Likhachev. Hoy en día, esta última es la más utilizada.

  El algoritmo, de funcionamiento similar al algoritmo de Dijkstra y al A*, resuelve problemas de planificación de ruta donde un robot tiene que navegar hasta las coordenadas de un objetivo, dado en un terreno desconocido y mediante la realización de suposiciones (por ejemplo: ningún obstáculo). Cuando se capta información nueva del entorno, el robot recalcula la ruta desde su posición actual hasta el objetivo y, de ser necesario, modificará su trayectoria anterior.

  Al atravesar terrenos desconocidos, es frecuente obtener información de nuevos obstáculos no anticipados, por lo que una nueva planificación tiene que ser rápida. Este algoritmo de búsqueda heurística incremental acelera sus posteriores búsquedas mediante el uso de la experiencia con planificaciones anteriores, lo que lo vuelve más eficiente que el original A*.

- Algoritmos de búsqueda basados en cuadrícula (grid).

  Este tipo de algoritmos basa su enfoque en la superposición de una cuadrícula (grid) sobre el entorno. Los puntos de la cuadrícula designan aquellos nodos que constituirán la red por la que puede circular el robot.

  Se debe tener en cuenta que los puntos de la cuadrícula pueden situarse en espacio libre o espacio ocupado, por lo que son necesarios algoritmos auxiliares de detección de áreas de obstáculo. Asimismo, en cada punto de la cuadrícula el robot únicamente será capaz de trasladarse a los puntos adyacentes y, en algoritmos más complejos, los nodos adyacentes quedarán limitados por las capacidades de cambio de rumbo del robot móvil, dándose la posibilidad de restringir algunos nodos aun siendo adyacentes.

  Otro aspecto importante de estos algoritmos es la resolución de la cuadrícula, es decir, la separación entre puntos. Las resoluciones más altas traerán consigo rutas más óptimas, pero el coste computacional también se incrementará.

  Algunos algoritmos oportunos para generar trayectorias sobre la cuadrícula son los basados en redes de nodos, como el algoritmo de Dijkstra o el algoritmo A*.

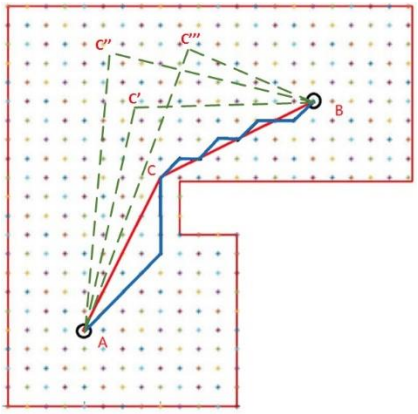

***Figura 19**. Simulación de un algoritmo de planificación de movimiento basado en cuadrícula.*
Se busca la ruta más corta que conecta el punto A con el punto B, siendo accesibles los nodos que marcan la cuadrícula.

- Algoritmos geométricos.

  Son algoritmos que basan su enfoque en el aprovechamiento del modelado poligonal de los obstáculos para su detección y posterior cálculo de ruta del robot. Existen numerosas herramientas desarrolladas a partir de este enfoque, entre las que se pueden destacar:

  - Grafos de visibilidad. Partiendo de un entorno construido a partir de obstáculos poligonales, el grafo de visibilidad es aquel que se construye conectando (mediante segmentos) cada vértice de cada obstáculo con todos los vértices accesibles, es decir, aquellos que no crucen a través de la región ocupada de un obstáculo. A partir la comunicación entre vértices se puede crear una red de nodos para planificar el movimiento. Los algoritmos de Dijkstra o A* pueden ser útiles para estos procesos.

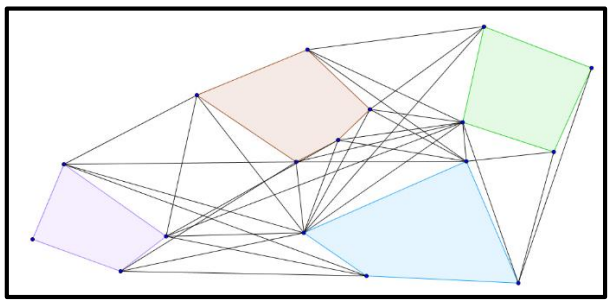

***Figura 20**. Grafo de visibilidad de un entorno compuesto por obstáculos poligonales.*

  - Descomposición en celdas. A partir de un entorno geométrico, los algoritmos de descomposición en celdas tratan de distinguir el entorno ocupado por obstáculos del entorno libre, a partir de una discretización del área del entorno. Dentro de esta perspectiva existen numerosos planteamientos de descomposición en celdas. Ejemplos de ellos son celdas verticales, celdas uniformes, quadtree u octotree (para casos de tres dimensiones). Una vez delimitado el entorno en celdas, se procede a conectar aquellas celdas adyacentes y accesibles en

búsqueda de la ruta óptima. Este proceso puede ser realizado mediante redes de grafos, campos potenciales u otras alternativas.

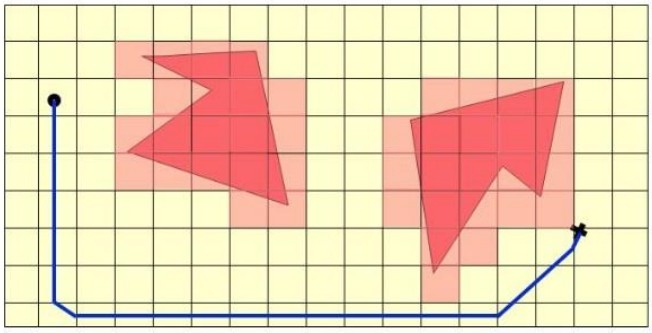

***Figura 21**. Simulación de un entorno descompuesto en celdas.*

- Expansión del entorno. Es un pretratamiento del entorno que consiste en tomar la tolerancia adecuada alrededor de los obstáculos de forma que permita considerar al robot como un punto en el espacio, que coincide con el sistema de referencia móvil. Este proceso puede ser de mayor o menor complejidad en función del grado de precisión y tolerancia que se pretenda alcanzar.

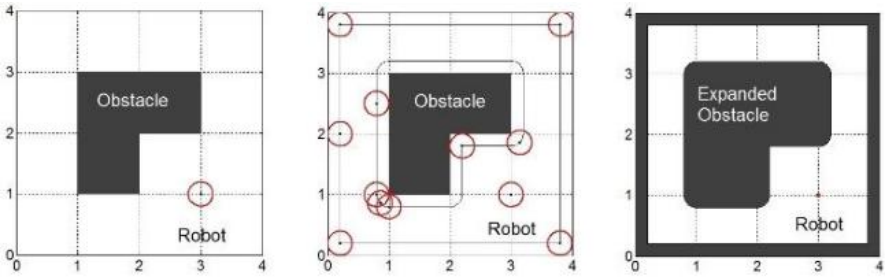

***Figura 22**. Simulación secuencial de una expansión del entorno.*
A la izquierda, entorno sin expandir, robot representado como una figura de dos dimensiones. En el centro, entorno expandiéndose, se comprueba para cada segmento y cada vértice de cada obstáculo que la tolerancia tomada es adecuada. A la derecha, entorno expandido, robot representado como un punto en el plano.

Como se ha descrito, la expansión del entorno puede tomar distinta complejidad en función de la precisión deseada. En el presente documento se ha implementado la expansión del entorno basada en la suma de Minkowski (sección '*5. Expansión del entorno mediante suma de Minkowski*', pág. 36, véase *Índice General*), que trata de tomar en cuenta la orientación del robot para delimitar en mayor o menor medida el espacio de configuración del robot.

- Trazado del rayo más lejano. Se trata de un método utilizado especialmente para el cálculo de rutas a salidas de emergencia en edificios. Consiste en trazar, desde la posición actual del robot, un haz de rayos alrededor del robot, en todas las direcciones. El robot deberá seguir la ruta cuyo rayo llegue más lejos hasta chocar con un obstáculo, a menos que se encuentre la puerta de salida. Se trata de un método iterativo que emite el haz cada cierta frecuencia, y termina cuando la ruta conecta con la salida.

- Algoritmos basados en campos potenciales.

  Se trata de un enfoque que considera el punto final como un foco de atracción y los obstáculos como focos de repulsión para la creación de un campo potencial a través del espacio libre. Una vez creado el campo, la ruta óptima se determina como aquella de menor coste, entendido como el potencial del campo.

  Esta perspectiva tiene la ventaja de que la trayectoria se produce con pocos cálculos. Sin embargo, el robot móvil puede quedar atrapado en mínimos locales del campo potencial y no encontrar un camino, o puede encontrar un camino no óptimo.

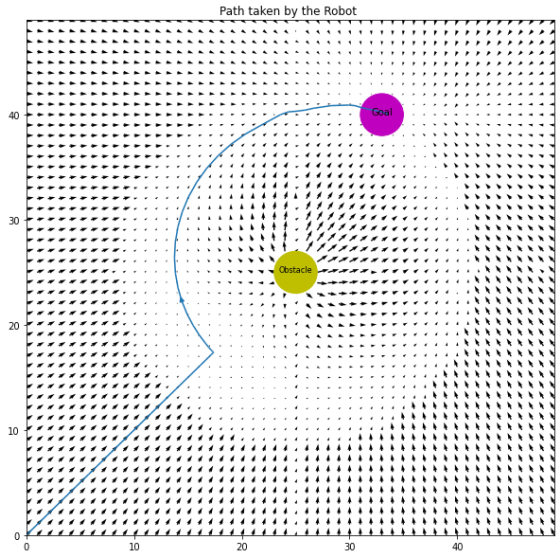

***Figura 23**. Simulación de un algoritmo de campo potencial.*

- Algoritmos basados en el muestreo.

  Los algoritmos basados en muestreo presentan un enfoque similar a los algoritmos de búsqueda en cuadrícula. Sin embargo, en vez de superponer todos los nodos posibles de la cuadrícula sobre el entorno, se representa con cada muestreo el espacio de configuración adyacente que considera que se acerca al punto objetivo (además de comprobar mediante algoritmos auxiliares que el espacio es libre mediante detección de colisiones con obstáculos), creando un itinerario tal como una hoja de ruta de configuraciones muestreadas.

  El algoritmo retiene las configuraciones para usarlas como hitos, construyendo una hoja de ruta que conecta los puntos. Si una ruta en la hoja de ruta vincula el punto inicial y el punto final, el planificador tiene éxito y devuelve esa ruta. Si no es así, la razón no es definitiva: o no hay ruta en el espacio libre, o el planificador no muestreó suficientes hitos.

  Estos algoritmos funcionan adecuadamente para espacios de configuración de alta dimensión porque su tiempo de ejecución no depende exponencialmente de la dimensión del entorno. También son generalmente más fáciles de implementar. Son probabilísticamente completos, lo que significa que la probabilidad de que produzcan una solución se aproxima a 1 a medida que se emplea más tiempo. Sin embargo, no pueden determinar si no existe una solución.

Entre los algoritmos que siguen esta filosofía, sobresalen dos:

- Algoritmo RRT (rapidly exploring random tree). El algoritmo RRT consiste en la construcción de un árbol que explora aleatoriamente el área libre de un entorno, hasta encontrar el punto objetivo de la ruta. Como un algoritmo basado en el muestreo, la ramificación del árbol se produce por iteraciones.

  El algoritmo comienza desde la configuración inicial desde la que parte el robot móvil y expande una rama en una dirección aleatoria, cuya longitud suele estar ponderada a partir de algún valor o procedimiento. Si la ramificación es realizable, es decir, pertenece al área libre y no atraviesa obstáculos (además de otros criterios concretos del caso en cuestión, como que sea posible tomando en consideración la orientación del robot), la ramificación se incorpora al árbol.

  En el siguiente muestreo, se produce otra ramificación aleatoria, desde cualquier punto del árbol construido hasta el momento, ponderando la longitud y comprobando su realizabilidad. Este proceso continúa hasta que se logra encontrar el punto de destino (o una región que se admite tolerantemente como el punto de destino, ya que se trata de un método de exploración del entorno que no discretiza en nodos).

  Adicionalmente, aunque se trata de un método aleatorio, la exploración puede incorporar sesgos para direccionar el crecimiento del árbol, buscando una mayor eficiencia del algoritmo. La mayoría de algoritmos suelen incluir este sesgo, cuya implementación puede ser, por ejemplo, aumentando ligeramente la probabilidad de crecimientos cuya dirección coincida con el punto de destino.

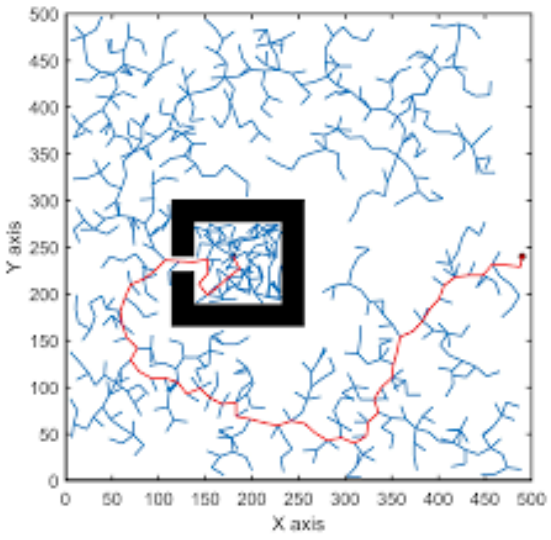

***Figura 24**. Simulación de un algoritmo RRT.*
Se observa el crecimiento de un árbol desde un punto inicial (contenido en el recinto que posee una sola salida) hasta un punto final (a la derecha en la gráfica).

- Hoja de ruta probabilística o algoritmo PRM (probabilistic roadmap). Se trata de un algoritmo basado en el muestreo que lleva a cabo la construcción, sobre el espacio libre, de una hoja de ruta que conecte los puntos inicial y final. El razonamiento en el que se basa este algoritmo para crear la hoja de ruta es el siguiente:

Se 'prueba' muestras aleatorias del espacio de configuración del robot en el entorno y se comprueba si estas pertenecen al área libre. Si esto es así, se utiliza un planificador local para conectar las configuraciones nuevas a otras configuraciones cercanas conocidas. Agregando además las posiciones de inicio y objetivo, el algoritmo es capaz de encontrar la solución a la planificación de movimiento cuando existe una hoja de ruta de configuraciones del robot que proporciona un camino desde el punto inicial hasta el final, atravesando las configuraciones cuyo coste (calculado hasta ese momento) sea mínimo.

Cada muestreo del planificador probabilístico PRM consta de dos fases:

· Una fase de construcción, en la que se conforma y actualiza la hoja de ruta. El paso inicial es crear una configuración aleatoria y realizable (que se encuentre en el área libre). Tras ello, se conecta a algunos vecinos, normalmente los k vecinos más cercanos o todos los vecinos a menos de una distancia predeterminada. Las configuraciones y conexiones se agregan al gráfico hasta que la hoja de ruta sea lo suficientemente densa.

· Una fase de consulta, las configuraciones de inicio y meta están conectadas al gráfico. Mediante algoritmos de exploración de grafos como Dijkstra o A* se calcula si la ruta es realizable.

Como desventaja, se trata de un algoritmo que posee problemas de eficiencia cuando los caminos posibles son estrechos, pues al tratar de probar configuraciones aleatorias, las regiones pequeñas son improbables de muestrearse, lo que deriva en un alto coste en términos de tiempo y cómputo en la exploración del entorno.

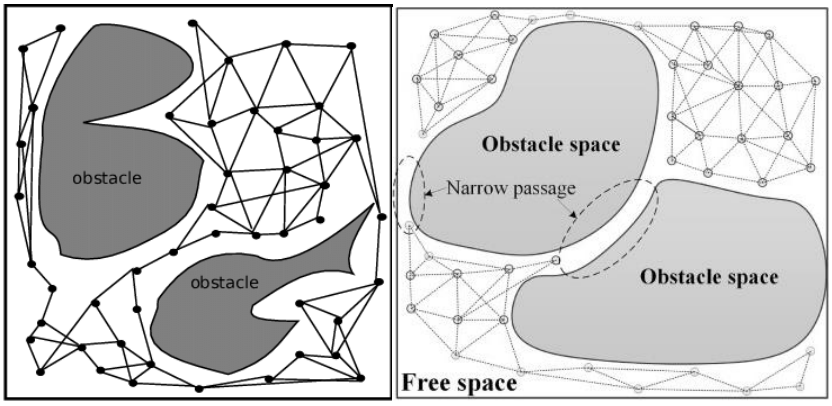

***Figura 25**. Simulación de dos algoritmos PRM.*
A la izquierda, se presenta un entorno cuya exploración por PRM ha sido exitosa. A la derecha, el entorno explorado por PRM presenta problemas de eficiencia en la expansión de la hoja de ruta debido a la presencia de caminos estrechos necesarios para calcular la ruta.

3. Objetivos

Como se adelantaba en las secciones anteriores, este proyecto de trabajo de fin de grado tiene su origen en un trabajo previo desarrollado para la disciplina de planificación de movimiento, concretamente, para la descomposición en celdas del entorno. El trabajo previo, desarrollado por los antiguos alumnos de Máster *JL. Guardiola y R. Sebastiá*, consistía en la descomposición y discretización en celdas uniformes del entorno, para posteriormente aplicar un campo potencial que permitiera el cálculo de la ruta que conectaba los puntos inicial y final.

Este trabajo nace, a su vez, de la Toolbox de Matlab perteneciente al profesor *Ranko Zotovic Stanisic*, un paquete de archivos de Matlab relativos a la planificación de movimiento, herramienta auxiliar para el Máster que imparte y que se encuentra en evolución. El objetivo principal del proyecto ha sido ser partícipe de la evolución de esta toolbox, ampliándola en la senda abierta por los antiguos estudiantes, bajo el contexto de la descomposición en celdas del entorno.

El trabajo desarrollado consta de seis algoritmos principales desarrollados, así como la revisión y documentación del caso de la descomposición en celdas uniformes. Estos seis algoritmos son:

- Expansión del entorno mediante la suma de Minkowski. Se trata de un algoritmo auxiliar y previo a la descomposición en celdas. Antes de proceder a planificar el movimiento, resulta necesario un pretratamiento del entorno, de forma que el espacio de configuración del robot móvil quede simplificado al problema de un punto que posee dos grados de libertad.
- Descomposición del entorno mediante celdas verticales. Es un algoritmo de descomposición que abarca diversos métodos de funcionamiento. Se ha desarrollado dos de ellos:
  - Descomposición mediante bandas verticales. El entorno se descompone en bandas verticales uniformes en su anchura, que delimitan el espacio en espacio libre y espacio ocupado por obstáculos.
  - Descomposición mediante configuración trapezoidal. Las celdas verticales que dividen el entorno poseen sus límites en las proyecciones verticales de los vértices de los obstáculos. Las celdas resultantes no son uniformes (poseen forma trapezoidal) y la resolución no puede ser arbitrariamente introducida por el usuario; no obstante, ofrece otras ventajas en términos de conectividad de nodos.

  En el presente documento de memoria se lleva a cabo un análisis comparativo de ambas técnicas (véase *Índice General*).
- Descomposición en celdas uniformes. Se trata de un algoritmo basado en fuerza bruta que descompone todo el entorno en celdas de tamaño uniforme, para posteriormente analizarlas una por una. Por un lado, se ha revisado la implementación del trabajo previo y se ha aportado documentación para el mismo. Y, por otro lado, se ha desarrollado una versión propia de este algoritmo, ofreciendo una nueva perspectiva de diseño y ofreciendo al usuario ciertos ajustes arbitrarios adicionales. Además, se ofrece una valoración comparativa de ambos diseños (véase *Índice General*).
- Descomposición del entorno en celdas mediante Quadtree. Este algoritmo procede a la descomposición en celdas desde un enfoque de eficiencia y capacidad resolutiva. No

obstante, posee ciertos inconvenientes relativos al coste exponencial ante el aumento de nivel de resolución o el cálculo posterior de rutas.

- Diagrama de Voronoi. Se trata de un recurso auxiliar en el estudio y análisis de los entornos. No constituye un método de planificación de movimientos en sí mismo, pero permite conocer las rutas más seguras posibles para el robot.

Mediante el desarrollo de estos algoritmos, se pretende enriquecer la toolbox de Matlab sobre planificación de movimiento, así como realizar un análisis comparativo posterior de los resultados obtenidos mediante el uso de cada algoritmo y combinándolos, e inferir en las conclusiones oportunas.

4. Estándares de la Toolbox de Matlab

Como ya ha sido mencionado a lo largo de este documento, la toolbox de Matlab es el 'paquete' que aloja los archivos, scripts y funciones que sirven como herramientas académicas para la disciplina de la planificación de movimiento, para el Máster impartido por el profesor *Ranko Zotovic Stanisic*.

Respecto a la toolbox, se caracteriza por una serie de estándares mantenidos en todas sus funciones, cuya razón de ser es permitir la compatibilidad de las funciones entre sí, de modo que una vez acostumbrado a los estándares y la filosofía de trabajo de la toolbox, su uso se vuelva intuitivo. Ello es especialmente útil recordando el hecho de que se trata de un contenido académico para ser impartido en la universidad.

La planificación de movimiento característica del proyecto (y del resto de la toolbox) posee un espacio de configuración en dos dimensiones, es decir, está enfocado a robots móviles terrestres, capaces de desplazarse por la superficie. Partiendo de esta premisa, a continuación, se enlistan los estándares sobre los que está construida la toolbox de Matlab:

- No se plantea la existencia de irregularidades en el terreno (desniveles, inclinación, etc.) que modifique la carga de trabajo del robot en el recorrido de un sentido u otro de alguna región del entorno. Asimismo, el coste en los algoritmos de cálculo de rutas es entendido únicamente como la distancia.

- Aunque se trata de un espacio euclídeo de dos dimensiones, los puntos del entorno se representan como un vector de tres dimensiones cartesianas, en el que el valor de la altura será nulo.

  Un punto P, respecto al sistema de coordenadas global (world), se representa como:

  $$P = [P_x \quad P_y \quad P_z]$$

- Los segmentos se codifican mediante un vector de seis columnas que aloja las coordenadas del punto inicial en sus tres primeras columnas y el punto final en sus tres últimas.

  Un segmento S cuyos puntos inicial y final son P_0 y P_1, respectivamente, se representa como:

  $$S = [P_{o\_x} \quad P_{0\_y} \quad P_{z\_0} \quad P_{1\_x} \quad P_{1\_y} \quad P_{z\_1}]$$

- El entorno es un espacio codificado mediante una matriz de segmentos que, a diferencia de los segmentos previos, poseerán ocho columnas. Las seis primeras columnas contienen la misma información que los segmentos ya descritos.

  La séptima columna representa el ángulo, en radianes, que el segmento describe desde su punto inicial hasta su punto final. Se puede escoger diferentes formas de codificar este ángulo. En la mayoría de scripts de la toolbox este ángulo equivale al ángulo encerrado en el obstáculo, entre el segmento de estudio actual y el previo. No obstante, a lo largo de este proyecto, el ángulo asoicado a cada segmento se construirá tomando como referencia el eje vertical negativo. En palabras más intuitivas, si se considera un reloj donde una aguja es el segmento en cuestión y la referencia es otra aguja marcando las 6. El ángulo que forman ambas agujas es el valor de la séptima columna. Se ha

desarrollado una función auxiliar que permite calcular estos ángulos (*reescribir_entorno.m*, véase *Anexos*).

La octava columna es un índice que permite identificar el tipo de segmento, distinguiéndose dos tipos:

- Los límites del entorno, que se presupone rectangular. El índice de los segmentos que codifican los límites del entorno debe ser 0.
- Los obstáculos del entorno. Cada obstáculo poseerá un índice distinto, de modo que todos los segmentos pertenecientes al mismo obstáculo poseerán el mismo valor de índice en su octava columna. Se comienza desde el valor 1 y se incrementa también en 1 recorriendo los números naturales.

Un ejemplo de declaración de un entorno (Ent) definido por sus límites y dos obstáculos triangulares sería:

$$Ent = \begin{bmatrix} 0 & 0 & 0 & 0 & 20 & 0 & \pi & 0 \\ 0 & 20 & 0 & 20 & 20 & 0 & \pi/2 & 0 \\ 20 & 20 & 0 & 20 & 0 & 0 & 0 & 0 \\ 20 & 0 & 0 & 0 & 0 & 0 & -\pi/2 & 0 \\ 1.5 & 3 & 0 & 7 & 7 & 0 & 2.1996 & 1 \\ 7 & 7 & 0 & 12 & 1 & 0 & 0.6947 & 1 \\ 12 & 1 & 0 & 1.5 & 3 & 0 & -1.7590 & 1 \\ 18 & 17.2 & 0 & 15 & 16 & 0 & -1.1903 & 2 \\ 15 & 16 & 0 & 19 & 9.5 & 0 & 0.5517 & 2 \\ 19 & 9.5 & 0 & 18 & 17.2 & 0 & -3.0124 & 2 \end{bmatrix}$$

Las primeras cuatro filas codifican los límites del entorno. Las siguientes tres el primer triángulo y, las últimas tres, el segundo triángulo. Nótese que el punto donde termina un segmento es el mismo en el que comienza el siguiente segmento, hasta cerrar el área volviendo al punto inicial del primer segmento.

Mediante el uso de la función *dibuent.m* (función ya existente en la toolbox) se puede observar el entorno descrito por Ent:

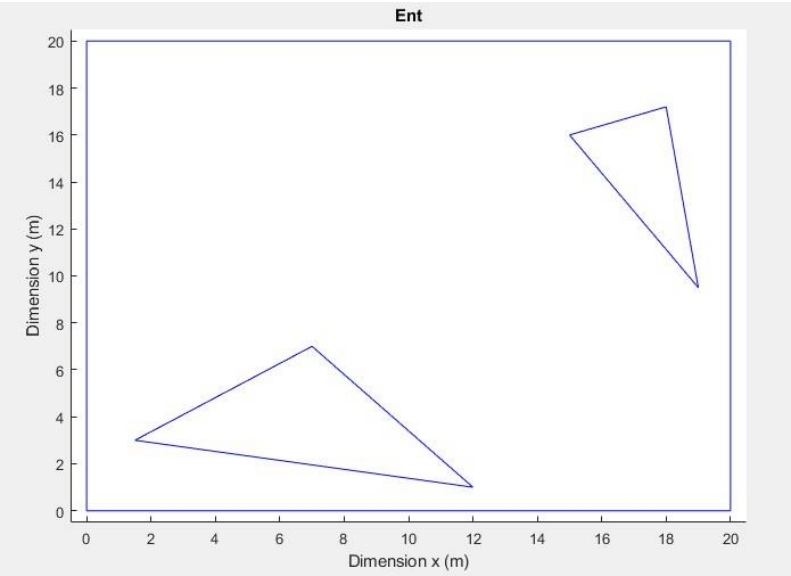

***Figura 26**. Representación del entorno Ent.*

- Los obstáculos poseen una representación polinomial. En otras palabras, se codifican internamente como una serie de segmentos unidos en un punto (llamado vértice) que delimitan un área cerrada. Se deben declarar en orden todos los segmentos que construyen un obstáculo. Algunos algoritmos requieren que su declaración siga un sentido estrictamente horario o antihorario. Se ha desarrollado una función auxiliar que permite cambiar su declaración de un sentido a otro (*reescribir_entorno.m*, véase *Anexos*).

- El robot móvil es modelado como un polígono, con aspecto rectangular, en el que un lado de distinto color identifica la cara frontal del robot. En la toolbox existente, se facilitan funciones que permiten su representación gráfica, traslación, rotación, etc.

  Sobre el entorno Ent que se observa en la *Figura 26* se puede representar el robot móvil. Como ejemplo, se modela un robot móvil con las siguientes características:

  - Longitud: $1.5\ m$.
  - Anchura: $1\ m$.
  - Orientación (respecto a la horizontal): $70°$.
  - Sistema de referencia móvil (considerando que el robot reposa sobre los ejes globales con orientación de 0 grados): $[0.5\ 0.5\ 0]\ m$.
  - Posición del sistema de referencia móvil respecto al sistema de referencia global: $[14\ 11\ 0]\ m$.

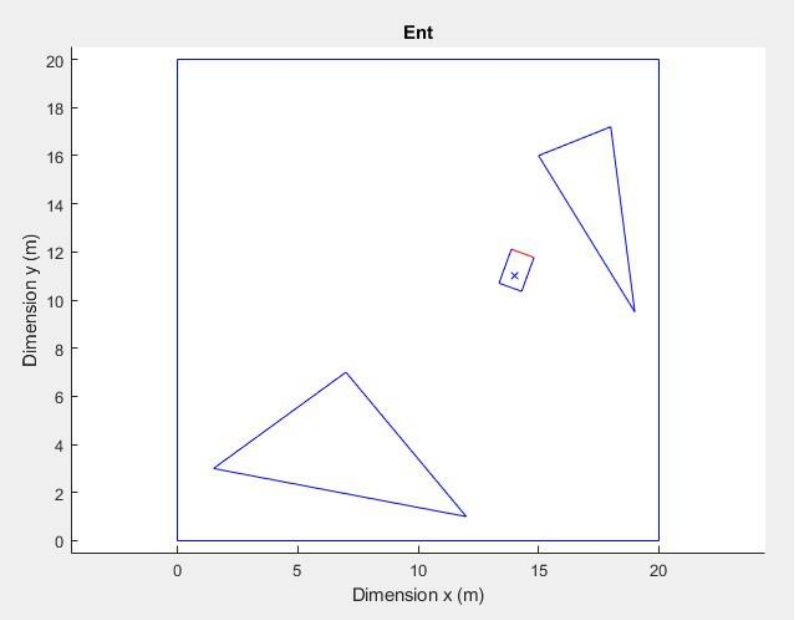

***Figura 27**. Representación del robot móvil en el entorno Ent.*

- Se plantea un estándar de nueva incorporación durante la realización de este Trabajo de Fin de Grado. En los algoritmos de descomposición en celdas, estas estarán codificadas por una matriz de $2x6$ que aloja los cuatro vértices de la celda. Siendo P_1 el vértice inferior izquierdo de la celda, P_2 el vértice superior izquierdo, P_3 el vértice superior derecho y P_4 el vértice inferior derecha, la matriz que codifica la celda quedaría tal que:

$$\begin{bmatrix} P_{1\_x} & P_{1\_y} & P_{1\_z} & P_{2\_x} & P_{2\_y} & P_{2\_z} \\ P_{3\_x} & P_{3\_y} & P_{3\_z} & P_{4\_x} & P_{4\_y} & P_{4\_z} \end{bmatrix}$$

No obstante, para el caso concreto del Quadtree, las celdas estarán codificadas por una matriz de $2x7$. La submatriz de $2x6$ poseerá la misma información que la que se acaba de describir. Por otro lado, el elemento $(1,7)$ poseerá el estado de la celda (libre, parcialmente ocupada o íntegramente ocupada), mientras que el elemento $(2,7)$ indicará al programa el nivel de ramificación de la celda en cuestión. Este aspecto y el resto del funcionamiento del Quadtree se encuentra detallado en la sección '*9. Descomposición del entorno en Quadtree*' (pág. 80, véase *Índice General*).

Además, estas matrices se integrarán en arreglos tipo 'cell' de Matlab, que permitirán contener la información de todas las celdas en una única variable.

Por último, se incluye una lista de los archivos de la toolbox. Por un lado, aquellos que han sido desarrollados y revisados durante el proyecto. Y, por otro lado, los archivos ya existentes en la toolbox, previos a la existencia de este proyecto, pero que son requeridos para el funcionamiento de los nuevos scripts.

Toda la información referida al funcionamiento de los algoritmos principales y auxiliares queda detallada en sus respectivas secciones (véase *Índice General*). La siguiente lista se expone como un sumario que no distingue qué funciones requiere cada algoritmo.

Los archivos principales desarrollados a lo largo de este proyecto de final de grado han sido:

- *expandir_entorno_minkowski.m*

  Las funciones auxiliares implementadas para este algoritmo:

  - *minkowski_paralela_preliminar.m*
  - *minkowski_puntos_vertice.m*

- *bandas_verticales.m*

  La función auxiliar implementada para este algoritmo:

  - *bandas_preparaciones_ruta.m*

- *celdas_verticales_configuracion_trapezoidal.m*

  La función auxiliar implementada para este algoritmo:

  - *trapezoidal_preparaciones_ruta.m*

- *celdas_uniformes.m* (implementación propia)

  Las funciones auxiliares implementadas para este algoritmo:

  - *ruta_celdas_uniformes_campo_potencial.m*
  - *celdas_uniformes_explorar_campo_potencial.m*

- *quadtree.m*

  Las funciones auxiliares implementadas para este algoritmo:

- *quadtree_preparaciones.m*
- *quadtree_preparaciones_ruta.m*
- *quadtree_localizar_punto.m*
- *quadtree_extraer_nodos.m*
- *quadtree_extraer_celda.m*
- *quadtree_celdas_contiguas.m*
- *punto_pertenece_segmento.m*

➢ *diagrama_voronoi.m*

La función auxiliar implementada para este algoritmo:

- *punto_pertenece_voronoi.m*

➢ Por último, se destacan aquellas funciones auxiliares desarrolladas que no son propias del funcionamiento de un solo algoritmo, sino que son utilizadas en más de una sección:

- *reescribir_entorno.m*
- *extrae_informacion_vertices.m*
- *celda_ocupada_parcial.m*
- *celda_ocupada_integra.m*
- *punto_dentro_obstaculo.m*

Los archivos propios del trabajo original sobre descomposición en celdas uniformes son los siguientes:

➢ *celdas1.m* (script principal, el resto son funciones)
➢ *demamm.m*
➢ *buscaborde.m*
➢ *etiquetado.m*
➢ *reetiquetado.m*
➢ *discretizapunto.m*
➢ *puntosvalidos.m*
➢ *generaruta.m*
➢ *dibujodelarutamascorta.m*

Los archivos previos a la realización de este proyecto, pero necesarios para su funcionamiento son:

➢ *despl.m*
➢ *dibuagv.m*
➢ *dibuent.m*

- *dibulin.m*
- *dibuobj.m*
- *extrae.m*
- *extrae_obstaculo.m*
- *inter_seg.m*
- *interseccion2d.m*
- *modlin.m*
- *paralela_segmento.m*
- *precision.m*
- *rotaz.m*
- *xamh.m*
- *Dijkstra.m*
- *dibujar_camino.m*
- *intersección_normal.m*

5. Expansión del entorno mediante suma de Minkowski

5.1. Introducción

La primera herramienta que se ha desarrollado para el proyecto es la expansión del entorno mediante suma de Minkowski. No constituye un método de cálculo de ruta en sí mismo. En vez de ello, es un pretratamiento del entorno que permite simplificar la concepción del robot móvil para el posterior algoritmo.

La expansión del entorno es un algoritmo basado en geometría que consiste en tomar la tolerancia adecuada alrededor de los obstáculos de forma que permita representar el espacio de configuración del robot como un único punto en el espacio, que coincide con el sistema de referencia móvil. Es decir, el algoritmo realiza una conversión del entorno (sus límites y los obstáculos que aloja) y el robot móvil pasa de una configuración de trabajo en el plano compuesta por tres grados de libertad (x, y, θ) a únicamente dos (x, y).

Este proceso puede ser de mayor o menor complejidad en función del grado de precisión y tolerancia que se pretenda alcanzar. En la toolbox de Matlab ya existían precedentes de estas funciones, que se enlistan continuación:

- *expandir_entorno.m*
- *expandir_entorno_cortar.m*
- *expandir_entorno_disco.m*
- *expandir_entorno_suavizar.m*

La primera de todas (*expandir_entorno.m*) se basa en extraer la dimensión máxima del robot móvil, concebido como un rectángulo. La dimensión extraída pasa a ser la tolerancia con la que se expanden todos los obstáculos y límites del entorno, de forma que se asegura (en exceso), que la región libre resultante tras la expansión es segura para el movimiento del robot.

Los demás scripts incorporan nuevos tratamientos sobre los resultados de *expandir_entorno.m.* Concretamente, delimitan con mayor precisión los vértices de la expansión mediante un único corte, mediante varios cortes o suavizándolos mediante gran cantidad de cortes. La desventaja de estos scripts, sobre todo en el último, es que, en los obstáculos resultantes de la expansión, se incrementa el número de vértices. Ello desemboca en que, algoritmos que basan su funcionamiento a partir de los vértices del entorno (como diagramas de visibilidad), son sometidos a un incremento muy alto en el coste computacional.

A continuación, se presenta el resultado de expandir un entorno mediante el primer script enlistado. A partir de un entorno constituido por cuatro obstáculos:

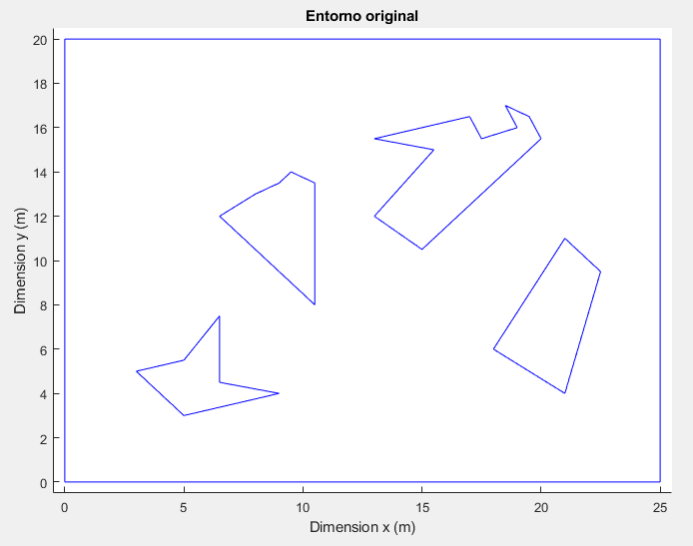

***Figura 28**. Representación de un entorno con cuatro obstáculos.*

Expandiendo el entorno a partir de la función *expandir_entorno.m* se obtiene la siguiente representación:

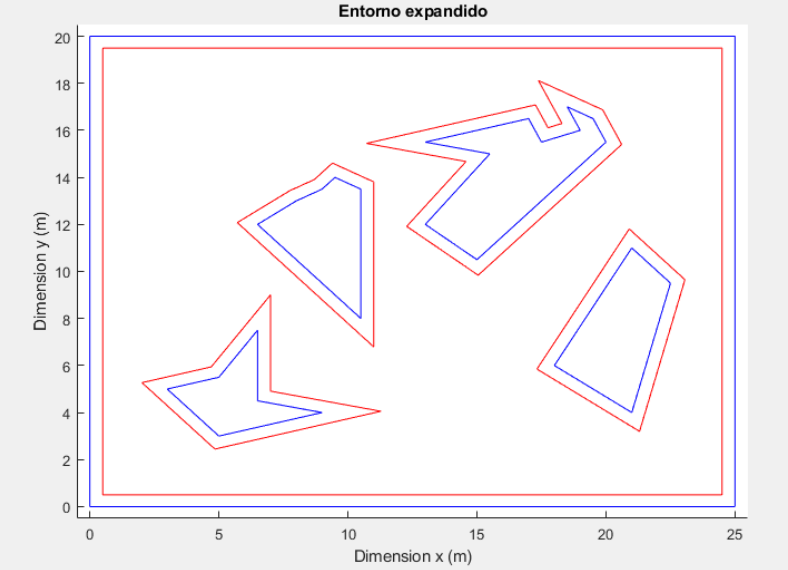

***Figura 29**. Representación de un entorno con cuatro obstáculos expandido.* Se ha realizado la expansión del entorno de la *Figura 27* mediante el script *expandir_entorno.m*, tomando una distancia de 0.5 metros. El entorno original se encuentra representado en azul, mientras que el expandido en rojo.

Como se puede observar en la *Figura 29*, el espacio expandido garantiza la seguridad del robot, pero en algunas regiones (vértices especialmente) la expansión se excede y resulta ineficiente. Es en este punto en el que surge la expansión del entorno mediante suma de Minkowski.

5.2. Suma de Minkowski

En geometría, la suma de Minkowski (también conocida como dilatación) es una operación entre dos vectores de posición (a y b) pertenecientes a dos conjuntos geométricos del espacio euclídeo A y B, respectivamente, de forma que se suma cada vector de A sobre cada vector de B. Formalmente:

$$A + B = \{a + b \mid a \epsilon A, \ \ b \epsilon B\}$$

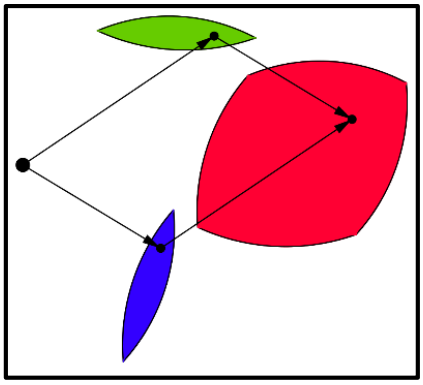

***Figura 30**. Representación de la suma de Minkowski de dos conjuntos. Los dos sumandos, conjuntos verde y azul, dan lugar al conjunto rojo como resultado.*

Como se observa en la *Figura 30*, la suma de Minkowski es un método equivalente a sumar todos los vectores de posición de un conjunto sobre todos los vectores de posición del otro conjunto, lo que da lugar a un nuevo conjunto. Aunque pueda parecer una herramienta lejana a la planificación de movimientos, considerando los conjuntos A y B como el robot móvil y un obstáculo poligonal, se observa la utilidad de esta suma. Tomando los siguiente triángulos como el robot móvil (A) y un obstáculo (B):

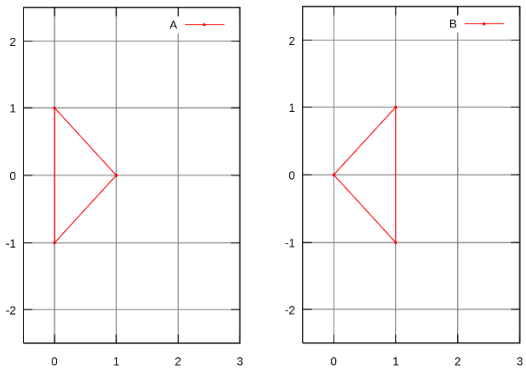

***Figura 31**. Representación de dos triángulos: A (izquierda) y B (derecha).*

La suma de Minkowski de los triángulos es equivalente a 'rodear' (aunque incompletamente) con el triángulo A (robot móvil) el contorno del triángulo B (obstáculo):

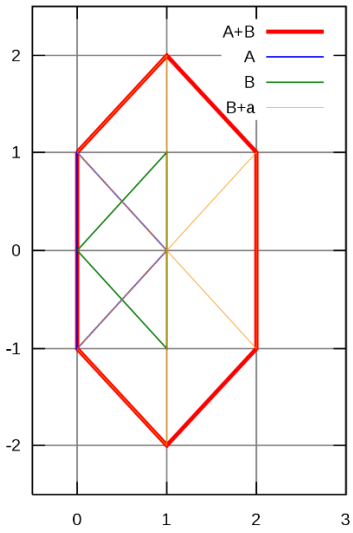

***Figura 32**. Representación de la suma de Minkowski de los triángulos A y B de la Figura 31.*

Extendiendo el concepto a todo el obstáculo, se obtiene un nuevo conjunto que permite expandir el obstáculo con la máxima eficiencia espacial. Cabe recalcar que la suma de Minkowski es únicamente posible para conjuntos convexos. Por ello, aquellos segmentos que formen vértices cóncavos no tienen cabida en este método, ya que, por otro lado, ya poseen máxima eficiencia con la expansión descrita en la Figura 25 (script de Matlab *expandir_entorno.m*).

5.3. Implementación

La función desarrollada que realiza la expansión del entorno por suma de Minkowski recibe el nombre de *expandir_entorno_minkowski.m*. Aunque se trata inherentemente de una suma, el algoritmo basa su funcionamiento en obtener el mismo resultado mediante otros cálculos. Se ha decidido realizar procedimientos alternativos ya que, formalmente, los conjuntos poseen vectores de posición infinitos, por lo que una primera aproximación se basaría en discretizar los conjuntos para que la suma fuese realizable. Una solución basada en valores discretos resultaría en una pérdida de precisión, así como un aumento en el coste computacional.

Existen métodos alternativos como descomponer en triángulos los conjuntos para simplificar la suma de Minkowski, pero en este proyecto se ha optado por desarrollar un razonamiento propio.

El código del programa (y funciones auxiliares) se encuentra como anexo (véase *Índice de Anexos*).

5.3.1. Sintaxis

$$entorno\_expandido = expandir\_entorno\_minkowski\ (robot\_x, robot\_y, robot\_or, robot\_sist, tolerancia, entorno)$$

Donde:

- La salida *entorno_expandido* devuelve una matriz de segmentos tipo entorno que codifica el resultado de la expansión.
- La entrada *robot_x* es un valor numérico real que contiene la longitud del robot en metros.
- La entrada *robot_y* es un valor numérico real que contiene la anchura del robot en metros.
- La entrada *robot_or* es un valor numérico real que informa de la orientación del robot respecto a la horizontal, en grados.
- La entrada *robot_sist* es un vector de posición (tres valores numéricos reales que codifican las dimensiones x, y, z, en metros) que indica el sistema de referencia móvil sobre la región que ocupa el robot. Para introducirlo se debe considerar al robot situado (su esquina inferior izquierda) en el origen de los ejes de coordenadas globales.
- La entrada *tolerancia* representa una extensión en la expansión del entorno referida sobre la dimensión máxima del robot. Su valor se introduce en tanto por ciento.
- La entrada *entorno* aloja la información del entorno original que se pretende expandir, como una matriz de segmentos. Tiene el requerimiento de estar descrito en sentido horario.

5.3.2. Diagrama de flujo

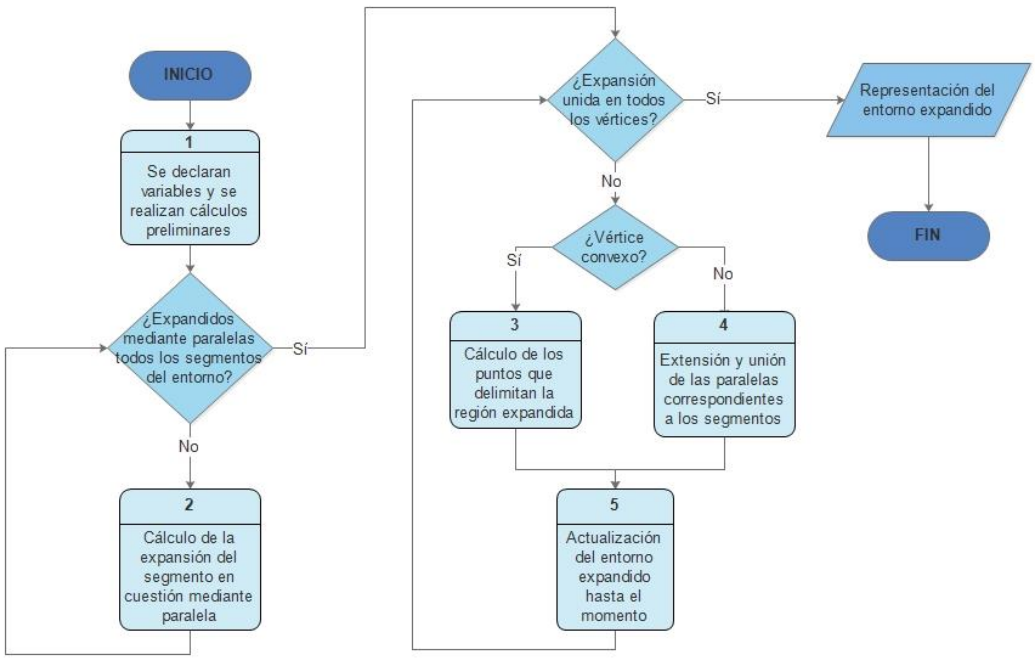

***Figura 33**. Diagrama de flujo de la función expandir_entorno_minkowski.m.*

5.3.3. Lógica del código

A continuación, se expone el razonamiento lógico con el que trabaja el programa y que origina el entorno expandido resultante. Para ello, se procede a detallar los procesos enumerados en la *Figura 33*.

- Proceso 1. Es un proceso preliminar en el que se definen:
  - Una variable entera que contiene el número de segmentos totales.
  - Una matriz que contendrá un entorno expandido preliminar (salida del proceso 2).
  - Una matriz que alojará el entorno expandido como resultado final.

- Proceso 2. Es un proceso que calcula un segmento paralelo al segmento de estudio de tal forma que la separación entre ambos segmentos sea la distancia que habría entre el segmento y el sistema de referencia móvil del robot, cuando este se encuentre 'tocando' el segmento del obstáculo en un solo punto. Gráficamente:

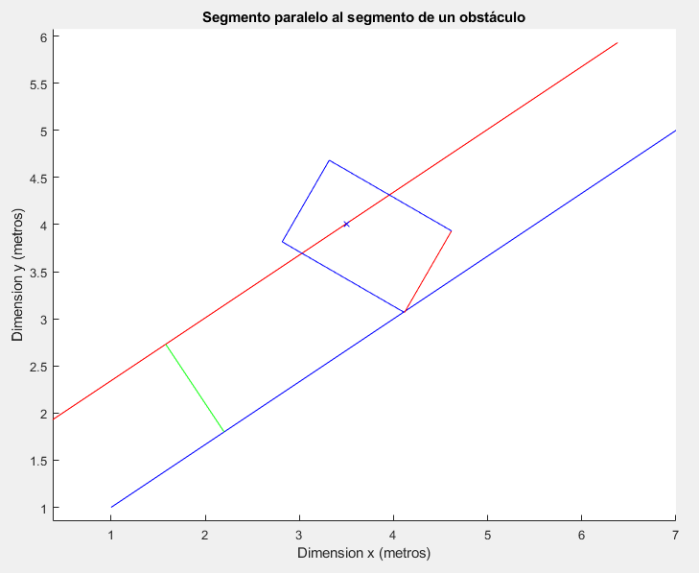

***Figura 34**. Representación gráfica del proceso 2 del algoritmo expandir_entorno_minkowski.m para un segmento.*
Se representa un robot móvil cuyo sistema de referencia se encuentra marcado por una 'x' y el segmento de un obstáculo (en azul). Se muestra, además, el segmento paralelo (en rojo) cuya separación (en verde) permite llevar el sistema de referencia móvil al límite máximo en el que no choca con el obstáculo.

Realizando el cálculo para todos los segmentos del entorno:

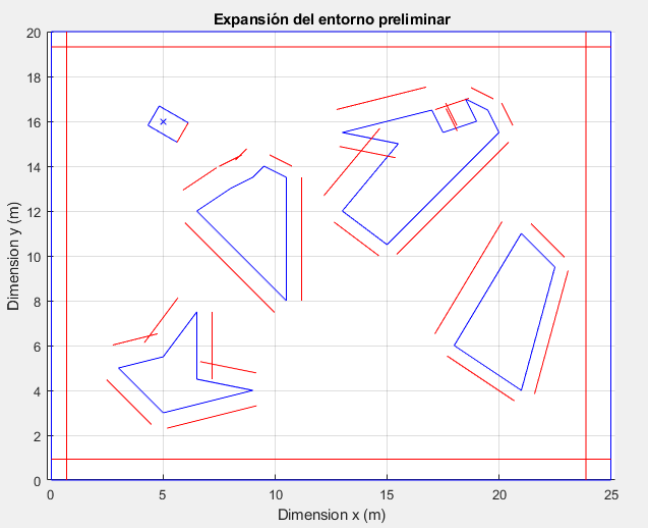

***Figura 35**. Representación gráfica del proceso 2 del algoritmo expandir_entorno_minkowski.m para todo el entorno. Se representa un robot móvil en un entorno (entorno de la Figura 28) expandido preliminarmente mediante paralelas (segmentos en rojo). El entorno original se representa en azul.*

Como se observa en la *Figura 35*, en función de la orientación que tendría cada segmento con el robot si este estuviera en sus proximidades, se calcula una distancia u otra. Este proceso es llevado a cabo por una función auxiliar desarrollada que se ha llamado *minkoswki_paralela_preliminar.m*.

- Proceso 3. Es un proceso que calcula los puntos del trazado que delimita la región expandida de los vértices convexos de los obstáculos. El funcionamiento de este proceso es bastante complejo y podría simplificarse en que clasifica estrictamente el caso en el que se encuentra el robot móvil respecto del vértice para llevar a cabo las oportunas operaciones. A continuación, se enumera los pasos que realiza desde una visión general:

  1) A partir del vértice que toma como entrada, extrae los dos segmentos que lo forman y distingue (teniendo en cuenta un sentido de introducción horario) cuál es el primer segmento y cuál el segundo.

  2) En función de la orientación relativa entre cada segmento y el robot móvil, realizará unos cálculos u otros, siendo posible que la región que rodee al vértice esté delimitada por un máximo de cuatro puntos.

  3) Una vez obtenidos los puntos todos los puntos, se lleva a cabo una comprobación de que no resultan subregiones cerradas dentro de la propia región expandida.

  4) Por último, una vez obtenidos los puntos resultantes que delimitan el entorno, se estructura sus uniones como segmentos del entorno expandido.

  La representación gráfica de este proceso se adjunta tras detallar el proceso 4, pues se ha desarrollado una función auxiliar *minkowski_puntos_vertice.m* que realiza ambos procesos.

- Proceso 4. Es un proceso que calcula el punto de trazado de la región expandida para los vértices cóncavos y los límites del entorno (ya que se podría considerar que los límites del entorno son convexos desde una perspectiva fuera del entorno). Para ello

busca la intersección entre las paralelas preliminares del proceso 2. Esta tarea se lleva a cabo en la función *minkowski_puntos_vertice.m*, conjuntamente con el proceso 3.

El resultado de los procesos 3 y 4 son los siguientes puntos:

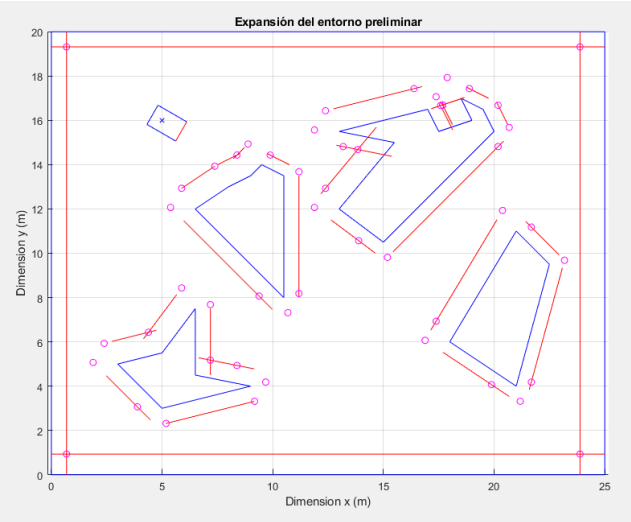

***Figura 36**. Representación gráfica de los procesos 3 y 4 del algoritmo expandir_entorno_minkowski.m.*
Se grafica un robot móvil en un entorno (entorno de la *Figura 29*) expandido preliminar mediante paralelas (segmentos en rojo) y en el que se marca los puntos que delimitan la región expandida. El entorno original se representa en azul.

- Proceso 5. Como último procedimiento, el algoritmo une los puntos formando segmentos y conforma el entorno expandido. Cuando este proceso termina de estructurar el entorno de salida, se realiza una representación gráfica del resultado:

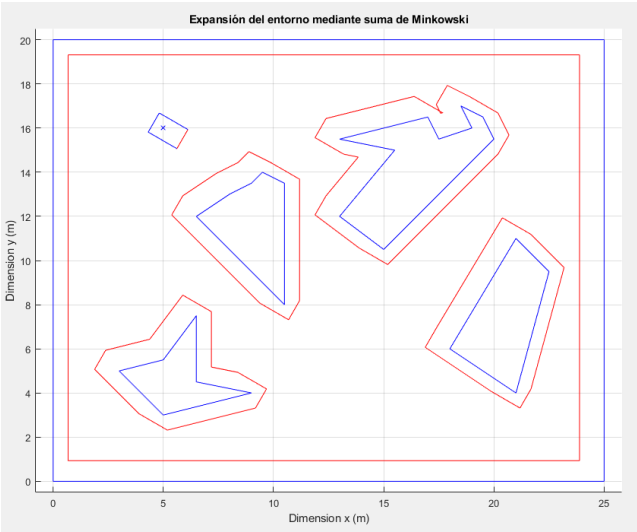

***Figura 37**. Representación gráfica de un entorno expandido por el algoritmo expandir_entorno_minkowski.m.*
Se grafica un robot móvil en un entorno (entorno de la Figura 29) expandido mediante suma de Minkowski (en rojo). El entorno original se representa en azul. No se ha tomado tolerancia adicional en la expansión.

5.3.4. Limitaciones

El algoritmo desarrollado posee ciertas limitaciones que impiden su flexibilidad total, por lo que, si se incurre en alguna de las siguientes condiciones, es probable que el programa no funcione adecuadamente.

- El robot móvil ha sido concebido como un rectángulo. Cualquier otro modelo del robot móvil no será funcional en el algoritmo. No obstante, a partir de la premisa de que debe ser un robot móvil rectangular, el algoritmo provee cierta libertad de caracterización del robot móvil, que coincide con las entradas de la función: longitud del robot, ancho, orientación, posición del sistema de referencia móvil y tolerancia adicional respecto a la dimensión máxima del robot.
- El entorno de entrada al algoritmo debe estar escrito en sentido horario. Es decir, la estructuración de los segmentos que definen tanto los límites del entorno como los obstáculos debe tener un orden de introducción horario.
- La expansión resultante del algoritmo posee una limitación gráfica. Si la expansión es lo suficientemente grande, existe la posibilidad de superponerse con la expansión de otro segmento (no contiguo) u obstáculo. Ello provocaría subregiones dentro del espacio expandido.

  Este hecho constituye una representación gráfica limitada; sin embargo, aquellos algoritmos cuya detección de obstáculos se base en el estudio de los segmentos que los conforman seguirían funcionando adecuadamente, pues estas subregiones se encuentran en el espacio ocupado, sin afectar al espacio libre.

***Figura 38**. Representación de la limitación gráfica que posee el algoritmo expandir_entorno_minkowski.m.*
En la gráfica (no originada por el algoritmo; creada como representación intuitiva) se observa un robot en un entorno (azul) expandido (rojo) en el que aparece una subregión (naranja) dentro de la región expandida.

- La última limitación es compuesta por las funciones auxiliares que son necesarias para el funcionamiento de la función principal *expandir_entorno_minkowski.m*. Distinguiendo entre las desarrolladas en este proyecto y ya existentes en la toolbox:

  - Funciones propias del Trabajo de Fin de Grado:
    - *minkowski_paralela_preliminar.m*
    - *minkowski_puntos_vertice.m*
    - *reescribir_entorno.m*
    - *extrae_informacion_vertice.m*

  - Funciones previamente existentes en la toolbox:
    - *extrae.m*
    - *extrae_obstaculo.m*
    - *inter_seg.m*
    - *interseccion2d.m*
    - *paralela_segmento.m*
    - *precisión.m*
    - *dibuent.m*
    - *despl.m**
    - *dibuagv.m**
    - *dibulin.m**
    - *dibuobj.m**
    - *modlin.m**
    - *rotaz.m**
    - *xamh.m**

Las funciones marcadas con un asterisco (*) únicamente serán requeridas si se desea representar el robot móvil en el entorno.

6. Técnicas de descomposición en celdas y métodos de cálculo de ruta implementados

Como ya ha sido mencionado a lo largo del documento, la planificación de movimiento es una disciplina en constante desarrollo que se encarga de revisar métodos existentes y crear nuevas técnicas que permitan obtener rutas y trayectorias que eviten los obstáculos presentes en el entorno, para que el robot móvil puede circular sin colisiones. En este contexto, se distingue una gran diversidad de filosofías de trabajo, que enfocan el problema desde distintas perspectivas. El Trabajo de Fin de Grado presente pretende abordar una de estas perspectivas, la planificación de movimiento mediante descomposición en celdas del entorno.

Esta sección se plantea como un sumario de los tres siguientes puntos del proyecto, en el que se detalla las cuatro técnicas de descomposición en celdas que han sido implementadas, así como los métodos de cálculo de ruta que son llevados a cabo tras la descomposición. A continuación, se enlista las herramientas de descomposición en celdas abordadas:

- Descomposición en celdas verticales. Es una técnica que busca diferenciar el espacio libre del espacio ocupado por obstáculos mediante celdas verticales, de forma que posteriormente se pueda implementar algoritmos de cálculo de rutas en grafos a partir de las celdas libres. Respecto a esta rama de la descomposición en celdas, se ha desarrollado desde cero dos algoritmos:

  - Descomposición en bandas verticales. Esta técnica consiste en crear celdas rectangulares verticales que delimitan y ocupan el área libre del entorno, a partir de una resolución que el usuario ajuste (entendida en términos de longitud de celda). A partir de este último dato, el algoritmo realiza un barrido horizontal del entorno y estudia como los segmentos que componen los obstáculos delimitarían el área libre del área ocupada.

    Una vez obtenidas las celdas que delimitan el área libre, se crea una red de nodos en la que cada punto accesible del grafo es el punto central de cada celda libre. No obstante, respecto a esta última cuestión, durante el desarrollo propio del algoritmo se ha optado por introducir más puntos accesibles en las celdas cuando ello ha sido conveniente (se trata de un ajuste que puede introducir el usuario). Este aspecto está relacionado con uno de los inconvenientes de este método: la conectividad entre los nodos de celdas vecinas no está asegurada.

    A partir de la red de nodos confeccionada, el entorno es susceptible de aplicar algoritmos de planificación de movimiento basados en grafos, como es el caso del algoritmo de Dijkstra (descrito en la sección '*2.3.2. Configuración del espacio de trabajo y algoritmos de cálculo de rutas*', pág. 19, véase *Índice General*). Para poder hacer uso de este algoritmo (presente en la toolbox) se ha implementado una función auxiliar que permite compatibilizar los resultados obtenidos de la descomposición vertical con las entradas requeridas por el algoritmo de Dijkstra.

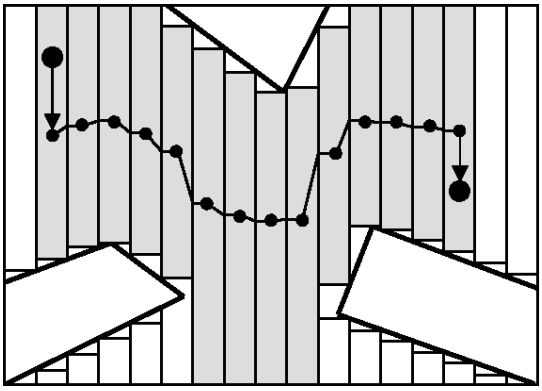

***Figura 39**. Representación de una descomposición en bandas verticales.*

- Descomposición en celdas verticales en configuración trapezoidal. Esta técnica consiste en crear celdas irregulares cuyos límites se sitúen en proyecciones verticales de los vértices de los obstáculos del entorno. El método resulta en celdas (muchas de ellas con forma trapezoidal) que delimitan el área libre del área ocupada.

  Para obtener ese resultado, el algoritmo busca las posibles proyecciones verticales de cada vértice, hacia arriba y hacia abajo, y expande esas proyecciones hasta que intersectan con algún obstáculo o los límites del entorno. Este método, a diferencia del barrido, presenta una desventaja: no es posible para el usuario ajustar una resolución de celda, ya que depende de los propios obstáculos del entorno. Sin embargo, esta técnica permite asegurar la conectividad de los nodos de celdas vecinas, a diferencia del barrido.

  Una vez obtenidos los segmentos que, indirectamente, representan las celdas, se crea una red de nodos en la que cada punto perteneciente al grafo es el punto central de cada segmento vertical.

  A partir de la red de nodos confeccionada, el entorno es susceptible de aplicar algoritmos de planificación de movimiento basados en grafos, como es el caso del algoritmo de Dijkstra (descrito en la sección '*2.3.2. Configuración del espacio de trabajo y algoritmos de cálculo de rutas*', pág. 19, véase *Índice General*). Al igual que en el caso de las bandas verticales, para poder hacer uso de este algoritmo se ha diseñado una función auxiliar que permite compatibilizar Dijkstra con la descomposición en configuración trapezoidal.

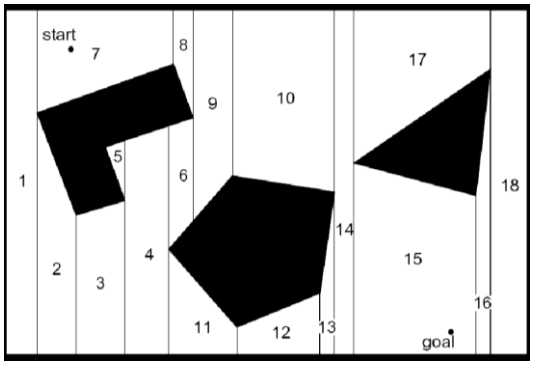

***Figura 40**. Representación de una descomposición en celdas verticales por configuración trapezoidal.*

- Descomposición en celdas uniformes. Se trata de una metodología que descompone el entorno en celdas rectangulares y uniformes entre sí, a partir de la resolución (referida tanto al largo como al ancho de cada celda) ajustada por el usuario.

  Una vez se posee la plantilla (en celdas) que descompone el entorno, se analiza cada celda para determinar si esta se encuentra en la región libre del entorno o la región ocupada por obstáculos. Cabe destacar que las celdas ocupadas poseerán a su vez dos subestados, íntegramente ocupadas (cuando toda el área de la celda se encuentre dentro del obstáculo) y parcialmente ocupadas (cuando una región, pero no toda el área de la celda, esté ocupada). Sin embargo, a efectos prácticos de este método, se debe considerar ambas opciones como celdas ocupadas, sin distinciones entre ellas.

  Tras obtener la descomposición finalizada, el entorno es susceptible de aplicar diversos algoritmos de planificación de movimientos. Al igual que en el caso de las celdas verticales, se podría extraer el punto central de cada celda libre y construir una red de nodos, para posteriormente aplicar algoritmos de búsqueda en grafos como Dijkstra o A*. No obstante, en el presente Trabajo de Final de Grado, retomando la senda del anterior trabajo en celdas uniformes, se ha optado por aplicar un campo potencial de costes que barre todas las celdas desde la celda inicial (en la que está contenida el punto inicial de la ruta) hasta la celda final (en la que se encuentra el punto de destino).

  La principal limitación de esta herramienta es el coste computacional asociado a la resolución de celda deseada. Puesto que el análisis abarca a todas las celdas (aunque en el desarrollo propio se ha buscado alternativamente búsquedas más eficientes) y los métodos de estudio de intersección con obstáculos son también exhaustivos, una precisión elevada se traduce en tiempos de simulación deficientes, especialmente para el caso de entornos complejos.

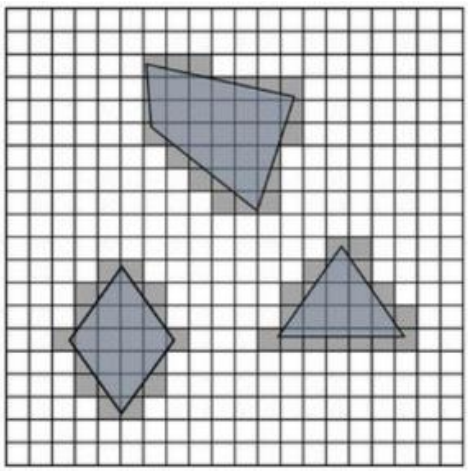

***Figura 41**. Representación de una descomposición en celdas uniformes.*

- Descomposición en Quadtree. Es la última de las herramientas desarrolladas relativas a la descomposición en celdas. Su utilidad radica en la principal limitación de las celdas uniformes, que es el coste computacional asociado al análisis del elevado número de celdas resultantes de la descomposición. Para solventar este problema, los algoritmos que descomponen el entorno en Quadtree tratan de delimitar en cada iteración celdas lo más grande posible que distingan las regiones del entorno en libres y ocupadas.

El método recibe su nombre en que la división del entorno sigue un esquema en diagrama de árbol (tree), en la que cada iteración ramifica o divide el entorno en cuatro celdas (quad). Los pasos que sigue son los siguientes:

1) Divide la celda actual (en la primera iteración es el entorno en su conjunto) en cuatro celdas uniformes.
2) Analiza cada una de las celdas para distinguir entre tres estados posibles: parcialmente ocupada por un obstáculo, íntegramente ocupada por un obstáculo o libre.
3) En el caso de que el estado sea libre o íntegramente ocupada, la celda no requiere de un análisis en celdas más pequeñas, por lo que la ramificación concluye.
4) En el caso de que el estado sea parcialmente ocupada, la celda posee una región que aún se puede delimitar con mayor precisión, por lo que se toma la celda como la 'actual' y se vuelve al paso 1).
5) El proceso finaliza cuando se alcanza el nivel de ramificación ajustado por el usuario.

El mayor inconveniente de esta técnica se encuentra en la resolución de celda deseada, es decir, el nivel de ramificación. Puesto que cada iteración divide las celdas en cuatro, incrementar el grado de ramificación posee una relación exponencial de coste computacional de base 4. Esta es la razón por la que ajustar adecuadamente la resolución es crítico para algoritmos basados en Quadtree.

Una vez realizada la descomposición en Quadtree, como sucedía en la descomposición en celdas verticales, se puede construir una red de nodos en la que se incluya cada punto central de las celdas libres del Quadtree. Con la posterior aplicación de algoritmos de planificación de movimiento basados en la búsqueda en grafos, como el algoritmo de Dijkstra (descrito en la sección '*2.3.2. Configuración del espacio de trabajo y algoritmos de cálculo de rutas*', pág. 19, véase *Índice General*), se puede calcular rutas sobre el entorno descompuesto en celdas. Del mismo modo que con la descomposición en celdas verticales, ha sido necesario desarrollar una función auxiliar que permite compatibilizar ambos algoritmos.

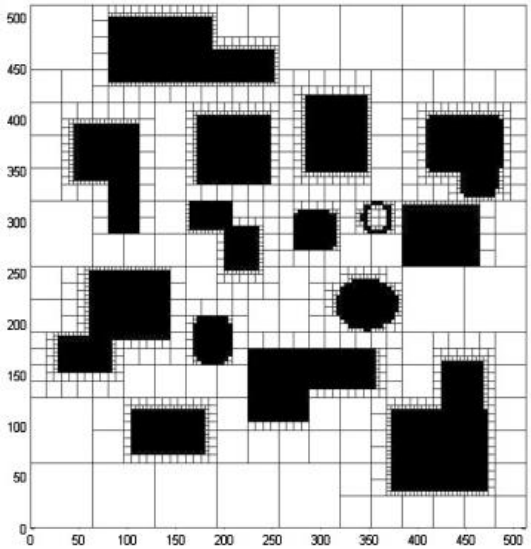

***Figura 42**. Representación de una descomposición en Quadtree*

7. Descomposición del entorno en celdas verticales

Descomponer el entorno en celdas verticales constituye un método útil para la planificación de movimientos, ya que permite delimitar el espacio libre del espacio ocupado mediante técnicas más simples y menos demandantes en términos de coste computacional que las celdas uniformes. Sobre el entorno descompuesto originado es susceptible aplicar posteriormente algoritmos de cálculo y generación de trayectorias. Como se adelantaba en la sección '6. Técnicas de descomposición en celdas y métodos de cálculo de rutas implementados', se ha implementado dos técnicas de esta filosofía: descomposición en bandas verticales y descomposición en celdas verticales mediante configuración trapezoidal.

7.1. Implementación de bandas verticales

La función desarrollada que realiza la descomposición del entorno en bandas verticales es *bandas_verticales.m*. A continuación, se expone su metodología y funcionamiento.

El código del programa (y funciones auxiliares) se encuentra como anexo (véase *Índice de Anexos*).

7.1.1. Sintaxis

$$[celdas\_resultantes,\ celdas\_obstaculos,\ nodos] = bandas\_verticales(entorno,\ subs,\ longitud\_celda,\ num\_nodos)$$

Donde:

- La salida *celdas_resultantes* devuelve un arreglo tipo cell que posee codificadas todas las celdas libres que delimitan la región libre de colisión en el entorno.
- La salida *celdas_obstaculos* retorna un arreglo tipo cell que posee codificadas todas las celdas ocupadas que delimitan los obstáculos.
- La salida nodos es una matriz de $nx3$ que devuelve los nodos accesibles de las celdas libres. n es el número de nodos en celdas libres. Cada nodo se representa por sus tres coordenadas cartesianas respecto al sistema de referencia global: $[x \quad y \quad z]$.
- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos. Posee el requerimiento de estar descrito en sentido horario.
- La entrada *subs* es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión x. Este valor de entrada deberá valer 0 si la entrada *longitud_celda* posee un valor no nulo.
- La entrada *longitud_celda* es un valor real que posee la resolución en la dimensión x de cada una de las celdas en las que se dividirá el entorno. Este valor de entrada deberá valer 0 si la entrada *subs* posee un valor no nulo.
- La entrada *num_nodos* es un valor entero que representa el número de nodos máximo que se podría extraer de una banda completamente libre. En función del tamaño de cada celda, ajusta este valor máximo y distribuye los nodos uniformemente.

7.1.2. Diagrama de flujo

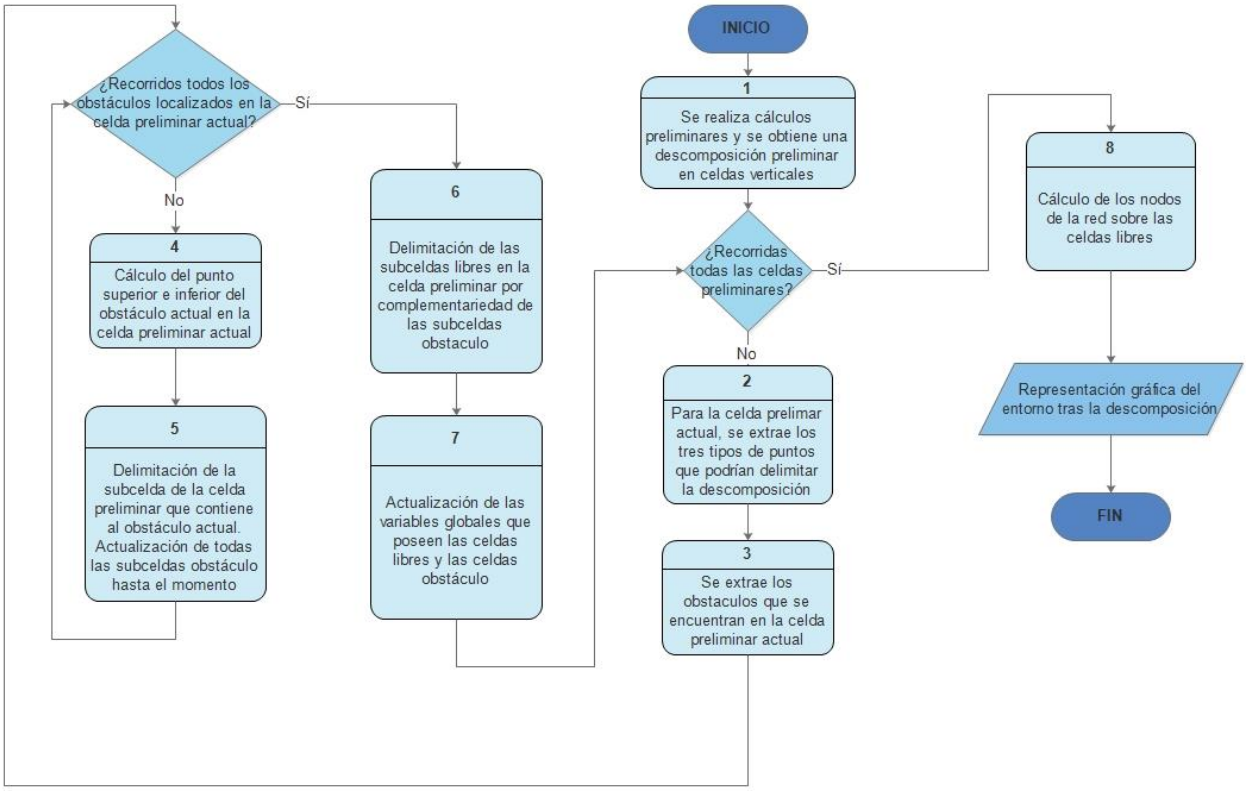

***Figura 43**. Diagrama de flujo de la función bandas_verticales.m.*

7.1.3. Lógica del código

A continuación, se expone el razonamiento lógico con el que trabaja el programa y que origina el entorno descompuesto en bandas verticales. Para ello, se procede a detallar los procesos enumerados en la *Figura 43*.

- Proceso 1. Es un proceso preliminar en el que se definen:
  - Una matriz que contendrá los nodos de la red.
  - Las dimensiones del entorno para simplificar posteriores cálculos.
  - En función de la entrada introducida *subs* o *longitud_celda*, se recalcula su alternativa. En caso de haber introducido ambas entradas, se prioriza el parámetro *longitud_celda* para continuar el algoritmo.
  - Se calcula y delimita unas bandas verticales preliminares que se encuentran vacías, es decir, no poseen información de delimitación de obstáculos. Gráficamente:

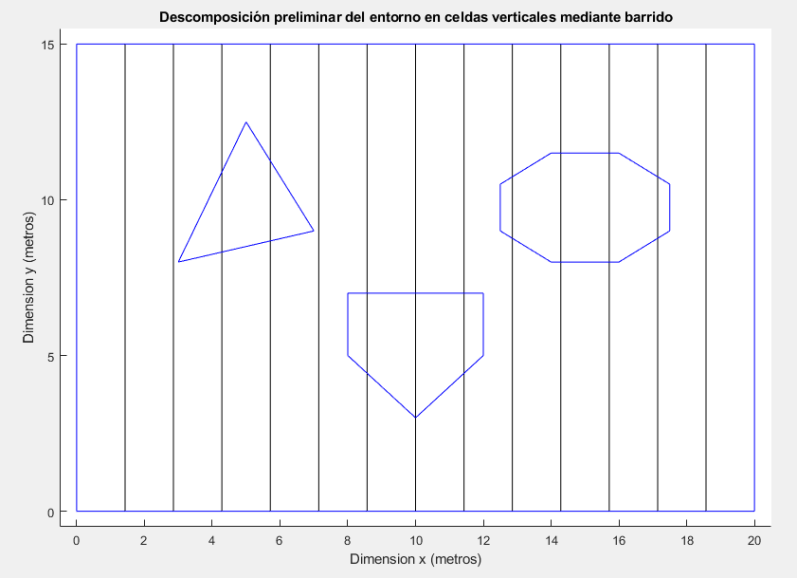

***Figura 44**. Representación de una descomposición vertical preliminar.*

- Proceso 2. Este proceso trata de obtener los puntos que son susceptibles de delimitar los obstáculos y el área libre para construir posteriormente las celdas. En la banda preliminar vertical que se encuentra estudiando el algoritmo, busca tres tipos de puntos: intersecciones del lado izquierdo de la celda, intersecciones del lado derecho y vértices convexos que 'sobresalgan' verticalmente. Partiendo de la *Figura 44*, los puntos susceptibles de delimitar las celdas serán:

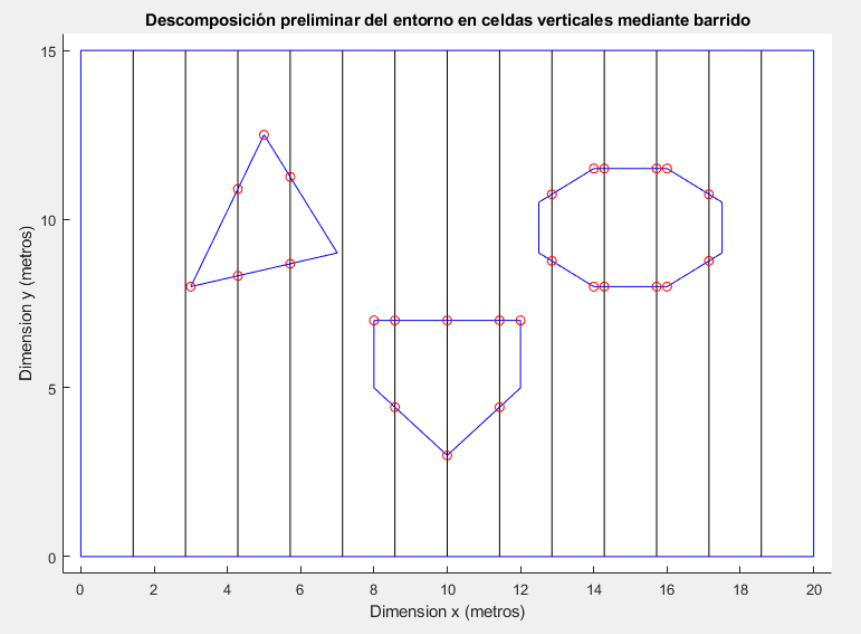

***Figura 45**. Representación de una descomposición vertica preliminar con los puntos susceptibles de delimitar las celdas.*

- Proceso 3. Se trata de un proceso que prepara al algoritmo para cálculos posteriores. Sobre la banda preliminar vertical de estudio, se consulta los puntos susceptibles del proceso 2. De ellos se extrae en un vector los índices de los obstáculos que se encuentran en la banda. Por último, reestructura el vector de forma que solo aparezca cada índice una vez y en orden ascendente.

- Proceso 4. Este proceso consulta, sobre la actual banda preliminar, cada obstáculo a partir de los índices obtenidos en el proceso 3. En el obstáculo que se encuentre estudiando, busca el punto con mayor y con menor dimensión vertical de entre los puntos susceptibles (intersecciones con el lado izquierdo, intersecciones con el lado derecho y vértices convexos que 'sobresalen' verticalmente). Cada par de puntos obtenidos representan los límites de la subcelda que contendrá al obstáculo.

- Proceso 5. A partir del par de puntos obtenido del proceso 4, se puede delimitar la subcelda que aloja al obstáculo de estudio sobre la actual banda preliminar. Una vez obtenida esta subcelda, se incorpora a un arreglo tipo cell que posee todas las subceldas de obstáculos de la actual banda preliminar.

- Proceso 6. Una vez se posee todas las subceldas que alojan los obstáculos de la celda preliminar actual, por complementariedad, se calculan las subceldas libres.

- Proceso 7. Durante este proceso se actualiza las variables globales que poseen todas las celdas libres y ocupadas analizadas hasta el momento.

- Proceso 8. Una vez finaliza la descomposición del entorno en bandas verticales, se puede extraer los nodos accesibles para posteriores generaciones de rutas con algoritmos basados en grafos. Para solventar el principal problema que posee la descomposición en bandas; de cada celda, en función de su tamaño vertical, se extrae hasta un máximo de nodos que puede ajustar el usuario (procurando una distribución eficiente y uniforme). Ello se debe a que, si únicamente se extrajera el punto central de cada celda como posible nodo de ruta, habría muchas celdas contiguas cuyos nodos no estarían conectados, ya que atravesarían celdas de obstáculos.

Una vez finalizados todos los procesos, se representa gráficamente el resultado:

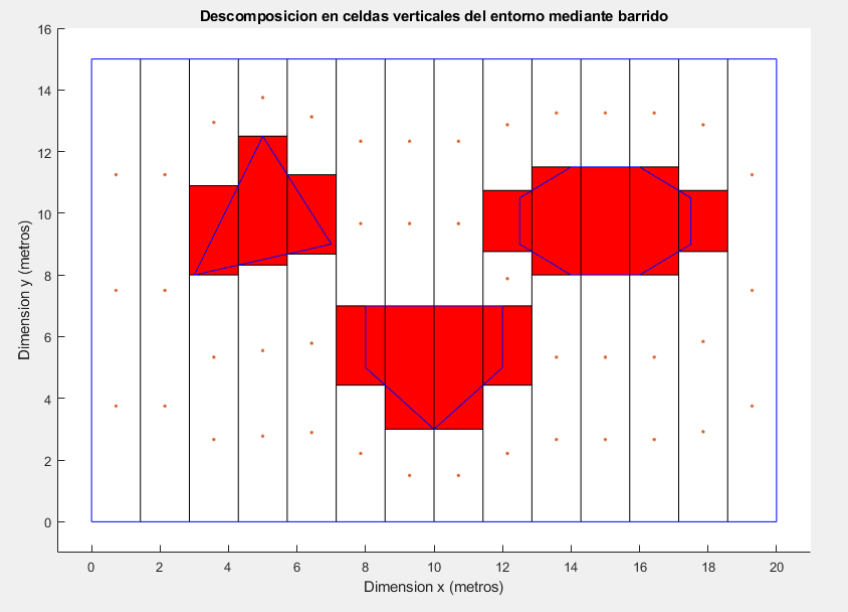

***Figura 46**. Representación gráfica de la descomposición de un entorno en bandas verticales.*
En blanco se representan las celdas libres, en rojo las ocupadas y marcados en naranja los nodos extraídos sobre las celdas libres (se ha escogido un valor de 3 para num_nodos).

7.1.4. Limitaciones

El algoritmo desarrollado posee ciertas limitaciones que impiden su flexibilidad total. La principal limitación de la herramienta desarrollada se debe a la propia naturaleza del método. Otras limitaciones han surgido como resultado del desarrollo realizado.

- El entorno de entrada al algoritmo debe estar escrito en sentido horario. Es decir, la estructuración de los segmentos que definen tanto los límites del entorno como los obstáculos debe tener un orden de introducción horario.
- El principal inconveniente de esta técnica es que no se puede asegurar la conectividad entre nodos cuyas celdas son adyacentes. Un claro ejemplo se puede observar en la *Figura 46*. Marcando la celda en cuestión:

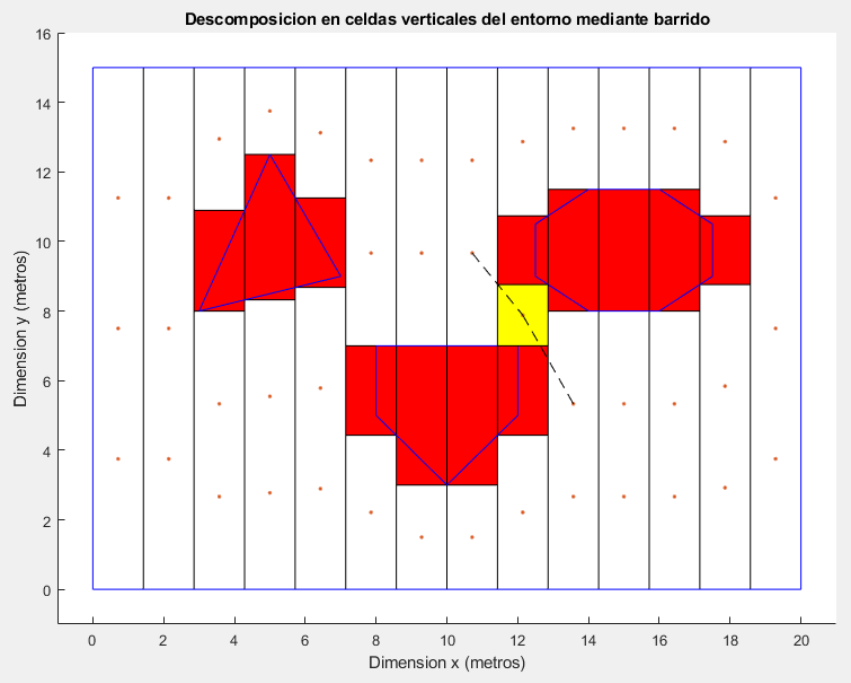

***Figura 47**. Representación de la limitación en conectividad de la descomposición de un entorno en bandas verticales. En blanco se representan las celdas libres, en rojo las ocupadas y marcados en naranja los nodos extraídos sobre las celdas libres (se ha escogido un valor de 3 para num_nodos). Se ha destacado, además, la celda que presenta la limitación en amarillo.*

Se podría concluir para este caso que el resultado no afectaría ya que, aunque atraviesa una región ocupada por un obstáculo, no colisiona con el obstáculo en realidad. Sin embargo, no se puede inferir que la región de la celda ocupada que atraviesa siempre será libre, por lo que se debe considerar que los nodos poseen un arco de coste infinito.

Aunque se trata de un problema relevante de este método, se puede llegar a solucionar sin mayor dificultad aumentando el número de nodos accesibles por celda libre o aumentando la resolución en cantidad de celdas. El único inconveniente de esta solución es que la precisión requerida es incierta, ya que solo se puede comprobar su conectividad una vez finalizada la descomposición. Es por ello que la solución resulta arbitraria aun siendo funcional en muchas simulaciones.

- Otra limitación de esta herramienta se produce cuando existen regiones libres que se encuentran delimitadas en la dimensión y por el mismo obstáculo. Un ejemplo de un entorno en el que sucede:

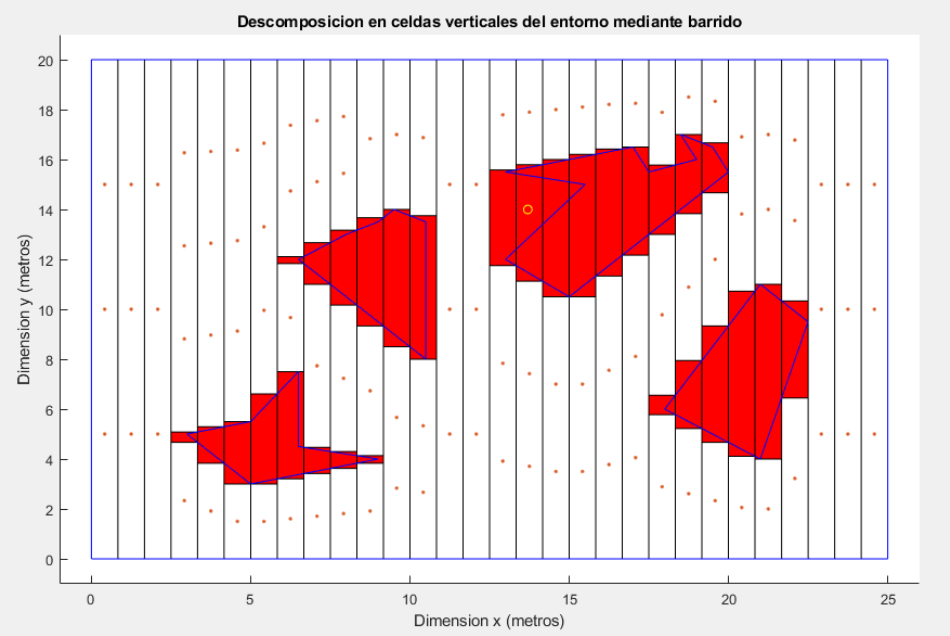

***Figura 48**. Representación de la limitación de regiones libres marcadas como ocupadas en la descomposición de un entorno en bandas verticales.*
En blanco se representan las celdas libres, en rojo las ocupadas y marcados en naranja los nodos extraídos sobre las celdas libres (se ha escogido un valor de 3 para num_nodos). Se ha destacado, además, la región que presenta la limitación con un punto amarillo.

Este inconveniente puede resultar insignificante o dar lugar a que no sea posible generar la trayectoria. De ello depende que alguno de los puntos inicial y/o final se encuentre en esta área incorrectamente delimitada.

- La última limitación es compuesta por las funciones auxiliares que son necesarias para el funcionamiento de la función principal *bandas_verticales.m*. Distinguiendo entre las desarrolladas en este proyecto y ya existentes en la toolbox:

  - Funciones propias del Trabajo de Fin de Grado:
    - *reescribir_entorno.m*
    - *extrae_informacion_vertice.m*
  - Funciones previamente existentes en la toolbox:
    - *extrae_obstaculo.m*
    - *inter_seg.m*
    - *interseccion2d.m*
    - *precision.m*
    - *dibuent.m*

7.2. Implementación configuración trapezoidal

La función desarrollada que realiza la descomposición del entorno en celdas verticales en configuración trapezoidal es *celdas_verticales_configuracion_trapezoidal.m*. A continuación, se expone su metodología y funcionamiento:

El código del programa (y funciones auxiliares) se encuentra como anexo (véase *Índice de Anexos*).

7.2.1. Sintaxis

$$[segmentos\_trapezoidal,\ nodos] = celdas\_verticales\_configuracion\_trapezoidal\ (entorno)$$

Donde:

- La salida *segmentos_trapezoidal* retorna una matriz de segmentos que son los que delimitan las celdas originadas por la descomposición.
- La salida *nodos* devuelve una matriz de puntos con todos aquellos nodos accesibles de la región libre tras la descomposición.
- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos. Posee el requerimiento de estar descrito en sentido horario.

7.2.2. Diagrama de flujo

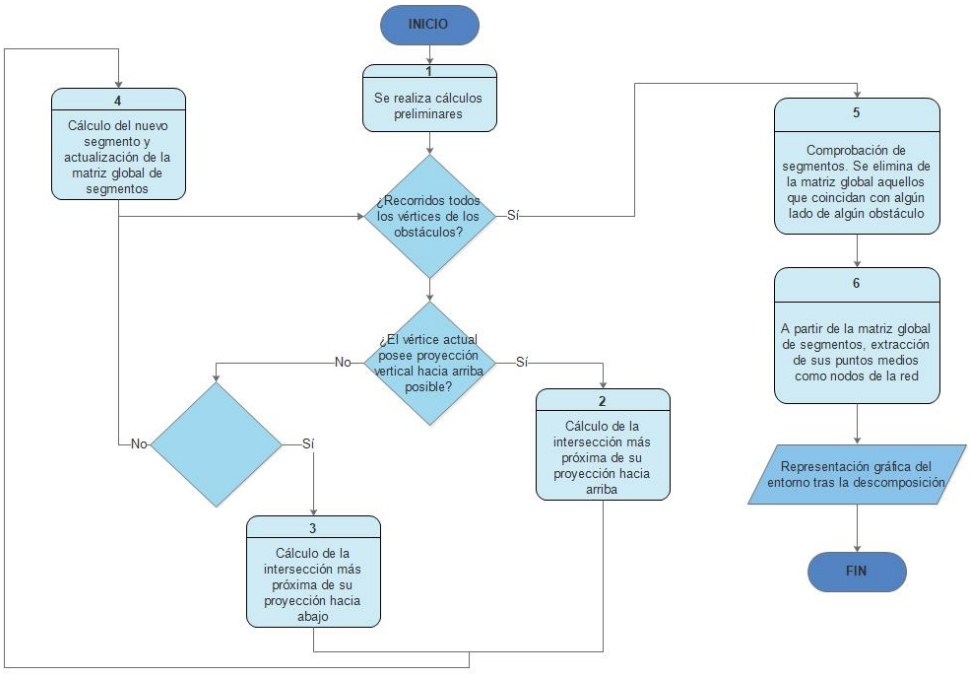

***Figura 49**. Diagrama de flujo de la función celdas_verticales_configuracion_trapezoidal.m.*

7.2.3. Lógica del código

A continuación, se expone el razonamiento lógico con el que trabaja el programa y que origina el entorno descompuesto mediante configuración trapezoidal. Para ello, se procede a detallar los procesos enumerados en la *Figura 49*.

- Proceso 1. Es un proceso preliminar en el que se definen:
  - Una matriz que contendrá los segmentos de salida de la función.
  - Las dimensiones del entorno para simplificar posteriores cálculos.
  - Una matriz con todos los vértices de los obstáculos del entorno.

- Proceso 2. Este proceso, que en realidad también abarca el análisis de si el vértice de estudio posee proyección vertical hacia arriba, cosiste en obtener la intersección más próxima de la proyección vertical hacia arriba del vértice de estudio. Para ello, se crea un segmento vertical preliminar desde el vértice en cuestión hasta el límite superior del entorno. A partir de este segmento, se recorre todos los segmentos que codifican el entorno buscando intersecciones. La intersección más próxima al vértice de estudio será el punto final del segmento que delimitará la celda.

- Proceso 3. Sigue un razonamiento similar al proceso 2. Este proceso, que en realidad también abarca el análisis de si el vértice de estudio posee proyección vertical hacia abajo, cosiste en obtener la intersección más próxima de la proyección vertical hacia abajo del vértice de estudio. Para ello, se crea un segmento vertical preliminar desde el vértice en cuestión hasta el límite inferior del entorno. A partir de este segmento, se recorre todos los segmentos que codifican el entorno buscando intersecciones. La intersección más próxima al vértice de estudio será el punto final del segmento que delimitará la celda.

- Proceso 4. A partir de los puntos inicial y final obtenidos de los procesos 2 y 3, se estructura un nuevo segmento que delimita un lado de una celda. Además, se actualiza la matriz global que almacena estos segmentos incorporando el nuevo calculado.

- Proceso 5. Debido a los métodos de cálculo implementados, existe la posibilidad de que aquellos lados completamente verticales de los obstáculos sean detectados propiamente como segmentos que delimitan las celdas. Estos segmentos deben ser eliminados de la matriz global ya que, aunque gráficamente delimitan correctamente el área libre del área ocupada, no se puede extraer de ellos nodos accesibles para posteriores procedimientos de generación de trayectorias.

- Proceso 6. De la matriz global que posee todos los segmentos que delimitan las celdas se extrae el punto medio de cada una de ellas. Estos puntos constituirán la red de nodos que permitan calcular rutas basadas en grafos. Además, de las celdas límite situadas a la izquierda y derecha del entorno, se calcula su punto centro y se añade a los nodos accesibles. Ello permite trazar rutas por tales celdas, ya que de otra manera no sería posible.

Una vez finalizados todos los procesos, se representa gráficamente el resultado:

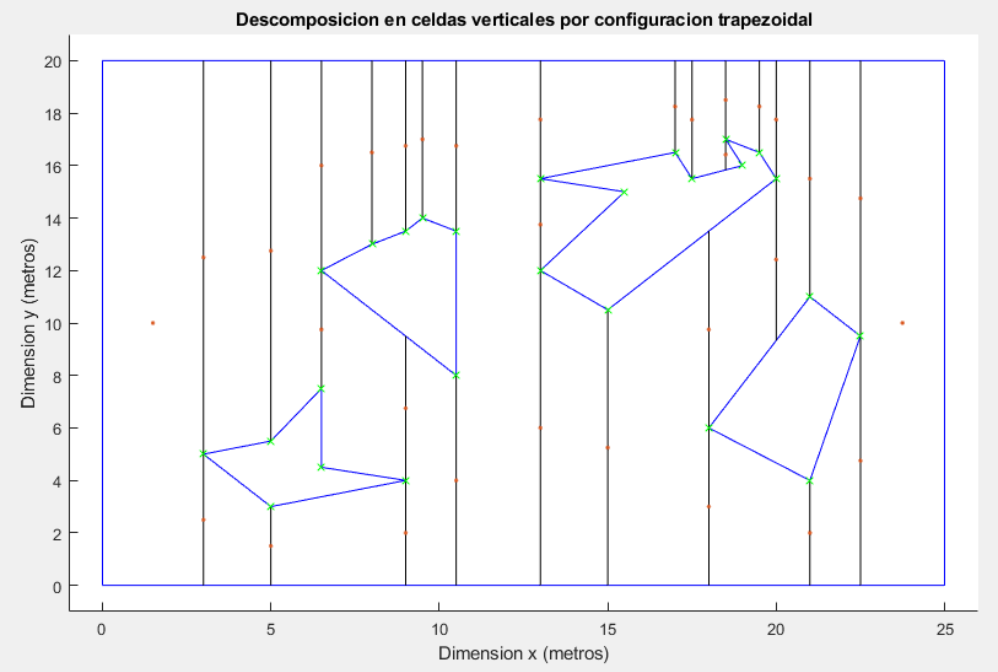

***Figura 50**. Representación gráfica de la descomposición de un entorno en celdas verticales en configuración trapezoidal.*
Se ha resaltado los vértices de los obstáculos con cruces verdes. Asimismo, se encuentran marcados en naranja los nodos extraídos de los segmentos y de las celdas inicial y final.

7.2.4. Limitaciones

El algoritmo desarrollado posee ciertas limitaciones que impiden su flexibilidad total. La principal limitación de la herramienta desarrollada es la resolución de la ruta, que, al depender únicamente del entorno, su trazado puede ser complicado para las capacidades móviles del robot.

- El entorno de entrada al algoritmo debe estar escrito en sentido horario. Es decir, la estructuración de los segmentos que definen tanto los límites del entorno como los obstáculos debe tener un orden de introducción horario.
- El principal inconveniente de esta técnica es la ineficiencia en términos de resolución de la ruta que se pretenda generar. Puesto que el método cosiste en extraer nodos a partir de los propios vértices de los obstáculos del entorno, a mayor complejidad del entorno mayor cantidad de nodos en la red. Por tanto, entornos cuyos obstáculos se modelen con mayor simplicidad poseen la desventaja de que el trazado posterior de la ruta será más errático e ineficiente.
- La última limitación es compuesta por las funciones auxiliares que son necesarias para el funcionamiento de la función principal *celdas_verticales_configuracion_trapezoidal.m*. Distinguiendo entre las desarrolladas en este proyecto y ya existentes en la toolbox:

- Funciones propias del Trabajo de Fin de Grado:
  - *reescribir_entorno.m*
  - *extrae_informacion_vertice.m*
- Funciones previamente existentes en la toolbox:
  - *extrae_obstaculo.m*
  - *inter_seg.m*
  - *interseccion2d.m*
  - *precisión.m*
  - *dibuent.m*

7.3. Algoritmo de Dijkstra en entornos descompuestos en celdas verticales

Una vez se posee el entorno descompuesto en celdas verticales es posible aplicar algoritmos de cálculo y generación rutas basados en exploración de grafos. Para este caso, se ha escogido el algoritmo de Dijkstra, que, a partir de los nodos de una red y la información referida al coste de los arcos que conectan los nodos de esa red, es capaz de explorar todos los nodos y buscar el recorrido más corto desde un nodo inicial hasta un nodo final. Para más información, el funcionamiento de este algoritmo ha sido expuesto en la sección '*2.3.2. Configuración del espacio de trabajo y algoritmos de cálculo de rutas*' (pág. 19, véase *Índice General*).

El algoritmo de Dijkstra ya se encuentra desarrollado en la toolbox de Matlab (*Dijkstra.m*) de la que parte este Trabajo de Fin de Grado. Su sintaxis es la siguiente:

$$[Camino,\ Coste] = Dijkstra\ (matriz\_costes,\ nodo\_inicial,\ nodo\_final)$$

Donde:

- La salida *Camino* devuelve un vector que posee los índices de los nodos que se debe recorrer consecutivamente para alcanzar el punto final desde el inicial.
- La salida *Coste* retorna un valor real que es suma de todos los arcos que se debe atravesar para alcanzar el punto final desde el inicial.
- La entrada *matriz_costes* es una matriz de nxn donde n es el número total de nodos, incluyendo el nodo inicial y final. Cada elemento (i, j) poseerá el coste asociado al arco que une el nodo i con el nodo j. Además, para los grafos construidos por descomposición en celdas verticales, los arcos son bidireccionales. Por ello el coste (i, j) es igual al coste (j, i).
- La entrada *nodo_inicial* es un valor entero (desde 1 hasta n) que identifica la posición del nodo inicial en la matriz de costes.
- La entrada *nodo_final* es un valor entero (desde 1 hasta n) que identifica la posición del nodo final en la matriz de costes.

El algoritmo de Dijkstra de la toolbox trabaja conjuntamente con la función *dibujar_camino.m* para trazar gráficamente el *Camino* salida de la función *Dijkstra.m*. La sintaxis de esta función es la siguiente:

$$Trayecto = dibujar\_camino\ (Camino,\ vertices,\ Color)$$

Donde:

- La salida *Trayecto* es una matriz de gráficos de líneas que almacena la representación gráfica de la ruta.
- La entrada *Camino* es un vector que posee los índices de los nodos que se debe recorrer consecutivamente para alcanzar el punto final desde el inicial. Coincide con la salida del mismo nombre de la función *Dijkstra.m*.
- La entrada *vertices* es una matriz que posee todos los nodos del grafo, incluidos los puntos inicial y final.
- La entrada *Color* es un vector de tres elementos que codifica mediante RGB el color de la ruta cuando esta se representa.

Para poder aplicar este algoritmo sobre los entornos descompuestos en bandas verticales y por configuración trapezoidal, se ha desarrollado dos funciones auxiliares que calculan la matriz de costes para cada uno de los métodos, y ajustan los nodos inicial y final a los índices 1 y 2, respectivamente, de la matriz de costes.

7.3.1. Generación de trayectorias en entornos descompuestos en bandas verticales

Para el cálculo de la matriz de costes que permita aplicar el algoritmo de Dijkstra a entornos descompuestos en bandas verticales, se ha desarrollado la función *bandas_preparaciones_ruta.m*.

Su sintaxis:

$$[pts\_en\_area\_libre,\ nodos,\ matriz\_costes] = bandas\_preparaciones\_ruta\ (entorno,\ celdas\_libres,\ celdas\_obstaculos,\ subs,\ longitud\_celda,\ nodos,\ pt\_in,\ pt\_fin)$$

Donde:

- La salida *pts_en_area_libre* es un valor entero que valdrá 1 cuando ambos puntos, inicial y final, se encuentren en celdas libres. Su valor será 0 en caso contrario.
- La salida *nodos* retorna una matriz que posee los nodos de la red con los puntos inicial y final añadidos en los índices 1 y 2, respectivamente. Se diferencia de la entrada con el mismo nombre en que la entrada no posee los nodos de los puntos inicial y final añadidos.
- La salida *matriz_costes* es una matriz de nxn donde n es el número total de nodos, incluyendo el nodo inicial y final cuyos índices son 1 y 2, respectivamente. Cada elemento (i, j) poseerá el coste asociado al arco que une el nodo i con el nodo j. Además, los arcos son bidireccionales, por lo que el coste (i, j) es igual al coste (j, i).

- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos.
- La entrada *celdas_libres* es un arreglo tipo cell que es la salida *celdas_resultantes* de la función *bandas_verticales.m*. Codifica las celdas cuya región no posee obstáculos.
- La entrada *celdas_obstaculos* es un arreglo tipo cell que es la salida *celdas_obstaculos* de la función *bandas_verticales.m*. Codifica las celdas en cuya región se localizan los obstáculos.
- La entrada *subs* es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión x. Este valor de entrada deberá valer 0 si la entrada *longitud_celda* posee un valor no nulo.
- La entrada *longitud_celda* es un valor real que posee la resolución en la dimensión x de cada una de las celdas en las que se dividirá el entorno. Este valor de entrada deberá valer 0 si la entrada *subs* posee un valor no nulo.
- La entrada *nodos* es una matriz de puntos que posee los nodos del grafo originados por la función *bandas_verticales.m*. Se diferencia de la salida con el mismo nombre en que a la salida se le incorporan los puntos inicial y final.
- La entrada *pt_in* es un vector de tres coordenadas cartesianas que localiza el punto inicial de la ruta.
- La entrada *pt_fin* es un vector de tres coordenadas cartesianas que localiza el punto final de la ruta.

La función calcula todos los costes entre todos los nodos de la red y los almacena en la matriz de costes, que es el argumento de entrada para la función *Dijkstra.m*. También actualiza la variable nodos, que será el parámetro de entrada con nombre *vertices* de la función *dibujar_camino.m*.

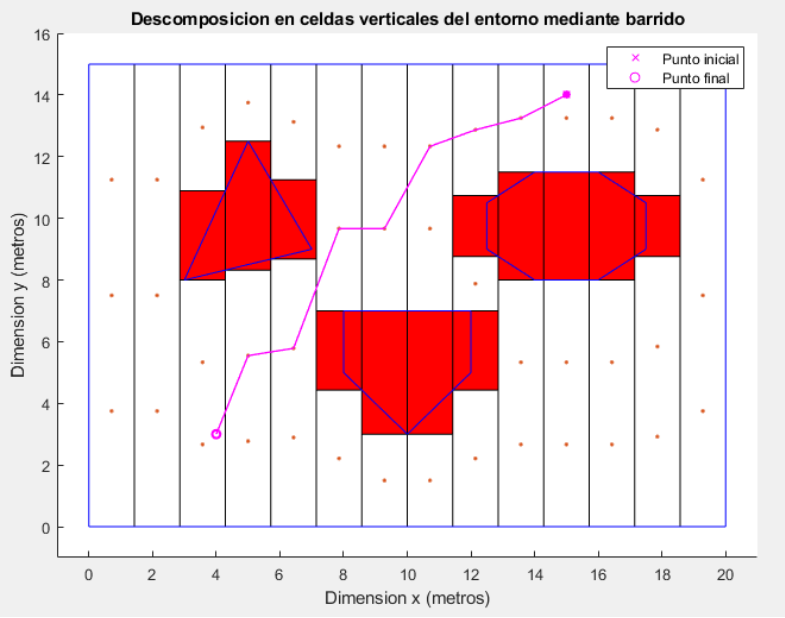

***Figura 51**. Representación gráfica del cálculo de ruta mediante Dijkstra a partir del entorno descompuesto en bandas verticales (Figura 46).*

7.3.2. Generación de trayectorias en entornos descompuestos en celdas verticales por configuración trapezoidal

Para el cálculo de la matriz de costes que permita aplicar el algoritmo de Dijkstra a entornos descompuestos en celdas verticales mediante configuración trapezoidal, se ha desarrollado la función *trapezoidal_preparaciones_ruta.m*.

Su sintaxis:

$$[pts\_en\_area\_libre,\ nodos,\ matriz\_costes] = trapezoidal\_preparaciones\_ruta\ (entorno,\ segmentos\_trapezoidal,\ nodos,\ pt\_in,\ pt\_fin)$$

Donde:

- La salida *pts_en_area_libre* es un valor entero que valdrá 1 cuando ambos puntos, inicial y final, se encuentren en celdas libres. Su valor será 0 en caso contrario.
- La salida *nodos* retorna una matriz que posee los nodos de la red con los puntos inicial y final añadidos en los índices 1 y 2, respectivamente. Se diferencia de la entrada con el mismo nombre en que la entrada no posee los nodos de los puntos inicial y final añadidos.
- La salida *matriz_costes* es una matriz de nxn donde n es el número total de nodos, incluyendo el nodo inicial y final cuyos índices son 1 y 2, respectivamente. Cada elemento (i, j) poseerá el coste asociado al arco que une el nodo i con el nodo j. Además, los arcos son bidireccionales, por lo que el coste (i, j) es igual al coste (j, i).
- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos.
- La entrada *segmentos_trapezoidal* es una matriz que posee todos aquellos segmentos que delimitan los lados de las celdas que descomponen el entorno por configuración trapezoidal. Se trata de la salida, con mismo nombre, de la función *celdas_verticales_configuracion_trapezoidal.m*.
- La entrada *nodos* es una matriz de puntos que posee los nodos del grafo originados por la función *celdas_verticales_configuracion_trapezoidal.m*. Se diferencia de la salida con el mismo nombre en que a la salida se le incorporan los puntos inicial y final.
- La entrada *pt_in* es un vector de tres coordenadas cartesianas que localiza el punto inicial de la ruta.
- La entrada *pt_fin* es un vector de tres coordenadas cartesianas que localiza el punto final de la ruta.

La función calcula todos los costes entre todos los nodos de la red y los almacena en la matriz de costes, que es el argumento de entrada para la función *Dijkstra.m*. También actualiza la variable nodos, que será el parámetro de entrada con nombre *vertices* de la función *dibujar_camino.m*.

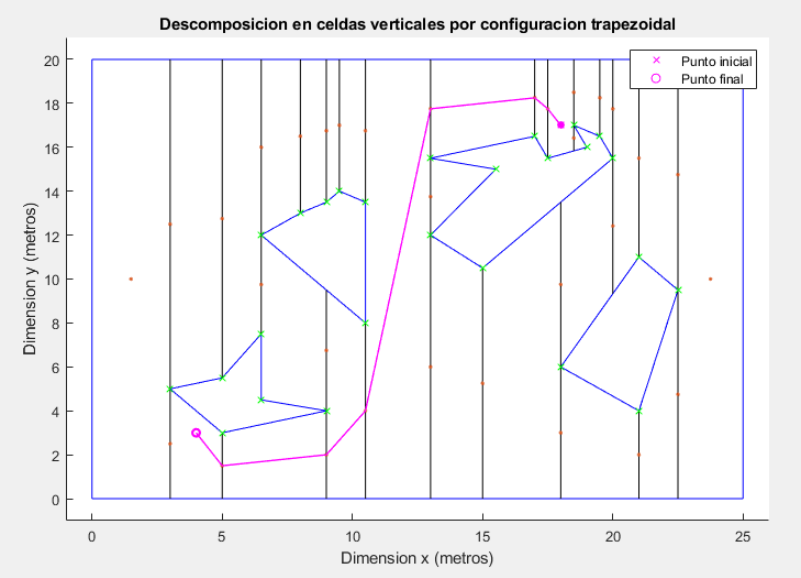

***Figura 52**. Representación gráfica del cálculo de ruta mediante Dijkstra a partir del entorno descompuesto en celdas verticales en configuración trapezoidal (Figura 50).*

7.4. Comparativa entre barrido y configuración trapezoidal

Como se ha dejado entrever a lo largo de la sección, el principal inconveniente de cada técnica es la principal ventaja de la otra. La decisión de implementación, de un método u otro, debe ser sopesada tomando en consideración estos aspectos.

En el caso de las bandas verticales, la principal ventaja que se propone es la precisión de la ruta a realizar. Incrementando la resolución de las celdas, la trayectoria entre los puntos inicial y final mejora su eficiencia espacial, permitiendo acercase a una ruta óptima. Sin embargo, este incremento de la resolución también va acompañado de un aumento en el coste computacional. En contrapartida, la configuración trapezoidal da lugar a celdas que dependen del propio entorno, por lo que la precisión de la ruta no es ajustable por el usuario. Ello puede dar lugar a giros bruscos en la trayectoria.

En el caso de la configuración trapezoidal la principal ventaja es que se asegura la conectividad entre nodos pertenecientes a celdas adyacentes, lo que simplifica enormemente el diseño del algoritmo. Por otro lado, las bandas no pueden garantizar la conectividad entre nodos pertenecientes a celdas adyacentes, lo que obliga a la comprobación de todas las conexiones. Asimismo, puede producir caminos no realizables si no se toma la suficiente resolución.

Por último, se puede concluir preliminarmente que el método de bandas es más útil para entornos simples cuya descomposición no dé lugar a espacios pequeños entre obstáculos o regiones libres dentro de los mismos, de modo que la conectividad entre el punto inicial y final sea posible. Para el caso opuesto, la técnica de descomposición por configuración trapezoidal resulta más útil en entornos complejos, ya que la conectividad está asegurada y la complejidad del entorno conlleva mayor resolución y cantidad de nodos en el grafo. Sin embargo, ninguna opción es definitiva y, en función de la aplicación concreta, se deberá valorar otros indicadores.

8. Descomposición del entorno en celdas uniformes

La descomposición del entorno en celdas uniformes es un método geométrico que analiza regiones discretas y uniformes en las que se divide el entorno, para determinar en cada caso si el área comprendida es transitable por el robot. El objetivo de esta técnica es delimitar a partir de una resolución ajustada por el usuario el espacio libre del espacio ocupado por obstáculos. En este sentido, se trata de un procedimiento similar a la descomposición en celdas verticales. No obstante, la descomposición en celdas uniformes ofrece una resolución mayor que permitirá calcular posteriormente trayectorias más precisas, a costa también de un coste computacional más elevado.

Es la implementación de esta herramienta en Matlab la que ha servido de precedente al presente Trabajo de Fin de Grado, que ha abierto la puerta a explorar técnicas de planificación de movimiento de robots móviles basadas en la descomposición del entorno en celdas.

Los antiguos alumnos de Máster *JL. Guardiola* y *R. Sebastiá* desarrollaron un script funcional de la descomposición en celdas uniformes de un entorno y posterior aplicación de un campo potencial. A partir de este trabajo, se ha desarrollado una implementación propia que pretende ofrecer una nueva perspectiva. En esta sección se abarcará el trabajo original, el algoritmo nuevo y su comparativa.

8.1. Implementación del algoritmo original

La script original es un programa que realiza todas las etapas de la descomposición y las va representando progresivamente en diferentes mapas. El nombre del script en Matlab es *celdas1.m*. Puesto que se trata de un script y no una función, se detallará las características del programa con un guion un poco distinto al llevado a cabo hasta ahora en el presente documento.

8.1.1. Diagrama de flujo

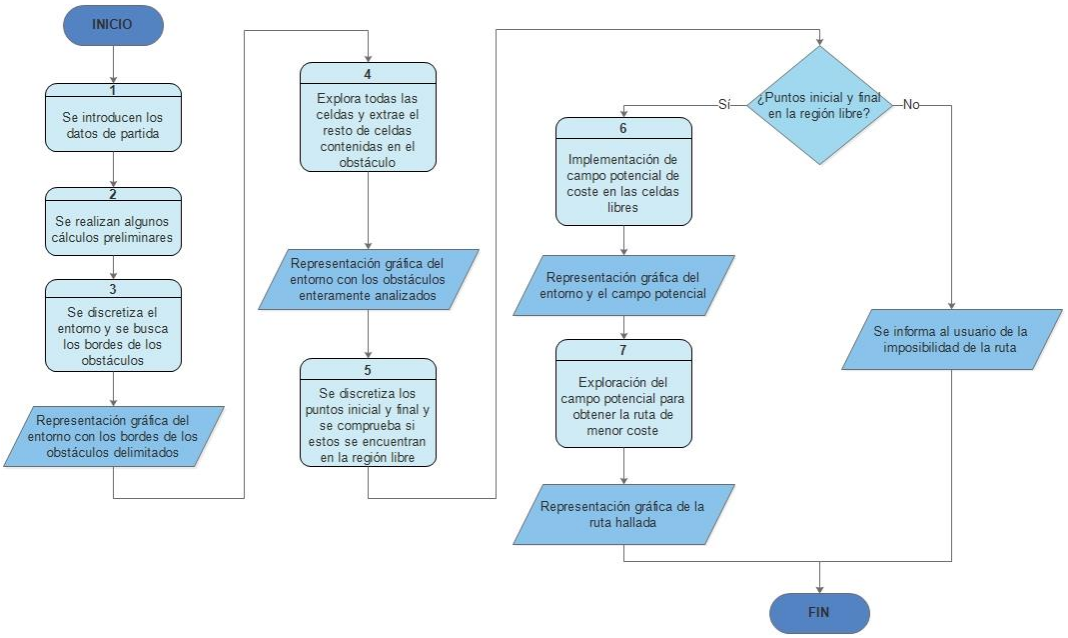

***Figura 53**. Diagrama de flujo del script celdas1.m.*

8.1.2. Lógica del código

A continuación, se expone el razonamiento lógico con el que trabaja el programa original y que construye el entorno descompuesto en celdas uniformes. Para ello, se procede a detallar los procesos enumerados en la *Figura 53*.

- Proceso 1. Es un proceso en el que se introducen los datos de partida. Estos son:
  - El entorno sobre el que se pretende realizar la descomposición.
  - Los puntos inicial y final de la ruta. Estos puntos son vectores de tres coordenadas en los que se introduce primero la coordenada y y después la coordenada x. Aunque la revisión de este algoritmo parece indicar que ha sido un error al explorar las posteriores matrices, ya que el primer elemento son las filas (es decir, coordenada y) y después las columnas (es decir, coordenada x), no se ha podido concluir definitivamente en ello.
  - Un valor llamado *paso* que representa el número por el que se dividirá las dimensiones del entorno (previamente pasadas a milímetros) y dará lugar al número de celdas del entorno discretizado.

- Proceso 2. Realiza una conversión de los parámetros del entorno y los puntos inicial y final. Supone los datos de entrada en metros y los transforma a milímetros. Para estas conversiones utiliza la función *demamm.m*.

- Proceso 3. Este proceso discretiza el entorno y lo transforma en una matriz cuyo tamaño depende del *paso* introducido. Por otro lado, obtiene una delimitación preliminar de los obstáculos. Para ello, extrae del entorno original (pasado a milímetros) los vértices que componen los segmentos, y estructura la ecuación de la recta que los modela. Tras ello

discretiza los valores de la recta y estudia la correspondencia en posición para determinar donde estarían situados en el entorno discretizado. Por último, realiza una representación gráfica del proceso. Todo ello es realizado por la función *buscaborde.m*.

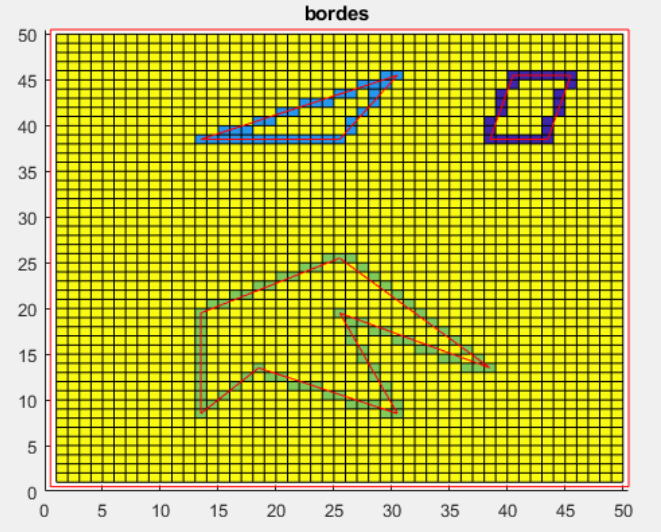

***Figura 54**. Ejemplo de representación gráfica del proceso 3 del script celdas1.m.*

- Proceso 4. Consiste en el barrido de la matriz resultante del proceso 2 para completar el interior de los obstáculos. Mediante el recorrido de todas las celdas, cada vez que se detecta un borde de un obstáculo, se registra el cambio. Una vez finalizado el barrido, las celdas contenidas entre dos cambios asignados al mismo obstáculo se traducirán en celdas contenidas dentro del obstáculo. En caso contrario, serán celdas libres. Tras ello, se exhibe una representación gráfica del proceso. Las funciones que se encargan de realizar este procedimiento son dos: *etiquetado.m* y *reetiquetado.m*.

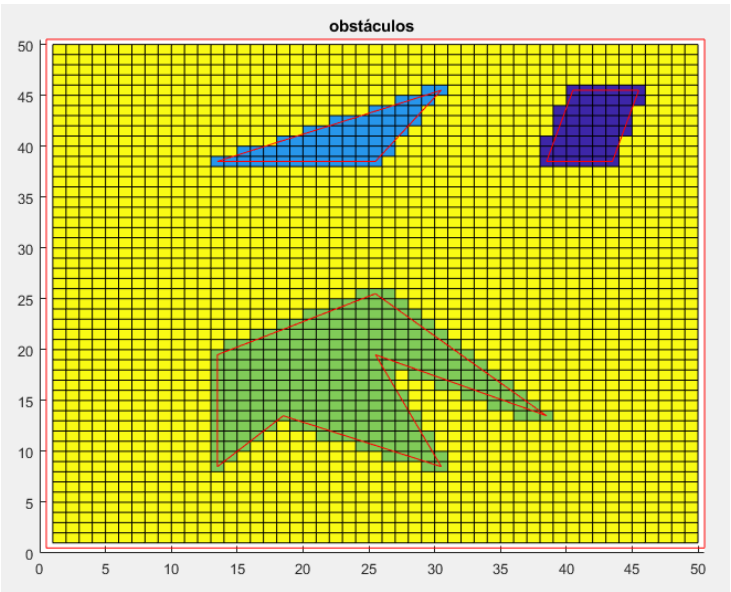

***Figura 55**. Ejemplo de representación gráfica del proceso 4 del script celdas1.m.*

- Proceso 5. Este proceso discretiza los puntos inicial y final para obtener su equivalencia en la matriz discreta que codifica el entorno (salida del proceso 4). Tras ello, comprueba que ambos puntos se localizarían en celdas designadas como libres. Las funciones encargadas de este proceso son *discretizapunto.m* y *puntosvalidos.m*.

- Proceso 6. El proceso consiste en expandir un campo potencial de coste desde la celda en la que se localiza el punto inicial hasta la celda en la que se encuentra el punto final. La expansión se realiza por 8-vecinos (es decir, el coste de transición entre celdas horizontales, verticales y diagonales se supone el mismo). Tras ello, muestra al usuario una representación gráfica del campo potencial.

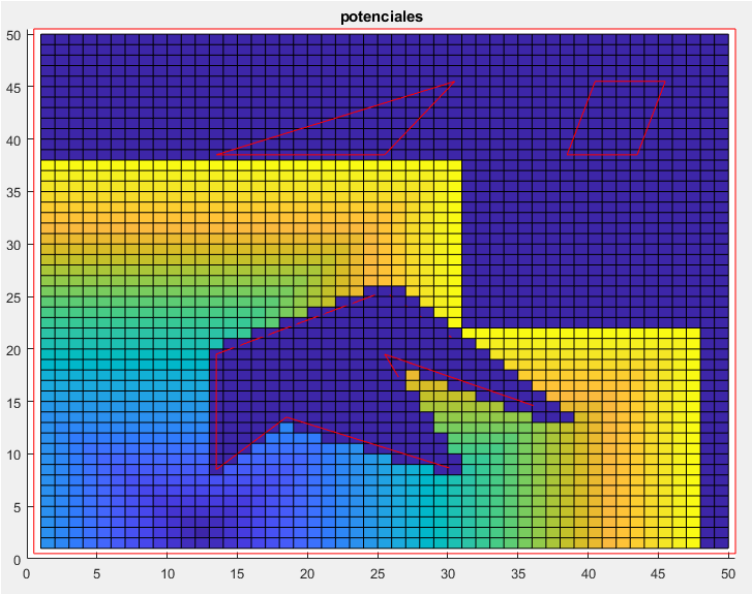

***Figura 56**. Ejemplo de representación gráfica del proceso 6 del script celdas1.m.*
Se ha tomado el punto de origen (azul más intenso, menor coste) desde el que se expande el campo hasta el punto final (amarillo más intenso, mayor coste).

- Proceso 7. Mediante la exploración del campo potencial obtenido del proceso 7, se realiza una exploración que busca conectar el nodo inicial con el nodo final, procurando el menor coste posible. Una vez se obtiene la trayectoria, esta se grafica para que el usuario pueda observarla. La función que se ocupa de esta tarea es *dibujodelarutamascorta.m*.

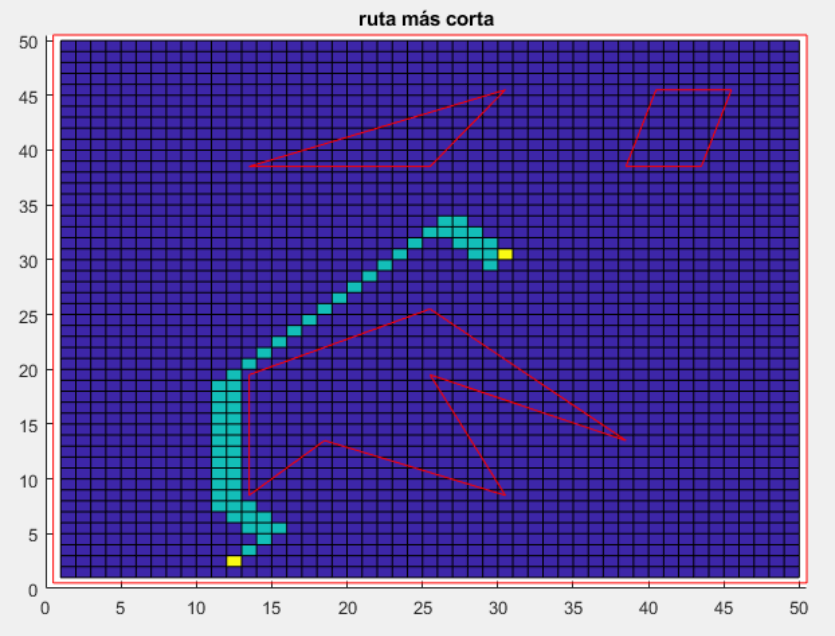

***Figura 57**. Ejemplo de representación gráfica del proceso 7 del script celdas1.m.*
Calcula la ruta a partir del campo potencial de la Figura 54.

8.1.3. Limitaciones

El algoritmo original posee ciertas limitaciones que surgen fruto de su método de implementación.

- El principal inconveniente, aunque no constituye una limitación por sí misma, es la estructura del script. Se presenta un programa no integrado en una única función y que no permite separar en su ejecución la descomposición en celdas uniformes de posteriores herramientas de cálculo de trayectorias. Además, el programa realiza pasos muy complejos con poca documentación relativa a las funciones implementadas, así como pocos comentarios.
- Como ya se mencionaba en el proceso 1, la introducción de los puntos inicial y final no es intuitiva.
- La descomposición resultante posee una resolución que depende del parámetro paso. Este parámetro se puede variar para ajustar la resolución, pero su ajuste no es intuitivo y se vuelve un ajuste indirecto.
- La delimitación de los obstáculos no es completamente rigurosa. Al realizar una conversión discreta de los segmentos que codifican los obstáculos para buscar los bordes, puede ocurrir que ciertas celdas ocupadas no sean correctamente delimitadas. Ello se puede observar en la *Figura 54*, en el triángulo superior el lado más prolongado interseccciona a una celda que no es marcada como ocupada.
- La representación gráfica del campo potencial se lleva a cabo mediante la función surface de Matlab. Esta función en realidad es una función derivada de la función surf, que realiza una representación gráfica en tres dimensiones. La tercera coordenada de esta representación es la altura de esa región. Mediante el comando surface el programa toma una proyección en el plano de la representación gráfica en tres

dimensiones, y ajusta a cada valor que se le introduce (coste) un color en función de su altura en la tercera coordenada. Aunque el método es correcto y muy útil para representar un campo potencial, no se ha terminado de depurar adecuadamente las alturas de los elementos en su representación gráfica. Es por ello que se puede observar como los obstáculos no son adecuadamente trazados en la *Figura 56*.

- La última limitación es compuesta por las funciones auxiliares que son necesarias para el funcionamiento del script principal *celdas1.m*. Las funciones requeridas para su funcionamiento:
    - *buscaborde.m*
    - *demamm.m*
    - *dibuent.m*
    - *dibujodelarutamascorta.m*
    - *discretizapunto.m*
    - *etiquetado.m*
    - *reetiquetado.m*
    - *puntosvalidos.m*
    - *generaruta.m*

8.2. Implementación del algoritmo propio

La función nueva desarrollada que realiza la descomposición del entorno en celdas uniformes es *celdas_uniformes.m*. A continuación, se expone su metodología y funcionamiento.

El código del programa (y funciones auxiliares) se encuentra como anexo (véase *Índice de Anexos*).

8.2.1. Sintaxis

$[celdas\_un,\ matriz\_un,\ subs\_x,\ subs\_y] = celdas\_uniformes\ (entorno,\ subs\_x,\ subs\_y,\ longitud\_celda,\ altura\_celda)$

Donde:

- La salida *celdas_un* retorna un arreglo tipo cell de dimensiones $subs\_y \times subs\_x$ compuesto por matrices que poseen las coordenadas de cada una de las celdas en las que se descompone el entorno.
- La salida *matriz_un* devuelve una matriz de dimensiones $subs\_y \times subs\_x$, cuyos elementos son valores numéricos asociados a cada celda del arreglo *celdas_un* de sus mismos índices. Se utiliza 0 para designar celda libre y -2 para designar celda ocupada. El -1 se reserva para las celdas que posean los puntos inicial y final de la ruta. Los valores positivos se guardan para representar el coste

- La salida *subs_x* es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión x. Se devuelve por si fuese recalculado y fuera preciso para aplicar el campo potencial (cálculo de rutas).
- La salida *subs_y* es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión y. Se devuelve por si fuese recalculado y fuera preciso para aplicar el campo potencial (cálculo de rutas).
- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos.
- La entrada *subs_x* es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión x. Este valor de entrada deberá valer 0 si la entrada *longitud_celda* posee un valor no nulo.
- La entrada *longitud_celda* es un valor real que posee la resolución en la dimensión x de cada una de las celdas en las que se dividirá el entorno. Este valor de entrada deberá valer 0 si la entrada *subs_x* posee un valor no nulo.
- La entrada *subs_y* es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión y. Este valor de entrada deberá valer 0 si la entrada *altura_celda* posee un valor no nulo.
- La entrada *altura_celda* es un valor real que posee la resolución en la dimensión y de cada una de las celdas en las que se dividirá el entorno. Este valor de entrada deberá valer 0 si la entrada *subs_y* posee un valor no nulo.

8.2.2. Diagrama de flujo

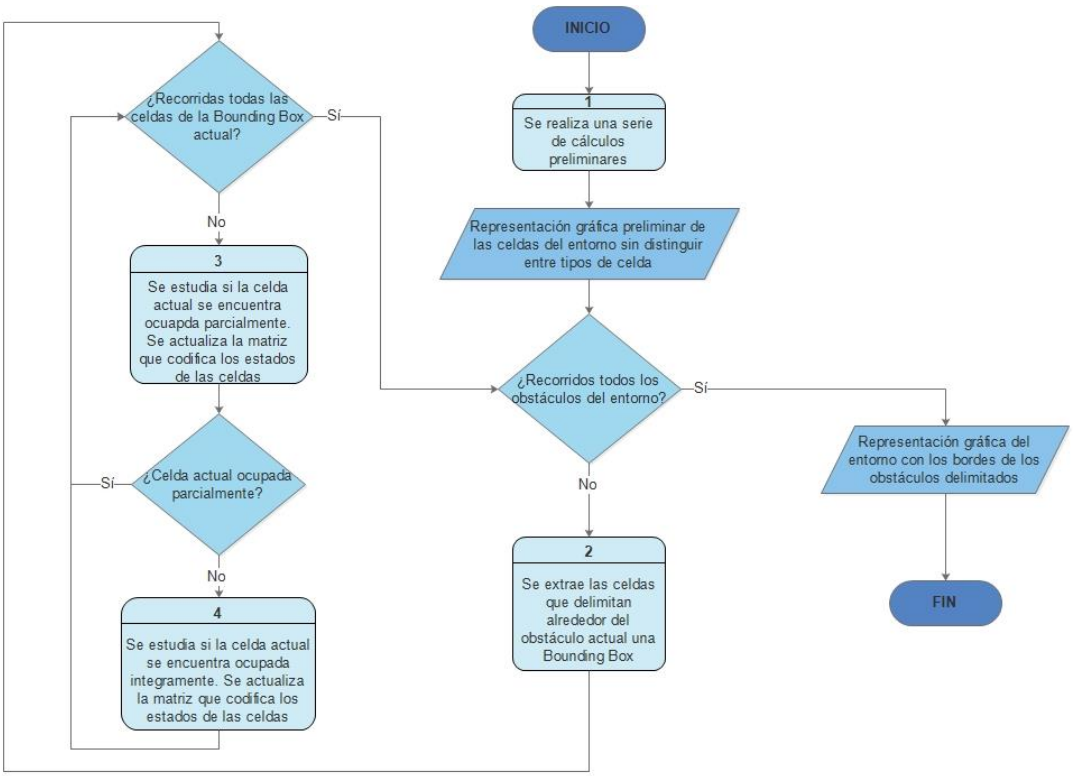

***Figura 58**. Diagrama de flujo de la función celdas_uniformes.m.*

8.2.3. Lógica del código

A continuación, se expone el razonamiento lógico con el que trabaja el programa nuevo y que origina el entorno descompuesto en celdas uniformes. Para ello, se procede a detallar los procesos enumerados en la *Figura 58*.

- Proceso 1. Es un proceso preliminar en el que se definen:
  - Las dimensiones del entorno para simplificar posteriores cálculos.
  - En función de la entrada introducida *subs_x* o *longitud_celda*, se recalcula su alternativa. En caso de haber introducido ambas entradas, se prioriza el parámetro *longitud_celda* para continuar el algoritmo
  - En función de la entrada introducida *subs_y* o *altura_celda*, se recalcula su alternativa. En caso de haber introducido ambas entradas, se prioriza el parámetro *altura_celda* para continuar el algoritmo
  - Se construye un arreglo cell (la salida *celdas_un*) que posee la descomposición del entorno en celdas uniformes, con las coordenadas que las identifican.
  - Se declara la variable de salida *matriz_un*, inicializada como una matriz de ceros.
  - Por último, se muestra la representación gráfica preliminar, en la que todas las celdas están 'vacías' (no poseen información aún).

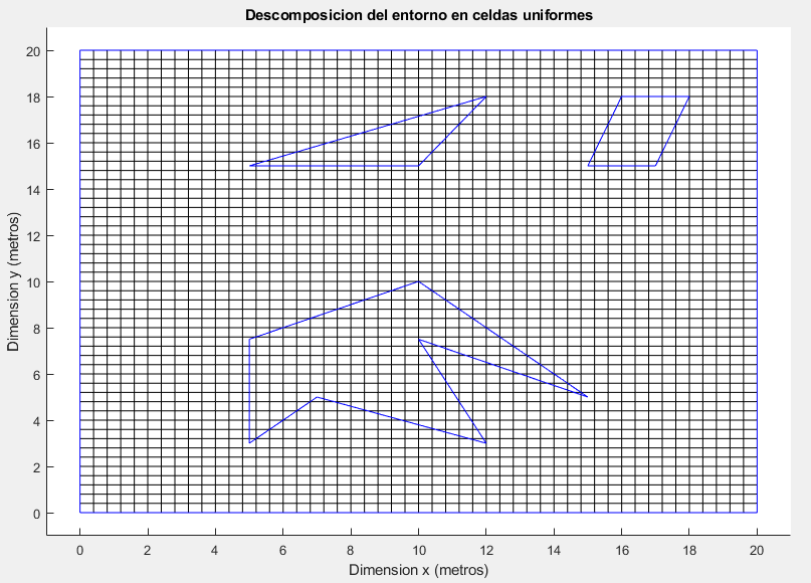

***Figura 59**. Ejemplo de representación gráfica del proceso 1 de la función celdas_uniformes.m.*

- Proceso 2. A partir del obstáculo en cuestión que se encuentre estudiando, extrae una Bounding Box que lo rodee. Una Bounding Box es el rectángulo que posee un polígono cuyos límites verticales se encuentran en las coordenadas mínima y máxima de la dimensión y, y sus límites horizontales en las coordenadas mínima y máxima de la dimensión x. La Bounding Box resultante se traslada a las celdas que ocuparía en la descomposición preliminar. Estas serán las celdas de estudio para el obstáculo en cuestión, lo que evita tener que revisar todas las celdas.

- Proceso 3. Recorriendo las celdas de la Bounding Box, se determina si la celda actual se encuentra ocupada parcialmente por algún obstáculo. Para ello se hace uso de la función *celda_ocupada_parcial.m*. Esta función determina si una celda se encuentra ocupada parcialmente si algún segmento de algún obstáculo del entorno produce alguna intersección en algún lado de la celda, o si un mismo segmento posee sus puntos inicial y final dentro de la propia celda.

  En caso de determinar afirmativamente que la celda se encuentra ocupada parcialmente, se actualiza la matriz de salida que codifica los estados de las celdas. El elemento cuyos índices coincidan con los de la celda de estudio pasa a valer -2.

- Proceso 4. Si la celda no está ocupada parcialmente, se procede a determinar si esta se encuentra ocupada íntegramente. Para ello se hace uso de la función *celda_ocupada_integra.m*. Esta función determina si una celda se encuentra ocupada íntegramente si ningún segmento de ningún obstáculo del entorno produce intersección alguna en algún lado de la celda, y si, además, todos los vértices de la celda se encuentran dentro del obstáculo.

  La última condición se comprueba mediante la función *punto_dentro_obstaculo.m*, que, a partir del punto de entrada y del obstáculo, traza un segmento aleatorio desde el punto hasta algún punto límite del entorno. Si este segmento nuevo produce intersecciones impares con los segmentos del obstáculo significará que el punto se localiza dentro del obstáculo.

  En caso de determinar afirmativamente que la celda se encuentra ocupada íntegramente, se actualiza la matriz de salida que codifica los estados de las celdas. El elemento cuyos índices coincidan con los de la celda de estudio pasa a valer -2.

Una vez finalizados todos los procesos, se representa gráficamente el resultado:

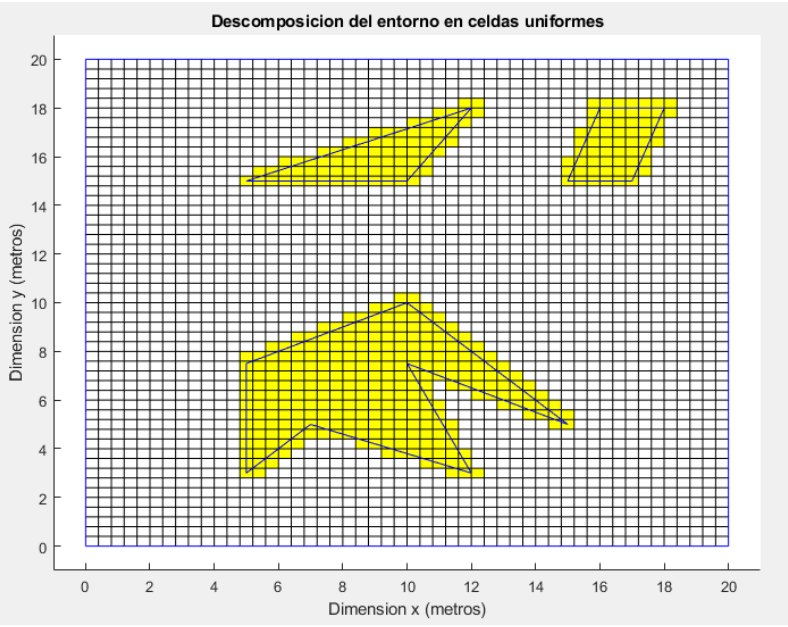

***Figura 60**. Representación gráfica de un entorno descompuesto en celdas uniformes.*

8.2.4. Cálculo de ruta mediante campo potencial

Al igual que el trabajo original expandía un campo potencial desde la celda inicial hasta la celda final para calcular la trayectoria, se ha decidido implementar la misma técnica desde una perspectiva propia. La función diseñada que realiza esta función es *ruta_celdas_uniformes_campo_potencial.m*, que requiere a su vez de la función auxiliar *celdas_uniformes_explorar_campo_potencial.m*. La sintaxis de la función principal:

$$[matriz\_campo,\ ruta,\ coste\_total] = ruta\_celdas\_uniformes\_campo\_potencial\ (celdas\_un,\ matriz\_un,\ pt\_in,\ pt\_fin,\ subs\_x,\ subs\_y,\ observar\_campo)$$

Donde:

- La salida *matriz_campo* es una matriz del tamaño de *matriz_un*, es, decir, de tantas filas por columnas como este dividido en celdas el entorno. Codifica internamente el coste incremental del campo, en tanto por 1 respecto a la dimensión máxima del entorno.
- La salida *ruta* retorna una matriz que posee los índices $(fila, columna)$ que identifican las celdas a recorrer para ir desde el punto inicial hasta el punto final. Al corresponder a la matriz en la que se divide el entorno, no se debe confundir estos índices como coordenadas (x, y).
- La salida *coste_total* es una medida del coste del recorrido. No se debe entender como la distancia exacta a recorrer, ya que se considera el mismo coste en el campo potencial tanto si la transición es vertical, horizontal o diagonal. Devuelve una medida aproximada que se debe analizar respecto a la dimensión máxima del entorno.
- La entrada *celdas_un* (salida de *celdas_uniformes.m*) es un arreglo tipo cell de dimensiones $subs\_y \times subs\_x$ compuesto por matrices que poseen las coordenadas de cada una de las celdas en las que se descompone el entorno.
- La entrada *matriz_un* (salida de *celdas_uniformes.m*) es una matriz de dimensiones $subs\_y \times subs\_x$, cuyos elementos son valores numéricos asociados a cada celda del arreglo *celdas_un* de sus mismos índices. Se utiliza 0 para designar celda libre y -2 para designar celda ocupada. El -1 se reserva para las celdas que posean los puntos inicial y final de la ruta.
- La entrada *pt_in* es un vector de tres coordenadas cartesianas que localiza el punto inicial de la ruta.
- La entrada *pt_fin* es un vector de tres coordenadas cartesianas que localiza el punto final de la ruta.
- La entrada *subs_x* (salida de *celdas_uniformes.m*) es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión x.
- La entrada *subs_y* (salida de *celdas_uniformes.m*) es un valor entero que contiene el número de subdivisiones en las que se divide el entorno en su dimensión y.
- La salida *observar_campo* es un indicador de 1 o 0 que puede introducir el usuario si deseo o no, respectivamente, observar mediante una animación cómo se expande el campo potencial.

La función se divide en tres partes secuenciales:

- Cálculos preliminares: se realiza una serie de cálculos para identificar sobre las celdas los puntos inicial y final, para así determinar si ambos se encuentran en la región libre y pueda existir una ruta que los conecte.
- Expansión del campo: desde el punto inicial, con paso (o iteración) de una celda y hasta el punto final, se expande un campo potencial que asocia un coste incremental en función del número de paso actual. El coste es entendido como:

$$coste = paso \cdot coste\_elemental\_de\_transicion$$

Donde el *paso* es el número actual de iteraciones, es decir, el número de transiciones desde la celda de origen hasta la actual. Y el *coste_elemental_de_transicion* es la inversa de la dimensión máxima del entorno. Por tanto, el coste puede ser entendido como una proporción, en tanto por 1, de la dimensión máxima del entorno. Una conclusión importante de este cálculo es que se entiende como el mismo coste de expansión ya sea vertical, horizontal o diagonal.

Cabe destacar, además, que el *coste_total* que se le ofrece al usuario no se encuentra escalado a tanto por 1, sino que es una medida aproximada de la distancia en función también de la dimensión máxima del entorno.

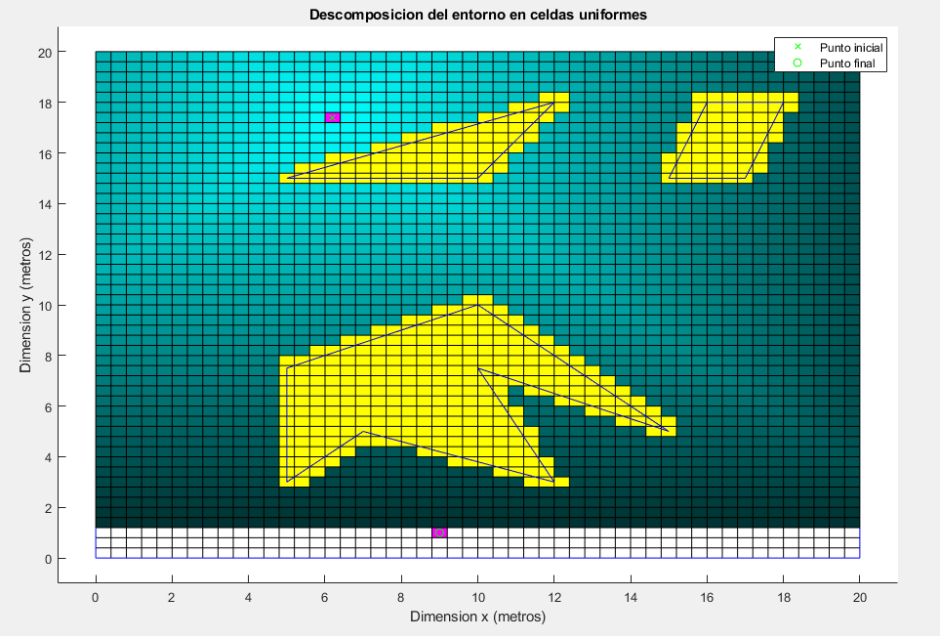

***Figura 61**. Representación gráfica de la expansión del campo potencial.*

- Exploración del campo potencial para obtener la ruta: este paso es llevado a cabo íntegramente por la función auxiliar *celdas_verticales_explorar_campo_potencial.m*. Se trata de una función recursiva compleja que explora el campo desde el punto final hasta el inicial (tras ello, invierta la ruta obtenida).

  La función analiza, a partir de la celda actual de estudio, cuáles de las celdas vecinas son las que poseen una orientación más propensa a acercar la ruta al nodo de origen. Ordena en función de la orientación a las celdas vecinas, se asegura que no se toma de nuevo la celda previa de la que viene y estudia si la siguiente celda más propensa es libre

y posee un coste decreciente respecto a la actual (ya que la ruta se crea desde la celda final hasta la inicial).

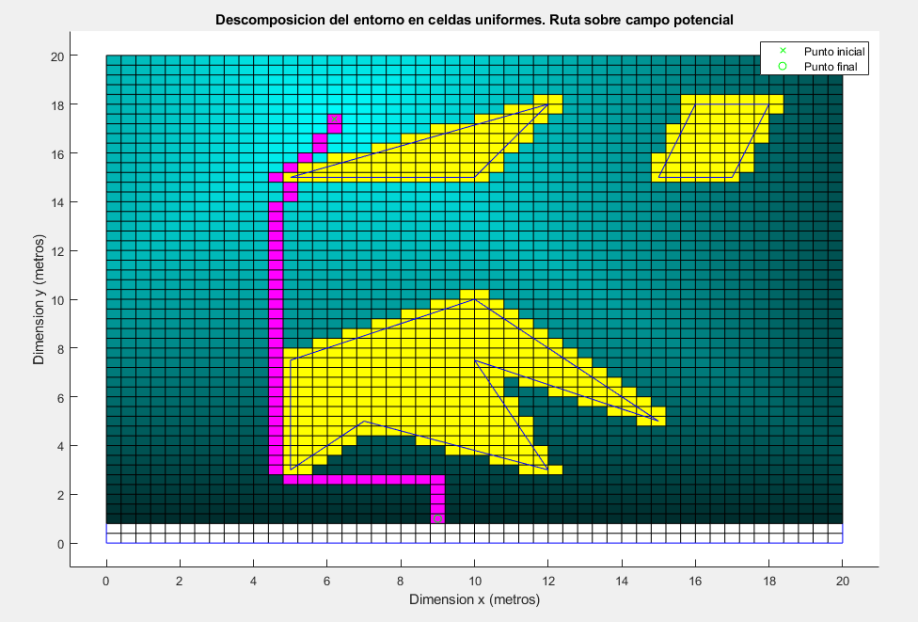

***Figura 62**. Representación gráfica de la ruta sobre el campo potencial.*

8.2.5. Limitaciones

El algoritmo desarrollado posee algunas limitaciones que impiden su flexibilidad total y deben ser consideradas para su correcta y eficiente implementación.

- El principal inconveniente de esta técnica es el compromiso entre resolución y coste computacional. Aunque se ha incorporado ciertos procedimientos que evitan tener que consultar todas y cada una de las celdas en las que se divide el entorno, las funciones auxiliares desarrolladas presentan un grado de complejidad y coste computacional relevante para lograr una gran precisión en los cálculos. Es por ello que resoluciones muy elevadas pueden comprometer la eficiencia del algoritmo, así como la posterior búsqueda de ruta mediante campo potencial.
- Durante la generación del campo potencial, este no distingue entre coste de transición entre celdas verticales, horizontales o diagonales, y considera una expansión 8-vecinos. Por lo que el coste no puede ser entendido exactamente como la distancia.

  Por otro lado, no se ha podido implementar una funcionalidad adicional característica de este tipo de algoritmos de campo potencial: que los obstáculos actúen como focos 'que repulsan' la ruta (aumentando el coste del campo en las regiones alrededor de ellos).
- Otro de los principales inconvenientes es el coste de exploración del campo potencial. Se ha desarrollado una función recursiva compleja que busca la ruta de coste óptimo teniendo en consideración muchas variables auxiliares para dirigir adecuadamente la ruta. Ello conlleva un coste computacional muy alto especialmente si el trazado debe evitar obstáculos de forma compleja.

- La última limitación es compuesta por las funciones auxiliares que son necesarias para el funcionamiento de la función principal *celdas_uniformes.m*. Distinguiendo entre las desarrolladas en este proyecto y ya existentes en la toolbox:

  - Funciones propias del Trabajo de Fin de Grado:
    - *celda_ocupada_integra.m*
    - *celda_ocupada_parcial.m*
    - *punto_dentro_obstaculo.m*
    - *reescribir_entorno.m*
    - *ruta_celdas_verticales_campo_potencial.m* (si se desea calcular rutas)
    - *celdas_verticales_explorar_campo_potencial.m* (si se desea calcular rutas)

  - Funciones previamente existentes en la toolbox:
    - *extrae_obstaculo.m*
    - *inter_seg.m*
    - *interseccion2d.m*
    - *precision.m*
    - *dibuent.m*

8.3. Comparativa entre el algoritmo original y el propio

Los dos algoritmos de celdas uniformes y cálculo de rutas mediante campo potencial son funcionales y presentan conclusiones útiles para la disciplina de la planificación de movimiento y para posteriores trabajos. No obstante, ninguno de los algoritmos es completamente flexible y cada uno destaca sobre el otro en algunas características.

Para realizar una comparativa adecuada, tomando en consideraciones las limitaciones oportunas, se va realizar un análisis desde diferentes perspectivas.

- Respecto a la introducción intuitiva de datos y funcionalidades ajustables por el usuario. Es uno de los principales inconvenientes del script original. A diferencia del nuevo diseño, el anterior no estaba completamente integrado en una función y realizaba cada paso por separado. Asimismo, los parámetros ajustables por el usuario eran limitados.
- Referido a la representación gráfica del proceso, el script original dividía los procesos en demasiadas representaciones que no aportan información relevante para un usuario que no pretende 'diseccionar' el programa, sino simplemente observar el resultado. Además, la discretización que se lleva a cabo sobre el entorno no se vuelve a tratar inversamente en su representación, por lo que las escalas en las que se representa el resultado aparentan arbitrariedad y no aportan información útil. Este aspecto es solventado por la nueva implementación.
- La delimitación de los obstáculos posee ventajas e inconvenientes en cada diseño. Como era mencionado en las limitaciones del script original, la delimitación de los

bordes de los obstáculos es aproximada, no estricta, lo que puede resultar en celdas parcialmente ocupadas que son marcadas como libres. En el nuevo script el análisis es más riguroso, lo que delimita con mayor precisión los obstáculos. No obstante, ello constituye también una desventaja para el nuevo diseño. Esta rigurosidad viene acompañada de un coste computacional más elevado, mientras que el diseño original es menos demandante de recursos de cómputo.

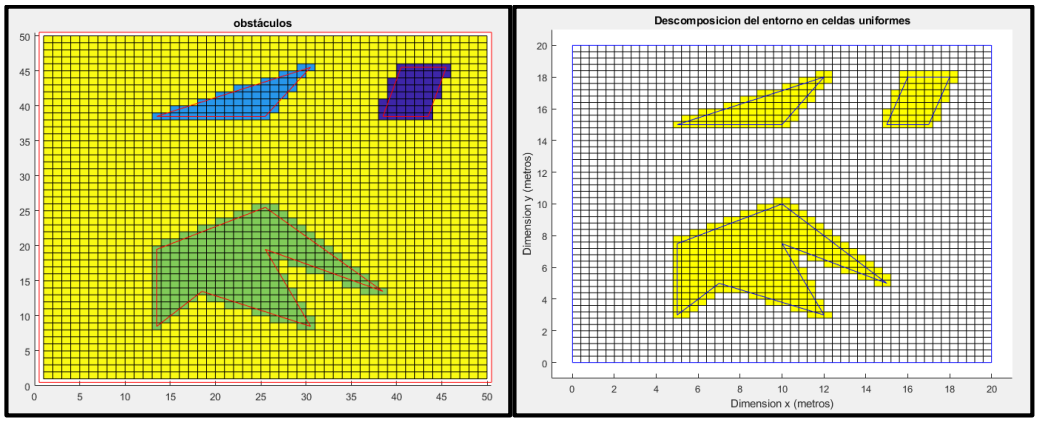

***Figura 63**. Comparativa de la representación gráfica de la delimitación de obstáculos.*
A la izquierda, algoritmo original. A la derecha, algoritmo propio.

- Respecto de la expansión del campo potencial, el script original realiza una serie de cálculos que aprovechan mejor las funcionalidades de Matlab (comando s*urface*) para la representación. A diferencia de ello, el diseño propio ha optado por añadir la funcionalidad de poder observar mediante animación la expansión del campo.
- Por último, referido al cálculo de ruta el script original presenta un mejor balance. Aunque ambos algoritmos son funcionales, el script original explora el campo en tiempos de cómputo mucho mejores que el nuevo desarrollo. Ello se debe a la limitación expuesta respecto al diseño propio: la exploración del campo es un procedimiento recursivo que busca optimizar la ruta tomando en consideración gran cantidad de variables (como determinar la celda vecina más adecuada en función de la dirección).

A continuación, se presenta una comparativa de los tiempos de cómputo (uso del comando *tic toc* de Matlab) para el mismo entorno, misma resolución y mismos puntos inicial y final:

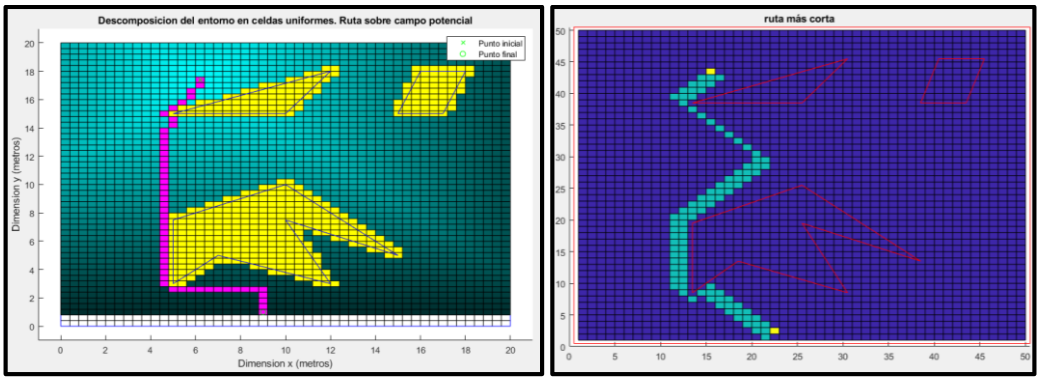

***Figura 64**. Comparativa de coste computacional bajo las mismas condiciones.*

Los tiempos de cómputo para cada una de las operaciones requeridas:

	Tiempo de cómputo (segundos)	
	Algoritmo original	Algoritmo propio
Delimitación de los obstáculos	1.1077	2.6425
Generación del campo potencial y cálculo de ruta	0.1687	22.8167

***Tabla 1**. Comparativa de coste computacional entre los algoritmos de descomposición en celdas uniformes.*

La conclusión que se puede extraer de estos resultados es que el nuevo desarrollo, con el fin de obtener una precisión mejor que su predecesor, ha incurrido en un coste computacional mucho más elevado.

9. Descomposición del entorno en Quadtree

Las técnicas basadas en descomponer el entorno en Quadtree son herramientas muy útiles en el estudio de entornos dentro de la geometría computacional, pues delimitan los obstáculos del entorno buscando la mayor eficiencia en las celdas en las que se divide el entorno. Estos métodos también son muy importantes en la planificación de movimiento, proporcionando una delimitación relativamente rápida (aunque siempre dependerá de la complejidad del entorno y resolución deseada) de las regiones libres y las ocupadas.

9.1. Preproceso: *quadtree_preparaciones.m*

La función desarrollada que realiza la descomposición del entorno en Quadtree es *quadtree.m*. Esta función posee naturaleza recursiva, es decir, se llama sí misma hasta que finaliza el proceso. Es por ello que, previo a llevar a cabo la descomposición del entorno en Quadtree, es requerido un preproceso de preparación de algunos argumentos de entrada de quadtree.m, a partir de otra función desarrollada: *quadtree_preparaciones.m*.

9.1.1. Sintaxis

$$[ent\_dimenx,\ ent\_dimeny,\ cell\_in,\ offsetx,\ offsety,\ sub\_max] = quadtree\_preparaciones(entorno,\ sub\_max,\ lado\_x\_celda,\ lado\_y\_celda)$$

Donde:

- La salida *ent_dimenx* es un valor real que posee la dimensión x del entorno.
- La salida *ent_dimeny* es un valor real que posee la dimensión y del entorno.
- La salida *cell_in* devuelve un arreglo tipo cell que codifica la celda inicial previo al comienzo de las subdivisiones. Su tamaño es de $2x4$.
- La salida *offsetx* es un real que indica la coordenada horizontal que sirve como sistema de referencia para dividir y ramificar en cada recursión.
- La salida *offsety* es un real que indica la coordenada vertical que sirve como sistema de referencia para dividir y ramificar en cada recursión.
- La salida *sub_max* es un entero que indica el nivel de ramificación máximo a alcanzar por el Quadtree.
- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos.
- La entrada *sub_max* es un valor entero que contiene el nivel de ramificación máximo a alcanzar por el Quadtree. Este valor de entrada deberá valer 0 si las entradas *lado_x_celda* y *lado_y_celda* poseen un valor no nulo.
- La entrada *lado_x_celda* es un valor real que posee la resolución en la dimensión x de la celda más ramificada del Quadtree. Se trata de un método aproximado, ya que la resolución no es ajustable en todo el rango de valores numéricos. Este valor de entrada deberá valer 0 si la entrada *sub_max* posee un valor no nulo.

- La entrada *lado_y_celda* es un valor real que posee la resolución en la dimensión y de la celda más ramificada del Quadtree. Se trata de un método aproximado, ya que la resolución no es ajustable en todo el rango de valores numéricos. Este valor de entrada deberá valer 0 si la entrada *sub_max* posee un valor no nulo.

9.1.2. Funcionamiento

El preproceso realiza cuatro operaciones básicas para preparar el Quadtree:

- Extrae del entorno sus dimensiones (*ent_dimenx* y *ent_dimeny*) y ajusta el sistema de referencia (*offsetx*, *offsety*) desde el que partirá la primera división y ramificación.
- Calcula el nivel de ramificación máxima en caso de que este no haya sido introducido directamente mediante *sub_max*, sino por introducción de los parámetros *lado_x_celda* y *lado_y_celda*. El nivel de ramificación máxima será aquel que permita tener tanta resolución o más que el más limitante de las resoluciones *lado_x_celda* y *lado_y_celda*. Se debe ser precavido con esta opción, ya que introducir resoluciones muy elevadas se traduce en niveles de ramificación muy altos, y estos niveles de ramificación poseen de una unidad a otra un salto en el coste computacional de escala exponencial con base 4.
- Se estructura la celda inicial (*cell_in*) como un arreglo tipo cell de tamaño $2x4$, que puede ser estudiada como dos cells cuadradas de $2x2$:

  - La primera cell cuadrada de $2x2$ estará constituida por matrices $2x7$ que codificará las celdas del actual nivel de ramificación. Los elementos de estas matrices:
    - Los elementos $[(1,1) \quad (1,2) \quad (1,3)]$ es un vector de tres coordenadas cartesianas que identifica el vértice inferior izquierdo de la celda.
    - Los elementos $[(1,4) \quad (1,5) \quad (1,6)]$ es un vector de tres coordenadas cartesianas que identifica el vértice superior izquierdo de la celda.
    - Los elementos $[(2,1) \quad (2,2) \quad (2,3)]$ es un vector de tres coordenadas cartesianas que identifica el vértice superior derecho de la celda.
    - Los elementos $[(2,4) \quad (2,5) \quad (2,6)]$ es un vector de tres coordenadas cartesianas que identifica el vértice inferior derecho de la celda.
    - El elemento $(1,7)$ será un valor entero que codificará el estado de la celda: 0 significará libre, 1 será parcialmente ocupada y 2 íntegramente ocupada.
    - El elemento $(2,7)$ será un valor entero que identificará el nivel de ramificación actual de la división. La celda inicial posee el nivel 0.
  - La segunda cell cuadrada de $2x2$ poseerá el resto de ramificaciones de la división actual del Quadtree.

    Siguiendo el esquema propuesto, una vez finalizado el Quadtree, este poseerá la siguiente estructura:

$$Quadtree = \{cell(2x2) \quad cell(2x2)\}$$

    Expandiendo:

$$Quadtree = \begin{Bmatrix} [celda\_inf\_izq] & [celda\_inf\_der] & \{ramif\_celda\_inf\_izq\} & \{ramif\_celda\_inf\_der\} \\ [celda\_sup\_izq] & [celda\_sup\_der] & \{ramif\_celda\_sup\_izq\} & \{ramif\_celda\_sup\_der\} \end{Bmatrix}$$

- Por último, se realiza una representación gráfica preliminar del entorno, generando la plantilla gráfica sobre la que se ramificará el Quadtree.

9.2. Implementación *quadtree.m*

Una vez se poseen los parámetros de salida de la función *quadtre_preparaciones.m*, se puede llamar a la función principal *quadtree.m* para dividir el entorno.

El código del programa (y funciones auxiliares) se encuentra como anexo (véase *Índice de Anexos*).

9.2.1. Sintaxis

$arbol = quadtree\ (entorno,\ arbol,\ ent\_dimenx,\ ent\_dimeny,\ offsetx,\ offsety,\ sub\_max,\ observar\_quadtree)$

Donde:

- La salida *arbol* retorna un arreglo tipo cell con toda la información del Quadtree, siguiendo un esquema tal que el presentado en la sección '*9.1. Preproceso: quadtree_preparaciones.m*', página 81.
- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos.
- La entrada *ent_dimenx* es un valor real que posee la dimensión x del entorno. El valor cambia en cada recursión para calcular las nuevas coordenadas de cada celda.
- La entrada *ent_dimeny* es un valor real que posee la dimensión y del entorno. El valor cambia en cada recursión para calcular las nuevas coordenadas de cada celda.
- La entrada *offsetx* es un real que indica la coordenada horizontal que sirve como sistema de referencia para dividir y ramificar en cada recursión.
- La entrada *offsety* es un real que indica la coordenada vertical que sirve como sistema de referencia para dividir y ramificar en cada recursión.
- La entrada *sub_max* es un valor entero que contiene el nivel de ramificación máximo a alcanzar por el Quadtree. Independientemente del método de resolución escogido, la salida *sub_max* de la función *quadtree_preparaciones.m* es la entrada *sub_max* de *quadtree.m*.
- La entrada *observar_quadtree* es un valor entero que puede valer 0 o 1. Se trata de un indicador que con un 1 informa al programa que se desea observar la animación de descomposición del entorno en Quadtree. En caso de introducir 0, no se observará la animación.

9.2.2. Diagrama de flujo

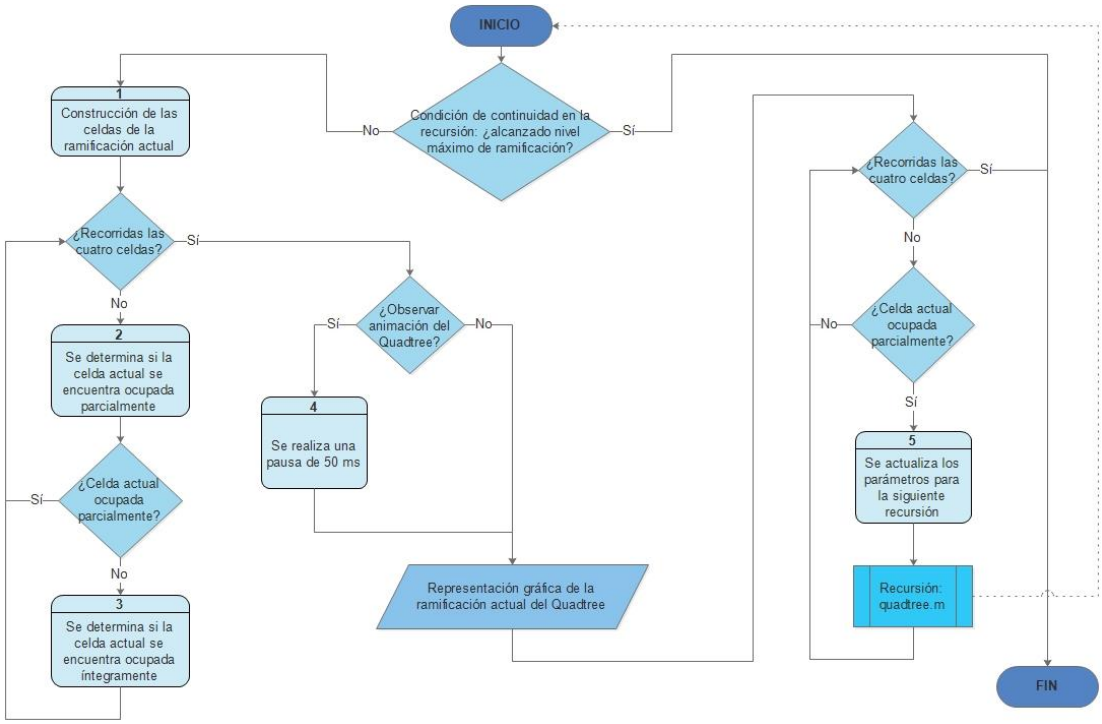

***Figura 65**. Diagrama de flujo de la función quadtree.m.*

9.2.3. Lógica del código

A continuación, se expone el razonamiento lógico con el que trabaja el programa y que origina el entorno ramificado en Quadtree. Para ello, se procede a detallar los procesos enumerados en la *Figura 65*.

- Proceso 1. Es un proceso que construye las celdas de la ramificación actual calculando las coordenadas sus vértices, a partir de los datos de entrada de *ent_dimenx*, *ent_dimeny*, *offsetx* y *offsety*.

- Proceso 2. Durante este proceso se determina si la celda actual se encuentra ocupada parcialmente por algún obstáculo. Para ello se hace uso de la función *celda_ocupada_parcial.m*. Esta función determina si una celda se encuentra ocupada parcialmente si algún segmento de algún obstáculo del entorno produce alguna intersección en algún lado de la celda, o si un mismo segmento posee sus puntos inicial y final dentro de la propia celda.

  En caso de determinar afirmativamente que la celda se encuentra ocupada parcialmente, se actualiza el elemento $(1,7)$ de la matriz de coordenadas de la celda, el cual codifica el estado de la misma. Se introduce un 1 para codificar celda parcialmente ocupada.

- Proceso 3. Si la celda no está ocupada parcialmente, se procede a analizar si lo está íntegramente. Para ello se hace uso de la función *celda_ocupada_integra.m*. Esta

función determina si una celda se encuentra ocupada íntegramente si ningún segmento de ningún obstáculo del entorno produce intersección alguna en algún lado de la celda, y si, además, todos los vértices de la celda se encuentran dentro del obstáculo.

La última condición se comprueba mediante la función *punto_dentro_obstaculo.m*, que, a partir del punto de entrada y del obstáculo, traza un segmento aleatorio desde el punto hasta algún punto límite del entorno. Si este segmento nuevo produce intersecciones impares con los segmentos del obstáculo significará que el punto se localiza dentro del obstáculo.

En caso de determinar afirmativamente que la celda se encuentra ocupada íntegramente, se actualiza el elemento $(1,7)$ de la matriz de coordenadas de la celda, el cual codifica el estado de la misma. Se introduce un 2 para codificar celda íntegramente ocupada.

- Proceso 4. Este proceso únicamente consiste en una espera de 50 milisegundos que se produce cuando la entrada *observar_entorno* vale 1. Durante la ejecución del Quadtree ello permite observar como si de una animación se tratase cómo se descompone el entorno en celdas.

- Proceso 5. Si la celda actual se ha determinado como una celda parcialmente ocupada, ello significa que puede descomponerse con una resolución mayor. A partir de las coordenadas de la celda en cuestión se actualizan los parámetros *ent_dimenx*, *ent_dimeny*, *offsetx* y *offsety*. Con ellos se realiza una nueva recursión sobre la celda actual parcialmente ocupada. Es necesario además actualizar los elementos $(2,7)$ de las matrices de coordenadas de la siguiente iteración, sumando uno al nivel de ramificación presente.

Una vez finalizados todos los procesos, se representa gráficamente el resultado:

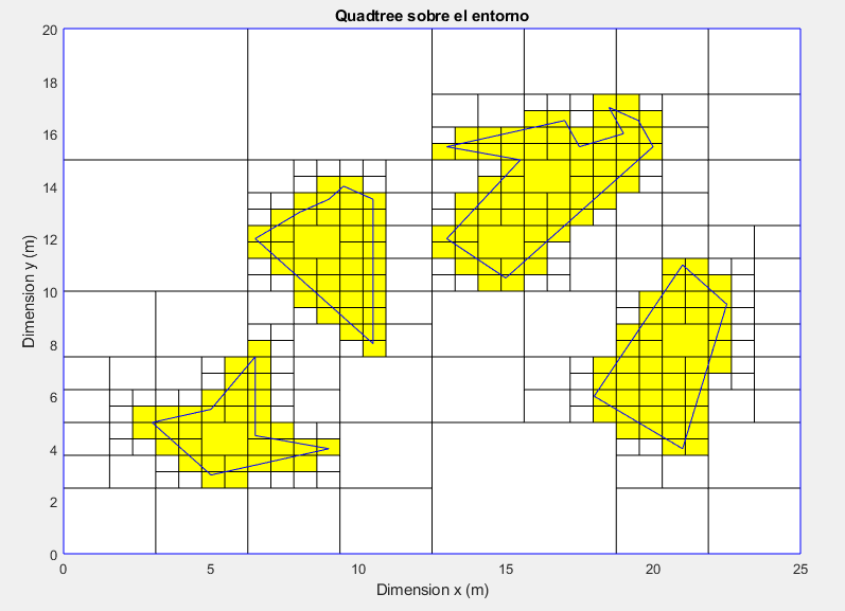

***Figura 66**. Representación gráfica de la descomposición de un entorno en Quadtree.*

9.2.4. Generación de trayectorias mediante el uso del algoritmo de Dijkstra

Al igual que sucedía con la descomposición en celdas verticales, los entornos descompuestos en Quadtree son susceptibles de aplicar algoritmos de exploración de grafos, a través de la extracción de nodos de las celdas libres. Del mismo modo que las celdas verticales en configuración trapezoidal, las celdas del Quadtree aseguran su conectividad cuando son colindantes. No obstante, debido a la naturaleza de la descomposición, las celdas resultantes son irregulares en tamaño, lo que deriva en un diseño más exigente. Para la extracción de los nodos y el cálculo del coste de los arcos asociados ha sido necesaria la implementación de diversas funciones que permiten compatibilizar los algoritmos.

El algoritmo de búsqueda en grafos seleccionado es el algoritmo de Dijkstra, cuyo funcionamiento ha sido expuesto en la sección '*2.3.2. Configuración del espacio de trabajo y algoritmos de cálculo de rutas*' (pág. 19, véase *Índice General*). Lo primero es recordar la sintaxis de la función *Dijkstra.m* (presente en la toolbox):

$$[Camino,\ Coste] = Dijkstra\ (matriz\_costes,\ nodo\_inicial,\ nodo\_final)$$

El algoritmo de Dijkstra de la toolbox trabaja conjuntamente con la función *dibujar_camino.m* para trazar gráficamente el *Camino* salida de la función *Dijkstra.m*. La sintaxis de esta función es la siguiente:

$$Trayecto = dibujar\_camino\ (Camino,\ vertices,\ Color)$$

Los parámetros de ambas funciones son detallados en la sección '*7.3. Algoritmo de Dijkstra en entornos descompuestos en celdas verticales*' (pág. 60, véase *Índice General*).

Para compatibilizar los resultados de la función *quadtree.m* con las entradas de Dijkstra se ha desarrollado la función *quadtree_preparaciones_para_ruta.m*. La sintaxis de esta nueva función:

$$[pts\_en\_area\_libre,\ nodos,\ matriz\_costes] = quadtree\_preapraciones\_para\_ruta\ (arbol,\ sub\_max,\ pt\_in,\ pt\_fin)$$

Donde:

- La salida *pts_en_area_libre* es un valor entero que valdrá 1 cuando ambos puntos, inicial y final, se encuentren en celdas libres. Su valor será 0 en caso contrario.
- La salida *nodos* retorna una matriz que posee los nodos de la red con los puntos inicial y final añadidos en los índices 1 y 2, respectivamente.
- La salida *matriz_costes* es una matriz de nxn donde n es el número total de nodos, incluyendo el nodo inicial y final cuyos índices son 1 y 2, respectivamente. Cada elemento (i, j) poseerá el coste asociado al arco que une el nodo i con el nodo j. Además, los arcos son bidireccionales, por lo que el coste (i, j) es igual al coste (j, i).
- La entrada *arbol* (salida de *quadtree.m*) es un arreglo tipo cell con toda la información del Quadtree, siguiendo un esquema tal que el presentado en la sección …, página ….
- La salida *sub_max* es un valor entero que contiene el nivel de ramificación máximo a alcanzar por el Quadtree. Independientemente del método de resolución escogido, la salida *sub_max* de la función *quadtree_preparaciones.m* es la entrada *sub_max* de *quadtree_preparaciones_para_ruta.m* (también para *quadtree.m)*.

- La entrada *pt_in* es un vector de tres coordenadas cartesianas que localiza el punto inicial de la ruta.
- La entrada *pt_fin* es un vector de tres coordenadas cartesianas que localiza el punto final de la ruta.

Los procedimientos llevados a cabo por la función se pueden dividir en tres partes secuenciales:

- Cálculos preliminares: se realiza una serie de cálculos para identificar sobre las celdas los puntos inicial y final, para así determinar si ambos se encuentran en la región libre y pueda existir una ruta que los conecte. Para localizar los puntos se ha desarrollado una función auxiliar recursiva: *quadtree_localizar_punto.m*.

  La función *quadtree_localizar_punto.m*, a partir del punto y del propio Quadtree, explora recursivamente el Quadtree hasta localizar la ramificación final en la que se encuentra el punto. Devuelve dos salidas: un indicador de 0 y 1 si el punto se encuentra en la región ocupada o libre, respectivamente; y un vector que representa el itinerario de las ramificaciones a seguir para localizar al punto en el diagrama de árbol. Cada elemento del vector es un número del 1 al 4 que identifica:

  - 1 si el punto se encuentra en la celda inferior izquierda.
  - 2 si el punto se encuentra en la celda inferior derecha.
  - 3 si el punto se encuentra en la celda superior izquierda.
  - 4 si el punto se encuentra en la celda superior derecha.

- Extracción de los nodos: previo a construir la matriz de costes para el argumento de la función de *Dijkstra.m,* es requerido la extracción de los nodos de la red, que se han supuesto como los puntos centro de las celdas libres. Se ha implementado una función auxiliar recursiva que se encarga del proceso: *quadtree_extraer_nodos.m*.

  La función se encarga de explorar todo el Quadtree hasta llegar a cada ramificación terminal. Una vez se encuentra en ella, verifica que esta corresponda a una celda libre y, de ser así, actualiza la matriz de nodos que conformará la red.

- Construcción de la matriz de costes. A su vez, distingue tres subprocesos:

  - Cálculo de coste entre los puntos inicial y final. Para ello, se comprueba si ambos poseen el mismo itinerario (función *quadtree_localizar_punto.m*), lo que querría decir que ambos se encontrarían en la misma celda. De ser así, el coste será la distancia que los separa. En caso contrario, el coste será infinito.
  - Cálculo de coste entre los nodos inicial y final y el resto de nodos. Se considera que los puntos inicial y final solo serán accesibles por los propios nodos de la celda en la que se encuentran, siendo su coste la distancia entre puntos. El resto de costes será infinito.
  - Cálculo de costes entre nodos. Este subproceso se realiza recorriendo todos los nodos de la red y comprobando, respecto a cada uno del resto de nodos, si se localizan en celdas contiguas, tanto horizontales como verticales y diagonales (conectividad 8-vecinos). Este paso es realizado por la función auxiliar *quadtree_celdas_contiguas.m*.

Esta función posee un diseño más complejo que requiere a su vez de tres subfunciones: *quadtree_localizar_punto.m*, *quadtree_extraer_celda.m* y *punto_pertenece_segmento.m*.

La primera ya ha sido detallada anteriormente. La segunda es una función paralela a la primera, recibe el itinerario de *quadtree_localizar_punto.m* y devuelve la información de la celda que codifica tal itinerario. La tercera es una función más simple que devuelve un 1 si el punto introducido pertenece al segmento de entrada.

Con estas tres funciones, *quadtree_celdas_contiguas.m* busca, para cada nodo, la celda en la que se encuentra (*quadtree_localizar_punto.m* y *quadtree_extraer_celda.m*) y la compara con el resto de celdas. Aquellas con las que comparta al menos un punto de sus lados o vértices (*punto_pertenece_segmento.m*) será una celda contigua.

De esta forma, para aquellas celdas colindantes, el coste asociado a sus nodos será la distancia. En caso contrario el coste será infinito.

El resultado de aplicar estas funciones y el algoritmo de Dijkstra será:

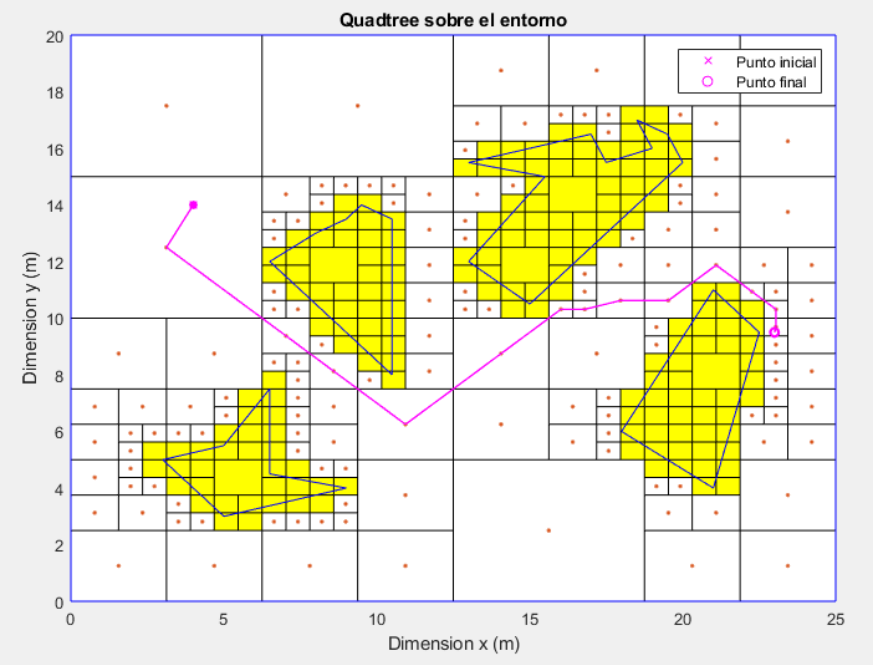

***Figura 67**. Representación gráfica de la aplicación de Dijkstra sobre una descomposición en Quadtree.*

9.2.5. Limitaciones

El algoritmo desarrollado se caracteriza por algunas limitaciones e inconvenientes en su aplicación.

- Uno de los inconvenientes de esta técnica es el coste computacional asociado a cada aumento del nivel de ramificación deseado. Debido a que una celda, al proceder a su subdivisión, origina cuatro subceldas; el coste computacional sigue una evolución

exponencial de base 4 respecto a la resolución. Es por ello que, valores superiores a 6 para el nivel de ramificación máximo, comienzan a requerir un tiempo de cómputo excesivo para una planificación eficiente.

- Otro inconveniente es la carencia de uniformidad entre las celdas que origina (que es el efecto directo de su principal ventaja: el aumento de la eficiencia al descomponer el espacio en celdas que no requieren divisiones más pequeñas para delimitar correctamente las regiones libres de las ocupadas), lo cual se encuentra ligado a posteriores métodos de cálculo de trayectorias sobre el Quadtree.

  La aplicación de campos potenciales se vuelve ineficiente (ya que requeriría discretizar con mayor resolución las celdas más grandes) y su diseño altamente complicado. Ello limita las opciones, entre las que se puede destacar algoritmos probabilísticos o algoritmos basados en grafos.

  Para implementar estos algoritmos es preciso extraer del Quadtree los nodos accesibles y los costes asociados a ellos. Esta tarea puede ser de alta complejidad de diseño, ya que requiere explorar el Quadtree y ser capaz de determinar celdas contiguas y nodos cuyos arcos sean posibles. Por otro lado, vuelve a tomar importancia el primer inconveniente, una resolución elevada originará una cantidad enorme de nodos a explorar que, además, son ineficientes para el cálculo de rutas (ya que estos se situarían delimitando casi al extremo los bordes de los obstáculos, lo que no ofrece mayor beneficio frente al coste asociado).

- La última limitación es compuesta por las funciones auxiliares que son necesarias para el funcionamiento de la función principal *quadtree.m*. Distinguiendo entre las desarrolladas en este proyecto y ya existentes en la toolbox:

  - Funciones propias del Trabajo de Fin de Grado:
    - *quadtree_preparaciones.m*
    - *celda_ocupada_integra.m*
    - *celda_ocupada_parcial.m*
    - *punto_dentro_obstaculo.m*
    - *quadtree_preparaciones_para_ruta.m* (si se desea calcular rutas)
    - *quadtree_localizar_punto.m* (si se desea calcular rutas)
    - *quadtree_extraer_nodos.m* (si se desea calcular rutas)
    - *quadtree_extraer_celda.m* (si se desea calcular rutas)
    - *quadtree_celdas_contiguas.m* (si se desea calcular rutas)
    - *punto_pertenece_segmento.m* (si se desea calcular rutas)

  - Funciones previamente existentes en la toolbox:
    - *extrae_obstaculo.m*
    - *inter_seg.m*
    - *interseccion2d.m*
    - *precision.m*
    - *dibuent.m*

- *Dijkstra.m* (si se desea calcular rutas)
- *Dibujar_camino.m* (si se desea calcular rutas)

10. Diagrama de Voronoi

El diagrama de Voronoi es la última de las herramientas desarrolladas en este Trabajo de Fin de Grado. Como sucedía con la expansión del entorno por suma de Minkowski, no constituye en sí mismo un método de planificación de movimiento y tampoco es una técnica basada en descomposición de celdas (aunque aprovechando el recorrido de este proyecto se ha implementado a partir del Quadtree). Sin embargo, es una herramienta que ofrece mucha información del entorno que se analiza, lo que constituye un buen procedimiento para obtener conclusiones previas a implementar algún algoritmo de planificación de movimiento.

10.1. Introducción

El diagrama de Voronoi es una representación gráfica sobre el plano euclídeo que consiste en el trazado de aquellos puntos del espacio libre cuya distancia a sus dos obstáculos más próximos sea la misma. Esta es una definición adecuada para la planificación de movimientos. No obstante, el diagrama de Voronoi no tiene su origen en esta disciplina.

El primer matemático que definió estos diagramas fue el matemático ucraniano Gueorgui Feodósievich Voronói, a quien debe su nombre. Originalmente el diagrama surge como la subdivisión del plano euclídeo en polígonos a partir de puntos (en vez de obstáculos), de manera que el perímetro de cada polígono resultante es equidistante de sus dos vecinos más próximos. Se construye trazando segmentos que unen los puntos con sus vecinos. De estos segmentos se trazan las mediatrices. Los polígonos resultantes serán los delimitados por las intersecciones de las mediatrices.

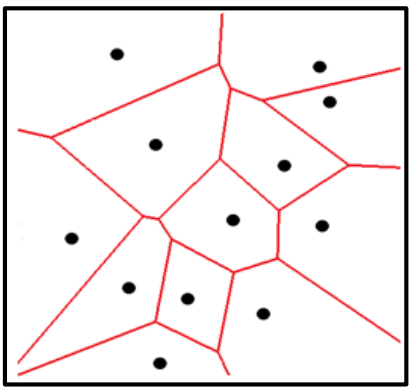

***Figura 68**. Representación gráfica de un diagrama de Voronoi basado en puntos.*

El diagrama de Voronoi es una herramienta gráfica que aporta mucha información. Para los diagramas basados en puntos, el diagrama informa del área de influencia de cada punto respecto a sus vecinos. Ello tiene utilidades en gran variedad de ámbitos de estudio, sobre todo enfocados con la distribución (ie: en un mapa de una ciudad, si se desea establecer un nuevo negocio, el diagrama de Voronoi puede ofrecer información muy valiosa del área de influencia de cada negocio de la competencia).

Volviendo al contexto de este trabajo, el diagrama de Voronoi trazado sobre obstáculos ofrece una conclusión diferente: el trazado originado informará de las trayectorias entre

obstáculos vecinos más seguras posibles para el robot, ya que estas se situarán siempre a la misma distancia de sus dos obstáculos más cercanos.

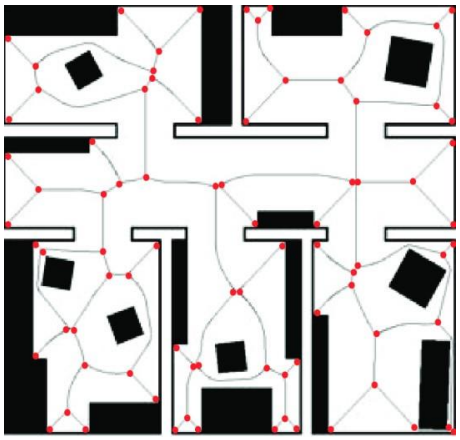

***Figura 69**. Representación gráfica de un diagrama de Voronoi basado en obstáculos, en un entorno que simula una casa.*

Como se puede observar en la *Figura 69*, el diagrama de Voronoi expone la trayectoria más segura para el robot, pero no la más eficiente en términos de distancia. Mediante el uso de algoritmos auxiliares de planificación de movimiento se puede analizar el compromiso entre seguridad y eficiencia.

10.2. Implementación

La función desarrollada que realiza el diagrama de Voronoi sobre el entorno es *diagrama_voronoi.m*. A continuación, se expone su metodología y funcionamiento.

El código del programa (y funciones auxiliares) se encuentra como anexo (véase *Índice de Anexos*).

10.2.1. Sintaxis

$$cell\_voronoi = diagrama\_voronoi\,(entorno,\ precision)$$

Donde:

- La salida *cell_voronoi* es un arreglo tipo cell que contiene todos los puntos pertenecientes al diagrama de Voronoi situándolos en los elementos cuyos índices coinciden con la discretización del entorno llevada a cabo internamente.
- La entrada *entorno* aloja la información del entorno original que se pretende descomponer en celdas, como una matriz de segmentos.
- La entrada *precision* es un parámetro entero, del 1 al 5, que sirve como escala al programa para determinar la cantidad puntos con los que trazar el diagrama de Voronoi. A mayor precisión, mayor resolución y mayor coste computacional y viceversa.

10.2.2. Diagrama de flujo

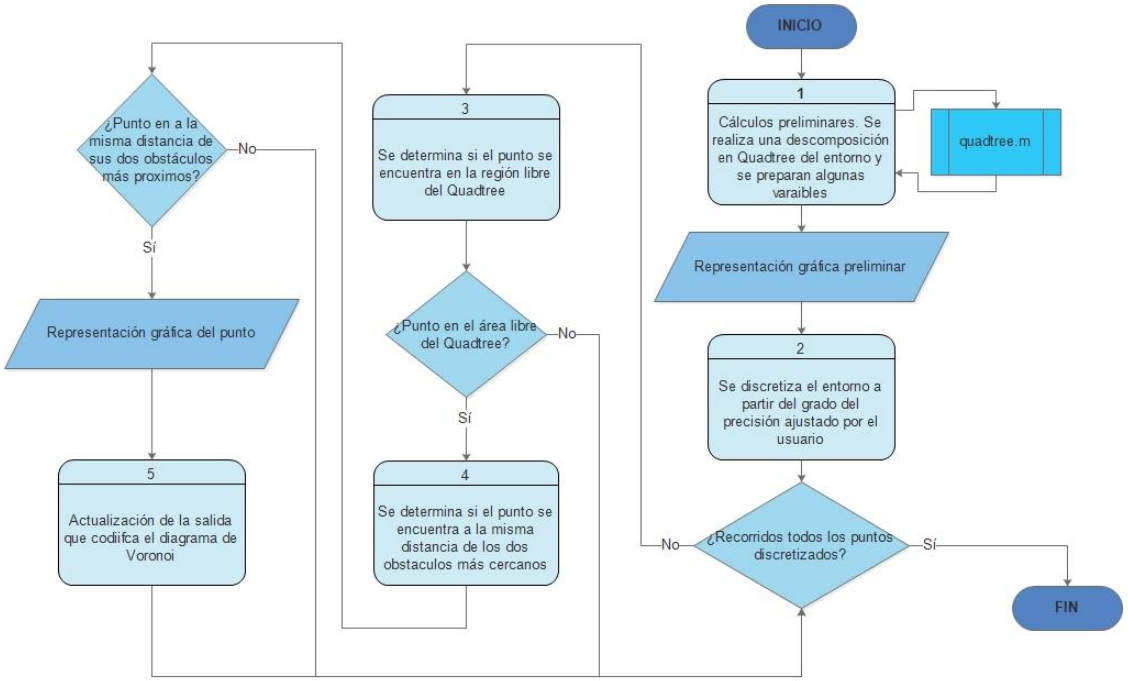

***Figura 70**. Diagrama de flujo de la función diagrama_voronoi.m.*

10.2.3. Lógica del código

A continuación, se expone el razonamiento lógico con el que trabaja el programa y que origina el diagrama de Voronoi. Para ello, se procede a detallar los procesos enumerados en la *Figura 70*.

- Proceso 1. Es un proceso preliminar que lleva cabo diversas tareas:
  - Lleva a cabo una descomposición en Quadtree del entorno (función *quadtree.m*). Se utiliza posteriormente para distinguir entre regiones ocupadas y libres.
  - Se calcula e introduce en el entorno los ángulos que forman los segmentos que lo componen con la vertical negativa (función *reescribir_entorno.m*).
  - A partir de la escala de precisión ajustada por el usuario, se calcula el paso de discretización con el que se discretizará el entorno en el proceso 2.
  - Se declara dos arreglos tipo cell: el primero poseerá todo el entorno discretizado en el que los elementos son vectores de tres coordenadas que identifican cada punto, y el segundo será la salida *cell_voronoi*, que poseerá únicamente los puntos discretizados que pertenecen al trazado del diagrama de Voronoi.

- Proceso 2. El segundo proceso se encarga de construir el entorno discretizado a partir del paso de discretización calculado preliminarmente.

- Proceso 3. Para cada punto del entorno discretizado, se explora el Quadtree para determinar si el punto se encuentra en la región libre o en la ocupada. De este proceso se encarga la función auxiliar *quadtree_localizar_punto.m*.

  La función *quadtree_localizar_punto.m*, a partir del punto y del propio Quadtree, explora recursivamente el Quadtree hasta localizar la ramificación final en la que se encuentra el punto. Devuelve dos salidas: un indicador de 0 y 1 si el punto se encuentra en la región ocupada o libre, respectivamente; y un vector que representa el itinerario de las ramificaciones a seguir para localizar al punto en el diagrama de árbol. Cada elemento del vector es un número del 1 al 4 que identifica:

  - 1 si el punto se encuentra en la celda inferior izquierda.
  - 2 si el punto se encuentra en la celda inferior derecha.
  - 3 si el punto se encuentra en la celda superior izquierda.
  - 4 si el punto se encuentra en la celda superior derecha.

- Proceso 4. Si el punto discretizado se encuentra en el área libre, pasa a comprobarse si este, además, se localiza a la misma distancia (con cierta tolerancia) de sus dos obstáculos más próximos. De este punto se ocupa la función auxiliar *punto_pertenece_voronoi.m*.

  Esta función recibe como parámetros el punto y el entorno. A partir del punto, encuentra los dos obstáculos (o límites del entorno) más cercanos al punto, calcula la mínima distancia entre el punto y cada obstáculo y las compara. Si ambas distancias son iguales (tomando en consideración cierta tolerancia para considerar iguales), se devuelve un 1 que indica que el punto efectivamente pertenece al diagrama de Voronoi.

- Proceso 5. El último proceso es representar el punto si este pertenece al trazado del diagrama de Voronoi y actualizar la salida *cell_voronoi* que aloja todos los puntos.

Una vez finalizados todos los procesos, se representa gráficamente el resultado:

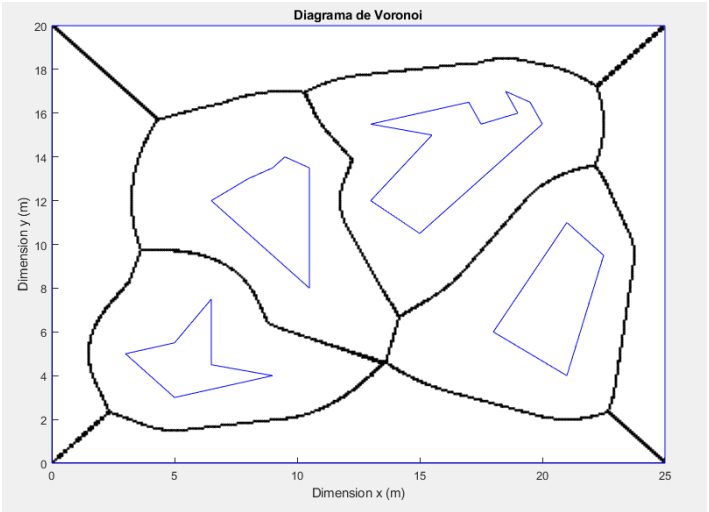

***Figura 71**. Representación gráfica del diagrama de Voronoi trazado sobre un entorno.*

10.2.4. Limitaciones

El algoritmo desarrollado posee ciertas limitaciones que impiden su total versatilidad. Los principales inconvenientes se relacionan con la necesidad preliminar del Quadtree y el compromiso coste-eficiencia del parámetro de entrada *precision*.

- Como se adelantaba, una de las principales limitaciones del diagrama de Voronoi implementado es que requiere una distinción previa entre el área ocupada por obstáculos y el área libre. Para este cometido se ha escogido uno de los propios métodos de celdas implementados, el que destaca por mayor eficiencia: el Quadtree. No obstante, este requerimiento preliminar añade al proceso un coste computacional adicional.
- Otra de las limitaciones que posee el programa a causa del diseño efectuado son las consecuencias de la discretización requerida. Puesto que los obstáculos del entorno no son puntos en el espacio, la técnica original de extraer segmentos y mediatrices para trazar los polígonos del diagrama de Voronoi no es aplicable. En su caso se ha optado por comprobar por fuerza bruta (con previa discretización) los puntos del espacio, y determinar para cada uno si pertenece al diagrama de Voronoi.

  Es en este punto donde toma relevancia el parámetro de entrada *precision*, pues se presenta como una escala de resolución del usuario para que este determine la cantidad de puntos del entorno discretizado. Valores de *precision* elevados dan lugar a mayor cantidad de puntos a analizar, lo que se traduce en mayor resolución del diagrama, pero también mayor coste computacional.
- La última limitación es compuesta por las funciones auxiliares que son necesarias para el funcionamiento de la función principal *diagrama_voronoi.m*. Distinguiendo entre las desarrolladas en este proyecto y ya existentes en la toolbox:

  - Funciones propias del Trabajo de Fin de Grado:
    - *reescribir_entorno.m*
    - *punto_pertenece_voronoi.m*
    - *quadtree.m*
    - *quadtree_preparaciones.m*
    - *celda_ocupada_parcial.m*
    - *celda_ocupada_integra.m*
    - *punto_dentro_obstaculo.m*
    - *quadtree_localizar_punto.m*
  - Funciones previamente existentes en la toolbox:
    - *extrae_obstaculo.m*
    - *inter_seg.m*
    - *interseccion2d.m*
    - *precision.m*
    - *intersección_normal.m*
    - *dibuent.m*

11. Análisis comparativo de las técnicas empleadas. Conclusiones y trabajos futuros.

En esta sección se realizará una simulación conjunta en la que estén implicados todos los algoritmos descritos en esta Memoria. Extrayendo, además, los tiempos de cómputo entre cada proceso, se procederá a hacer un análisis comparativo de los resultados obtenidos de todas las técnicas de planificación de movimiento implementadas, que permitirá inferir en las conclusiones del proyecto. Por último, se destacará aquellos aspectos más susceptibles de ser incluidos en trabajos futuros.

11.1. Simulación conjunta

Para que la simulación arroje información útil para contrastar, las condiciones iniciales para todos los algoritmos serán lo más próximas posibles. El entorno y los puntos inicial y final sobre el que se llevará a cabo la simulación son los siguientes:

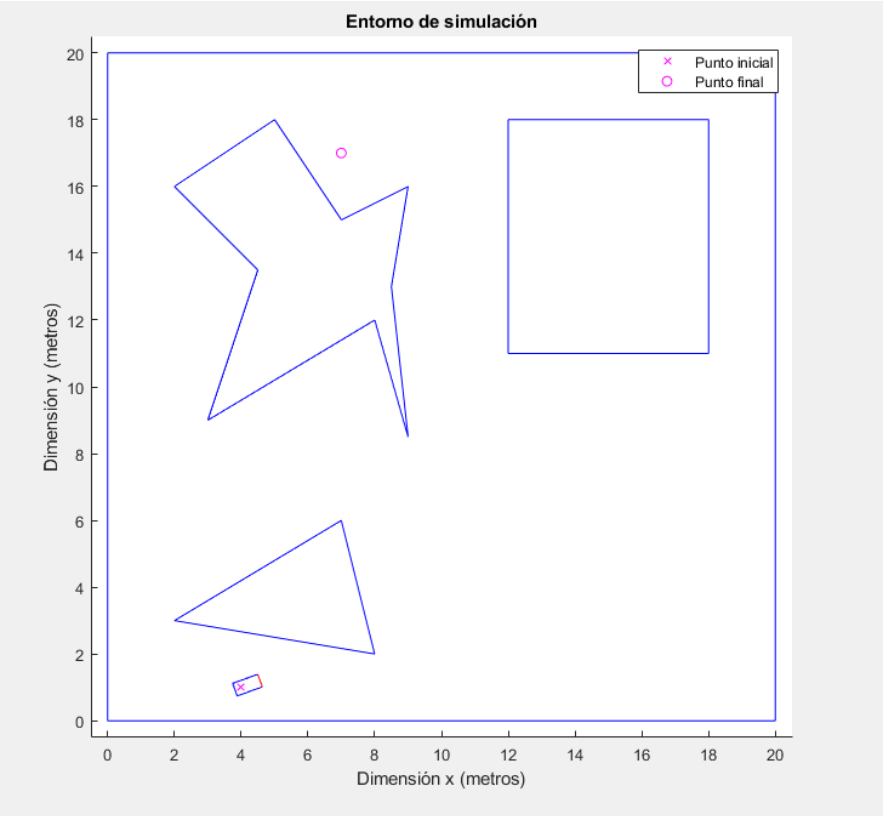

***Figura 72**. Representación gráfica de las condiciones iniciales del entorno para la simulación conjunta.*

- Ruta:
  - Punto inicial: $[4 \quad 1 \quad 0]\, m$
  - Punto final: $[7 \quad 17 \quad 0]\, m$

➢ Características del robot:

- Longitud: $0.7\ m$.
- Anchura: $0.4\ m$.
- Orientación (respecto a la horizontal): $45°$.
- Sistema de referencia móvil (considerando que el robot reposa sobre los ejes globales con orientación de 0 grados): $[0.2\ 0.2\ 0]\ m$.
- Posición del sistema de referencia móvil respecto al sistema de referencia global: $[4\ 1\ 0]\ m$ (posición inicial).

El orden secuencial de simulación que se va lleva a cabo es el siguiente:

1) Expansión del entorno mediante suma de Minkowski. Se toma una tolerancia adicional de expansión sobre la dimensión máxima del robot (longitud: $0.7\ m$) del 5%.

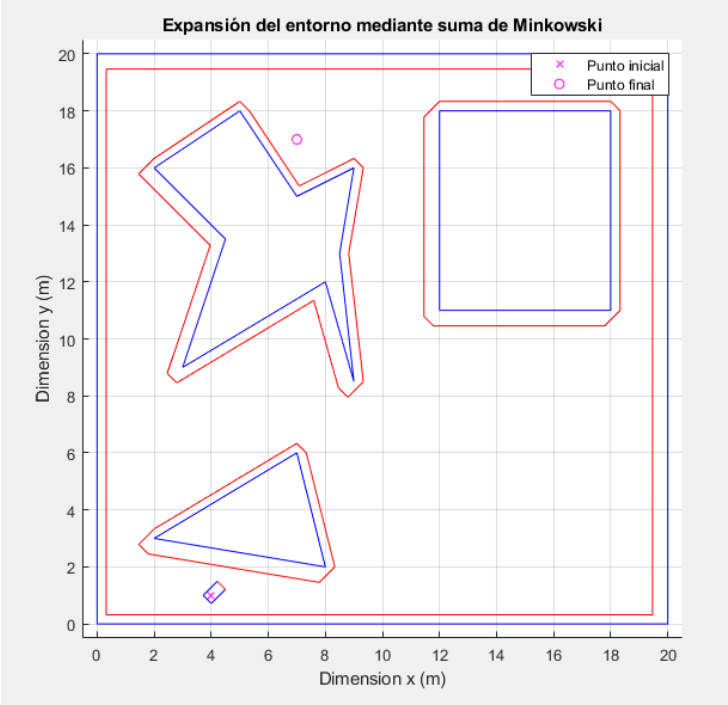

***Figura 73**. Expansión del entorno mediante suma de Minkowski en la simulación conjunta.*

A continuación, se realiza el resto de pasos en el entorno expandido, a excepción del diagrama de Voronoi, que se observará para ambos.

2) Descomposición del entorno en bandas verticales y cálculo de ruta mediante Dijkstra. Se produce una descomposición en 50 divisiones verticales, es decir, puesto que el entorno posee una dimensión x máxima de 20 metros (aproximadamente, ya que tras la expansión disminuye), la longitud elemental por banda será de 0.4 metros (aproximadamente). El número de nodos máximo por banda (*num_nodos*) será de 5.

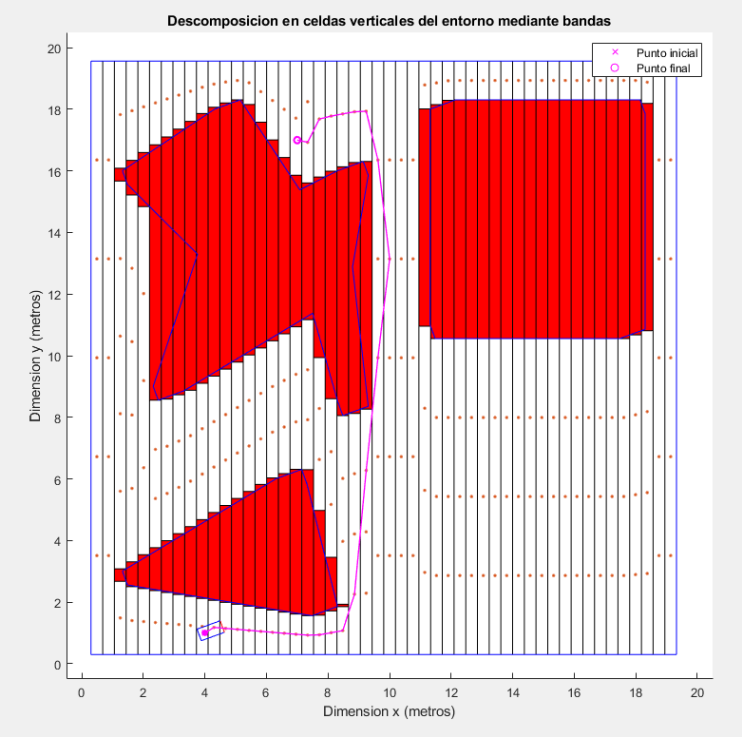

***Figura 74**. Descomposición en bandas verticales y cálculo de ruta mediante Dijkstra en la simulación conjunta.*

3) Descomposición del entorno en celdas verticales por configuración trapezoidal. Únicamente depende de los vértices de los obstáculos:

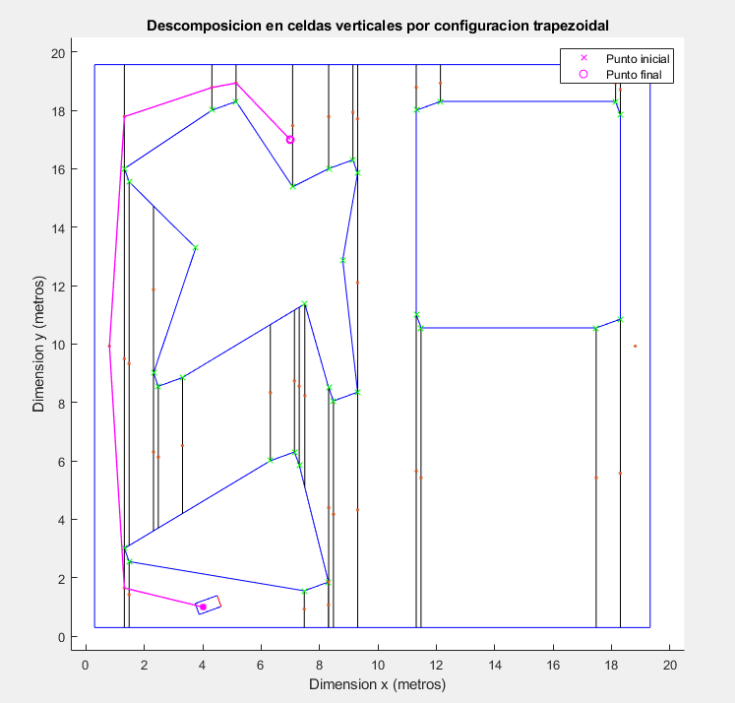

***Figura 75**. Descomposición en celdas verticales por configuración trapezoidal y cálculo de ruta mediante Dijkstra en la simulación conjunta.*

4) Descomposición del entorno en celda verticales y cálculo de ruta mediante campo potencial. La resolución escogida es de 50 divisiones horizontales por 50 verticales. Debido a las dimensiones originales del entorno (20 metros), el ancho y altura de cada celda será de aproximadamente 0.4 metros.

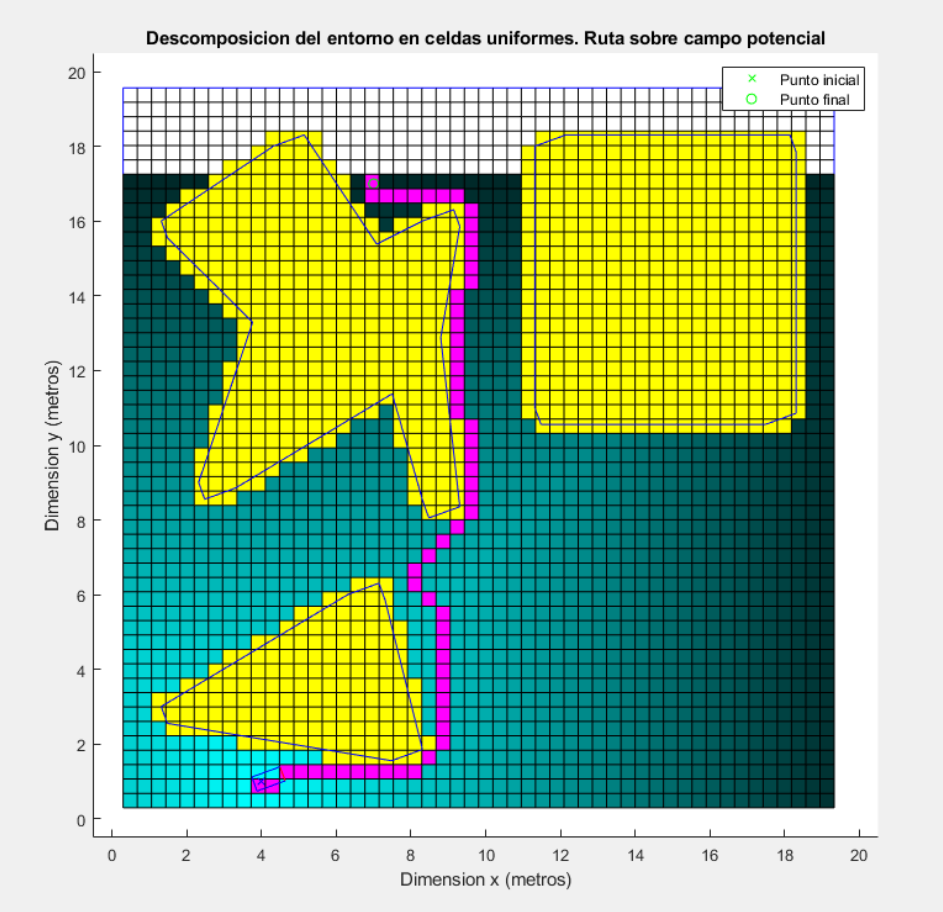

***Figura 76**. Descomposición en celdas uniformes y cálculo de ruta mediante campo potencial en la simulación conjunta.*

5) Descomposición del entorno en Quadtree y posterior cálculo de ruta mediante Dijkstra. Se ha escogido un nivel de ramificación de 4 subdivisiones consecutivas. Debido a las dimensiones originales del entorno (20 metros), las celdas más ramificadas poseerán una anchura y una altura de aproximadamente 1.25 metros.

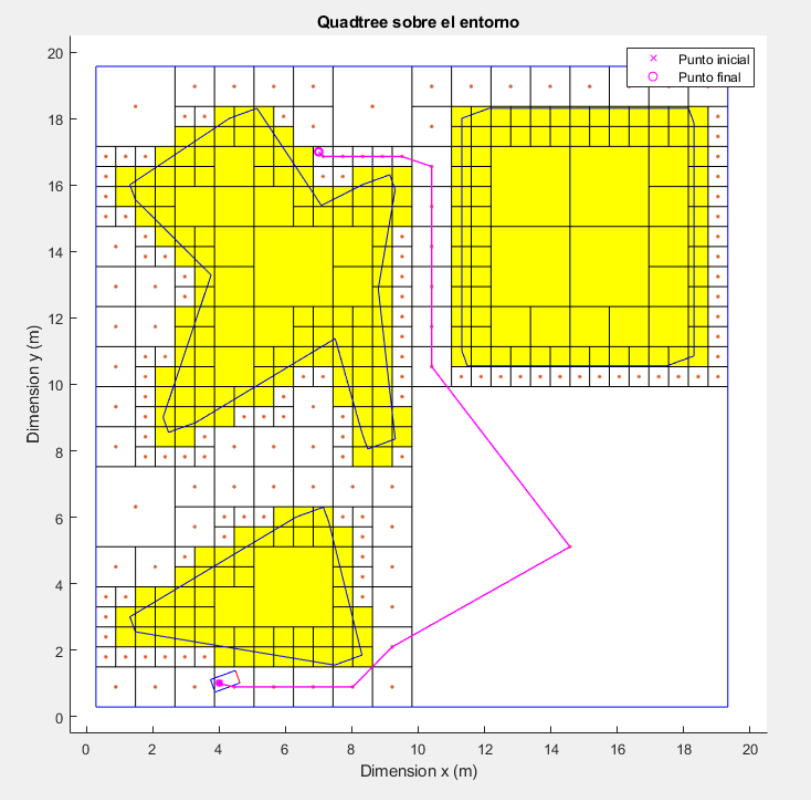

***Figura 77**. Descomposición en Quadtree y cálculo de ruta mediante Dijkstra en la simulación conjunta.*

6) Representación del diagrama de Voronoi. Tomado un nivel de precisión de 4 (sobre 5) y simulando para el entorno original y para el expandido.

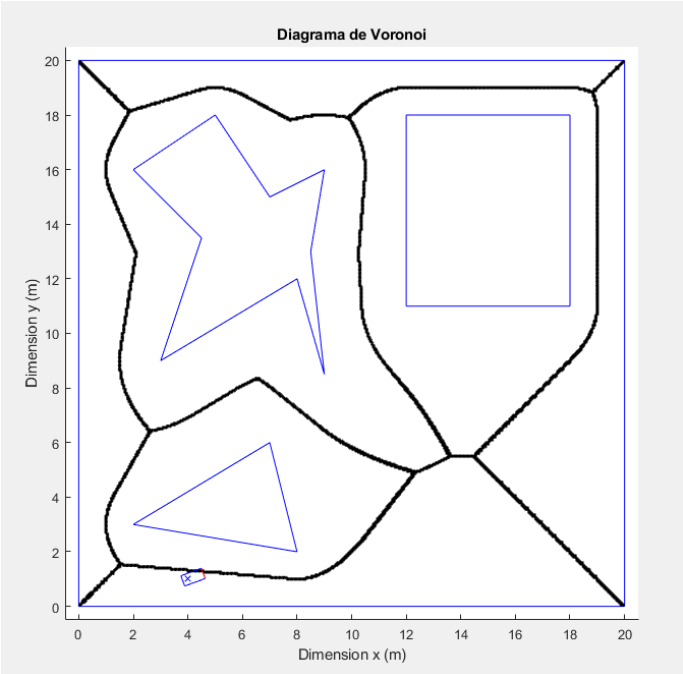

***Figura 78**. Diagrama de Voronoi en el entorno original de la simulación conjunta.*

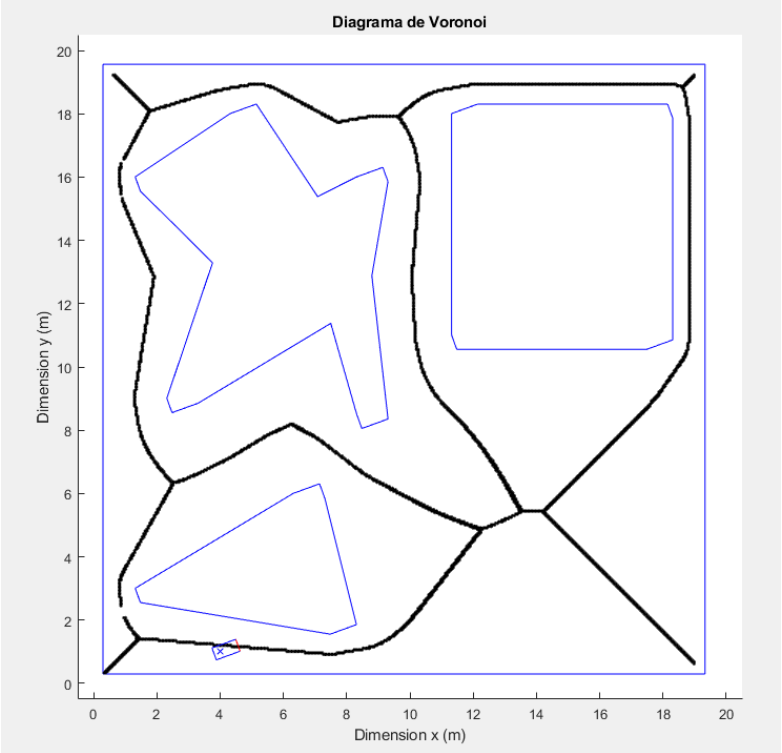

***Figura 79**. Diagrama de Voronoi en el entorno expandido de la simulación conjunta.*

Además, para un correcto análisis, se ha medido los tiempos de ejecución de cada uno de los procesos (comando *tic toc* de Matlab). A continuación, se muestra una tabla en la que se exponen:

	Tiempo de cómputo (segundos)
Expansión del entorno mediante suma de Minkowski	0.366319
Bandas verticales: descomposición	0.514409
Bandas verticales Dijkstra	7.813008
Configuración trapezoidal: descomposición	0.179657
Configuración trapezoidal: Dijkstra	0.484182
Celdas uniformes: descomposición	5.911295
Celdas uniformes: campo potencial	1.927009
Quadtree: descomposición	2.905137

Quadtree: Dijkstra	7.374844
Diagrama de Voronoi del entorno original	12.682108
Diagrama de Voronoi del entorno expandido	13.714545

***Tabla 2**. Comparativa de coste computacional entre todas las ejecuciones de la simulación conjunta.*

11.2. Conclusiones

A partir de los resultados obtenidos y del propio desarrollo del proyecto, se puede extraer las siguientes conclusiones:

- La expansión del entorno mediante suma de Minkowski es un método muy útil porque permite economizar el espacio disponible al máximo permitido por las características físicas del robot. No obstante, esta gran eficiencia espacial viene en contrapartida de un exigente coste computacional. Este no se debe a que el cálculo de la expansión demande muchos recursos de computación (véase *Tabla 2*), sino que, al depender de la orientación del robot, los resultados solo son válidos mientras el robot no modifique su orientación.

  Por ello, en el transcurso de una ruta, el robot deberá comprobar la expansión del entorno periódicamente en función de la orientación en cada momento. Y, puesto que la ruta se calcula sobre el entorno expandido, esta deberá ser recalculada también cada vez que se actualicen los resultados de la expansión.

- De entre las técnicas de descomposición en celdas que se ha implementado, las más demandantes de recursos de cómputo son las celdas uniformes y la descomposición a partir de bandas verticales (véase *Tabla 2*).

  Las celdas uniformes es un método muy útil en lo que descomposición y análisis del entorno se refiere, pero, para la planificación de rutas, esta técnica puede resultar deficiente en tiempo y en coste cuando la resolución deseada es demasiado elevada. Asimismo, la expansión y posterior exploración del campo vectorial, aunque resulta en rutas muy precisas, no se ha logrado (durante este proyecto) una implementación eficiente en el uso de los recursos.

  Por otro lado, la descomposición en celdas verticales mediante bandas también puede acarrear un coste computacional elevado. No obstante, este coste resulta incierto para esta técnica, debido a la capacidad de ajuste del usuario. Como se puede observar en la Tabla 2, el coste asociado a meramente la descomposición no es elevado. Sin embargo, debido a la resolución de las bandas y a la gran cantidad de nodos extraídos de cada una (se ha escogido que cada banda pueda tener hasta un máximo de 5 nodos), la exploración que debe realizar el algoritmo de Dijkstra sobre el grafo es muy extensa, y es la que ocasiona que el coste computacional aumente tan drásticamente. Cabe recalcar, además, que la selección de 5 nodos de máximo por banda no ha sido fortuita.

En simulaciones previas se escogía un número menor, pero esto incurría en una las mayores limitaciones de esta técnica: la conectividad entre nodos pertenecientes a celdas adyacentes no está garantiza. Debido a esta razón, se aumentó para la simulación el número de nodos accesibles para el grafo, lo que llevó a la generación exitosa de una trayectoria, pero a costa inevitablemente de un aumento drástico en el coste computacional.

- Referido al Quadtree, se trata de una técnica muy útil para delimitar con gran precisión la región libre de la región ocupada por obstáculos. Como se puede notar en la *Tabla 2*, únicamente respecto al proceso de descomposición, este es mucho más eficiente que las celdas uniformes. Sin embargo, la naturaleza del Quadtree lo conduce a generar una gran cantidad de celdas. Al aplicar algoritmos de cálculo de ruta basados en grafos, se requiere la extracción de una cantidad elevada de nodos. Como se mencionaba con las bandas verticales, ello produce un aumento en el coste computacional asociado a la exploración del grafo y el cálculo de la trayectoria.

  Asimismo, presenta otra desventaja que puede ser observada en la *Figura 77*. Ante la irregularidad de tamaño y conexión entre celdas, puede surgir rutas que realizan trazados ineficientes.

- Por otro lado, resalta el coste, tanto de descomposición como de generación de trayectorias, de las celdas verticales en configuración trapezoidal (véase *Tabla 2*). Se trata de una técnica más simple, tanto a nivel de diseño como de ejecución, que produce los mejores resultados en tiempo. El principal inconveniente de este método es que depende completamente de la modelización del entorno, por lo que el usuario no puede introducir los ajustes oportunos para cada caso. La conclusión es que resulta en una técnica que puede ser muy útil, como en la simulación observada, cuando el entorno posee muchos vértices (mayor complejidad). En entorno con pocos obstáculos o modelados de forma muy simple, los nodos accesibles de la red son escasos y las rutas implementadas bruscas.

  No obstante, la descomposición en celdas verticales en configuración trapezoidal aporta una ventaja muy relevante y en contraposición a las bandas verticales: la conectividad entre los nodos de celdas adyacentes está garantizada.

- Por último, el diagrama de Voronoi es una técnica desde la que se puede visualizar el camino más seguro que podría trazar el robot en el entorno. A partir de la *Figura 78* y *Figura 79*, entre el entorno expandido y el original, el diagrama no difiere demasiado, por lo que es una técnica aplicable directamente al entorno original sin la necesidad de preprocesos sobre el entorno. Ello resulta lógico ya que la ruta más segura casi inevitablemente deberá estar en la región libre delimitada por la expandida.

  Como inconveniente, se puede observar a partir de las *Figura 79* (diagrama de Voronoi sobre el entorno expandido) y la *Figura 77* (Quadtree) las limitaciones del algoritmo debido al preproceso del Quadtree. Al requerir este pretratamiento, es posible delimitar algunos puntos del trazado de Voronoi como celdas ocupadas. Ello podría solventarse aumentando la resolución interna del Quadtree (nivel de ramificación máxima), pero ello iría acompañado de un incremento en el coste computacional.

11.3. Trabajos futuros

Como se ha presentado a lo largo de esta Memoria, el trabajo de Fin de Grado desarrollado ha pretendido ampliar la Toolbox de Matlab referida a planificación de movimiento de robots móviles. Mediante la implementación de diversos algoritmos, se ha explorado nuevas técnicas de descomposición en celdas del entorno y se ha diseñado algunos preprocesos y tratamientos como la expansión del entorno mediante suma de Minkowski o el diagrama de Voronoi.

En este contexto, un solo proyecto como el presente no es suficiente para abarcar con toda la disciplina de la planificación de movimiento, ni siquiera para la descomposición en celdas. Es por ello que, para trabajos futuros, se deja la senda abierta a nuevas técnicas.

Un buen ejemplo de estas técnicas es el uno de los programas propios desarrollados en este proyecto: celdas verticales mediante configuración trapezoidal. Este tipo de descomposición es en realidad un ejemplo de las denominadas descomposiciones 'Morse', en la que la delimitación entre las celdas (segmentos) sigue la ecuación de una recta. No obstante, no esta no es el único método. Existen otros que, a partir de un punto del entorno, expanden circunferencias que interseccionan con los vértices del entorno o trazan tangentes a hacia los obstáculos.

Por otro lado, además de celdas verticales y uniformes, existe otros métodos que optan por dividir el entorno en mallas triangulares que buscan encerrar cada uno de los obstáculos en un polígono. Se obtiene una representación que se denomina triangulación de Delaunay, íntimamente relacionada con el diagrama de Voronoi.

Por último, retomando la estela que deja este trabajo, los algoritmos desarrollados son susceptibles de mejora, sobre todo el caso de las celdas uniformes, cuya expansión del campo se puede optimizar e incluir aspectos adicionales como considerar a los obstáculos focos de repulsión del campo. También sobresale el diagrama de Voronoi, en el que podría ser muy interesante la generación de trayectorias sobre el trazado.

12. Referencias

Wikimedia Foundation, Inc. (Junio de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/Motion_planning

© Microsoft 2022. (2022). *Microsoft*. Obtenido de https://www.microsoft.com/es-es/microsoft-365/p/microsoft-365-personal/CFQ7TTC0K5BF/007R?source=googleshopping&ef_id=Cj0KCQjw8uOWBhDXARIsAOxKJ2GTIT7HTEaiIgUpL70u_yh_rvUFPP4cg6AbWUdutUv-51qfrKbCnF4aAuGGEALw_wcB:G:s&OCID=AIDcmm474qp8el_SEM_Cj0KCQjw8uOWBhDXA

Aboy, D. J. (2008). *SlidePlayer.es Inc.* Obtenido de https://slideplayer.es/slide/2261858/

Aradas, A. (Mayo de 2021). *CuestionesLaborales.es*. Obtenido de https://www.cuestioneslaborales.es/como-calcular-la-jornada-de-trabajo/#:~:text=liviana%20o%20intensiva.-,Jornada%20anual,18%20de%20septiembre%20de%202007).

CHC Energía. (Junio de 2019). *CHC Energía*. Obtenido de https://chcenergia.es/blog/cuanto-consume-un-ordenador-o-pc/#:~:text=Cuanta%20electricidad%20consume%20un%20ordenador&text=Y%20es%20que%20al%20igual,en%208%20horas%20de%20trabajo.

Choset, H. (s.f.). Presentation of Robotic Motion Planning: Cell Decompositions. Pittsburgh, Pensilvania, Estados Unidos.

cyberne1, P. (Marzo de 2013). *CyberneticZoo.com*. Obtenido de http://cyberneticzoo.com/early-industrial-robots/1958-62-versatran-industrial-robot-harry-johnson-veljko-milenkovic/

EDS Robotics. (Febrero de 2021). *EDS Robotics*. Obtenido de https://www.edsrobotics.com/blog/evolucion-robotica-industrial/

GAME España. (s.f.). *GAME España*. Obtenido de https://www.game.es/msi-nightblade-mi2-205eu-i7-6700-gtx-1060-3gb-8gb-1tb-hdd-128gb-ssd-w10-pc-hardware-132698

Ghosh, S., Halder, A., & Sinha, M. (2011). *Micro air vehicle path planning in fuzzy quadtree framework.* Elsevier.

Gil, M. (Octubre de 2014). *ActioGlobal*. Obtenido de https://www.actioglobal.com/es/karakuri/

Glassdoor. (22 de Julio de 2022). *glassdoor.es*. Obtenido de https://www.glassdoor.es/Sueldos/ingeniero-junior-sueldo-SRCH_KO0,16.htm

Hernandez, P. P. (s.f.). *World of πer*. Obtenido de https://pier.guillen.com.mx/algorithms/07-geometricos/07.4-interseccion_segmentos.htm

HibridosyElectricos.com. (Junio de 2019). *Tecnofisis Global, S.L.* Obtenido de https://www.hibridosyelectricos.com/articulo/tecnologia/roboat-robots-acuaticos-autonomos-reconocen-forman-estructuras/20190630203415028686.html

LaValle, S. M. (2006). Combinatorial Motion Planning. En S. M. LaValle, *Planning Algorithms* (págs. 250-309). Cambridge: Cambridge University Press.

LaValle, S. M. (2006). Planning Algorithms. Cambridge, Reino Unido.

Markos, P. (Marzo de 2013). *Blogger.com*. Obtenido de http://georitmos.blogspot.com/2013/03/interseccion-de-dos-segmentos.html

Masa, A. P. (Mayo de 2015). *alpoma.net*. Obtenido de https://alpoma.net/tecob/?p=11359

Mecalux, S.A. (Mayo de 2021). *Mecalux Esmena*. Obtenido de https://www.mecalux.es/blog/amr-vs-agv

Moravec, H. (1983). *The Stanford Cart and the CMU Rover.* Pittsburgh: IEEE.

Parra, S. (Noviembre de 2020). *Xataka Ciencia*. Obtenido de https://www.xatakaciencia.com/robotica/este-fue-primer-robot-historia-podia-decir-700-palabras-voz-alta-fumaba-cigarrillos#:~:text=ELEKTRO%2C%20el%20primer%20androide,un%20fon%C3%B3grafo%20de%2078%20rpm).

PC Componentes y Multimedia SLU. (2022). *PC Componentes*. Obtenido de https://www.pccomponentes.com/microsoft-windows-10-home-64bits-oem

PC Componentes y Multimedia SLU. (s.f.). *PC Componentes*. Obtenido de https://www.pccomponentes.com/aoc-agon-ag251fz2e-245-led-fullhd-240hz-freesync-premium

Pérez, A. E. (Julio de 2022). *OpenWebinars S.L.* Obtenido de https://openwebinars.net/blog/robotica-movil-que-es-y-sus-aplicaciones/

Revista de Robots. (5 de Julio de 2021). *Revista de Robots*. Obtenido de https://revistaderobots.com/robots-y-robotica/robotica-movil-que-es-la-robotica-movil-y-para-que-sirve/

Robotnik Automation S.L.L. (Agosto de 2021). *Robotnik Automation S.L.L.* Obtenido de https://robotnik.eu/es/que-es-un-robot-movil-autonomo-amr-lo-que-aportan-nuestros-robot-moviles-a-tu-empresa/

Rodriguez, P. B. (2016). *Control y planificación de robots móviles.* Sevilla: Universidad de Sevilla.

Šeda, M. (Febrero de 2007). *Roadmap Methods vs. Cell Decomposition in Robot Motion Planning.* Brno: Institute of Automation and Computer Science: Brno University of Technology.

Selectra. (24 de Julio de 2022). *TarifaLuzHora.es*. Obtenido de https://tarifaluzhora.es/info/precio-kwh#precio-kwh-mercado-libre

Sieira, A. G. (2011). *PLANIFICACIÓN DE MOVIMIENTO EN ROBÓTICA MÓVIL.* Santiago de Compostela: Universidad de Santiago de Compostela.

Simplilearn. (Junio de 2022). *Simplilearn Solutions*. Obtenido de https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm

Stanford University. (s.f.). *Stanford Engineering Compuetr Science*. Obtenido de https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/robotics/basicmotion.html

The MathWorks, Inc. (1994-2022). *The MathWorks, Inc.* Obtenido de https://es.mathworks.com/products/matlab.html

The MathWorks, Inc. (2022). *The MathWorks, Inc.* Obtenido de https://es.mathworks.com/pricing-licensing.html

U. Acar, E., Choset, H., A. Rizzi, A., N. Atkar, P., & Hull, D. (2002). Morse Decomposition for Coverage Tasks. *The International Journal of Robotics Research*, 331-344.

Wikimedia Foundation, Inc. (Diciembre de 2021). *Wikipedia The Free Encyclopedia*. Obtenido de https://es.wikipedia.org/wiki/Aut%C3%B3mata_(mec%C3%A1nico)

Wikimedia Foundation, Inc. (Julio de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/Mobile_robot

Wikimedia Foundation, Inc. (Julio de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://es.wikipedia.org/wiki/Rob%C3%B3tica

Wikimedia Foundation, Inc. (Julio de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/History_of_robots

Wikimedia Foundation, Inc. (Mayo de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/Unimate

Wikimedia Foundation, Inc. (Julio de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/Autonomous_underwater_vehicle

Wikimedia Foundation, Inc. (Junio de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree

Wikimedia Foundation, Inc. (Marzo de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/Probabilistic_roadmap

Wikimedia Foundation, Inc. (Abril de 2022). *Wikipedia The Free Encyclopedia*. Obtenido de https://en.wikipedia.org/wiki/D*

Zyania Post. (Noviembre de 2015). *Zyania SA*. Obtenido de https://www.zyania.com.mx/blog/inmoov-un-robot-de-codigo-abierto-impreso-en-3d/

Anexos

Índice Anexos

Presupuesto

Aunque el presente trabajo se proyecta en un contexto físico intangible, basado en la simulación; la etapa de diseño que se aborda se puede enmarcar en las connotaciones de un anteproyecto ingenieril, por lo que esta etapa es susceptible de realizar un análisis en costes. Debido a esta razón se procede a la elaboración de un presupuesto.

El siguiente presupuesto divide los costes en tres categorías (costes materiales, intangibles y en personal) y considera una etapa de diseño de 300 horas divididas en cuatro meses (desde marzo hasta junio, ambos incluidos).

Presupuesto		
Costes materiales		
Concepto	**Coste mensual (si procede)**	**Coste total**
1. Equipo informático		1.830,00 €
a) Ordenador de sobremesa: MSI gama Nightblade		1.200,00 €
b) Monitor AOC serie AGON		520,00 €
c) Periféricos		130,00 €
2. Electricidad*	5,04 €	20,16 €
3. Conexión a internet	30,00 €	120,00 €
Costes intangibles		
Concepto	**Coste mensual (si procede)**	**Coste total**
1. Sistema operativo para el equipo informático (Windows 10 Home 64 bits)**		- €
2. Licencia de Office 365	7,00 €	28,00 €
3. Licencia de Matlab (840 € anuales)	70,00 €	280,00 €
Costes en personal		
Concepto	**Coste por hora de trabajo**	**Coste total**
1. Ingeniero***	13,75 €	4.125,00 €
COSTE TOTAL		**8.253,16 €**

*Para el cálculo del coste de electricidad se ha considerado un consumo medio de la CPU de 200 W/h y el monitor de 24 W/h. A partir de un precio medio unitario para el kWh de 0,30€, se recalcula para las 300 horas de trabajo estimadas.

**No se ha tenido en cuenta un sobrecoste en lo que a sistema operativo se refiere, ya que las ofertas propias del ordenador de sobremesa incluyen este paquete en su venta. En caso contrario, el sistema operativo sugerido se valora en aproximadamente 135€.

***A partir de un sueldo medio base anual en España de 25.097€ para un ingeniero junior y de una jornada laboral anual aproximada en 1825 horas, se recalcula para las 300 horas de trabajo estimadas.

Todas las fuentes consultadas para la extracción de estos datos se encuentran en la sección de *'12. Referencias'* (pág. 104, véase *Índice General*).

Código en Matlab de *expandir_entorno_minkowski.m*:

```
function expandido  = expandir_entorno_minkowski(robot_x, robot_y,
robot_or, robot_sist, tolerancia, entorno)
%A partir de las dimensiones estructurales de un robot movil rectangular
%(robot_x: longitud, robot_y: ancho), la orientacion del robot
(robot_or),
%el sistema de referencia movil del robot (robot_sist) y una tolerancia
%añadida sobre la expansión exacta, calcula y representa el entorno
%expandido mediante suma de Minkowski correspondiente al entorno que se
%recibe como argumento (entorno).
%
%Requerimiento: todo el entorno debe estar definido en sentido horario.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se calculan los angulos por si estos no han sido introducidos:
entorno=reescribir_entorno(entorno,0,0);
%Se definen algunas variables:
L=size(entorno,1);  %Número de segmentos del entorno original
expandido_0 = []; %Matriz que contendra la primera expansion de los
segmentos del entorno
expandido = []; %Matriz que contendra la expansion final del entorno
%-------------------------------------------------------------------------
--
%Calculo de paralelas tomando en cuenta la posicion relativa del robot
respecto al obstaculo:
%-------------------------------------------------------------------------
%Se recorren los segmentos
for(i=1:L)
    expandido_0=[expandido_0;

minkowski_paralela_preliminar(entorno(i,:),robot_x,robot_y,robot_or,robot
_sist,tolerancia)]; %Se calculan paralelas preliminares
end
%-------------------------------------------------------------------------
--
%Calcula de la region expandida en los vertices:
%-------------------------------------------------------------------------
%Se recorren los obstaculos
for(i=0:max(entorno(:,8)))
    obstaculo_actual=extrae_obstaculo(i,entorno); %Se extrae el obstaculo
actual
    L=size(obstaculo_actual,1);  %Número de vertices del obstaculo
original (mismo numero que segmentos)
    puntos_lim=[]; %Matriz que contendra los puntos limite de la region
expandida para el obstaculo actual
    obstaculo_expandido=[]; %Matriz que alojara el resultado de expandir
el obstaculo actual
    for(j=1:L) %Se recorren los vertices del obstaculo actual
        puntos_lim = [puntos_lim;
```

```
minkowski_puntos_vertice(obstaculo_actual(j,1:3),entorno,expandido_0,robo
t_x,robot_y,robot_or,robot_sist,tolerancia)];
    end

    %Debido a la forma de calculo de minkowski_puntos_vertice.m pueden
    %aparecer subregiones expandidas dentro de la propia region
expandida.
    %Se procede a eliminarlas:
    k=1;
    while(k<length(puntos_lim(:,1)))
        punto_lim_actual = puntos_lim(k,:);
        indice_actual=k;
        r=1;
        while(r<=length(puntos_lim(:,1)))

if((punto_lim_actual(1)==puntos_lim(r,1))&(punto_lim_actual(2)==puntos_li
m(r,2))&(punto_lim_actual(3)==puntos_lim(r,3))&(r~=indice_actual))
                puntos_lim(indice_actual:r-1,:)=[];
            end
            r=r+1;
        end
        k=k+1;
    end
    %Se construye el obstaculo y se añade al entorno resultado final
    k=1;
    while(k<length(puntos_lim(:,1)))
        obstaculo_expandido = [obstaculo_expandido;
                               puntos_lim(k,:), puntos_lim(k+1,:), 0, i];
        k=k+1;
    end
    obstaculo_expandido = [obstaculo_expandido;
                           puntos_lim(k,:), puntos_lim(1,:), 0, i];
    expandido = [expandido;
                 obstaculo_expandido];
end
%-------------------------------------------------------------------------
--
%Representacion grafica:
%-------------------------------------------------------------------------
--
figure;
title("Expansión del entorno mediante suma de Minkowski");
xlabel("Dimension x (m)");
ylabel("Dimension y (m)");
dibuent(entorno,[0,0,1]); %Entorno original
axis([min(entorno(:,1))-0.5 max(entorno(:,1))+0.5 min(entorno(:,2))-0.5
max(entorno(:,2))+0.5]);  % Definir los límites.
hold on;
grid;
dibuent(expandido,[1 0 0]); %Entorno expandido
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *minkowski_paralela_preliminar.m*:

```
function paralela_preliminar = minkowski_paralela_preliminar(segmento,
robot_x, robot_y, robot_or, robot_sist, tolerancia)
%Funcion auxiliar de expandir_entorno_minkowski.m.
%
%Devuelve la paralela preliminar del segmento (en formato
%entorno, 8 columnas) que se le introduce como argumento de entrada.
%Requiere, ademas, como argumentos de entrada: el entorno, los datos
%estructurales del robot (robot_x: longitud, robot_y: ancho), la
%orientacion del robot (robot_or), el sistema de referencia movil del
robot
%(robot_sist), y la tolerancia (en %) añadida a la expansión.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se definen algunas variables:
robot_vector = [cos(robot_or*pi/180.0), sin(robot_or*pi/180.0), 0];
%Vector director que identifica la orientacion del robot
tolerancia = tolerancia/100*max(robot_x,robot_y); %Se expresa en metros
la tolerancia
%Se calcula el angulo que forman el vector director del robot y el
segmento:
segmento_vector = [segmento(4)-segmento(1), segmento(5)-segmento(2), 0];
beta =
acos((robot_vector(1)*segmento_vector(1)+robot_vector(2)*segmento_vector(
2))/(sqrt((segmento_vector(1))^2+(segmento_vector(2))^2)));
%-------------------------------------------------------------------------
--
%Determinacion de la distancia entre paralelas en funcion de la posicion
relativa entre robot y segmento:
%-------------------------------------------------------------------------
--
if(segmento(8)==0) %Si es un limite del entorno
    if(beta==0)
        dist_seg = robot_y - robot_sist(2);
    end
    if(beta>0)&(beta<pi/2)
        prod_vect = cross(robot_vector, segmento_vector);
        if(prod_vect(3)<0)
            dist_seg = sin(beta)*(robot_x-robot_sist(1)+(robot_y-
robot_sist(2))/(tan(beta)));
        else
            dist_seg = sin(beta)*(robot_sist(1)+(robot_y-
robot_sist(2))/(tan(beta)));
        end
    end
    if(beta==pi/2)
        prod_vect = cross(robot_vector, segmento_vector);
        if(prod_vect(3)<0)
            dist_seg = robot_x - robot_sist(1);
        else
            dist_seg = robot_sist(1);
```

```
        end
    end
    if(beta>pi/2)&(beta<pi)
        beta = pi - beta;
        prod_vect = cross(robot_vector, segmento_vector);
        if(prod_vect(3)<0)
            dist_seg = sin(beta)*(robot_x-
robot_sist(1)+(robot_sist(2))/(tan(beta)));
        else
            dist_seg =
sin(beta)*(robot_sist(1)+(robot_sist(2))/(tan(beta)));
        end
    end
    if(beta==pi)
        dist_seg = robot_sist(2);
    end

%Para los obstaculos
else
    if(beta==0)
        dist_seg = robot_sist(2);
    end
    if(beta>0)&(beta<pi/2)
        prod_vect = cross(robot_vector, segmento_vector);
        if(prod_vect(3)<0)
            dist_seg =
sin(beta)*(robot_sist(1)+(robot_sist(2))/(tan(beta)));
        else
            dist_seg = sin(beta)*(robot_x-
robot_sist(1)+(robot_sist(2))/(tan(beta)));
        end
    end
    if(beta==pi/2)
        prod_vect = cross(robot_vector, segmento_vector);
        if(prod_vect(3)<0)
            dist_seg = robot_sist(1);
        else
            dist_seg = robot_x - robot_sist(1);
        end
    end
    if(beta>pi/2)&(beta<pi)
        beta = pi - beta;
        prod_vect = cross(robot_vector, segmento_vector);
        if(prod_vect(3)<0)
            dist_seg = sin(beta)*(robot_sist(1)+(robot_y-
robot_sist(2))/(tan(beta)));
        else
            dist_seg = sin(beta)*(robot_x-robot_sist(1)+(robot_y-
robot_sist(2))/(tan(beta)));
        end
    end
    if(beta==pi)
        dist_seg = robot_y - robot_sist(2);
    end
```

```
end
%-------------------------------------------------------------------------
--
%Calculo de la paralela preliminar:
%-------------------------------------------------------------------------
--
%Se actualiza la distancia segura ampliando la expansion del entorno en
un factor de seguridad
dist_seg = dist_seg + tolerancia;
%Debido a la naturaleza de la posterior funcion paralela_segmento.m, la
%distancia a introducir a los obstaculos es negativa
if(segmento(8)~=0)
    dist_seg=-dist_seg;
end
%Se calcula recta paralela
paralela_preliminar = paralela_segmento(segmento, dist_seg);
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *minkowski_puntos_vertice.m*:

```
function puntos_lim = minkowski_puntos_vertice(vertice, entorno,
expandido_0, robot_x, robot_y, robot_or, robot_sist, tolerancia)
%Funcion auxiliar de expandir_entorno_minkowski.m.
%
%Devuelve los puntos limite del entorno expandido en la region cercana al
%vertice que se le introduce como entrada. Requiere como entradas,
además,
%el entorno original, el entorno expandido preliminar (expandido_0),
%los datos estructurales del robot (robot_x: longitud, robot_y: ancho),
la
%orientacion del robot (robot_or), el sistema de referencia movil del
robot
%(robot_sist), y la tolerancia (en %) añadida a la expansión.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se definen algunas variables:
robot_vector = [cos(robot_or*pi/180.0), sin(robot_or*pi/180.0), 0];
%Vector director que identifica la orientacion del robot
tolerancia = tolerancia/100*max(robot_x,robot_y); %Se expresa en metros
la tolerancia
%Se extrae los segmentos del entorno original que forman el vertice:
L=size(entorno,1);  %Número de segmentos del entorno original
for(i=1:L)

if((vertice(1)==entorno(i,1))&(vertice(2)==entorno(i,2))&(vertice(3)==ent
orno(i,3)))
        segmento2=entorno(i,:);
        indice2=i;
    end

if((vertice(1)==entorno(i,4))&(vertice(2)==entorno(i,5))&(vertice(3)==ent
orno(i,6)))
        segmento1=entorno(i,:);
        indice1=i;
    end
end
%-------------------------------------------------------------------------
--
%Obtencion de los puntos que delimitan la region expandida del vertice de
entrada:
%-------------------------------------------------------------------------
--
%Se comprueba si el vertice es concavo o convexo
if((extrae_informacion_vertice(entorno,vertice)==0)|(segmento1(8)==0))
%Vertice concavo
    pc=inter_seg(expandido_0(indice1,1:2), expandido_0(indice1,4:5),
expandido_0(indice2,1:2), expandido_0(indice2,4:5));
    puntos_lim = [pc(1:2), 0];
else %Vertice convexo
    %Se saca un maximo de dos puntos a partir de cada segmento.
```

```
    %Segmento 1:

    %Se calcula la posicion relativa entre el robot movil y el segmento
en cuestion:
    segmento1_vector = [segmento1(4)-segmento1(1), segmento1(5)-
segmento1(2), 0];
    beta1 =
acos((robot_vector(1)*segmento1_vector(1)+robot_vector(2)*segmento1_vecto
r(2))/(sqrt((segmento1_vector(1))^2+(segmento1_vector(2))^2)));

    %En funcion de la posicion realtiva, se calcula los puntos limitantes
potenciales para el segmento 1
    if(beta1==0)
        puntos_lim =
[vertice+robot_vector*(robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/18
0.0), sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
    end
    if(beta1>0)&(beta1<pi/2)
        prod_vect = cross(robot_vector, segmento1_vector);
        if(prod_vect(3)<0)
            puntos_lim =
[vertice+robot_vector*(robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/18
0.0), sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia);

vertice+robot_vector*(robot_sist(1)+tolerancia)-
[cos((robot_or+90)*pi/180.0), sin((robot_or+90)*pi/180.0), 0]*(robot_y-
robot_sist(2)+tolerancia)];
        else
            puntos_lim = [vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia);

vertice+robot_vector*(robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180
.0), sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
        end
    end
    if(beta1==pi/2)
        prod_vect = cross(robot_vector, segmento1_vector);
        if(prod_vect(3)<0)
            puntos_lim =
[vertice+robot_vector*(robot_sist(1)+tolerancia)-
[cos((robot_or+90)*pi/180.0), sin((robot_or+90)*pi/180.0), 0]*(robot_y-
robot_sist(2)+tolerancia)];
        else
            puntos_lim = [vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
        end
    end
    if(beta1>pi/2)&(beta1<pi)
        beta1 = pi - beta1;
        prod_vect = cross(robot_vector, segmento1_vector);
        if(prod_vect(3)<0)
```

```
            puntos_lim =
[vertice+robot_vector*(robot_sist(1)+tolerancia)-
[cos((robot_or+90)*pi/180.0), sin((robot_or+90)*pi/180.0), 0]*(robot_y-
robot_sist(2)+tolerancia);
                            vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)-[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_y-robot_sist(2)+tolerancia)];
        else
            puntos_lim = [vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)-[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_y-robot_sist(2)+tolerancia);
                            vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
        end
    end
    if(beta1==pi)
        puntos_lim = [vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)-[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_y-robot_sist(2)+tolerancia), 2];
    end

    %Segmento 2:

    %Se calcula la posicion relativa entre el robot movil y el segmento
en cuestion:
    segmento2_vector = [segmento2(4)-segmento2(1), segmento2(5)-
segmento2(2), 0];
    beta2 =
acos((robot_vector(1)*segmento2_vector(1)+robot_vector(2)*segmento2_vecto
r(2))/(sqrt((segmento2_vector(1))^2+(segmento2_vector(2))^2)));

    %En funcion de la posicion realtiva, se calcula los puntos limitantes
potenciales para el segmento 2
    if(beta2==0)
        puntos_lim = [puntos_lim;
                        vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
    end
    if(beta2>0)&(beta2<pi/2)
        prod_vect = cross(robot_vector, segmento2_vector);
        if(prod_vect(3)<0)
            puntos_lim = [puntos_lim;
                            vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia);

vertice+robot_vector*(robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180
.0), sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
        else
            puntos_lim = [puntos_lim;
```

```
                            vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)-[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_y-robot_sist(2)+tolerancia);
                            vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
        end
    end
    if(beta2==pi/2)
        prod_vect = cross(robot_vector, segmento2_vector);
        if(prod_vect(3)<0)
            puntos_lim = [puntos_lim;

vertice+robot_vector*(robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180
.0), sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia)];
        else
            puntos_lim = [puntos_lim;
                            vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)-[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_y-robot_sist(2)+tolerancia)];
        end
    end
    if(beta2>pi/2)&(beta2<pi)
        beta2 = pi - beta2;
        prod_vect = cross(robot_vector, segmento2_vector);
        if(prod_vect(3)<0)
            puntos_lim = [puntos_lim;

vertice+robot_vector*(robot_sist(1)+tolerancia)+[cos((robot_or+90)*pi/180
.0), sin((robot_or+90)*pi/180.0), 0]*(robot_sist(2)+tolerancia);

vertice+robot_vector*(robot_sist(1)+tolerancia)-
[cos((robot_or+90)*pi/180.0), sin((robot_or+90)*pi/180.0), 0]*(robot_y-
robot_sist(2)+tolerancia)];
        else
            puntos_lim = [puntos_lim;

vertice+robot_vector*(robot_sist(1)+tolerancia)-
[cos((robot_or+90)*pi/180.0), sin((robot_or+90)*pi/180.0), 0]*(robot_y-
robot_sist(2)+tolerancia);
                            vertice-robot_vector*(robot_x-
robot_sist(1)+tolerancia)-[cos((robot_or+90)*pi/180.0),
sin((robot_or+90)*pi/180.0), 0]*(robot_y-robot_sist(2)+tolerancia)];
        end
    end
    if(beta2==pi)
        puntos_lim = [puntos_lim;
                        vertice+robot_vector*(robot_sist(1)+tolerancia)-
[cos((robot_or+90)*pi/180.0), sin((robot_or+90)*pi/180.0), 0]*(robot_y-
robot_sist(2)+tolerancia)];
    end

    %Se comprueba si alguno de los puntos se repite para que solo
aparezca una vez:
```

```
    i=1;
    while(i<length(puntos_lim(:,1)))
        punto_lim_actual=puntos_lim(i,:);
        indice=0;
        j=1;
        while(j<length(puntos_lim(:,1)))

if((punto_lim_actual(1)==puntos_lim(j,1))&(punto_lim_actual(2)==puntos_li
m(j,2))&(punto_lim_actual(3)==puntos_lim(j,3))&(j~=i))
                indice=j;
            end
            j=j+1;
        end
        if(indice~=0)
            puntos_lim(indice,:) = [];
        end
        i=i+1;
    end
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *bandas_verticales.m*:

```
function [celdas_resultantes, celdas_obstaculos, nodos] =
bandas_verticales(entorno, subs, longitud_celda, num_nodos)
%Funcion que devuelve la descomposicion de un entorno de entrada en
bandas
%verticales, ademas de los nodos del grafo resultantes. También
%ofrece la representacion grafica del proceso.
%Se le puede introducir para ello el numero de bandas a dividir (subs) o
%la longitud de cada celda (longitud_celda). La variable que no se
%pretenda introducir debera valer 0. En caso de que se introduzcan ambos
%valores, se prioriza la longitud de la celda.
%
%Requerimiento: todo el entorno debe estar definido en sentido horario
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se declara la matroz nodos, salida de la funcion
nodos = [];
%Se extraen las dimensiones del entorno:
o_x_ent = min(entorno(:,1));
o_y_ent = min(entorno(:,2));
dimen_x_ent = max(entorno(:,1));
dimen_y_ent = max(entorno(:,2));
%En caso de que se haya introducido la longitud de la celda, se calcula
el
%numero de bandas y se recalcula la longitud de celda correspondiente
if(longitud_celda~=0)
    subs = ceil((dimen_x_ent-o_x_ent)/longitud_celda);
    longitud_celda = (dimen_x_ent-o_x_ent)/subs;
else
    %En caso de que se haya introducido el numero de bandas, recalcula la
    %longitud de la celda:
    if(subs~=0)
        longitud_celda = (dimen_x_ent-o_x_ent)/subs;
    end
end
%Se crea un arreglo tipo cell que contiene el entorno dividido en bandas
%verticales, sin considerar la presencia de obstaculos:
celdas_preliminares={};
%figure
for(i=1:subs)
    celdas_preliminares{1,i}=[o_x_ent+longitud_celda*i-longitud_celda
o_y_ent      0  o_x_ent+longitud_celda*i-longitud_celda  dimen_y_ent  0;
                                o_x_ent+longitud_celda*i
dimen_y_ent  0  o_x_ent+longitud_celda*i                 o_y_ent      0];
end
%dibuent(entorno,[0 0 1]);
%-------------------------------------------------------------------------
--
%Calculo de las celdas libres y ocupadas
%-------------------------------------------------------------------------
--
```

```
for(i=1:subs) %Se recorre cada banda preliminar

    %Se declara  tres matrices que contendran los puntos que son
    %potenciales limites horizontales
    intersecciones_izq=[]; %Contiene las intersecciones del lado
izquierdo de la celda preliminar con los obstaculos del entorno
    intersecciones_der=[]; %Contiene las intersecciones del lado derecho
de la celda preliminar con los obstaculos del entorno
    vertices_lim=[]; %Contiene aquellos vertices de obstaculos que pueden
limitar las celdas
    for(j=5:length(entorno(:,1))) %Para la banda preliminar actual, se
recorren todos los segmentos de los obstaculos

        %Se calcula las intersecciones del lado izquierdo

pc=inter_seg(celdas_preliminares{1,i}(1,1:2),celdas_preliminares{1,i}(1,4
:5),entorno(j,1:2),entorno(j,4:5));
        if(pc(3))
            intersecciones_izq=[intersecciones_izq;
                                pc(1:2), 0, entorno(j,8)];
            %plot(pc(1),pc(2),'or');
        end
        %Se calcula las intersecciones del lado derecho

pc=inter_seg(celdas_preliminares{1,i}(2,1:2),celdas_preliminares{1,i}(2,4
:5),entorno(j,1:2),entorno(j,4:5));
        if(pc(3))
            intersecciones_der=[intersecciones_der;
                                pc(1:2), 0, entorno(j,8)];
            %plot(pc(1),pc(2),'or');
        end
        %Se extrae los vertices limitantes posibles

if((entorno(j,1)>=celdas_preliminares{1,i}(1,1))&(entorno(j,1)<=celdas_pr
eliminares{1,i}(2,1)))
            [vertice_convexo, angulo_vertice, angulo1, angulo2] =
extrae_informacion_vertice(entorno, [entorno(j,1:3)]);

if(((angulo1>=0)&(angulo1<=90)&(((angulo2>=0)&(angulo2<=90)&(angulo1<angu
lo2))|((angulo2>=270)&(angulo2<=360))))|((angulo1>=270)&(angulo1<=360)&((
(angulo2>=270)&(angulo2<=360)&(angulo1<angulo2))|((angulo2>=0)&(angulo2<=
90)))))
                vertices_lim = [vertices_lim;
                                entorno(j,1:3), entorno(j,8), 1];
                %plot(entorno(j,1),entorno(j,2),'or');
            else

if((angulo2>=90)&(angulo2<=270)&(angulo1>=90)&(angulo1<=270)&(angulo2>ang
ulo1))
                    vertices_lim = [vertices_lim;
                                    entorno(j,1:3), entorno(j,8), -1];
                    %plot(entorno(j,1),entorno(j,2),'or');
                end
            end
```

```
        end
    end %fin extraccion de puntos potenciales limites horizontales

    %Se crea y calcula un vector que poseera los indices de los
obstaculos
    %que aparecen en la banda
    indices=[];
    if(size(intersecciones_izq,1)~=0)
        indices=[indices;
                 intersecciones_izq(:,4)];
    end
    if(size(intersecciones_der,1)~=0)
        indices=[indices;
                 intersecciones_der(:,4)];
    end
    if(size(vertices_lim,1)~=0)
        indices=[indices;
                 vertices_lim(:,4)];
    end
    indices=sort(indices); %Se ordena el vector de menor a mayor

    %Se reescribre el vector de indices de forma que solo aparezca cada
indice una vez
    indice_actual=0;
    j=1;
    while(j<=length(indices))
        if(indice_actual==indices(j))
            indices(j)=[];
        else
            indice_actual=indices(j);
            j=j+1;
        end
    end

    %Se declara tres arreglos tipo celda que contendran informacion del
proceso
    celdas_obstaculos_no_ord = cell(length(indices),1); %Codifica las
celdas que delimitan el espacio ocupado por los obstaculos sin ordenar
    celdas_obstaculos_ord = cell(length(indices),1); %Igual que la
anterior, pero ordenados los obstaculos en orden de aparacion vertical,
desde arriba a abajo
    celdas_libres = cell(length(indices)+1,1); %Codifica las celdas que
delimitan el espacio libre. Ordenada verticalmente, de arriba a abajo
    %Se recorren los obstaculos que aparecen en la banda preliminar, y se
    %busca la interseccion mayor y la menor para cada uno:
    for(j=1:length(indices))
        indice_actual=indices(j);
        maximo_actual=min(entorno(:,2))-1;
        minimo_actual=dimen_y_ent+1;
        %Busqueda en las intersecciones izquierdas
        if(size(intersecciones_izq,1)~=0)
            for(k=1:length(intersecciones_izq(:,1)))
```

```
if((intersecciones_izq(k,4)==indice_actual)&(intersecciones_izq(k,2)>maxi
mo_actual))
                    maximo_actual=intersecciones_izq(k,2);
                end

if((intersecciones_izq(k,4)==indice_actual)&(intersecciones_izq(k,2)<mini
mo_actual))
                    minimo_actual=intersecciones_izq(k,2);
                end
            end
        end
        %Busqueda en las intersecciones derechas
        if(size(intersecciones_der,1)~=0)
            for(k=1:length(intersecciones_der(:,1)))

if((intersecciones_der(k,4)==indice_actual)&(intersecciones_der(k,2)>maxi
mo_actual))
                    maximo_actual=intersecciones_der(k,2);
                end

if((intersecciones_der(k,4)==indice_actual)&(intersecciones_der(k,2)<mini
mo_actual))
                    minimo_actual=intersecciones_der(k,2);
                end
            end
        end
        %Busqueda en los vertices limitantes
        if(size(vertices_lim,1)~=0)
            for(k=1:length(vertices_lim(:,1)))

if((vertices_lim(k,4)==indice_actual)&(vertices_lim(k,2)>maximo_actual))
                    maximo_actual=vertices_lim(k,2);
                end

if((vertices_lim(k,4)==indice_actual)&(vertices_lim(k,2)<minimo_actual))
                    minimo_actual=vertices_lim(k,2);
                end
            end
        end

        %Se escribe la celda que ocupa el obstaculo actual, ordenado
        %respecto a los indices de los obstaculo, no verticalmente
        celdas_obstaculos_no_ord{j,1} = [celdas_preliminares{1,i}(1,1),
minimo_actual, 0, celdas_preliminares{1,i}(1,4), maximo_actual, 0;
                                         celdas_preliminares{1,i}(2,1),
maximo_actual, 0, celdas_preliminares{1,i}(2,4), minimo_actual, 0];
    end

    %Se reordena las celdas que contienen los obstaculos para que su
orden
    %sea en orden vertical de aparicion, de arriba a abajo
    for(r=1:length(indices))
        maximo_actual=min(entorno(:,2))-1;
```

```
        for(j=1:length(indices))
            if(maximo_actual<celdas_obstaculos_no_ord{j,1}(1,5))
                maximo_actual=celdas_obstaculos_no_ord{j,1}(1,5);
                indice_celda=j;
            end
        end
        celdas_obstaculos_ord{r,1} =
celdas_obstaculos_no_ord{indice_celda,1};
        celdas_obstaculos_no_ord{indice_celda,1}(1,5)=min(entorno(:,2))-
1;
    end

    %A partir de los celdas de los obstaculos ordenadas, se extrae por
    %complementariedad las celdas libres restando de la banda preliminar
    if(size(indices,1)~=0)
        for(r=1:length(indices)+1)
            if(r==1)
                celdas_libres{r,1}=[celdas_preliminares{1,i}(1,1),
celdas_obstaculos_ord{r,1}(1,5), celdas_preliminares{1,i}(1,3:6);
                                    celdas_preliminares{1,i}(2,1:4),
celdas_obstaculos_ord{r,1}(1,5), 0];
            end
            if((r~=1)&(r~=length(indices)+1))

if(celdas_obstaculos_ord{r,1}(1,2)<celdas_obstaculos_ord{r-1,1}(1,5))
                    celdas_libres{r,1}=[celdas_preliminares{1,i}(1,1),
celdas_obstaculos_ord{r,1}(1,5), 0, celdas_preliminares{1,i}(1,1),
celdas_obstaculos_ord{r-1,1}(1,2), 0;
                                        celdas_preliminares{1,i}(2,1),
celdas_obstaculos_ord{r-1,1}(1,2), 0, celdas_preliminares{1,i}(2,4),
celdas_obstaculos_ord{r,1}(1,5), 0];
                end
            end
            if(r==length(indices)+1)
                celdas_libres{r,1}=[celdas_preliminares{1,i}(1,1:4),
celdas_obstaculos_ord{r-1,1}(1,2), 0;
                                    celdas_preliminares{1,i}(2,1),
celdas_obstaculos_ord{r-1,1}(1,2), celdas_preliminares{1,i}(2,3:6)];
            end
        end
    else
        celdas_libres{1,1}=celdas_preliminares{1,i};
    end

    %Se actualiza la salida celdas_obstaculos que contendra todas las
    %celdas ocupadas por obstaculos
    if(i==1)
        celdas_obstaculos = celdas_obstaculos_ord;
    else
        if(size(celdas_obstaculos_ord,1)==size(celdas_obstaculos,1))
            celdas_obstaculos = [celdas_obstaculos,
celdas_obstaculos_ord];
        else
            if(size(celdas_obstaculos_ord,1)>size(celdas_obstaculos,1))
```

```
                celdas_obstaculos = [celdas_obstaculos;
cell(size(celdas_obstaculos_ord,1)-size(celdas_obstaculos,1),
size(celdas_obstaculos,2))];
                celdas_obstaculos = [celdas_obstaculos,
celdas_obstaculos_ord];
            else
                celdas_obstaculos_ord = [celdas_obstaculos_ord;
cell(size(celdas_obstaculos,1)-size(celdas_obstaculos_ord,1),1)];
                celdas_obstaculos = [celdas_obstaculos,
celdas_obstaculos_ord];
            end
        end
    end
    %Se actualiza la salida celdas_resultantes que contendra todas las
    %celdas libres
    if(i==1)
        celdas_resultantes = celdas_libres;
    else
        if(size(celdas_libres,1)==size(celdas_resultantes,1))
            celdas_resultantes = [celdas_resultantes, celdas_libres];
        else
            if(size(celdas_libres,1)>size(celdas_resultantes,1))
                celdas_resultantes = [celdas_resultantes;
cell(size(celdas_libres,1)-size(celdas_resultantes,1),
size(celdas_resultantes,2))];
                celdas_resultantes = [celdas_resultantes, celdas_libres];
            else
                celdas_libres = [celdas_libres;
cell(size(celdas_resultantes,1)-size(celdas_libres,1),1)];
                celdas_resultantes = [celdas_resultantes, celdas_libres];
            end
        end
    end
end
%-------------------------------------------------------------------------
--
%Calculo de los nodos de las celdas libres resultantes:
%-------------------------------------------------------------------------
--
for(j=1:size(celdas_resultantes,1)) %Se recorren las dimensiones de
celdas_resultantes
    for(i=1:size(celdas_resultantes,2))
        if(size(celdas_resultantes{j,i},1)~=0) %Si la celda actual no
esta vacia
            %Se distribuyen los nodos uniformemente en funcion de la
altura de cada celda. Si, por ejemplo, num_nodos = 5 y
            %(dimen_y_ent-o_y_ent) = 20, solo las celdas que posean una
altura entre 16 y 20 poseeran 5 nodos. Entre 12 y 16 poseeran 4. Entre 8
y 12 poseeran 3, etc.
            for(k=1:num_nodos)
                if((celdas_resultantes{j,i}(2,2)-
celdas_resultantes{j,i}(1,2)>(dimen_y_ent-o_y_ent)*(num_nodos-
k)/num_nodos)&(celdas_resultantes{j,i}(2,2)-
```

```
celdas_resultantes{j,i}(1,2)<=(dimen_y_ent-o_y_ent)*(num_nodos-
k+1)/num_nodos))
                    for(r=1:(num_nodos-k+1))
                        nodos = [nodos;

(celdas_resultantes{j,i}(1,1)+celdas_resultantes{j,i}(2,1))/2,
celdas_resultantes{j,i}(1,2)+(celdas_resultantes{j,i}(2,2)-
celdas_resultantes{j,i}(1,2))*(num_nodos-k+2-r)/(num_nodos-k+2), 0];
                    end
                end
            end
        end
    end
end
%-------------------------------------------------------------------------
--
%Representacion grafica
%-------------------------------------------------------------------------
--
figure;
hold on;
title("Descomposicion en celdas verticales del entorno mediante bandas");
xlabel("Dimension x (metros)");
ylabel("Dimension y (metros)");
axis([o_x_ent-1, dimen_x_ent+1, o_y_ent-1, dimen_y_ent+1]);
%Se recorre las dimensiones de celdas_resultantes para representar las
celdas libres
for(i=1:size(celdas_resultantes,2))
    for(j=1:size(celdas_resultantes,1))
        if(size(celdas_resultantes{j,i},1)~=0)
            patch([celdas_resultantes{j,i}(1,1),
celdas_resultantes{j,i}(1,4), celdas_resultantes{j,i}(2,1),
celdas_resultantes{j,i}(2,4)], [celdas_resultantes{j,i}(1,2),
celdas_resultantes{j,i}(1,5), celdas_resultantes{j,i}(2,2),
celdas_resultantes{j,i}(2,5)], [1 1 1]);
        end
    end
end
%Se recorre las dimensiones de celdas_obstaculos para representar las
celdas ocupadas
for(i=1:size(celdas_obstaculos,2))
    for(j=1:size(celdas_obstaculos,1))
        if(size(celdas_obstaculos{j,i},1)~=0)
            patch([celdas_obstaculos{j,i}(1,1),
celdas_obstaculos{j,i}(1,4), celdas_obstaculos{j,i}(2,1),
celdas_obstaculos{j,i}(2,4)], [celdas_obstaculos{j,i}(1,2),
celdas_obstaculos{j,i}(1,5), celdas_obstaculos{j,i}(2,2),
celdas_obstaculos{j,i}(2,5)], [1 0 0]);
        end
    end
end
%Se recorre la matriz de nodos para representarlos, como circulos
naranjas
for(i=1:length(nodos(:,1)))
```

```
    plot(nodos(i,1),nodos(i,2),'.','Color',[0.8500 0.3250 0.0980]);
end
%Se grafica, por ultimo, el entorno original:
dibuent(entorno,[0 0 1]);
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *banda_preparaciones_ruta.m*:

```
function [pts_en_area_libre, nodos, matriz_costes] =
banda_preparaciones_ruta(entorno, celdas_libres, celdas_obstaculos, subs,
longitud_celda, nodos, pt_in, pt_fin)
%Funcion que devuelve la matriz de costes y los nodos del grafo
resultante
%de las bandas verticales, para poder calcular rutas mediante Dijsktra.
%Requiere las celdas libres (celdas_libres) y las celdas ocupadas
%(celdas_obstaculos) resultado de las bandas, la longitud horizontal de
las
%celdas (longitud_celda), los nodos de las celdas libres (nodos) y los
%puntos inicial (pt_in) y final (pt_fin) de la ruta.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se declara matriz_costes y se actualiza la de nodos
matriz_costes = zeros(length(nodos(:,1))+2); %El tamaño de la matriz de
costes sera el numero de nodos mas el inicial y el final
%Se declaran variables que indicaran con un 1 que los puntos inicial y
%final se encuentran en el area libre. Valdran 0 en caso contrario.
pt_in_libre = 0;
pt_fin_libre = 0;
pts_en_area_libre = 0;
%Se extraen las dimensiones del entorno:
o_x_ent = min(entorno(:,1));
o_y_ent = min(entorno(:,2));
dimen_x_ent = max(entorno(:,1));
dimen_y_ent = max(entorno(:,2));
%En caso de que se haya introducido la longitud de la celda, se calcula
el
%numero de bandas y se recalcula la longitud de celda correspondiente
if(longitud_celda~=0)
    subs = ceil((dimen_x_ent-o_x_ent)/longitud_celda);
    longitud_celda = (dimen_x_ent-o_x_ent)/subs;
else
    %En caso de que se haya introducido el numero de bandas, recalcula la
    %longitud de la celda:
    if(subs~=0)
        longitud_celda = (dimen_x_ent-o_x_ent)/subs;
    end
end
%Se extrae de las celdas de obstaculos una matriz de segmentos que
codifica
%las celdas (se utilizara para comprobar la trazabilidad de las
conexiones entre nodos)
matriz_obstaculos = [];
for(i=1:size(celdas_obstaculos,2))
    for(j=1:size(celdas_obstaculos,1))
        if(size(celdas_obstaculos{j,i},1)~=0)
            matriz_obstaculos = [matriz_obstaculos;
                                 celdas_obstaculos{j,i}(1,1:6);
```

```
                                    celdas_obstaculos{j,i}(1,4:6),
celdas_obstaculos{j,i}(2,1:3);
                                    celdas_obstaculos{j,i}(2,1:6);
                                    celdas_obstaculos{j,i}(2,4:6),
celdas_obstaculos{j,i}(1,1:3)];
        end
    end
end
%Se comprueba que los puntos inicial y final se encuentren en el  area
libre
for(i=1:size(celdas_libres,2))
    for(j=1:size(celdas_libres,1))
        if(size(celdas_libres{j,i},1)~=0)

if((pt_in(1)>=celdas_libres{j,i}(1,1))&(pt_in(1)<=celdas_libres{j,i}(2,1)
)&(pt_in(2)>=celdas_libres{j,i}(1,2))&(pt_in(2)<=celdas_libres{j,i}(2,2))
)
                pt_in_libre = 1;
            end

if((pt_fin(1)>=celdas_libres{j,i}(1,1))&(pt_fin(1)<=celdas_libres{j,i}(2,
1))&(pt_fin(2)>=celdas_libres{j,i}(1,2))&(pt_fin(2)<=celdas_libres{j,i}(2
,2)))
                pt_fin_libre = 1;
            end
        end
    end
end
%Se representan los puntos inicial y final
p1=plot(pt_in(1), pt_in(2), 'xm', 'DisplayName','Punto inicial');
p2=plot(pt_fin(1), pt_fin(2), 'om', 'DisplayName','Punto final');
legend([p1 p2],'AutoUpdate','off');
%-------------------------------------------------------------------------
--
%Construccion de la matriz de costes y nodos (si procede)
%-------------------------------------------------------------------------
--
if((pt_in_libre==0)|(pt_fin_libre==0))
    disp("Alguno de los puntos, inicial o final, no se encuentran en el
area libre. Imposible calcular ruta");
else
    pts_en_area_libre = 1;

    %Calculo de costes entre los puntos inicial y final
    if((abs(pt_in(1)-pt_fin(1))<longitud_celda))
        arco_posible=1;
        for(i=1:length(matriz_obstaculos(:,1)))
            pc=inter_seg(pt_in(1:2), pt_fin(1:2),
matriz_obstaculos(i,1:2), matriz_obstaculos(i,4:5));
            if(pc(3)==1)
                arco_posible=0;
            end
        end
        if(arco_posible==1)
```

```
            matriz_costes(1,2) = sqrt((pt_in(1)-pt_fin(1))^2+(pt_in(2)-
pt_fin(2))^2);
            matriz_costes(2,1) = matriz_costes(1,2);
        end
    end

    %Calculo de coste entre los puntos inicial y final y los nodos
    for(i=1:length(nodos(:,1)))

        arco_posible=1;
        if(abs(pt_in(1)-nodos(i,1))<longitud_celda) %Punto inicial
            for(j=1:length(matriz_obstaculos(:,1)))
                pc_in=inter_seg(pt_in(1:2), nodos(i,1:2),
matriz_obstaculos(j,1:2), matriz_obstaculos(j,4:5));
                if(pc_in(3)==1)
                    arco_posible = 0;
                end
            end
            if(arco_posible==1)
                matriz_costes(1,i+2) = sqrt((pt_in(1)-
nodos(i,1))^2+(pt_in(2)-nodos(i,2))^2);
                matriz_costes(i+2,1) = matriz_costes(1,i+2);
            end
        end

        arco_posible=1;
        if(abs(pt_fin(1)-nodos(i,1))<longitud_celda) %Punto final
            for(j=1:length(matriz_obstaculos(:,1)))
                pc_fin=inter_seg(pt_fin(1:2), nodos(i,1:2),
matriz_obstaculos(j,1:2), matriz_obstaculos(j,4:5));
                if(pc_fin(3)==1)
                    arco_posible = 0;
                end
            end
            if(arco_posible==1)
                matriz_costes(2,i+2) = sqrt((pt_fin(1)-
nodos(i,1))^2+(pt_fin(2)-nodos(i,2))^2);
                matriz_costes(i+2,2) = matriz_costes(2,i+2);
            end
        end
    end

    %Se actualiza la matriz de nodos añadiendo los puntos inicial y final
al comienzo
    nodos = [pt_in; pt_fin; nodos];

    %Calculo de costes entre nodos
    for(i=3:length(matriz_costes(:,1)))
        for(j=3:length(matriz_costes(:,1)))
            if((abs(nodos(i,1)-nodos(j,1))>=longitud_celda-
longitud_celda*0.1)&(abs(nodos(i,1)-
nodos(j,1))<=longitud_celda+longitud_celda*0.1))
                arco_posible = 1;
                for(k=1:length(matriz_obstaculos(:,1)))
```

```
pc=inter_seg(nodos(i,1:2),nodos(j,1:2),matriz_obstaculos(k,1:2),matriz_ob
staculos(k,4:5));
                    if(pc(3)==1)
                        arco_posible = 0;
                    end
                end
                if(arco_posible==1)
                    matriz_costes(i,j) = sqrt((nodos(i,1)-
nodos(j,1))^2+(nodos(i,2)-nodos(j,2))^2);
                    matriz_costes(j,i) = matriz_costes(i,j);
                end
            end
        end
    end

    %Los costes entre nodos que no han sido considerados por las
    %condiciones previas deben ser inaccesibles entre si, es decir, coste
inf
    for(i=1:length(matriz_costes(:,1)))
        for(j=1:length(matriz_costes(1,:)))
            if(matriz_costes(i,j)==0)
                matriz_costes(i,j) = inf;
            end
        end
    end
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *celdas_verticales_configuracion_trapezoidal.m*:

```
function [segmentos_trapezoidal, nodos] =
celdas_verticales_configuracion_trapezoidal(entorno)
%Funcion que realiza una descomposicion en celdas por configuracion
%trapezoidal del entorno que se le introduce como entrada. Devuelve los
%segmentos que delimitan las celdas (segmentos_trapezoidal) y los nodos
%para calculo de rutas sobre estas celdas (nodos). AdemÃ¡s, realiza una
%representacion grafica de la descomposicion en celdas verticales.
%
%Requerimiento: el entorno debe estar definido en sentido horario.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se declaran las variables que almacenaran las salidas
segmentos_trapezoidal = [];
%Se extraen algunas variables del entorno
y_min_ent = min(entorno(:,2));
y_max_ent = max(entorno(:,2));
%Se extraen todos los vertices de los obstaculos del entorno:
vertices=entorno(5:length(entorno(:,1)),1:3);
%-------------------------------------------------------------------------
--
%Calculo de los segmentos que delimitan las celdas
%-------------------------------------------------------------------------
--
%Se recorren los vertices
for(i=1:length(vertices(:,1)))
    vertice_actual = vertices(i,:); %Se extrae el vertice de estudio
    %Se extrae los angulos que forman los segmentos que originan el
vertice
    [vertice_convexo, angulo_vertice, angulo1, angulo2] =
extrae_informacion_vertice(entorno, vertice_actual);

    %Calculo del segmento vertical hacia arriba (si es posible):
    %Condicion de posible:

if(((angulo1>=180)&(angulo1<=360)&(((angulo2>=180)&(angulo2<=360)&(angulo
2>angulo1))|((angulo2>=0)&(angulo2<=180))))|((angulo1>=0)&(angulo1<=180)&
(angulo2>=0)&(angulo2<=180)&(angulo1<angulo2)))

        %Se declara un segmento preliminar desde el vertice hasta el
limite por arriba del entorno
        segmento_arriba_preliminar = [vertice_actual, vertice_actual(1),
y_max_ent, 0];
        segmento_arriba = segmento_arriba_preliminar;
        %Se busca la interseccion del segmento preliminar con cualquier
        %segmento del entorno (se busca la interseccion con menor valor
en la coordenada vertical)
        minimo_actual=y_max_ent;
        for(j=1:length(entorno(:,1))) %Se recorren los segmentos
```

```
pc=inter_seg([segmento_arriba_preliminar(1:2)],[segmento_arriba_prelimina
r(4:5)],[entorno(j,1:2)],[entorno(j,4:5)]); %Interseccion entre segmentos
            if(pc(3)==1) %Si se produce interseccion
                if(pc(2)~=vertice_actual(2)) %Distinta al propio vertice
de origen
                    if(pc(2)<minimo_actual)
                        segmento_arriba(4:5)=pc(1:2); %Se actualiza el
punto final del segmento si la coordenada vertical es la mas pequeÃ±a
                        minimo_actual=pc(2);
                    end
                end
            end
        end
        %Se actualiza la matriz segmentos_trapezoidal, que posee todos
los segmentos
        segmentos_trapezoidal = [segmentos_trapezoidal; segmento_arriba];
    end
    %Calculo del segmento vertical hacia abajo (si es posible):
    %Condicion de posible:
    if(angulo1<=angulo2)
        %Se declara un segmento preliminar desde el vertice hasta el
limite por abajo del entorno
        segmento_abajo_preliminar = [vertice_actual, vertice_actual(1),
y_min_ent, 0];
        segmento_abajo = segmento_abajo_preliminar;
        %Se busca la interseccion del segmento preliminar con cualquier
        %segmento del entorno (se busca la interseccion con mayor valor
en la coordenada vertical)
        maximo_actual=y_min_ent;
        for(j=1:length(entorno(:,1))) %Se recorren los segmentos

pc=inter_seg([segmento_abajo_preliminar(1:2)],[segmento_abajo_preliminar(
4:5)],[entorno(j,1:2)],[entorno(j,4:5)]); %Interseccion entre segmentos
            if(pc(3)==1) %Si se produce interseccion
                if(pc(2)~=vertice_actual(2)) %Distinta al propio vertice
de origen
                    if(pc(2)>maximo_actual)
                        segmento_abajo(4:5)=pc(1:2); %Se actualiza el
punto final del segmento si la coordenada vertical es la mas grande
                        maximo_actual=pc(2);
                    end
                end
            end
        end
        %Se actualiza la matriz segmentos_trapezoidal, que posee todos
los segmentos
        segmentos_trapezoidal = [segmentos_trapezoidal; segmento_abajo];
    end
end
%Es requerida una comprobacion final.
%Debido a las operaciones llevadas a cabo para calcular los segmentos, es
%posible que los lados completamente verticales de los obstaculos sean
```

```
%identificados tambien como segmentos que delimitan las celdas. Se
procede
%a eliminar estos segmentos:
for(i=5:length(entorno(:,1))) %Se recorren todos los segmentos de los
obstaculos en busca de segmentos verticales
    if((entorno(i,4)-entorno(i,1))==0)
        segmento_vertical = entorno(i,1:6); %Se extrae el segmento
vertical
        %Se busca coincidencia del segmento en segmentos_trapezoidal
        j=1;
        while(j<=length(segmentos_trapezoidal(:,1))) %Se recorre
segmentos_trapezoidal

if(((segmentos_trapezoidal(j,1)==segmento_vertical(1))&((segmentos_trapez
oidal(j,2)==segmento_vertical(2)))&((segmentos_trapezoidal(j,4)==segmento
_vertical(4)))&((segmentos_trapezoidal(j,5)==segmento_vertical(5))))|((se
gmentos_trapezoidal(j,1)==segmento_vertical(4))&((segmentos_trapezoidal(j
,2)==segmento_vertical(5)))&((segmentos_trapezoidal(j,4)==segmento_vertic
al(1)))&((segmentos_trapezoidal(j,5)==segmento_vertical(2)))))
                segmentos_trapezoidal(j,:) = []; %Si se produce
coincidencia, se elimina el segmento del resultado
            end
            j=j+1;
        end
    end
end
%Se extraen todos los nodos por los que se puede trazar una ruta:
nodos = [(min(entorno(:,1))+min(segmentos_trapezoidal(:,1)))/2,
(min(entorno(:,2))+max(entorno(:,2)))/2, 0]; %Se añade un nodo inical de
la primera celda
for(i=1:length(segmentos_trapezoidal(:,1)))
    nodos = [nodos; segmentos_trapezoidal(i,1),
(segmentos_trapezoidal(i,2)+segmentos_trapezoidal(i,5))/2, 0];
end
nodos = [nodos; (max(segmentos_trapezoidal(:,1))+max(entorno(:,1)))/2,
(min(entorno(:,2))+max(entorno(:,2)))/2, 0]; %Se añade un nodo final de
la ultima celda
%-------------------------------------------------------------------------
--
%Representacion grafica:
%-------------------------------------------------------------------------
--
figure;
hold on;
title("Descomposicion en celdas verticales por configuracion
trapezoidal");
xlabel("Dimension x (metros)");
ylabel("Dimension y (metros)");
axis([min(entorno(:,1))-1, max(entorno(:,1))+1, y_min_ent-1,
y_max_ent+1]);
%Se grafica el entorno original y los segmentos trapezoidales:
dibuent(entorno,[0 0 1]);
dibuent(segmentos_trapezoidal,[0 0 0]);
%Se marca los vertices de los obstaculos mediante una cruz verde
```

```
for(i=1:length(vertices(:,1)))
    plot(vertices(i,1),vertices(i,2),'xg');
end
%Se representa tambien los nodos de los segmentos como circulos naranjas:
for(i=1:length(nodos(:,1)))
    plot(nodos(i,1),nodos(i,2),'.','Color',[0.8500 0.3250 0.0980]);
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *trapezoidal_preparaciones_ruta.m*:

```
function [pts_en_area_libre, nodos, matriz_costes] =
trapezoidal_preparaciones_ruta(entorno, segmentos_trapezoidal, nodos,
pt_in, pt_fin)
%Funcion que devuelve la matriz de costes y los nodos del grafo
resultante
%de la configuracion trapezoidal vertical, para poder calcular rutas
%mediante Dijsktra o A*. Requiere los segmentos resultado de la
%descomposicion, los nodos de las celdas libres (nodos), los
%puntos inicial (pt_in) y final (pt_fin) de la ruta y el entorno
original.
%Calculos preliminares:
%-------------------------------------------------------------------
--
%Se declara matriz_costes
matriz_costes = zeros(length(nodos(:,1))+2); %El tamaño de la matriz de
costes sera el numero de nodos mas el inicial y el final
%Se extrae la dimension x del entorno:
dimen_x_ent = max(entorno(:,1));
%Se representan los puntos inicial y final
p1=plot(pt_in(1), pt_in(2), 'xm', 'DisplayName','Punto inicial');
p2=plot(pt_fin(1), pt_fin(2), 'om', 'DisplayName','Punto final');
legend([p1 p2],'AutoUpdate','off');
%-------------------------------------------------------------------
--
%Construccion de la matriz de costes y nodos
%-------------------------------------------------------------------
--
%Calculo de costes entre nodos (incluyendo punto inicial y final):
nodos=[pt_in; pt_fin; nodos]; %Se actualiza la matriz de nodos
introduciendo el punto inicial y final los dos primeros
for(i=1:length(nodos(:,1)))
    %Para cada nodo de estudio, se busca los nodos mas cercanos en la
dimension x
    %Se extrae la minima distancia por la izquierda (negativa) y por la
derecha (positiva)
    distancia_minima_positiva = dimen_x_ent;
    distancia_minima_negativa = -dimen_x_ent;

    for(j=1:length(nodos(:,1)))
       %Se comprueba que el arco sea realizable
        arco_posible=1;
        for(k=5:length(entorno(:,1)))

pc=inter_seg(nodos(i,1:2),nodos(j,1:2),entorno(k,1:2),entorno(k,4:5));
            if(pc(3)==1)
                arco_posible=0;
            end
        end
        if(arco_posible==1)
            if((nodos(j,1)-nodos(i,1)<0)&(nodos(j,1)-
nodos(i,1)>=distancia_minima_negativa))
```

```
                distancia_minima_negativa=nodos(j,1)-nodos(i,1);
            end
            if((nodos(j,1)-nodos(i,1)>0)&(nodos(j,1)-
nodos(i,1)<=distancia_minima_positiva))
                distancia_minima_positiva=nodos(j,1)-nodos(i,1);
            end
        end
    end


    %Aquellos nodos que cumplan la condicion de distancia minima a la
    %izquierda o a la derecha, se comprueba si su arco cruza algun
    %obstaculo. Si no cruza ninguno, el arco es realizable y tiene un
    %coste asociado distinto de inf.
    for(j=1:length(nodos(:,1)))
        if((nodos(j,1)-
nodos(i,1)>=distancia_minima_negativa+distancia_minima_negativa*0.1)&(nod
os(j,1)-nodos(i,1)<=distancia_minima_negativa-
distancia_minima_negativa*0.1)) %Puntos a la izquierda
            arco_posible = 1;
            for(k=5:length(entorno(:,1)))

pc=inter_seg(nodos(i,1:2),nodos(j,1:2),entorno(k,1:2),entorno(k,4:5));
                if(pc(3)==1)
                    arco_posible = 0;
                end
            end
            if(arco_posible==1)
                matriz_costes(i,j) = sqrt((nodos(i,1)-
nodos(j,1))^2+(nodos(i,2)-nodos(j,2))^2);
            end
        end

        if((nodos(j,1)-nodos(i,1)>=distancia_minima_positiva-
distancia_minima_positiva*0.1)&(nodos(j,1)-
nodos(i,1)<=distancia_minima_positiva+distancia_minima_positiva*0.1))
%Puntos a la derecha
            arco_posible = 1;
            for(k=5:length(entorno(:,1)))

pc=inter_seg(nodos(i,1:2),nodos(j,1:2),entorno(k,1:2),entorno(k,4:5));
                if(pc(3)==1)
                    arco_posible = 0;
                end
            end
            if(arco_posible==1)
                matriz_costes(i,j) = sqrt((nodos(i,1)-
nodos(j,1))^2+(nodos(i,2)-nodos(j,2))^2);
            end
        end
    end
end
%Aquellos costes entre nodos no considerados hasta el momento y que sean
```

```
%nulos en la matriz de costes en realidad no son realizables, por lo que
se
%cambia su coste a inf
for(i=1:length(matriz_costes(:,1)))
    for(j=1:length(matriz_costes(:,1)))
        if(matriz_costes(i,j)==0)
            matriz_costes(i,j)=inf;
        end
    end
end
%Por ultimo, se extrae de matriz_conclusiones conclusiones para la salida
%pts_en_area_libre:
%Si no poseen conexion con ningun nodo, los puntos inicial y/o final se
encuentran en el area ocupada
pt_in_libre = 0;
pt_fin_libre = 0;
for(i=3:length(matriz_costes(:,1)))
    if(matriz_costes(1,i)~=inf)
        pt_in_libre = 1;
    end
    if(matriz_costes(2,i)~=inf)
        pt_fin_libre = 1;
    end
end
if((pt_in_libre==1)&(pt_fin_libre==1))
    pts_en_area_libre = 1;
else
    pts_en_area_libre = 0;
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *celdas_uniformes.m*:

```
function [celdas_un, matriz_un, subs_x, subs_y] =
celdas_uniformes(entorno, subs_x, subs_y, longitud_celda, altura_celda)
%Funcion que descompone en celdas uniformes un entorno. Requiere como
%argumentos de entrada el entornor a descomponer (entorno), el numero de
%subdivisiones en columnas (subs_x) o la longitud de las celdas
%(longitud_celda), y el numero de subdivisiones en dilas (subs_y) o la
%altura de las celdas (altura_celda).
%Devuelve las celdas (celdas_un) como matrices que alojan sus vertices y
%una matriz (matriz_un) que codifica el estado de la celda que posee sus
%mismos indices. Un valor de -2 codifica una celda ocupada, mientras que
%una celda libre se representa por un 0.
%Calculos Preliminares:
%-------------------------------------------------------------------------
--
%Dimensiones del entorno:
o_x_ent = min(entorno(:,1));
o_y_ent = min(entorno(:,2));
dimen_x_ent = max(entorno(:,1));
dimen_y_ent = max(entorno(:,2));
%En caso de que se haya introducido la longitud de la celda, se calcula
el
%numero de divisiones en celdas en horizontal y se recalcula la longitud
de
%celda correspondiente
if(longitud_celda~=0)
    subs_x = ceil((dimen_x_ent-o_x_ent)/longitud_celda);
    longitud_celda = (dimen_x_ent-o_x_ent)/subs_x;
else
    %En caso de que se haya introducido las divisiones horizontales en
    %celdas, recalcula la longitud de la celda:
    if(subs_x~=0)
        longitud_celda = (dimen_x_ent-o_x_ent)/subs_x;
    end
end
%En caso de que se haya introducido la altura de la celda, se calcula el
%numero de divisiones en celdas en vertical y se recalcula la altura de
%celda correspondiente
if(altura_celda~=0)
    subs_y = ceil((dimen_y_ent-o_y_ent)/altura_celda);
    altura_celda = (dimen_y_ent-o_y_ent)/subs_y;
else
    %En caso de que se haya introducido las divisiones horizontales en
    %celdas, recalcula la longitud de la celda:
    if(subs_y~=0)
        altura_celda = (dimen_y_ent-o_y_ent)/subs_y;
    end
end
%Se construye un arreglo cell que posee la descomposicion del entorno en
%celdas uniformes, con las coordenadas que las identifican:
celdas_un = cell(subs_y,subs_x);
for(i=1:subs_y)
```

```
    for(j=1:subs_x)
        celdas_un{i,j} = [o_x_ent+round((j-1)*longitud_celda,5),
o_y_ent+round((i-1)*altura_celda,5), 0, o_x_ent+round((j-
1)*longitud_celda,5), o_y_ent+round(i*altura_celda,5),     0;
                          o_x_ent+round(j*longitud_celda,5),
o_y_ent+round(i*altura_celda,5),     0,
o_x_ent+round(j*longitud_celda,5),     o_y_ent+round((i-
1)*altura_celda,5), 0];
    end
end
%Se construye una matriz que posee un valor que corresponde a cada celda,
%para su misma posicion (i,j)
matriz_un = zeros(subs_y,subs_x);
%Representacion grafica preliminar
figure
hold on
axis([-1 dimen_x_ent+1 -1 dimen_y_ent+1]);
title("Descomposicion del entorno en celdas uniformes");
xlabel("Dimension x (metros)");
ylabel("Dimension y (metros)");
for(i=1:subs_y)
    for(j=1:subs_x)
        patch([celdas_un{i,j}(1,1) celdas_un{i,j}(1,1)
celdas_un{i,j}(2,1) celdas_un{i,j}(2,1)],[celdas_un{i,j}(1,2)
celdas_un{i,j}(2,2) celdas_un{i,j}(2,2) celdas_un{i,j}(1,2)],'w');
    end
end
dibuent(entorno,[0 0 1]);
%-------------------------------------------------------------------------
--
%Delimitacion de los obstaculos:
%-------------------------------------------------------------------------
--
for(i=1:max(entorno(:,8))) %Para cada obstaculo del entorno
    obstaculo_actual=extrae_obstaculo(i,entorno); %Se extrae el obstaculo
actual
    %Se calculan las coordenadas de las celdas que delimitarian al
obstaculo
    %formando un rectangulo base que lo contiene (Bounding Box)
    max_x = max(obstaculo_actual(:,1));
    celda_x_max = ceil(max_x/dimen_x_ent*subs_x);
    if(celda_x_max+1<subs_x)
        celda_x_max=celda_x_max+1;
    end
    min_x = min(obstaculo_actual(:,1));
    celda_x_min = floor(min_x/dimen_x_ent*subs_x);
    if(celda_x_min-1>1)
        celda_x_min=celda_x_min-1;
    end
    max_y = max(obstaculo_actual(:,2));
    celda_y_max = ceil(max_y/dimen_y_ent*subs_y);
    if(celda_y_max+1<subs_y)
        celda_y_max=celda_y_max+1;
    end
```

```
    min_y = min(obstaculo_actual(:,2));
    celda_y_min = floor(min_y/dimen_y_ent*subs_y);
    if(celda_y_min-1>1)
        celda_y_min=celda_y_min-1;
    end
    %Se suma o resta uno al rectangulo para analizar un area de mayor
seguridad

    %Para cada celda de la Bounding Box
    for(j=celda_x_min:celda_x_max)
        for(k=celda_y_min:celda_y_max)
            if(celda_ocupada_parcial(celdas_un{k,j},entorno)) %Se
comprueba si esta ocupada parcialmente
                matriz_un(k,j) = -2;
            end
            if(matriz_un(k,j)==0) %Condicion de eficiencia: no es
necesario comprobar si esta ocupada integramente si ya esta ocupada
parcialmente
                if(celda_ocupada_integra(celdas_un{k,j},entorno)) %Se
comprueba si esta ocupada integramente
                    matriz_un(k,j) = -2;
                end
            end
        %Se utiliza el -2 para designar una celda ocupada
        end
    end
end
%Representacion grafica de las celdas ocupadas
for(i=1:subs_y)
    for(j=1:subs_x)
        if(matriz_un(i,j)==-2)
            patch([celdas_un{i,j}(1,1) celdas_un{i,j}(1,1)
celdas_un{i,j}(2,1) celdas_un{i,j}(2,1)],[celdas_un{i,j}(1,2)
celdas_un{i,j}(2,2) celdas_un{i,j}(2,2) celdas_un{i,j}(1,2)],'y');
        end
    end
end
dibuent(entorno,[0 0 1]);
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *ruta_celdas_uniformes_campo_potencial.m*:

```
function [matriz_campo, ruta, coste_total] =
ruta_celdas_uniformes_campo_potencial(celdas_un, matriz_un, pt_in,
pt_fin, subs_x, subs_y, observar_campo)
%Funcion que expande un campo potencial desde el punto inicial hasta el
%final en un entorno descompuesto en celdas uniformes y calcula la ruta
de
%menor coste. Requiere como argumentos de entrada el arreglo tipo cell
que
%codifica las celdas (celdas_un), la matriz que codifica sus estados
%(matriz_un), el punto inicial de la ruta (pt_in), el punto final de la
%ruta (pt_fin), el numero de subdivisiones en x o numero de columnas
%(subs_x), el numero de subdivisiones en y o numero de filas (subs_y) y
un
%indicardor de 1 o 0 para observar o no, respectivamente, la animacion de
%expansion del campo potencial.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se extrae los indices en los que se encuentra los puntos inicial y final
for(i=1:length(matriz_un(:,1)))
    for(j=1:length(matriz_un(1,:)))

if((pt_in(1)>=celdas_un{i,j}(1,1))&(pt_in(1)<=celdas_un{i,j}(2,1))&(pt_in
(2)>=celdas_un{i,j}(1,2))&(pt_in(2)<=celdas_un{i,j}(2,2)))
            indices_in = [i, j];
        end

if((pt_fin(1)>=celdas_un{i,j}(1,1))&(pt_fin(1)<=celdas_un{i,j}(2,1))&(pt_
fin(2)>=celdas_un{i,j}(1,2))&(pt_fin(2)<=celdas_un{i,j}(2,2)))
            indices_fin = [i, j];
        end
    end
end
%Se declara una matriz que poseera la informacion del campo
matriz_campo = [];
%Se comprueba que las celdas en las que se encuentran los puntos inicial
y final no esten en area ocupada:
if(matriz_un(indices_in(1),indices_in(2))==-2)
    disp("El punto inicial no se encuentra en el area libre.")
    return
else
    matriz_un(indices_in(1),indices_in(2)) = -1;
    patch([celdas_un{indices_in(1),indices_in(2)}(1,1),
celdas_un{indices_in(1),indices_in(2)}(1,4),
celdas_un{indices_in(1),indices_in(2)}(2,1),
celdas_un{indices_in(1),indices_in(2)}(2,4)],...
          [celdas_un{indices_in(1),indices_in(2)}(1,2),
celdas_un{indices_in(1),indices_in(2)}(1,5),
celdas_un{indices_in(1),indices_in(2)}(2,2),
celdas_un{indices_in(1),indices_in(2)}(2,5)],...
          [1 0 1]);
```

```
    p1=plot(pt_in(1),pt_in(2),'x','Color',[0 1 0],'DisplayName','Punto
inicial');
end
if(matriz_un(indices_fin(1),indices_fin(2))==-2)
    disp("El punto final no se encuentra en el area libre.")
    return
else
    matriz_un(indices_fin(1),indices_fin(2)) = -1;
    patch([celdas_un{indices_fin(1),indices_fin(2)}(1,1),
celdas_un{indices_fin(1),indices_fin(2)}(1,4),
celdas_un{indices_fin(1),indices_fin(2)}(2,1),
celdas_un{indices_fin(1),indices_fin(2)}(2,4)],...
          [celdas_un{indices_fin(1),indices_fin(2)}(1,2),
celdas_un{indices_fin(1),indices_fin(2)}(1,5),
celdas_un{indices_fin(1),indices_fin(2)}(2,2),
celdas_un{indices_fin(1),indices_fin(2)}(2,5)],...
          [1 0 1]);
    p2=plot(pt_fin(1),pt_fin(2),'o','Color',[0 1 0],'DisplayName','Punto
final');
end
legend([p1 p2],'AutoUpdate','Off');
%Se utiliza el -1 para designar los puntos inicial y final
%-------------------------------------------------------------------------
--
%Extension del campo potencial:
%-------------------------------------------------------------------------
--
i=1;
destino_alcanzado=0;
incremento_coste = 1/max(subs_x,subs_y);
%El maximo nivel de expansion posible (no necesario) del campo potencial
sera el
%numero maximo de divisiones en x o en y. Ademas, la expansion parara si
el campo se extiende hasta el punto de destino
while((i<=max(subs_x,subs_y))&(destino_alcanzado==0))

    %Pausa si se desea observar como se expande el campo potencial
    if(observar_campo==1)
        pause(0.2);
    end
    %Extension del campo por la izquierda:
    if(indices_in(1)-i>=1)
        for(j=0:i*2)
            if((indices_in(2)-i+j>=1)&(indices_in(2)-i+j<=subs_y))
                if(matriz_un(indices_in(1)-i,indices_in(2)-i+j)==0)
                    matriz_campo(indices_in(1)-i,indices_in(2)-
i+j)=i*incremento_coste;
                    patch([celdas_un{indices_in(1)-i,indices_in(2)-
i+j}(1,1), celdas_un{indices_in(1)-i,indices_in(2)-i+j}(1,4),
celdas_un{indices_in(1)-i,indices_in(2)-i+j}(2,1),
celdas_un{indices_in(1)-i,indices_in(2)-i+j}(2,4)],...
                          [celdas_un{indices_in(1)-i,indices_in(2)-
i+j}(1,2), celdas_un{indices_in(1)-i,indices_in(2)-i+j}(1,5),
```

```
celdas_un{indices_in(1)-i,indices_in(2)-i+j}(2,2),
celdas_un{indices_in(1)-i,indices_in(2)-i+j}(2,5)],...
                            [0 1-i*incremento_coste 1-i*incremento_coste]);
                end
                if(matriz_un(indices_in(1)-i,indices_in(2)-i+j)==-1)
                    matriz_campo(indices_in(1)-i,indices_in(2)-
i+j)=i*incremento_coste;
                    destino_alcanzado=1;
                    coste_total = i*incremento_coste*max(subs_x,subs_y);
                end
            end
        end
    end

    %Extension del campo por la derecha
    if(indices_in(1)+i<=subs_x)
        for(j=0:i*2)
            if((indices_in(2)-i+j>=1)&(indices_in(2)-i+j<=subs_y))
                if(matriz_un(indices_in(1)+i,indices_in(2)-i+j)==0)
                    matriz_campo(indices_in(1)+i,indices_in(2)-
i+j)=i*incremento_coste;
                    patch([celdas_un{indices_in(1)+i,indices_in(2)-
i+j}(1,1), celdas_un{indices_in(1)+i,indices_in(2)-i+j}(1,4),
celdas_un{indices_in(1)+i,indices_in(2)-i+j}(2,1),
celdas_un{indices_in(1)+i,indices_in(2)-i+j}(2,4)],...
                            [celdas_un{indices_in(1)+i,indices_in(2)-
i+j}(1,2), celdas_un{indices_in(1)+i,indices_in(2)-i+j}(1,5),
celdas_un{indices_in(1)+i,indices_in(2)-i+j}(2,2),
celdas_un{indices_in(1)+i,indices_in(2)-i+j}(2,5)],...
                            [0 1-i*incremento_coste 1-i*incremento_coste]);
                end
                if(matriz_un(indices_in(1)+i,indices_in(2)-i+j)==-1)
                    matriz_campo(indices_in(1)+i,indices_in(2)-
i+j)=i*incremento_coste;
                    destino_alcanzado=1;
                    coste_total = i*incremento_coste*max(subs_x,subs_y);
                end
            end
        end
    end

    %Extension del campo por arriba
    if(indices_in(2)+i<=subs_y)
        for(j=0:i*2)
            if((indices_in(1)-i+j>=1)&(indices_in(1)-i+j<=subs_x))
                if(matriz_un(indices_in(1)-i+j,indices_in(2)+i)==0)
                    matriz_campo(indices_in(1)-
i+j,indices_in(2)+i)=i*incremento_coste;
                    patch([celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(1,1), celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(1,4), celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(2,1), celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(2,4)],...
```

```
                        [celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(1,2), celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(1,5), celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(2,2), celdas_un{indices_in(1)-
i+j,indices_in(2)+i}(2,5)],...
                        [0 1-i*incremento_coste 1-i*incremento_coste]);
                end
                if(matriz_un(indices_in(1)-i+j,indices_in(2)+i)==-1)
                    matriz_campo(indices_in(1)-
i+j,indices_in(2)+i)=i*incremento_coste;
                    destino_alcanzado=1;
                    coste_total = i*incremento_coste*max(subs_x,subs_y);
                end
            end
        end
    end

    %Extension del campo por abajo
    if(indices_in(2)-i>=1)
        for(j=0:i*2)
            if((indices_in(1)-i+j>=1)&(indices_in(1)-i+j<=subs_x))
                if(matriz_un(indices_in(1)-i+j,indices_in(2)-i)==0)
                    matriz_campo(indices_in(1)-i+j,indices_in(2)-
i)=i*incremento_coste;
                    patch([celdas_un{indices_in(1)-i+j,indices_in(2)-
i}(1,1), celdas_un{indices_in(1)-i+j,indices_in(2)-i}(1,4),
celdas_un{indices_in(1)-i+j,indices_in(2)-i}(2,1),
celdas_un{indices_in(1)-i+j,indices_in(2)-i}(2,4)],...
                        [celdas_un{indices_in(1)-i+j,indices_in(2)-
i}(1,2), celdas_un{indices_in(1)-i+j,indices_in(2)-i}(1,5),
celdas_un{indices_in(1)-i+j,indices_in(2)-i}(2,2),
celdas_un{indices_in(1)-i+j,indices_in(2)-i}(2,5)],...
                        [0 1-i*incremento_coste 1-i*incremento_coste]);
                end
                if(matriz_un(indices_in(1)-i+j,indices_in(2)-i)==-1)
                    matriz_campo(indices_in(1)-i+j,indices_in(2)-
i)=i*incremento_coste;
                    destino_alcanzado=1;
                    coste_total = i*incremento_coste*max(subs_x,subs_y);
                end
            end
        end
    end
    i=i+1;
end
%-------------------------------------------------------------------------
--
%Calculo de la ruta:
%-------------------------------------------------------------------------
--
%Se declaran alguinas variables para hacer uso de la funcion recursiva
que
%explora el campo potencial
ruta_preliminar = indices_fin;
```

```
destino_alcanzado=0;
indices_celda_origen=indices_in;
[ruta_preliminar, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado,
ruta_preliminar, celdas_un, matriz_campo, matriz_un,
indices_celda_origen, subs_x, subs_y);
%La ruta preliminar esta construida desde la celda final hasta la
original,
%asi que se le da la vuelta
ruta=[];
for(i=1:length(ruta_preliminar(:,1)))
    ruta = [ruta; ruta_preliminar(length(ruta_preliminar(:,1))+1-i,:)];
end
%Se recorre la ruta y se representa:
for(i=2:length(ruta(:,1))-1)
    patch([celdas_un{ruta(i,1),ruta(i,2)}(1,1),
celdas_un{ruta(i,1),ruta(i,2)}(1,4), celdas_un{ruta(i,1),ruta(i,2)}(2,1),
celdas_un{ruta(i,1),ruta(i,2)}(2,4)],...
          [celdas_un{ruta(i,1),ruta(i,2)}(1,2),
celdas_un{ruta(i,1),ruta(i,2)}(1,5), celdas_un{ruta(i,1),ruta(i,2)}(2,2),
celdas_un{ruta(i,1),ruta(i,2)}(2,5)],...
          [1 0 1]);
end
title("Descomposicion del entorno en celdas uniformes. Ruta sobre campo
potencial");
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *celdas_uniformes_explorar_campo_potencial.m*:

```
function [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x, subs_y)
%Subfuncion (recursiva) que recibe el campo potencial expandido sobre las
%celdasuniformes y halla la ruta de menor coste entre el nodo inicial y
el
%final.
if(destino_alcanzado==0) %Condicion de recursividad
%Se saca un vector unitario que va desde el punto centro de la celda
actual hasta el punto centro de la celda de origen
pt_cent_cell_act =
[(celdas_un{ruta(length(ruta(:,1)),1),ruta(length(ruta(:,1)),2)}(1,1)+cel
das_un{ruta(length(ruta(:,1)),1),ruta(length(ruta(:,1)),2)}(2,1))/2,...

(celdas_un{ruta(length(ruta(:,1)),1),ruta(length(ruta(:,1)),2)}(1,2)+celd
as_un{ruta(length(ruta(:,1)),1),ruta(length(ruta(:,1)),2)}(2,2))/2, 0];
%Punto centro celda actual
pt_cent_cell_or =
[(celdas_un{indices_celda_origen(1),indices_celda_origen(2)}(1,1)+celdas_
un{indices_celda_origen(1),indices_celda_origen(2)}(2,1))/2,...

(celdas_un{indices_celda_origen(1),indices_celda_origen(2)}(1,2)+celdas_u
n{indices_celda_origen(1),indices_celda_origen(2)}(2,2))/2, 0]; %Punto
centro celda origen
vect_act_a_or = [pt_cent_cell_or(1)-pt_cent_cell_act(1),
pt_cent_cell_or(2)-pt_cent_cell_act(2), 0]; %Vector celda actual a celda
origen
mod_vect_act_a_or = sqrt((vect_act_a_or(1))^2+(vect_act_a_or(2))^2);
%Modulo del vector celda actual a celda origen
vect_act_a_or_unit = [vect_act_a_or(1)/mod_vect_act_a_or,
vect_act_a_or(2)/mod_vect_act_a_or, 0]; %Vector unitario celda actual a
celda origen
%Con el vector unitario obtenido, se puede calcular el angulo que forma
con
%la horizontal para obtener a que direccion se debe apuntar en la
%exploracion del campo
if(vect_act_a_or_unit(1)==0)
    if(vect_act_a_or_unit(2)>0)
        beta = pi/2;
    else
        beta = 3*pi/2;
    end
else
    beta0 = atan(vect_act_a_or_unit(2)/vect_act_a_or_unit(1));
    if(beta0==0)
        if(vect_act_a_or_unit(1)>0)
            beta = 0;
        else
            beta = pi;
        end
    else
```

```
        if((vect_act_a_or_unit(1)<0)&(vect_act_a_or_unit(2)<0))
            beta = pi+beta0;
        else
            if((vect_act_a_or_unit(1)<0)&(vect_act_a_or_unit(2)>0))
                beta = pi+beta0;
            else
                if((vect_act_a_or_unit(1)>0)&(vect_act_a_or_unit(2)<0))
                    beta = 2*pi+beta0;
                else
                    beta = beta0;
                end
            end
        end
    end
end
%Se extrae la celda de la que se viene para que no se repita
if(length(ruta(:,1))>1)
    indices_previos = [ruta(length(ruta(:,1))-1,1)-
ruta(length(ruta(:,1)),1), ruta(length(ruta(:,1))-1,2)-
ruta(length(ruta(:,1)),2)];
    if(indices_previos(1)==0)
        if(indices_previos(2)==1)
            vecino_previo=1;
        end
        if(indices_previos(2)==-1)
            vecino_previo=5;
        end
    else
        if(indices_previos(1)==1)
            if(indices_previos(2)==1)
                vecino_previo=2;
            end
            if(indices_previos(2)==0)
                vecino_previo=3;
            end
            if(indices_previos(2)==-1)
                vecino_previo=4;
            end
        else
            if(indices_previos(2)==1)
                vecino_previo=8;
            end
            if(indices_previos(2)==0)
                vecino_previo=7;
            end
            if(indices_previos(2)==-1)
                vecino_previo=6;
            end
        end
    end
else
    vecino_previo=0;
end
```

```
%Se ordenan las direcciones de los vecinos de la celda actual en funcion
de
%que direccion queda mas cerca de la direccion del vector que apunta al
origen
diferencias_angulos_vecinos = [abs(0-beta), abs(pi/4-beta), abs(pi/2-
beta), abs(3*pi/4-beta), abs(pi-beta), abs(5*pi/4-beta), abs(3*pi/2-
beta), abs(7*pi/4-beta), abs(2*pi-beta);
                                1, 2, 3, 4, 5, 6, 7, 8, 1];
orden_vecinos = [];
for(i=1:5)
    [valor, indice] = min(diferencias_angulos_vecinos(1,:));
    orden_vecinos = [orden_vecinos;
diferencias_angulos_vecinos(2,indice)];
    diferencias_angulos_vecinos(:,indice)=[];
end
%Del 1 al 8, indican los vecinos de la celda actual comenzado por la que
%esta inmediatamente a la derecha y prosiguiendo en sentido antihoria.
Asi
%pues, el 6, por ejemplo, sería la celda inmediatamente abajo.
%Se explora los vecinos en orden de proximidad, en terminos de direccion
i=1;
while((i<=5)&(destino_alcanzado==0))
    if(orden_vecinos(i)~=vecino_previo) %Si el vecino no es el vecino del
que viene la ruta
        proximo_vecino = orden_vecinos(i);
    else
        proximo_vecino = 0;
    end
    if(proximo_vecino==1)
        indices_siguientes = [ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2)+1]; %Se extraen los indices y se comprueba que
son posibles
        if(indices_siguientes(2)<=subs_x)
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
                end
            end
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
```

```
        end
    end
    if(proximo_vecino==2)
        indices_siguientes = [ruta(length(ruta(:,1)),1)+1,
ruta(length(ruta(:,1)),2)+1]; %Se extraen los indices y se comprueba que
son posibles

if((indices_siguientes(1)<=subs_x)&(indices_siguientes(2)<=subs_y))
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
                end
            end
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
        end
    end
    if(proximo_vecino==3)
        indices_siguientes = [ruta(length(ruta(:,1)),1)+1,
ruta(length(ruta(:,1)),2)]; %Se extraen los indices y se comprueba que
son posibles
        if(indices_siguientes(1)<=subs_y)
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
                end
            end
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
```

```
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
        end
    end
    if(proximo_vecino==4)
        indices_siguientes = [ruta(length(ruta(:,1)),1)+1,
ruta(length(ruta(:,1)),2)-1]; %Se extraen los indices y se comprueba que
son posibles
        if((indices_siguientes(2)>=1)&(indices_siguientes(1)<=subs_y))
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
                end
            end
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
        end
    end
    if(proximo_vecino==5)
        indices_siguientes = [ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2)-1]; %Se extraen los indices y se comprueba que
son posibles
        if(indices_siguientes(2)>=1)
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
                end
            end
```

```
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
        end
    end
    if(proximo_vecino==6)
        indices_siguientes = [ruta(length(ruta(:,1)),1)-1,
ruta(length(ruta(:,1)),2)-1]; %Se extraen los indices y se comprueba que
son posibles
        if((indices_siguientes(1)>=1)&(indices_siguientes(2)>=1))
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
                end
            end
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
        end
    end
    if(proximo_vecino==7)
        indices_siguientes = [ruta(length(ruta(:,1)),1)-1,
ruta(length(ruta(:,1)),2)]; %Se extraen los indices y se comprueba que
son posibles
        if(indices_siguientes(1)>=1)
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
```

```
                end
            end
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
        end
    end
    if(proximo_vecino==8)
        indices_siguientes = [ruta(length(ruta(:,1)),1)-1,
ruta(length(ruta(:,1)),2)+1]; %Se extraen los indices y se comprueba que
son posibles
        if((indices_siguientes(2)<=subs_x)&(indices_siguientes(1)>=1))
            if((matriz_campo(indices_siguientes(1),
indices_siguientes(2))>0)&(matriz_campo(indices_siguientes(1),
indices_siguientes(2))<=matriz_campo(ruta(length(ruta(:,1)),1),
ruta(length(ruta(:,1)),2))))
                ruta = [ruta; indices_siguientes]; %Se actualiza la ruta
si hay un decremento en el coste
                %Se explora la nueva celda mediante recursividad
                [ruta, destino_alcanzado] =
celdas_uniformes_explorar_campo_potencial(destino_alcanzado, ruta,
celdas_un, matriz_campo, matriz_un, indices_celda_origen, subs_x,
subs_y);
                if(destino_alcanzado==0)
                    ruta(length(ruta(:,1)),:) = []; %Si la funcion vuelve
sobre sus pasos, se borra la ruta seguida
                end
            end
            if(matriz_un(indices_siguientes(1), indices_siguientes(2))==-
1) %Si la funcion alcanza el origen, termina
                destino_alcanzado=1;
                ruta = [ruta; indices_celda_origen];
            end
        end
    end
    i=i+1;
end
end
end
```

Código en Matlab de *quadtree.m*:

```
function arbol = quadtree(entorno, arbol, ent_dimenx, ent_dimeny,
offsetx, offsety, sub_max, observar_quadtree)
%La funcion quadtree.m es una funcion recursiva (se llama a si misma
%hasta terminar el proceso que se plantea) que permite la subdivision en
%celdas para delimitar obstaculos del entorno que se le introduce como
%argumento de entrada (entorno). Recibe tambien como entrada el arreglo
%tipo cell (arbol) que contendra el diagrama de arbol que codifica
%internamente las ramificaciones. Asimismo, la funcion recibe como
%argumentos las dimensiones del entorno (ent_dimenx, ent_dimeny), los
%offset en coordendas que permiten identificar el punto de partida de la
%ramificacion en cada iteracion (offsetx, offsety) y la variable sub_max,
%que permite finalizar la ramificacion cuando esta llega al nivel que se
le
%introduce.
if(arbol{1,1}(2,7)<=sub_max) %condicion de continuidad en la recursion

    %Primer paso. Actualizar los valores de los vertices de las celdas:
    %-------------------------------------------------------------------
--
    %Se divide entre dos las dimensiones del entorno/celda que se estudia
para realizar la posterior ramificacion
    ent_dimenx=ent_dimenx/2;
    ent_dimeny=ent_dimeny/2;
    %Se ajustan los vertices de las 4 celdas resultantes de la
subdivision
    %Celda inferior izquierda
    arbol{1,1}=[offsetx,            offsety,            0, offsetx,
offsety+ent_dimeny, 0, 0;
                offsetx+ent_dimenx, offsety+ent_dimeny, 0,
offsetx+ent_dimenx, offsety,            0, arbol{1,1}(2,7)];

    %Celda inferior derecha
    arbol{1,2}=[offsetx+ent_dimenx,   offsety,            0,
offsetx+ent_dimenx,   offsety+ent_dimeny, 0, 0;
                offsetx+2*ent_dimenx, offsety+ent_dimeny, 0,
offsetx+2*ent_dimenx, offsety,            0, arbol{1,2}(2,7)];

    %Celda superior izquierda
    arbol{2,1}=[offsetx,            offsety+ent_dimeny,   0, offsetx,
offsety+2*ent_dimeny, 0, 0;
                offsetx+ent_dimenx, offsety+2*ent_dimeny, 0,
offsetx+ent_dimenx, offsety+ent_dimeny,   0, arbol{2,1}(2,7)];
    %Celda superior derecha
    arbol{2,2}=[offsetx+ent_dimenx,   offsety+ent_dimeny,   0,
offsetx+ent_dimenx,   offsety+2*ent_dimeny, 0, 0;
                offsetx+2*ent_dimenx, offsety+2*ent_dimeny, 0,
offsetx+2*ent_dimenx, offsety+ent_dimeny,   0, arbol{2,2}(2,7)];
    %-------------------------------------------------------------------
--
    %Segundo paso. Determinar si la celda esta libre, parcialmente
ocupada o integramente ocupada:
```

```
    %-------------------------------------------------------------------
--
    %Se comprueba para cada una de las cuatro celdas
    for(i=1:2)
        for(j=1:2)
            if(celda_ocupada_parcial(arbol{i,j}(:,1:6),entorno)) %Se
comprueba si esta ocupada parcialmente
                arbol{i,j}(1,7)=1; %1 para celdas parcialmente ocupadas
            end
            if(arbol{i,j}(1,7)==0) %Condicion de eficiencia: no es
necesario comprobar si esta ocupada integramente si ya esta ocupada
parcialmente
                if(celda_ocupada_integra(arbol{i,j}(:,1:6),entorno)) %Se
comprueba si esta ocupada integramente
                    arbol{i,j}(1,7)=2; %2 para celdas integramente
ocupadas
                end
            end
            %0 (sin cambios) para celdas libres
        end
    end
    %-------------------------------------------------------------------
--
    %Tercer paso. Representacion grafica:
    %-------------------------------------------------------------------
--
    %Pausa para observar la representacion grafica
    if(observar_quadtree==1)
        pause(0.05)
    end

    %Se representan en amarillo las celdas ocupadas y en blanco las
libres
    for(i=1:2)
        for(j=1:2)
            if ((arbol{i,j}(1,7)==1)|(arbol{i,j}(1,7)==2)) %condicion de
celda ocupada: el elemento (1,7) sera 1 o 2 cuando la celda este ocupada
                patch([arbol{i,j}(1,1), arbol{i,j}(1,4), arbol{i,j}(2,1),
arbol{i,j}(2,4)], [arbol{i,j}(1,2), arbol{i,j}(1,5), arbol{i,j}(2,2),
arbol{i,j}(2,5)], 'yellow');
            else %Celda libre
                patch([arbol{i,j}(1,1), arbol{i,j}(1,4), arbol{i,j}(2,1),
arbol{i,j}(2,4)], [arbol{i,j}(1,2), arbol{i,j}(1,5), arbol{i,j}(2,2),
arbol{i,j}(2,5)], 'white');
            end
        end
    end

    %Se reescribe el entorno sobre la representacion grafica ya que las
celdas al rellenarse se solapan encima del entorno
    dibuent(entorno,[0 0 1]);
    %-------------------------------------------------------------------
--
    %Cuarto paso. Ramificar:
```

```
    %-------------------------------------------------------------------
--
    %Se recorren las celdas para realizar las ramificaciones oportunas
    %Se debe ramificar sobre las celdas parcialmente ocupadas unicamente
    for(i=1:2)
        for(j=1:2)
            if(arbol{i,j}(1,7)==1) %condicion de celda parcialmente
ocupada: el elemento (1,7) sera 1 cuando la celda este parcialmente
ocupada

                %Se ajusta las coordenadas de offset a la esquina
inferior
                %izquierda de la celda parcialmente ocupada en cuestion
                offsetx=arbol{i,j}(1,1);
                offsety=arbol{i,j}(1,2);

                %Se produce una ramificacion en el arbol
                arbol{i,j+2}=cell(2,4);
                %Celda inferior izquierda (1,1)
                arbol{i,j+2}{1,1}=[0 0 0 0 0 0 0; 0 0 0 0 0 0
arbol{i,j}(2,7)+1]; %suma 1 al nivel de ramificacion actual de la rama
                %Celda inferior derecha (1,2)
                arbol{i,j+2}{1,2}=[0 0 0 0 0 0 0; 0 0 0 0 0 0
arbol{i,j}(2,7)+1]; %suma 1 al nivel de ramificacion actual de la rama
                %Celda superior izquierda (2,1)
                arbol{i,j+2}{2,1}=[0 0 0 0 0 0 0; 0 0 0 0 0 0
arbol{i,j}(2,7)+1]; %suma 1 al nivel de ramificacion actual de la rama
                %Celda superior derecha (2,2)
                arbol{i,j+2}{2,2}=[0 0 0 0 0 0 0; 0 0 0 0 0 0
arbol{i,j}(2,7)+1]; %suma 1 al nivel de ramificacion actual de la rama

                %Se llama a la funcion quadtree con el resultado de la
ramificacion (recursividad)
                arbol{i,j+2} = quadtree(entorno, arbol{i,j+2},
ent_dimenx, ent_dimeny, offsetx, offsety, sub_max, observar_quadtree);
            end
        end
    end
    %-------------------------------------------------------------------
--
end
end
```

Código en Matlab de *quadtree_preparaciones.m*:

```
function [ent_dimenx, ent_dimeny, cell_in, offsetx, offsety, sub_max] =
quadtree_preparaciones(entorno, sub_max, lado_x_celda, lado_y_celda)
%Funcion que prepara los parametros de entrada de la posterior funcion
%'quadtree.m' para division de un entorno. Requiere como argumentos de
%entrada: el entorno sobre el que desarrollar el quadtree (entorno) y el
nivel
%maximo de ramificacion del entorno (sub_max) o las sensibilidades x e y
de
%las celdas mas ramificadas. En caso de escoger un metodo u otro de
%sensibilidad, introducir la(s) variable(s) no escogida(s) igualada(s) a
0.
%Dimensiones totales del entorno y punto inicial de descomposicion:
%-------------------------------------------------------------------------
--
%Se extrae las dimensiones del entorno:
ent_dimenx=max(entorno(:,1))-min(entorno(:,1)); %Se extrae la dimension x
total que delimita el entorno en su conjunto
ent_dimeny=max(entorno(:,2))-min(entorno(:,2)); %Se extrae la dimension y
total que delimita el entorno en su conjunto
%Las variables offset indican el punto de partida desde su esquina
inferior izquierda en el que se
%comenzara a subdividir el entorno para formar el quadtree. Inicialmente
comienzan en (0,0),
%pero es necesario declararlos desde el comienzo ya que la funcion
quadtree.m es una funcion recursiva
offsetx=min(entorno(:,1));
offsety=min(entorno(:,2));
%-------------------------------------------------------------------------
--
%Resolucion del quadtree:
%-------------------------------------------------------------------------
--
%Se reajusta la sensibilidad maxima si se ha escogido el metodo de las
%dimensiones de la celda mas ramificada:
if((lado_x_celda~=0)&(lado_y_celda~=0))
    sub_max=ceil(max(log10((ent_dimenx-
min(entorno(:,1)))/lado_x_celda)/log10(2),log10((ent_dimeny-
min(entorno(:,2)))/lado_y_celda)/log10(2)));
end
%-------------------------------------------------------------------------
--
%Celda inicial del arbol:
%-------------------------------------------------------------------------
--
%Se plantea la variable cell inicial del diagrama en arbol, requerida por
la funcion recursiva
cell_in=cell(2,4);
%Al declarar la variable cell inicial comienzan como elementos vacios
%Celda inferior izquierda (1,1)
cell_in{1,1}=zeros(2,7);
%Celda inferior derecha (1,2)
```

```
cell_in{1,2}=zeros(2,7);
%Celda superior izquierda (2,1)
cell_in{2,1}=zeros(2,7);
%Celda superior derecha (2,2)
cell_in{2,2}=zeros(2,7);
%-------------------------------------------------------------------------
--
%Representacion grafica del quadtree:
%-------------------------------------------------------------------------
--
%Se crea la plantilla grafica donde se representara todo el proceso
figure
hold on;
title('Quadtree sobre el entorno');
xlabel('Dimension x (m)');
ylabel('Dimension y (m)');
dibuent(entorno,[0 0 1]); %Se observa el entorno antes de comenzar las
subdivisiones del quadtree
%-------------------------------------------------------------------------
--
```

Código en Matlab de *quadtree_preparaciones_para_ruta.m*:

```
function [pts_en_area_libre, nodos, matriz_costes] =
quadtree_preparaciones_para_ruta(arbol, sub_max, pt_in, pt_fin)
%Funcion que prepara algunos parametros para el calculo de rutas sobre el
%quadtree. Devuelve los parametros necesarios para el uso de las funcion
%Dijkstra.m
%
% --> Para la funcion Dijkstra.m, que posee una estructura tal que:
% [Camino, Coste]=Dijkstra(matriz_costes,nodo_inicial,nodo_final)
%
%La funcion presente devuelve los argumentos de entrada con la siguiente
equivalencia:
% · matriz_costes -> matriz_costes_entre_nodos
% · nodo_inicial -> 1
% · nodo_final -> 2
%
%Requiere como argumentos de entrada el quadtree (arbol), el numero de
%ramificaciones maxima (sub_max) y los puntos inicial (pt_in) y final
%(pt_fin).
%Calculos preliminares
%-------------------------------------------------------------------------
--
%El primer paso es localizar los puntos inicial y final para comprobar
que se encuentren en la region libre
%A partir de las funciones quadtree_localizar_punto.m y
quadtree_extraer_celda.m
%se puede localizar un punto como un itinerario de ramificaciones y la
celda concreta en la que se encuentra:
itinerario_in = quadtree_localizar_punto(arbol, pt_in, sub_max, [], 0);
itinerario_fin = quadtree_localizar_punto(arbol, pt_fin, sub_max, [], 0);
%Se extraen las celdas correspondientes a los itinerarios
celda_in = quadtree_extraer_celda(arbol, itinerario_in);
celda_fin = quadtree_extraer_celda(arbol, itinerario_fin);
%Si el elemento (1,7) de cada celda es 0, ambos puntos se encuentran en
el area libre
if((celda_in{1,1}(1,7)==0)&(celda_fin{1,1}(1,7)==0))
    pts_en_area_libre = 1;
else
    pts_en_area_libre = 0;
    disp("Alguno de los puntos inicial y/o final no se encuentran en el
area libre. Ruta no realizable")
end
%Se representan graficamente los puntos inicial y final
p1=plot(pt_in(1), pt_in(2),'xm','DisplayName', 'Punto inicial');
p2=plot(pt_fin(1), pt_fin(2),'om','DisplayName', 'Punto final');
legend([p1 p2],'AutoUpdate','Off');
%-------------------------------------------------------------------------
--
%Nodos del quadtree
%-------------------------------------------------------------------------
%Lo primero es contruir la matriz de nodos del quadtree
nodos = quadtree_extraer_nodos(arbol, sub_max, []);
```

```
%Se representan graficamente los nodos
for(i=1:length(nodos(:,1)))
    plot(nodos(i,1),nodos(i,2),'.','Color',[0.8500 0.3250 0.0980]);
%Color naranja
end
%Se añade los puntos inicial y final al comienzo de la matriz
nodos = [pt_in; pt_fin; nodos];
%A partir del tamaño de nodos se declara la matriz de costes:
matriz_costes = zeros(size(nodos,1));
matriz_costes(:,:) = inf; %Se inicializa en infito sus valores
%-------------------------------------------------------------------
%Matriz de costes
%-------------------------------------------------------------------
--
%Coste entre punto inicial y final
%-----------------------------------------------------------
%Si los itinerarios inicial y final son iguales, significa que ambos
puntos
%se encuentran en la misma celda, por lo el coste entre ellos sera la
distancia
itinerarios_iguales = 1; %Se presupone que son iguales
%Se recorren los vectores comparando cada elemento
if(length(itinerario_in)==length(itinerario_fin))
    for(i=1:length(itinerario_in))
        if(itinerario_in(i)~=itinerario_fin(i)) %Si un elemento no es
igual -> itinerarios diferentes
            itinerarios_iguales = 0;
        end
    end
else
    itinerarios_iguales = 0; %Si los vectores no poseen el mismo tamaño,
seguro que son diferentes
end
if(itinerarios_iguales==1) %Si los itineraios son iguales -->
coste=distancia
    matriz_costes(1,2) = sqrt((pt_in(1)-pt_fin(1))^2+(pt_in(2)-
pt_fin(2))^2);
    matriz_costes(2,1) = matriz_costes(1,2);
end
%-----------------------------------------------------------
%Coste entre punto inicial y final y los nodos
%-----------------------------------------------------------
%Se calculan los nodos de las celdas en las que se encuentran los puntos
inicial y final:
nodo_celda_in = [(celda_in{1,1}(1,1)+celda_in{1,1}(2,1))/2,
(celda_in{1,1}(1,2)+celda_in{1,1}(2,2))/2, 0];
nodo_celda_fin = [(celda_fin{1,1}(1,1)+celda_fin{1,1}(2,1))/2,
(celda_fin{1,1}(1,2)+celda_fin{1,1}(2,2))/2, 0];
%De la matriz de nodos, se busca el indice en el que se encuentran los
%nodos de las celdas inicial y final, y se actualiza la matriz de costes
for(i=1:length(nodos(:,1)))
    if((nodo_celda_in(1)==nodos(i,1))&(nodo_celda_in(2)==nodos(i,2)))
        matriz_costes(1,i) = sqrt((pt_in(1)-nodos(i,1))^2+(pt_in(2)-
nodos(i,2))^2);
```

```
        matriz_costes(i,1) = matriz_costes(1,i);
    end
end
for(i=1:length(nodos(:,1)))
    if((nodo_celda_fin(1)==nodos(i,1))&(nodo_celda_fin(2)==nodos(i,2)))
        matriz_costes(2,i) = sqrt((pt_fin(1)-nodos(i,1))^2+(pt_fin(2)-
nodos(i,2))^2);
        matriz_costes(i,2) = matriz_costes(2,i);
    end
end
%-------------------------------------------------------------
%Costes entre nodos
%-------------------------------------------------------------
%Se recorren los nodos buscando sus contiguos y actualizando su coste:
for(i=3:length(nodos(:,1))) %Comienza en 3 porque el 1 y el 2 son los
nodos inicial y final
    for(j=3:length(nodos(:,1)))

if(quadtree_celdas_contiguas(nodos(i,:),nodos(j,:),arbol,sub_max)) %Si
los dos nodos de estudio poseen celdas contiguas
            dist_nodos = sqrt((nodos(i,1)-nodos(j,1))^2+(nodos(i,2)-
nodos(j,2))^2); %coste=distancia
            if(dist_nodos>0) %Condicion para que no se actualice la
distancia a si mismo (ya que esta seria 0 y se quiere que siga siendo
inf)
                matriz_costes(i,j)=dist_nodos;
                matriz_costes(j,i)=dist_nodos;
            end
        end
    end
end
%-------------------------------------------------------------
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *quadtree_localizar_punto.m*:

```
function [itinerario, pt_libre] = quadtree_localizar_punto(arbol, punto,
sub_max, itinerario, pt_libre)
%Funcion que recibe como datos de entrada el arbol (quadtree), un punto,
%el numero de subdivisiones (o ramificaciones) maxima del arbol,
%y los propios argumentos de salida:
% --> itinerario: vector que indica la ruta (ramificacion) que localiza
la
%celda en la que se encuentra el punto, sea de area 'libre',
'parcialmente
%ocupada' o 'integramente ocupada'. Se debe introducir inicialmente en la
%funcion como un vector vacio (itinerario=[]).
% --> pt_libre: bandera que indica con un 1 que el punto introducido se
%encuentra en el area 'libre' del arbol. Devuelve 0 en caso contrario.
%Inicialmente, debe introducirse como argumento de entrada a la funcion
con
%valor nulo (pt_libre=0).
if((pt_libre==0)&(arbol{1,1}(2,7)<=sub_max)) %condicion de continuidad en
la recursion
    for(i=1:2)
        for(j=1:2) %Se recorre las 4 ramificaciones de cada iteracion

if((punto(1)>=arbol{i,j}(1,1))&(punto(1)<=arbol{i,j}(2,1))&(punto(2)>=arb
ol{i,j}(1,2))&(punto(2)<=arbol{i,j}(2,2))) %condicion de estar entre los
cuatro vectices de una celda

                %Se actualiza el itinerario
                if((i==1)&(j==1))
                    itinerario=[itinerario, 1];
                end
                if((i==1)&(j==2))
                    itinerario=[itinerario, 2];
                end
                if((i==2)&(j==1))
                    itinerario=[itinerario, 3];
                end
                if((i==2)&(j==2))
                    itinerario=[itinerario, 4];
                end

                if(arbol{i,j}(1,7)==0) %condicion de que la celda en
cuestion sea 'libre'
                    pt_libre=1;
                else %Si no es 'libre', pero es 'parcialmente ocupada',
se debe seguir ramificando hasta localizar el punto
                    if((arbol{i,j}(1,7)==1)) %Condicion de que la rama
pueda seguir ramificandose (solo cuando la ocupación de la celda es
parcial)
                        [itinerario, pt_libre] =
quadtree_localizar_punto(arbol{i,j+2}, punto, sub_max, itinerario,
pt_libre);
                    end
```

```
                    end
                end
            end
        end
    end
    end
```

Código en Matlab de *quadtree_extraer_nodos.m*:

```
function nodos = quadtree_extraer_nodos(arbol, sub_max, nodos)
%Funcion que devuelve una matriz de tamaño nx3, donde n es el numero de
%nodos y 3 son las columnas que localizan cada nodo a partir de sus
%coordenadas (xi yi zi). Es requerido como argumentos de entrada el
%quadtree (arbol), la subdivision maxima del quadtree (sub_max) y la
%propia salida de la funcion inicializada como una matriz vacia
(nodos=[])
if(arbol{1,1}(2,7)<=sub_max)
    %Se recorre las celdas actuales
    for(i=1:2)
        for(j=1:2)
            if(arbol{i,j}(1,7)==0) %condicion de celda libre
                %Se actualiza la lista de nodos
                nodos=[nodos;
                       (arbol{i,j}(1,1)+arbol{i,j}(2,1))/2,
(arbol{i,j}(1,2)+arbol{i,j}(2,2))/2, 0];
            else
                if(arbol{i,j}(1,7)==1) %condicion de celda parcialmente
ocupada
                    %Se explora la celda parcialmente ocupada
                    nodos=quadtree_extraer_nodos(arbol{i,j+2}, sub_max,
nodos);
                end
            end
        end
    end
end
end
```

Código en Matlab de *quadtree_extraer_celda.m*:

```
function ramificacion = quadtree_extraer_celda(arbol,itinerario)
%Funcion que devuelve la celda deseada y toda la rama posterior de la
%celda en cuestion. Se le debe introducir como argumentos de entrada el
%propio arbol y un vector que simule el itinerario a seguir.
%Ejemplo: [1 3 4 2]
%El 1 representa la celda inferior izquierda. El 2 la celda inferior
%derecha. El 3 la celda superior izquierda. Y el 4 la celda superior
%derecha.
ramificacion={{}, arbol};
%Variables auxiliares para el bucle
error=0; %bandera que se activa si la funcion detecta que la ramificacion
no llega tan lejos como exige el itinerario
i=1;
while((i<=length(itinerario))&(error==0))
    switch(itinerario(i))
        case 1
            ramificacion={ramificacion{1,2}{1,1},
ramificacion{1,2}{1,3}};
        case 2
            ramificacion={ramificacion{1,2}{1,2},
ramificacion{1,2}{1,4}};
        case 3
            ramificacion={ramificacion{1,2}{2,1},
ramificacion{1,2}{2,3}};
        case 4
            ramificacion={ramificacion{1,2}{2,2},
ramificacion{1,2}{2,4}};
        otherwise
            disp('Error en quadtree_extraer_celda.m. Algun valor
introducido en el itinerario no es correcto.');
    end
    if((size(ramificacion{1,2})==0)&(i<length(itinerario)))
        error=1;
        disp('Error en quadtree_extraer_celda.m. Se ha introducido un
itinerario no posible. La ramificacion no llega tan lejos como exige el
itinerario. Se ha devuelto la ramificacion de nivel inmediatamente
anterior');
    end

    i=i+1;
end
end
```

Código en Matlab de *quadtree_celdas_contiguas.m*:

```
function out = quadtree_celdas_contiguas(pt1, pt2, arbol, sub_max)
%Funcion que recibe dos puntos del entorno (pt1 y pt2), el quadtree
%desarrollado (arbol) y el nivel de ramificacion maxima del quadtree
%(sub_max). Devuelve un 1 si ambos puntos estan en celdas contiguas y un
0
%en caso contrario.
%Se presupone que no son contiguas hasta que se demuestre lo contrario
out=0;
%Se obtiene el itinerario que localiza cada punto
itinerario1 = quadtree_localizar_punto(arbol, pt1, sub_max, [], 0);
itinerario2 = quadtree_localizar_punto(arbol, pt2, sub_max, [], 0);
%Se extrae la celda que identifica cada itinerario
celda1 = quadtree_extraer_celda(arbol, itinerario1);
celda2 = quadtree_extraer_celda(arbol, itinerario2);
%Para que dos celdas sean contiguas, algun vertice de alguna de ellas
debe
%estar incluido en alguno de los lados de la otra celda
%Se comprueban los vertices de la celda1 sobre los lados de la celda2:
j=1;
while(j<=2)
    i=0;
    while(i<=3)
        if(punto_pertenece_segmento(celda1{1,1}(j,(1+i):(3+i)),
celda2{1,1}(1,1:6)))
            out=1;
        end
        if(punto_pertenece_segmento(celda1{1,1}(j,(1+i):(3+i)),
[celda2{1,1}(1,4:6) celda2{1,1}(2,1:3)]))
            out=1;
        end
        if(punto_pertenece_segmento(celda1{1,1}(j,(1+i):(3+i)),
celda2{1,1}(2,1:6)))
            out=1;
        end
        if(punto_pertenece_segmento(celda1{1,1}(j,(1+i):(3+i)),
[celda2{1,1}(2,4:6) celda2{1,1}(2,1:3)]))
            out=1;
        end
        if(out==1)
            i=4;
        else
            i=i+3;
        end
    end
    if(out==1)
        j=3;
    else
        j=j+1;
    end
end
```

```
if(out==0) %Se comprueban los vertices de la celda2 sobre los lados de la
celda1:
    j=1;
    while(j<=2)
        i=0;
        while(i<=3)
            if(punto_pertenece_segmento(celda2{1,1}(j,(1+i):(3+i)),
celda1{1,1}(1,1:6)))
                out=1;
            end
            if(punto_pertenece_segmento(celda2{1,1}(j,(1+i):(3+i)),
[celda1{1,1}(1,4:6) celda1{1,1}(2,1:3)]))
                out=1;
            end
            if(punto_pertenece_segmento(celda2{1,1}(j,(1+i):(3+i)),
celda1{1,1}(2,1:6)))
                out=1;
            end
            if(punto_pertenece_segmento(celda2{1,1}(j,(1+i):(3+i)),
[celda1{1,1}(2,4:6) celda1{1,1}(2,1:3)]))
                out=1;
            end
            if(out==1)
                i=4;
            else
                i=i+3;
            end
        end
        if(out==1)
            j=3;
        else
            j=j+1;
        end
    end
end
end
```

Código en Matlab de *punto_pertenece_segmento.m*:

```
function out = punto_pertenece_segmento(punto,segmento)
%Funcion que devuelve un 1 si el punto de entrada (punto) pertenece al
%segmento de entrada (segmento). Retorna un 0 en caso contrario.
%Se nombra el punto a comprobar como A
A=punto(1:2);
%Se reescriben los dos puntos que componen el segmento. El primero es B y
%el segundo C
B=segmento(1:2);
C=segmento(4:5);
%Si el punto a comprobar ya es uno de los puntos inciales o finales del
%segmento, no es necesario mas calculos
if(((round(A(1),4)==round(B(1),4))&(round(A(2),4)==round(B(2),4)))|((roun
d(A(1),4)==round(C(1),4))&((round(A(2),4)==round(C(2),4)))))
    out=1;
else
    %Se crea el vector que une los dos puntos del segmento (BC) y dos
vectores
    %desde el punto a comprobar (A) hasta el punto inicial y final del
    %segmento: (AB y AC)
    AB=[B(1)-A(1), B(2)-A(2)];
    AC=[C(1)-A(1), C(2)-A(2)];
    BC=[C(1)-B(1), C(2)-B(2)];
    %Se calculan los modulos de los vectores
    AB_mod=sqrt((AB(1))^2+(AB(2))^2);
    AC_mod=sqrt((AC(1))^2+(AC(2))^2);
    BC_mod=sqrt((BC(1))^2+(BC(2))^2);
    %Angulo que forman BC (segmento) y AB (punto a comprobar -> punto
inicial segmento):
    angulo1=acos((BC(1)*AB(1)+BC(2)*AB(2))/(BC_mod*AB_mod));
    %Angulo que forman BC (segmento) y AC (punto a comprobar -> punto
final segmento):
    angulo2=acos((BC(1)*AC(1)+BC(2)*AC(2))/(BC_mod*AC_mod));
    %Se calcula el producto escalar entre AB y AC (vectorer creados a
partir
    %del punto a comprobar y el punto inicial y final del segmento):
    producto=AB(1)*AC(1)+AB(2)*AC(2);
    %Condicion de pertenencia: el angulo 1 o 2 debe ser nulo y el que no
sea
    %nulo debe ser igual a 180 grados (pi radianes). Ademas, el producto
AB·AC
    %debe ser negativo:
    if(((angulo1<0.001)&(angulo1>-0.001) ... %((condicion de angulo1 muy
proximo a 0
        &(angulo2>3.14)&(angulo2<3.15)) ... %condicion de angulo2 muy
proximo a pi)
        | ...                                %or
        ((angulo2<0.001)&(angulo2>-0.001) ... %(condicion de angulo2 muy
proximo a 0
        &(angulo1>3.14)&(angulo1<3.15)) ... %condicion de angulo1 muy
proximo a pi))
        & ...                                %and
```

```
          producto<0)                          %condicion de producto
negativo
         out=1;
    else
         out=0;
    end

end
end
```

Código en Matlab de *diagrama_voronoi.m*:

```
function cell_voronoi = diagrama_voronoi(entorno, precision)
%Funcion que traza graficamente el diagrama de Voronoi de un entorno
%(entorno) que se le introduce como argumento de entrada. Requiere un
%valor de precision introducido por el usuario. Este valor va de 1 a 5,
%siendo 5 la mayor precision y 1 la menor. Tengase en cuenta que una
mayor
%precision requerira un coste computacional mayor.
%Precision recomendable: 3~4.
%
%Para el uso de esta funcion es necesaria las funciones relativas al
%quadtree, ya que es requerido preliminarmente un proceso de distincion
%entre area libre y area ocupada, y se ha escogido el quadtree como
%herramienta.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Para calcular el diagrama de Voronoi es necesario realizar
preliminarmente
%un metodo que distinga el area ocupada del area libre. Se escoge el
%quadtree con nivel de ramificacion 4
%Preparacion del quadtree
[ent_dimenx, ent_dimeny, cell_in, offsetx, offsety, sub_max] =
quadtree_preparaciones(entorno, 4, 0, 0);
%Quadtree
arbol = quadtree(entorno, cell_in, ent_dimenx, ent_dimeny, offsetx,
offsety, sub_max, 0);
%Se genera la representacion grafica preliminar donde se observara el
diagrama de Voronoi
figure
hold on;
title('Diagrama de Voronoi');
xlabel('Dimension x (m)');
ylabel('Dimension y (m)');
dibuent(entorno,[0 0 1]); %Se observa el entorno antes del diagrama de
Voronoi
%Para el uso de la funcion 'diagrama_voronoi.m' es necesario que todos
los
%segmentos que componen los obstaculos posean en su septima columna el
%angulo que forman con la vertical (al igual que los segmentos que
%delimitan el entorno):
entorno = reescribir_entorno(entorno,0,0);
%Se discretiza el entorno en funcion del grado de precision introducido
por el usuario:
precision=precision*100;
paso_discretizacion = max(ent_dimenx,ent_dimeny)/precision;
ent_dis={}; %Celda que contiene el entorno discretizado
cell_voronoi={}; %Celda que contendra los puntos pertenecientes al
diagrama de Voronoi
%-------------------------------------------------------------------------
--
%Discretizacion del entorno:
```

```
%-------------------------------------------------------------------------
--
%Cada elemento de la celda sera un vector que identifica el punto
discretizado
for(i=1:(round(ent_dimeny/paso_discretizacion)-1))
    for(j=1:(round(ent_dimenx/paso_discretizacion)-1))

        ent_dis{i,j}(1,1)=j*paso_discretizacion;
        ent_dis{i,j}(1,2)=i*paso_discretizacion;
        ent_dis{i,j}(1,3)=0;
    end
end
%-------------------------------------------------------------------------
--
%Diagrama de Voronoi:
%-------------------------------------------------------------------------
--
%Funcionamiento:
%1. Se recorren todos los puntos del entorno discretizado
%2. Se comprueba si el punto actual pertenece al area 'libre' hallada por
el quadtree
%3. Si pertenece al area 'libre', se comprueba si la distancia entre ese
%punto y los dos obstaculos o lados del entorno mas cercanos se aproxima
a
%ser la misma ('punto_pertenece_voronoi.m').
%4. Si ello se cumple, se representa el punto
for(i=1:(round(ent_dimeny/paso_discretizacion)-1))
    for(j=1:(round(ent_dimenx/paso_discretizacion)-1))

        [itinerario, pt_libre] = quadtree_localizar_punto(arbol,
[ent_dis{i,j}(1,1), ent_dis{i,j}(1,2)], sub_max, [], 0);

        if(pt_libre)
            if(punto_pertenece_voronoi(ent_dis{i,j}, entorno,
paso_discretizacion))
                cell_voronoi{i,j} = ent_dis{i,j};
                plot(ent_dis{i,j}(1,1),ent_dis{i,j}(1,2),'k.');
            end
        end

    end
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *punto_pertenece_voronoi.m*:

```
function output = punto_pertenece_voronoi(punto,ent,paso)
%Funcion que devuelve un 1 si el punto del espacio pertenece al diagrama
de
%Voronoi y un 0 si no lo hace
distancias=[];
%Se calcula la distancia del punto introducido con todos los segmentos y
se
%guarda en la matriz columna: 'distancias'
for(i=1:length(ent(:,1)))
    %Para el segmento en cuestion que se estudia, se calcula su punto
    %mas cercano al punto introducido

    %Se calcula los angulos que formarian el segmento dado con dos
vectores
    %creados a partir del punto introducido y los punto inicial y final
del
    %segmento
    vector1=[punto(1)-ent(i,1), punto(2)-ent(i,2)];
    vector2=[punto(1)-ent(i,4), punto(2)-ent(i,5)];
    mod_vect1=sqrt((punto(1)-ent(i,1))^2+(punto(2)-ent(i,2))^2);
    mod_vect2=sqrt((punto(1)-ent(i,4))^2+(punto(2)-ent(i,5))^2);
    mod_seg=sqrt((ent(i,4)-ent(i,1))^2+(ent(i,5)-ent(i,2))^2);
    prod_esc1=(vector1(1)*(ent(i,4)-ent(i,1)))+(vector1(2)*(ent(i,5)-
ent(i,2)));
    prod_esc2=(vector2(1)*(ent(i,1)-ent(i,4)))+(vector2(2)*(ent(i,2)-
ent(i,5)));
    angulo1=acos(prod_esc1/(mod_vect1*mod_seg))*180/pi;
    angulo2=acos(prod_esc2/(mod_vect2*mod_seg))*180/pi;
    %Si alguno de los angulos es igual o mayor que 90 grados, significa
que el
    %vertice del segmento con el que forma ese angulo es el punto mas
cercano
    if(angulo1>=90)
        distancias=[distancias;
                    mod_vect1];
    else
        if(angulo2>=90)
            distancias=[distancias;
                        mod_vect2];
        else
            %Si no se cumple, significa que el punto mas cercano es la
            %distancia normal del segmento al punto

punto_interseccion_normal=[interseccion_normal(punto,ent(i,1:7)), 0];
            distancias=[distancias;
                        sqrt((punto_interseccion_normal(1)-
punto(1))^2+(punto_interseccion_normal(2)-punto(2))^2)];
        end
    end
end
%Una vez se tiene configurada la matriz columna 'distancias', se busca la
```

```
%minima distancia:
[distancia_min indice]=min(distancias);
%Se sustituye en la matriz el elemento extraido por infinito
distancias(indice)=inf;
%Se busca la siguiente distancia minima con la condicion de que no sea
%perteneciente al mismo obstaculo:
bandera_fin_while=0; %bandera que permite ajustar la condicion del fin
del while
while(bandera_fin_while==0)
    [distancia_min_2 indice_2]=min(distancias);
    if((indice>4)&(ent(indice_2,8)==ent(indice,8)))
        distancias(indice_2)=inf;
    else
        bandera_fin_while=1;
    end
end
%Si la diferencia entre las distancias minimas es menor que el paso de
%discretizacion/10, se considera que el punto esta a la misma distancia
de
%ambos segmentos que se han estudiado y, por tanto, el punto pertenece al
%diagrama de Voronoi
if(abs(distancia_min-distancia_min_2)<paso)
    output=1;
else
    output=0;
end
end
```

Código en Matlab de *reescribit_entorno.m*:

```
function ent_reescrito = reescribir_entorno(ent,
cambiar_sentido_obstaculos, cambiar_sentido_limites)
%Funcion que reescribe el entorno que se le introduce como entrada (ent).
%Por un lado, reescribe el orden de introducción de los segmentos que
%definen los limites y los obstaculos, cambiando el orden actual de
%introduccion de horario a antihorario o viceversa. Para cambiar el orden
%los argumentos cambiar_sentido_obstaculos y cambiar_sentido_limites
%deberan valer 1. En caso contrario, no se producira cambio.
%Por otro lado, calcula los angulos de los segmentos para la septima
%columna.
ent_reescrito=ent(1:4,:);
%Reescritura de los segmentos:
%------------------------------------------------------------------------
--
if(cambiar_sentido_limites==1)
    for(i=1:3)
        ent_reescrito(i,4:6)=ent(5-i,1:3);
        ent_reescrito(i+1,1:3)=ent_reescrito(i,4:6);
    end
end
if(cambiar_sentido_obstaculos==1)
    for(i=1:max(ent(:,8)))
        obstaculo=extrae_obstaculo(i,ent);
        obstaculo_reescrito=obstaculo;
        for(j=1:length(obstaculo(:,1))-1)

obstaculo_reescrito(j,4:6)=obstaculo(length(obstaculo(:,1))+1-j,1:3);
            obstaculo_reescrito(j+1,1:3)=obstaculo_reescrito(j,4:6);
        end
        ent_reescrito=[ent_reescrito; obstaculo_reescrito];
    end
else
    ent_reescrito=[ent_reescrito; ent(5:length(ent(:,1)),:)];
end
%------------------------------------------------------------------------
--
%Cálculo de los ángulos para la séptima columna de cada segmento:
%------------------------------------------------------------------------
--
for(i=1:length(ent_reescrito(:,1)))

if(ent_reescrito(i,5)>ent_reescrito(i,2))&(ent_reescrito(i,4)>=ent_reescr
ito(i,1))
        ent_reescrito(i,7)=atan((ent_reescrito(i,5)-
ent_reescrito(i,2))/(ent_reescrito(i,4)-ent_reescrito(i,1)))+pi/2;
    else

if(ent_reescrito(i,5)>=ent_reescrito(i,2))&(ent_reescrito(i,4)<ent_reescr
ito(i,1))
            ent_reescrito(i,7)=atan((ent_reescrito(i,5)-
ent_reescrito(i,2))/(ent_reescrito(i,4)-ent_reescrito(i,1)))-pi/2;
```

```
        else

if((ent_reescrito(i,5)<ent_reescrito(i,2))&(ent_reescrito(i,4)<ent_reescr
ito(i,1)))
                ent_reescrito(i,7)=atan((ent_reescrito(i,5)-
ent_reescrito(i,2))/(ent_reescrito(i,4)-ent_reescrito(i,1)))-pi/2;
            else
                ent_reescrito(i,7)=atan((ent_reescrito(i,5)-
ent_reescrito(i,2))/(ent_reescrito(i,4)-ent_reescrito(i,1)))+pi/2;
            end
        end
    end
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *extrae_informacion_vertice.m*:

```
function [vertice_convexo, angulo_vertice, angulo1, angulo2] =
extrae_informacion_vertice(entorno, vertice)
%Función que retorna (en la salida vertice_convexo) un 1 si el vertice
%(vector de tres dimensiones) introducido del entorno es convexo. Retorna
0
%en caso contrario, es decir, concavo. Si el vertice forma 180 grados se
%considerara concavo. Además, también devuelve el angulo que forma el
%vertice (angulo_vertice), y los angulo de los segmentos que lo originan
%con la vertical negativa (angulo1, angulo2).
%
%Requerimiento: todo el entorno debe estar definido en sentido horario.
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se calculan los angulos por si estos no han sido calculados
entorno=reescribir_entorno(entorno,0,0);
%Se extrae el obstaculo que contiene el vertice y el indice en el que se
%encuentra en ese obstaculo
for(i=1:length(entorno(:,1)))

if((vertice(1)==entorno(i,1))&(vertice(2)==entorno(i,2))&(vertice(3)==ent
orno(i,3)))
        obstaculo=extrae_obstaculo(entorno(i,8),entorno);
    end
end
for(i=1:length(obstaculo(:,1)))

if((vertice(1)==obstaculo(i,1))&(vertice(2)==obstaculo(i,2))&(vertice(3)=
=obstaculo(i,3)))
        indice=i;
    end
end
%Extrae los dos segmentos que se cortan en el vertice
if(indice==1)
    segmento1=obstaculo(length(obstaculo(:,1)),:);
    segmento2=obstaculo(1,:);
else
    segmento1=obstaculo(indice-1,:);
    segmento2=obstaculo(indice,:);
end
%-------------------------------------------------------------------------
--
%Calculo de los angulos:
%-------------------------------------------------------------------------
--
%Angulo que forma con el primer segmento
if(segmento1(7)==0)
    angulo1=pi;
end
if((segmento1(7)>0)&(segmento1(7)<pi/2))
    angulo1=-pi+segmento1(7);
```

```
end
if(segmento1(7)==pi/2)
    angulo1=-pi/2;
end
if((segmento1(7)>pi/2)&(segmento1(7)<pi))
    angulo1=-pi+segmento1(7);
end
if((segmento1(7)<0)&(segmento1(7)>-pi/2))
    angulo1=pi+segmento1(7);
end
if(segmento1(7)==-pi/2)
    angulo1=pi/2;
end
if((segmento1(7)<-pi/2)&(segmento1(7)>-pi))
    angulo1=pi+segmento1(7);
end
if(segmento1(7)==pi)
    angulo1=0;
end
if(angulo1<0)
    angulo1=2*pi+angulo1;
end
%Angulo que forma con el segundo segmento
angulo2=segmento2(7);
if(angulo2<0)
    angulo2=2*pi+angulo2;
end
%Angulo total del vertice
if(angulo1>angulo2)
    angulo_vertice=(2*pi-angulo1)+angulo2;
else
    angulo_vertice=angulo2-angulo1;
end
%Se pasa los angulos a grados
angulo1=angulo1*180.0/pi;
angulo2=angulo2*180.0/pi;
angulo_vertice=angulo_vertice*180.0/pi;
%Si el angulo total del vertice es superior a 180 grados, el vertice sera
%convexo
if(angulo_vertice<180)
    vertice_convexo=1;
else
    vertice_convexo=0;
end
%------------------------------------------------------------------------
--
end
```

Código en Matlab de *celda_ocupada_parcial.m*:

```
function output = celda_ocupada_parcial(celda, entorno)
%Funcion que recibe como parametros de entrada una matriz con las
%coordenadas de una celda (celda) y el entorno original (entorno).
%Devuelve 1 si la celda se encuentra ocupada parcialmente por algun
%obstaculo del entorno
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%La funcion presupone que output sera 0 a menos que se compruebe lo
contrario
output=0;
%Se extrae los cuatro segmentos (lados) que designan la celda de entrada
%(en sentido horario, desde el lado derecho):
lados_celda = [celda(1,1:6); celda(1,4:6), celda(2,1:3); celda(2,1:6);
celda(2,4:6), celda(1,1:3)];
%-------------------------------------------------------------------------
--
%Analisis de celda ocupada parcialmente:
%-------------------------------------------------------------------------
--
%Una celda estara ocupada parcialmente si algun segmento del algun
%obstaculo intersecciona con alguno de los lados de la celda. Existe una
%posibilidad adicional: que todo el obstaculo este contenido en la celda.
%Para abordar esta posibilidad la condicion a valorar es que dos puntos
del
%mismo segmento se encuentren en la celda
i=5;
while((i<=length(entorno(:,1)))&(output==0)) %Se recorren los segmentos
del obstaculo mientras no exista ya una interseccion
    %Se extraen los puntos inicial y final del segmento actual
    x0=entorno(i,1);
    y0=entorno(i,2);
    x1=entorno(i,4);
    y1=entorno(i,5);

    %Se calcula si se produce alguna interseccion en el segmento actual y
cualquiera de los lados de la celda
    for(j=1:length(lados_celda(:,1)))
        pc=inter_seg([lados_celda(j,1:2)],[lados_celda(j,4:5)],[x0
y0],[x1 y1]);
        if(pc(3)==1)
            output=1;
        end
    end

    %Se comprueba que ambos puntos del segmento no se encuentren en la
celda

if(((x0>lados_celda(1,1))&(x0<lados_celda(3,1))&(y0>lados_celda(1,2))&(y0
<lados_celda(3,2)))&((x1>lados_celda(1,1))&(x1<lados_celda(3,1))&(y1>lado
s_celda(1,2))&(y1<lados_celda(3,2))))
```

```
        output=1;
    end

    i=i+1;
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *celda_ocupada_integra.m*:

```
function output = celda_integra(celda, entorno)
%Funcion que devuelve un 1 si la celda que se le introduce como argumento
%se encuentra dentro integramente de algun obstaculo del entorno (ent).
%Devuelve un 0 si no. Para ello el criterio que sigue la funcion es
%comprobar si todos los vertices que componen la celda se encuentran
dentro
%del mismo obstaculo. Asimismo, un vertice se encuentra dentro de un
%obstaculo si, al trazar una linea aleatoria desde el vertice hasta un
%punto en el limite del entorno, esta linea intersecta un numero impar de
%veces con las lineas que constituyen el obstaculo en cuestion
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se presopone la salida output como 0 hasta que se compruebe lo contrario
output=0;
%Se extrae los cuatro segmentos (lados) que designan una celda
%(en sentido horario, desde el lado derecho):
lados_celda = [celda(1,1:6); celda(1,4:6), celda(2,1:3); celda(2,1:6);
celda(2,4:6), celda(1,1:3)];
%Se extrae los vertices de la celda (en sentido horario, desde la esquina
%inferior izquierda):
vertices_celda = [lados_celda(1,1:3); lados_celda(2,1:3);
lados_celda(3,1:3); lados_celda(4,1:3)];
%Se declara un contador que sera utilizado posteriormente
vertices_dentro = 0;
%-------------------------------------------------------------------------
--
%Analisis de celda ocupada integramente:
%-------------------------------------------------------------------------
--
%Una celda estara integramente ocupada por un obstaculo si los cuatro
%vertices de la celda se encuentran dentro del obstaculo y ningun lado de
%la celda produce interseccion alguna con el obstaculo
%Puntos dentro de obstaculo:
%Se recorren los obstaculos
i=1;
while((i<=max(entorno(:,8)))&(vertices_dentro~=4))
    obstaculo_actual = extrae_obstaculo(i, entorno); %Se extrae el
obstaculo actual
    vertices_dentro = 0; %Variable que contara los vertices dentro del
obstaculo actual, si llega a 4 todos los vertices estan dentro
    j=1;
    while((j<=4)&(vertices_dentro~=-1)) %Se recorren los vertices

if(punto_dentro_obstaculo(vertices_celda(j,:),obstaculo_actual,entorno))
            vertices_dentro=vertices_dentro+1;
        else
            vertices_dentro = -1; %Cuando se produce un punto fuera del
obstaculo, no es necesario comprobar mas
        end
        j=j+1;
```

```
    end
    i=i+1;
end
%Si se cumple que todos los puntos estan dentro del mismo obstaculo, se
%comprueba que no se produzca interseccion con ningun lado de la celda:
if(vertices_dentro==4)
    output=1;
    i=1;
    while((i<=length(obstaculo_actual(:,1)))&(output==1))
        for(j=1:length(lados_celda(:,1)))

pc=inter_seg((obstaculo_actual(i,1:2)),(obstaculo_actual(i,4:5)),(lados_c
elda(j,1:2)),(lados_celda(j,4:5)));
            if(pc(3)==1)
                output=0;
            end
        end
        i=i+1;
    end
end
%-------------------------------------------------------------------------
--
end
```

Código en Matlab de *punto_dentro_obstaculo.m*:

```
function output = punto_dentro_obstaculo(punto, obstaculo, entorno)
%Funcion que, a partir de un punto (punto, tres coordenadas x, y ,z), una
%matriz que contiene un obstaculo (obstaculo) y el entorno original
(entorno), devuelve un 1 el
%punto esta contenido dentro del obstaculo. Retorna un 0 en caso
contrario
%Calculos preliminares:
%-------------------------------------------------------------------------
--
%Se extraen las dimensiones del entorno
dimen_x_ent = max(entorno(:,1));
dimen_y_ent = max(entorno(:,2));
%Se extrae los vertices del obstaculo
vertices_obstaculo = obstaculo(:,1:3);
%Calculo de segmento aleatorio:
%Se crea un segmento cuyo origen se encuentra en el punto de entrada de
la
%funcion y finaliza aleatoriamente en un limite del entorno
num_aleat = rand();
if((num_aleat>=0)&(num_aleat<0.25))
    segmento_aleatorio = [punto, 0, max(entorno(:,2))*num_aleat/0.25, 0];
end
if((num_aleat>=0.25)&(num_aleat<0.50))
    segmento_aleatorio = [punto, max(entorno(:,1))*(num_aleat-0.25)/0.25,
max(entorno(:,2)), 0];
end
if((num_aleat>=0.50)&(num_aleat<0.75))
    segmento_aleatorio = [punto, max(entorno(:,1)),
max(entorno(:,2))*(num_aleat-0.50)/0.25, 0];
end
if((num_aleat>=0.75)&(num_aleat<=1))
    segmento_aleatorio = [punto, max(entorno(:,1))*(num_aleat-0.75)/0.25,
0, 0];
end
%Se calcula dos segmentos
%Uno horizontal desde el punto en cuestion al limite del entorno derecho
segmento_horizontal = [punto(1:3), dimen_x_ent, punto(2:3)];
%Otro vertical desde el punto en cuestion al limite del entorno superior
segmento_vertical = [punto(1:3), punto(1), dimen_y_ent, punto(3)];
%-------------------------------------------------------------------------
--
%Analisis punto dentro obstaculo
%-------------------------------------------------------------------------
--
%El punto estara contenido dentro del obstaculo si el numero de
%intersecciones del segmento aleatorio con los lados del obstaculo es
impar
cont_inter = 0; %Contador de intersecciones
for(i=1:length(obstaculo(:,1))) %Se recorren los segmentos del obstaculo
```

```
    pc =
inter_seg(segmento_aleatorio(1:2),segmento_aleatorio(4:5),obstaculo(i,1:2
),obstaculo(i,4:5));
    if(pc(3)==1)
        cont_inter = cont_inter + 1;
    end
end
if(mod(cont_inter,2)==1)
    output = 1;
else
    output = 0;
end
%-------------------------------------------------------------------------
--
end
```