UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Implementación de un entorno para la planificación de movimientos de robots en Matlab

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

AUTOR/A: Leach Rodríguez, Ricardo

Tutor/a: Zotovic Stanisic, Ranko

CURSO ACADÉMICO: 2022/2023

Índice

Introducción

El estudio y diseño de trayectorias juegan un papel fundamental en numerosos campos, desde la robótica y la navegación autónoma hasta la aviación y la logística. La generación de trayectorias eficientes y seguras se ha convertido en un desafío clave en la búsqueda de soluciones innovadoras y avanzadas para optimizar el desplazamiento de objetos y vehículos en entornos complejos.

La generación de trayectorias ha experimentado un notable avance en las últimas décadas, impulsada por la demanda de soluciones más eficientes y seguras en diferentes áreas de aplicación. Los avances en este campo han sido impulsados por el desarrollo de algoritmos y técnicas que abordan los desafíos de movimientos en entornos complejos y dinámicos.

Por ello, resulta de gran importancia contar con diversas soluciones eficientes que permitan la planificación de trayectos de forma precisa, valorando tanto el tiempo de generación de las trayectorias, así como la optimización del coste de la ruta final sin contratiempos y, sobre todo, evitar la colisión con los obstáculos del entorno.

En este trabajo de final de grado se implementan dos entornos orientados a la planificación de movimientos de robots en Matlab. En primer lugar, se plantea un robot de planta circular, que se aproxima a un robot cilíndrico. En segundo lugar, se considera un robot de planta poligonal, que podrá tener hasta tres grados de libertad.

Una vez generado el entorno para el robot apropiado se muestra su aplicación mediante la técnica del diagrama de visibilidad, el método de caminos probabilísticos o los árboles de búsqueda rápidamente explorables. Después de obtener un grafo incluyendo un punto inicial y un punto final, se utiliza el algoritmo de Dijkstra para encontrar la ruta más rápida siguiendo los caminos generados.

La finalidad de este trabajo es explorar el desarrollo de estos entornos mediante la aplicación de las técnicas mencionadas, mostrando la eficacia de cada método y el funcionamiento de cada entorno. Con estos avances y resultados se espera contribuir al avance en el campo de la generación de trayectorias y proporcionar información relevante para futuras investigaciones y aplicaciones prácticas.

Este trabajo cumple en un grado alto con el Objetivo de Desarrollo Sostenible *9. Industria, innovación e infraestructuras*; y en grado bajo con los objetivos *11. Ciudades y comunidades sostenibles*; y *8. Trabajo decente y crecimiento económico*.

Finalmente, el entorno parte de los estudios previos realizados por el alumno Iván Esteve González en “Planificación de movimiento de robots mediante la descomposición del entorno en celdas” presentado en 2022.

Ilustración 1. Objetivo de Desarrollo Sostenible 9. Industria, innovación e infraestructura.

Alcance

En este proyecto se estudia la introducción de objetos redondos a la simulación de obstáculos que hay en el entorno, así como una reestructuración completa de lo hecho hasta ahora implementando la programación orientada a objetos. Abarca un entorno para robots poligonales regulares o irregulares teniendo en cuenta la orientación del robot a la hora de expandir el entorno, así como un entorno para robots redondos. Finalmente se ponen en práctica ambos entornos mediante la generación de trayectorias a través de la técnica del diagrama de visibilidad, el método de caminos probabilísticos o los árboles de búsqueda rápidamente explorables.

Queda fuera del alcance de este estudio la introducción de la orientación como variable dinámica a la hora de generar trayectorias en entornos para robots poligonales, así como robots cuya geometría es simultáneamente poligonal y curva o robots elípticos, parabólicos o con curvas sinusoidales, o lo que es lo mismo, cualquier robot que no sea redondo o poligonal.

También queda fuera del alcance la generación de trayectorias cartesianas tridimensionales o la aplicación a brazos robóticos. Queda fuera del alcance la unión de obstáculos, así como el rellenado de éstos en caso de quedar huecos inaccesibles. En el caso de los robots poligonales, se ha tenido en cuenta exclusivamente la posibilidad de tener obstáculos poligonales, dejando la implementación de obstáculos curvos para un proyecto futuro.

En este trabajo no se han tenido en cuenta restricciones cinemáticas, y tan solo se ha aplicado la generación de trayectorias para el método básico de caminos probabilísticos o el método básico de los árboles de búsqueda rápidamente explorables, debido a que ya existen trabajos que profundizan en la optimización de éstos y su comparación.

Estado del arte

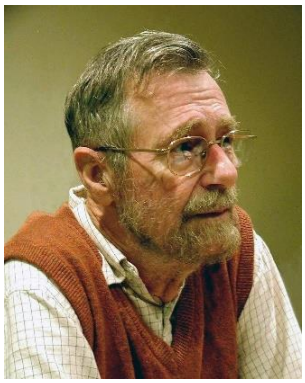


Ilustración 2. Edsger Dijkstra.

En 1956 el científico y matemático Edsger W. Dijkstra desarrolló el llamado algoritmo Dijkstra mientras trabajaba en el Mathematical Centre de Ámsterdam. Dijkstra investigaba métodos para optimizar el flujo de información en redes de telecomunicaciones. El algoritmo se publicó por primera vez en 1959 en un artículo titulado “A Note on Two Problems in Connexion with Graphs”. Éste fue revolucionario y se convirtió en una herramienta fundamental en la teoría de grafos. Se aplicó en diversas disciplinas como son la optimización de redes, resolución de problemas de enrutamiento de sistemas de comunicación, o en nuestro caso, la generación de trayectorias. Este algoritmo sentó las bases para futuros avances en la planificación de rutas y se considera un hito en la historia de la informática y la ciencia de la computación.

En 1996, Lydia Kavraki, Jean-Claude Latombe, P. Svestka y M. H. Overmars desarrollaron el método de los mapas de caminos probabilísticos o PRM por sus siglas en inglés (Probabilistic Road Map). Este enfoque innovador se desarrolló con el objetivo de abordar la generación de trayectorias en entornos complejos con muchos grados de libertad y con múltiples restricciones. El método se basa en la construcción de un grafo de muestreo que representa el espacio de configuración del sistema. Para ello, se generan nodos aleatorios en el espacio de configuración y se evalúa su factibilidad y colisión con obstáculos. Los nodos válidos se conectan mediante rectas para formar una estructura de grafo, junto al punto inicial y final. Posteriormente se utiliza el grafo para buscar trayectorias efectivas por el interior del entorno.

En el año 1998, el informático y profesor Steven M. LaValle propone el algoritmo de los árboles de búsqueda rápidamente explorables o RRT por sus siglas en inglés (Rapidly-Exploring Random Trees). El algoritmo surge como una solución para la generación de trayectorias en entornos complejos con gran número de dimensiones. Se basa en la construcción de un árbol que explora el espacio de configuración del sistema. En cada iteración se genera un nodo que se añade al nodo ya existente más cercano, expandiendo así el árbol. Es un algoritmo utilizado en robótica de tiempo real, animación y otras áreas donde se requiere generar trayectorias de forma eficiente y flexible.

Ilustración 3. Steven M. LaValle.

El algoritmo RRT ha sido desde su introducción objeto de numerosas variantes y mejoras a lo largo de los años como son el RRT Bidireccional, RRT Connect, Lazy RRT o el RRT Star.

Por otro lado, la robótica móvil es una rama de la robótica que se enfoca en el diseño y desarrollo de robots capaces de moverse de manera autónoma en entornos variables y complejos. Esta disciplina combina diferentes áreas de conocimiento, como la percepción sensorial, la planificación de movimientos o la inteligencia artificial, para permitir que los robots se desplacen, interactúen con el entorno o realicen tareas.

El sector de la robótica móvil ha experimentado avances significativos en las últimas décadas por el desarrollo de tecnologías como la localización y mapeo simultáneos o SLAM por sus signas en inglés (Simultaneous Localization And Mapping), los algoritmos de navegación, los sensores avanzados y los sistemas de control. Estos avances han permitido que la robótica móvil se aplique a campos como son la exploración espacial, la logística, agricultura de precisión o seguridad y vigilancia, entre otros.

Uno de los ejemplos más conocidos de robótica móvil cilíndrica es la Roomba. Desarrollada por la empresa iRobot, la Roomba es un robot autónomo diseñado para la limpieza de suelos en entornos domésticos. Su forma cilíndrica compacta le permite moverse fácilmente por diferentes áreas y maniobrar en espacios reducidos.

Ilustración 4. Robot aspirador Roomba de iRobot.

Algo menos evidente es su uso en aplicaciones industriales, como es el ejemplo de Amazon. A mediados del año 2022 la empresa norteamericana poseía 520.000 unidades de accionamiento robótico, como son los robots “Proteus”. Estos dispositivos son capaces de movilizar inventario de forma autónoma, así como guiarse en el interior de espacios esquivando objetos, guiándose entre un punto inicial y punto final, cambiando por completo la concepción de la dificultad y riesgo de trabajar de forma conjunta robots y humanos.

Ilustración 5. Robot Proteus de Amazon.

Objetivo

Este proyecto tiene como objetivo la implementación de un entorno para la planificación de movimientos de robots en Matlab. Para ello se toman dos tipos de robot: robots esféricos o cilíndricos, y robots poligonales.

Inicialmente se plantea un escenario o habitación y posiblemente uno o más obstáculos. El conjunto de escenario y obstáculos forma un entorno. Al introducirse en el programa junto con el robot, se debe calcular un área expandida donde el robot no colisione con los obstáculos. Éste área debe incluirse en el entorno. Gracias a esta expansión se puede considerar el robot como un punto sin dimensiones, por lo que, al generar la trayectoria permite no tener que valorar continuamente las colisiones del espacio del robot con los obstáculos.

Posteriormente se debe poder generar una trayectoria válida entre un punto inicial y un punto final en el entorno utilizando las técnicas del diagrama de visibilidad o el método de los mapas de caminos probabilísticos junto con el algoritmo de Dijkstra o bien el algoritmo de los árboles de búsqueda rápidamente explorables.

Todo este proceso se realiza utilizando programación orientada a objetos, de forma que la aplicación sea posible para entornos con características diversas, distinto número de obstáculos y favorecer que el futuro desarrollo del proyecto sea lo más sencillo posible a la hora implementar más dimensiones como es el tercer eje cartesiano o la orientación de robots poligonales.

Entorno Básico

Hasta ahora la expansión de un entorno se ha realizado a grandes rasgos mediante la creación de una matriz que albergaba las propiedades de cada una de las rectas que formaban el entorno, así como un ángulo normal a la recta que indica la dirección hacia la cual se encuentra el objeto delimitado.

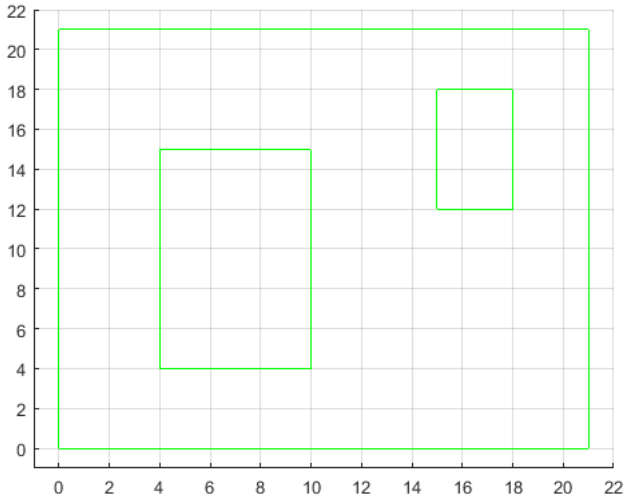


Ilustración 6. Entorno generado mediante matriz básica.

Para poder generar trayectorias dentro de un entorno, se debe tener en cuenta un punto de referencia del robot, y delinear los objetos en el punto de colisión del obstáculo con el robot, generando un área donde el punto de referencia no podrá entrar. Éste área será la que delimite el objeto a efectos prácticos para el robot.

De esta forma para expandir el objeto, se desplazaba cada una de las rectas hacia el exterior el radio del objeto y se alargaban la cantidad necesaria hasta que intersecten con otra recta del mismo objeto. De esta forma se consigue una expansión a grandes rasgos y poco detallada.

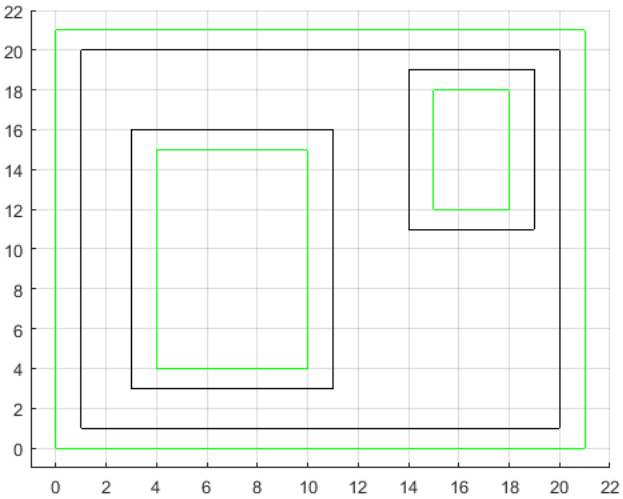


Ilustración 7. Entorno básico expandido.

En este caso, para un robot cilíndrico existen puntos dentro de la zona delimitada que sí serían accesibles, pero por las limitaciones del método no se tienen en cuenta. Por ejemplo, dos objetos

entre los cuales un robot cilíndrico sí podría pasar, pero que, con este método, no solo no se tiene en cuenta, sino que además se encuentra en el solape entre los dos objetos.

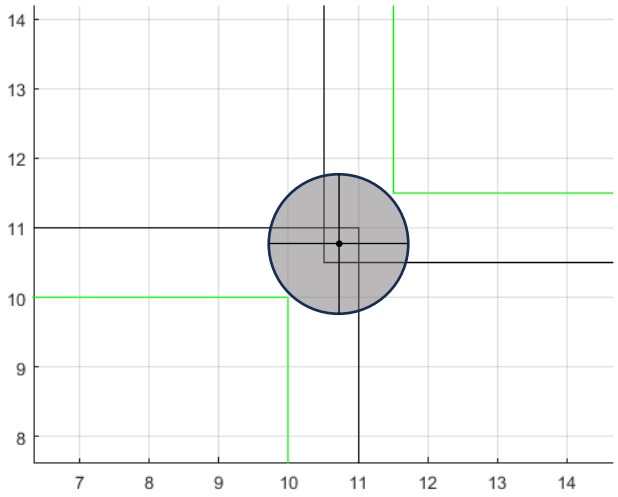


Ilustración 8. Demostración del error provocado por el método empleado.

Una vez generado el entorno expandido, tomando la matriz de éste, se puede generar las rectas del diagrama de visibilidad, o lo que es lo mismo, todas las posibles trayectorias que se puede crear entre vértices. Las rectas generadas son objetos de tipo “Line” de Matlab que se guardan en una lista, y se guarda una matriz con las distancias entre cada vértice.

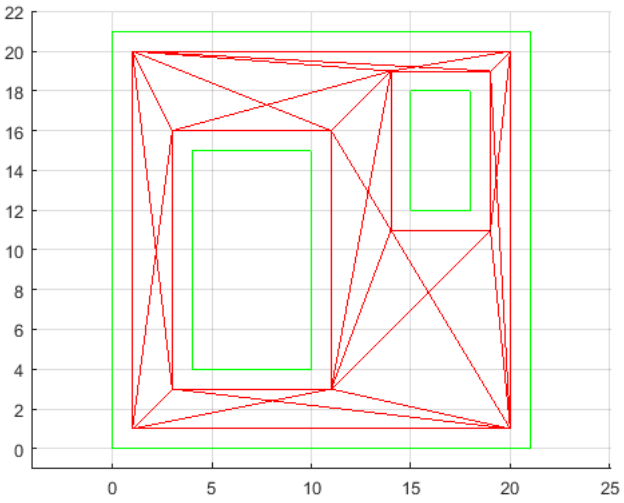


Ilustración 9. Diagrama de visibilidad sobre entorno básico.

Una vez obtenido el diagrama de visibilidad, podremos aplicar el algoritmo de Dijkstra para calcular el trayecto más rápido entre un punto inicial y un punto final.

Con el entorno expandido también se podrá crear un árbol con el algoritmo de los árboles de búsqueda rápidamente explorables, o con el método de los mapas de caminos probabilísticos.

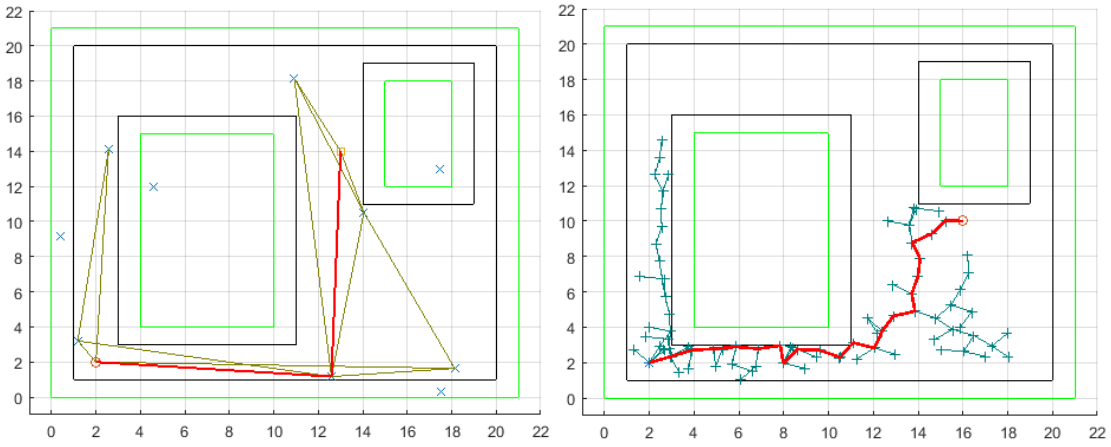


Ilustración 10. RRT y PRM en entorno básico.

Desarrollo del entorno para un robot cilíndrico

Un entorno se considera un conjunto de obstáculos junto con un escenario. Se entiende que el objetivo de un entorno es visualizar y parametrizar el espacio libre, por donde el robot pueda transportarse sin colisiones por el interior de dicho espacio.

Creación de un entorno para un robot cilíndrico

En primer lugar, se desarrolla un entorno para la generación de trayectorias de un robot cilíndrico.

En este trabajo no se utilizará la matriz de entorno utilizada en el entorno básico pues de esta forma se obtienen resultados más flexibles a la hora de implementar una nueva función en el futuro. Como se ha mencionado anteriormente, uno de los objetivos de este proyecto es realizar la expansión del entorno mediante la programación orientada a objetos.

Creación de un objeto

Por ello, se crean las clases necesarias para la formación de objetos. En primer lugar, se crea la clase “Objeto”.

Un “Objeto” vendrá definido por una serie de vértices, aristas y arcos.

```
classdef Objeto
    properties
        vertices
        aristas
        arcos
    end

    methods
        function obj = Objeto(vertices, aristas, arcos)
            obj.vertices = vertices;
            obj.aristas = aristas;
            obj.arcos = arcos;
        end
    end
end
```

Código 1. Clase Objeto.m

Se ha programado un constructor de forma que la creación de un objeto sea más sencilla aportando los parámetros internos del objeto.

Para crear un objeto a su vez se debe crear una matriz de vértices, así como una matriz de aristas y una matriz de arcos.

Cada arista viene definida por los vértices que lo componen, un número que la identifica y un ángulo normal que define la dirección exterior del objeto en perpendicular a la arista. Estos parámetros se encuentran como propiedades de la clase “Arista”

```
classdef Arista
    properties
        vertices
        numero
        normal
    end

    methods
        function arista = Arista(vertices, numero, normal)
            arista.vertices = vertices;
            arista.numero = numero;
            arista.normal = normal;
        end
    end
end
```

Código 2. Clase Arista.m

como las coordenadas cartesianas del centro del arco X, Y y Z; un ángulo inicial y final correspondiente a la trazada del arco; el radio del arco desde el centro y una variable que ayudará a distinguir un arco cóncavo de un arco convexo. Al igual que con la creación de aristas, éstas son propiedades de la clase "Arco".

Al igual que para la clase "Objeto" se aporta un constructor tanto para la clase Arista como para

```
classdef Arco
    properties
        vertices
        numero
        xc
        yc
        zc
        AngInicial
        AngFinal
        radio
        concavo
    end
        methods
            function arco = Arco(vertices, numero, xc, yc, zc, angInicial,
angFinal, radio, concavo)
            arco.vertices = vertices;
            arco.numero = numero;
            arco.xc = xc;
            arco.yc = yc;
            arco.zc = zc;
            arco.AngInicial = angInicial;
            arco.AngFinal = angFinal;
            arco.radio = radio;
            arco.concavo = concavo;
        end
    end
end
```

Código 3. Clase Arco.m

la clase Arco. De esta forma la creación de un objeto está completa, creando como se ha

mencionado anteriormente una matriz de vértices, una matriz de aristas y una matriz de arcos. Todo esto se introduce en el constructor para crear un objeto de la clase objeto.

```
vertices = [1,1;1,2;2,2;2,1];
aristas = [Arista([1,2],1,3), Arista([3,4],3,0.5)];
arcos = [Arco([2,3],2,1.5,2,0,-pi,0,0.5,1), Arco([4,1],4,1.5,1,0,0,-pi,0.5,0)];
obj = Objeto(vertices, aristas, arcos);
```

Código 4. Creación de un objeto.

Para poder observar el objeto representado se implementa la función dibujarObjeto(). Esta función representa los vértices como puntos y recorre cada arista representándolas basado en los vértices a los que hace referencia del objeto, y cada arco, haciendo uso del ángulo inicial y final, el radio del arco y las coordenadas cartesianas del punto central.

En la función también se implementa una utilidad. Por la naturaleza de la orientación inexacta de las aristas, se puede encontrar ángulos normales con valores peculiares, donde su cálculo previo puede ser una tarea tediosa. Para ello, cuando se dibuja un objeto, se calcula los dos posibles valores de normal para cada arista, basándose en la orientación de ésta, y se le asigna el valor exacto tomando el valor introducido inicialmente como valor orientativo.

Para poner esto en práctica, se observa como en el *Código 4. Creación de un objeto* se introducen los valores 3 y 0.5 para los ángulos normales a dos rectas verticales, que a la hora de aplicar la función dibujarObjeto(), se corrigen a π y a 0.

Se representa el objeto creado anteriormente mediante la función *dibujarObjeto(obj)*.

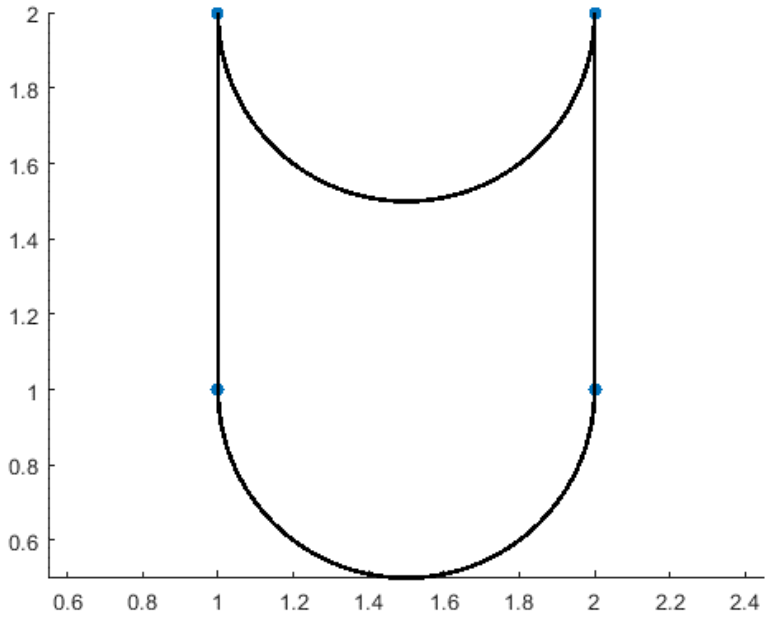


Ilustración 11. Objeto representado mediante dibujarObjeto().

Creación de un escenario

Una vez los obstáculos han sido creados mediante la clase “Objeto”, se precisa de un espacio en el que contenerlos de forma que delimite los extremos del entorno. Esto aparte de algo estético, será algo funcional, ya que será utilizado para contener los métodos de generación de trayectorias que implican la generación aleatoria de puntos, como son el método RRT o el PRM.

A la hora de crear un escenario se aplicará la creación de un obstáculo como ha sido mostrado anteriormente, con la particularidad de tener los ángulos normales orientados hacia el interior, de forma que se identifique como ocupado el espacio exterior a la figura.

El siguiente código muestra, de forma similar a los mostrados anteriormente con la particularidad mencionada, la creación de un escenario.

```
vertices = [0,4;2,4;2,5;0,5];
aristas = [Arista([1,2],1,pi/2), Arista([2,3],2,pi), Arista([3,4],3,3*pi/2),
Arista([4,1],4,0)];
arcos = [];
obj = Objeto(vertices, aristas, arcos);
```

Código 5. Creación de un escenario rectangular.

Una vez generado el escenario, se precisa de la función *dibujarObjeto()* para representar el objeto, obteniendo la siguiente figura.

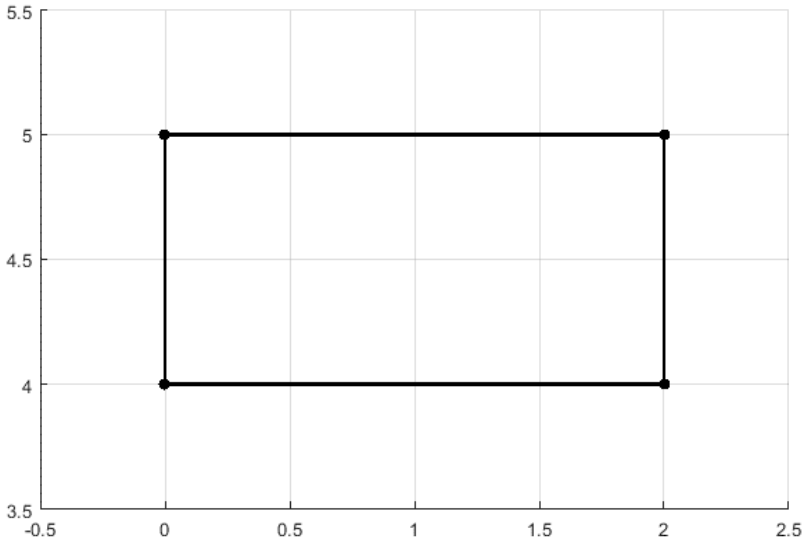


Ilustración 12. Representación de escenario rectangular.

Observando la ilustración superior no se reconoce sin previamente analizar el código que la genera, si la figura rectangular es un obstáculo o un espacio libre. Esto no será un problema, pues una vez expandido el entorno completo, se puede reconocer el lado por el que se delimita el objeto, o en este caso, el escenario.

Expansión del entorno para un robot cilíndrico

A continuación, se expone el proceso de expansión de un objeto para un robot cilíndrico.

Creación del objeto expandido

Después de haber implementado la generación de objetos y la generación de escenarios, se procede a la expansión individual de cada uno de ellos para el movimiento de un robot cilíndrico. Para ello, se va a realizar una pequeña reflexión lógica de los siguientes pasos.

El hecho de expandir un objeto conlleva delimitar un nuevo perímetro, donde el punto central del robot, que es su origen de coordenadas o representación puntual equivalente, pueda recorrer el nuevo perímetro manteniendo contacto con el objeto, pero sin colisionar, o lo que es lo mismo, a la distancia del radio del robot cilíndrico.

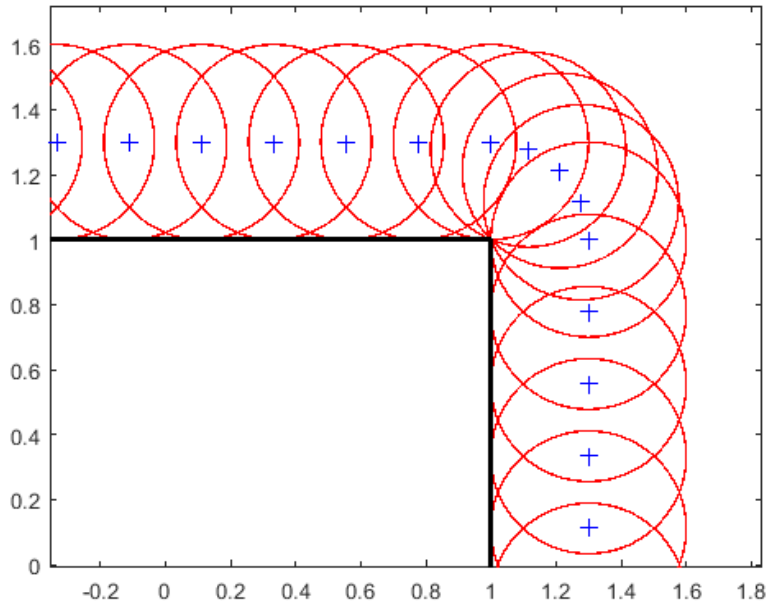


Ilustración 13. Demostración de área expandida con circulo tangente.

Como se puede observar, esto provoca una curvatura alrededor de los vértices, que es la equivalencia al robot manteniendo el vértice como punto tangente.

A la hora de expandir el objeto, se pueden hallar tres posibles situaciones: encontrar un arco, encontrar una arista o encontrar un vértice.

La arista

El caso más simple es el de la arista. En esta situación se hará uso del radio del cilindro del robot, así como la normal y los vértices de la arista. La *Ilustración 13. Demostración de área expandida con circulo tangente* muestra cómo el área expandida debe definirse como una recta de iguales dimensiones a la Arista, por lo que se podrá definir de igual manera como Arista. También demuestra que esta línea estará a una distancia igual al radio de la Arista original, y que la dirección en la que se expande es de forma directa el ángulo normal de la recta.

Por esto, se concluye que, para la expansión de una Arista, se creará un vector unitario del ángulo normal, que se multiplica por el radio del cilindro que compone el robot. Después se trasladan los vértices de la arista el valor de este vector.

Llegados a este punto, en un caso estándar donde se expanda un cubo, puede ser razonado rápidamente cómo cada vértice deberá desplazarse para cada una de las aristas, a un punto diferente. Por ello, se concluye que el número de vértices se multiplicará por dos. Para esto se propone el siguiente sistema, para el vértice inicial de una arista, se utilizará un valor doble del valor actual de la arista menos uno. De esta forma, la primera arista se mantendrá como la primera, la segunda pasará a la tercera, la tercera pasará a la quinta, y así sucesivamente. En el caso del vértice final de la arista, se multiplicará el valor por dos únicamente, de forma que la primera pase a la segunda, la segunda a la cuarta, y la tercera a la sexta.

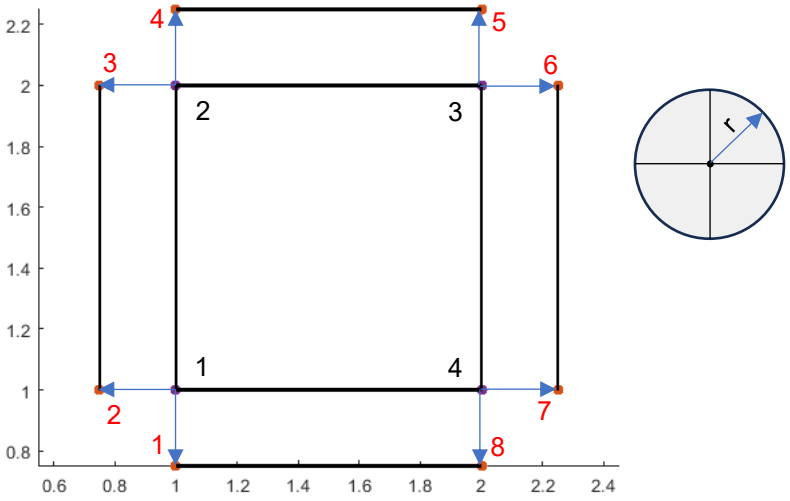


Ilustración 14. Ejemplo de expansión de las aristas de un objeto para un robot cilíndrico.

Después de este proceso, si el objeto no tiene arcos, se dejará como guardados los vértices del objeto expandido.

Los arcos

En caso de existir arcos, la aplicación dependerá de la concavidad del arco. Para ello se cuenta con el siguiente ejemplo.

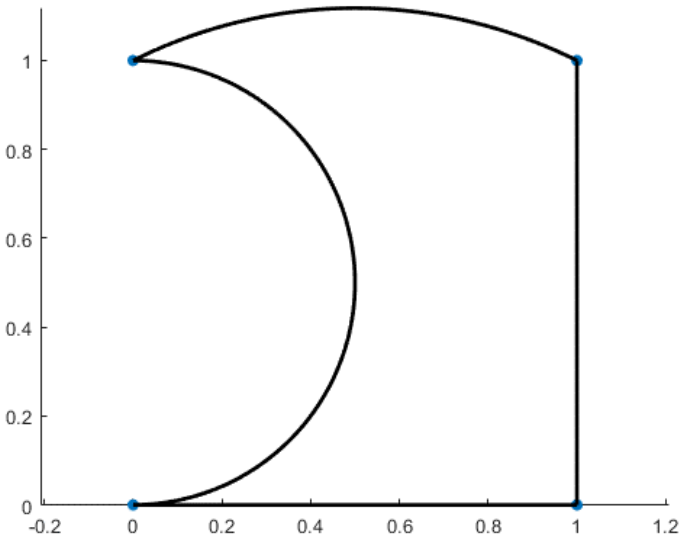


Ilustración 15. Ejemplo de objeto con arco cóncavo y convexo.

Para el caso de un arco cóncavo, considerando el punto central del mismo y su radio, así como su ángulo inicial y final, se podrá observar la siguiente relación. A la hora de expandir el arco, se debe mantener una distancia constante igual al radio del robot, restando así al radio del arco perteneciente al objeto, el valor del radio del robot. En el siguiente ejemplo se podrá observar dicha expansión.

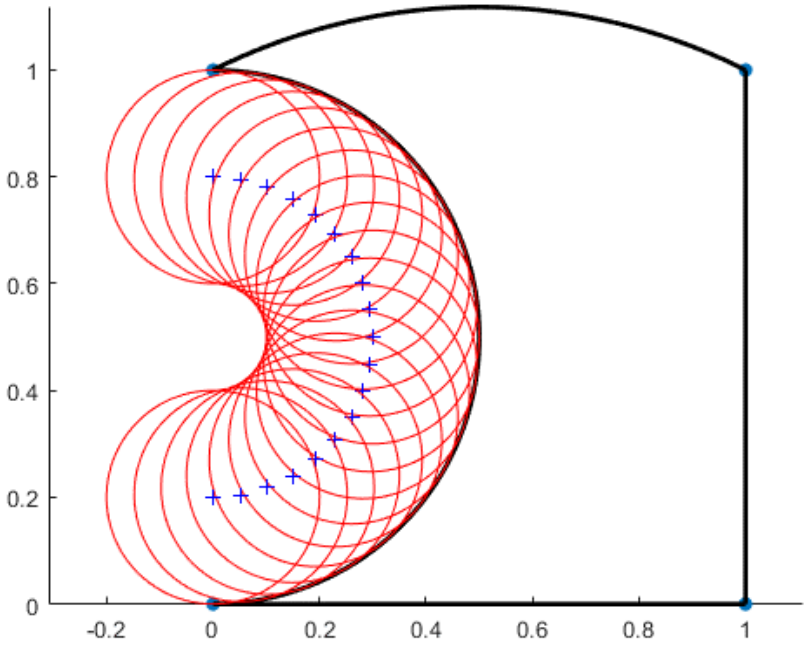


Ilustración 16. Demostración de arco cóncavo expandido en un objeto.

De la misma forma para un objeto con un arco convexo, se llega a la conclusión, por el mismo razonamiento que en el caso anterior, que, sumando el radio del robot al radio actual del arco del objeto, se obtendrá el mismo arco expandido. Se observa en la siguiente ilustración.

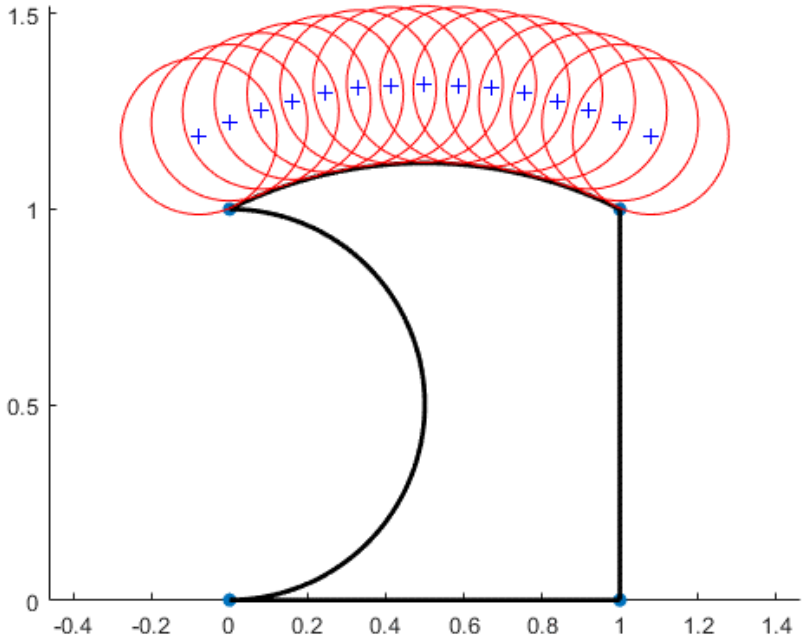


Ilustración 17. Demostración de arco convexo expandido en un objeto.

De la misma forma que se añaden vértices a la hora de expandir las aristas, en la expansión de arcos, tanto cóncavos como convexos también se verán modificados los vértices de la misma forma.

El programa será capaz de identificar si el arco es cóncavo o convexo mediante la propiedad incluida en su clase llamada “concavo”, que será verdadero en caso de ser cóncavo y falso en caso de ser convexo.

De esta forma los arcos cóncavos y convexos se verán expandidos de la siguiente manera.

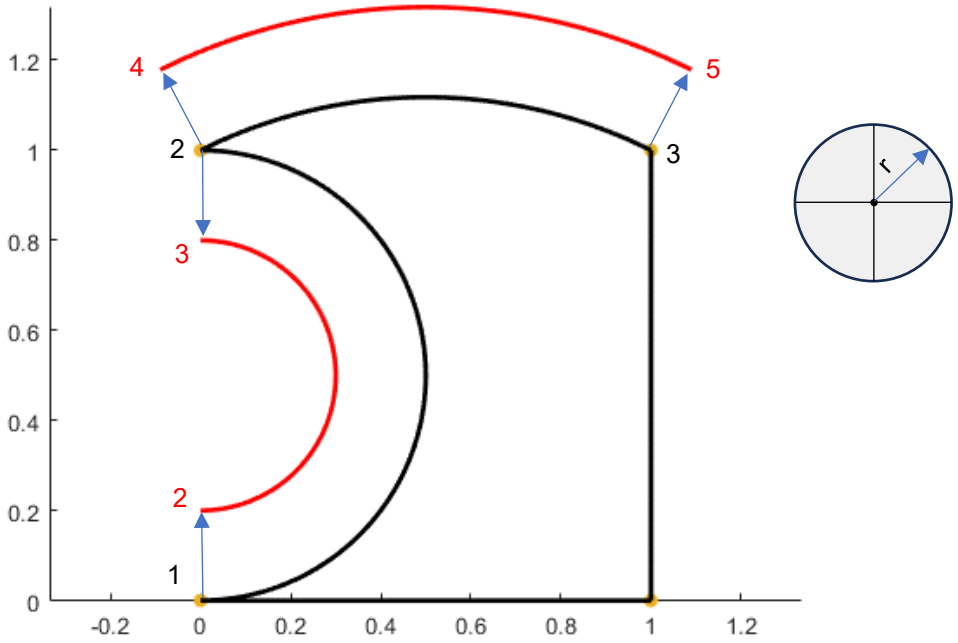


Ilustración 18. Ejemplo de expansión de los arcos de un objeto para un robot cilíndrico.

Los vértices

Por último, la única parte que falta del objeto por recorrer son los vértices, o lo que es lo mismo, todas las uniones entre los diferentes arcos y aristas del objeto.

En esta expansión no se añadirán vértices nuevos, ya que, el vértice, se podría argumentar que tan sólo une los vértices anteriormente creados por los arcos y aristas expandidos.

Para ello, de forma similar a lo realizado con los arcos y aristas, se puede llegar al razonamiento de que habrá que mantener una distancia igual al radio del robot en todo momento, implicando la creación de un área circular de dicho radio alrededor del vértice. Se puede observar de dónde nace esta área en la siguiente representación:

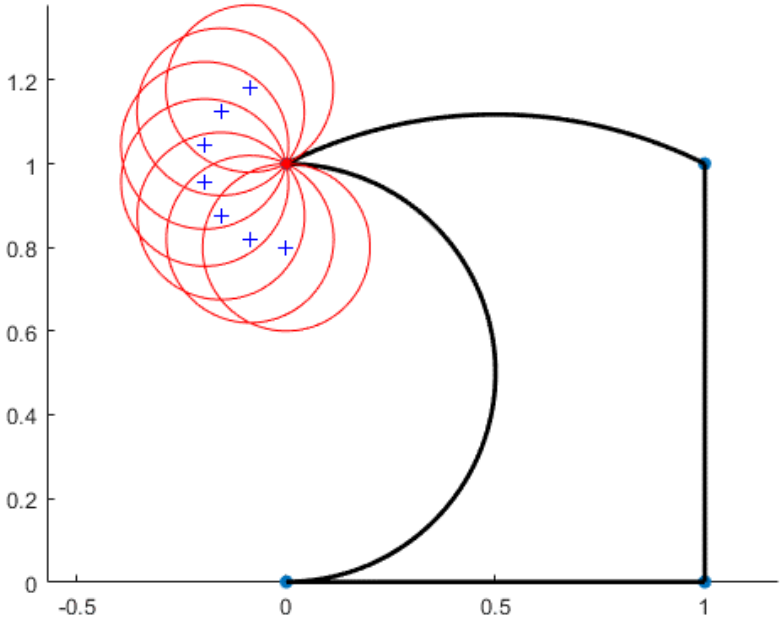


Ilustración 19. Demostración de vértice expandido en un objeto.

De esta forma la expansión de los vértices se realiza como una circunferencia de radio igual al radio del robot, como se puede apreciar en la siguiente imagen.

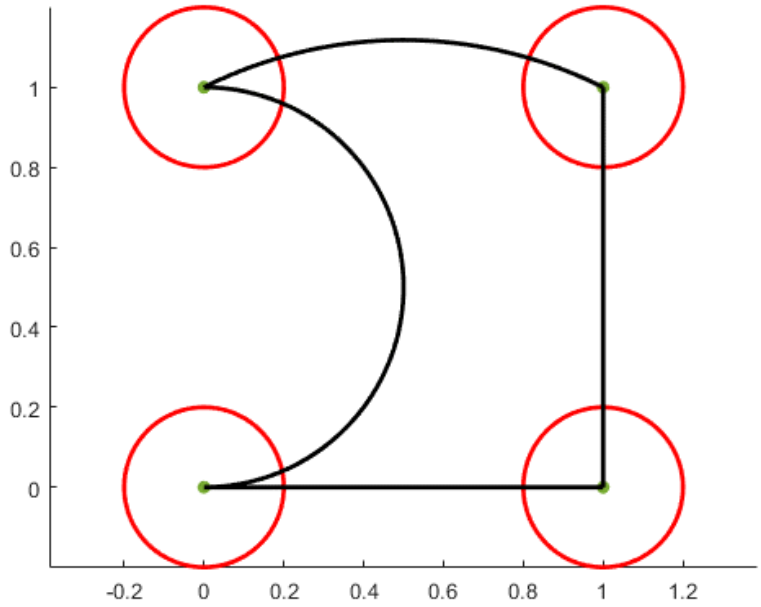


Ilustración 20. Ejemplo de expansión de los vértices de un objeto para robot cilíndrico.

Debido a la forma resultante, se concluye con que los vértices darán lugar a un nuevo arco a la hora de ser expandidos.

Después de haber expandido cada arista, arco y vértice de forma individual, se obtiene un objeto de la clase Objeto que se guardará de forma similar al objeto original. Así pues, para la figura utilizada de ejemplo anteriormente se concluye en el siguiente objeto expandido.

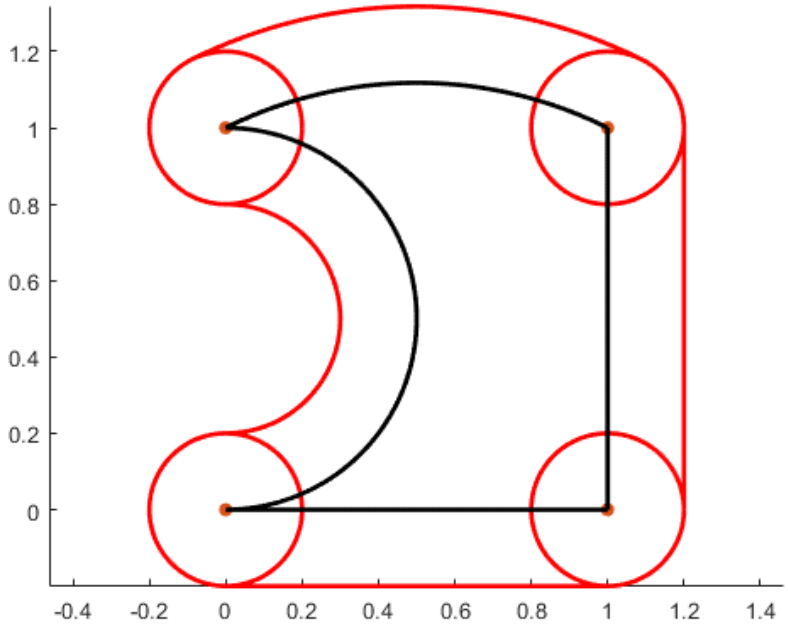


Ilustración 21. Ejemplo de expansión completa de un objeto para un robot cilíndrico.

Se puede observar cómo los vértices, al crear toda la circunferencia, no forman una figura continua. Por ello se debe depurar y limpiar.

Pulido del objeto expandido

Una vez el objeto ha sido expandido, y se ha guardado en formato de Objeto, se deben limpiar las intersecciones entre los arcos, las aristas y los vértices, de forma que quede tan solo un único perímetro.

Vértices

El primer paso será corregir los vértices. Para ello, se aprovechan las coordenadas de cada uno de los nuevos vértices generados. Se debe recordar que los vértices son tan sólo arcos que recorren una vuelta completa, formando una circunferencia. De esta forma, tomando los nuevos vértices que delimitan el arco de la expansión del vértice del objeto original, se puede calcular el ángulo inicial y ángulo final con el punto central del vértice respecto a la coordenada X, horizontal.

Se puede observar la aplicación en el ejemplo anterior.

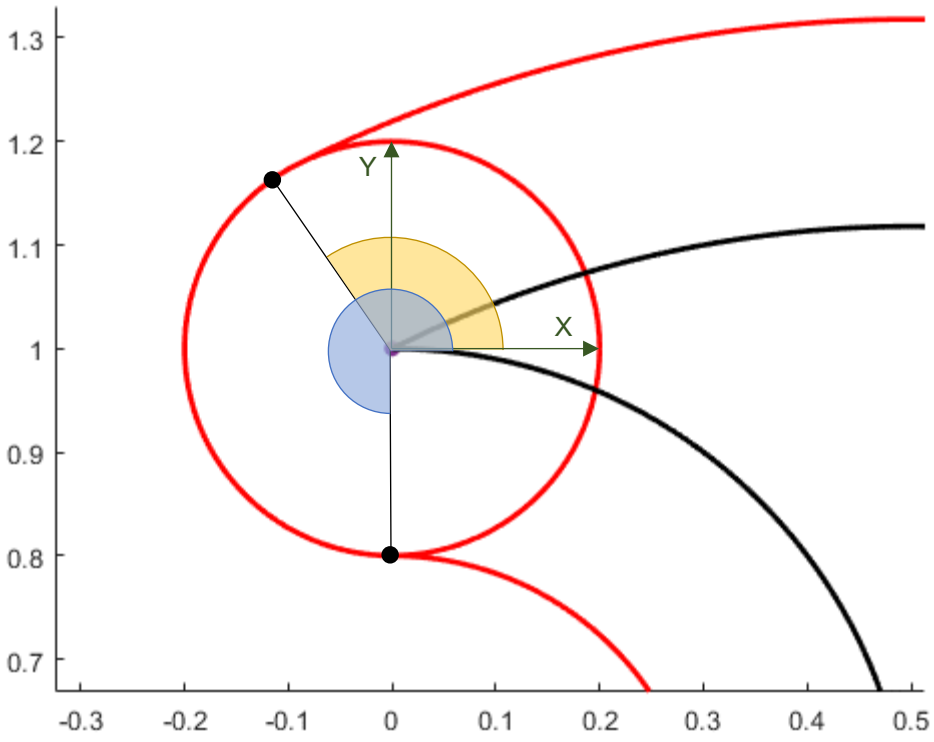


Ilustración 22. Representación de cálculo de ángulo inicial (amarillo) y ángulo final (azul) para el pulido del objeto expandido para un robot cilíndrico.

Una vez calculados el ángulo inicial y el ángulo final se asignan al arco que forma el vértice expandido. De esta forma se consigue recortar la parte sobrante. Se representa la misma figura con la parte sobrante recortada.

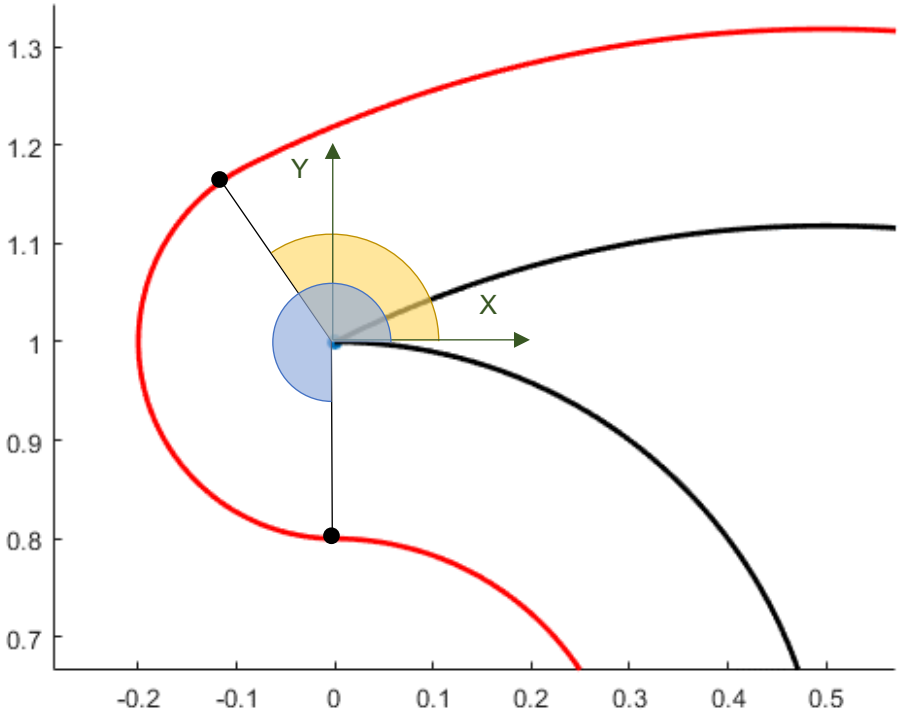


Ilustración 23. Demostración de depuración satisfactoria de un vértice expandido para un robot cilíndrico.

A la hora de implementar esta función a todos los vértices, se observa un comportamiento no deseado a la hora de depurar aquellos vértices cuyo ángulo inicial se encuentra en el tercer o

cuarto cuadrante, o sea, entre π y 2π radianes y el punto final se encuentra en el primer o segundo cuadrante, o sea, entre 0 y π radianes.

Esto se debe a que el punto inicial se encuentra después del punto final, por lo que, por la manera implementada de dibujar el arco, se dibuja de forma inversa en algunos casos. Se explica con dos situaciones con los mismos ángulo inicial y ángulo final.

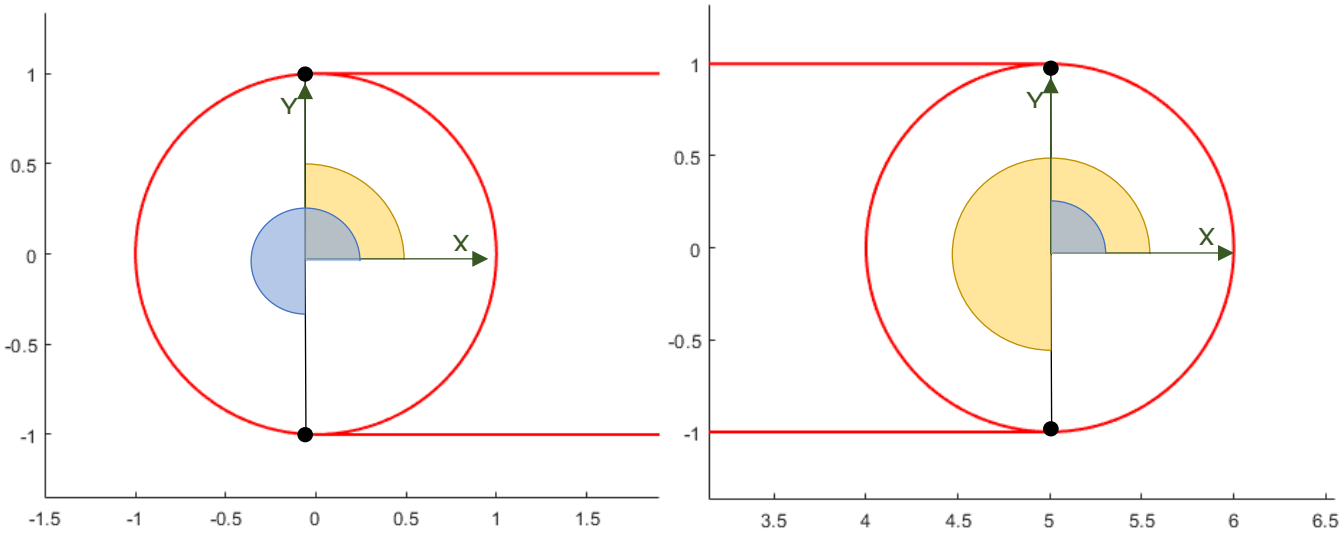


Ilustración 24. Ejemplificación de dos vértices expandidos con los mismos ángulos iniciales y finales.

Según la imagen superior, se observa cómo en la figura de la izquierda el arco debe recorrer desde π/2 hasta 3π/2, y en la figura de la derecha el arco debe recorrer desde -π/2 hasta π/2.

Esto da lugar a errores como los siguientes en la limpieza de los vértices expandidos.

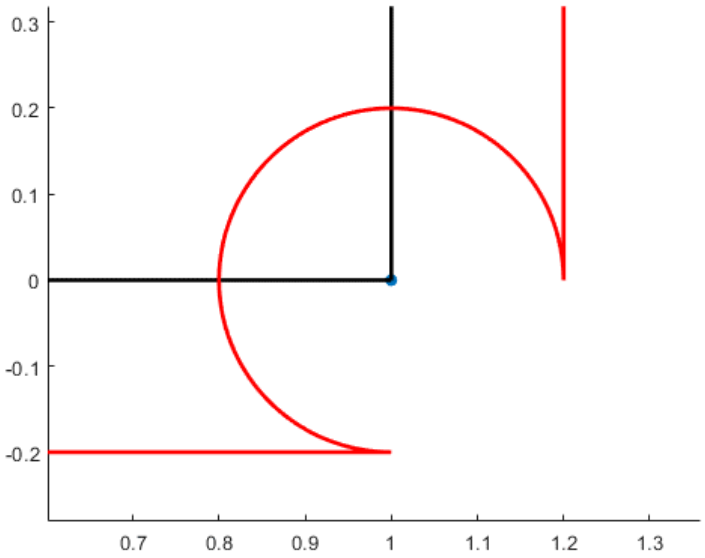


Ilustración 25. Error en la depuración de un vértice expandido para un robot cilíndrico.

Tras un breve razonamiento, se llega a la conclusión de que, debido a que los vértices expandidos deben ser tangentes a los arcos y aristas expandidos, calculando el vector dirección para cada uno de los puntos tangentes y la dirección en la que comienza la circunferencia, se podrá identificar si es la dirección correcta o contraria. Se muestra el razonamiento gráficamente.

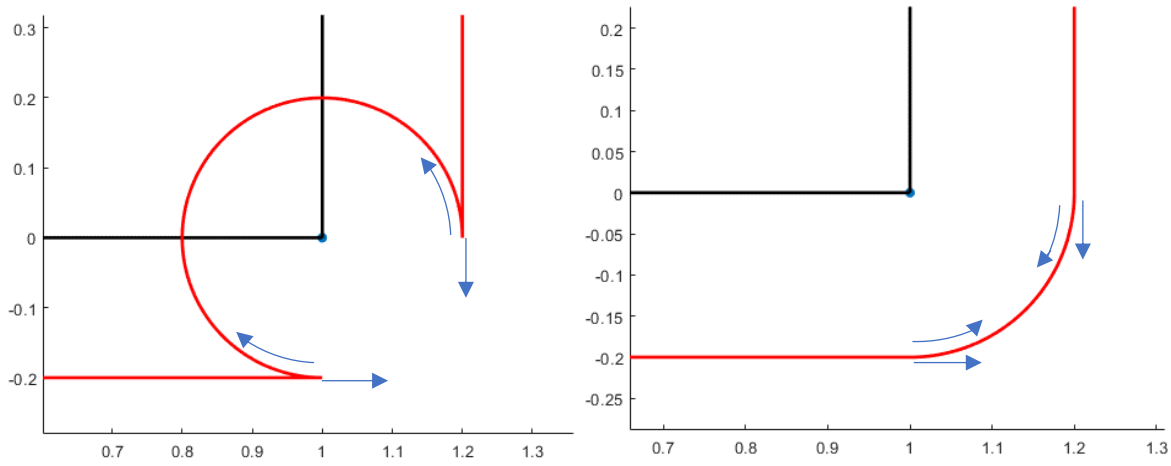


Ilustración 26. Ejemplificación de vectores finales de la arista junto a vectores iniciales del arco en una situación incorrecta (izquierda) y su posterior corrección (derecha) en la limpieza de un vértice expandido para robots cilíndricos.

Después de la limpieza de la parte innecesaria del vértice, quedará tan solo en el contorno del objeto expandido los arcos, las aristas y los vértices expandidos que serán tangentes y continuos a los dos previos. Se observa la figura final para el ejemplo anterior.

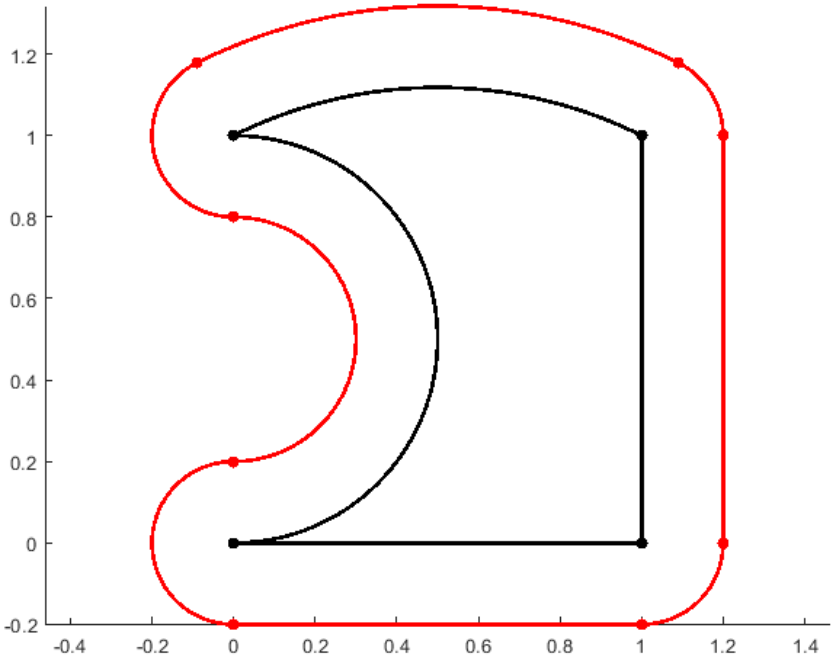


Ilustración 27. Ejemplo de figura completa expandida para un robot cilíndrico y depurada.

Intersección arco con arco

Adicionalmente, en caso de colisión entre componentes de la expansión como son una arista o un arco, también deberá tenerse en cuenta para su unión en un único perímetro. Para ello se toma como ejemplo un escenario donde se encuentran colisiones entre las expansiones de las aristas que conforman las paredes y dos las expansiones de dos arcos. Cabe tener en mente que esta figura se encuentra hueca hacia el interior.

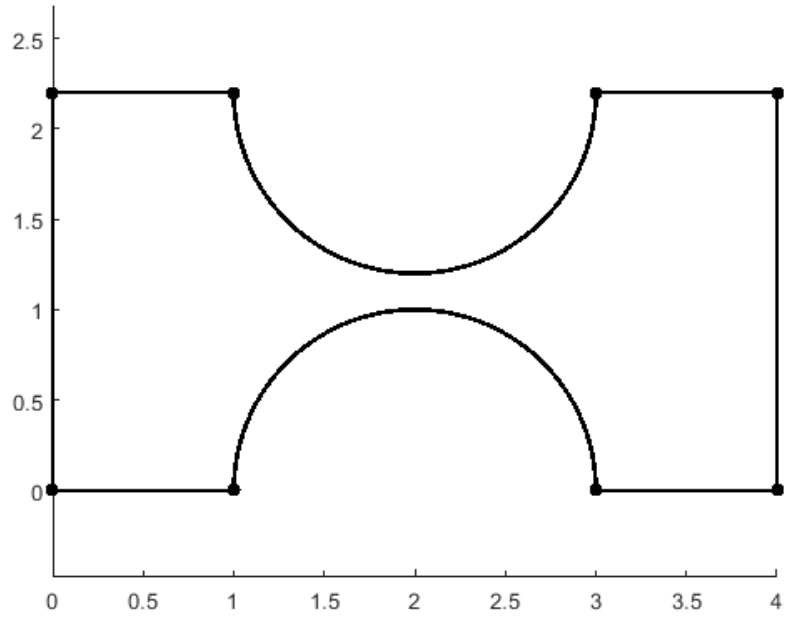


Ilustración 28. Ejemplo de intersecciones entre componentes.

Al expandir el objeto se observa como existen colisiones entre el arco superior y el arco inferior, como se observa en la siguiente figura.

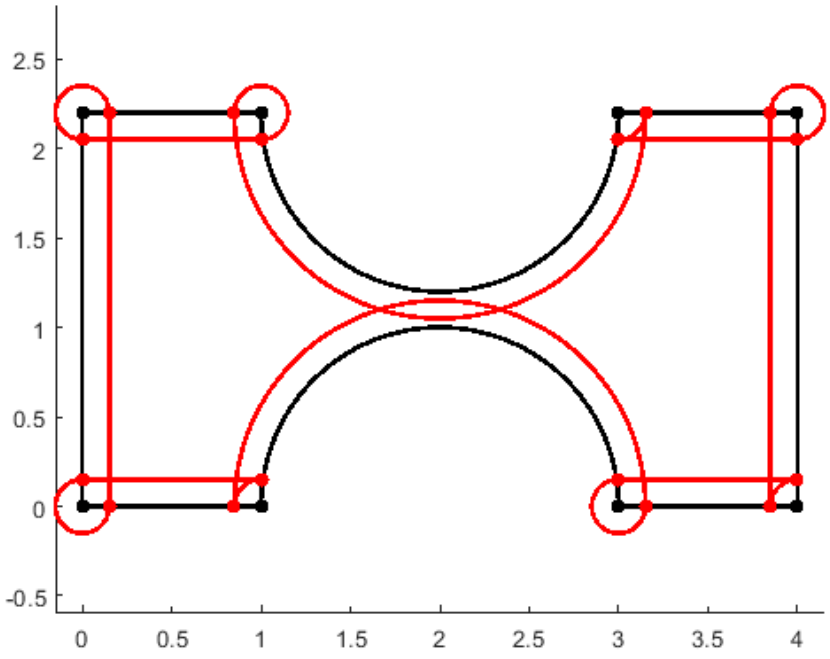


Ilustración 29. Demostración de la colisión del área expandida.

En primer lugar, se corrigen los arcos. Para ello se debe calcular el ángulo de intersección entre las áreas expandidas. Este punto se utiliza para el cálculo del ángulo inicial y ángulo final a través de los cuales no debe haber objeto, pues existe una colisión.

Para el cálculo de estos ángulos se utiliza el teorema del coseno, formando un triángulo entre el punto de colisión y los centros de ambos arcos, sabiendo que la distancia entre el punto y cada uno de los centros de cada arco es el radio del propio arco y teniendo en cuenta la distancia entre los centros de estos. Se explica de forma visual en la siguiente figura.

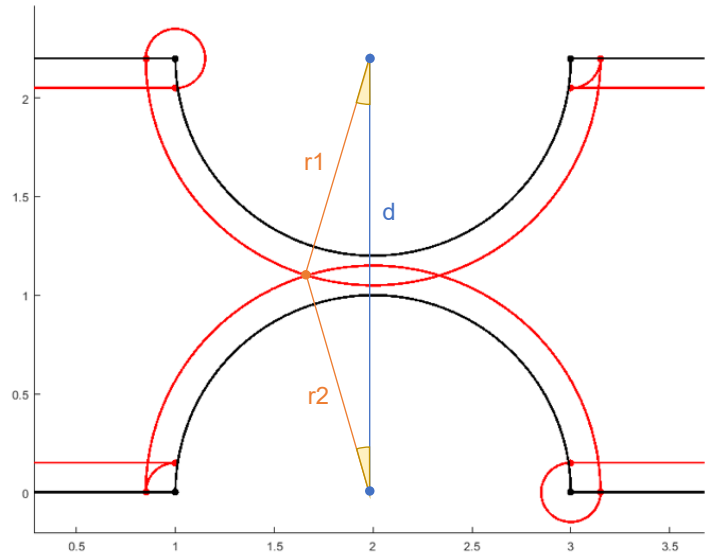


Ilustración 30. Explicación del cálculo de los ángulos de intersección para la colisión entre dos arcos expandidos para un robot cilíndrico, con los ángulos Alpha (superior) y Beta (inferior).

Una vez calculados estos ángulos se toma el vector de dirección entre un arco y el otro, y a partir del ángulo que forme con el eje X horizontal, se calculan los ángulos de cada arco en los que colisionan partiendo de los ángulos Alpha y Beta calculados anteriormente. Después se comprueba si el arco se extiende a lo largo de esos ángulos. Dependiendo de la prolongación de cada arco se modificarán los arcos de diferente manera. Siguiendo con el ejemplo propuesto, se representa la explicación de la siguiente forma.

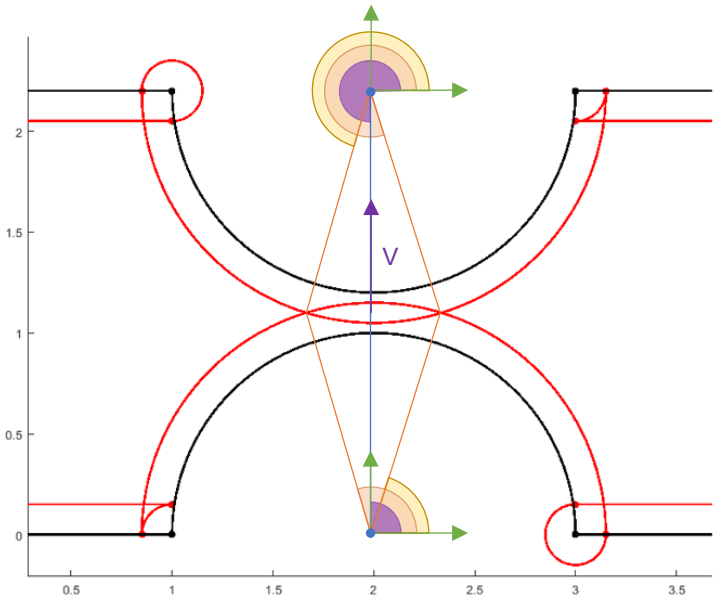


Ilustración 31. Explicación del cálculo de los ángulos iniciales (amarillo) y finales (naranja) incluyendo el vector entre los centros de los arcos y el ángulo que forman entre ellos (morado).

Una vez obtenidos los ángulos iniciales y finales, y comprobados que están dentro del recorrido del arco, se elimina la parte común, dividiendo en este caso cada uno de los arcos en dos. Se observa el resultado final en la siguiente figura.

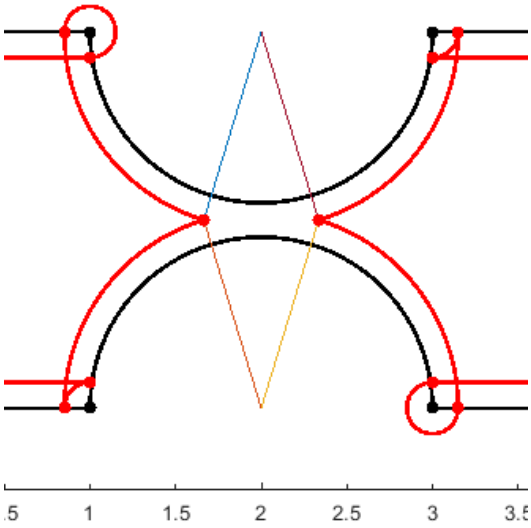


Ilustración 32. Intersección entre arcos satisfactoriamente depurada.

En la figura anterior se observa también errores en los vértices, causados por su expansión hacia el interior. Estos serán corregidos más adelante.

Intersección arista con arista

De la misma forma se debe depurar las colisiones entre aristas. Para ello se presenta un escenario de ejemplo.

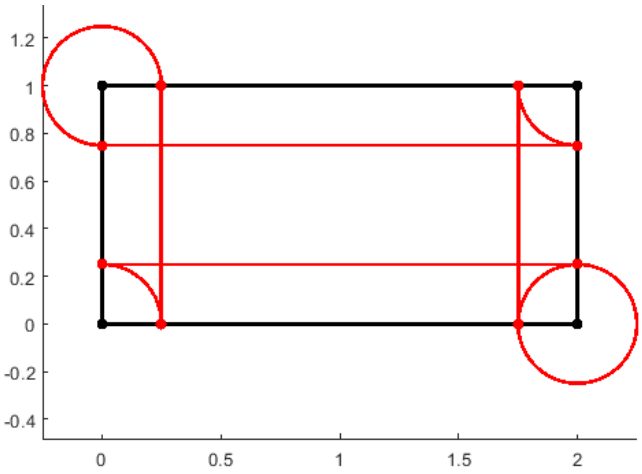


Ilustración 33. Ejemplo de errores en la expansión aristas por colisión.

Una vez más en esta figura se observa cómo los vértices no tienen un comportamiento correcto. Esto será corregido más adelante.

Para corregir la colisión entre las áreas expandidas se calcula primero la intersección de las rectas mediante la función *interseccion2d()*. En esta fórmula no sólo se calcula el punto de intersección, sino que además se comprueba que éste pertenezca a ambas rectas. De esta forma, se calculan ambas rectas mediante su definición por los vértices (lo que sería equivalente a un punto inicial y un punto final), y posteriormente se calcula la distancia desde la recta hasta el punto de intersección. En caso de encontrarse fuera de cualquiera de las rectas, el punto no sería válido.

El siguiente paso es corregir los vértices, debido a que las aristas son definidas basándose en los valores de vértice del objeto al que pertenecen. Para ello debe compararse uno de los vértices

de la recta para comprobar si se debe modificar éste, o el vértice contrario. Una vez más se representa la explicación en la siguiente figura.

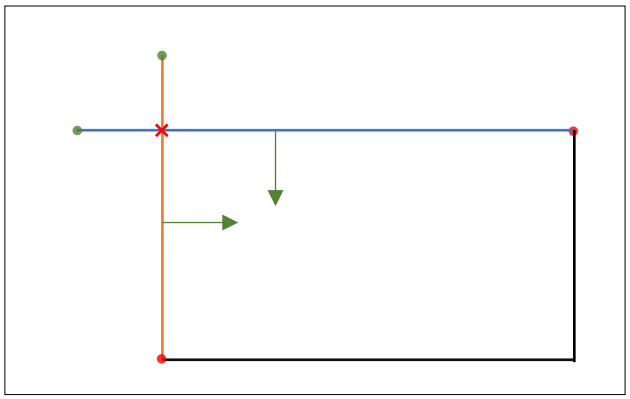


Ilustración 34. Explicación de la identificación de vértices a corregir en una intersección de aristas expandidas, con los vértices a corregir (verde), los vértices contrarios (rojo), las normales de cada recta (verde) y el punto de intersección (rojo).

En la imagen superior se puede apreciar cómo los puntos que quedan en la dirección opuesta de la normal respecto de la recta son clasificados como puntos a corregir, marcados en verde. Esto mismo, aplicado al ejemplo mencionado anteriormente resulta en la siguiente figura.

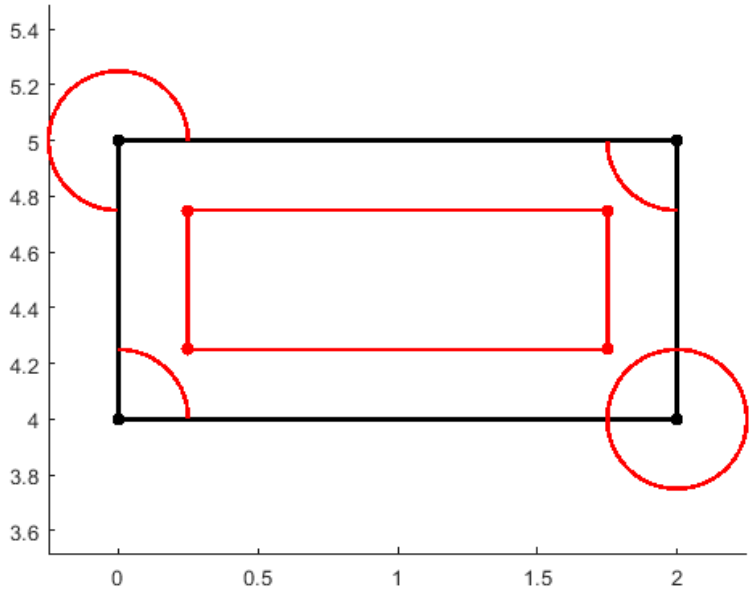


Ilustración 35. Intersección entre aristas satisfactoriamente depurada con errores en los vértices.

En esta figura se puede observar, al igual que en el ejemplo anterior para la depuración de arcos, cómo los vértices no son correctos por motivos triviales. Aplicando esta solución para la colisión entre aristas en el ejemplo anterior para arcos devuelve el siguiente resultado.

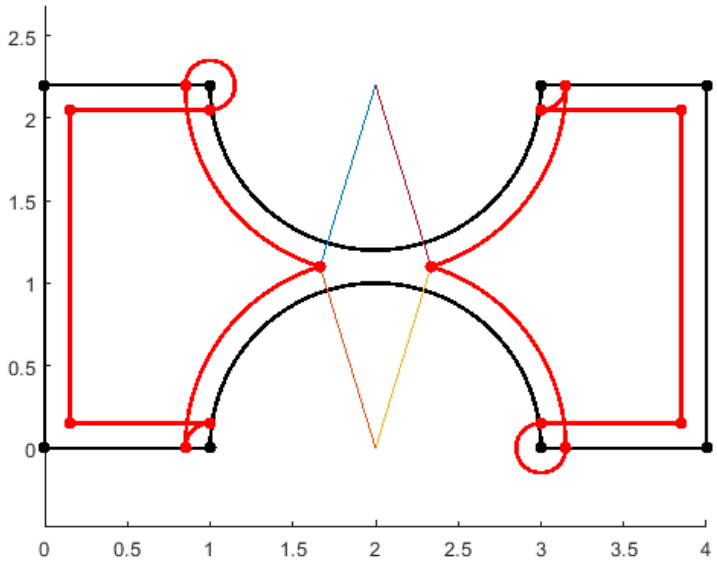


Ilustración 36. Depuración de intersecciones entre aristas para objeto expandido.

En esta figura, además se han retirado los vértices correspondientes a las intersecciones entre aristas, de forma que no resulten las figuras circulares que se han podido apreciar anteriormente.

Intersección arco con arista

De forma similar a lo realizado anteriormente, se podrá aplicar una mezcla entre las dos técnicas de manera que se puedan depurar las intersecciones entre arco y recta. Para ello, se tomará como referencia la recta y el punto central del arco, y se calcula si algún punto de la recta está a exactamente un radio del arco. En caso de existir tan solo un punto significará que la recta es tangente al arco, por lo que no será interesante interpretarlo como una intersección. En el caso objetivo, se tratará con dos puntos, de modo que tendremos una "cuerda" en la circunferencia.

Una vez obtenidos ambos puntos, se calculará si pertenecen al recorrido del arco, así como de la recta, y en caso de ser así, se calculará que lado del arco es el lado "ocupado" de la recta, o lo que es lo mismo, el lado contrario respecto al vector normal de la recta.

Una vez más se recurre a un gráfico para apoyar la explicación.

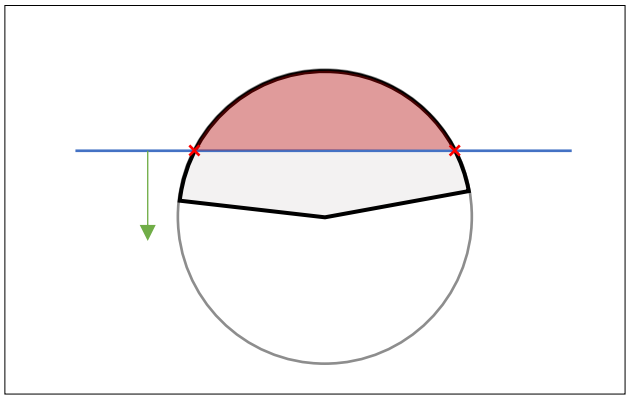


Ilustración 37. Explicación de colisiones con circunferencia, y recorrido de arco, así como parte a eliminar.

De esta forma, en caso de tener colisión en dos puntos y el vector normal como en el ejemplo, se dividirá tanto el arco como la recta en dos piezas, una a cada lado de la recta. Esto aplicado al ejemplo anterior queda según se refleja en la siguiente ilustración:

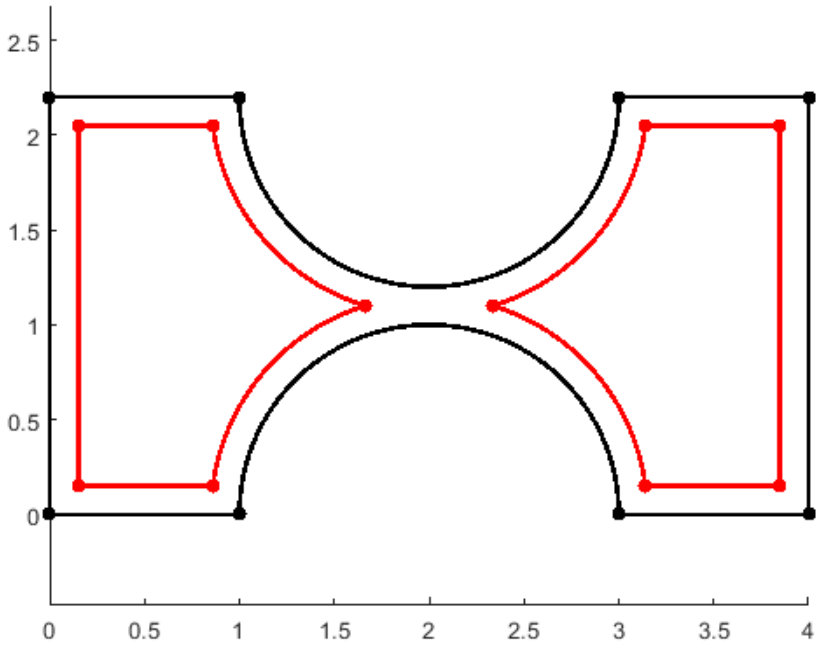


Ilustración 38. Figura totalmente depurada.

Como se puede observar, en las intersecciones entre el arco y las aristas, tanto la arista como el arco han sido recortados hasta la intersección entre ellas. Además, se ha eliminado el vértice expandido.

Un ejemplo más similar a la explicación propuesta anteriormente sería el siguiente.

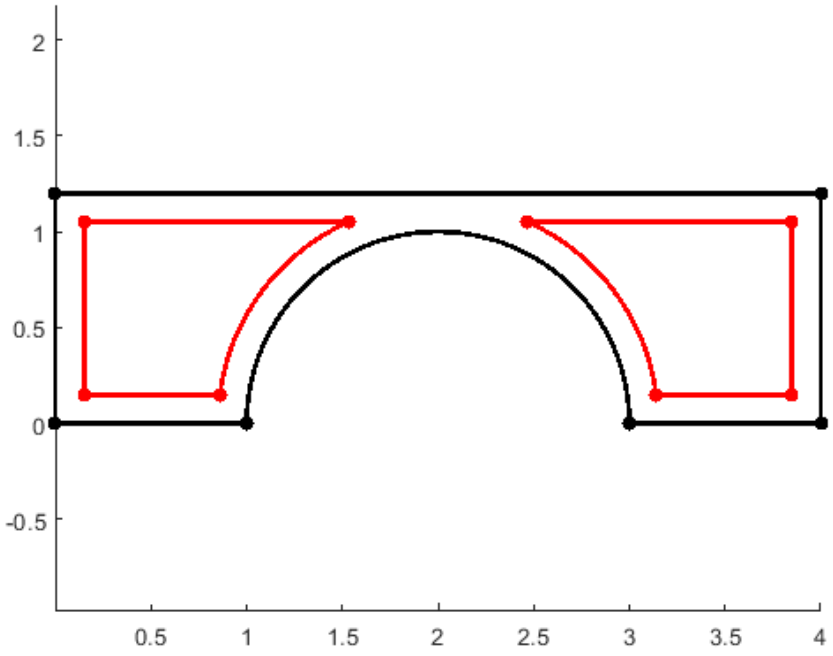


Ilustración 39. Ejemplo de intersección entre arco y recta depurada satisfactoriamente.

Para la expansión de objetos se utiliza la función *expandirObjeto()*, y para la contracción de escenarios se utiliza la función *contraerObjeto()*. Posteriormente se podrá utilizar la función *limpiarObjeto()* para depurar los bordes sobrantes, intersecciones y demás vistos anteriormente.

Creación del entorno

Una vez los objetos deseados y el escenario han sido creados, expandidos y depurados se podrán incluir en un Entorno.

Un entorno será una variable de Matlab de la clase Entorno. Esta clase tiene como propiedades una matriz unidimensional de los objetos originales que lo conforman, y una matriz unidimensional de los objetos expandidos que lo conforman. Éstos deberán estar en el mismo orden para mayor cohesión, aunque no es estrictamente necesario. En la primera posición de la matriz de cada una de las propiedades deberá ir el escenario y el escenario expandido, respectivamente.

```
classdef Entorno
    properties
        objetos
        objetosExpandidos
    end

    methods
        function entorno = Entorno(objetos,objetosExpandidos)
            entorno.objetos = objetos;
            entorno.objetosExpandidos = objetosExpandidos;
        end

        function entorno = dibujarEntorno(entorno)
            for i = 1:length(entorno.objetos)
                dibujarObjeto(entorno.objetos(i));
            end
            for i = 1:length(entorno.objetosExpandidos)
                dibujarObjeto(entorno.objetosExpandidos(i),'r');
            end
        end
    end
end
```

Código 6. Clase Entorno.m

La clase Entorno incluye, además de su constructor, una función llamada *dibujarEntorno()* la cual recorre cada uno de los objetos y los dibuja en negro, y cada uno de los objetos expandidos y los dibuja en rojo.

De esta forma se puede hacer uso de esta clase con el siguiente código para la representación de un entorno expandido para un robot cilíndrico.

```
radio = 0.45;

vertices = [0,0;10,0;10,10;0,10];
aristas = [Arista([1,2],1,pi/2), Arista([2,3],2,pi), Arista([3,4],3,3*pi/2),
Arista([4,1],4,0)];
escenario = Objeto(vertices, aristas, []);
escenarioExpandido = contraerObjeto(escenario, radio);
escenarioLimpio = limpiarIntersecciones(escenarioExpandido);

vertices = [3,3;7,3;7,6;3,6];
aristas = [Arista([1,2],1,3*pi/2), Arista([2,3],2,0), Arista([3,4],3,pi/2),
Arista([4,1],4,pi)];
obj1 = Objeto(vertices, aristas, []);
objExpandido = expandirObjeto(obj1, radio);
objLimpio1 = limpiarIntersecciones(objExpandido);

vertices = [1,8;3,8;2,9];
aristas = [Arista([1,2],1,3*pi/2), Arista([2,3],2,pi/4), Arista([3,1],3,3*pi/4)];
obj2 = Objeto(vertices, aristas, []);
objExpandido = expandirObjeto(obj2, radio);
objLimpio2 = limpiarIntersecciones(objExpandido);


entorno = Entorno([escenario,obj1,obj2],[escenarioLimpio,objLimpio1,objLimpio2]);
clear aristas escenario escenarioLimpio escenarioExpandido obj obj1 obj2
clear objExpandido objLimpio objLimpio1 objLimpio2 vertices
entorno = dibujarEntorno(entorno);
```

Código 7. Código de generación de un entorno.

En este código se observa cómo se define un radio para el robot, el cual después se utiliza para calcular la expansión de los objetos, o contracción en caso del escenario. Se declara cada uno de los obstáculos, en este caso hay dos, y el escenario. Por último, se crea el objeto de clase Entorno y se dibuja el entorno completo. Adicionalmente, se llama al comando *clear* para limpiar las variables temporales que, una vez creado, se encuentran en el objeto de clase Entorno. El resultado se observa en la siguiente figura.

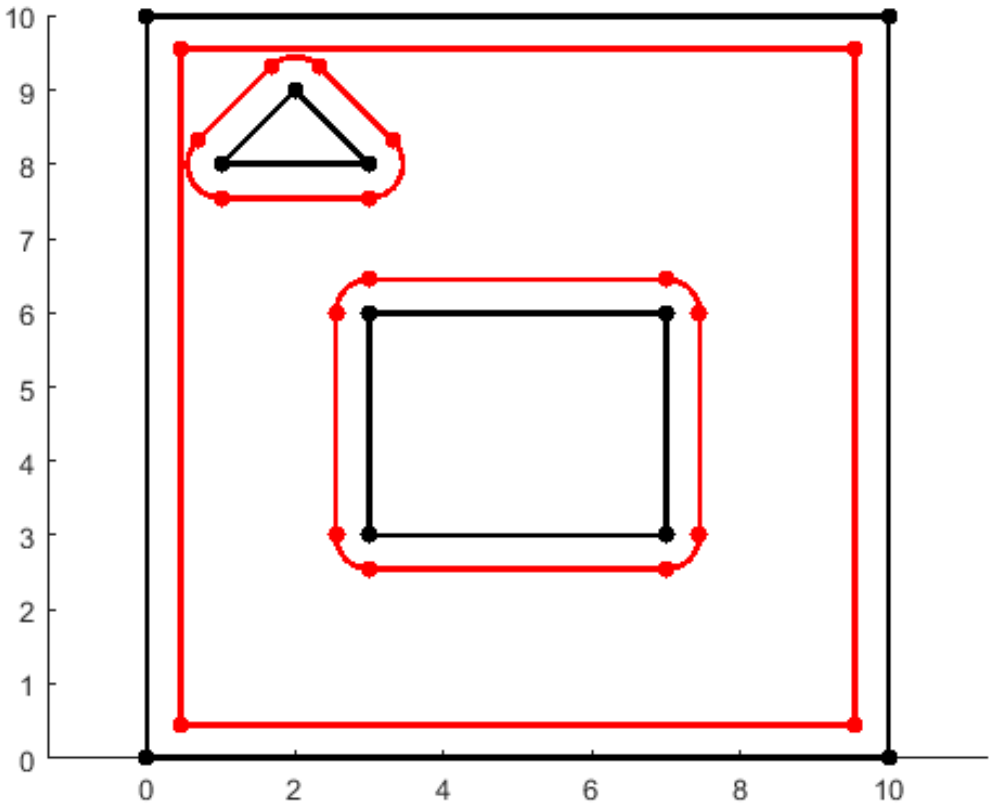


Ilustración 40. Entorno expandido para un robot cilíndrico satisfactoriamente.

Desarrollo del entorno para un robot poligonal

A continuación, se expone el proceso para la expansión de un entorno poligonal.

Creación de un entorno para un robot poligonal

Nuevamente, se crea un entorno para la generación de trayectorias de un robot, en este caso poligonal. Para ello se toma como base algunas de las propiedades de los objetos de las clases anteriores, además de ligeras modificaciones que simplifican el proceso de creación de objetos, así como la expansión de los obstáculos.

Se recuerda que, el objetivo de expandir un entorno es poder considerar el robot como un punto y de esta forma agilizar los cálculos de trayectorias.

Creación de un objeto

Al igual que en la implementación para un robot cilíndrico, el primer paso es la creación de una clase que permita definir un obstáculo, para su posterior expansión.

En este caso, se tendrá en cuenta que un obstáculo en el entorno para un robot poligonal será exclusivamente conformado por aristas y no contendrá curvas. Esto será para mantener una perspectiva sencilla y se deja abierto a futuros proyectos el desarrollo de esta área.

Por ello, el objeto no tendrá la necesidad de albergar arcos, por lo que estará exclusivamente formado por aristas y vértices. Si seguimos el proceso de razonamiento, dichas aristas se componen de dos vértices y un ángulo normal. Por esto mismo, un objeto de este tipo se puede descomponer en una matriz unidimensional de vértices, y una matriz unidimensional de ángulos normales. De esta forma, uniendo los vértices y aplicando los ángulos normales en orden, se obtiene el objeto completo.

En el código del constructor de estos obstáculos encontramos que la única variable de entrada es la matriz unidimensional de vértices, la cual debe ser introducida en el sentido de las agujas del reloj. De esta forma, en la creación del objeto, se calculan las normales de forma automática y se asignan al obstáculo. El código queda de la siguiente forma.

```
classdef ObstaculoPoligonal
    properties
        Vertices
        Normales
    end
    methods
        function Obstaculo = ObstaculoPoligonal(Vertices)
            % Los vertices se introducen en el sentido de las agujas del reloj
            Obstaculo.Vertices = Vertices;
            for i=1:length(Vertices)-1
                numerador = Vertices(i,1)-Vertices(i+1,1);
                denominador = Vertices(i,2)-Vertices(i+1,2);
                normal = atan2(-numerador,denominador);
                if(normal < 0)
                    normal = normal + 2*pi;
                end
                Obstaculo.Normales(i) = normal;
            end
            if(length(Vertices)>1)
                numerador = Vertices(length(Vertices),1)-Vertices(1,1);
                denominador = Vertices(length(Vertices),2)-Vertices(1,2);
                normal = atan2(-numerador,denominador);
                if(normal < 0)
                    normal = normal + 2*pi;
                end
                Obstaculo.Normales(length(Vertices)) = normal;
            end
        end
    end
end
```

Código 8. Clase ObstaculoPoligonal.m

De la misma manera, a la hora de dibujar un obstáculo tan solo se debe seguir sus vértices en orden, y cerrar la figura en el último. Se demuestra la creación de la figura y su posterior representación en el siguiente código.

```
VerticesObstaculo1 = [6, 6; 6, 7; 7, 7; 7, 6];
Obstaculo1 = ObstaculoPoligonal(VerticesObstaculo1);
dibujarPoligonal(Obstaculo1,'k');
```

Código 9. Creación y representación de un obstáculo en un entorno para un robot poligonal.

De esta forma, su representación será la unión de los cuatro vértices en orden, formado un cuadrado.

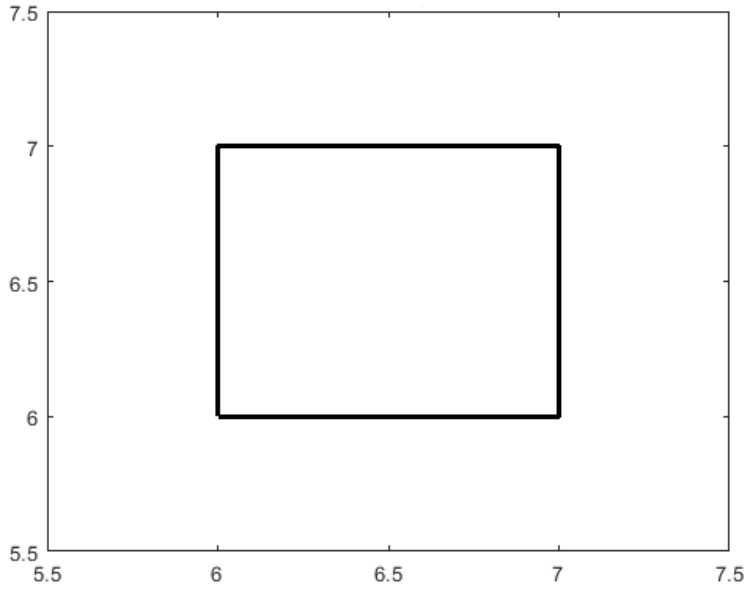


Ilustración 41. Representación de obstáculo para un robot poligonal.

Creación de un escenario

Para la creación de un escenario, debido a la simplificación del entorno, se ha optado por crear una función que, utilizando las coordenadas de la esquina inferior izquierda y la esquina superior derecha, cree un escenario cuadrado. Esta función además expandirá el escenario, y devolverá tanto el escenario básico como el escenario expandido. La expansión del escenario se comentará más adelante.

El escenario se podrá crear y representar de la siguiente forma.

```
[escenario, escenarioContraido] = crearEscenario([0,0], [10,10], Robot);
dibujarPoligonal(escenario);
dibujarPoligonal(escenarioContraido, 'r');
```

Código 10. Creación y representación de un escenario.

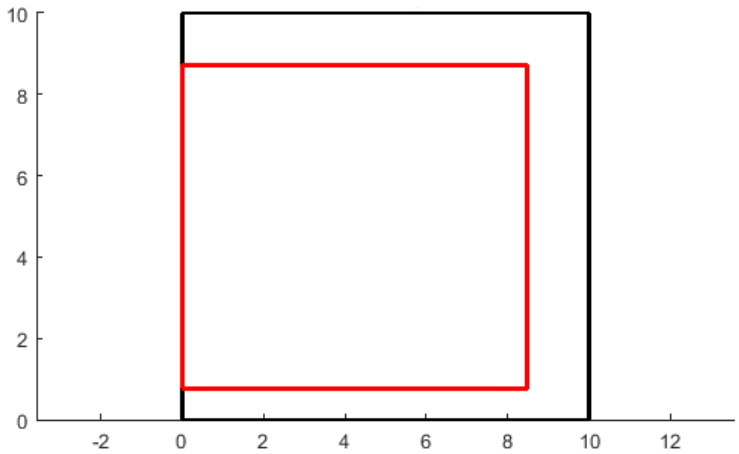


Ilustración 42. Ejemplo de representación de un escenario poligonal expandido para un robot poligonal.

Creación de un robot

En este caso, debido a la peculiaridad del robot, y siendo este un polígono con lados variables, es necesario definirlo de forma precisa mediante una clase.

En la clase RobotPoligonal se encuentran unos parámetros similares a los de la clase ObstaculoPoligonal, y de la misma forma, se definen los vértices del robot en una matriz unidimensional, con la diferencia de que, en este caso, se introducirán en el sentido contrario a las agujas del reloj. Esto se debe a que los ángulos normales en este caso estarán dirigidos hacia el interior del robot, en lugar de hacia el exterior como sucede en los obstáculos.

```
classdef RobotPoligonal
    properties
        Vertices
        Normales
    end
    methods
        function Robot = RobotPoligonal(Vertices)
            % Los vertices se deben introducir en el sentido contrario a
            % las agujas del reloj
            Robot.Vertices = Vertices;
            for i=1:length(Vertices)-1
                numerador = Vertices(i,1)-Vertices(i+1,1);
                denominador = Vertices(i,2)-Vertices(i+1,2);
                normal = atan2(-numerador,denominador);
                if(normal < 0)
                    normal = normal + 2*pi;
                end
                Robot.Normales(i) = normal;
            end
            if(length(Vertices)>1)
                numerador = Vertices(length(Vertices),1)-Vertices(1,1);
                denominador = Vertices(length(Vertices),2)-Vertices(1,2);
                normal = atan2(-numerador,denominador);
                if(normal < 0)
                    normal = normal + 2*pi;
                end
                Robot.Normales(length(Vertices)) = normal;
            end
        end
    end
end
```

Código 11. Clase RobotPoligonal.m

Al igual que en el caso de los obstáculos para robots poligonales, en este caso se podrá representar el robot mediante la función *dibujarPoligonal()*.

De ahora en adelante el robot será representado en color azul para distinguirlo del entorno, que será negro en su forma sin expandir, y rojo en su forma expandida. Además, se representará respecto del eje de coordenadas (0,0), siendo este su punto de referencia, el cual servirá para delimitar los objetos expandidos.

De esta forma un ejemplo de creación y representación de un robot con forma de triangulo rectángulo es el siguiente.

```
VerticesRobot = [0, 0; 0, -1; 2, 0];
Robot = RobotPoligonal(VerticesRobot);
dibujarPoligonal(Robot,'b');
```

Código 12. Creación y representación de un Robot Poligonal.

El resultado de ejecutar este código es el siguiente.

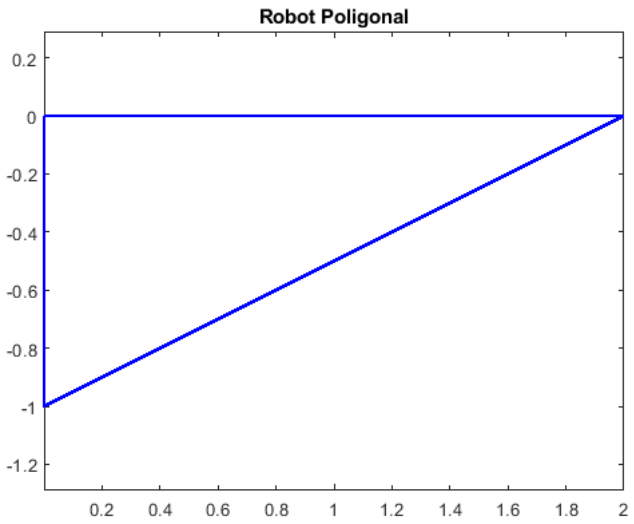


Ilustración 43. Representación de Robot Poligonal.

Adicionalmente, para poder tener en cuenta la orientación del robot, se crea la función *rotar_vector()*. Esta función toma como parámetros un Robot Poligonal y un ángulo, y devolverá un Robot Poligonal rotado dicha cantidad en el sentido contrario a las agujas del reloj.

Expansión del entorno para un robot poligonal

Al igual que en el entorno para el robot cilíndrico, en el caso del robot poligonal se deberá expandir el entorno para poder tratarlo como una unidad puntual.

Creación del objeto expandido

A la hora de expandir entornos para robots poligonales, se aplicará un algoritmo publicado por Steven M. LaValle en su libro *"Planning Algorithms"*. LaValle propone el ejemplo de un obstáculo cuadrado y un robot rectangular, de forma similar al planteado anteriormente,

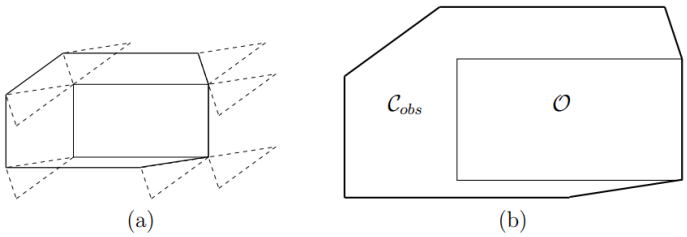


Figure 4.14: (a) Slide the robot around the obstacle while keeping them both in contact. (b) The edges traced out by the origin of $\mathcal{A}$ form $\mathcal{C}_{obs}$.

Ilustración 44. Ejemplo de robot triangular y obstáculo rectangular propuesto por Steven M. LaValle en su libro "Planning Algorithms".

En este ejemplo, como menciona él mismo, al recorrer la totalidad de la figura con el robot, y trazando su recorrido basado en el vértice de referencia, se obtiene la figura final, en la que diferencia el objeto original del espacio de configuración ocupado por la expansión de este.

Inicialmente puede parecer que esta técnica se debe llevar a cabo de la misma forma mencionada, recorriendo y detectando colisión de los obstáculos, pero rápidamente se llega a un razonamiento.

Si se presta atención al caso superior, en el lado extremo derecho del objeto, que coincide con el perímetro expandido, se puede imaginar la figura del robot deslizando verticalmente y, al coincidir el vértice inferior del obstáculo con el vértice superior izquierdo del robot, deslizar la arista superior del robot, hasta tocar ambos vértices de nuevo. Se ayuda a la explicación con una ilustración.

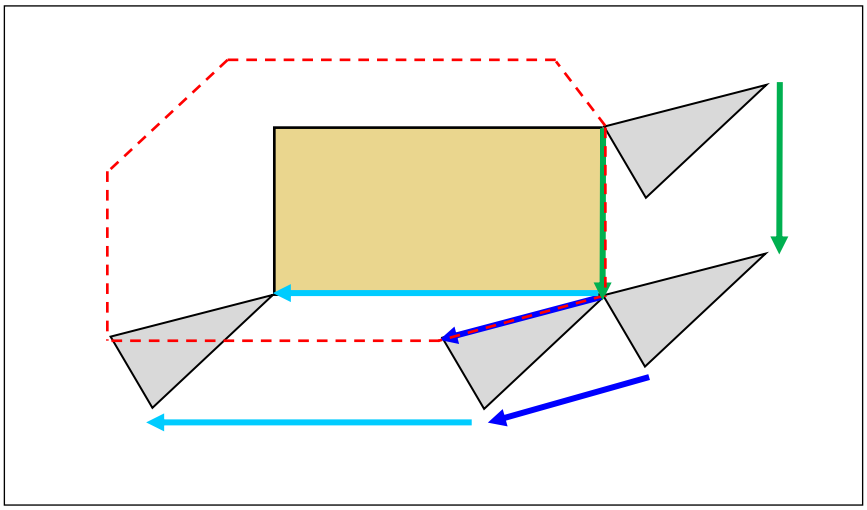


Ilustración 45. Demostración del movimiento y seguimiento de los vectores de traslación.

Prestando atención a los movimientos que realiza el objeto recorriendo el perímetro, siendo el recorrido del objeto expandido, se observa cómo es una secuencia de vectores de traslación igual a las aristas del robot y el objeto. De esta forma, tan solo queda definir un patrón en el que se aplican cada una de las aristas del robot.

Para esto, el profesor Steven M. LaValle plantea una solución basándose en los ángulos normales de las aristas pertenecientes tanto al robot como al obstáculo. Para ello se ordenan

todos los ángulos, tomando los ángulos normales hacia el exterior para el obstáculo y los ángulos normales hacia el interior para el robot poligonal.

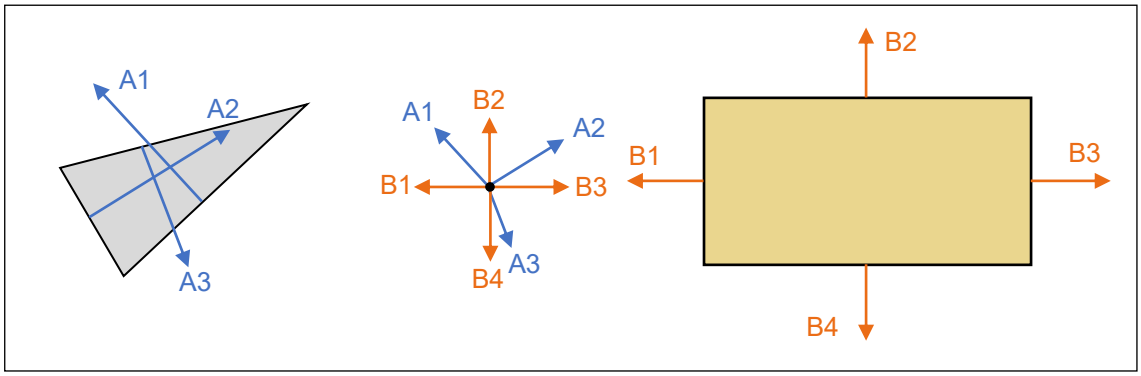


Ilustración 46. Explicación del método de expansión poligonal según Steven M. LaValle.

Este es el motivo por el que, al introducir los vértices del obstáculo, estos se introducen en el sentido de las agujas del reloj y que, al introducir los vértices del robot, se introduzcan en el sentido contrario.

Recorriendo los ángulos normales en orden, y añadiéndolos a la figura, se obtiene la figura expandida, esto se puede demostrar fácilmente con una representación del área expandida donde se muestren los vectores de la ilustración anterior.

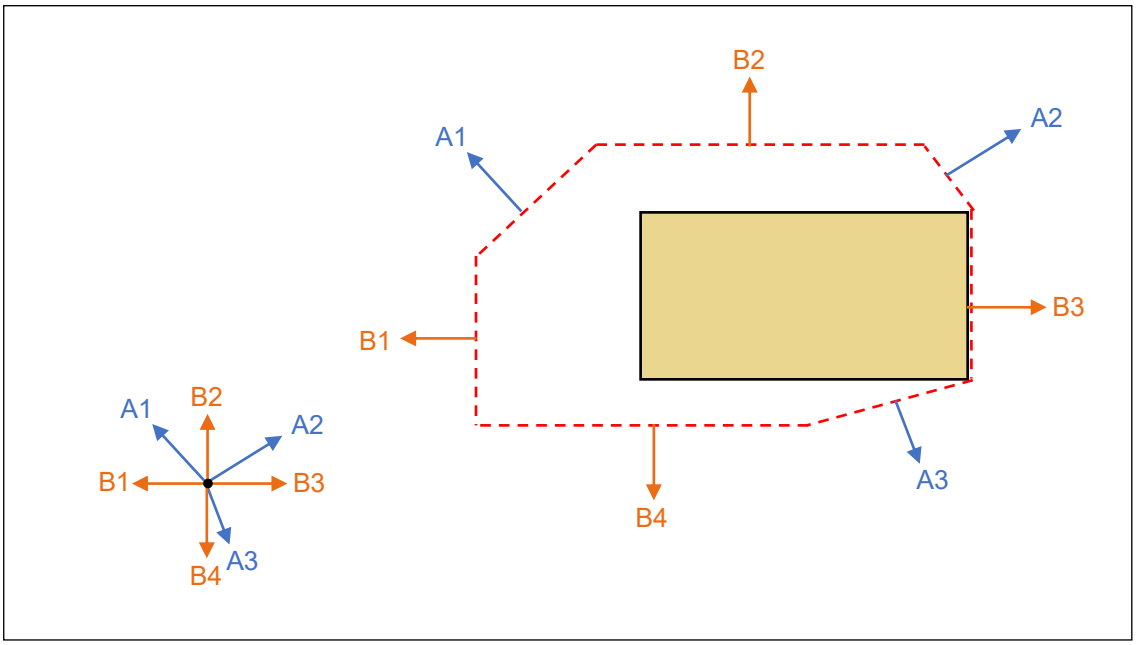


Ilustración 47. Demostración de expansión de un obstáculo según el método de Steven M. LaValle.

De esta manera y ejecutando la función *expandirPoligonal()*, que toma como argumentos el obstáculo a expandir y el robot poligonal, se obtiene el obstáculo expandido de la clase ObstaculoPoligonal, por lo que tendrá a su vez normales y vértices como parámetros.

Creación del escenario expandido

Al igual que los obstáculos, el escenario también se debe expandir, solo que se hace en la propia función de creación del escenario. Cuando se ejecuta la función *crearEscenario()* se tienen como argumentos la coordenada inferior derecha, la coordenada superior izquierda, y el robot.

En primer lugar, se toman las coordenadas extremas y se crea un rectángulo con sus valores, de forma que sea el escenario sin expandir.

A continuación, se debe medir el robot respecto del punto de referencia, de forma que se considere el margen en cada una de las dimensiones, positivas y negativas. Se representa para mayor comprensión.

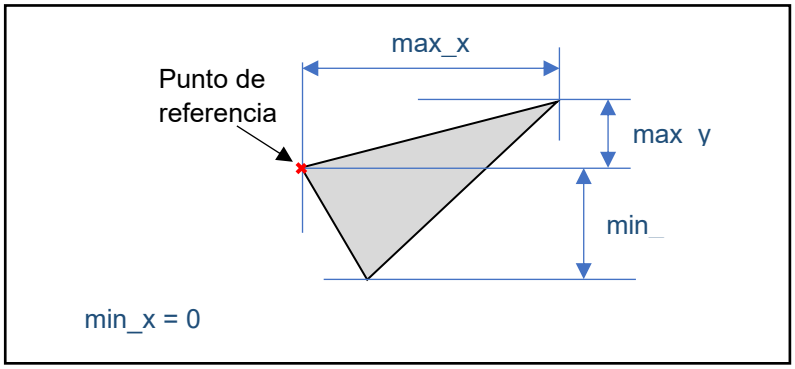


Ilustración 48. Demostración de medidas máximas y mínimas del robot.

Una vez obtenidos los valores máximos y mínimos en cada eje, se toma cada uno de los límites, superior, inferior, derecho e izquierdo; y se expande la figura hacia el interior las unidades medidas.

Se muestra un ejemplo del uso de esta función.

```
VerticesRobot = [0, 0; 0, -1; 2, 0];
Robot = RobotPoligonal(VerticesRobot);
Robot = rotar_vector(Robot,30);

[escenario, escenarioContraido] = crearEscenario([0,0],[10,10],Robot);
hold on;
dibujarPoligonal(escenario);
dibujarPoligonal(escenarioContraido,'r');
dibujarPoligonal(Robot,'b');
```

Código 13. Demostración de uso de la función crearEscenario().

El resultado de ejecutar el Código 13 es el siguiente:

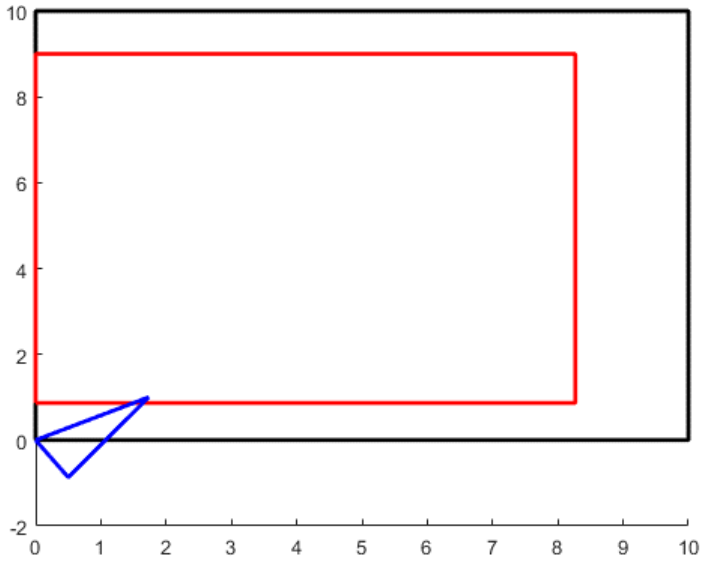


Ilustración 49. Representación de escenario poligonal expandido.

Creación de un entorno

Una vez el robot poligonal ha sido creado junto a los obstáculos y el entorno, y los obstáculos han sido expandidos, el entorno está completo. Para crearlo se debe llamar al constructor de la clase entorno.

```
classdef Entorno
    properties
        robot
        objetos
        objetosExpandidos
    end

    methods
        function obj = Entorno(robot, objetos, objetosExpandidos)
            obj.robot = robot;
            obj.objetos = objetos;
            obj.objetosExpandidos = objetosExpandidos;
        end
    end
end
```

Código 14. Clase Entorno.m para robots poligonales.

De esta forma se demuestra creando un entorno completo y representándolo.

```
VerticesRobot = [0, 0; 0, -1; 2, 0];
Robot = RobotPoligonal(VerticesRobot);
Robot = rotar_vector(Robot,30);

[escenario, escenarioContraido] = crearEscenario([0,0],[10,10],Robot);
hold on;
axis equal;
dibujarPoligonal(escenario);
dibujarPoligonal(escenarioContraido,'r');
dibujarPoligonal(Robot,'b');

VerticesObstaculo1 = [6, 6; 6, 7; 7, 7; 7, 6];
Obstaculo1 = ObstaculoPoligonal(VerticesObstaculo1);
dibujarPoligonal(Obstaculo1,'k');
obstaculoExpandido1 = expandirPoligonal(Robot,Obstaculo1);
dibujarPoligonal(obstaculoExpandido1,'r');

VerticesObstaculo2 = [3, 2; 3, 3; 5, 3; 5, 2];
Obstaculo2 = ObstaculoPoligonal(VerticesObstaculo2);
dibujarPoligonal(Obstaculo2,'k');
obstaculoExpandido2 = expandirPoligonal(Robot,Obstaculo2);
dibujarPoligonal(obstaculoExpandido2,'r');
entorno = Entorno(Robot,[escenario, Obstaculo1, Obstaculo2], [escenarioContraido,
obstaculoExpandido1, obstaculoExpandido2]);
```

Código 15. Creación de un entorno para robot poligonal.

Generación de trayectorias

Una vez ambos entornos han sido desarrollados, la forma más efectiva de ponerlos a prueba es mediante la aplicación de varios métodos de generación de trayectorias.

Para ello, se implementa el método del diagrama de visibilidad y el método de los mapas de caminos probabilísticos para la generación de un grafo, que será resuelto por el algoritmo de Dijkstra. También se implementa el algoritmo de los árboles de búsqueda rápidamente explorable.

Diagrama de visibilidad

En primer lugar, el mejor método para obtener la ruta más efectiva entre dos puntos en un entorno. Este es el método del diagrama de visibilidad, en el que se forma un grafo uniendo los vértices de los obstáculos y los puntos iniciales y finales con rectas, de forma que todos los puntos que tengan visibilidad estén conectados, de ahí su nombre.

Entorno para un robot cilíndrico

En primer lugar, se implementa para un robot cilíndrico, para ello se creará un entorno complejo en el que se pueda poner a prueba tanto la expansión como la complejidad del grafo y la generación de trayectorias.

El entorno en el que se comprobará el funcionamiento de los diferentes métodos de generación de trayectorias será el siguiente, donde las estrellas azules serán el punto inicial y el punto final.

De ahora en adelante cuando se mencione ‘objeto’ se hará referencia al objeto expandido, pues el objeto básico no es de interés a la hora de interpretar el robot como una unidad puntual.

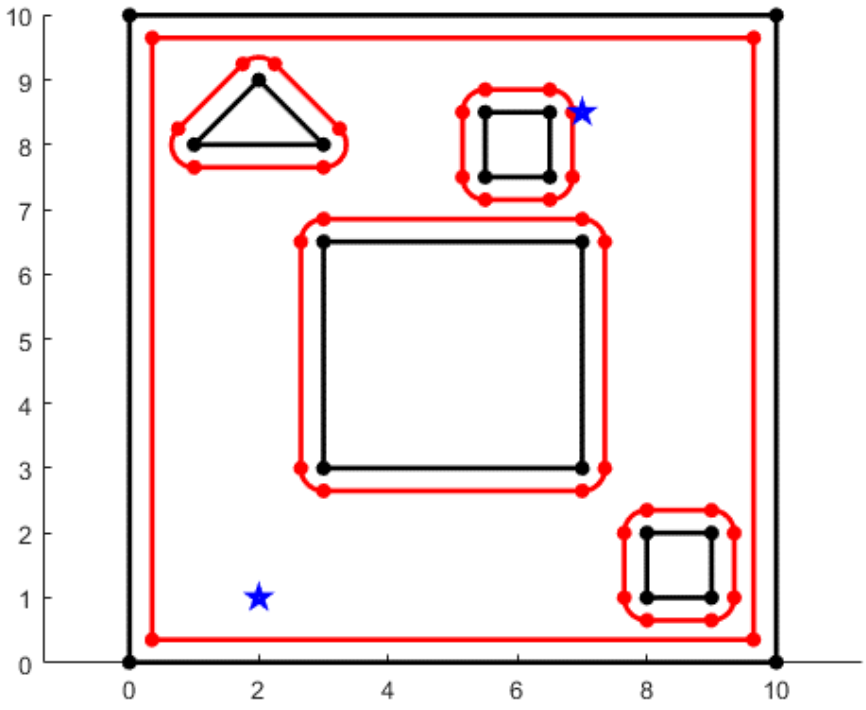


Ilustración 50. Entorno en el que se prueban los diferentes métodos de generación de trayectorias.

El entorno anterior viene definido por el siguiente código:

```
radio = 0.35;
puntoInicial = [2,1];
puntoFinal = [7,8.5];
hold on;

vertices = [0,0;10,0;10,10;0,10];
aristas = [Arista([1,2],1,pi/2), Arista([2,3],2,pi), Arista([3,4],3,3*pi/2),
Arista([4,1],4,0)];
escenario = Objeto(vertices, aristas, []);
clear vertices aristas
escenarioExpandido = contraerObjeto(escenario, radio);
escenarioLimpio = limpiarIntersecciones(escenarioExpandido);
clear escenarioExpandido

vertices = [3,3;7,3;7,6.5;3,6.5];
aristas = [Arista([1,2],1,3*pi/2), Arista([2,3],2,0), Arista([3,4],3,pi/2),
Arista([4,1],4,pi)];
obj1 = Objeto(vertices, aristas, []);
clear vertices aristas
objExpandido = expandirObjeto(obj1, radio);
objLimpio1 = limpiarIntersecciones(objExpandido);
clear objExpandido

vertices = [1,8;3,8;2,9];
aristas = [Arista([1,2],1,3*pi/2), Arista([2,3],2,pi/4), Arista([3,1],3,3*pi/4)];
obj2 = Objeto(vertices, aristas, []);
clear vertices aristas
objExpandido = expandirObjeto(obj2, radio);
objLimpio2 = limpiarIntersecciones(objExpandido);
clear objExpandido

vertices = [5.5,7.5;6.5,7.5;6.5,8.5;5.5,8.5];
aristas = [Arista([1,2],1,3*pi/2), Arista([2,3],2,0), Arista([3,4],3,pi/2),
Arista([4,1],4,pi)];
obj3 = Objeto(vertices, aristas, []);
clear vertices aristas
objExpandido = expandirObjeto(obj3, radio);
objLimpio3 = limpiarIntersecciones(objExpandido);
clear objExpandido

vertices = [8,1;9,1;9,2;8,2];
aristas = [Arista([1,2],1,3*pi/2), Arista([2,3],2,0), Arista([3,4],3,pi/2),
Arista([4,1],4,pi)];
obj4 = Objeto(vertices, aristas, []);
clear vertices aristas
objExpandido = expandirObjeto(obj4, radio);
objLimpio4 = limpiarIntersecciones(objExpandido);
clear objExpandido

objetos = [escenario,obj1,obj2,obj3,obj4];
objetosExpandidos = [escenarioLimpio,objLimpio1,objLimpio2,objLimpio3,objLimpio4];
entorno = Entorno(objetos,objetosExpandidos);
entorno = dibujarEntorno(entorno);
```

Código 16. Creación de escenario para generación de trayectorias.

Para obtener un grafo correspondiente al diagrama de visibilidad para un robot cilíndrico en primer lugar se debe generar una serie de matrices donde se almacene la posibilidad de comunicación directa entre vértices, o lo que es lo mismo, si hay visibilidad. Para obtener el grafo se llamará a la función *rectasVisibilidad()*.

Para ello, tomando los puntos inicial y final y el entorno se genera, en primer lugar, una matriz en la que se guardarán las distancias o costes entre dos vértices. Esta matriz tiene una codificación dependiente del entorno, pues cuantos más vértices u objetos existan, más valores deberá almacenar.

De esta forma se muestra un ejemplo de esta matriz en el momento de su inicialización, previo a haber calculado cualquier valor.

0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf
Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	8

Ilustración 51. Ejemplo de matrizDistancias inicializada.

Para entender el funcionamiento de esta matriz se ha de observar el valor encontrado en la casilla de la última fila y columna. Este valor es el número máximo de componentes de un objeto, en este caso ocho. Como se puede observar, la matriz está dividida en tres grupos de ocho valores, más la columna y fila finales cuyo único uso es aportar ese valor. Cada uno de los tres grupos define un objeto y sus vértices, de forma que la primera columna será el primer componente (vértice, arista o arco) del primer objeto o, por ejemplo, la décima columna será el segundo componente del segundo objeto, o la vigésima columna será el cuarto componente del tercer objeto.

Las columnas penúltima y antepenúltima corresponden al punto inicial y final respectivamente.

Esto será igual para las filas, de forma que cualquier posición de igual columna y fila será un valor que indique la distancia de un componente a sí mismo. Por esa misma razón se ha rellenado esta diagonal con valores cero.

De esta forma, por ejemplo, el valor 'matrizDistancias[13,6]' será el valor de la distancia entre el quinto componente del segundo objeto y el sexto componente del primer objeto.

También se formará una matriz que almacenará los componentes de cada objeto, la cual tendrá un número de columnas igual al valor máximo de componentes de cualquier objeto del entorno y un número de filas igual a los objetos del entorno, contando el escenario.

Una vez generadas las matrices necesarias para la interpretación del entorno, se tratará de reconocer las uniones directas entre componentes.

En primer lugar, se comprobará la posibilidad de unión entre el punto inicial y el punto final. Después se comprobará la posibilidad de unión entre el punto inicial y el resto de los componentes, y entre el punto final y el resto de las componentes.

Finalmente, se recorre cada componente y se comprueba su visibilidad con el resto de los componentes individualmente.

En caso de ser posible una unión en línea recta, se crea un objeto de clase Linea y se guarda la distancia entre estos puntos en la matrizDistancias.

```
classdef Linea
    properties
        curvo
        p1
        p2
        x
        y
        z
        radio
        anguloInicial
        anguloFinal
        valido
    end
```

Código 17. Clase Linea.m

Para agilizar la creación de una línea se ha implementado la función *crearCamino()*. Esta función variará sus parámetros dependiendo de el origen y destino de la línea, en el caso de que ambos sean puntos, en el caso de que sean componentes, o en el caso de que haya un punto y un componente.

En el caso de ser un componente, se tomará una línea tangente al mismo que permita la unión. Se exponen las tres posibilidades en la siguiente figura.

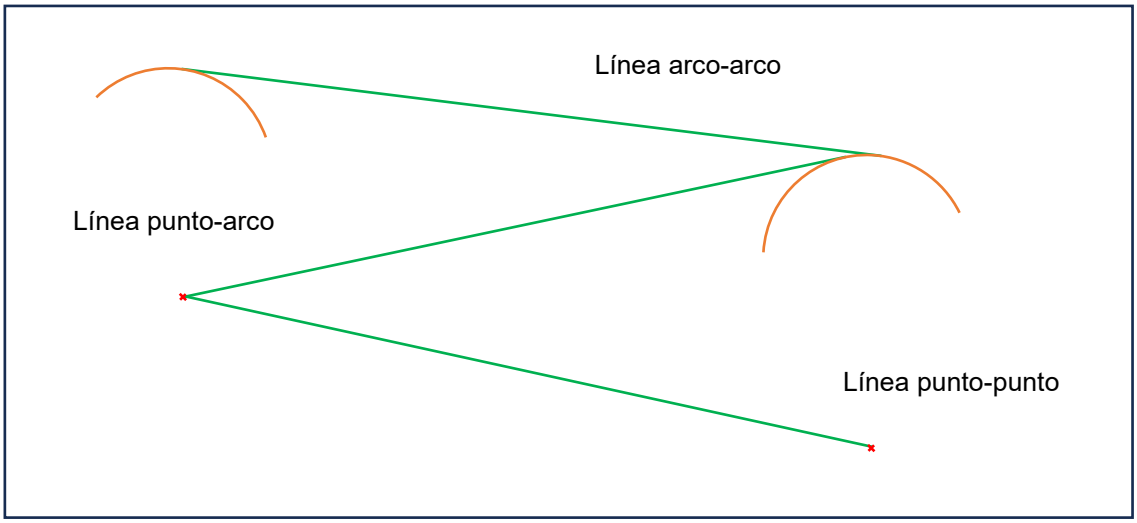


Ilustración 52. Posibilidades de creación de lineas con la función crearCaminos().

De esta forma se relacionan los extremos de las figuras creando posibles caminos incluyendo el punto inicial y el punto final, creando el grafo por el cual se generará el recorrido final. La representación de este será la siguiente:

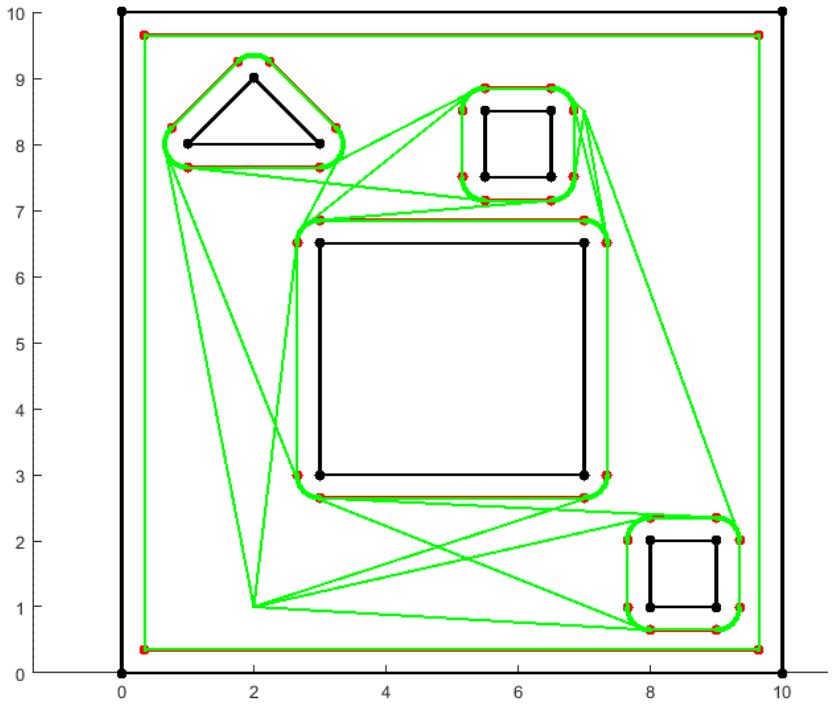


Ilustración 53. Representación del grafo creado por el método de visibilidad.

Aplicando el algoritmo de Dijkstra con la función *Dijkstra()*, que toma como argumentos la matriz de rectas entre componentes de los objetos del entorno, así como la matrizDistancias y el radio del objeto. Este último se utiliza exclusivamente para poder calcular la distancia recorrida en contacto con los arcos de las figuras, ya que, al sólo aportar las rectas entre componentes, no se tiene acceso a los mismos.

De esta forma, la función *Dijkstra()* implementa el algoritmo explicado en el Anexo A: Algoritmo de Dijkstra. El algoritmo se basa en, desde la perspectiva de un grafo, el análisis desde el punto inicial, del punto más cercano a éste. Se recorrerán todos los puntos no analizados, y se

observará si el recorrido a través del punto analizado tiene menor coste. En caso de ser así, se sustituye el recorrido actual desde el punto inicial hasta dicho punto.

Para el manejo de rutas y recorridos se crea una matriz unidimensional de celdas en la que se guarda la ruta actual más corta, o lo que es lo mismo, menos costosa, desde el punto inicial. Esta matriz se inicializa en todas las rutas como la unión directa entre el punto inicial y cada uno de los puntos, aunque sea imposible su acceso. En este caso guardará una distancia, o coste, infinitos.

A medida que se vayan comprobando los puntos, se irán sustituyendo las distancias de la matrizDistancias y actualizando los recorridos más cortos.

De esta forma, una vez el siguiente punto por explorar sea el punto final, se confirmará que ese recorrido es el más rápido, pues pasar por cualquier otro punto no estudiado implicaría un mayor coste.

Al aplicarlo al entorno, encontramos el problema de que los puntos realmente no son una coordenada puntual, sino que son arcos o vértices curvos, por lo que se debe tener en cuenta el coste de recorrer la distancia curva que une las dos rectas.

Para ello se calcula la diferencia en el ángulo que forman las rectas, y se calcula el perímetro recorrido teniendo en cuenta el radio del robot. Se muestra en el siguiente ejemplo.

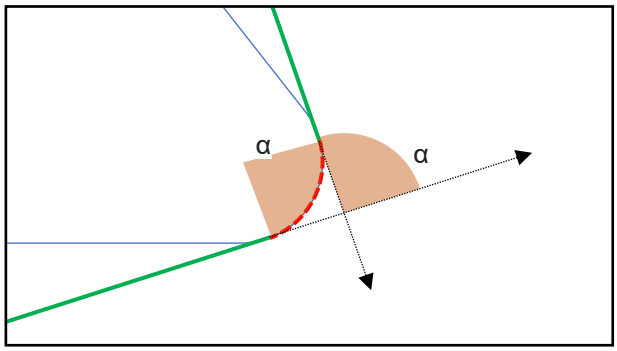


Ilustración 54. Demostración de cálculo de ángulo recorrido alrededor de vértice.

El valor calculado en radianes α se multiplicará por el valor del radio para obtener este tramo, y contabilizarlo a la hora de calcular el coste del recorrido.

Una vez obtenida la ruta final se puede representar en el mismo entorno.

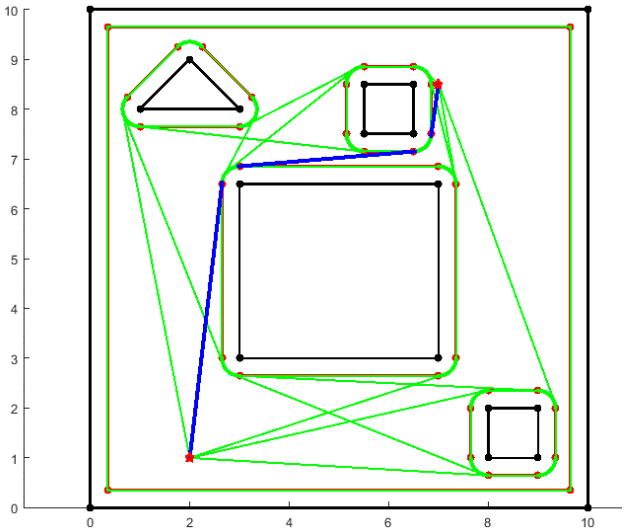


Ilustración 55. Recorrido final generado con el método del diagrama de visibilidad.

Entorno para un robot poligonal

Debido a la similitud entre el entorno para un robot poligonal y un entorno para un robot cilíndrico una vez creados y expandidos, se considera que la diferencia a la hora de aplicar el método del diagrama de visibilidad resulta mínima, por lo que únicamente se explican sus diferencias.

En primer lugar, al igual que en entorno para un robot cilíndrico, se plantea un entorno que desafíe la generación de trayectorias. Por ello se elige un entorno con dos obstáculos de diferentes tamaños.

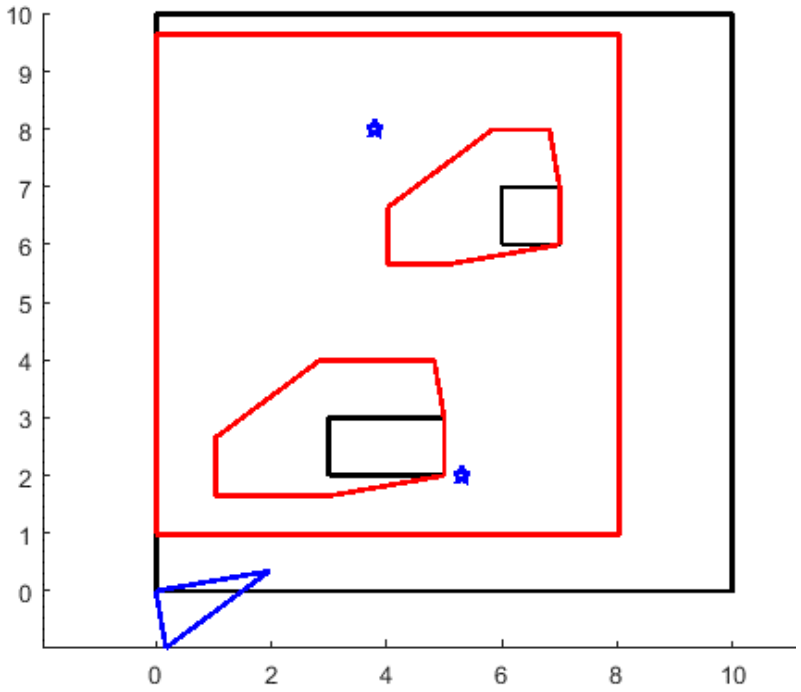


Ilustración 56. Entorno para un robot expandido triangular (azul) con dos obstáculos.

El anterior entorno se genera con el siguiente código:

```
VerticesRobot = [0, 0; 0, -1; 2, 0];
Robot = RobotPoligonal(VerticesRobot);
Robot = rotar_vector(Robot,10);

[escenario, escenarioContraido] = crearEscenario([0,0],[10,10],Robot);
hold on;
dibujarPoligonal(escenario);
dibujarPoligonal(escenarioContraido,'r');
dibujarPoligonal(Robot,'b');

VerticesObstaculo1 = [6, 6; 6, 7; 7, 7; 7, 6];
Obstaculo1 = ObstaculoPoligonal(VerticesObstaculo1);
dibujarPoligonal(Obstaculo1,'k');
obstaculoExpandido1 = expandirPoligonal(Robot,Obstaculo1);
dibujarPoligonal(obstaculoExpandido1,'r');

VerticesObstaculo2 = [3, 2; 3, 3; 5, 3; 5, 2];
Obstaculo2 = ObstaculoPoligonal(VerticesObstaculo2);
dibujarPoligonal(Obstaculo2,'k');
obstaculoExpandido2 = expandirPoligonal(Robot,Obstaculo2);
dibujarPoligonal(obstaculoExpandido2,'r');

entorno = Entorno(Robot,[escenario, Obstaculo1, Obstaculo2], [escenarioContraido,
obstaculoExpandido1, obstaculoExpandido2]);
axis equal

puntoInicial = [5.3,2];
puntoFinal = [3.8,8];
```

Código 18. Creación del entorno para la generación de trayectorias para robot poligonal.

De la misma forma que en la expansión para un robot cilíndrico, se crea una matrizDistancias, en la que se almacenarán las posibles distancias entre los puntos que formen el grafo, y se trata de unir todos los puntos del entorno de forma directa, incluyendo los puntos inicial y final. En el caso de ser posible la unión, se crea un objeto de la clase 'Linea'. Para este entorno, al no tener vértices formados por arcos, no se debe contemplar más que el caso de unión entre dos coordenadas puntuales.

De esta forma se obtiene la siguiente representación de las uniones, o diagrama de grafo:

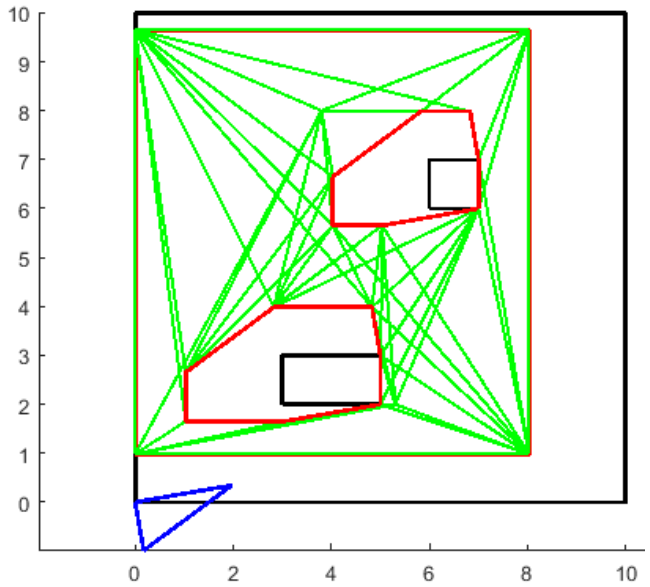


Ilustración 57. Representación del grafo creado por el método de visibilidad.

Una vez obtenido el grafo correspondiente al entorno, se ejecuta la función *Dijkstra()*, la cual al igual que en el entorno para un robot cilíndrico, recorre cada uno de los puntos comenzando por el más cercano al punto inicial, hasta llegar al punto final y obteniendo así su ruta más corta. De esta forma, se obtiene la ruta final.

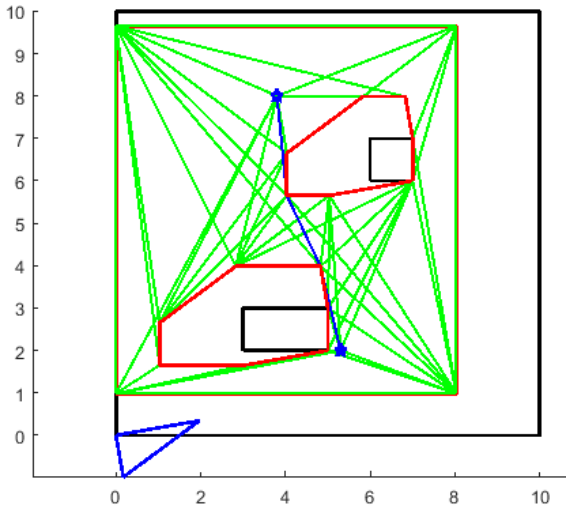


Ilustración 58. Recorrido final generado con el método del diagrama de visibilidad.

Método de los mapas de caminos probabilísticos

El siguiente método que se aplica para poner a prueba los entornos creados es el método de los mapas de caminos probabilísticos. Este método se creó teniendo en mente entornos complejos con objetos desconocidos. En este método se crean puntos aleatorios, que más tarde se unirán para generar posibles rutas desde el punto inicial y el punto final.

El resultado de este método es un grafo sobre el cual se podrá aplicar el algoritmo de Dijkstra para encontrar la ruta más rápida.

Entorno para un robot cilíndrico

Para la implementación del método de los mapas de caminos probabilísticos se va a utilizar de nuevo el entorno utilizado en el método del diagrama de visibilidad, con la finalidad de ponerlo en práctica. El código del entorno se encuentra en el Código 16. Creación de escenario para generación de trayectorias. Su representación es la siguiente:

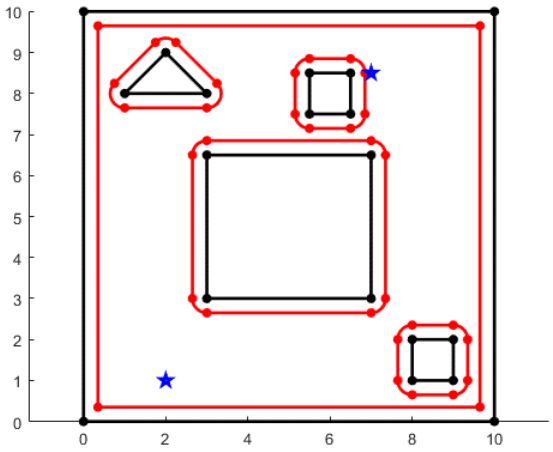


Ilustración 59. Entorno para la generación de trayectorias.

Para la aplicación de este método, el primer paso es la generación aleatoria de puntos en las dimensiones del escenario. Para ello, se utiliza la función *Generar_puntos_aleatorios()* a la que se le introducen el entorno, y un valor asignado que representa la cantidad de puntos por unidad al cuadrado. Este valor define lo poblado que estará el entorno, de forma que cuanta más densidad de puntos, más probable será conseguir un camino satisfactorio.

La función anterior lee del entorno la zona permitida por el escenario expandido, en la que se generan una cantidad de puntos igual a su área multiplicada por la densidad de puntos. Genera una matriz unidimensional del tamaño de la cantidad de puntos, y la rellena con objetos de tipo Nodo vacíos.

```
classdef Nodo
    properties
        coordenadas
        valido
    end

    methods
        function obj = Nodo(coordenadas)
            obj.coordenadas = coordenadas;
            obj.valido = 1;
        end
    end
end
```

Código 19. Clase Nodo.m

A continuación, se recorre la matriz de objetos de tipo Nodo y se rellena la propiedad 'coordenadas' con un valor aleatorio entre los valores máximo y mínimo de la coordenada vertical y horizontal.

Después de haber creado el punto aleatorio, se comprueba si se encuentra en un lugar válido. Para ello, se recorre cada uno de los objetos y se observa si existe algún componente respecto del cual se encuentre en el exterior. Si no existe ningún componente que declare que el objeto de tipo Nodo se encuentra en el exterior, se asignará la propiedad 'valido' como falso y después lo dibuja, en verde si el punto es válido, o en rojo, si el punto está en el interior.

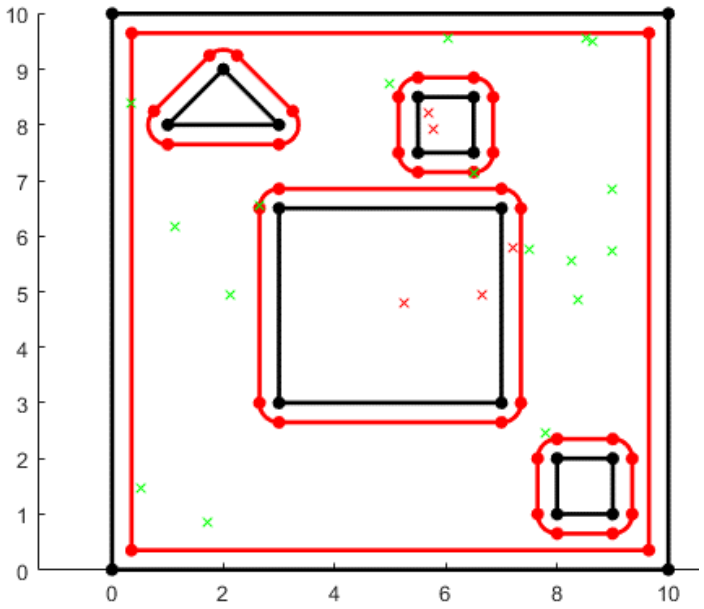


Ilustración 60. Entorno con Nodos generados y validados.

El siguiente paso será añadir los nodos inicial y final en caso de que sean válidos, y se tratará de unir todos los puntos posibles, de la misma manera que se une en el método del diagrama de visibilidad, formando un grafo con las rectas de tipo Linea generadas.

A su vez, se creará la matriz que incluirá todas las rectas y la matrizDistancias, que albergará los costes de todos los posibles caminos. Para más información ver Diagrama de visibilidad. Se muestra a continuación la representación de los posibles caminos.

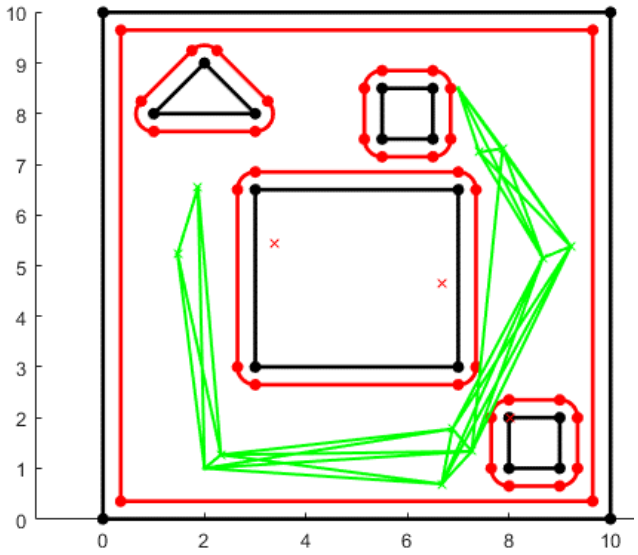


Ilustración 61. Grafo creado mediante el método de los mapas de caminos probabilísticos.

Una vez creado el grafo, se implementa el algoritmo de Dijkstra. Este algoritmo comienza en el punto inicial, y recorre uno por uno los puntos, desde el más cercano al más lejano, analizando la posibilidad de trazar una ruta hasta el resto de los puntos, en caso de que sea menos costosa y por tanto más corta. En este caso, la ruta actual se verá sustituida por la nueva ruta y su coste sobrescrito.

En el método de los mapas de caminos probabilísticos existe la posibilidad de no conseguir un camino posible, ya que pueden no existir puntos suficientes en el área libre que permitan la unión entre el punto inicial y el punto final. En caso de ocurrir esto, a la hora de implementar el algoritmo

de Dijkstra, se observará que no se detecta un punto siguiente más cercano, y que no se ha llegado al punto final.

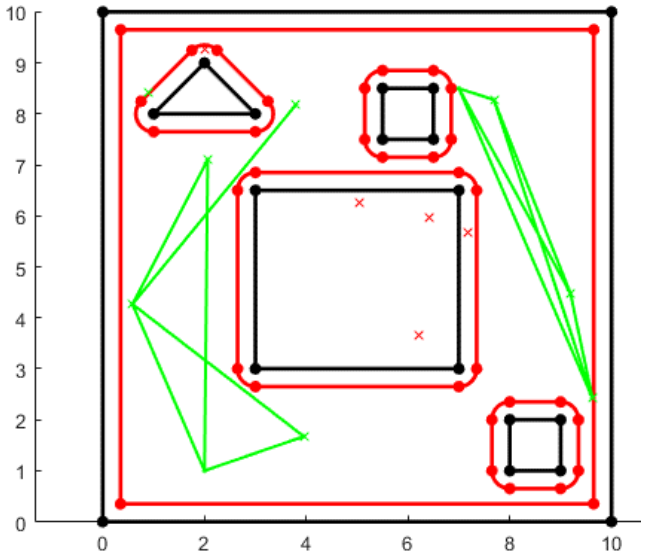


Ilustración 62. Grafo creado mediante el método de los mapas de caminos probabilísticos.

En una aplicación habitual, el programa debería terminar y anotar que no ha habido una solución satisfactoria, por lo que se debería reintentar la prueba para un nuevo resultado. En esta implementación, debido a que la idea principal es la prueba del entorno, en caso de no encontrar un camino válido, se creará una cantidad de puntos adicionales igual al 20% de la densidad, y se incrementará el valor de densidad un 20%. De esta forma, cada vez que el entorno no es capaz de crear una unión válida, aumenta la cantidad de puntos que añade al entorno.

De esta forma se aumenta el grafo pudiendo completar el recorrido entre el punto inicial y el punto final, se muestra la continuación a la ilustración anterior.

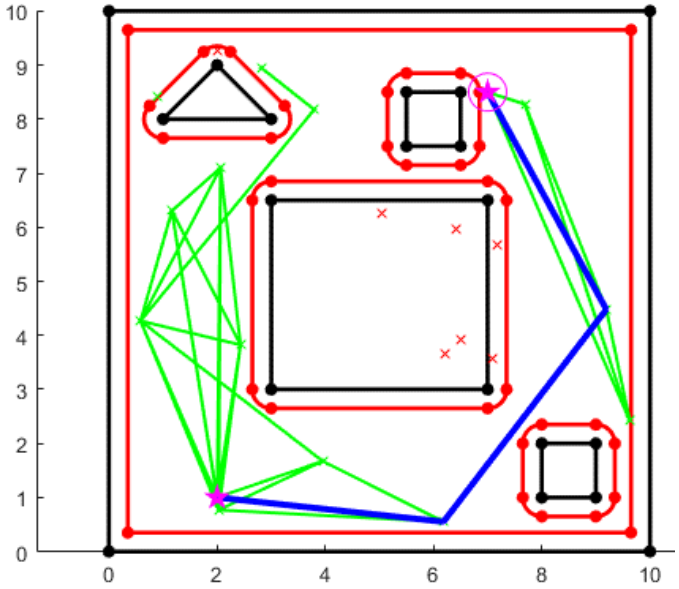


Ilustración 63. Recorrido completado tras el aumento de puntos en el entorno.

Después de cada expansión del entorno se trata de conseguir un camino válido entre el punto inicial y el punto final. En caso de ser posible, se genera la ruta más rápida y se representa la figura final.

Entorno para un robot poligonal

Al igual que en el caso del diagrama de visibilidad, a la hora de implementar el método de los mapas de caminos probabilísticos, tampoco se halla una gran diferencia entre el entorno para un robot cilíndrico y el entorno para un robot poligonal una vez han sido creados y expandidos. Por ello se comentará este caso basado en el caso del entorno para un robot cilíndrico.

En primer lugar, de nuevo se aplicará el mismo entorno que para la generación de trayectorias mediante el método del diagrama de visibilidad. El código para crear este entorno es el Código 18. Creación del entorno para la generación de trayectorias para robot poligonal. Este crea la siguiente figura:

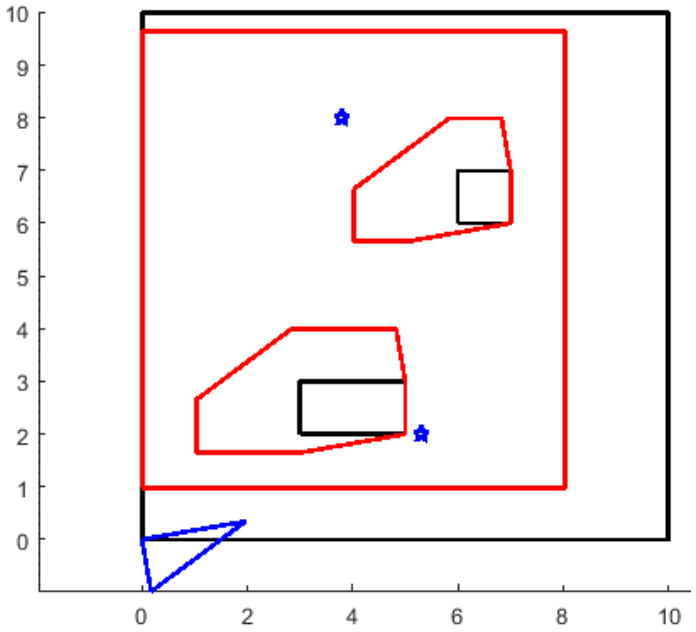


Ilustración 64. Entorno para la generación de trayectorias en un entorno para un robot poligonal.

A continuación, se crea una matriz unidimensional de objetos de tipo Nodo en la que se almacenan los puntos generados aleatoriamente. Estos se evalúan y, en caso de ser válidos, se dibujan en verde. En caso de no ser válidos se dibujan en rojo.

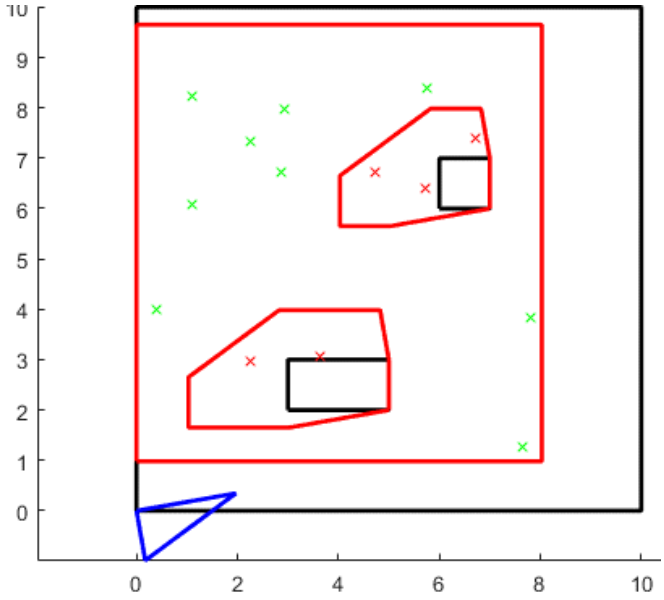


Ilustración 65. Entorno con nodos generados y validados.

A continuación, se recorren todos los puntos y se unen los que sea posible unir, de la misma forma que se hace en el entorno para un robot cilíndrico. En los que sea posible unir se crea un objeto de tipo Linea. Se genera la matrizDistancias y la matriz que almacena las rectas de tipo Linea.

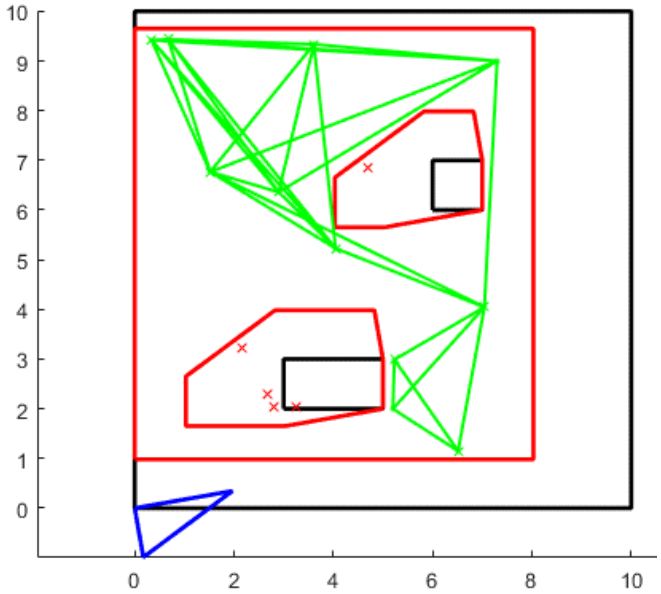


Ilustración 66. Grafo creado mediante el método de los mapas de caminos probabilísticos.

Una vez creado el grafo, se implementa el algoritmo de Dijkstra. Este algoritmo comienza en el punto inicial, y recorre uno por uno los puntos, desde el más cercano al más lejano, analizando la posibilidad de trazar una ruta hasta el resto de los puntos, en caso de que sea menos costosa y por tanto más corta. En este caso, la ruta actual se verá sustituida por la nueva ruta y su coste sobrescrito. En el caso de no encontrar una ruta posible se aumentará el número de puntos hasta que se encuentre una ruta posible.

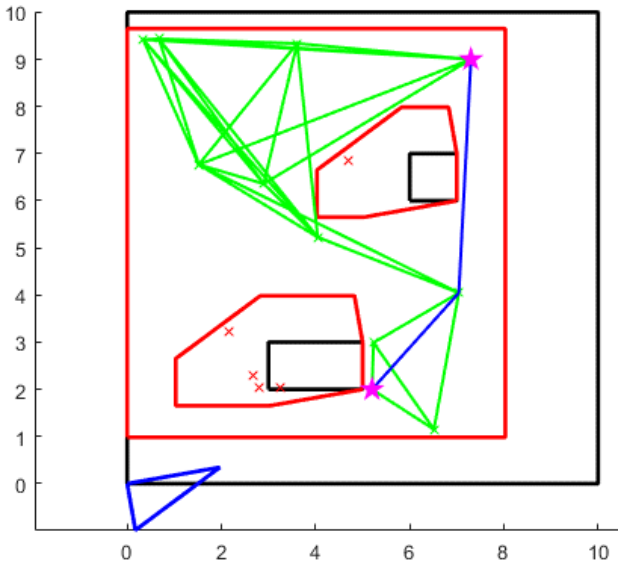


Ilustración 67. Recorrido final tras aplicar el algoritmo de Dijkstra.

Método de los árboles de búsqueda rápidamente explorables

Finalmente, para poner a prueba el entorno, se aplica el método de los árboles de búsqueda rápidamente explorables, o RRT por sus siglas en inglés. En este método se identifica el punto inicial como el punto inicial del árbol RRT. A continuación, se generará un punto aleatorio en el espacio, y se busca el punto existente más cercano. Desde este se tendrá que expandir una rama nueva en la dirección del nuevo punto. Este proceso se repite hasta que se alcanza una distancia mínima al punto final y se une.

Entorno para un robot cilíndrico

En primer lugar, como hasta ahora, se comenzará por el caso para un robot cilíndrico. En este caso se aplicará el mismo entorno que en el método del diagrama de visibilidad y el método de los mapas de caminos probabilísticos.

Este entorno se crea con el Código 15. Creación de un entorno para robot poligonal. Se representa de la siguiente forma.

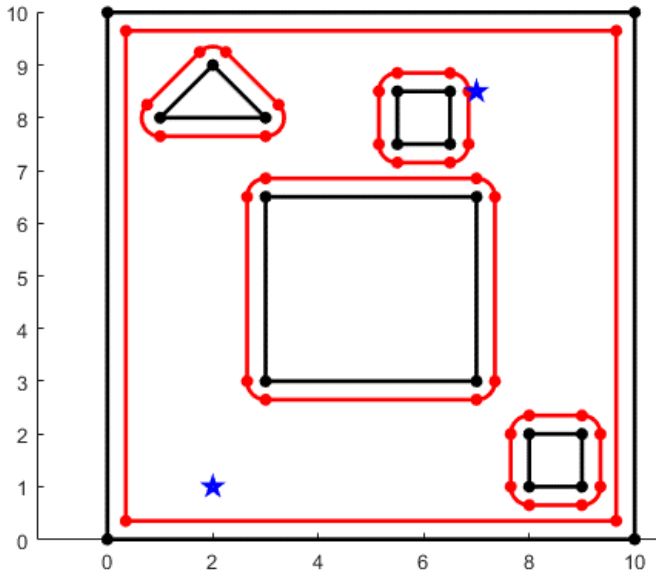


Ilustración 68. Entorno para la generación de trayectorias.

El código que se utiliza de este punto en adelante es una modificación del código aportado por el Dr. Ranko Zotovic Stanisic.

En el código se crea un nodo inicial en el punto inicial, a partir del cual se genera el árbol, y se representa como un ‘o’ el punto final. Después se inicializa una lista de puntos en la que se guardarán las coordenadas del árbol. Se inicializa el valor máximo de iteraciones y el valor de longitud de las ramas.

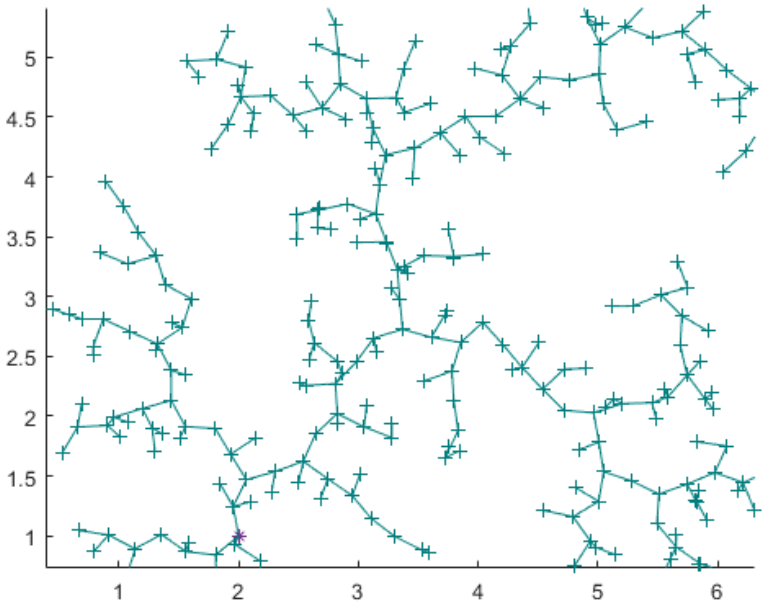


Ilustración 69. Creación de un árbol de puntos aleatorios desde un punto inicial.

A continuación, se delimitan los valores extremos del escenario, dentro del cual se generarán los puntos posibles puntos aleatorios, y se extraen los vértices del entorno para, posteriormente, poder calcular las posibles colisiones de las futuras ramas.

Una vez las variables necesarias han sido inicializadas, comienza un bucle mientras no haya terminado el proceso o se hayan alcanzado las iteraciones máximas. En este bucle se genera un objeto tipo Nodo nuevo, y se comprueba que no haya ninguna colisión. En caso de no ser posible su creación, se repetirá el proceso y se creará uno nuevo.

De esta forma, la creación de un punto a lo largo del escenario incita al árbol a expandirse en esa dirección desde su punto más próximo. Tras la creación de varios puntos, un árbol se puede ver de la siguiente forma.

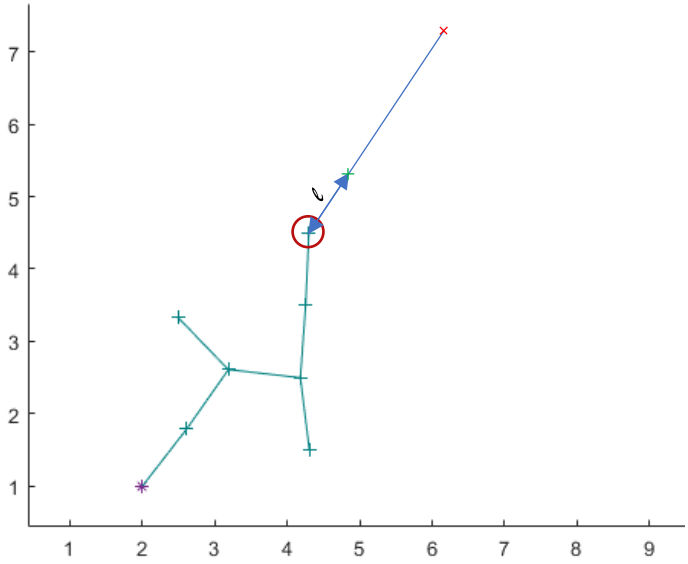


Ilustración 70. Demostración de creación de una nueva rama.

Para la expansión del árbol se genera un punto aleatoriamente en el espacio libre del escenario y, después de comprobar que es un punto válido, se busca el punto perteneciente al árbol más cercano. Para ello, se ordenan todos los puntos con un valor de menor a mayor en distancia, y se toma el primer valor.

En la anterior ilustración se puede encontrar el punto generado aleatoriamente, representado en color rojo, unido por una recta con su punto más cercano, rodeado en granate.

Una vez obtenido el vector que une los dos puntos, se extiende una distancia "ℓ" la rama desde el punto más cercano ya perteneciente al árbol. De esta forma, se obtiene un vector del tamaño de la rama y, en caso de no haber colisiones con objetos, se ubica el punto definitivo, representado en la figura en verde, y se crea una nueva rama entre el nuevo punto y el punto más cercano.

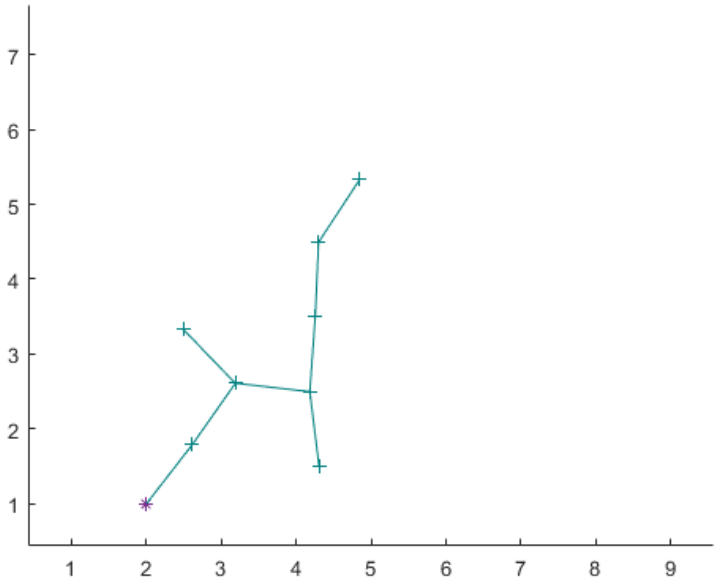


Ilustración 71. Nuevo punto añadido al árbol.

En caso de haber colisiones con algún objeto, se volvería a crear un punto aleatorio comenzando el proceso de nuevo.

Si el punto generado aleatoriamente está más cerca de un punto del árbol que la distancia "ℓ", el vector no se expandirá y se situará el punto definitivo en el mismo punto generado, siempre y cuando no haya colisiones.

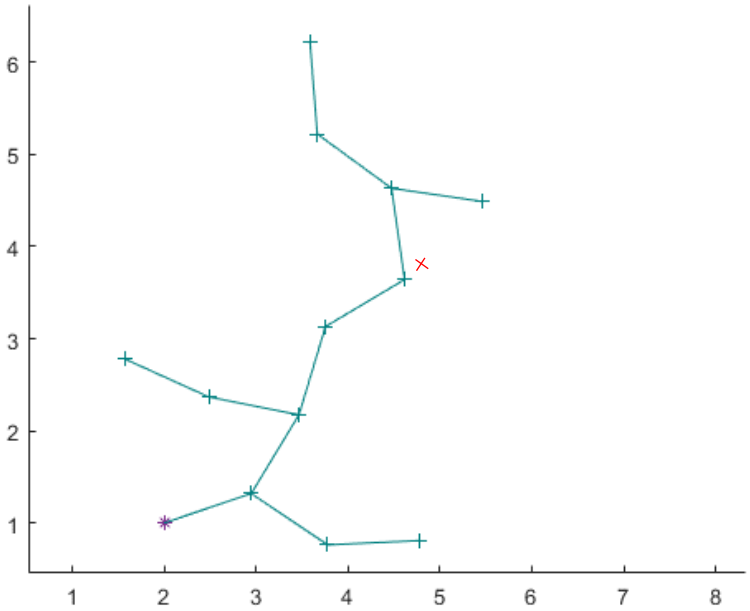


Ilustración 72. Punto generado de forma aleatoria.

Como se observa en la figura anterior, el punto se puede quedar más cerca de cualquier punto perteneciente al árbol, sin importar que sea un punto extremo, creando una nueva rama.

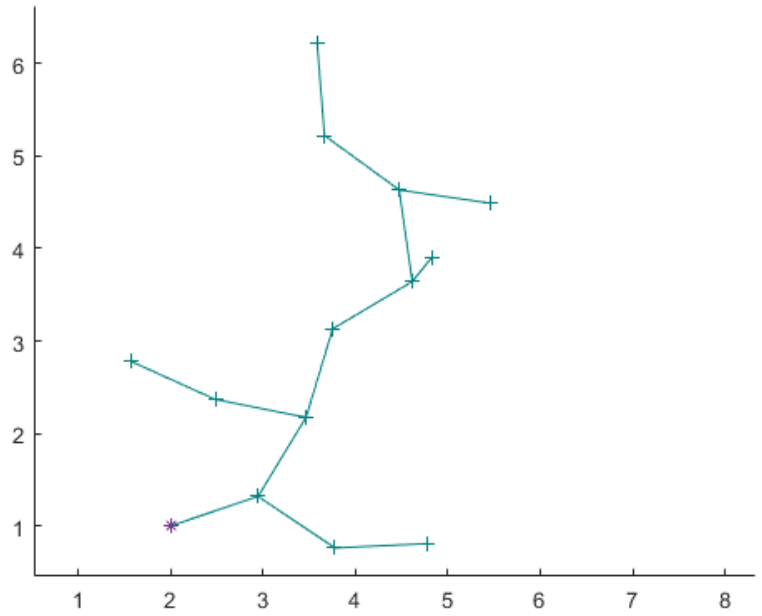


Ilustración 73. Punto generado a menos de la distancia "l" de su punto más cercano.

Se observa la incorporación de la rama hasta el punto (4.8, 3.9) a pesar de encontrarse a menos de la distancia asignada a la longitud de la rama en la figura anterior.

Con cada expansión del árbol, se comprueba la proximidad al punto final del nuevo punto generado. En caso de encontrarse a menos de una distancia asignada, que generalmente será la misma que la longitud de la rama, se unirán estos puntos con una nueva rama, terminando el árbol.

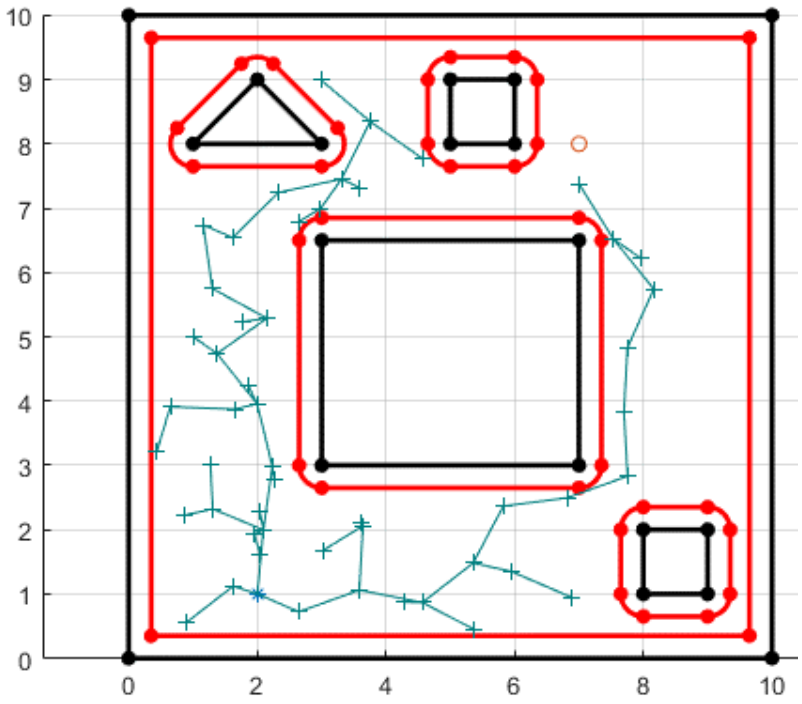


Ilustración 74. Ejemplo de rama próxima a punto final.

Una vez terminado el árbol, se realza la figura en verde para que el camino más rápido quede visible.

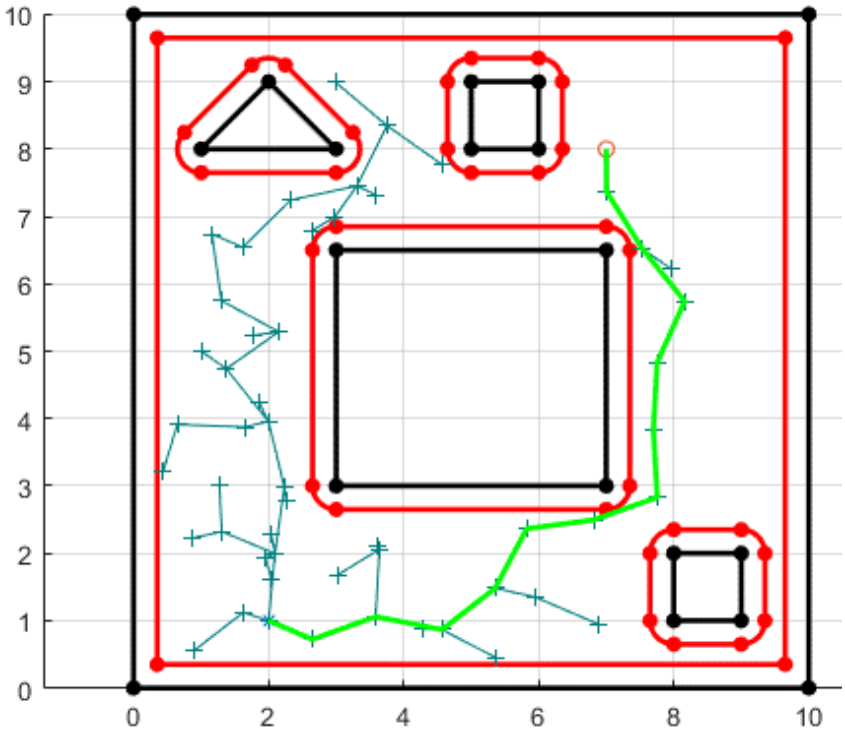


Ilustración 75. Camino final remarcado en color verde.

Como se ha mencionado anteriormente, la longitud de las ramas viene definida en el inicio del programa. Con esto se puede considerar una longitud más grande de rama, que dará lugar a menor precisión a la hora de recorrer huecos pequeños, pero una velocidad de computación mucho mayor. Se muestra un ejemplo para una longitud de rama más alta.

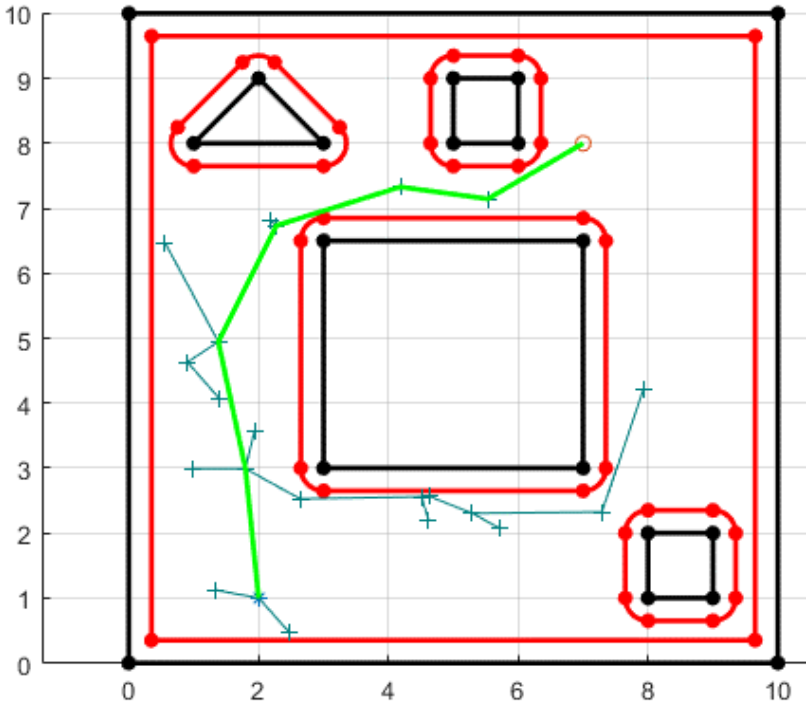


Ilustración 76. Ejemplo con longitud de rama de 2 unidades.

La anterior ilustración es la representación de un árbol generado con longitud de rama de 2 unidades, siendo la velocidad de computación de 1.45 segundos, una longitud de 11.06 unidades y un total de 22 puntos, sin incluir el punto final. A continuación, se muestra el ejemplo contrario.

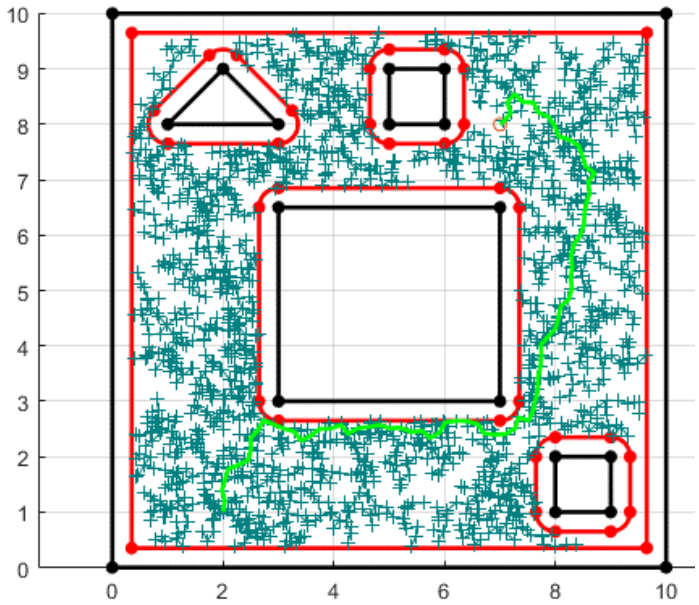


Ilustración 77. Ejemplo con longitud de rama de 0.2 unidades.

En este ejemplo se muestra un árbol con una longitud de rama de 0.2 unidades. Como se puede observar da lugar a un gran número de puntos que, en su mayor parte no son útiles, y que dan lugar a un coste mucho mayor, de 15.28 y un tiempo de computación de 13.30 segundos, con un total de 1651 puntos, sin contar el punto final.

A diferencia del ejemplo de la Ilustración 71, en el ejemplo anterior se ha conseguido acceder a puntos como detrás del triángulo superior o encima del cuadrado superior, lugares no accesibles con facilidad para longitudes de rama superiores.

Ante todo, se puede aplicar un valor adicional, el sesgo. El sesgo permite forzar al algoritmo de creación de puntos a aproximarse hacia el punto final, sesgando, como su nombre indica, la creación de las ramas hacia el punto final. Se ejecuta el ejemplo anterior con un sesgo de 0.1

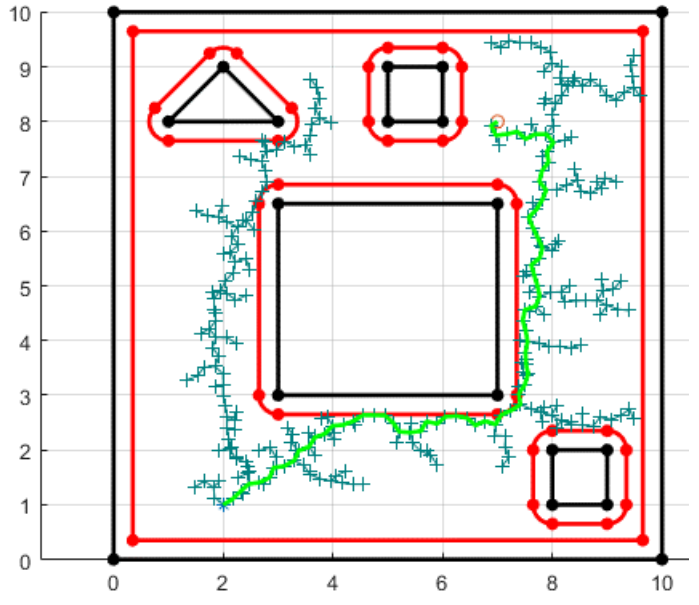


Ilustración 78. Ejemplo con longitud de rama de 0.2 unidades y un sesgo de 0.1

Como se puede observar, la aplicación de un sesgo consigue un resultado en solo 3.71 segundos y un coste de 13.54, contando con únicamente 301 puntos, contando con el punto final.

Entorno para un robot poligonal

Al igual que en los casos anteriores, se utilizará el mismo entorno, el cual se puede generar con el Código 18. Creación del entorno para la generación de trayectorias para robot poligonal. Este entorno se representa de la siguiente forma.

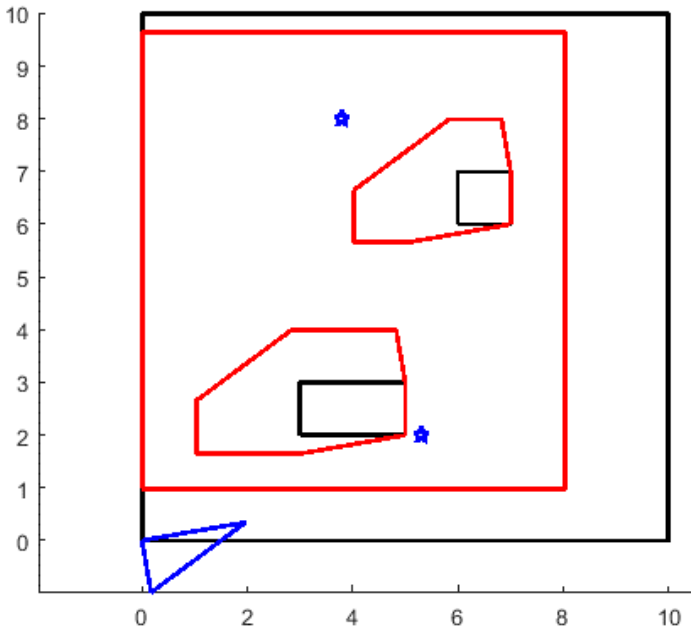


Ilustración 79. Entorno para la generación de trayectorias en un entorno para un robot poligonal.

De la misma manera que se ejecuta el programa para el caso de un entorno para un robot cilíndrico, en este caso, se genera el comienzo de un árbol en el punto inicial, se delimitan los límites del escenario, y se asignan la distancia de la rama y la distancia mínima al punto final.

Así pues, de la misma forma, se obtiene un punto aleatorio dentro del espacio de configuración, es decir, el espacio libre. Una vez obtenido el punto, se calcula el punto más cercano al punto generado, y se une con una línea. Después se utiliza esta línea para crear una rama desde el punto más cercano, con la longitud establecida.

En caso de no ser posible debido a posibles intersecciones con objetos ya existentes, el punto se eliminará y se generará un nuevo punto, hasta que no haya colisiones con los objetos del entorno.

De la misma forma, una vez alcanzada una distancia mínima, se extiende una rama desde el último punto hasta el punto final. Adicionalmente, se repasa el recorrido desde el punto inicial hasta el punto final.

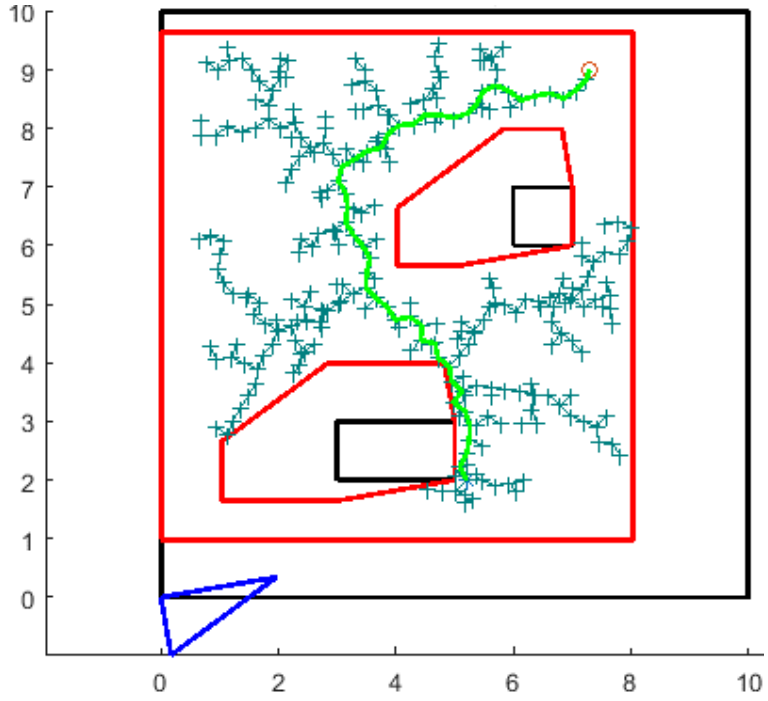


Ilustración 80. Árbol generado en entorno para robot poligonal.

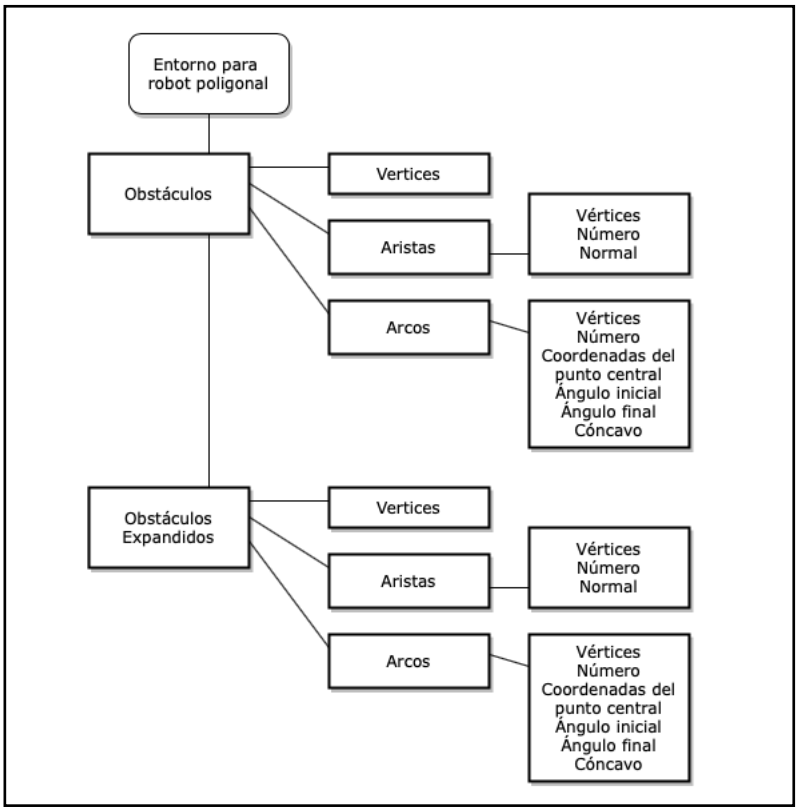
Conclusión

Se concluye que la implementación final de ambos entornos cumple con el objetivo principal del proyecto, poder representar de forma fiel un entorno, y poder generar trayectorias a su alrededor, tanto para robots cilíndricos como poligonales.

Para el alcance de este proyecto se parte del estudio de antecedentes, como son la implementación básica ya existente, y la aplicación de los distintos métodos de generación de trayectorias.

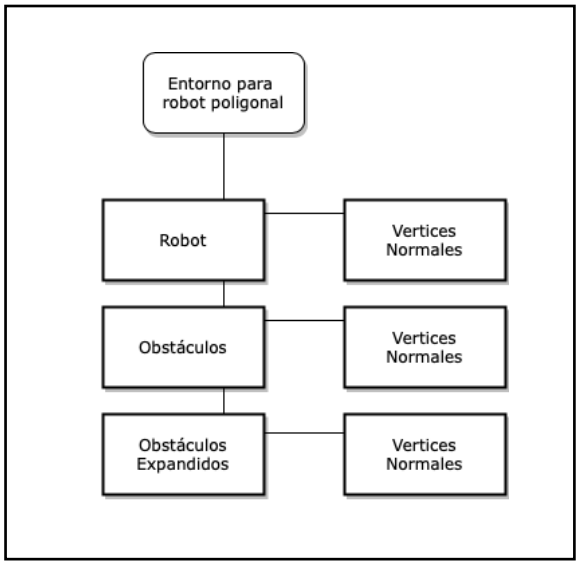
Otro de los puntos principales del proyecto es la realización de este mediante programación orientada a objetos. Con este principio se debe desarrollar una serie de objetos que son incluidos en las propiedades de otros objetos.

De esta forma en el entorno para un robot cilíndrico se tiene el siguiente árbol de objetos:



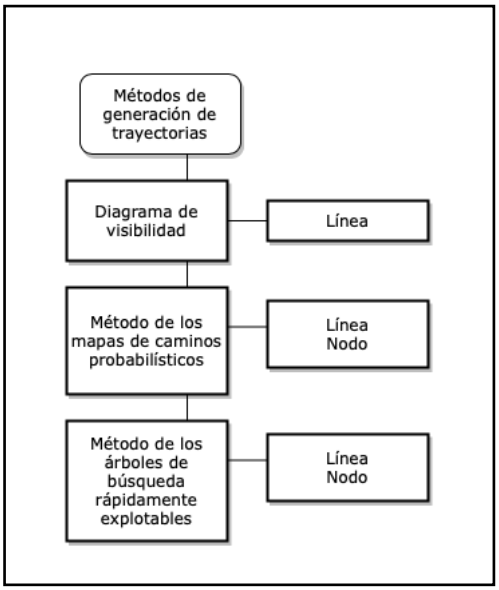
De esta manera, con tan solo un objeto siendo el entorno, al introducirlo en una función, podemos obtener la totalidad del entorno, permitiendo acceso a cada una de las propiedades de todos los objetos que pertenecen a este.

De la misma forma, en un entorno para un robot poligonal, se encuentra el siguiente árbol de objetos:



Al igual que en el caso anterior, en un robot poligonal se encuentran los obstáculos y los obstáculos expandidos, aunque carece de objetos bajo un segundo nivel y, incluye también el propio robot.

Por último, los entornos se ponen a prueba mediante la aplicación de generadores de trayectorias, demostrando su utilización y posibilidades de ajuste de los distintos métodos. En este caso, se encuentran objetos utilizados exclusivamente en ciertos métodos de generación de trayectorias, como son las líneas y los nodos.



Debido a la necesidad de mantener una gran cantidad de puntos y su validación, se aplica la clase Nodo a los puntos que implican la generación aleatoria de puntos, en lugar de almacenar sus propiedades individualmente.

Gracias a todo este proceso el proyecto finaliza con la implementación de dos entornos para la planificación de movimientos de robots en Matlab, pudiendo modificar la estructura y disposición de los diferentes obstáculos y la forma y tamaño del robot, además de poder elegir el método de generación de trayectorias, utilizando también métodos de generación de puntos aleatorios.

Índice de Código

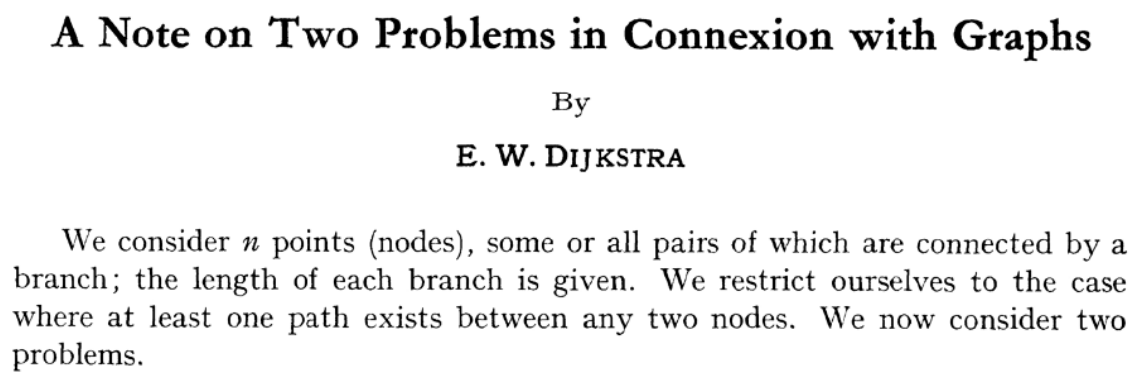
Índice de Ilustraciones

Anexos

Anexo A: Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo clásico de grafos, desarrollado por Edsger W. Dijkstra en 1956, cuyo objetivo principal es encontrar el camino más corto desde un nodo de origen a todos los demás nodos en un grafo ponderado y conexo, donde todos los nodos están conectados entre sí mediante aristas y cada arista tiene un peso o coste asociado.

Este método es actualmente de los métodos más eficaces, si no el que más, para la resolución de caminos en grafos, siempre y cuando no existan caminos con costes negativos.



A Note on Two Problems in Connexion with Graphs

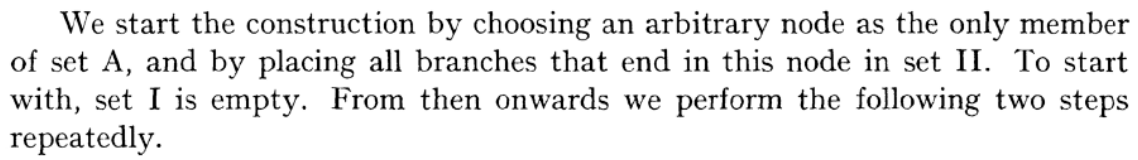
By

E. W. DIJKSTRA

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Ilustración 81. Planteamiento de Dijkstra en su 'A Note on Two Problems in Connexion with Graphs'.

La idea fundamental en la que se basa el algoritmo de Dijkstra es mantener un conjunto de nodos abiertos y cerrados, donde los nodos que ya han sido cerrados tendrán sus distancias desde el nodo de origen determinadas de manera definitiva, y donde los nodos abiertos permanecen con distancias temporales estimadas que pueden actualizarse a medida que el algoritmo avanza.



We start the construction by choosing an arbitrary node as the only member of set A, and by placing all branches that end in this node in set II. To start with, set I is empty. From then onwards we perform the following two steps repeatedly.

Ilustración 82. Planteamiento de Dijkstra en su 'A Note on Two Problems in Connexion with Graphs'.

En primer lugar, se crea un conjunto para contabilizar los nodos abiertos, y se inicializa una matriz para almacenar todas las distancias desde el nodo de origen al resto de nodos, excepto en el caso de un nodo consigo mismo. En este caso es la llamada matrizDistancias. Dentro de esta matriz se contabilizan el punto inicial y el punto final. También se crea una matriz donde se almacenan todas las posibles rutas, que inicialmente consistirán en el punto inicial y el propio punto al que corresponda la celda.

Step 1. The shortest branch of set II is removed from this set and added to set I. As a result one node is transferred from set B to set A.

Step 2. Consider the branches leading from the node, that has just been transferred to set A, to the nodes that are still in set B. If the branch under consideration is longer than the corresponding branch in set II, it is rejected; if it is shorter, it replaces the corresponding branch in set II, and the latter is rejected.

Ilustración 83. Bucle según Dijkstra en su 'A Note on Two Problems in Connexion with Graphs'.

A continuación, se selecciona el nodo más cercano, y se examinan todos los puntos aún no seleccionados, o abiertos, revisando si la distancia entre el punto inicial y el nodo revisado es más corta pasando por el nodo seleccionado. En caso de ser así, comparándose con el valor de matrizDistancias, se actualiza con la distancia y la ruta nuevas.

Una vez examinados todos los puntos el punto seleccionado se cierra y se elimina de los puntos abiertos, después se selecciona el siguiente punto más cercano. Revisando de nuevo todos los puntos abiertos para averiguar posibles rutas más rápidas a través del punto seleccionado, se repite el proceso de nuevo.

Cuando el punto seleccionado, es decir el siguiente punto más cercano, sea el punto final, se habrá demostrado que no hay una ruta más corta desde el punto inicial hasta el punto final, pues elegir cualquiera de los puntos restantes como parte del recorrido, incrementaría el coste más del valor actual hasta el punto final.

En el algoritmo original, se plantea la posibilidad de tener tan solo un punto inicial, y la idea es calcular las distancias hasta todos los puntos constituyentes del grafo. En el caso de este proyecto no será necesario, pues tan solo interesa el valor hasta el punto final.

Cabe destacar que el algoritmo de Dijkstra tiene una complejidad temporal donde el número de nodos se encuentra de forma cuadrática. Esto implica que aumentar el número de nodos por un factor X provocará un factor de incremento de tiempo de X^2.

We then return to step 1 and repeat the process until sets II and B are empty. The branches in set I form the tree required.

Ilustración 84. Final del bucle según Dijkstra en su 'A Note on Two Problems in Connexion with Graphs'.

Anexo B: Código Matlab

A continuación, se expone el código Matlab utilizado para la realización del proyecto. Las siguientes funciones pueden no ser exactas o haber sufrido modificaciones a medida que ha avanzado el proyecto.

Función dibujarObjeto():

```
function obj = dibujarObjeto(obj, color)
hold on;

% Dibuja los vertices
if nargin < 2
    color = 'k';
end
scatter(obj.vertices(:,1), obj.vertices(:,2), 'filled', 'MarkerFaceColor', color);

% Dibujar aristas
for i = 1:length(obj.aristas)
    a = obj.aristas(i);
    v1 = obj.vertices(a.vertices(1),:);
    v2 = obj.vertices(a.vertices(2),:);
    %Corregir la normal
    numerador = obj.vertices(a.vertices(1),1) - obj.vertices(a.vertices(2),1);
    denominador = obj.vertices(a.vertices(1),2) - obj.vertices(a.vertices(2),2);
    normal = atan2(-numerador,denominador);
% Se calcula la normal
    if(normal < 0)
        normal = normal + 2*pi;
% Se hace positiva
    end
    if (normal - a.normal < pi/2 && normal - a.normal > -pi/2 )
        obj.aristas(i).normal = normal;
% Si cuadra se asigna a la normal
    else
        obj.aristas(i).normal = normal + pi;
% Si no cuadra se asigna la contraria
    end
    if (obj.aristas(i).normal < 0)
        obj.aristas(i).normal = obj.aristas(i).normal + 2*pi;
    elseif (obj.aristas(i).normal >= 2*pi)
        obj.aristas(i).normal = obj.aristas(i).normal - 2*pi;
    end

    plot([v1(1), v2(1)], [v1(2), v2(2)], color, 'LineWidth', 2);
end

% Dibujar arcos
for i = 1:length(obj.arcos)
    a = obj.arcos(i);
    Vertice1 = obj.vertices(a.vertices(1),:);
    Vertice2 = obj.vertices(a.vertices(2),:);
    if (Vertice1(1)~=Vertice2(1) || Vertice1(2)~=Vertice2(2))
        t = linspace(a.AngInicial, a.AngFinal);
        x = a.xc + a.radio*cos(t);
        y = a.yc + a.radio*sin(t);
```

```
        plot(x, y, color, 'LineWidth', 2);
    end
end

axis equal
hold on;

end
```

Función expandirObjeto():

```
function objExpandido = expandirObjeto(obj, distancia)

    hold on;
    VerticesExpandidos = zeros(length(obj.vertices),2,2);   % Variable que almacena
los vertices expandidos (dos para cada vertice original)
    VerticesNuevos = zeros(length(obj.vertices),2);         % Variable que almacena
los vertices para crear el objeto expandido
    VectoresExpandidos = zeros(length(obj.vertices),2);     % Variable que almacena
los vectores direccion final de cada Arista o Arco
    AristasExpandidas = [];                                 % Lista de Aristas
nuevas
    ArcosExpandidos = [];                                   % Lista de Arcos nuevos
    % Expandir aristas
    for i = 1:length(obj.aristas)

        a = obj.aristas(i);
        Vo = obj.vertices(a.vertices(1),:);             % Se obtiene el vector
inicial
        Vf = obj.vertices(a.vertices(2),:);             % Se obtiene el vector
final
        normal = a.normal;
        Xnormal = cos(normal);                          % Se divide la normal en
coordenada unitaria X
        Ynormal = sin(normal);                          % y en coordenada unitaria
Y
        Vx = [Vo(1), Vf(1)] + Xnormal*distancia;        % Se calcula las
coordenadas X expandidas
        Vy = [Vo(2), Vf(2)] + Ynormal*distancia;        % y las coordenadas Y
expandidas

        VerticesExpandidos(a.vertices(1),1,2) =  Vx(1);     % Se guarda la
coordenada Xi
        VerticesExpandidos(a.vertices(1),2,2) =  Vy(1);     % y la coordenada Yi
        VerticesExpandidos(a.vertices(2),1,1) =  Vx(2);     % Se guarda la
coordenada Xf
        VerticesExpandidos(a.vertices(2),2,1) =  Vy(2);     % y la coordenada Yf

        VectorX = Vf(1) - Vo(1);                        % Se calcula el vector
dirección de las aristas
        VectorY = Vf(2) - Vo(2);                        % para el eje X e Y
        n = norm([VectorX, VectorY]);                   % Se calcula el modulo
        VectoresExpandidos(a.vertices(2),1) = VectorX/n;    % Se calcula el vector
unitario
        VectoresExpandidos(a.vertices(2),2) = VectorY/n;
```

```
        AristaNueva = Arista([a.vertices(1)*2,a.vertices(2)*2-1], a.numero*2,
a.normal);      % Se crea una nueva arista como la expandida
        AristasExpandidas = [AristasExpandidas, AristaNueva];               % Se
guarda la arista expandida recien creada
    end

% Dibujar arcos
    for i = 1:length(obj.arcos)
        a = obj.arcos(i);
        if a.concavo == 1
            radio = a.radio - distancia;            % En caso de ser un arco
concavo se resta la distancia del radio
        else
            radio = a.radio + distancia;            % En caso de ser un arco
convexo se suma la distancia del radio
        end
        t = linspace(a.AngInicial, a.AngFinal);     % Se dibuja el arco desde el
angulo inicial y final del arco
        x = a.xc + radio*cos(t);
        y = a.yc + radio*sin(t);

        VerticesExpandidos(a.vertices(1),1,2) =  x(1);          % Se guarda la
coordenada Xi
        VerticesExpandidos(a.vertices(1),2,2) =  y(1);          % y la coordenada
Yi
        VerticesExpandidos(a.vertices(2),1,1) =  x(end);        % Se guarda la
coordenada Xf
        VerticesExpandidos(a.vertices(2),2,1) =  y(end);        % y la coordenada
Yf

        VectorX = x(end) - x(end - 1);                      % Se calcula el vector
de direccion final
        VectorY = y(end) - y(end - 1);                      % de los ejes X e Y
        n = norm([VectorX, VectorY]);                       % Se calcula el modulo
        VectoresExpandidos(a.vertices(2),1) = VectorX/n;    % Se calcula el vector
unitario
        VectoresExpandidos(a.vertices(2),2) = VectorY/n;

        ArcoNuevo = Arco([a.vertices(1)*2,a.vertices(2)*2-1], a.numero*2, a.xc,
a.yc, a.zc, a.AngInicial, a.AngFinal, radio, a.concavo);
        ArcosExpandidos = [ArcosExpandidos, ArcoNuevo];

    end

    % Expandir vertices
    for i = 1:length(obj.vertices)
        v = obj.vertices(i,:);
        anguloInicial = atan2(VerticesExpandidos(i,2,1)-
v(2),VerticesExpandidos(i,1,1)-v(1));   % Se calcula el angulo inicial basado en
los vertices nuevos (expandidos) respecto del centro de las aristas
        anguloFinal = atan2(VerticesExpandidos(i,2,2)-
v(2),VerticesExpandidos(i,1,2)-v(1));

        VerticesNuevos(i*2-1, 1) = VerticesExpandidos(i,1,1);       % Se almacenan
los nuevos vertices Xi
        VerticesNuevos(i*2-1, 2) = VerticesExpandidos(i,2,1);       % Yi
```

```
        VerticesNuevos(i*2, 1) = VerticesExpandidos(i,1,2);          % Xf
        VerticesNuevos(i*2, 2) = VerticesExpandidos(i,2,2);          % Yf para el
objeto expandido
        if (anguloInicial < 0)
            anguloInicial = anguloInicial + 2*pi;   % Se corrige en caso de ser
negativos
        end
        if (anguloFinal < 0)
            anguloFinal = anguloFinal + 2*pi;       % Se corrige en caso de ser
negativos
        end
        t = linspace(anguloInicial,anguloFinal);    % Se calcula el circulo
alrededor del
        x = v(1) + distancia*cos(t);                % vector basado en los angulos
        y = v(2) + distancia*sin(t);
        VectorX = x(2) - x(1);                      % Se calcula el vector
direccion del comienzo
        VectorY = y(2) - y(1);                      % del arco del vertice
        n = norm([VectorX, VectorY]);
        dir = [VectorX/n, VectorY/n];
        if(dir(1)-VectoresExpandidos(i, 1) > 0.5 || dir(1)-VectoresExpandidos(i, 1)
< -0.5)     % En caso de no ser una
            anguloInicial = anguloInicial - 2*pi;
% direccion similar la del
            t = linspace(anguloInicial,anguloFinal);
% comienzo del arco del vertice
            x = v(1) + distancia*cos(t);
% a la del final del arco/arista
            y = v(2) + distancia*sin(t);
% se le da la vuelta al arco
        end
% invirtiendo el anguloInicial y repitiendo el arco
        ArcoNuevo = Arco([i*2-1,i*2], i*2-1, v(1), v(2), 0, anguloInicial,
anguloFinal, distancia, 0);
        ArcosExpandidos = [ArcosExpandidos, ArcoNuevo];
    end
    objExpandido = Objeto(VerticesNuevos, AristasExpandidas, ArcosExpandidos);
end
```

Función interseccion2d():

```
% Devuelve el punto de intersección entre dos rectas dadas por un par de puntos
cada una.
% Si son paralelas devuelve Inf.
function [xf, yf] = interseccion2d(p1,p2,q1,q2)
% Coordenadas de la primera recta:
x11=p1(1);
x21=p2(1);
y11=p1(2);
y21=p2(2);
% Coordenadas de la segunda recta:
x12=q1(1);
x22=q2(1);
y12=q1(2);
y22=q2(2);
```

```
% Si la primera es vertical:
if x11==x21
   % Si la segunda tambien es vertical:
   if x12==x22
      punto=[Inf,Inf];
   else
      punto=[x11,y12+(y22-y12)*(x11-x12)/(x22-x12)];
   end
% Si la primera es horizontal:
elseif y11==y21
   % Si la segunda tambien es horizontal:
   if y12==y22
      punto=[Inf,Inf];
   else
      punto=[x12+(x22-x12)*(y11-y12)/(y22-y12),y11];
   end
else
   % Si la primera es inclinada:
   % Si la segunda es vertical:
   if x12==x22
      punto=[x12,y11+(y21-y11)*(x12-x11)/(x21-x11)];
   % Si la segunda es horizontal:
   elseif y12==y22
            punto=[x11+(x21-x11)*(y22-y11)/(y21-y11),y22];
      % Si ambas son oblicuas:
      else
    % Coeficientes de la primera recta:
      D0=det([x11 1;x21 1]);
      D1=det([y11 1;y21 1]);
      D2=det([x11 y11;x21 y21]);
      a1=D1/D0;
    b1=D2/D0;
      % Coeficientes de la segunda recta:
      D0=det([x12 1;x22 1]);
      D1=det([y12 1;y22 1]);
      D2=det([x12 y12;x22 y22]);
      a2=D1/D0;
      b2=D2/D0;
      % Encontrar las coordenadas del punto de intersección:
            if a1==a2
      % (Caso cuando son paralelas)
            punto=[Inf Inf];
            else
                  D0=det([1 -a1;1 -a2]);
                  D1=det([b1 -a1;b2 -a2]);
                  D2=det([1 b1;1 b2]);
                  x=D2/D0;
                  y=D1/D0;
            punto=[x,y];
      end
   end
end
   c1 = [(x11+x21)/2,(y11+y21)/2];
   c2 = [(x12+x22)/2,(y12+y22)/2];
   d1 = sqrt((c1(1) - x11)^2 + (c1(2) - y11)^2);
   dp1 = sqrt((c1(1) - punto(1))^2 + (c1(2) - punto(2))^2);
   d2 = sqrt((c2(1) - x12)^2 + (c2(2) - y12)^2);
```

```
   dp2 = sqrt((c2(1) - punto(1))^2 + (c2(2) - punto(2))^2);

   if(dp1 > d1 || dp2 > d2)
       punto=[Inf Inf];
   end
    xf = punto(1);
    yf = punto(2);
end
```

Función interseccion_recta_recta():

```
function obj=interseccion_recta_recta(obj, i, j)

a1 = obj.aristas(i);
a2 = obj.aristas(j);
% Verificar si las aristas se intersectan
vi1 = obj.vertices(a1.vertices(1),:);
vf1 = obj.vertices(a1.vertices(2),:);
vi2 = obj.vertices(a2.vertices(1),:);
vf2 = obj.vertices(a2.vertices(2),:);
[x, y] = interseccion2d(vi1, vf1, vi2, vf2);

% Si las aristas se intersectan, corregir los vértices
if ~isinf(x) && ~isinf(y)
    if verticeCorrecto(a1, obj, vi2)
        obj.vertices(a2.vertices(1),:) = [x y];
    else
        obj.vertices(a2.vertices(2),:) = [x y];
    end

    if verticeCorrecto(a2, obj, vi1)
        obj.vertices(a1.vertices(1),:) = [x y];
    else
        obj.vertices(a1.vertices(2),:) = [x y];
    end
end
end
```

Función interseccion_arco_arco():

```
function obj = interseccion_arco_arco(objeto, i, j)
% Funcion para calcular los angulos de interseccion entre dos circulos
% Entradas:
% - objeto: Objeto inicial.
% - i: índice del primer arco.
% - j: índice del segundo arco.
% Salidas:
% - obj: Objeto inicial con los arcos recortados
% Informacion adicional:
% - d: distancia entre los centros de los dos circulos.
% - Si los circulos no se intersectan, la funcion devuelve [0,0].
% - Los angulos se miden en radianes.
arco1 = objeto.arcos(i);
arco2 = objeto.arcos(j);
```

```
V = [arco2.xc-arco1.xc,arco2.yc-arco1.yc];

r1 = arco1.radio;
r2 = arco2.radio;

if (r1==r2 && V(1)==0 && V(2)==0)
    obj=Objeto(Inf,[],[]);
    return;
end

d = sqrt(V(1)^2+V(2)^2);

Angulo_relativo = atan2(V(2),V(1));

if (d<abs(r1-r2) || d>r1+r2 || abs(d-(r1+r2))<=0.0000000000000001) % Se hace esto
porque a pesar de que los valores de "d" deberían ser exactos, y iguales a r1+r2,
devuelve falso a la igualdad
    obj=Objeto(Inf,[],[]);
    return;
end

alpha = (2 * acos((r1^2 + d^2 - r2^2) / (2 * r1 * d)));
arco1_angulo1 = Angulo_relativo + alpha/2;
arco1_angulo2 = Angulo_relativo - alpha/2;
beta = (2 * acos((r2^2 + d^2 - r1^2) / (2 * r2 * d)));
arco2_angulo1 = Angulo_relativo + pi + beta/2;
arco2_angulo2 = Angulo_relativo + pi - beta/2;

if(arco2_angulo1>2*pi || arco2_angulo1<0)
    arco2_angulo1 = mod(arco2_angulo1,2*pi);
end

% Crear un vector con los ángulos
angulos = [arco1_angulo1, arco1_angulo2, arco2_angulo1, arco2_angulo2];
angulos = mod(angulos,2*pi);
angulos2 = flip(angulos);
condition = zeros(4,1);
ArcosNuevos = objeto.arcos;
VerticesNuevos = objeto.vertices;
% Representar los arcos
arco = arco1;
otroarco = arco2;
%r = r1;
for n=1:length(angulos)
    angulo = angulos(n);
    angulo2 = angulos2(n);
    if n==3
        arco = arco2;
        otroarco = arco1;
        %r = r2;
    end

    %plot([arco.xc, arco.xc + r*cos(angulo)], [arco.yc, arco.yc + r*sin(angulo)]);

    if (angulo > min(arco.AngInicial, arco.AngFinal) && angulo <
max(arco.AngInicial, arco.AngFinal)) && (angulo2 > min(otroarco.AngInicial,
otroarco.AngFinal) && angulo2 < max(otroarco.AngInicial, otroarco.AngFinal))
```

```
        %plot([arco.xc, arco.xc + r*cos(angulo)], [arco.yc, arco.yc +
r*sin(angulo)]);
        condition(n)=1;
    end
end

q=0;
arcoActual = arco1;
r=[1,2];
for p=[i,j]
    if(condition(q+1)==1 && condition(q+2)==1)
        ArcosNuevos(p).AngFinal = angulos(q+r(2));
        x = ArcosNuevos(p).xc + ArcosNuevos(p).radio*cos(angulos(q+r(2)));
        y = ArcosNuevos(p).yc + ArcosNuevos(p).radio*sin(angulos(q+r(2)));
        VerticesNuevos = [VerticesNuevos;x,y];
        ArcosNuevos(p).vertices(2) = length(VerticesNuevos);
        ArcosNuevos(length(ArcosNuevos)+1) = arcoActual;
        ArcosNuevos(length(ArcosNuevos)).AngInicial = angulos(q+r(1));
        x = ArcosNuevos(p).xc + ArcosNuevos(p).radio*cos(angulos(q+r(1)));
        y = ArcosNuevos(p).yc + ArcosNuevos(p).radio*sin(angulos(q+r(1)));
        VerticesNuevos = [VerticesNuevos;x,y];
        ArcosNuevos(length(ArcosNuevos)).vertices(1) = length(VerticesNuevos);
    end

    if(xor(condition(1)==1,condition(2)==1))
        if(ArcosNuevos(p).AngInicial<angulos(q+r(2)) &&
ArcosNuevos(p).AngInicial>angulos(q+r(1)))
            if(condition(q+r(1)))
                ArcosNuevos(p).AngInicial = angulos(q+r(1));
                x = ArcosNuevos(p).xc + ArcosNuevos(p).radio*cos(angulos(q+r(1)));
                y = ArcosNuevos(p).yc + ArcosNuevos(p).radio*sin(angulos(q+r(1)));
                VerticesNuevos = [VerticesNuevos;x,y];
                ArcosNuevos(p).vertices(1) = length(VerticesNuevos);
            else
                ArcosNuevos(p).AngInicial = angulos(q+r(2));
                x = ArcosNuevos(p).xc + ArcosNuevos(p).radio*cos(angulos(q+r(2)));
                y = ArcosNuevos(p).yc + ArcosNuevos(p).radio*sin(angulos(q+r(2)));
                VerticesNuevos = [VerticesNuevos;x,y];
                ArcosNuevos(p).vertices(1) = length(VerticesNuevos);
            end
        else
            if(condition(q+1))
                ArcosNuevos(p).AngFinal = angulos(q+r(1));
                x = ArcosNuevos(p).xc + ArcosNuevos(p).radio*cos(angulos(q+r(1)));
                y = ArcosNuevos(p).yc + ArcosNuevos(p).radio*sin(angulos(q+r(1)));
                VerticesNuevos = [VerticesNuevos;x,y];
                ArcosNuevos(p).vertices(2) = length(VerticesNuevos);
            else
                ArcosNuevos(p).AngFinal = angulos(q+r(2));
                x = ArcosNuevos(p).xc + ArcosNuevos(p).radio*cos(angulos(q+r(2)));
                y = ArcosNuevos(p).yc + ArcosNuevos(p).radio*sin(angulos(q+r(2)));
                VerticesNuevos = [VerticesNuevos;x,y];
                ArcosNuevos(p).vertices(2) = length(VerticesNuevos);
            end
        end
    end
    q=2;
```

```
    arcoActual=arco2;
end

obj=Objeto(VerticesNuevos,objeto.aristas,ArcosNuevos);

% if ~(min(i,j)==1)
%     for k=1:min(i,j)-1
%         ArcosNuevos(k) = objeto.arcos(k);
%     end
% end
% if ~(min(i,j)+1>max(i,j)-1)
%     for k=min(i,j)+1:max(i,j)-1
%         ArcosNuevos(k) = objeto.arcos(k);
%     end
% end
% if ~(max(i,j)+1>length(objeto.arcos))
%     for k=max(i,j)+1:length(objeto.arcos)
%         ArcosNuevos(k) = objeto.arcos(k);
%     end
% end

end
```

Función interseccion_arco_recta():

```
function obj = interseccion_arco_recta(objeto, i, j)
% Calcula el punto de intersección de la recta.
% objeto: item a analizar
% i: Número de arco
% j: Número de arista


xr1 = objeto.vertices(objeto.aristas(j).vertices(1),1);
yr1 = objeto.vertices(objeto.aristas(j).vertices(1),2);
xr2 = objeto.vertices(objeto.aristas(j).vertices(2),1);
yr2 = objeto.vertices(objeto.aristas(j).vertices(2),2);
xc = objeto.arcos(i).xc;
yc = objeto.arcos(i).yc;
radio = objeto.arcos(i).radio;

% En caso de que la linea sea vertical
if xr2 == xr1
    % Comprueba si la linea está fuera del circulo
    if abs(xr1 - xc) > radio
        obj=Objeto(Inf,[],[]);
        return
        % Calcula las coordenadas "y" de los puntos de intersección
    else
        y_offset = sqrt(radio^2 - (xr1 - xc)^2);
        y1 = yc + y_offset;
        y2 = yc - y_offset;
        % Asigna la coordenada "x"
        x1 = xr1;
        x2 = xr1;
    end
```

```
else
    % Obtenermos la fórmula de la recta
    m = (yr2 - yr1) / (xr2 - xr1);
    b = yr1 - m * xr1;

    % Calcula los coefficientes de la ecuacion cuadratica de la
    % intersección
    A = 1 + m^2;
    B = 2 * (m * b - m * yc - xc);
    C = xc^2 + (b - yc)^2 - radio^2;

    % Calculate the discriminant of the quadratic equation
    discriminante = B^2 - 4 * A * C;

    % Comprueba el caso de que no tenga solución (no hay intersección)
    if discriminante < 0
        obj=Objeto(Inf,[],[]);
        return
    end

    % Calculate the two possible solutions
    x1 = (-B + sqrt(discriminante)) / (2 * A);
    y1 = m * x1 + b;
    x2 = (-B - sqrt(discriminante)) / (2 * A);
    y2 = m * x2 + b;
end

%Comprueba si no es el final de la recta
if (x1==xr1 && y1==yr1) || (x1==xr2 && y1==yr2) || (x2==xr1 && y2==yr1) || (x2==xr2
&& y2==yr2)
    obj=Objeto(Inf,[],[]);
    return
end

if (abs(x1-x2)<10^-6 && abs(y1-y2)<10^-6)
    obj=Objeto(Inf,[],[]);
    return
end

ang1 = atan2(y1-yc,x1-xc);
if ang1 < 0
    ang1 = ang1 + 2*pi;
end
ang2 = atan2(y2-yc,x2-xc);
if ang2 < 0
    ang2 = ang2 + 2*pi;
end

dist_recta = sqrt((xr1-xr2)^2+(yr1-yr2)^2)/2;
xcr = (xr1+xr2)/2;
ycr = (yr1+yr2)/2;

if((sqrt((xcr-x1)^2+(ycr-y1)^2)/2)<=dist_recta && ang1>=objeto.arcos(i).AngInicial
&& ang1<=objeto.arcos(i).AngFinal) && ((sqrt((xcr-x2)^2+(ycr-y2)^2)/2)<=dist_recta
&& ang2>=objeto.arcos(i).AngInicial && ang2<=objeto.arcos(i).AngFinal)
    % Ambos puntos intersectan
```

```
    if(sqrt((xr1-x1)^2+(yr1-y1)^2)/2)<(sqrt((xr1-x2)^2+(yr1-y2)^2)/2)
        % Punto 1 está más cerca del inicio de la recta que el punto 2
        objeto.aristas(length(objeto.aristas)+1) = objeto.aristas(j);
        objeto.arcos(length(objeto.arcos)+1) = objeto.arcos(i);
        xtemp = cos(ang1 + 10^-9)*radio+xc;
        ytemp = sin(ang1 + 10^-9)*radio+yc;
        objeto.vertices(length(objeto.vertices)+1,1) = x1;
        objeto.vertices(length(objeto.vertices),2) = y1;
        objeto.aristas(j).vertices(2) = length(objeto.vertices);
        if(verticeCorrecto(objeto.aristas(j), objeto, [xtemp, ytemp]))
            objeto.arcos(i).AngFinal = ang1;
            objeto.arcos(i).vertices(2) = length(objeto.vertices);
            objeto.arcos(length(objeto.arcos)).AngInicial = ang2;
            objeto.arcos(length(objeto.arcos)).vertices(1) =
length(objeto.vertices)+1;
        else
            objeto.arcos(i).AngInicial = ang1;
            objeto.arcos(i).vertices(1) = length(objeto.vertices);
            objeto.arcos(length(objeto.arcos)).AngFinal = ang2;
            objeto.arcos(length(objeto.arcos)).vertices(2) =
length(objeto.vertices)+1;
        end
        objeto.vertices(length(objeto.vertices)+1,1) = x2;
        objeto.vertices(length(objeto.vertices),2) = y2;
        objeto.aristas(length(objeto.aristas)).vertices(1) =
length(objeto.vertices);
    else
        % Punto 2 está más cerca del inicio de la recta que el punto 1
        objeto.aristas(length(objeto.aristas)+1) = objeto.aristas(j);
        objeto.arcos(length(objeto.arcos)+1) = objeto.arcos(i);
        xtemp = cos(ang2 + 10^-9)*radio+xc;
        ytemp = sin(ang2 + 10^-9)*radio+yc;
        objeto.vertices(length(objeto.vertices)+1,1) = x2;
        objeto.vertices(length(objeto.vertices),2) = y2;
        objeto.aristas(j).vertices(2) = length(objeto.vertices);
        if(verticeCorrecto(objeto.aristas(j), objeto, [xtemp, ytemp]))
            objeto.arcos(i).AngFinal = ang2;
            objeto.arcos(i).vertices(2) = length(objeto.vertices);
            objeto.arcos(length(objeto.arcos)).AngInicial = ang1;
            objeto.arcos(length(objeto.arcos)).vertices(1) =
length(objeto.vertices)+1;
        else
            objeto.arcos(i).AngInicial = ang2;
            objeto.arcos(i).vertices(1) = length(objeto.vertices);
            objeto.arcos(length(objeto.arcos)).AngFinal = ang1;
            objeto.arcos(length(objeto.arcos)).vertices(2) =
length(objeto.vertices)+1;
        end
        objeto.vertices(length(objeto.vertices)+1,1) = x1;
        objeto.vertices(length(objeto.vertices),2) = y1;
        objeto.aristas(length(objeto.aristas)).vertices(1) =
length(objeto.vertices);
    end
    obj=objeto;
else
    if((sqrt((xcr-x1)^2+(ycr-y1)^2)/2)<=dist_recta &&
ang1>=objeto.arcos(i).AngInicial && ang1<=objeto.arcos(i).AngFinal)
```

```
        %Punto 1 si intersecta y Punto 2 no
        xtemp = cos(ang1 + 10^-9)*radio+xc;
        ytemp = sin(ang1 + 10^-9)*radio+yc;
        if(verticeCorrecto(objeto.aristas(j), objeto, [xtemp, ytemp]))
            objeto.arcos(i).AngFinal = ang1;
            objeto.vertices(objeto.arcos(i).vertices(2),1) = x2;
            objeto.vertices(objeto.arcos(i).vertices(2),2) = y2;
        else
            objeto.arcos(i).AngInicial = ang1;
            objeto.vertices(objeto.arcos(i).vertices(1),1) = x2;
            objeto.vertices(objeto.arcos(i).vertices(1),2) = y2;
        end
        if xor((sqrt((xcr-xr1)^2+(ycr-yr1)^2))<(sqrt((xcr-xr2)^2+(ycr-
yr2)^2)),objeto.arcos(i).concavo==1)
            % Se mira la geometría para saber que punto de la arista hay
            % que sustituir
            objeto.vertices(objeto.aristas(j).vertices(1),1) = x1;
            objeto.vertices(objeto.aristas(j).vertices(1),2) = y1;
            obj=objeto;
        else
            objeto.vertices(objeto.aristas(j).vertices(2),1) = x1;
            objeto.vertices(objeto.aristas(j).vertices(2),2) = y1;
            obj=objeto;
        end
    elseif ((sqrt((xcr-x2)^2+(ycr-y2)^2)/2)<=dist_recta &&
ang2>=objeto.arcos(i).AngInicial && ang2<=objeto.arcos(i).AngFinal)
        %Punto 2 si intersecta y Punto 1 no
        xtemp = cos(ang2 + 10^-9)*radio+xc;
        ytemp = sin(ang2 + 10^-9)*radio+yc;
        if(verticeCorrecto(objeto.aristas(j), objeto, [xtemp, ytemp]))
            objeto.arcos(i).AngFinal = ang2;
            objeto.vertices(objeto.arcos(i).vertices(2),1) = x2;
            objeto.vertices(objeto.arcos(i).vertices(2),2) = y2;
        else
            objeto.arcos(i).AngInicial = ang2;
            objeto.vertices(objeto.arcos(i).vertices(1),1) = x2;
            objeto.vertices(objeto.arcos(i).vertices(1),2) = y2;
        end
        if xor((sqrt((xc-xr1)^2+(yc-yr1)^2))<(sqrt((xc-xr2)^2+(yc-
yr2)^2)),objeto.arcos(i).concavo==1)
            % Se mira la geometría para saber que punto de la arista hay
            % que sustituir.
            % Está mas cerca el vertice 1 si es convexo ó
            % Está mas cerca el vertice 2 si es concavo
            objeto.vertices(objeto.aristas(j).vertices(1),1) = x2;
            objeto.vertices(objeto.aristas(j).vertices(1),2) = y2;
            obj=objeto;
        else
            % Está mas cerca el vertice 2 si es convexo ó
            % Está mas cerca el vertice 1 si es concavo
            objeto.vertices(objeto.aristas(j).vertices(2),1) = x2;
            objeto.vertices(objeto.aristas(j).vertices(2),2) = y2;
            obj=objeto;
        end
    else
```

```
        %Ningun punto intersecta
        obj=Objeto(Inf,[],[]);
        return
    end
end
end
```

Función limpiarIntersecciones():

```
function objetoLimpio = limpiarIntersecciones(objeto)

objetoLimpio = objeto;

% Loop a través de todas las aristas para verificar si se intersectan
for i = 1:length(objetoLimpio.aristas)
    for j = i+1:length(objetoLimpio.aristas)
        objetoLimpio = interseccion_recta_recta(objetoLimpio, i, j);
    end
end

for i = 1:length(objetoLimpio.arcos)
    for j = i+1:length(objetoLimpio.arcos)
        if ~(interseccion_arco_arco(objetoLimpio, i, j).vertices==Inf)
            objetoLimpio=interseccion_arco_arco(objetoLimpio, i, j);
        end
    end
end

i = 1;
while i<=length(objetoLimpio.arcos)
    j=1;
    while j<=length(objetoLimpio.aristas)

         if ~(interseccion_arco_recta(objetoLimpio, i, j).vertices==Inf)
            objetoLimpio=interseccion_arco_recta(objetoLimpio, i, j);
            j=1;
            i=1;
        end
        j = j+1;
    end
    i = i+1;
end
end
```

Función verticeCorrecto():

```
function [lado] = verticeCorrecto(arista, objeto, punto)
%Obtener los puntos de la arista
p1 = objeto.vertices(arista.vertices(1),:);
p2 = objeto.vertices(arista.vertices(2),:);

if size(p2, 2) < 2
    error('La arista debe tener al menos dos vértices.');
end
```

```
normal = arista.normal;

if normal > pi
    normal = normal - 2*pi;
elseif normal < -pi
    normal = normal + 2*pi;
end

% Calcular el ángulo entre la normal y la recta perpendicular
angulo = mod(atan2(punto(2)-p1(2), punto(1)-p1(1)), 2*pi);

angulo = angulo - normal;

if angulo > pi
    angulo = angulo - 2*pi;
elseif angulo < -pi
    angulo = angulo + 2*pi;
end

if angulo >= pi/2 || angulo <= -pi/2
    lado = true;
    %color = 'y';
else
    lado = false;
    %color = 'g';
end

%     % Dibujar el rectángulo (Esto se hizo por el proceso de debug)
%     figure
%     rectangle('Position',[0, 4, 2, 1],'LineWidth',1.5);
%     axis equal;
%     hold on;
%
%     % Dibujar la arista en rojo y la normal en azul
%     plot([p1(1), p2(1)], [p1(2), p2(2)], 'r', 'LineWidth', 1.5);
%     quiver(p1(1), p1(2), cos(normal), sin(normal), 'b', 'LineWidth', 1.5,
'MaxHeadSize', 0.8);
%
%     % Dibujar el punto en verde o amarillo según el lado en que esté
%     scatter(punto(1), punto(2), 100, color, 'filled');
%     scatter(p1(1), p1(2), 10, 'k', 'filled');
%     scatter(p2(1), p2(2), 10, 'k', 'filled');
%
%     hold off;
end
```

Función contraerObjeto():

```
function objContraido = contraerObjeto(obj, distancia)

    hold on;
    VerticesExpandidos = zeros(length(obj.vertices),2,2);   % Variable que almacena
los vertices expandidos (dos para cada vertice original)
    VerticesNuevos = zeros(length(obj.vertices),2);         % Variable que almacena
los vertices para crear el objeto expandido
```

```
    VectoresExpandidos = zeros(length(obj.vertices),2);     % Variable que almacena
los vectores direccion final de cada Arista o Arco
    AristasExpandidas = [];                                 % Lista de Aristas
nuevas
    ArcosExpandidos = [];                                   % Lista de Arcos nuevos
    % Expandir aristas
    for i = 1:length(obj.aristas)

        a = obj.aristas(i);
        Vo = obj.vertices(a.vertices(1),:);             % Se obtiene el vector
inicial
        Vf = obj.vertices(a.vertices(2),:);             % Se obtiene el vector
final
        normal = a.normal;
        Xnormal = cos(normal);                          % Se divide la normal en
coordenada unitaria X
        Ynormal = sin(normal);                          % y en coordenada unitaria
Y
        Vx = [Vo(1), Vf(1)] + Xnormal*distancia;        % Se calcula las
coordenadas X expandidas
        Vy = [Vo(2), Vf(2)] + Ynormal*distancia;        % y las coordenadas Y
expandidas


        VerticesExpandidos(a.vertices(1),1,2) =  Vx(1);     % Se guarda la
coordenada Xi
        VerticesExpandidos(a.vertices(1),2,2) =  Vy(1);     % y la coordenada Yi
        VerticesExpandidos(a.vertices(2),1,1) =  Vx(2);     % Se guarda la
coordenada Xf
        VerticesExpandidos(a.vertices(2),2,1) =  Vy(2);     % y la coordenada Yf

        VectorX = Vf(1) - Vo(1);                            % Se calcula el vector
dirección de las aristas
        VectorY = Vf(2) - Vo(2);                            % para el eje X e Y
        n = norm([VectorX, VectorY]);                       % Se calcula el modulo
        VectoresExpandidos(a.vertices(2),1) = VectorX/n;    % Se calcula el vector
unitario
        VectoresExpandidos(a.vertices(2),2) = VectorY/n;


        AristaNueva = Arista([a.vertices(1)*2,a.vertices(2)*2-1], a.numero*2,
a.normal);      % Se crea una nueva arista como la expandida
        AristasExpandidas = [AristasExpandidas, AristaNueva];               % Se
guarda la arista expandida recien creada

    end

% Dibujar arcos
    for i = 1:length(obj.arcos)

        a = obj.arcos(i);

        if a.concavo == 1
            radio = a.radio - distancia;            % En caso de ser un arco
concavo se resta la distancia del radio
        else
```

```
            radio = a.radio + distancia;            % En caso de ser un arco
convexo se suma la distancia del radio
        end

        t = linspace(a.AngInicial, a.AngFinal);     % Se dibuja el arco desde el
angulo inicial y final del arco
        x = a.xc + radio*cos(t);
        y = a.yc + radio*sin(t);

        VerticesExpandidos(a.vertices(1),1,2) =  x(1);          % Se guarda la
coordenada Xi
        VerticesExpandidos(a.vertices(1),2,2) =  y(1);          % y la coordenada
Yi
        VerticesExpandidos(a.vertices(2),1,1) =  x(end);        % Se guarda la
coordenada Xf
        VerticesExpandidos(a.vertices(2),2,1) =  y(end);        % y la coordenada
Yf

        VectorX = x(end) - x(end - 1);                      % Se calcula el vector
de direccion final
        VectorY = y(end) - y(end - 1);                      % de los ejes X e Y
        n = norm([VectorX, VectorY]);                       % Se calcula el modulo
        VectoresExpandidos(a.vertices(2),1) = VectorX/n;    % Se calcula el vector
unitario
        VectoresExpandidos(a.vertices(2),2) = VectorY/n;

        ArcoNuevo = Arco([a.vertices(1)*2,a.vertices(2)*2-1], a.numero*2, a.xc,
a.yc, a.zc, a.AngInicial, a.AngFinal, radio, a.concavo);
        ArcosExpandidos = [ArcosExpandidos, ArcoNuevo];

    end

    % Expandir vertices
    for i = 1:length(obj.vertices)

%         v = obj.vertices(i,:);
%         anguloInicial = atan2(VerticesExpandidos(i,2,1)-
v(2),VerticesExpandidos(i,1,1)-v(1));   % Se calcula el angulo inicial basado en
los
%         anguloFinal = atan2(VerticesExpandidos(i,2,2)-
v(2),VerticesExpandidos(i,1,2)-v(1));     % vertices nuevos (expandidos) respecto
del centro de las aristas
        VerticesNuevos(i*2-1, 1) = VerticesExpandidos(i,1,1);       % Se almacenan
los nuevos vertices Xi
        VerticesNuevos(i*2-1, 2) = VerticesExpandidos(i,2,1);       % Yi
        VerticesNuevos(i*2, 1) = VerticesExpandidos(i,1,2);         % Xf
        VerticesNuevos(i*2, 2) = VerticesExpandidos(i,2,2);         % Yf para el
objeto expandido
%         if (anguloInicial < 0)
%             anguloInicial = anguloInicial + 2*pi;   % Se corrige en caso de ser
negativos
%         end
%
%         if (anguloFinal < 0)
%             anguloFinal = anguloFinal + 2*pi;       % Se corrige en caso de ser
negativos
```

```
%         end
%
%         t = linspace(anguloInicial,anguloFinal);    % Se calcula el circulo
alrededor del
%         x = v(1) + distancia*cos(t);                % vector basado en los
angulos
%         y = v(2) + distancia*sin(t);
%
%         VectorX = x(2) - x(1);                      % Se calcula el vector
direccion del comienzo
%         VectorY = y(2) - y(1);                      % del arco del vertice
%         n = norm([VectorX, VectorY]);
%         dir = [VectorX/n, VectorY/n];
%
%         if(dir(1)-VectoresExpandidos(i, 1) > 0.5 || dir(1)-VectoresExpandidos(i,
1) < -0.5)     % En caso de no ser una
%             anguloInicial = anguloInicial - 2*pi;
% direccion similar la del
%             t = linspace(anguloInicial,anguloFinal);
% comienzo del arco del vertice
%             x = v(1) + distancia*cos(t);
% a la del final del arco/arista
%             y = v(2) + distancia*sin(t);
% se le da la vuelta al arco
%         end
% invirtiendo el anguloInicial y
%
% repitiendo el arco
%
%         ArcoNuevo = Arco([i*2-1,i*2], i*2-1, v(1), v(2), 0, anguloInicial,
anguloFinal, distancia, 0);
%         ArcosExpandidos = [ArcosExpandidos, ArcoNuevo];
    end

    objContraido = Objeto(VerticesNuevos, AristasExpandidas, ArcosExpandidos);

end
```

Función anguloPuntoRecta():

```
function angulo = anguloPuntoRecta(p,p1,p2)
    % Verificar si la recta es vertical
    x0 = p(1);
    y0 = p(2);
    x1 = p1(1);
    y1 = p1(2);
    x2 = p2(1);
    y2 = p2(2);

    if x1 == x2
        y_proyectado = y0;
        x_proyectado = x1;
    else
        % Calcular las diferencias en x y y entre los puntos de la recta
        delta_x = x2 - x1;
        delta_y = y2 - y1;
```

```
        % Calcular la pendiente de la recta
        pendiente = delta_y / delta_x;

        % Calcular las coordenadas x e y del punto más cercano en la recta
        x_proyectado = (pendiente^2 * x1 + pendiente * (y0 - y1) + x0) /
(pendiente^2 + 1);
        y_proyectado = pendiente * (x_proyectado - x1) + y1;
    end

    % Calcular el ángulo respecto a la horizontal
    angulo = atan2((y0 - y_proyectado),(x0 - x_proyectado));
    angulo = mod(angulo+2*pi,2*pi);
end
```

Función comprobarRecta():

```
function Recta = comprobarRecta(Recta,entorno)

if Recta.valido == 1

    x = linspace(Recta.PuntoInicial(1), Recta.PuntoFinal(1), 20);
    y = linspace(Recta.PuntoInicial(2), Recta.PuntoFinal(2), 20);
    for n = 2:length(x)-1
        for i = 2:length(entorno.objetosExpandidos)
            if Recta.valido == 1
            a = entorno.objetosExpandidos(1,i);
            Fuera = 0;
            for j = 1:length(a.Vertices)-1
                angTemp = anguloPuntoRecta([x(n), y(n)],
[a.Vertices(j,1),a.Vertices(j,2)],[a.Vertices(j+1,1),a.Vertices(j+1,2)]);
                if abs(angTemp - a.Normales(j))>pi/2
                    Fuera = 1;
                end
            end
            if Fuera == 0
                Recta.valido = 0;
            end
            end
        end
    end

    for i = 2:length(entorno.objetosExpandidos)
        a = entorno.objetosExpandidos(1,i);
        for j = 1:length(a.Vertices)-1
            if Recta.valido == 1

                x1i = Recta.PuntoInicial(1);
                y1i = Recta.PuntoInicial(2);
                x1f = Recta.PuntoFinal(1);
                y1f = Recta.PuntoFinal(2);

                x2i = a.Vertices(j,1);
                y2i = a.Vertices(j,2);
                x2f = a.Vertices(j+1,1);
                y2f = a.Vertices(j+1,2);
```

```
                % Verificar si ambas rectas son verticales
                if x1i == x1f && x2i == x2f
                    % Comprobar si las rectas son paralelas
                    intersection = [Inf, Inf];
                    % Verificar si la recta 1 es vertical
                elseif x1i == x1f
                    % Calcular las coordenadas x e y del punto de intersección
                    x = x1i;
                    m2 = (y2f - y2i) / (x2f - x2i);
                    y = m2 * (x - x2i) + y2i;
                    intersection = [x, y];
                    % Verificar si la recta 2 es vertical
                elseif x2i == x2f
                    % Calcular las coordenadas x e y del punto de intersección
                    x = x2i;
                    m1 = (y1f - y1i) / (x1f - x1i);
                    y = m1 * (x - x1i) + y1i;
                    intersection = [x, y];
                else
                    % Calcular la pendiente de ambas rectas
                    m1 = (y1f - y1i) / (x1f - x1i);
                    m2 = (y2f - y2i) / (x2f - x2i);

                    % Comprobar si las rectas son paralelas
                    if m1 == m2
                        intersection = [Inf, Inf]; % No hay punto de intersección
                    else
                        % Calcular las coordenadas x e y del punto de intersección
                        x = (m1 * x1i - m2 * x2i - y1i + y2i) / (m1 - m2);
                        y = m1 * (x - x1i) + y1i;
                        intersection = [x, y];
                    end
                end
                xc1 = (x1i + x1f)/2;
                yc1 = (y1i + y1f)/2;
                r1 = sqrt(((x1i-xc1)*(x1i-xc1))+((y1i-yc1)*(y1i-yc1)));
                xc2 = (x2i + x2f)/2;
                yc2 = (y2i + y2f)/2;
                r2 = sqrt(((x2i-xc2)*(x2i-xc2))+((y2i-yc2)*(y2i-yc2)));

                if (sqrt(((intersection(1)-xc1)*(intersection(1)-
xc1))+((intersection(2)-yc1)*(intersection(2)-yc1))) < r1 &&
sqrt(((intersection(1)-xc2)*(intersection(1)-xc2))+((intersection(2)-
yc2)*(intersection(2)-yc2))) < r2)
                    Recta.valido = 0;
                end
            end
        end

        a = entorno.objetos(1,i);
        for j = 1:length(a.Vertices)-1
            if Recta.valido == 1

                x1i = Recta.PuntoInicial(1);
                y1i = Recta.PuntoInicial(2);
                x1f = Recta.PuntoFinal(1);
                y1f = Recta.PuntoFinal(2);
```

```
                x2i = a.Vertices(j,1);
                y2i = a.Vertices(j,2);
                x2f = a.Vertices(j+1,1);
                y2f = a.Vertices(j+1,2);

                % Verificar si ambas rectas son verticales
                if x1i == x1f && x2i == x2f
                    % Comprobar si las rectas son paralelas
                    if x1i ~= x2i
                        % No hay punto de intersección
                    else
                        % Las rectas son coincidentes, devolver uno de los puntos
                    End

                    % Verificar si la recta 1 es vertical
                elseif x1i == x1f
                    % Calcular las coordenadas x e y del punto de intersección
                    x = x1i;
                    m2 = (y2f - y2i) / (x2f - x2i);
                    y = m2 * (x - x2i) + y2i;
                    intersection = [x, y];

                    % Verificar si la recta 2 es vertical
                elseif x2i == x2f
                    % Calcular las coordenadas x e y del punto de intersección
                    x = x2i;
                    m1 = (y1f - y1i) / (x1f - x1i);
                    y = m1 * (x - x1i) + y1i;
                    intersection = [x, y];
                else
                    % Calcular la pendiente de ambas rectas
                    m1 = (y1f - y1i) / (x1f - x1i);
                    m2 = (y2f - y2i) / (x2f - x2i);

                    % Comprobar si las rectas son paralelas
                    if m1 == m2
                        intersection = [Inf, Inf]; % No hay punto de intersección
                    else
                        % Calcular las coordenadas x e y del punto de intersección
                        x = (m1 * x1i - m2 * x2i - y1i + y2i) / (m1 - m2);
                        y = m1 * (x - x1i) + y1i;
                        intersection = [x, y];
                    end
                end
                xc1 = (x1i + x1f)/2;
                yc1 = (y1i + y1f)/2;
                r1 = sqrt(((x1i-xc1)*(x1i-xc1))+((y1i-yc1)*(y1i-yc1)));
                xc2 = (x2i + x2f)/2;
                yc2 = (y2i + y2f)/2;
                r2 = sqrt(((x2i-xc2)*(x2i-xc2))+((y2i-yc2)*(y2i-yc2)));

                if (sqrt(((intersection(1)-xc1)*(intersection(1)-
xc1))+((intersection(2)-yc1)*(intersection(2)-yc1))) < r1 &&
sqrt(((intersection(1)-xc2)*(intersection(1)-xc2))+((intersection(2)-
yc2)*(intersection(2)-yc2))) < r2)
                    Recta.valido = 0;
                end
```

```
            end
        end
    end
end
end
```

Función crearEscenario():

```
function [escenario, escenarioContraido] = crearEscenario(p1,p2,robot)
% p1 = esquina inferior izquierda
% p2 = esquina superior derecha

Vertices = [p1;p2(1),p1(2);p2;p1(1),p2(2)];
escenario = ObstaculoPoligonal(Vertices);
max_x = max(robot.Vertices(:,1));
max_y = max(robot.Vertices(:,2));
min_x = min(robot.Vertices(:,1));
min_y = min(robot.Vertices(:,2));
VerticesContraido = [p1(1)-min_x,p1(2)-min_y;p2(1)-max_x,p1(2)-min_y;p2(1)-
max_x,p2(2)-max_y;p1(1)-min_x,p2(2)-max_y];
escenarioContraido = ObstaculoPoligonal(VerticesContraido);
end
```

Función dibujarPoligonal():

```
function dibujarPoligonal(objetoPoligonal, color)

    % Verificar si se especifica el color
    if nargin < 2
        color = 'k'; % Color por defecto: negro
    end

    % Crear un vector con los índices de los vértices
    i = [1:length(objetoPoligonal.Vertices) , 1];

    % Graficar las aristas del robot
    plot(objetoPoligonal.Vertices(i, 1), objetoPoligonal.Vertices(i, 2), [color,'-
'], 'LineWidth', 2);
end
```

Función expandirPoligonal():

```
function objExpandido = expandirPoligonal(Robot, Obstaculo)
NormalRobot = Robot.Normales;
NormalObstaculo = Obstaculo.Normales;
UnionNormales = [NormalRobot, NormalObstaculo];
NormalesIniciales = [NormalRobot(1),NormalRobot(end)];
k = 1;
NormalAnterior = 7;
i = 1;
if(NormalRobot(1)<NormalRobot(end))
    while((NormalObstaculo(i) > min(NormalesIniciales) && NormalObstaculo(i) <
max(NormalesIniciales)) && i < length(NormalObstaculo))
```

```
        i = i + 1;
    end
else
    while((NormalObstaculo(i) < min(NormalesIniciales) || NormalObstaculo(i) >
max(NormalesIniciales)) && i < length(NormalObstaculo))
        i = i + 1;
    end
end
NormalesOrdenadas = sort(UnionNormales);
indice = find(NormalesOrdenadas == NormalObstaculo(i),k);
PuntoOrigen = indice;
Vertices = zeros(length(NormalRobot)+length(NormalObstaculo), 2);
if(i==length(Obstaculo.Vertices))
    index = 0;
else
    index = i;
end
Vertices(1,1) = Obstaculo.Vertices(index + 1,1);
Vertices(1,2) = Obstaculo.Vertices(index + 1,2);
clear index;
indiceVertices = 2;
indice = indice - 1;
while (indice < length(UnionNormales))
    indice = indice + 1;
    if(NormalAnterior == NormalesOrdenadas(indice))
        k = k + 1;
    else
        k = 1;
    end
    indiceSinOrdenar = find(UnionNormales==NormalesOrdenadas(indice),k);
    NormalAnterior = NormalesOrdenadas(indice);
    indiceSinOrdenar = indiceSinOrdenar(k);
    if(indiceSinOrdenar <= length(NormalRobot))
        if(indiceSinOrdenar == length(NormalRobot))
            VectorIndependiente = [Robot.Vertices(indiceSinOrdenar, 1) -
Robot.Vertices(1, 1), Robot.Vertices(indiceSinOrdenar, 2) - Robot.Vertices(1, 2)];
        else
            VectorIndependiente = [Robot.Vertices(indiceSinOrdenar, 1) -
Robot.Vertices(indiceSinOrdenar + 1, 1), Robot.Vertices(indiceSinOrdenar, 2) -
Robot.Vertices(indiceSinOrdenar + 1, 2)];
        end
    else
        if(indiceSinOrdenar == length(NormalObstaculo) + length(NormalRobot))
            VectorIndependiente = [Obstaculo.Vertices(indiceSinOrdenar -
length(NormalRobot), 1) - Obstaculo.Vertices(1, 1),
Obstaculo.Vertices(indiceSinOrdenar - length(NormalRobot), 2) -
Obstaculo.Vertices(1, 2)];
        else
            VectorIndependiente = [Obstaculo.Vertices(indiceSinOrdenar -
length(NormalRobot), 1) - Obstaculo.Vertices(indiceSinOrdenar + 1 -
length(NormalRobot), 1), Obstaculo.Vertices(indiceSinOrdenar - length(NormalRobot),
2) - Obstaculo.Vertices(indiceSinOrdenar + 1 - length(NormalRobot), 2)];
        end
    end
    Vertices(indiceVertices, 1) = Vertices(indiceVertices - 1, 1) +
VectorIndependiente(1);
```

```
    Vertices(indiceVertices, 2) = Vertices(indiceVertices - 1, 2) +
VectorIndependiente(2);
    indiceVertices = indiceVertices + 1;
end
indice = 0;
while (indice < PuntoOrigen - 1)
    indice = indice + 1;
    if(NormalAnterior == NormalesOrdenadas(indice))
        k = k + 1;
    else
        k = 1;
    end
    indiceSinOrdenar = find(UnionNormales==NormalesOrdenadas(indice),k);
    NormalAnterior = NormalesOrdenadas(indice);
    indiceSinOrdenar = indiceSinOrdenar(k);
    if(indiceSinOrdenar <= length(NormalRobot))
        if(indiceSinOrdenar == length(NormalRobot))
            VectorIndependiente = [Robot.Vertices(indiceSinOrdenar, 1) -
Robot.Vertices(1, 1), Robot.Vertices(indiceSinOrdenar, 2) - Robot.Vertices(1, 2)];
        else
            VectorIndependiente = [Robot.Vertices(indiceSinOrdenar, 1) -
Robot.Vertices(indiceSinOrdenar + 1, 1), Robot.Vertices(indiceSinOrdenar, 2) -
Robot.Vertices(indiceSinOrdenar + 1, 2)];
        end
    else
        if(indiceSinOrdenar == length(NormalObstaculo) + length(NormalRobot))
            VectorIndependiente = [Obstaculo.Vertices(indiceSinOrdenar -
length(NormalRobot), 1) - Obstaculo.Vertices(1, 1),
Obstaculo.Vertices(indiceSinOrdenar - length(NormalRobot), 2) -
Obstaculo.Vertices(1, 2)];
        else
            VectorIndependiente = [Obstaculo.Vertices(indiceSinOrdenar -
length(NormalRobot), 1) - Obstaculo.Vertices(indiceSinOrdenar + 1 -
length(NormalRobot), 1), Obstaculo.Vertices(indiceSinOrdenar - length(NormalRobot),
2) - Obstaculo.Vertices(indiceSinOrdenar + 1 - length(NormalRobot), 2)];
        end
    end
    %Vertices(indiceVertices, 1) = Vertices(indiceVertices - 1, 1) +
VectorIndependiente(1);
    %Vertices(indiceVertices, 2) = Vertices(indiceVertices - 1, 2) +
VectorIndependiente(2);
    %indiceVertices = indiceVertices + 1;
end

%Vertices = Vertices(1:end - 1, :);

objExpandido = ObstaculoPoligonal(Vertices);

end
```

Función rotarVector():

```
function RobotRotado = rotar_vector(Robot, angulo_grados)
% ROTAR_VECTOR rota un robot en coordenadas polares alrededor del origen de
coordenadas
% dado un ángulo en grados.
vector = Robot.Vertices;
% Convertir ángulo a radianes
angulo_radianes = deg2rad(angulo_grados);

% Descomponer vector en coordenadas cartesianas
x = vector(:,1);
y = vector(:,2);

% Aplicar rotación
x_rot = x*cos(angulo_radianes) - y*sin(angulo_radianes);
y_rot = x*sin(angulo_radianes) + y*cos(angulo_radianes);


x_rot = round(x_rot,6);
y_rot = round(y_rot,6);

RobotRotado = RobotPoligonal([x_rot, y_rot]);

end
```

Función dijkstra():

```
function [rutaFinal, costeFinal] = dijkstra(matrizRectas, matrizDistancias,radio)
abiertos = 1:length(matrizDistancias)-2;
ruta = cell(1,length(matrizDistancias)-1);
IDmenor = 0;
IDinicial = length(matrizDistancias)-1;
for a = 1:IDinicial
    ruta{a} = [IDinicial a];
end
while ~isempty(abiertos) && IDmenor ~= length(matrizDistancias)-2
    menorDistancia = Inf;
    for i = 1:length(abiertos)
        if matrizDistancias(end-1,abiertos(i)) < menorDistancia
            menorDistancia = matrizDistancias(end-1,abiertos(i));
            IDmenor = abiertos(i);
        end
    end
    abiertos(abiertos == IDmenor) = [];

    for i = abiertos
        if matrizDistancias(IDmenor,i) ~= Inf
            rutaTemp = ruta{IDmenor};
            angTemp = anguloDijkstra(matrizRectas{IDmenor,rutaTemp(end-
1)},matrizRectas{i,IDmenor});
            if matrizDistancias(end-1,i) > matrizDistancias(end-1, IDmenor) +
matrizDistancias(IDmenor, i) + angTemp*radio
                ruta{i} = [ruta{IDmenor}, i];
                matrizDistancias(end-1,i) = matrizDistancias(end-1, IDmenor) +
matrizDistancias(IDmenor, i) + angTemp*radio;
```

```
                matrizDistancias(i,end-1) = matrizDistancias(end-1, IDmenor) +
matrizDistancias(IDmenor, i) + angTemp*radio;
            end
        end
    end
end
rutaFinal = ruta{end-1};
costeFinal = matrizDistancias(end-1,end-2);
for i = 1:length(rutaFinal)-1
    graficar(matrizRectas{rutaFinal(i),rutaFinal(i+1)});
end
end
```

Función rectasVisibilidad():

```
function [matrizRectas, matrizDistancias] =
rectasVisibilidad(entorno,puntoInicial,puntoFinal)
% Punto inicial end-1
% Punto final end-2
grosorDeLinea = 1.5;
[matrizVertices, matrizDistancias] = extraerVertices(entorno);
matrizRectas = cell(length(matrizDistancias(1,:)));
caminoTemp = crearCamino(puntoInicial,puntoFinal,matrizVertices,2);
if caminoTemp.valido == 1
    matrizRectas{end-1,end-2} = caminoTemp;
    matrizRectas{end-2,end-1} = caminoTemp;
    matrizDistancias(end-1,end-2) = distancia(caminoTemp);
    matrizDistancias(end-2,end-1) = distancia(caminoTemp);
    plot([caminoTemp.p1(1), caminoTemp.p2(1)],[caminoTemp.p1(2),
caminoTemp.p2(2)],'.-g', 'LineWidth', grosorDeLinea);
else
    %plot([caminoTemp.p1(1), caminoTemp.p2(1)],[caminoTemp.p1(2),
caminoTemp.p2(2)],'.-b');
end
for i = 1:length(matrizVertices(:,1))
    matrizTemp = [];
    for j = 1:length(matrizVertices(i,:))
        % Obtenemos cada vertice y comprobamos si es curvo y existe
        if ~isempty(matrizVertices{i,j})
            if matrizVertices{i,j}.curvo ~= 0
                %matrizRectas = [matrizRectas ,matrizVertices{i,j}];
                graficar(matrizVertices{i,j});
                % Intentamos crear un camino con el punto inicial
                caminoTemp =
crearCamino(matrizVertices{i,j},puntoInicial,matrizVertices,1);
                if caminoTemp.valido == 1
                    matrizRectas{(i-1)*matrizDistancias(end,end)+j,end-1} =
caminoTemp;
                    matrizRectas{end-1,(i-1)*matrizDistancias(end,end)+j} =
caminoTemp;
                    plot([caminoTemp.p1(1), caminoTemp.p2(1)],[caminoTemp.p1(2),
caminoTemp.p2(2)],'.-g', 'LineWidth', grosorDeLinea);
                    matrizDistancias((i-1)*matrizDistancias(end,end)+j,end-1) =
distancia(caminoTemp);
```

```
                    matrizDistancias(end-1,(i-1)*matrizDistancias(end,end)+j) =
distancia(caminoTemp);
                else
                    %plot([caminoTemp.p1(1), caminoTemp.p2(1)],[caminoTemp.p1(2),
caminoTemp.p2(2)],'.-b');
                end
                % Y con el punto final
                caminoTemp =
crearCamino(matrizVertices{i,j},puntoFinal,matrizVertices,1);
                if caminoTemp.valido == 1
                    matrizRectas{(i-1)*matrizDistancias(end,end)+j,end-2} =
caminoTemp;
                    matrizRectas{end-2,(i-1)*matrizDistancias(end,end)+j} =
caminoTemp;
                    plot([caminoTemp.p1(1), caminoTemp.p2(1)],[caminoTemp.p1(2),
caminoTemp.p2(2)],'.-g', 'LineWidth', grosorDeLinea);
                    matrizDistancias((i-1)*matrizDistancias(end,end)+j,end-2) =
distancia(caminoTemp);
                    matrizDistancias(end-2,(i-1)*matrizDistancias(end,end)+j) =
distancia(caminoTemp);
                else
                    %plot([caminoTemp.p1(1), caminoTemp.p2(1)],[caminoTemp.p1(2),
caminoTemp.p2(2)],'.-b');
                end
                % Probamos con el resto de vertices
                for k = i+1:length(matrizVertices(:,1))
                    for l = 1:length(matrizVertices(i,:))
                        if ~isempty(matrizVertices{k,l})
                            if matrizVertices{k,l}.curvo ~= 0
                                % Si existe y es curvo incluirlo en la lista
                                caminoTemp =
crearCamino(matrizVertices{i,j},matrizVertices{k,l},matrizVertices);
                                if caminoTemp .valido == 1
                                    matrizRectas{(i-
1)*matrizDistancias(end,end)+j,(k-1)*matrizDistancias(end,end)+l} = caminoTemp;
                                    matrizRectas{(k-
1)*matrizDistancias(end,end)+l,(i-1)*matrizDistancias(end,end)+j} = caminoTemp;
                                    plot([caminoTemp.p1(1), caminoTemp.p2(1)],
[caminoTemp.p1(2), caminoTemp.p2(2)], '.-g', 'LineWidth', grosorDeLinea);
                                    matrizDistancias((i-
1)*matrizDistancias(end,end)+j,(k-1)*matrizDistancias(end,end)+l) =
distancia(caminoTemp);
                                    matrizDistancias((k-
1)*matrizDistancias(end,end)+l,(i-1)*matrizDistancias(end,end)+j) =
distancia(caminoTemp);
                                else
                                    %plot([caminoTemp.p1(1),
caminoTemp.p2(1)],[caminoTemp.p1(2), caminoTemp.p2(2)],'.-b');
                                end
                            end
                        end
                    end
                end
            else
                matrizTemp = [matrizTemp ,matrizVertices{i,j}];
```

```
                plot([matrizVertices{i,j}.p1(1),
matrizVertices{i,j}.p2(1)],[matrizVertices{i,j}.p1(2),
matrizVertices{i,j}.p2(2)],'.-g', 'LineWidth', grosorDeLinea);
            end
        end
    end
    var = length(matrizTemp);
    for w = 1:var-1
        matrizDistancias((i-1)*matrizDistancias(end,end)+var+w,(i-
1)*matrizDistancias(end,end)+var+w+1) = distancia(matrizTemp(w));
        matrizDistancias((i-1)*matrizDistancias(end,end)+var+w+1,(i-
1)*matrizDistancias(end,end)+var+w) = distancia(matrizTemp(w));
        matrizRectas{(i-1)*matrizDistancias(end,end)+var+w,(i-
1)*matrizDistancias(end,end)+var+w+1} = matrizTemp(w);
        matrizRectas{(i-1)*matrizDistancias(end,end)+var+w+1,(i-
1)*matrizDistancias(end,end)+var+w} = matrizTemp(w);
    end
    matrizDistancias((i-1)*matrizDistancias(end,end)+var*2,(i-
1)*matrizDistancias(end,end)+var+1) = distancia(matrizTemp(var));
    matrizDistancias((i-1)*matrizDistancias(end,end)+var+1,(i-
1)*matrizDistancias(end,end)+var*2) = distancia(matrizTemp(var));
    matrizRectas{(i-1)*matrizDistancias(end,end)+var*2,(i-
1)*matrizDistancias(end,end)+var+1} = matrizTemp(w);
    matrizRectas{(i-1)*matrizDistancias(end,end)+var+1,(i-
1)*matrizDistancias(end,end)+var*2} = matrizTemp(w);
end
end
```

Función matrizCoste():

```
function [matrizCostes ,matrizRectas]=matrizCoste(entorno,puntoInicial,puntoFinal)

verticesMax = 0;
for j = 1:length(entorno.objetosExpandidos(1,:))
    verticesMax = max(verticesMax,length(entorno.objetosExpandidos(1,j).Vertices));
end
matrizCostes = ones(verticesMax*(length(entorno.objetosExpandidos))+3)*Inf;
for i = 1:length(matrizCostes)-1
    matrizCostes(i,i) = 0;
end
matrizCostes(end,end) = verticesMax;
matrizRectas = cell(length(matrizCostes(1,:)));

for i = 1:length(entorno.objetosExpandidos)
    a = entorno.objetosExpandidos(1,i);
    for j = 1:length(a.Vertices)
        RectaTemp = Recta(a.Vertices(j,:), puntoInicial);
        RectaTemp = comprobarRecta(RectaTemp,entorno);
        graficar(RectaTemp);
        if RectaTemp.valido == 1
            matrizRectas{(i-1)*matrizCostes(end,end)+j,end-1} = RectaTemp;
            matrizRectas{end-1,(i-1)*matrizCostes(end,end)+j} = RectaTemp;
            matrizCostes((i-1)*matrizCostes(end,end)+j,end-1) =
distancia(RectaTemp);
```

```
            matrizCostes(end-1,(i-1)*matrizCostes(end,end)+j) =
distancia(RectaTemp);
        end

        RectaTemp = Recta(a.Vertices(j,:), puntoFinal);
        RectaTemp = comprobarRecta(RectaTemp,entorno);
        graficar(RectaTemp);
        if RectaTemp.valido == 1
            matrizRectas{(i-1)*matrizCostes(end,end)+j,end-2} = RectaTemp;
            matrizRectas{end-2,(i-1)*matrizCostes(end,end)+j} = RectaTemp;
            matrizCostes((i-1)*matrizCostes(end,end)+j,end-2) =
distancia(RectaTemp);
            matrizCostes(end-2,(i-1)*matrizCostes(end,end)+j) =
distancia(RectaTemp);
        end

        if(a.Vertices(j,:)==a.Vertices(end,:))
            RectaTemp = Recta(a.Vertices(j,:), a.Vertices(1,:));
            RectaTemp = comprobarRecta(RectaTemp,entorno);
            graficar(RectaTemp);
            if RectaTemp.valido == 1
                matrizRectas{(i-1)*matrizCostes(end,end)+j,(i-
1)*matrizCostes(end,end)+1} = RectaTemp;
                matrizRectas{(i-1)*matrizCostes(end,end)+1,(i-
1)*matrizCostes(end,end)+j} = RectaTemp;
                matrizCostes((i-1)*matrizCostes(end,end)+j,(i-
1)*matrizCostes(end,end)+1) = distancia(RectaTemp);
                matrizCostes((i-1)*matrizCostes(end,end)+1,(i-
1)*matrizCostes(end,end)+j) = distancia(RectaTemp);
            end
        else
            RectaTemp = Recta(a.Vertices(j,:), a.Vertices(j+1,:));
            RectaTemp = comprobarRecta(RectaTemp,entorno);
            graficar(RectaTemp);
            if RectaTemp.valido == 1
                matrizRectas{(i-1)*matrizCostes(end,end)+j,(i-
1)*matrizCostes(end,end)+j+1} = RectaTemp;
                matrizRectas{(i-1)*matrizCostes(end,end)+j+1,(i-
1)*matrizCostes(end,end)+j} = RectaTemp;
                matrizCostes((i-1)*matrizCostes(end,end)+j,(i-
1)*matrizCostes(end,end)+j+1) = distancia(RectaTemp);
                matrizCostes((i-1)*matrizCostes(end,end)+j+1,(i-
1)*matrizCostes(end,end)+j) = distancia(RectaTemp);
            end
        end

        for k = (i+1):length(entorno.objetosExpandidos)
            b = entorno.objetosExpandidos(1,k);
            for l = 1:length(b.Vertices)
                RectaTemp = Recta(a.Vertices(j,:), b.Vertices(l,:));
                RectaTemp = comprobarRecta(RectaTemp,entorno);
                graficar(RectaTemp);
                if RectaTemp.valido == 1
                    matrizRectas{(i-1)*matrizCostes(end,end)+j,(k-
1)*matrizCostes(end,end)+l} = RectaTemp;
                    matrizRectas{(k-1)*matrizCostes(end,end)+l,(i-
1)*matrizCostes(end,end)+j} = RectaTemp;
```

```
                    matrizCostes((i-1)*matrizCostes(end,end)+j,(k-
1)*matrizCostes(end,end)+l) = distancia(RectaTemp);
                    matrizCostes((k-1)*matrizCostes(end,end)+l,(i-
1)*matrizCostes(end,end)+j) = distancia(RectaTemp);
                end
            end
        end
    end
end
end
```

Función Generar_puntos_aleatorios():

```
function puntosGenerados = Generar_puntos_aleatorios(entorno, densidad)
%La propiedad 'densidad' definirá la cantidad de puntos por unidad al
%cuadrado
x1 = entorno.objetosExpandidos(1).vertices(1,1);
y1 = entorno.objetosExpandidos(1).vertices(1,2);
x2 = entorno.objetosExpandidos(1).vertices(5,1);
y2 = entorno.objetosExpandidos(1).vertices(5,2);
Nodo0 = Nodo([0,0]);
puntosGenerados = repmat(Nodo0,1,ceil((x1-x2)*(y1-y2)*densidad)+2);
j = 1;
for i = 1:ceil((x1-x2)*(y1-y2)*densidad)

    % Generar coordenada x aleatoria dentro del rango [x1, x2]
    x = rand() * (x2 - x1) + x1;

    % Generar coordenada y aleatoria dentro del rango [y1, y2]
    y = rand() * (y2 - y1) + y1;

    % Crear el vector de coordenadas
    coordenadas = [x, y];
    NodoTemp = Nodo(coordenadas);
    NodoTemp = comprobarNodo(NodoTemp, entorno);
    graficar(NodoTemp);
    if NodoTemp.valido
        puntosGenerados(j) = NodoTemp;
        j = j + 1;
    end
end

puntosGenerados = puntosGenerados(1:j-1);
end
```

Función comprobarNodo():

```
function Nodo = comprobarNodo(Nodo, entorno)

for i = 2:length(entorno.objetosExpandidos)
    a = entorno.objetosExpandidos(i);
    fuera = 0;
    for j = 1:length(a.aristas)
p1=[a.vertices(a.aristas(j).vertices(1),1),a.vertices(a.aristas(j).vertices(1),2)];
```

```
p2=[a.vertices(a.aristas(j).vertices(2),1),a.vertices(a.aristas(j).vertices(2),2)];
        angTemp = anguloPuntoRecta(Nodo.coordenadas,p1,p2);
        if abs(angTemp - a.aristas(j).normal) <= pi/2
           fuera = 1;
        end
    end
    for j = 1:length(a.arcos)
        b = a.arcos(j);
        angTemp = atan2(Nodo.coordenadas(2) - b.yc, Nodo.coordenadas(1) - b.xc);
        if b.AngInicial >= 0 && b.AngFinal > 0
            angTemp = mod(angTemp + 2*pi, 2*pi);
        end
        if angTemp > b.AngInicial && angTemp < b.AngFinal
           d=sqrt((Nodo.coordenadas(2) - b.yc)^2 + (Nodo.coordenadas(1) - b.xc)^2);
            if d > b.radio
                fuera = 1;
            end
        end
    end
    if fuera == 0
        Nodo.valido = 0;
    end
end
end
```

Código de diagrama de visibilidad en entorno para robot cilíndrico:

```
 [matrizRectas, matrizDistancias] =
rectasVisibilidad(entorno,puntoInicial,puntoFinal);
 [rutaFinal, costeFinal] = dijkstra(matrizRectas, matrizDistancias,radio);
 plot(puntoInicial(1), puntoInicial(2), 'bp', 'MarkerSize', 7, 'LineWidth', 2);
 plot(puntoFinal(1), puntoFinal(2), 'bp', 'MarkerSize', 7, 'LineWidth', 2);
 axis equal
```

Código de diagrama de visibilidad en entorno para robot poligonal:

```
puntoInicial = [5.3,2];
puntoFinal = [6.5,9];
[matrizCostes, matrizRectas] = matrizCoste(entorno,puntoInicial,puntoFinal);
[rutaFinal, costeFinal] = Djikstra(matrizCostes, matrizRectas);
```

Código de método de los mapas de camino probabilístico en entorno para robot cilíndrico:

```
PuntosAleatorios = Generar_puntos_aleatorios(entorno, densidadDePuntos);

%Se añade el punto Final
NodoTemp = Nodo(puntoFinal);
NodoTemp = comprobarNodo(NodoTemp, entorno);
PuntosAleatorios(end+1) = NodoTemp;
%Se añade el punto Inicial
NodoTemp = Nodo(puntoInicial);
NodoTemp = comprobarNodo(NodoTemp, entorno);
PuntosAleatorios(end+1) = NodoTemp;

[matrizLineas, matrizDistancias] = GenerarRectasPRM(PuntosAleatorios, entorno);
[rutaFinal, costeFinal, matrizLineas] = dijkstra(matrizLineas, matrizDistancias,
entorno, densidadDePuntos, PuntosAleatorios);
plot(puntoInicial(1), puntoInicial(2), 'mp', 'MarkerSize', 7, 'LineWidth', 2);
plot(puntoFinal(1), puntoFinal(2), 'mp', 'MarkerSize', 7, 'LineWidth', 2);
axis equal
animar(radio, matrizLineas, rutaFinal, puntoInicial);
```

Código de método de los mapas de camino probabilístico en entorno para robot poligonal:

```
PuntosAleatorios = Generar_puntos_aleatorios(entorno, densidadDePuntos);

%Se añade el punto Final
NodoTemp = Nodo(puntoFinal);
NodoTemp = comprobarNodo(NodoTemp, entorno);
PuntosAleatorios(end+1) = NodoTemp;
%Se añade el punto Inicial
NodoTemp = Nodo(puntoInicial);
NodoTemp = comprobarNodo(NodoTemp, entorno);
PuntosAleatorios(end+1) = NodoTemp;

[matrizLineas, matrizDistancias] = GenerarRectasPRM(PuntosAleatorios, entorno);
[rutaFinal, costeFinal, matrizLineas] = dijkstra(matrizLineas, matrizDistancias,
entorno, densidadDePuntos, PuntosAleatorios);
plot(puntoInicial(1), puntoInicial(2), 'mp', 'MarkerSize', 7, 'LineWidth', 2);
plot(puntoFinal(1), puntoFinal(2), 'mp', 'MarkerSize', 7, 'LineWidth', 2);
axis equal
```

Código inicializar_RRT

```
clear all
close all
% Declaración y representación del entorno:
CrearEntorno;     % Declarar la matriz del entorno.
global arbol
global x_min
global x_max
global y_min
global y_max
% Crear el nodo de inicio:
start=Nodo;
start.nombre='inicio';
start.numero=0;
start.x=puntoInicial(1);
start.y=puntoInicial(2);
start.coste=0;
plot(start.x,start.y,'*')
% Crear el nodo objetivo:
goal=Nodo;
goal.nombre='objetivo';
goal.x=puntoFinal(1);
goal.y=puntoFinal(2);
goal.coste=Inf;
plot(goal.x,goal.y,'o')
lista_puntos=[start.x, start.y];  % Inicializar la lista de nodos del árbol.
nodo0000=start;
max_iteraciones=5000;  % Número máximo de iteraciones despues del cual se para la
búsqueda:
min_distancia=0.2;      % Distacia del objetivo a partir de la cual para la
busqueda;
long_rama=0.2;          % Longitud de las ramas
arbol=Nodo;
arbol=start;
```

Código de método de los árboles de búsqueda rápidamente explorables:

```
clear all
close all
% Declaración y representación del entorno:
CrearEntorno;     % Declarar la matriz del entorno.
global arbol
global x_min
global x_max
global y_min
global y_max
% Crear el nodo de inicio:
start=Nodo;
start.nombre='inicio';
start.numero=0;
start.x=puntoInicial(1);
start.y=puntoInicial(2);
start.coste=0;
plot(start.x,start.y,'*')
```

```
% Crear el nodo objetivo:
goal=Nodo;
goal.nombre='objetivo';
goal.x=puntoFinal(1);
goal.y=puntoFinal(2);
goal.coste=Inf;
plot(goal.x,goal.y,'o')
lista_puntos=[start.x, start.y];  % Inicializar la lista de nodos del árbol.
nodo0000=start;
max_iteraciones=5000;  % Número máximo de iteraciones despues del cual se para la
búsqueda:
min_distancia=0.2;      % Distacia del objetivo a partir de la cual para la
busqueda;
long_rama=0.2;          % Longitud de las ramas
arbol=Nodo;
arbol=start;
```

Bibliografía

- Esteve Gonzálvez, I. (2022). Planificación de movimiento de robots mediante la descomposición del entorno en celdas. Universitat Politècnica de València. http://hdl.handle.net/10251/185095
- Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269-271. https://gdz.sub.unigoettingen.de/id/PPN362160546_0001?tify=%7B%22pages%22%3A%5B273%5D%2C%22view%22%3A%22info%22%7D
- Markoff, J. (2002, August 27). Edsger Dijkstra and his algorithm. The New York Times. https://www.nytimes.com/2002/08/10/us/edsger-dijkstra-72-physicist-who-shaped-computer-era.html
- L. E. Kavraki, P. Svestka, J.-C. Latombe and M. H. Overmars (Aug. 1996), "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," in IEEE Transactions on Robotics and Automation, vol. 12, no. 4. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=508439&isnumber=11078
- Steven M. LaValle (1998), “Rapidly-Exploring Random Trees: A New Tool for Path Planning” http://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf
- Steven M. LaValle (2006), “Planning Algorithms” http://lavalle.pl/planning/
- Nehmzow, U. (2013). Mobile Robotics: A Practical Introduction. CRC Press. https://ebookcentral.proquest.com/lib/bibliotecaupves-ebooks/detail.action?docID=3339191&pq-origsite=primo
- Carlos Juárez (2022), “10 años de robots en Amazon: tecnología para clasificar paquetes y mover productos” The Logistics World. https://thelogisticsworld.com/innovacion/robots-de-amazon-para-clasificar-paquetes-y-mover-productos/
- iRobot. (s.f.). Roomba. Recuperado de https://www.irobot.es/es_ES/roomba.html