UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Simulación de métodos de planificación de movimiento de robots.

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

AUTOR/A: Martínez Martínez, Luis

Tutor/a: Zotovic Stanisic, Ranko

Cotutor/a externo: GONZALEZ QUERUBIN, EDWIN ALONSO

CURSO ACADÉMICO: 2021/2022

Resumen

Este trabajo consiste en la implementación de funciones en Matlab que permitan la planificación de movimientos para un brazo robótico de seis grados de libertad, concretamente el IRB 140 de ABB; utilizando los algoritmos PRM (*Probabilistic Roadmap*) y RRT (*Rapidly-exploring Random Tree*). También forma parte del proyecto el desarrollo de una simulación que posibilite la visualización de las trayectorias generadas. Las rutas creadas deben alcanzar un punto final partiendo de una configuración inicial evitando las colisiones tanto con elementos del entorno como entre eslabones del propio robot. Para ello, primeramente, se importa el modelo CAD (*Computer-Aided Design*) del robot en Simscape Multibody para la simulación, realizando los ajustes pertinentes que permitan el movimiento del brazo. A continuación, se define el entorno con los obstáculos presentes en él y se desarrollan funciones que verifiquen si se produce o no alguna colisión en una configuración dada. Posteriormente, se implementan los algoritmos PRM y RRT, además de una variante de este último, denominada RRT* (RRT estrella) que reduce el coste de la trayectoria generada. Por otro lado, se crea también un modelo en Simulink para la simulación en Simscape Multibody de los movimientos. Con todo ello, se obtienen programas capaces de planificar trayectorias que alcanzan el punto objetivo sin colisionar. La recogida y comparación de resultados ponen de manifiesto las ventajas e inconvenientes de un método sobre el otro. Así, mientras RRT es más rápido y garantiza el éxito si el camino existe, PRM obtiene una trayectoria con un coste menor.

Palabras clave: planificación de movimiento, trayectoria, brazo robótico, RRT, PRM, IRB 140 de ABB.

Resum

Aquest treball consisteix en la implemetació de funcions en Matlab que permeten la planificació de moviments per a un braç robòtic de sis graus de llibertat, concretament l'IRB 140 d'ABB; utilitzant els algoritmes PRM (*Probabilistic Roadmap*) i RRT (*Rapidly-exploring Random Tree*). També forma part del projecte el desenvolupament d'una simulació que possibilite la visualització de les trajectòries generades. Les rutes creades han d'arribar a un punt final partint d'una configuració inicial evitant les col·lisions tant amb elements de l'entorn com entre baules del propi robot. Amb aquesta finalitat, en primer lloc, s'importa el model CAD (*Computer-Aided Design*) del robot en Simscape Multibody per a la simulació, realitzant els ajustos necessaris que permeten el moviment del braç. A continuació, es defineix l'entorn amb els obstacles presents en ell y es desenvolupen funcions que verifiquen si es produix o no alguna col·lisió en una configuració determinada. Posteriorment, s'implemeten els algoritmes PRM i RRT, a més d'una variant d'aquest últim, denominada RRT* (RRT estrela) que reduix el cost de la trajectòria generada. Per un altre costat, es crea també un model en Simulink per a la simulació en Simscape Multibody dels moviments. Amb tot, s'obtenen programes capaços de planificar trajectòries que arriben al punt objectiu sense col·lisonar. La recollida i comparació dels resultats posen de manifest les avantatges e inconvenients d'un mètode respecte de l'altre. Així, metre que RRT es més ràpid i garanteix l'èxit si el camí existeix, PRM obté un trajecte amb un cost menor.

Paraules clau: planificació de moviment, trajectòria, braç robòtic, RRT, PRM, IRB 140 d'ABB.

Abstract

This work consists of the implementation of functions in Matlab which permit path planning for a robot with six degrees of freedom, in particular the IRB 140 of ABB; using the algorithms PRM (Probabilistic Roadmap) and RRT (Rapidly-exploring Random Tree). It is also a part of the project the development of a simulation which allows the visualization of the generated trajectories. The created paths must reach a final point starting from an initial configuration avoiding collisions both with objects in the environment and between links of the robot itself. To achieve that, firstly, the CAD (Computer-Aided Design) model of the robot is imported in Simscape Multibody for the simulation, making the necessary adjustments so that the arm can move. Then, the environment with all its obstacles is defined and functions to check if a collision takes place for a given configuration are developed. Lately, the algorithms PRM and RRT are implemented, and also a variant of the last one, known as RRT* (RRT star), which reduces the cost of the generated path. On the other hand, a model in Simulink for the simulation of the movements in Simscape Multibody is created. With all that, the project leads to a program able to plan paths that reach the objective point without any collision. The collection and comparison of results show the advantages and disadvantages of one method over the other. While RRT is faster and guaranties the success when a possible path exists, PRM obtains a trajectory with a lesser cost.

Key words: path planning, trajectory, robotic arm, RRT, PRM, IRB 140 of ABB.

Índice

Índice de figuras

Índice de tablas

Índice de gráficos

1. Introducción

En la actualidad, el uso de robots está muy extendido, especialmente en el ámbito industrial; aunque también está llegando al doméstico. Su producción y utilización han aumentado de forma exponencial en los últimos años y, según las previsiones, seguirán haciéndolo. Ante la creciente demanda y uso de dispositivos robotizados, se ha hecho necesario el desarrollo de algoritmos que permitan generar trayectorias de forma automática que los robots puedan seguir para desempeñar sus tareas

La planificación de trayectorias consiste en encontrar un camino para alcanzar un punto de destino partiendo desde otro de origen. En la generación de la ruta deben tenerse en cuenta las limitaciones del robot, así como la posibilidad de que los eslabones del robot choquen entre ellos o con los obstáculos que puedan hallarse en el entorno. El problema consiste, por tanto, en pasar de una configuración a otra sin sobrepasar los límites articulares del robot y evitando cualquier colisión, ya sea entre componentes del propio robot o con algún elemento externo presente en el área de trabajo.

Existen numerosos algoritmos enfocados a la resolución de este problema, tales como los gráficos de visibilidad, los mapas de Voronoi, el método de descomposición de celdas, la técnica de potencial artificial, PRM (*Probabilistic Roadmap*) o RRT (*Rapidly-exploring Random Tree*). Todos ellos han sido ampliamente desarrollados y estudiados en el ámbito de la robótica móvil, en entornos de dos dimensiones por los que se mueve el robot. Sin embargo, su aplicación en los brazos industriales ha sido menor, por la complejidad que supone trabajar en un entorno N-dimensional, donde N representa el número de grados de libertad del robot, que en estos casos suele ser superior a tres.

Este proyecto se centra en la aplicación de los algoritmos PRM y RRT a un brazo robótico industrial de seis grados de libertad. La elección de estos métodos frente al resto de las técnicas existentes es la mayor facilidad que presentan para trabajar en espacios de más de tres dimensiones, pues no requieren necesariamente de la modelación de los obstáculos en el espacio de configuraciones N-dimensional. En cuanto al modelo del robot, se ha optado por el IRB 140 de ABB, ya que se dispone de un ejemplar en los laboratorios del departamento de ingeniería de sistemas y automática de la UPV.

Para la realización del trabajo, se utilizan dos herramientas de Matlab ya desarrolladas. Se trata en primer lugar, de Multi-Parametric Toolbox, que permite operar con politopos, lo que será de gran ayuda para comprobar si se producen choques con los obstáculos, modelados como poliedros. También se emplea Simscape Multibody, que permite importar modelos en tres dimensiones de piezas mecánicas y simular su movimiento.

En el presente trabajo, se comienza estableciendo los sistemas de coordenadas asociados a cada eslabón, lo que permite calcular su cinemática, relacionando la posición y orientación del extremo con los valores articulares. Seguidamente, se importa el modelo CAD (*Computer-Aided Design)* del robot en Simscape Multibody para poder realizar simulaciones de las trayectorias generadas y se crea un entorno con una serie de obstáculos en el que debe operar el robot. Posteriormente se desarrollan funciones que permiten detectar colisiones tanto en una

configuración determinada como en el camino que une dos configuraciones diferentes. Seguidamente, se implementan los distintos algoritmos para generar las trayectorias, así como la simulación de estas. Finalmente, se estudian y comparan los resultados obtenidos con cada una de las técnicas utilizadas.

2. Objetivos

El objetivo de este proyecto es aplicar los algoritmos de planificación de trayectorias PRM (*Probablilistic Roadmap*) y RRT (*Rapidly-exploring Random Tree*) a un brazo robótico industrial de seis grados de libertad, concretamente al modelo IRB 140 de ABB. Con ello, se pretende obtener un programa que permita alcanzar una configuración determinada partiendo de otra inicial, también conocida, evitando en todo momento cualquier colisión tanto con el propio robot como con los objetos que se encuentran a su alrededor. Este fin se logra a través de los siguientes objetivos específicos:

- Definir un entorno que englobe el área de trabajo del brazo y los obstáculos que se hallen en ella.

- Desarrollar un algoritmo que permita comprobar si el robot colisiona en una configuración determinada o en la ruta que se ha de recorrer entre dos configuraciones, ya sea consigo mismo o con los obstáculos del entorno.

- Aplicar los métodos PRM y RRT para generar trayectorias que eviten aquellas configuraciones que provoquen la colisión del robot consigo mismo, con los obstáculos presentes en el área de trabajo, o que impliquen posiciones de las articulaciones que sobrepasen los límites establecidos por el fabricante.

- Implementar una simulación que permita visualizar los movimientos del robot en el entorno de trabajo para validar y comparar los resultados obtenidos con ambos métodos.

3. Marco teórico

En este apartado se explican los conceptos teóricos y las herramientas utilizadas durante el desarrollo del presente trabajo. Se exponen el método utilizado para la asignación de los sistemas de referencia de los eslabones del robot, el cálculo de su cinemática, la definición del espacio de configuraciones, distintos algoritmos de planificación de movimientos, haciendo hincapié en los que se desarrollan en este proyecto, y el software empleado.

3.1. Convención Denavit-Hartenberg

En el ámbito de la robótica se ha estandarizado el algoritmo Denavit-Hartenberg para describir la relación existente entre dos eslabones contiguos de un manipulador robótico. Es un modelo sencillo que permite describir las articulaciones y eslabones de un brazo robótico sin importar su secuencia o complejidad [1] .

Los robots pueden estar compuestos por distintos eslabones y articulaciones en cualquier orden. Las articulaciones pueden ser de revolución o prismáticas y estar en distintos planos, y los eslabones pueden tener cualquier longitud y moverse también en planos diferentes. Con el fin de poder analizar y modelar cualquier tipo de robot se asigna un sistema de referencia a cada eslabón y se definen transformaciones para pasar del sistema de un eslabón al sistema del siguiente. Combinando todas las transformaciones obtenidas, se puede establecer la relación entre el primero y el último [2].

Denavit y Hartenberg propusieron en 1955 un método matricial que dicta una serie de normas para determinar la localización de los sistemas de referencia asociados a cada uno de los eslabones del robot. Siguiendo tales directrices, los sistemas de referencia obtenidos son tales que puede pasarse de uno al siguiente mediante cuatro transformaciones básicas que dependen de la geometría del eslabón.

Las cuatro transformaciones que, según esta convención, permiten llegar al sistema de referencia *i* partiendo desde el *i-1* son las siguientes:

- Rotación con respecto al eje z_{i-1} un ángulo θ_i.
- Traslación en el eje z_{i-1} una distancia d_i.
- Traslación en el eje x_i una distancia a_i.
- Rotación con respecto al eje x_i un ángulo α_i.

Nótese que las transformaciones anteriores se refieren al sistema de referencia móvil y no al fijo, por lo que los sistemas se mueven junto con los eslabones de la cadena cinemática a los que están asociados.

La transformación para pasar del sistema *i-1* al *i* en forma matricial quedaría:

$$^{i-1}A_i = Rotz(\theta_i) \cdot Tras(0,0,d_i) \cdot Tras(a_i,0,0) \cdot Rotx(\alpha_i)$$

Desarrollando las matrices y realizando el producto se obtiene:

$$
{}^{i-1}A_i = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & 0 \\ s\theta_i & c\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c\alpha_i & -s\alpha_i & 0 \\ 0 & s\alpha_i & c\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
= \begin{bmatrix} c\theta_i & -c\alpha_i \cdot s\theta_i & s\alpha_i \cdot s\theta_i & a_i \cdot c\theta_i \\ s\theta_i & c\alpha_i \cdot c\theta_i & -s\alpha_i \cdot c\theta_i & a_i \cdot s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

donde θ_i, d_i, a_i y α_i son los parámetros Denavit-Hartenberg del eslabón *i* y las letras *c* y *s* indican coseno y seno respectivamente.

Las reglas que han de seguirse para situar los sistemas de referencia asociados a los eslabones se enumeran a continuación [1]:

- Se deben numerar los eslabones del robot utilizando números consecutivos. El eslabón cero será la base fija, el primer eslabón móvil será el uno y el último el *n*.
- Las articulaciones también se numeran del uno, correspondiente al primer grado de libertad, a *n*.
- Se localiza el eje de cada articulación. Si es prismática será el eje en el que se produce el desplazamiento del eslabón, mientras que si es de rotación será el eje de giro.
- Para cada eslabón, excluyendo el último (*i* de 0 a *n-1*), se sitúa el eje z_i en el eje de la articulación que lo une con el siguiente (articulación *i+1*).
- El sistema de origen de la base fija se podrá ubicar en cualquier punto del eje z_0. Los otros dos ejes (x_0 e y_0) deben situarse de forma tal que el sistema resultante sea dextrógiro.
- Para los eslabones móviles excepto el último (*i* de 1 hasta *n-1*), el origen del sistema se debe situar en la intersección entre el eje z_i y la normal común a z_{i-1} y z_i. En el caso de que ambos ejes se corten, el origen debe estar en el punto de intersección y si son paralelos, cualquiera de las infinitas rectas normales a ambos ejes sería válida para colocar el origen.
- El eje x_i debe coincidir con la normal común a z_{i-1} y z_i.
- El eje y_i debe situarse de forma tal que el sistema resultante sea dextrógiro.
- El último sistema de referencia, correspondiente a *n*, se ubica en el extremo de manera que las direcciones de z_n y z_{n-1} coincidan. El eje x_n debe coincidir con la normal común a z_n y z_{n-1}.
- Se calcula θ_i como el ángulo que se debe girar con respecto a z_{i-1} para que x_i y x_{i-1} queden paralelos. En las articulaciones de rotación este parámetro depende del valor de la articulación.
- Se obtiene d_i como la distancia que se debe desplazar el sistema de referencia *i-1* a lo largo del eje z_{i-1} para que x_i y x_{i-1} pasen a estar en la misma recta. En las articulaciones prismáticas este parámetro depende del valor de la articulación.
- Se halla a_i como la distancia que se debe desplazar el sistema de referencia *i-1* a lo largo del eje x_i para que su origen coincida con el del sistema *i*.
- Se calcula α_i como el ángulo que se debe girar con respecto al eje x_i para que los sistemas de referencia *i-1* e *i* coincidan.

Realizadas estas operaciones, se pueden obtener las matrices de transformación que relacionan el sistema de referencia de cada eslabón con el del eslabón siguiente. A partir de dichas matrices puede obtenerse otra que relaciona el sistema de la base con el del efector final del robot de la siguiente forma:

$$T = {}^{0}A_n = {}^{0}A_1 \cdot {}^{1}A_2 \cdot \ldots \cdot {}^{n-2}A_{n-1} \cdot {}^{n-1}A_n$$

La matriz T resultante define la posición y orientación del extremo del robot con respecto al sistema de referencia de la base fija [1].

3.2. Cinemática

En robótica, se utiliza este término para designar al estudio del movimiento de un robot con respecto a un sistema de referencia determinado sin atender a las fuerzas que lo provocan. Más concretamente, se centra en la relación entre los valores que toman las articulaciones y la posición y orientación del efector final para tales valores.

Este campo se divide en dos. Por un lado, la cinemática directa calcula la posición y orientación del extremo con respecto a un sistema de referencia a partir de los valores articulares y de la geometría del robot. Por otro lado, la cinemática inversa determina qué valor debe tomar cada una de las articulaciones para que el efector final alcance una posición y orientación concretas [1].

3.2.1. Cinemática Directa

La cinemática directa permite obtener la posición y orientación del extremo de un brazo robótico partiendo de los valores de sus articulaciones. El modelo cinemático directo puede calcularse por dos vías distintas: métodos geométricos y métodos basados en cambios de sistemas de referencia.

La primera opción no es un método sistemático, por lo que su aplicación queda limitada a robots simples con dos o tres grados de libertad. La segunda alternativa es mucho más utilizada, ya que aborda el problema de forma sistemática y, por tanto, permite resolverlo de manera más sencilla para robots con un número más elevado de grados de libertad.

Los métodos basados en cambios de sistemas de referencia utilizan el álgebra matricial para obtener la posición y orientación del extremo del robot con respecto a un sistema de referencia fijo, normalmente situado en la base. Puesto que el robot se puede describir como una sucesión de eslabones unidos por articulaciones que parten de la base y acaban en un extremo, basta con establecer las transformaciones entre sistemas asociados a los eslabones para, a partir de ellas, obtener una matriz de transformación que relacione el sistema fijo de la base con el extremo.

Existen múltiples transformaciones posibles para llevar a cabo estos cálculos, pero tal y como se ha comentado en el apartado 3.1, la convención Denavit-Hartenberg es el método más utilizado para obtener las matrices de transformación entre eslabones consecutivos y entre la base y el extremo.

Así pues, la forma más extendida de resolver la cinemática directa consiste simplemente en calcular la matriz de transformación que relaciona el sistema de referencia fijo de la base con el elemento final de la cadena cinemática mediante el algoritmo Denavit-Hartenberg. De este modo, se obtiene una submatriz de rotación, que define la orientación del extremo, y un vector de traslación, que indica su posición, en ambos casos con respecto a un sistema de referencia fijo [1].

3.2.2. Cinemática Inversa

El objetivo de la cinemática inversa es hallar los valores que deben tomar las articulaciones de una cadena cinemática para que su extremo alcance una posición y orientación determinadas. A diferencia del problema cinemático directo, que se puede resolver de manera sistemática a través de operaciones matriciales sin atender a la configuración del robot, los procedimientos para el cálculo de la cinemática inversa dependen de la configuración, ya que pueden existir diferentes configuraciones para las que el extremo presente igual posición y orientación.

La mayor parte de los robots presentan cinemáticas relativamente sencillas, lo que facilita en muchos casos los cálculos. En muchos robots de seis grados de libertad, los primeros tres, utilizados para posicionar el extremo, están contenidos en un mismo plano y los tres últimos, destinados a la orientación, suelen corresponder a giros realizados con respecto a ejes que se cortan en un punto. Estas características comunes a gran parte de los brazos robóticos simplifican los cálculos y permiten establecer ciertos procedimientos para el estudio de su cinemática inversa.

Además, en robots de seis articulaciones, es frecuente utilizar el método de desacoplamiento cinemático que divide el problema cinemático inverso en dos independientes, lo que facilita su resolución. Por un lado, se calculan las tres primeras articulaciones a partir de la posición del extremo y, por otro, las tres últimas a partir de su orientación.

Generalmente, los robots cuentan con tres grados de libertad al final de su cadena cinemática cuya finalidad es orientar el robot, si bien es cierto que también influyen en la posición final. Tal y como se ha comentado, habitualmente, los ejes de las articulaciones correspondientes a dichos grados de libertad se cortan en un punto, conocido como punto de muñeca. El método de desacoplo cinemático únicamente puede aplicarse cuando se da esta circunstancia.

Este procedimiento consiste en calcular, a partir de la posición y orientación del extremo, la posición del punto de muñeca para fijar, a partir de ella, los valores de las tres primeras articulaciones (las empleadas para la posición). Seguidamente, tomando como datos la orientación que se pretende lograr y las articulaciones previamente calculadas, se obtienen los valores de las articulaciones restantes (las correspondientes a la orientación) [1].

Los pasos que se han de seguir explican con mayor detalle a continuación:

En primer lugar, se obtiene la posición del punto de muñeca (p_m) según la expresión:

$$p_m = p_r - l_{mr} \cdot z_6$$

donde p_r es el vector con las coordenadas cartesianas (*x, y, z*) de la posición del extremo del robot, l_{mr} es la distancia entre el punto de muñeca y el extremo, y z_6 es la tercera columna de la matriz de orientación.

Tras el cálculo del punto de muñeca se hallan los valores de las tres primeras articulaciones que permiten alcanzar dicha posición.

El valor de la primera articulación (q_1) se obtiene como:

$$q_1 = \text{arctg}\left(\frac{p_y}{p_x}\right)$$

donde p_x y p_y son respectivamente las coordenadas *x* e *y* del punto de muñeca.

La expresión anterior tiene dos resultados válidos que difieren en 180°, por lo que existen dos valores de q_1 para una misma posición.

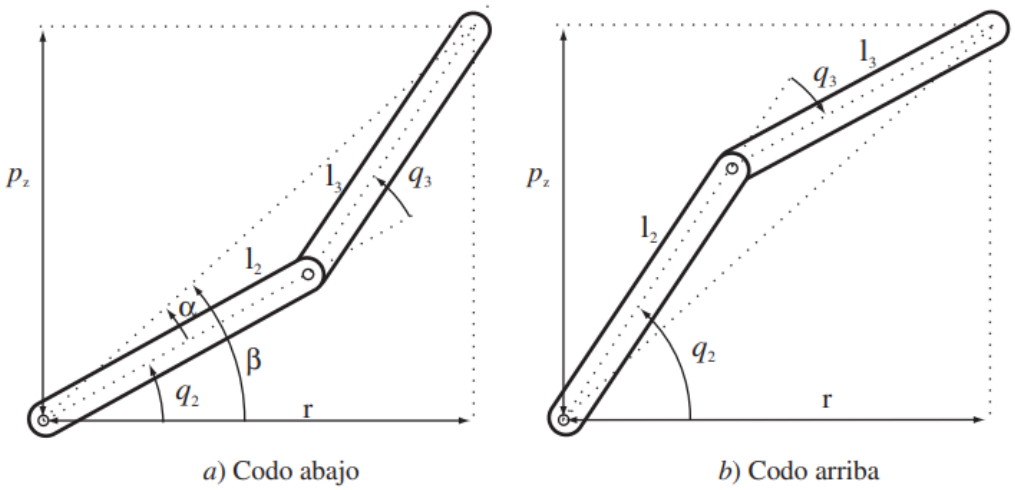


Figura 1. Posibles configuraciones del codo para el cálculo de la cinemática inversa [1]

Para calcular el valor de la segunda y la tercera articulación (q_2 y q_3) se hace uso de las variables r, α, β, l_2 y l_3 definidas en la Figura 1 donde p_z es la coordenada *z* del punto de muñeca.

Para alcanzar un mismo punto, hay dos posibilidades: una con el codo abajo y otra con el codo arriba. En ambos casos, r puede calcularse a partir de las coordenadas *x* e *y* del punto de muñeca mediante el teorema de Pitágoras.

$$r = \sqrt{p_x^2 + p_y^2}$$

Aplicando el teorema del coseno al triángulo formado por l_2, l_3 y la recta que une la segunda articulación con el final del eslabón l_3 se puede obtener q_3.

$$r^2 + p_z^2 = l_2^2 + l_3^2 + 2 \cdot l_2 \cdot l_3 \cdot \cos q_3$$

Despejando queda:

$$q_3 = \arccos\left(\frac{r^2 + p_z^2 - l_2^2 - l_3^2}{2 \cdot l_2 \cdot l_3}\right)$$

Se obtienen dos valores distintos, de igual módulo, pero signo opuesto. Uno corresponde a la configuración con el codo abajo y el otro a la del codo arriba.

Para el cálculo de q_2, se deben hallar los valores de los ángulos α y β.

$$\beta = \text{arctg}\left(\frac{p_z}{r}\right)$$

$$\alpha = \pm\text{arctg}\left(\frac{l_3 \cdot \sin q_3}{l_2 + l_3 \cdot \cos q_3}\right)$$

$$q_2 = \beta - \alpha$$

Hay dos valores posibles, según el signo de α, uno válido con el codo abajo y otro con el codo arriba.

Ahora faltan las articulaciones correspondientes a la orientación del elemento final del robot. Para ello, se parte de la matriz de transformación que relaciona la posición y orientación del extremo con el sistema de referencia fijo, ${}^0T_6$.

Se designa ${}^0R_6$ a la submatriz de rotación de la matriz de transformación ${}^0T_6$. Dicha submatriz puede expresarse como [1],[3]:

$${}^0R_6 = {}^0R_3 \cdot {}^3R_6$$

La matriz ${}^0R_6$ es conocida y ${}^0R_3$ puede obtenerse a partir de los parámetros de Denavit-Hartenberg y de las tres primeras articulaciones, con lo que se puede despejar ${}^3R_6$.

$${}^3R_6 = ({}^0R_3)^{-1} \cdot {}^0R_6$$

Se tiene

$${}^3R_4 = \begin{bmatrix} c_4 & 0 & -s_4 \\ s_4 & 0 & c_4 \\ 0 & -1 & 0 \end{bmatrix} \quad {}^4R_5 = \begin{bmatrix} c_5 & 0 & s_5 \\ s_5 & 0 & c_5 \\ 0 & 1 & 0 \end{bmatrix} \quad {}^5R_6 = \begin{bmatrix} c_6 & -s_6 & 0 \\ s_6 & c_6 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplicando queda:

$${}^3R_6 = {}^3R_4 \cdot {}^4R_5 \cdot {}^5R_6 = \begin{bmatrix} c_4 \cdot c_5 \cdot c_6 - s_4 \cdot s_6 & -c_4 \cdot c_5 \cdot s_6 - s_4 \cdot c_6 & c_4 \cdot s_5 \\ s_4 \cdot c_5 \cdot c_6 + c_4 \cdot s_6 & -s_4 \cdot c_5 \cdot s_6 + c_4 \cdot c_6 & s_4 \cdot s_5 \\ -s_5 \cdot c_6 & s_5 \cdot s_6 & c_5 \end{bmatrix}$$

Expresando la matriz como elementos indexados:

$${}^3R_6 = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Los valores de q_4, q_5 y q_6 se calculan como:

$$q_4 = \text{arctg}\left(\frac{r_{23}}{r_{13}}\right)$$

$$q_5 = \arccos(r_{33})$$

$$q_6 = \text{arctg}\left(\frac{r_{32}}{-r_{31}}\right)$$

Existen dos soluciones posibles para la combinación de estas tres articulaciones según la muñeca esté hacia arriaba o hacia abajo. Con estas dos posibilidades, junto con los dos posibles valores de q_1 y las dos posibles opciones para q_2 y q_3, es decir, codo arriba y codo abajo; se obtienen hasta ocho configuraciones distintas para una misma posición y orientación del extremo.

En el caso de que q_5 resulte ser ± 90°, entonces $c_5 = \pm 1$ y $s_5 = 0$. En tal caso, r_{13}, r_{23}, r_{31} y r_{32} son nulas, por lo que el cálculo se realiza de distinta forma.

$$ {}^3R_6(q_5 = \pm\, 90°) = \begin{bmatrix} \pm c_4 \cdot c_6 - s_4 \cdot s_6 & \mp c_4 \cdot c_5 \cdot s_6 - s_4 \cdot c_6 & 0 \\ \pm s_4 \cdot c_6 + c_4 \cdot s_6 & \mp s_4 \cdot c_5 \cdot s_6 + c_4 \cdot c_6 & 0 \\ 0 & 0 & \pm 1 \end{bmatrix} $$

Utilizando las expresiones del coseno y del seno de la suma de dos ángulos puede expresarse la matriz como:

$$ {}^3R_6(q_5 = \pm\, 90°) = \begin{bmatrix} c_{4\pm6} & -s_{4\pm6} & 0 \\ s_{4\pm6} & c_{4\pm6} & 0 \\ 0 & 0 & \pm 1 \end{bmatrix} $$

Puede obtenerse el valor de la suma de q_4 y q_6 según la expresión:

$$ q_4 \pm q_6 = \operatorname{arctg}\left(\frac{r_{21}}{r_{11}}\right) $$

En este caso, existiría un número infinito de soluciones para q_4 y q_6, cuya suma debe cumplir la igualdad anterior.

3.3. Espacio de configuraciones

Se conoce como espacio de configuraciones, designado por *C*, al conjunto de los posibles valores que las articulaciones de un robot pueden tomar [4]. Se trata de otra manera de afrontar el estudio de los sistemas robóticos que complementa a la consideración de la posición y orientación en coordenadas cartesianas de los sistemas de referencia fijados en el extremo del robot y en cada uno de sus eslabones [5].

El número de coordenadas generalizadas independientes se conoce como número de grados de libertad. Para un robot con N grados de libertad, cualquier configuración posible vendrá dada por un punto de N dimensiones. Es decir, la dimensión de *C* es N [5].

Una articulación de revolución implica un espacio de configuraciones $\mathbb{S}^1$ (esfera de dimensión 1), mientras que una articulación prismática implica un espacio de configuraciones $\mathbb{R}^1$ (euclídeo de dimensión 1). La combinación de varias articulaciones varía las dimensiones del espacio añadiendo una esférica si es de revolución y euclídea si es prismática. Así, un robot con tres articulaciones de revolución tendría un espacio de configuraciones $\mathbb{S}^3$, y en un robot de tipo SCARA (con tres articulaciones de revolución y una prismática), sería $\mathbb{S}^3\mathrm{x}\mathbb{R}^1$.

El espacio de configuraciones se divide en dos subespacios: el espacio libre de colisiones (C_{free}), en el que no se produce ningún choque, y el espacio de colisiones, (C_{obs}), en el que el robot se halla en colisión, con un obstáculo o consigo mismo [4]. Por este motivo, el espacio de configuraciones es ampliamente utilizado en la planificación de trayectorias, pues la clasificación de las configuraciones y de los trayectos que las unen en estos dos subespacios permite determinar si se produce o no colisión y evitar por tanto los obstáculos.

Por otra parte, se define como espacio de trabajo al conjunto de poses, es decir, posiciones y orientaciones, que el efector final del robot puede tomar. Su forma depende de la geometría del robot, de sus articulaciones y también de la herramienta que se emplee. Puede ser de tipo euclídeo, si el efector final solo se desplaza manteniendo fija su orientación, o una combinación de un espacio esférico y euclídeo si puede orientarse además de desplazarse. Así pues, el espacio de trabajo de un brazo robótico de seis grados de libertad sería $\mathbb{R}^3\mathrm{x}\mathbb{S}^3$, ya que puede desplazarse en los tres ejes cartesianos y orientarse también con respecto a estos los ejes. En el caso de un robot SCARA, sería $\mathbb{R}^3\mathrm{x}\mathbb{S}^1$, ya que permite la traslación en los tres ejes, pero solo el cambio de orientación con respecto a uno y, en el de un robot cartesiano, que puede desplazarse en los tres ejes, pero no orientar la herramienta; simplemente $\mathbb{R}^3$.

Finalmente, el espacio de tarea es el subespacio perteneciente al espacio de trabajo formado por el conjunto de poses que el extremo del robot debe alcanzar para desempeñar una actividad concreta. Por tanto, la dimensión del espacio de tarea debe ser menor o igual que la dimensión del espacio de trabajo, y también menor o igual que la del espacio de configuraciones. Dicho de otro modo, toda pose del espacio de tarea puede traducirse a una configuración del espacio de configuraciones, lo que no necesariamente ocurre a la inversa [5]. Por ejemplo, si se tiene un brazo robótico de seis grados de libertad y se pretende llevar el extremo a un punto sin importar su orientación, el espacio de tarea sería $\mathbb{R}^3$, cuya dimensión es 3; mientras que el espacio de trabajo sería $\mathbb{R}^3\mathrm{x}\mathbb{S}^3$, y el de configuraciones, $\mathbb{S}^6$, ambos de dimensión 6.

En aquellos casos en los que los grados de libertad del robot son mayores que las restricciones impuestas por la tarea que se pretende desarrollar, el robot tiene un número de grados de libertad de redundancia igual a la diferencia entre ambos parámetros. Esto le permite alcanzar el objetivo con distintas configuraciones [6].

3.4. Algoritmos de planificación de trayectorias

El problema de la planificación de rutas trata de hallar un camino libre de colisiones entre una configuración inicial y otra final en un entorno determinado. La situación más simple se da cuando debe planificarse el movimiento en un entorno conocido y estático, aunque pueden buscarse rutas en cualquier entorno, incluso si es dinámico y desconocido [7].

Existen multitud de algoritmos para la planificación de trayectorias. A continuación, se hace una breve explicación de algunos de ellos.

Gráfico de visibilidad: Sus nodos son los vértices de los obstáculos, que se unen entre ellos siempre que dicha unión se encuentre en el espacio libre de configuraciones. Se añaden los nodos inicial y final, que se unen a todos los nodos posibles sin entrar en el espacio de colisiones. Para acabar, se busca el camino óptimo en el gráfico para llegar al nodo final desde el inicial [7]. En la Figura 2 se puede ver un gráfico de visibilidad en un espacio de configuraciones de dos dimensiones.

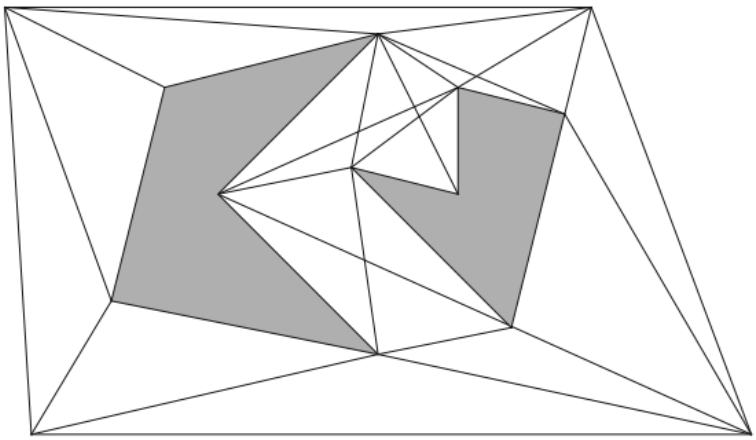


Figura 2. Ejemplo de gráfico de visibilidad [7]

Este método presenta la ventaja de encontrar una ruta óptima, pero tiene el inconveniente de ser muy costoso en términos computacionales en entornos complejos o de un alto número de dimensiones, ya que modelar los obstáculos para sistemas con más de tres grados de libertad no es sencillo.

Voronoi: Crea una serie de líneas a la misma distancia de los obstáculos en el espacio de configuraciones y une los nodos inicial y final a la línea más cercana. Finalmente, solo hay que hallar el camino que, siguiendo las líneas dibujadas, une los puntos inicial y final [8]. En la Figura 3 se observa un ejemplo de gráfico de Voronoi.

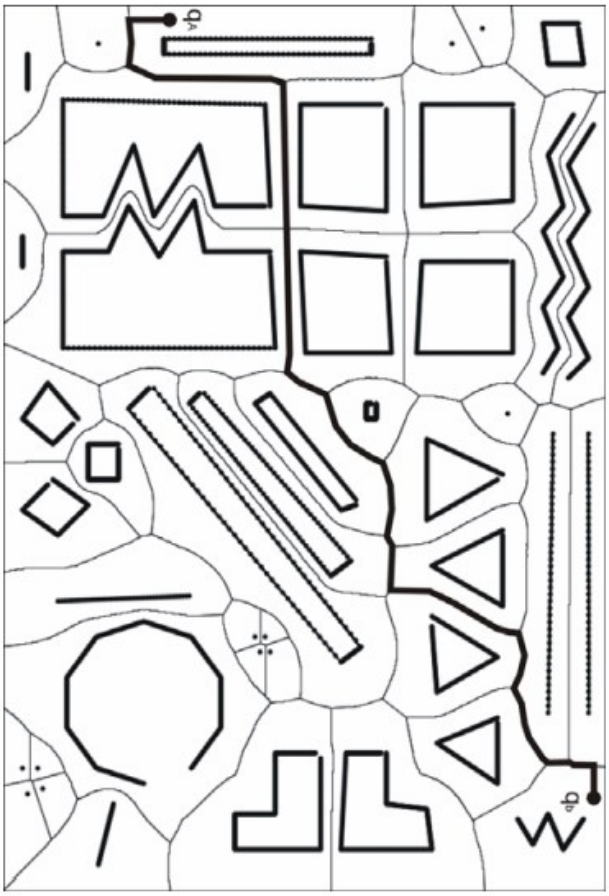


Figura 3. Ejemplo de mapa de Voronoi [9]

Entre sus ventajas destaca su mapeo automático y su facilidad de ejecución para sistemas con pocos grados de libertad. Tiene las desventajas de aumentar su complejidad en robots de un elevado número de grados de libertad y de dar lugar a rutas no óptimas.

Método de descomposición de celdas: El espacio de configuraciones se divide en múltiples regiones denominadas celdas de tal forma que la ruta entre las configuraciones correspondientes a dos celdas adyacentes pueda generarse fácilmente. Las celdas se clasifican según contengan o no un obstáculo. Las celdas parcialmente ocupadas se marcan igual que si lo estuvieran por completo [9]. A cada celda se le asigna un coste según su distancia al inicio. El coste puede aumentar más o menos según si el desplazamiento es vertical u horizontal, o si es diagonal. Finalmente, se busca la ruta óptima mediante algoritmos como Dijkstra o A*. En la Figura 4 se aprecia un ejemplo de esta técnica.

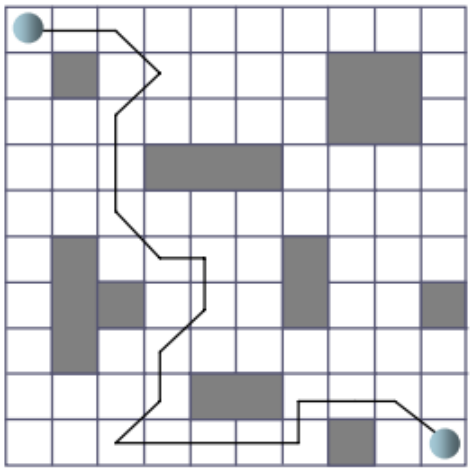


Figura 4. Ejemplo de mapa de celdas [9]

Es una forma relativamente sencilla de generar trayectorias en dos dimensiones, pero no es aplicable para espacios de configuraciones de más de tres dimensiones. Además, el hecho de que las celdas ocupadas solo en parte se consideren enteramente ocupadas puede dar lugar a que se pierdan posibles rutas.

Método de potencial artificial: Se basa en la idea de considerar al robot como una partícula sujeta a las fuerzas de un campo potencial generado por el nodo de destino y por los obstáculos. El potencial creado por la meta es de atracción y el de los obstáculos de repulsión [7]. Mediante el cálculo de estos potenciales opuestos, se guía al robot por el espacio de configuraciones hacia el nodo final evitando los obstáculos [8]. De esta manera, para una configuración dada, la siguiente viene determinada por la dirección de la fuerza artificial resultante a la que se está sometiendo al robot [7]. La Figura 5 muestra el concepto en el que se basa este algoritmo.

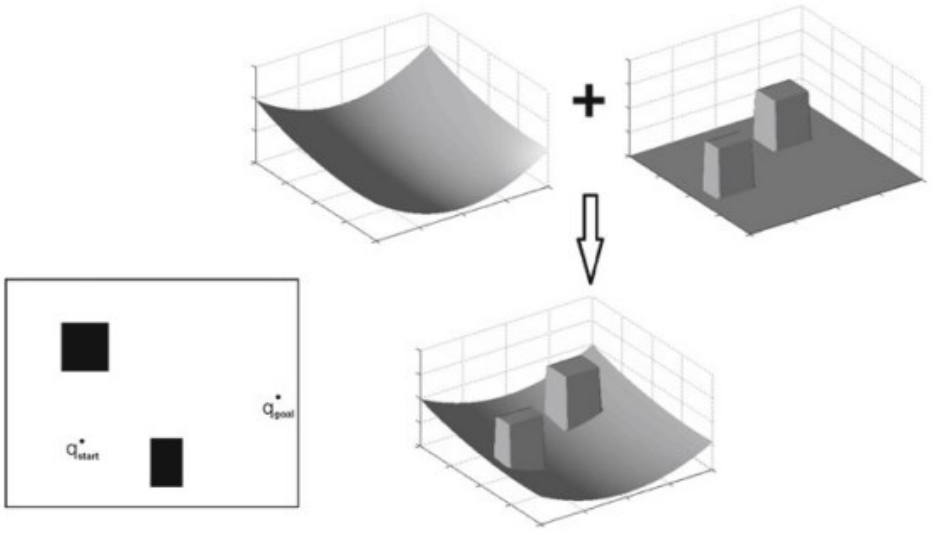


Figura 5. Ejemplo de potencial artificial [7]

Presenta el inconveniente de que los campos de potencial pueden tener mínimos locales, esto es, configuraciones en las que se produce un equilibrio entre fuerzas, lo que provoca que el algoritmo se bloquee.

PRM: Crea una serie de nodos aleatorios que conecta con los *k* vecinos más cercanos siempre que puedan unirse sin dar pie a una colisión. Se genera así un mapa de nodos unidos por rectas al que se añaden los nodos inicial y final. A continuación, se busca el camino óptimo para alcanzar el objetivo utilizando métodos como Dijkstra o A* [8]. En la Figura 6 se puede observar el funcionamiento básico de esta técnica.

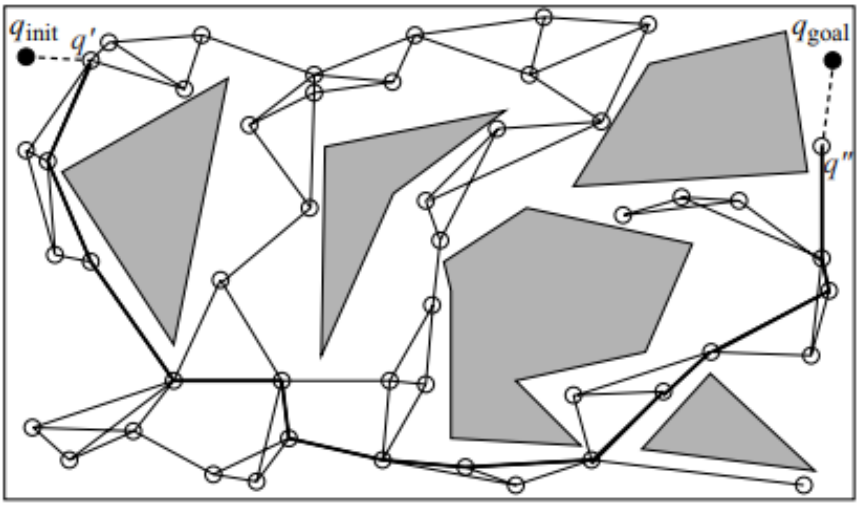


Figura 6. Ejemplo de PRM [10]

Presenta la ventaja de poder aplicarse a robots con un alto número de grados de libertad y de devolver la ruta con menor coste posible con los nodos obtenidos. Es posible, por otra parte, que, en una ejecución del algoritmo, los nodos generados no puedan unirse de forma que se cree una ruta entre los nodos inicial y final. Sin embargo, en tal caso, pueden generarse nuevos puntos hasta que se encuentre una trayectoria.

RRT: Crea un árbol en el espacio de configuraciones, partiendo del nodo inicial, al que añade nuevos nodos aleatorios, uniéndolos mediante rectas en el espacio libre de configuraciones con el nodo más cercano del árbol hasta alcanzar el nodo final. Existen variaciones de este algoritmo tales como RRT* (RRT estrella) que optimizan el camino encontrado [8]. Sirva la Figura 7 como ayuda visual para la comprensión de este procedimiento.

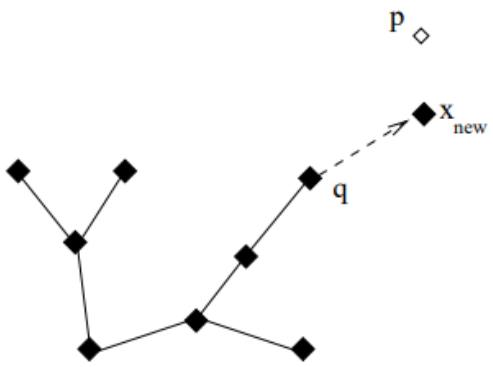


Figura 7. Ejemplo de RRT [11]

Se trata de un método más rápido de PRM y también aplicable a robots con más de tres grados de libertad. Presenta el inconveniente de no dar lugar a trayectorias óptimas, aunque existen variaciones como RRT* que permiten mejorar el coste de la ruta.

Este proyecto se centra en los algoritmos PRM y RRT por ser adecuados para robots con más de tres grados de libertad. A continuación, se realiza una explicación mucho más detallada de estas dos técnicas.

3.4.1. Probabilistic Roadmap (PRM)

La planificación de trayectorias por medio de este algoritmo se divide en dos fases: una de aprendizaje y otra de consulta. En la primera fase, se elabora el mapa, que está formado por configuraciones en C_{free}, representadas por nodos en el espacio de configuraciones, y por rectas, también en C_{free}, que los unen con los *k* vecinos más cercanos, cuando tales rectas existen. En la segunda, se añaden las configuraciones inicial y final ($q_{inicial}$ y q_{final}) y se conectan al resto de nodos mediante rectas en C_{free}. A continuación, se busca la ruta óptima del mapa para ir de $q_{inicial}$ a q_{final} [10].

El algoritmo empleado para la fase de aprendizaje es el siguiente [10]:

Algoritmo para la construcción del mapa

Entradas: N (número de nodos) y k (número de vecinos más próximos que se comprueban en cada nodo).

Salida: Un mapa M = (V, R) donde V son los nodos y R las rectas que los unen.

1: V = Ø

2: R = Ø

```
repetir mientras tamaño(V) < N
    repetir
        q = configuración aleatoria en C
    hasta que q esté en C_free
    añadir q a V
fin del bucle
repetir para todos los valores de q ∈ V
    N_q = k vecinos más próximos q
    repetir para todos los valores de q' ∈ N_q
        si la recta entre q y q' no está contenida en R y está en C_free
            añadir la recta entre q y q' a R
        fin de la condición
    fin del bucle
fin del bucle
```

Al inicio, *V* y *R*, que representan los nodos y rectas del mapa, están vacíos. Se crean configuraciones aleatorias que están en C_{free}, para lo cual se repite la generación de la configuración si no lo está hasta que así sea y se añade el nuevo nodo a *V*. Esta operación se repite un número *N* de veces, siendo *N* el número de nodos que se desea crear.

Seguidamente, para cada nodo *q* perteneciente a *V* se buscan los *k* nodos más cercanos y, para cada uno de ellos, se calcula la recta que lo une con *q*. Se comprueba, en cada una de las rectas, si está enteramente en C_{free} y, en caso afirmativo, se añade a *R*.

En la fase de consulta, se añaden las configuraciones inicial y final ($q_{inicial}$ y q_{final}) y se conectan al mapa generado mediante el algoritmo anterior. Para ello se pueden unir los dos nuevos nodos a los *k* más próximos mediante rectas en C_{free}, si tales rectas existen. Otra forma, la empleada en el algoritmo que se explica más adelante, consiste en ordenar los *k* vecinos más cercanos de menor a mayor distancia y tratar de realizar la unión entre el nodo inicial o final y los vecinos por orden creciente de lejanía hasta que se encuentre la primera unión posible en C_{free}. El valor del parámetro *k* puede coincidir o no con el utilizado para la elaboración del mapa. Finalmente, se calcula el camino óptimo teniendo en cuenta las distancias de las rectas entre nodos del mapa. Para esto último se utilizan algoritmos como Dijkstra o A*.

En pseudocódigo, el algoritmo para esta segunda fase es el que sigue [10]:

Algoritmo para la fase de consulta

Entradas: $q_{inicial}$ (configuración inicial), q_{final} (configuración final), k (número de vecinos más próximos para comprobar) y M = (V, R) (el mapa creado en la fase de aprendizaje)

Salida: Un camino entre $q_{inicial}$ y q_{final} o fallo.

1: $N_{q_{inicial}}$ = k vecinos más cercanos a $q_{inicial}$ pertenecientes a V

2: $N_{q_{final}}$ = k vecinos más cercanos a q_{final} pertenecientes a V

3: V = ($q_{inicial}$) U (q_{final}) U V

4: q' = nodo más cercano a $q_{inicial}$ en $N_{q_{inicial}}$

5: repetir

6: si la recta entre $q_{inicial}$ y q' está en C_{free}

7: añadir la recta entre $q_{inicial}$ y q' a R

8: en caso contrario

9: q' = siguiente nodo más cercano a $q_{inicial}$ en $N_{q_{inicial}}$

10: fin de la condición

11: hasta que se complete una conexión o $N_{q_{inicial}}$ se haya recorrido por completo

12: q' = nodo más cercano a q_{final} en $N_{q_{final}}$

13: repetir

14: si la recta entre q_{final} y q' está en C_{free}

15: añadir la recta entre q_{final} y q' a R

16: en caso contrario

17: q' = siguiente nodo más cercano a q_{final} en $N_{q_{final}}$

18: fin de la condición

19: hasta que se complete una conexión o $N_{q_{final}}$ se haya recorrido por completo

20: P = camino más corto ($q_{inicial}$, q_{final}, V)

21: si P no está vacío

22: devolver P

23: en caso contrario

24: devolver fallo

25: fin de la condición

Tal y como se ha comentado, se añaden los nodos inicial y final al mapa y se unen con el vecino más cercano posible sin colisión. Si se encuentra un camino lo devuelve y si no devuelve un mensaje de error.

En la Figura 8 se puede observar un ejemplo en un plano de dos dimensiones del funcionamiento del método PRM. Las configuraciones en C_{free} se representan en color verde, y en rojo, las que se hallan en colisión con un obstáculo. Se crea un mapa a partir de nodos aleatorios y se unen mediante rectas los *k* vecinos más próximos a cada uno siempre y cuando la recta no intersecte con ningún obstáculo. Seguidamente, se añaden los nodos inicial y final y se busca el camino más corto entre ambos. En el caso de la figura, los nodos inicial y final se han unido a los *k* vecinos más próximos y no solo al más cercano. Por otra parte, se han descartado los nodos en colisión en lugar de generar nuevos nodos libres en su lugar, para apreciar la clasificación entre nodos en C_{free} y C_{obs}.

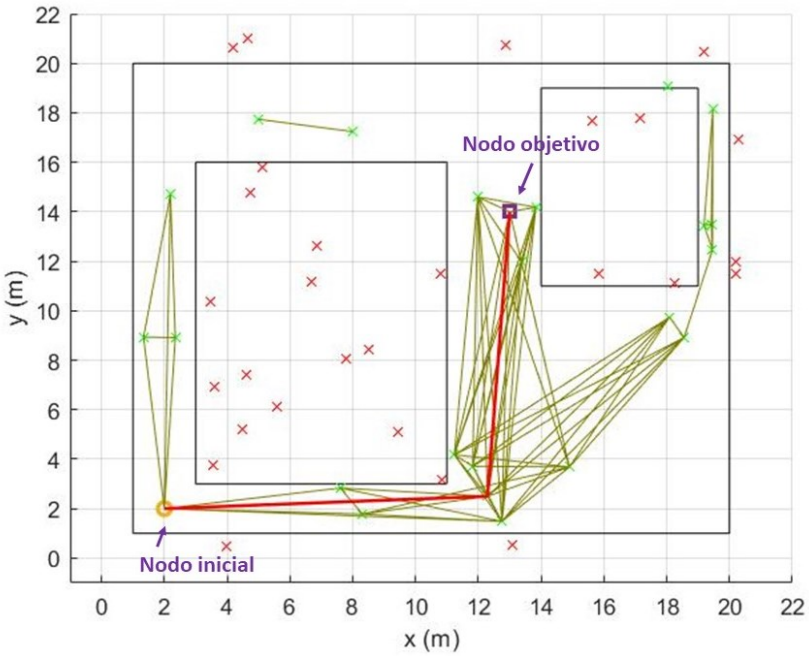


Figura 8. Trayectoria generada mediante PRM en un entorno de dos dimensiones

Si el algoritmo devuelve fallo, se pueden suprimir las dos primeras líneas del algoritmo de la fase de aprendizaje y volver a ejecutar ambas fases para así aumentar el número de nodos del mapa hasta encontrar un camino.

Cabe destacar que, al añadirse en la segunda fase los nodos inicial y de destino, estos pueden variarse manteniendo los puntos y rectas creados en la primera fase. Ello supone una importante ventaja por el ahorro de tiempo y de carga computacional que conlleva.

Tal y como ya se ha comentado, los nodos se unen en el espacio de configuraciones mediante una línea recta y se comprueba que en todo el segmento no se produce ninguna colisión. Para ello, existen dos algoritmos ampliamente empleados: el de verificación de colisión incremental y el de subdivisión. En los dos casos, el segmento que une los nodos se discretiza en una serie de configuraciones. La distancia entre ellas debe ser lo suficientemente pequeña como para garantizar que todas las posibles colisiones sean detectadas. La diferencia entre ambos métodos radica en el orden en el que las configuraciones obtenidas al discretizar la recta se evalúan. Suponiendo el segmento que une los nodos q' y q'', con el algoritmo incremental se parte de q' y se va avanzando la distancia establecida en la discretización a lo largo de la recta hasta alcanzar q'' o detectar alguna colisión. En el caso del método de subdivisión, se comprueba en primer lugar el punto medio del segmento (q_m), equidistante de q' y de q''. Después se vuelve a llamar a la función, pero esta vez comprobando los segmentos entre q' y q_m por un lado y entre q_m y q'' por otro, y así sucesivamente hasta que la distancia de los nuevos segmentos sea menor a la distancia elegida para la discretización o se encuentre algún choque con un obstáculo.

Desde el punto de vista teórico ninguno de estos dos métodos es más ventajoso que el otro. Sin embargo, en la práctica, el algoritmo de subdivisión tiende a funcionar mejor, puesto que los segmentos más pequeños tienen mayor probabilidad de estar libres de colisiones [10].

En las Figuras 9 y 10 se puede apreciar de forma gráfica el funcionamiento de los procedimientos previamente expuestos.

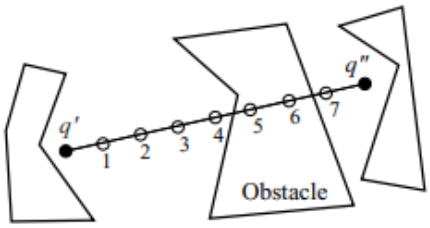


Figura 9. Muestreo de la recta entre dos nodos mediante el algoritmo de verificación de colisión incremental [10]

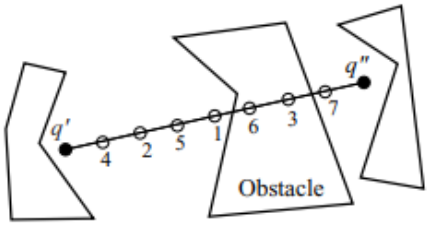


Figura 10. Muestreo de la recta entre dos nodos mediante el algoritmo de subdivisión [10]

Para hallar la ruta óptima se usan, como se ha mencionado anteriormente, algoritmos tales como A* o Dijkstra, que se detallan a continuación.

El algoritmo A* es un método de tipo informado, ya que, a diferencia de los no informados, que se mueven por el gráfico sin ninguna preferencia, trata de buscar el camino óptimo hasta el nodo objetivo basándose en una heurística elegida, como, por ejemplo, cuál de los nodos vecinos tiene una menor distancia euclídea al nodo final, ya que tiene más probabilidades de acercarse más a la meta [10].

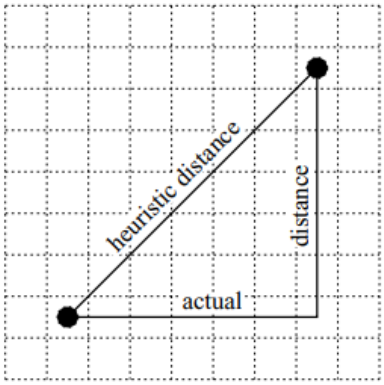


Figura 11. Distancia real y heurística entre dos nodos [10]

Tal y como se aprecia en la Figura 11, el coste heurístico siempre es menor o igual que el coste real para llegar de un nodo al siguiente en el mapa.

Se utilizan dos listas de nodos: abierta y cerrada. En cada iteración, se selecciona de la lista abierta aquel nodo cuyo coste de evaluación sea el mínimo y este pasa a la lista cerrada. Después, se añaden a la lista abierta los nodos vecinos del que se acaba de incorporar a la cerrada y, si ya estaban en ella, se actualizan los valores de los costes operativos de los nodos en el caso de que se reduzcan. El coste de evaluación se obtiene como la suma del coste operativo, definido como la distancia real para alcanzar el nodo desde el inicio, y el coste heurístico, que corresponde al estimado mediante la distancia euclídea al nodo de destino; tal y como expresa la igualdad:

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

donde $f(n)$ representa el coste de evaluación del nodo n, $g(n)$, su coste de operación y $h(n)$, su coste heurístico.

Se comienza añadiendo el nodo de inicio a la lista cerrada y todos los nodos adyacentes al primero a la lista abierta. Se calcula el coste de evaluación de los nodos de la lista abierta y se selecciona aquel cuyo coste sea menor. A continuación, el nodo elegido pasa a la lista cerrada y se añaden a la abierta los nodos conectados a él, junto con sus costes y se actualizan los de los nodos que ya figuraban en la lista. Seguidamente, se vuelve a elegir el de menor coste y se repite

todo el proceso hasta que se obtenga que el nodo de menos coste sea el de destino, momento en el que se alcanza la meta.

Una vez alcanzado el objetivo, tan solo hay que recorrer el camino hacia atrás desde el nodo de destino hasta el inicial. Para ello, es fundamental guardar las relaciones de precedencia entre los nodos seleccionados en cada iteración.

El algoritmo, en pseudocódigo, se muestra a continuación [10]:

Algoritmo A*

Entradas: Mapa de nodos

Salida: Una ruta entre los nodos de inicio y de destino

1: Crear dos listas vacías: Abiertos y Cerrados

2: Añadir el nodo de inicio a Cerrados

3: repetir

4: Elegir $n_{óptimo}$ de Abiertos tal que $f(n_{óptimo}) \leq f(n)\ \forall\ n\ \in$ Abiertos

5: Eliminar $n_{óptimo}$ de Abiertos y añadirlo a Cerrados

6: si $n_{óptimo} = n_{destino}$, salir

7: x=nodos adyacentes a $n_{óptimo}$ que no estén contenidos en Cerrados

8: repetir para todos los nodos $x_i\ \in$ x

9: si $x_i \notin$ Abiertos

10: añadir x_i a Abiertos

11: en caso contrario y si $g(n_{óptimo}) + c(n_{óptimo}, x_i) < g(x_i)$

12: Guardar $n_{óptimo}$ como el predecesor de x_i

13: fin de la condición

14: fin del bucle

15: hasta que Abiertos esté vacío

Como muestra el pseudocódigo anterior, tras crear las listas vacías y añadir a Cerrados el nodo inicial comienza el bucle, en el que se elige el nodo de menos coste de Abiertos y se pasa a Cerrados. El programa finaliza si el nodo seleccionado es el de destino. Mientras no sea así, se selecciona un nuevo nodo de entre los adyacentes al último que no estén en Cerrados. Estos se

añaden a Abiertos y, si ya estaban y el coste operativo hasta el inicio que tienen asociado es mayor que la suma del coste operativo del nodo previamente seleccionado y la distancia entre ambos nodos, se actualiza el nuevo coste y también el predecesor de dicho nodo x_i, que pasa a ser $n_{óptimo}$. En el pseudocódigo mostrado, $c(n_{óptimo}, x_i)$ representa el coste operativo de unir $n_{óptimo}$ con x_i.

El algoritmo Dijkstra es similar al A*, pero ahora se trata de un procedimiento no informado, por lo que no emplea ninguna heurística. En este caso, al evaluar el coste de la unión de dos nodos, tan solo se tiene en cuenta el coste operativo, y no se estima el coste para llegar al destino desde cada nodo seleccionado. Salvando esta discrepancia, su funcionamiento es análogo al explicado para A*.

3.4.2. Rapidly-exploring Random Tree (RRT)

Este algoritmo se basa en la creación de un árbol formado por nodos en C_{free} unidos por segmentos de una longitud previamente establecida. Dados un nodo inicial (q_{init}) y otro de destino (q_{goal}), el árbol parte desde el inicio y se va expandiendo hasta alcanzar el objetivo. En este trabajo se trabaja con este algoritmo tanto en su forma básica como en una de sus variantes, conocida como RRT* (RRT estrella).

3.4.2.1. RRT básico

Para construir el árbol, se crea una configuración aleatoria perteneciente a C_{free} (q_{rand}). A continuación, se busca la configuración contenida en el árbol más cercana a q_{rand} (q_{near}). Posteriormente, se alarga el árbol una distancia determinada a lo largo de la línea recta que une q_{rand} y q_{near}, donde se crea el nodo q_{new}. Esta última configuración se añade al árbol siempre y cuando su unión con q_{near} se encuentre libre de colisiones [10].

En cada iteración, se comprueba si la distancia entre el nodo generado (q_{new}) y el nodo de destino (q_{goal}) es menor o igual que la distancia establecida para las uniones entre nodos, en cuyo caso se unen q_{new} y q_{goal} y finaliza el proceso. Llegados a este punto, tan solo restaría recorrer el árbol generado en sentido inverso, desde la meta hasta el punto de partida. Para ello, es preciso guardar la relación de precedencia entre los nodos. En esta relación, al nodo predecesor se le suele conocer como nodo padre (en inglés *parent-node*) y al sucesor como nodo hijo (*child-node*) [12].

En la Figura 12 se expone un ejemplo en un plano bidimensional del funcionamiento del método RRT.

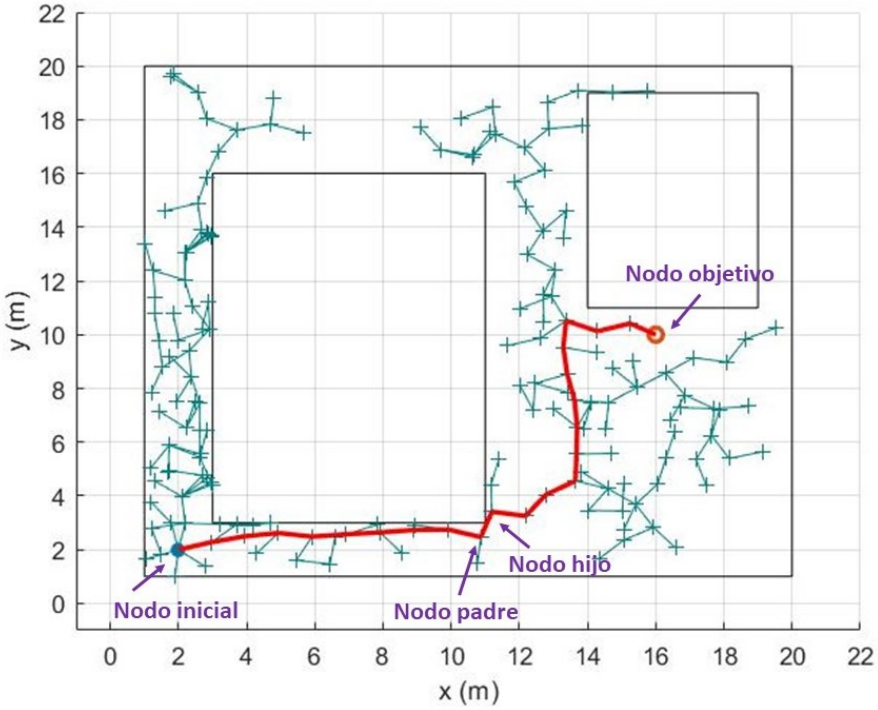


Figura 12. Trayectoria generada mediante RRT básico en un entorno de dos dimensiones

Tal y como puede apreciarse en la Figura 12, el árbol generado parte de un nodo inicial y se va ramificando evitando los obstáculos hasta alcanzar el nodo objetivo. Se observa también la relación de precedencia entre nodos padres e hijos. En la figura, las unidades se expresan en metros, pues en este caso se ha aplicado el procedimiento a un robot móvil. Para su aplicación en un robot manipulador, las unidades serían grados o radianes en lugar de metros y el espacio no sería el cartesiano sino el de configuraciones de las articulaciones.

El procedimiento, expresado en pseudocódigo, se muestra a continuación [13]:

Algoritmo RRT

Entradas: q_{init} (configuración inicial), q_{goal} (configuración de destino), d (longitud del segmento que une los nodos), k (número máximo de iteraciones)

Salida: T (árbol de configuraciones) o fallo

1: T=árbol (q_{init})

2: desde i=1 hasta i=k

3: q_{rand} = configuración aleatoria en C_{free}

4: q_{near} = vecino más cercano a q_{rand} perteneciente a T

5: q_{new} = nodo en la recta entre q_{rand} y q_{near} a una distancia d de q_{near}

6: si el segmento entre q_{near} y q_{new} está libre de colisiones

7: añadir q_{new} y el segmento entre q_{near} y q_{new} a T

8: si la distancia entre q_{new} y q_{goal} es menor que d

9: si el segmento entre q_{new} y q_{goal} está libre de colisiones

10: devolver T

11: salir

12: fin de la condición

13: fin de la condición

14: fin de la condición

15: fin del bucle

16: devolver fallo si no se ha hallado un camino

Tal y como se ha explicado y como se puede seguir en el pseudocódigo anterior, primeramente, se añade el nodo de partida al árbol y, acto seguido, se entra en un bucle que se repetirá como máximo un número k de veces. Se crea una configuración aleatoria en el espacio libre de colisiones, se busca su vecino más cercano en el árbol y se extiende el árbol, en la dirección de la recta que las une, la distancia seleccionada siempre que el segmento de unión esté también libre de impacto. Si el nodo final está lo suficientemente cerca, se une al último nodo creado si esto es posible sin provocar una colisión, se devuelve el árbol y concluye la función. En caso contrario se repite el proceso. Si la función llega a k iteraciones sin alcanzar el objetivo devuelve un mensaje de error.

3.4.2.2. RRT*

El algoritmo RRT* (RRT estrella) es una versión mejorada de RRT. El método RRT básico extiende el árbol de forma aleatoria uniendo el nuevo nodo al más cercano de los ya contenidos en el árbol hasta alcanzar un punto lo suficientemente cerca del nodo de destino como para poder unirlo a él. En cambio, RRT*, busca, de entre los nodos del árbol cercanos al punto aleatorio creado, el que menor coste tiene desde el nodo inicial para mejorar la ruta generada en términos de coste [10]. De este modo, mientras que las trayectorias generadas mediante RRT básico suelen zigzaguear, las obtenidas a través de RRT* se acercan más a la ruta óptima.

La creación de los nuevos nodos es análoga a la utilizada en RRT básico, esto es, se genera una configuración aleatoria, y, en la recta entre dicha configuración y el nodo más cercano se crea

un nodo a una cierta distancia establecida. La diferencia radica en que ahora el último nodo no necesariamente se une al más cercano, sino que se ordenan los puntos del árbol que se hallan en un cierto radio de la nueva configuración de menor a mayor coste desde el inicio. El nuevo nodo se une al de menor coste si dicha unión se haya en C_{free}.

A continuación, se muestra el algoritmo en pseudocódigo:

Algoritmo RRT*

Entradas: q_{init} (configuración inicial), q_{goal} (configuración de destino), d (longitud del segmento que une los nodos), k (número máximo de iteraciones), radio (radio en el que se buscan nodos para unir el nuevo al árbol)

Salida: T (árbol de configuraciones) o fallo

1: T=árbol (q_{init})

2: desde i=1 hasta i=k

3: q_{rand} = configuración aleatoria en C_{free}

4: q_{near} = vecino más cercano a q_{rand} perteneciente a T

5: q_{new} = nodo en la recta entre q_{rand} y q_{near} a una distancia d de q_{near}

6: extender árbol (T, q_{new}, radio)

7: si la distancia entre q_{new} y q_{goal} es menor que d

8: si el segmento entre q_{new} y q_{goal} está libre de colisiones

9: devolver T

10: salir

11: fin de la condición

12: fin de la condición

13: fin del bucle

14: devolver fallo

El procedimiento anterior es igual al de RRT, la diferencia está en la función que extiende el árbol, cuyo funcionamiento se detalla en el pseudocódigo que sigue [14]:

Función extender árbol para RRT*

Entradas: T (árbol de configuraciones), q_{new} (nueva configuración para unir al árbol), radio (radio en el que se buscan nodos para unir el nuevo al árbol)

Salida: T (árbol de configuraciones)

1: Q_{near} = nodos cercanos a q_{new} una distancia menor que radio

2: θ_{near} = elementos contenidos en Q_{near} ordenados de menos a más coste desde el inicio

3: desde i=1 hasta i=tamaño(θ_{near})

4: si la trayectoria entre q_{new} y $\theta_{near}(i)$ está en C_{free}

5: calcular coste entre q_{new} y $\theta_{near}(i)$

6: añadir q_{new} a T

7: añadir la recta entre q_{new} y $\theta_{near}(i)$ a T

8: salir del bucle

9: fin de la condición

10: fin del bucle

11: devolver T

En el pseudocódigo se observa cómo se crea un vector que contiene los nodos del árbol cuya distancia hasta el nuevo nodo que se desea añadir es menor que el radio seleccionado. Dichos nodos se ordenan según su coste desde el nodo de partida y se almacenan en otro vector. Este último se recorre hasta encontrar el primero que se pueda unir sin provocar una colisión.

La Figura 13 muestra un ejemplo en un plano de dos dimensiones del método descrito. En la imagen se observa cómo los nodos se unen al árbol, no mediante el vecino más cercano, sino mediante aquel, de entre los cercanos en un cierto radio, que se alcanza con menor coste desde el inicio. Es por ello, que el camino se aproxima más al óptimo que con el método RRT básico.

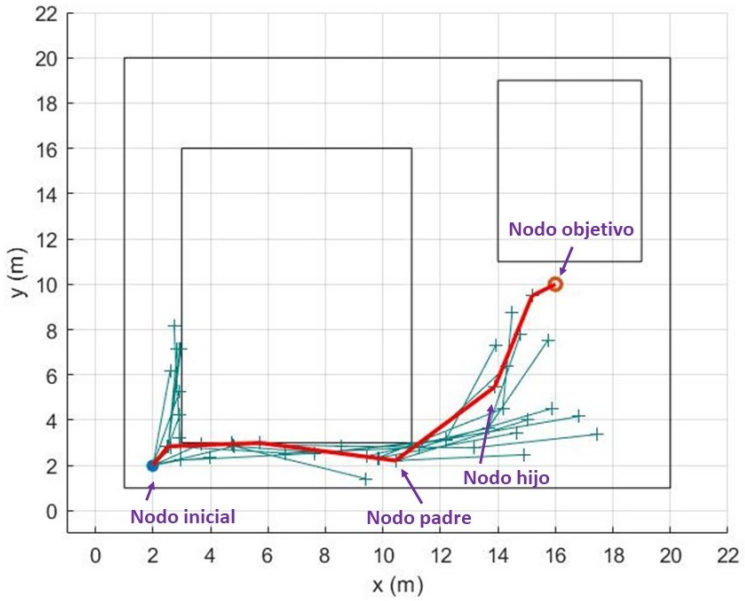


Figura 13. Trayectoria generada mediante RRT en un entorno de dos dimensiones*

.

3.5. Herramientas de software

Para la realización de este proyecto se ha utilizado Matlab, una herramienta ampliamente utilizada en control y en robótica debido a la gran cantidad de funciones de que dispone, la facilidad que ofrece para trabajar con matrices y vectores, así como con funciones de transferencia, o a la existencia de numerosas *toolboxes* y aplicaciones profesionales, entre otras razones.

Además de las herramientas básicas de Matlab se han utilizado otras dos por considerarse sus funcionalidades de gran ayuda para el desarrollo de este trabajo. La primera de ellas es Simscape Multibody, que permite importar y simular sistemas mecánicos. La segunda es la *toolbox* MPT (Multi-Parametric Toolbox), que permite trabajar con poliedros y politopos convexos.

3.5.1. Simscape Multibody

Se trata de una herramienta de Matlab, en versiones anteriores llamada SimMechanics, que ofrece un entorno de simulación para sistemas mecánicos en tres dimensiones. Es muy ampliamente utilizada para simular robots, sistemas de suspensión de vehículos, maquinaria, etc.

Se pueden modelar sistemas formados por distintos elementos a partir de bloques que pueden representar cuerpos, articulaciones de diferentes tipos, fuerzas, restricciones, sensores, etc. Estos elementos pueden ser importados desde modelos CAD, incluyendo, además de las características geométricas de los elementos mecánicos, masas, inercias, articulaciones y restricciones. Es compatible con Solidworks, lo que posibilita la importación de modelos diseñados mediante este software, exportándolos previamente como archivos XML. Permite también el uso de matrices de transformación para expresar las relaciones de posición y orientación de unos elementos con respecto a otros.

Simscape Multibody puede utilizarse de forma conjunta con Matlab y Simulink, lo que permite utilizar variables y cálculos de Matlab, así como implementar sistemas de control en Simulink que regulen el modelo del simulador [15].

3.5.2. Multi-Parametric Toolbox

Es una herramienta de código abierto basada en Matlab que permite crear polígonos, poliedros y politopos convexos y realizar distintas operaciones con ellos. Un polígono puede definirse como una figura plana delimitada por tres o más rectas y que presenta tres o más vértices y ángulos. Por otro lado, un poliedro es el volumen geométrico delimitado por varios polígonos denominados caras. De este modo, un polígono es bidimensional y un poliedro tridimensional.

El término politopo es una generalización para referirse a regiones geométricas n-dimensionales.

Un politopo es la envoltura convexa de un cierto número de puntos definidos en un espacio euclídeo. Un conjunto Y es convexo si para cualquier pareja de puntos A y B, contenidos en Y, cada punto en el segmento que los une también pertenece a Y [16]. En otras palabras, los politopos convexos no presentan huecos ni curvas cóncavas en sus caras. Otra forma de definir este concepto puede ser como el espacio de *n* dimensiones delimitado por un número finito de inecuaciones lineales de tal forma que el politopo sería la intersección de los semiespacios descritos por dichas desigualdades.

Si resulta imposible eliminar un punto del politopo sin cambiar su forma, a dicho punto se le llama vértice. Para una determinada desigualdad lineal que se cumple para todo el politopo, se llama cara al conjunto de los puntos que la cumplen exactamente. Cada cara es, asimismo, otro politopo. Se define como dimensión del politopo a la dimensión del espacio euclídeo más pequeño posible que pueda contenerlo. Por otra parte, se llama facetas a las caras de un politopo cuya dimensión es una unidad menor que la del politopo [16].

Cada politopo está formado a su vez por otros politopos, pudiendo expresarse el primero como una combinación de los segundos [16].

Matemáticamente, se puede definir un politopo a partir de vértices (Politopo-V) o de desigualdades lineales (Politopo-H) de la siguiente forma [17]:

Politopo-V: Envolvente convexa de un número finito de puntos $X = \{x^1, \dots, x^n\}$ en $\mathbb{R}^d$.

$$P = conv(X) = \left\{ \sum_{i=1}^{n} \lambda_i \cdot x_i \mid \lambda_1, \dots, \lambda_n \geq 0, \sum_{i=1}^{n} \lambda_i = 1 \right\}$$

Politopo-H: Solución de un número finito de inecuaciones lineales,

$$P = P(A, b) = \{x \in \mathbb{R}^d \mid a_i^T \cdot x \leq b_i \; para \; 1 \leq i \leq m\}$$

con la condición de que el conjunto de soluciones sea acotado, esto es, que exista una constante N, tal que $\|x\| \leq N$ para todo $x \in P$. En este caso, $A \in \mathbb{R}^{mxd}$ es una matriz de números reales con columnas a_i^T, y $b \in \mathbb{R}^m$es un vector cuyos elementos son b_i.

Para crear un poliedro utilizando Multi-Parametric Toolbox se deben definir, en primer lugar, sus vértices como una matriz de *m* filas por *n* columnas, donde *m* es el número de vértices y *n* es la dimensión del poliedro. En segundo lugar, se llama a la función *Polyhedron*, a la que se le pasa como parámetro de entrada la matriz anterior con los vértices. Si el poliedro creado es de tres dimensiones o menos, puede representarse mediante el comando *plot* [18]. En la Figura 14 se muestra un ejemplo de la creación y representación de un poliedro de tres dimensiones.

```
% Declaración de los vértices
vertices=[0 0 0;0 0 1;0 1 -1;0 1.5 0;0 1 1;2 2 0];
% Creación del poliedro P
P=Polyhedron(vertices);
% Representación
plot(P)
xlabel('x')
ylabel('y')
zlabel('z')
```

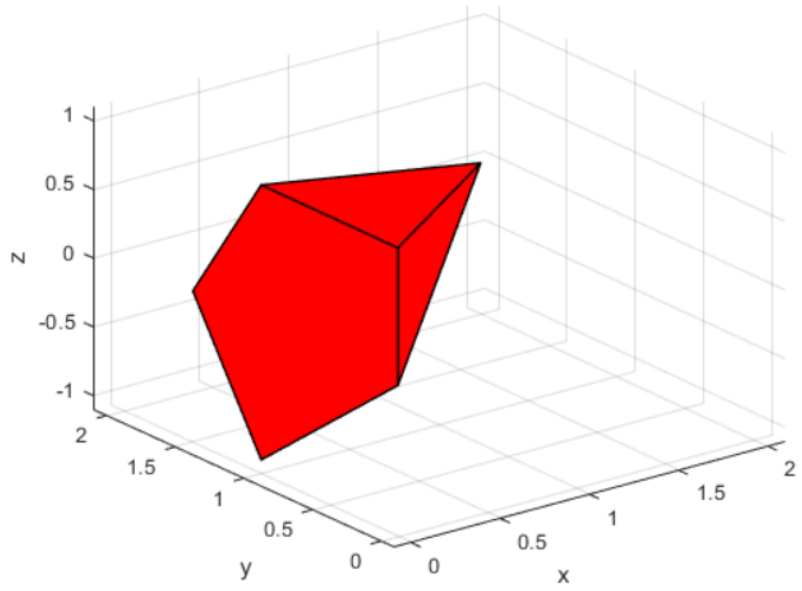


Figura 14. Ejemplo de creación de un politopo con Multi-Parametric Toolbox

Las principales funciones que incluye esta *toolbox* para trabajar con poliedros y politopos son las siguientes [18]:

- El comando *P.V* devuelve los vértices del poliedro *P*.
- La función *P.contains(p)* devuelve *true* si el punto *p* está contenido en el poliedro *P* y *false* en caso contrario.
- La instrucción *P.isEmptySet()* devuelve *true* si el poliedro *P* está vacío y *false* en caso contrario.
- *P.projection(a:b)* devuelve la proyección del poliedro *P* en las dimensiones entre *a* y *b*.

En la Figura 15 se muestran ejemplos de los comandos descritos aplicados al poliedro definido en el ejemplo de la Figura 14.

```
% Obtención de los vértices del poliedro
verticesPoliedro=P.V;
% Comprobar si un punto está contenido en el poliedro
punto1=[0.5;1;0];
punto2=[1;0;0];
Contiene_punto1=P.contains(punto1)
Contiene_punto2=P.contains(punto2)
% Comprobar si el poliedro está vacío
Vacio=P.isEmptySet()
% Proeyección
Proyeccion=P.projection(1:2);
plot(Proyeccion)
xlabel('x')
ylabel('y')
```

```
Contiene_punto1 =

  logical

   1


Contiene_punto2 =

  logical

   0


Vacio =

  logical

   0
```

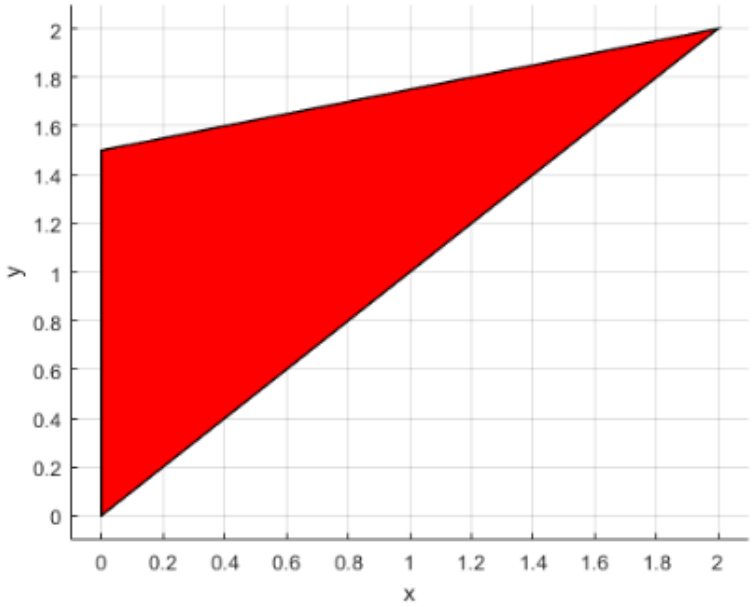


Figura 15. Ejemplo de aplicación de los comandos básicos de Multi-Parametric Toolbox

Tal y como muestra la Figura 15, el primer punto está contenido en el poliedro, mientras que el segundo no lo está. Además, el poliedro no está vacío y se obtiene la proyección en las dos primeras dimensiones.

También se pueden llevar a cabo las operaciones lógicas *AND* y *OR* entre politopos, de forma que la primera devuelve el politopo que resulta de la intersección de otros dos y la segunda de su superposición. En la Figura 16 se muestra un ejemplo del código y en la Figura 17, los resultados que se obtienen.

```
% Declaración de los vértices
vertices1=[0 0;1 0;0 1;1 1];
vertices2=[0.5 0.5;1.5 0.5;0.5 1.5;1.5 1.5];
% Creación de los poliedros P1 y P2
P1=Polyhedron(vertices1);
P2=Polyhedron(vertices2);
% Operación AND entre P1 y P2
P3=and(P1,P2);
%Representación
figure
plot(P3)
title('Operación AND')
xlabel('x')
ylabel('y')
xlim([0 1.5])
ylim([0 1.5])
% Operación OR entre P1 y P2
P4=or(P1,P2);
%Representación
figure
plot(P4)
title('Operación OR')
xlabel('x')
ylabel('y')
xlim([0 1.5])
ylim([0 1.5])
```

Figura 16. Ejemplo de código para realizar operaciones lógicas entre politopos con Multi-Parametric Toolbox

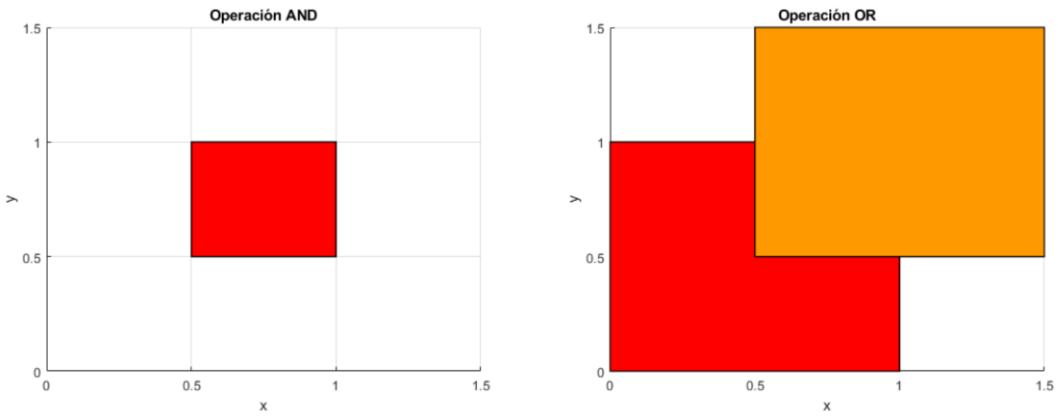


Figura 17. Resultados de las operaciones lógicas realizadas en el ejemplo de la Figura 16

En el caso de las Figuras 16 y 17, existe la intersección entre los politopos. Si no existiera, el resultado de la operación *AND* sería un politopo vacío, como se observa en el ejemplo de la Figura 18.

.

```
% Declaración de los vértices
vertices1=[0 0;1 0;0 1;1 1];
vertices2=[2 2;3 2;2 3;3 3];
% Creación de los poliedros P1 y P2
P1=Polyhedron(vertices1);
P2=Polyhedron(vertices2);
% Operación AND entre P1 y P2
P3=and(P1,P2);
%Comprabar si está vacío
Vacio=P3.isEmptySet()
%Representación
figure
plot(P3)
title('Operación AND')
xlabel('x')
ylabel('y')
```

```
Vacio =

  logical

   1
```

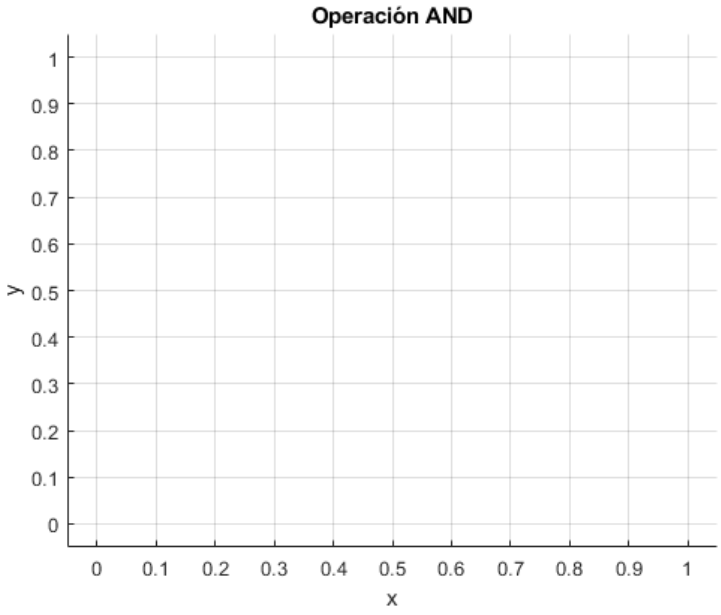


Figura 18. Ejemplo de función AND entre dos politopos que no intersectan

4. Desarrollo

En este capítulo se lleva a cabo una explicación detallada de la implementación de la solución adoptada en este trabajo para lograr los objetivos propuestos.

4.1. Aplicación del método Denavit-Hartenberg al IRB 140 de ABB

En primer lugar, con el fin de poder realizar los cálculos necesarios para la cinemática del robot, es necesario establecer los sistemas de referencia asociados a cada uno de los eslabones que lo componen, así como las relaciones entre ellos. Para ello, se aplica la convención Denavit-Hartenberg. Para elegir el sentido de los ejes z, se tiene en cuenta el sentido positivo de giro de cada articulación, que figura en el *datasheet* [19].

El sistema de referencia 0, se sitúa en la base del robot. El eje z_0 coincide con el eje de giro de la primera articulación y los ejes x_0 e y_0 se han colocado de forma que el sistema sea dextrógiro; tal y como se muestra en la Figura 19.

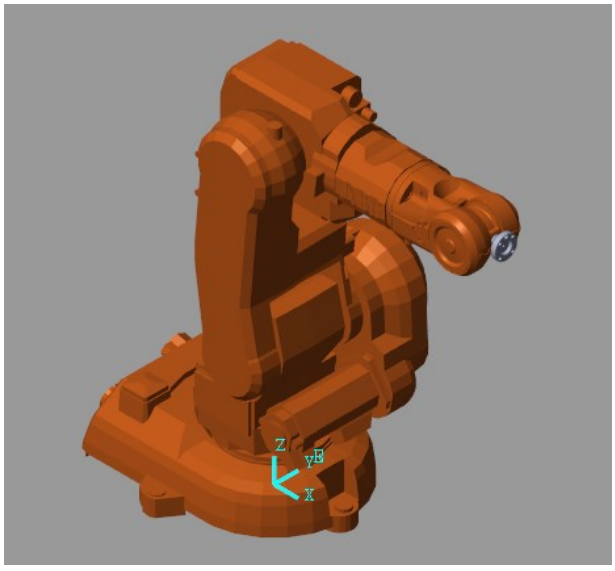


Figura 19. Sistema de referencia asociado a la base

El eje $z_1$coincide con el eje de la segunda articulación, y el origen del sistema se halla en el punto donde se cortan z_0 y z_1. El eje x_1 está en la normal a z_0 y z_1, e y_1, al igual que el resto de los ejes designados por la letra y, se coloca de manera que el sistema sea dextrógiro. Todo ello se puede apreciar en la Figura 20.

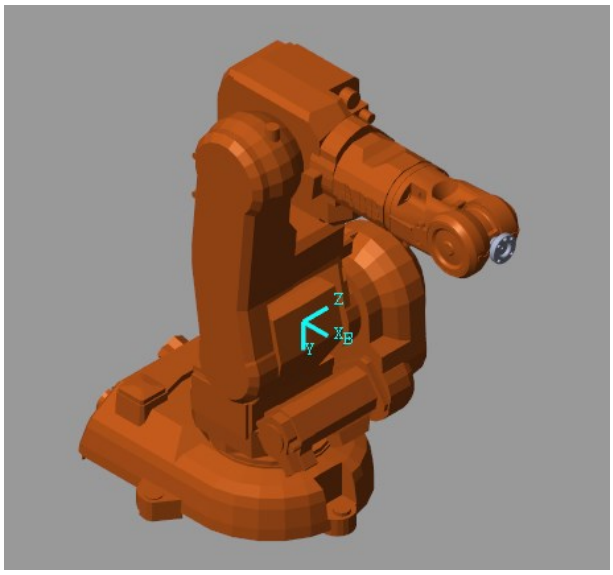


Figura 20. Sistema de referencia asociado al primer eslabón

El eje z_2 es el eje de giro de la tercera articulación. Al ser paralelos z_2 y z_1, cualquiera de las rectas normales a ambos es válida para situar x_2. Por simplicidad, se elige la que atraviesa el origen del sistema de referencia anterior. El nuevo sistema se expone en la Figura 21.

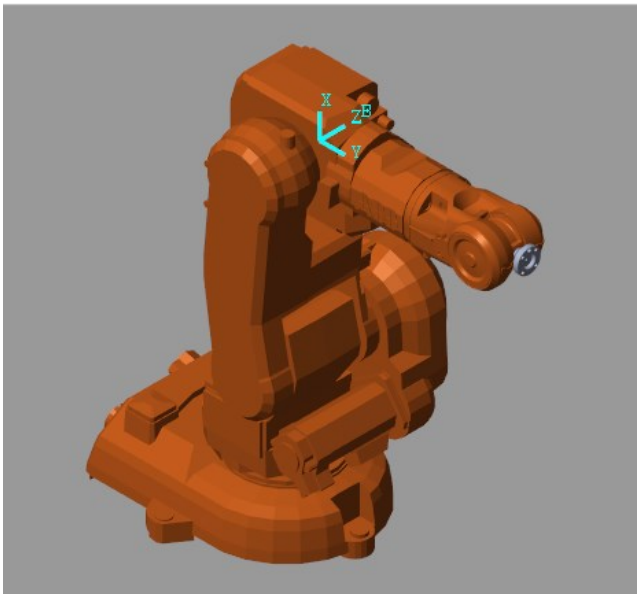


Figura 21. Sistema de referencia asociado al segundo eslabón

El eje z_3 es el de la cuarta articulación y el origen del sistema está en el punto donde se cortan z_3 y z_2, que en este caso coincide con el origen del sistema anterior. En la normal a z_3 y z_2, se coloca x_3, como se ve en la Figura 22.

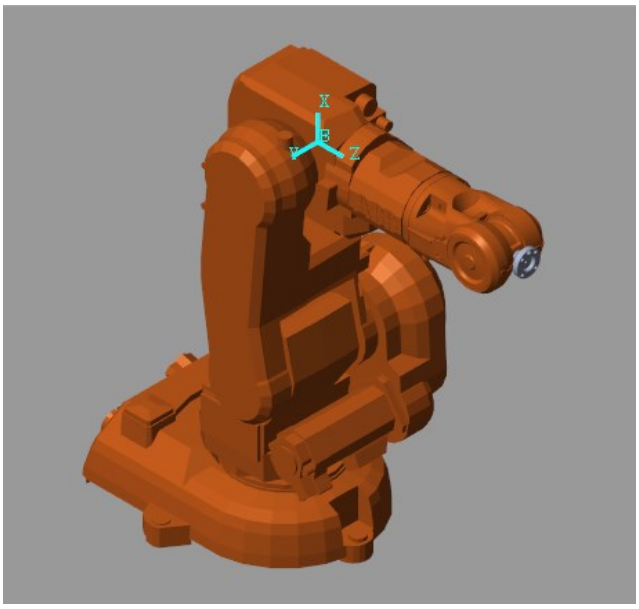


Figura 22. Sistema de referencia asociado al tercer eslabón

El eje z_4 es coincidente con el de la quinta articulación y el origen del sistema 4 está en el punto donde se cortan z_4 y z_5.En la normal a z_4 y z_5, se halla x_4, lo cual se aprecia en la Figura 23.

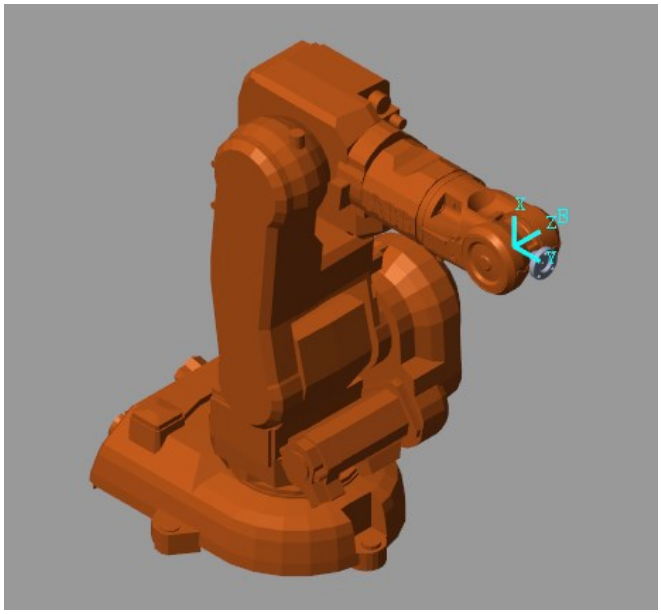


Figura 23. Sistema de referencia asociado al cuarto eslabón

El eje z_5 es el de la sexta y última articulación y el origen del sistema está en el punto donde se cortan z_5 y z_4, que coincide con el origen del sistema anterior. En la normal a z_5 y z_4, se coloca x_5. Todo ello se observa en la Figura 24.

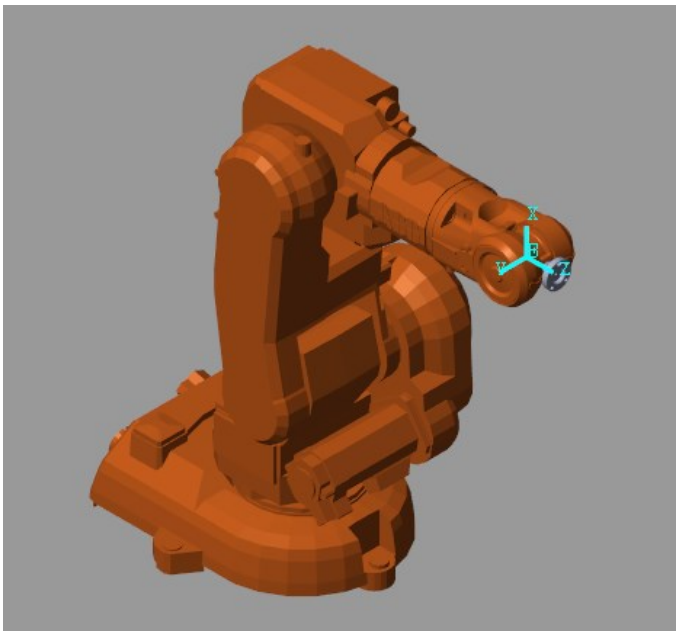


Figura 24. Sistema de referencia asociado al quinto eslabón

Finalmente, el último sistema de referencia tiene su origen en el extremo. El eje z_6 coincide con z_5, y el x_6 es normal a z_5 y z_6, como se puede ver en la Figura 25.

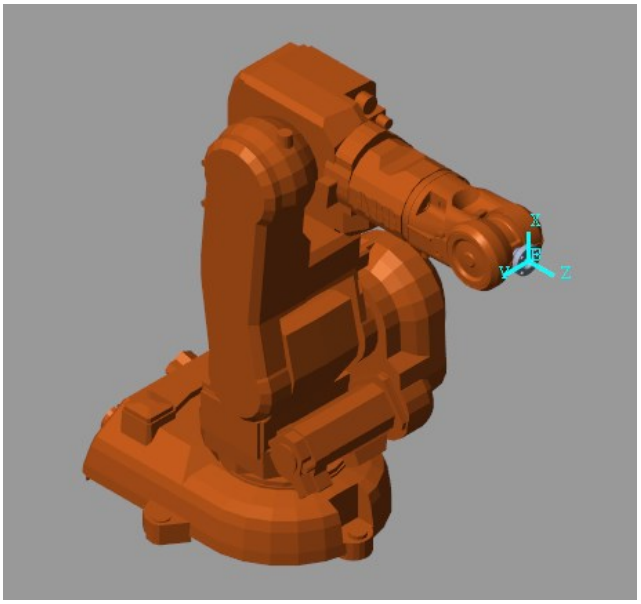


Figura 25. Sistema de referencia asociado al extremo

Una vez definidos los sistemas de referencia asociados a los eslabones se deben calcular los parámetros Denavit-Hartenberg, para lo cual, es preciso conocer la geometría del robot. Los

datos necesarios se extraen del *datasheet* del IRB 140 de ABB [19] y corresponden a la Figura 26.

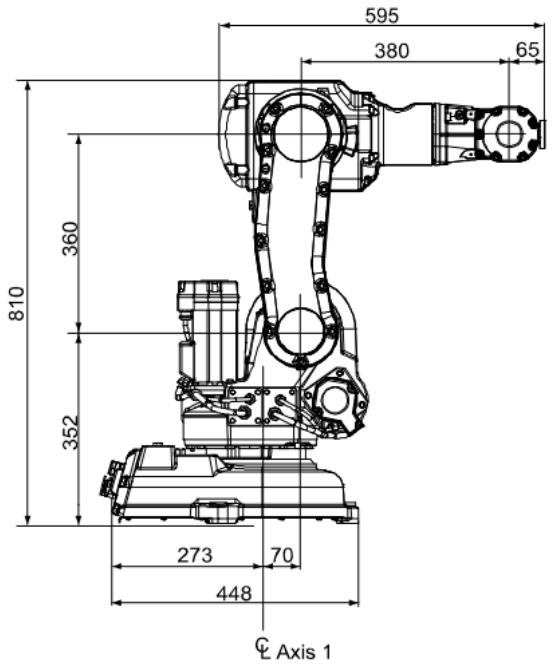


Figura 26. Dimensiones del robot extraídas del datasheet [19]

En la Tabla 1 se muestran los parámetros Denavit-Hartenberg para el brazo robótico IRB 140 de ABB.

i	θ_i	d_i	a_i	α_i
1	q_1	$352\ mm$	$70\ mm$	$-90°$
2	$q_2 - 90°$	0	$360\ mm$	0
3	q_3	0	0	$-90°$
4	q_4	$380\ mm$	0	$+90°$
5	q_5	0	0	$-90°$
6	q_6	$65\ mm$	0	0

Tabla 1. Parámetros Denavit-Hartenberg del IRB 140 de ABB

4.2. Importación de CAD en Simscape Multibody

Se pueden obtener los archivos CAD con los modelos en 3 dimensiones de los eslabones del robot, junto con sus datos de masa e inercia, en la página de ABB [20]. Se puede descargar el robot completo, en un solo archivo STEP, o por eslabones, en siete ficheros STEP distintos. Se elige la primera opción, en la que ya están ensambladas las piezas.

Para realizar la importación se deben instalar los *links* de Simscape Multibody, disponibles en [21], que permiten la exportación de archivos de Solidworks a Matlab (ficheros con extensión XML).

Tras la instalación de los *links*, se abre el modelo del robot en Solidworks y, en el menú Herramientas se busca la opción Simscape Multibody y se exporta a un archivo XML. También aparecen, en la carpeta donde se guarda este fichero, las piezas del robot en formato STEP y un fichero de Matlab llamado *DataFile*, que contiene datos de los eslabones tales como su masa, inercia, color, opacidad, etc. Este último *script* debe ejecutarse ates de realizar la simulación, ya que esta necesita los datos contenidos en él. Seguidamente, se ejecuta el comando *smimport*, al que se le pasa como parámetro de entrada el nombre del archivo XML. Se crea automáticamente un archivo de Simulink como el mostrado en la Figura 27 con una serie de bloques que representan las piezas del robot.

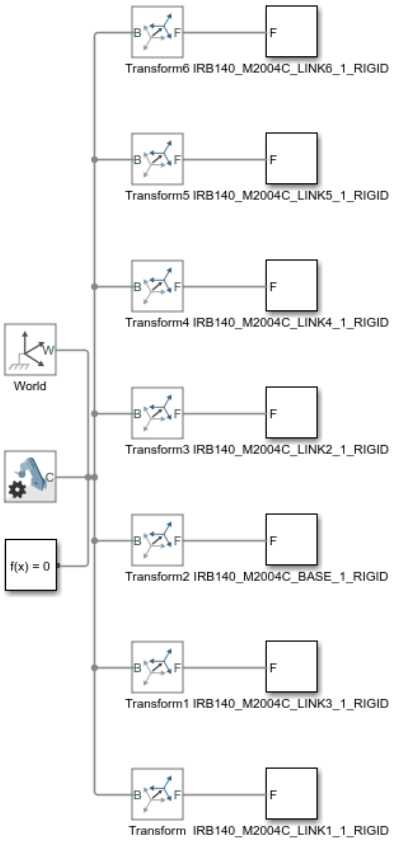


Figura 27. Modelo de Simulink importado

Los tres primeros bloques de la Figura 27, de arriba abajo, son el sistema de referencia fijo, la configuración del mecanismo, que contiene el vector de la aceleración de la gravedad, y la configuración del *solver*. Seguidamente hay siete bloques con la base y los eslabones a los que, previamente se aplica una matriz de transformación definida en el *script DataFile* y que, por defecto expresa una rotación y una traslación nulas.

La Figura 28 muestra el contenido de cada uno de los bloques de los eslabones. Cada bloque contiene un sólido, que representa al eslabón, y su sistema de referencia asociado.

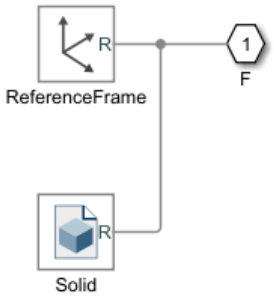


Figura 28. Contenido de los bloques de los eslabones del robot en Simulink

Al llevar a cabo la simulación, se abre Simscape Multibody y se observa el brazo, tal y como se ve en la Figura 29.

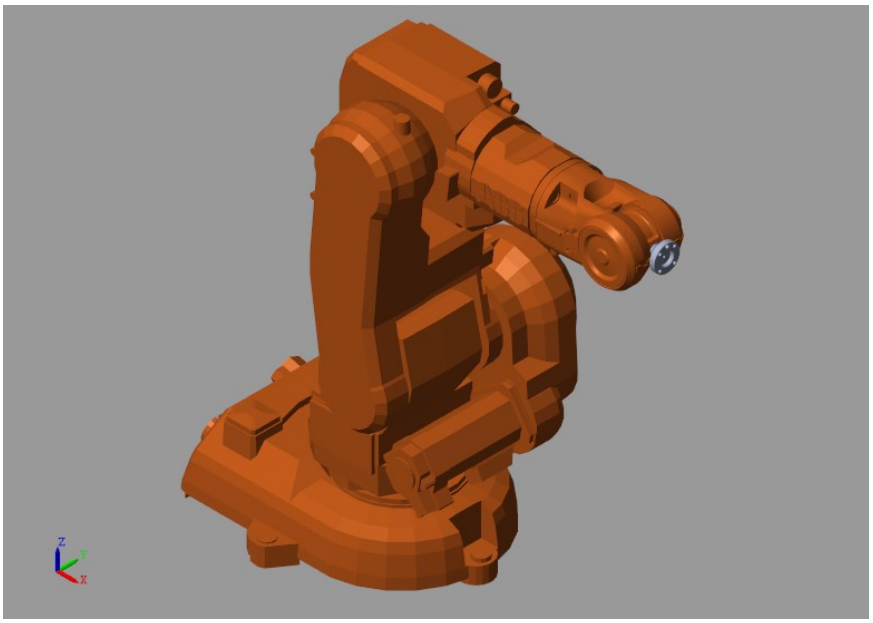


Figura 29. Resultado de la simulación con el robot estático

Esta importación, no permite todavía la simulación de trayectorias del robot ya que, en primer lugar, todos los eslabones están referenciados a un mismo sistema de referencia fijo en la base y, en segundo lugar, el esquema de Simulink no incorpora ninguna articulación, lo que convierte al robot en un mecanismo fijo sin posibilidad de movimiento. Así pues, es necesario modificar los sistemas de referencia para que coincidan con los calculados mediante la convención Denavit-Hartenberg y añadir las articulaciones de rotación necesarias.

Se extraen los sólidos de los bloques que los contienen para simplificar el esquema y, en lugar de colocarlos en paralelo, se colocan en serie, intercalando entre ellos el modelo de una articulación de revolución. Además, antes de cada una de las articulaciones se coloca una matriz de transformación, que relaciona el sistema fijo de la base con el sistema asociado a cada eslabón según el criterio Denavit-Hartenberg. Ahora bien, dado que los eslabones han sido creados tomando como referencia el sistema de la base, si se aplica la transformación y tras ella la rotación, el eslabón se traslada y rota según la matriz de transformación y, después gira según la articulación. En realidad, únicamente debería ocurrir lo segundo, pues se desea mover el sistema de referencia del eslabón sin mover este, lo cual no es posible porque dicho sistema está asociado al eslabón y mover uno conlleva mover el otro. Para solucionar este problema, se aplica la transformación, después el efecto de la articulación y, finalmente, se aplica la inversa de la matriz de transformación, para que el eslabón vuelva a su posición tras girar el valor que toma la articulación. El esquema modificado se muestra en la Figura 30.

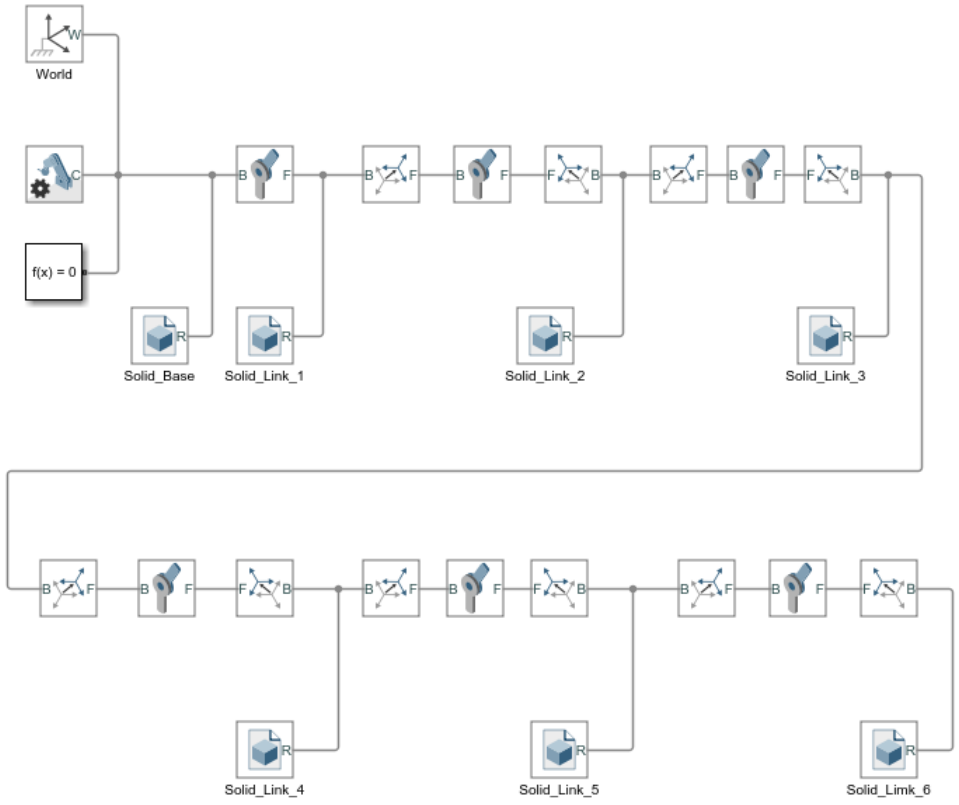


Figura 30. Modelo de Simulink con las articulaciones de revolución y las transformaciones necesarias

Cabe resaltar que, en el caso de la primera articulación no se aplican transformaciones ya que, el sistema de referencia del primer giro coincide con el de la base. El resto de las articulaciones se hallan precedidas de la transformación que relaciona el sistema fijo con el sistema con respecto al que giran para que la rotación se produzca con respecto al eje correcto, y sucedidas por la transformación inversa para que los eslabones giren con la articulación, pero no se muevan junto con los sistemas debido a la transformación previa. Por otro lado, también es importante destacar que se ha desactivado la gravedad en la configuración del mecanismo, ya que, al no haberse diseñado un sistema de control de la posición, la gravedad influye en el movimiento de las articulaciones como una perturbación.

Ahora, el robot puede moverse. Por ejemplo, si se establecen en los bloques de las articulaciones los valores $q_1 = -90°$, $q_2 = 30°$, $q_3 = 20°$, $q_4 = 90°$, $q_5 = 90°$ y $q_6 = 30°$ se obtiene el resultado que aparece en la Figura 31.

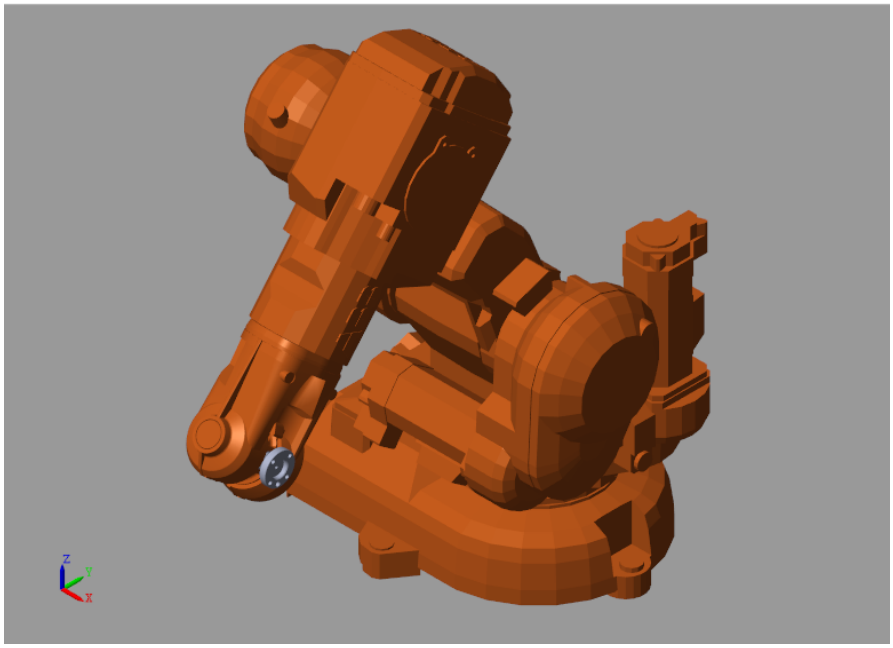


Figura 31. Simulación del robot con valores articulares de -90°, 30°, 20°, 90°, 90° y 30°

4.3. Verificación de colisiones

Una parte fundamental de la planificación de trayectorias radica en comprobar si en una configuración determinada, o en la ruta que une dos configuraciones, el robot colisiona consigo mismo o con alguno de los obstáculos presentes en el entorno.

Una posibilidad para resolver este problema podría ser modelar, mediante ecuaciones trigonométricas y la cinemática inversa, el espacio de configuraciones en seis dimensiones, distinguiendo C_{free} y C_{obs}. Sin embargo, las expresiones que se obtienen son complejas y resolverlas, así como trabajar en un espacio de seis dimensiones, conllevaría una enorme carga computacional y un elevado tiempo de ejecución. Como alternativa, se ha decidido trabajar en el espacio cartesiano de tres dimensiones y no directamente en el espacio de configuraciones.

Para verificar las colisiones se han implementado dos funciones en Matlab: *Colision_Configuracion_Obstaculos*, que comprueba si una configuración está o no en C_{free}, y *Colision_Curva_MoveAbsJ_Obstaculos*, que comprueba si en la trayectoria creada entre dos nodos mediante el movimiento absoluto de ejes se produce alguna colisión. En ambos casos la función devuelve un cero lógico si no se produce ningún choque y un uno lógico en caso contrario.

La idea básica de estas funciones es el cálculo de la posición de una serie de puntos pertenecientes al robot en el espacio cartesiano para comprobar una configuración, o de la trayectoria que siguen dichos puntos para verificar una ruta entre dos nodos del espacio de configuraciones. Para ello, se definen todos los obstáculos como poliedros utilizando Multi-Parametric Toolbox y se comprueba mediante la instrucción *contains*, explicada en el apartado 3.5.2, si los puntos tomados sobre el robot están contenidos en algún obstáculo, en cuyo caso se produce un choque.

Esto sería válido si el robot no tuviera volumen, pero los puntos seleccionados del robot se hallan en eslabones que tienen un cierto volumen, por lo que el punto podría no entrar en ningún obstáculo y si hacerlo el eslabón al que pertenece. Para solventar esta problemática, se ha optado por ampliar los obstáculos una distancia determinada para cada uno de los puntos del robot que se verifican. Esta distancia depende del eslabón al que pertenece cada punto, y es igual, o ligeramente mayor para dejar un margen de seguridad, que la distancia entre el punto elegido en el robot y el punto más lejano del eslabón o la parte del eslabón cuyas colisiones se encarga de comprobar dicho punto.

De este modo, se evitan colisiones en un radio determinado desde los puntos elegidos en el robot. Dicho radio no es otro que la distancia que se amplían virtualmente los obstáculos para comprobar si el punto o la trayectoria del punto se hallan contenidos en él. La Figura 32 muestra la situación de los puntos elegidos y la distancia que se deben aumentar los objetos del entorno. En la imagen, tal distancia se ha representado como el radio de una esfera, de modo que todo cuanto queda en el interior de las esferas no puede entrar en ningún obstáculo.

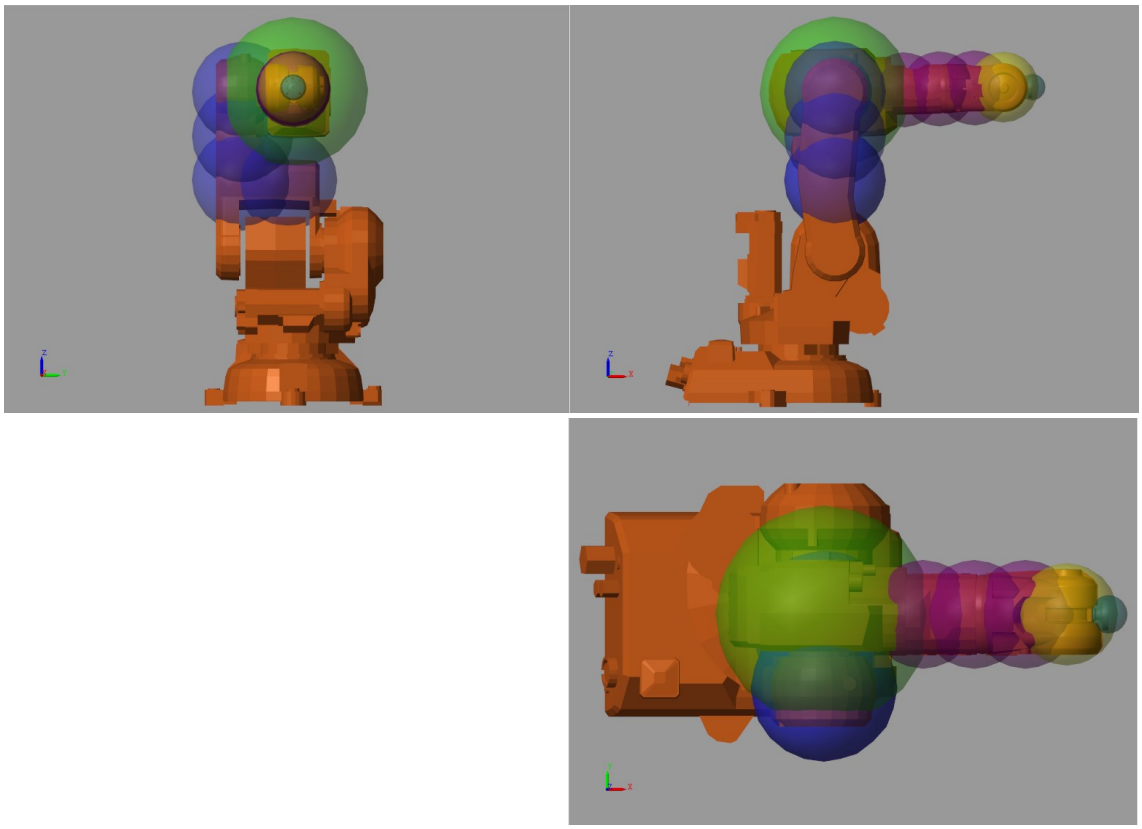


Figura 32. Puntos seleccionados en el robot para verificar las posibles colisiones rodeados por una esfera cuyo radio corresponde a la ampliación de los obstáculos para cada uno de ellos

Tal y como puede apreciarse en la Figura 32, se han seleccionado diez puntos para comprobar si el robot colisiona o no con algún obstáculo, que son los siguientes:

- Uno situado en el extremo del robot, coincidiendo con el origen del sistema de referencia número seis, con un radio de 30 mm y representado en cian en la imagen anterior.
- Otro situado en el punto de muñeca, coincidiendo con los orígenes de los sistemas cuatro y cinco. Su radio es de 80 mm y aparece en color amarillo en la Figura 32.
- Tres situados a lo largo de los eslabones tres y cuatro, representados en violeta y colocados a distancias de 155, 235 y 315 mm desde la tercera articulación. Tienen todas ellas un radio de 85 mm.
- Uno colocado en el origen de los sistemas dos y tres con un radio de 170 mm y dibujado en color verde.
- Cuatro en el segundo eslabón, en color azul en la imagen con un radio de 115 mm. Con respecto al sistema de referencia uno, se hallan, tres de ellos desplazados una distancia de 115 mm en el sentido negativo del eje Z; y 160, 260 y 360 mm en el sentido negativo del eje Y. El cuarto está trasladado 160 mm en el sentido negativo del eje Y, y 15 mm en el sentido negativo del eje Z.

En cuanto a los obstáculos, se debe considerar, primeramente, la colisión del robot consigo mismo. Es posible que los eslabones cuarto y sexto colisionen con la base y con el primer eslabón

en su parte cilíndrica, situada abajo. Las colisiones entre el resto de los eslabones no son posibles debido a los límites de los valores que pueden tomar las articulaciones.

Para estudiar la colisión con la base, se han creado tres envolventes para cubrirla. Estas envolventes se pueden representar en Simscape Multibody creando un sólido en Simulink y pueden ser tratadas como poliedros para verificar el posible impacto mediante las funciones de Multi-Parametric Toolbox. La base, tal y como se aprecia en la Figura 33, es cilíndrica por la parte delantera y tiene forma de prisma por detrás. Además, en la parte trasera hay elementos más altos que el resto de la base, como muestra la Figura 34.

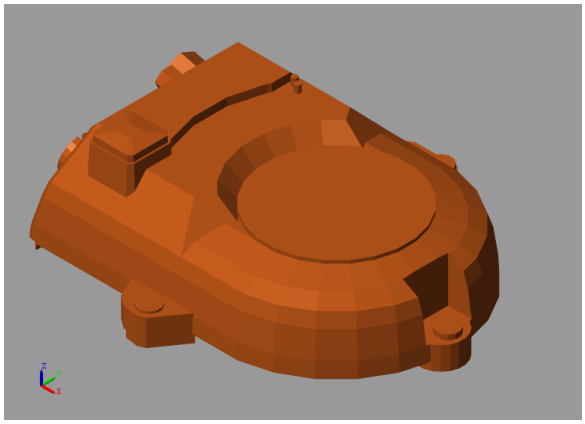


Figura 33. Vista isométrica de la base del IRB 140 de ABB

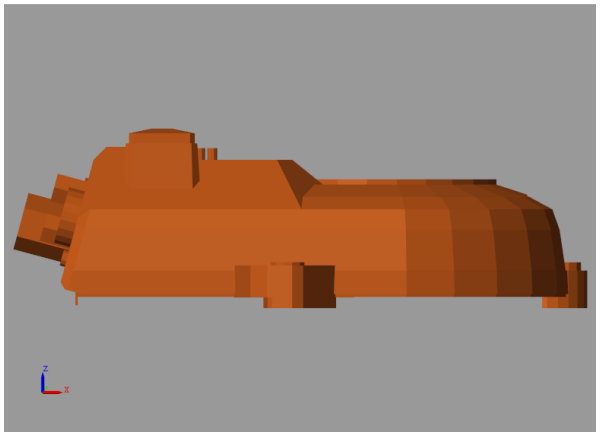


Figura 34. Vista de perfil de la base del IRB 140 de ABB

Para lograr una buena aproximación, se ha creado una primera envolvente con forma cilíndrica para la parte delantera, una segunda prismática para la parte trasera y una tercera, también prismática, de menor base que la anterior, pero de mayor altura para cubrir las partes que sobresalen. El resultado se aprecia en las Figuras 35 y 36. Estas envolventes son consideradas como obstáculos en la planificación de movimientos.

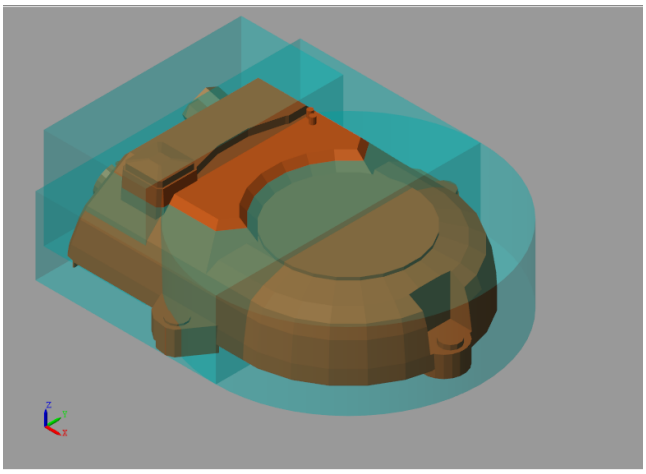


Figura 35. Vista isométrica de la base del IRB 140 de ABB con sus envolventes

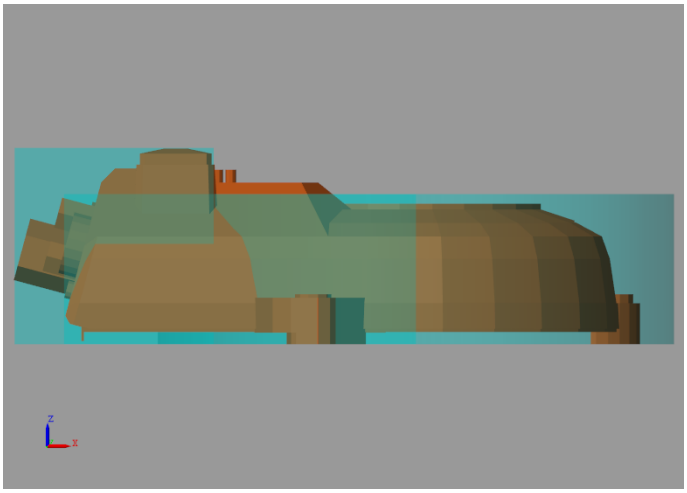


Figura 36. Vista de perfil de la base del IRB 140 de ABB con sus envolventes

En cuanto a la colisión con el primer eslabón, una posible solución es tomar también puntos a lo largo de él y calcular su posición o trayectoria para después verificar si la distancia entre los puntos de este eslabón y los de los eslabones 4 y 6 están lo suficientemente alejados en el espacio como para que no se produzca colisión. No obstante, dado que la forma de la parte baja de la pieza, que es la que podría colisionar puede aproximarse a un cilindro centrado en el eje de giro de la primera articulación, se ha optado por representar dicho cilindro como un obstáculo fijo para mayor simplicidad, como enseña la Figura 37. Si el extremo de robot o el cuarto eslabón entran en dicho cilindro, ampliado la distancia correspondiente según el caso, se considera que se produce un choque con el primer eslabón.

Figura 37. Eslabón primero con su envolvente

Por lo que a los obstáculos respecta, se ha creado un entorno con cuatros obstáculos además del suelo, con el que también se deben comprobar las colisiones. La Figura 38 muestra el robot en su área de trabajo.

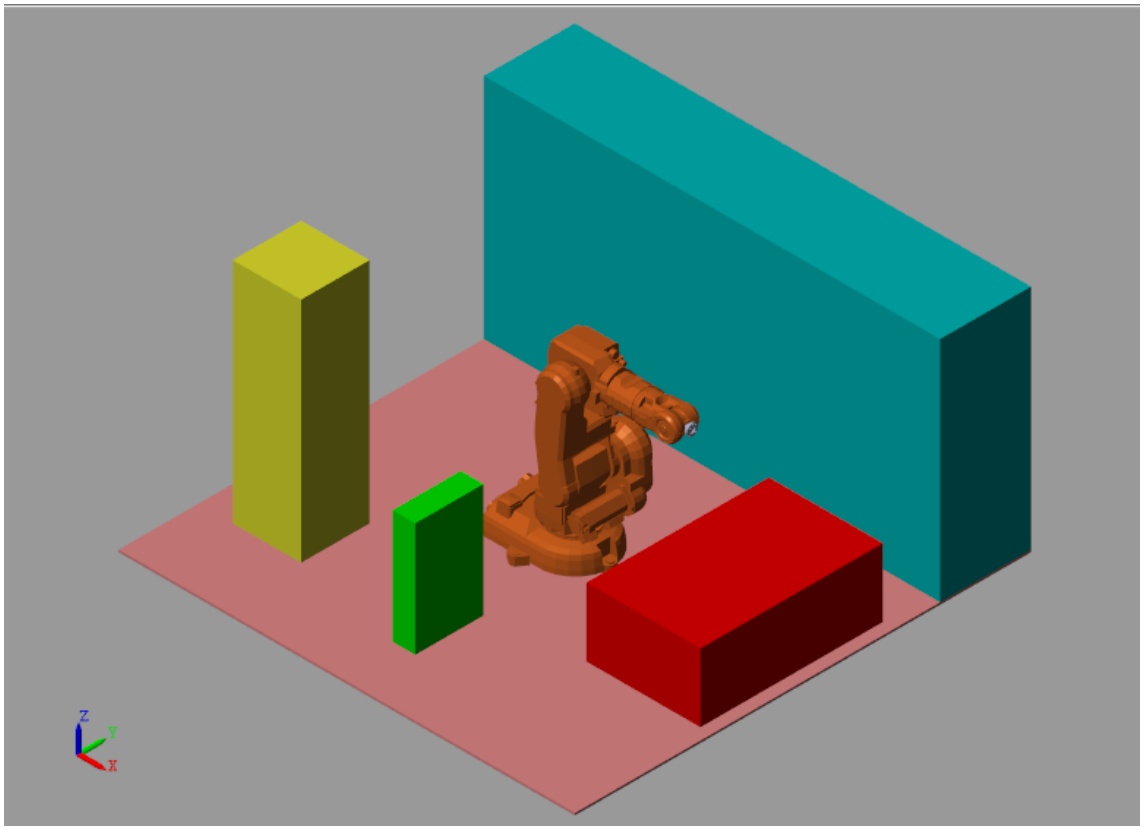


Figura 38. Entorno del robot con obstáculos

Además de los posibles choques con el propio robot deben comprobarse las colisicones entre los diez puntos elegidos del robot y los cuatro obstáculos y el suelo. Sin embargo, hay algunas excepciones que no necesitan comprobación. El segundo eslabón nunca llegará a chocar con el suelo de debido a los límites de la segunda articulación. Para la pared de color azul, basta con comprobar si chocan los puntos del extremo y de muñeca pues entrarán siempre antes que el resto de los puntos. En cuanto al suelo, podría darse el caso de que, en una configuración dada, el extremo y el punto de muñeca queden por debajo de la delgada capa que representa el suelo, con lo que habría que comprobar también los tres puntos situados entre el punto de muñeca y la tercera articulación, así como el punto situado en dicha articulación. Sin embargo, si al crear los poliedros en Matlab se le da al suelo un grosor hacia abajo lo bastante grande, bastaría con comprobar solo el extremo y el punto de muñeca, pues al descender por debajo del suelo quedarían siempre dentro del poliedro.

La Figura 39 muestra los obstáculos modelados como poliedros en Matlab mediante la herramienta Multi-Parametric Toolbox.

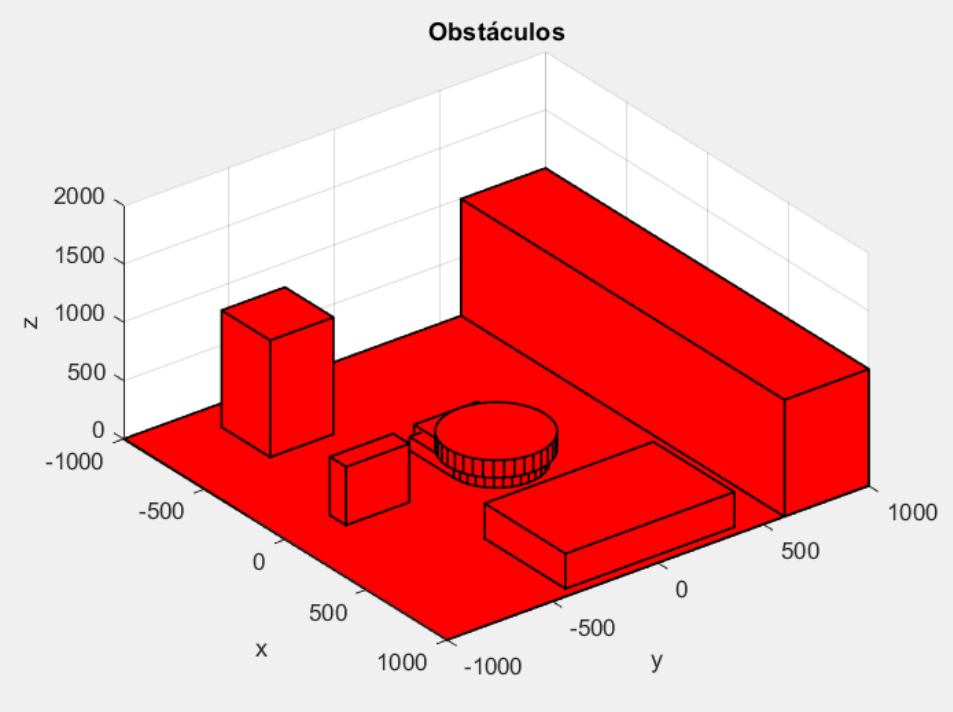


Figura 39. Modelo de los obstáculos mediante politopos

Una vez expuesto el procedimiento en el que se basa la verificación de colisiones se procede a explicar con detalle las funciones *Colision_Configuracion_Obstaculos* y *Colision_Curva_MoveAbsJ_Obstaculos*. Estas funciones llaman a su vez a otras funciones previamente creadas y que conviene presentar.

La función *moveAbsJ* crea una trayectoria cúbica que permite simular el movimiento del robot mediante el movimiento absoluto de ejes. Calcula los coeficientes de la función cúbica y la ley

que define los valores de las articulaciones en función del tiempo. Devuelve una matriz con dos columnas. La primera de ellas contiene el tiempo en segundos, que se va incrementando 5 ms desde un tiempo inicial hasta un tiempo final. La segunda se compone de los valores que deben tomar las articulaciones en cada instante. Como parámetros de entrada necesita las configuraciones de partida y de destino, el tiempo que dura la trayectoria (T) y un valor, denominado *num*, que indica el tiempo inicial. Este último dato multiplicado por el valor del tiempo que dura el movimiento es el instante inicial de la trayectoria.

La función *Trayectoria_xyz* devuelve las coordenadas cartesianas de los diez puntos tomados en el robot para la verificación de las colisiones. Como parámetros se le pasan los seis vectores con las configuraciones que describen la trayectoria entre dos nodos. Dichos vectores tendrán un único elemento cuando se utilice para verificar una única configuración y no una ruta entre dos configuraciones. Se recorren los seis vectores y, para cada grupo de seis elementos con los mismos índices, a partir de los parámetros Denavit-Hartenberg y de los desplazamientos necesarios para llegar a los puntos seleccionados en el robot que no coindicen con ningún sistema de referencia, se calculan las matrices de transformación que relacionan cada uno de esos diez puntos con el sistema de referencia fijo. Habiendo obtenido las matrices, se tiene la posición de los puntos. Dentro de esta función se llama a otra, denominada *MatrizTransformacion* que simplemente devuelve una matriz de transformación a partir de los parámetros Denavit-Hartenberg.

La función *Colision_Configuracion_Obstaculos* recibe como parámetros todos los obstáculos ampliados para las distintas distancias establecidas y un vector, q, que contiene la configuración del robot, esto es, los valores de las seis articulaciones. Primeramente, se comprueba que estos valores se hallan dentro de los límites de las articulaciones del robot establecidas por el fabricante. En caso afirmativo, se llama a la función *Trayectoria_xyz* que devuelve la posición de los diez puntos del robot utilizados para determinar si se produce o no colisión. En este caso, se está comprobando una configuración y no una ruta entre dos configuraciones, por lo que esta función devuelve puntos, y no vectores de puntos. A continuación, para cada uno de los diez puntos del robot se comprueba mediante la función *contains* si está contenido en alguno de los obstáculos, ampliados según el caso, con los que podría colisionar. Si ninguno está contenido en ningún obstáculo la función devuelve un cero lógico indicando que no hay colisión y un uno lógico en caso contrario, señalando que se produce un choque.

Por otro lado, la función *Colision_Curva_MoveAbsJ_Obstaculos* recibe como entradas los obstáculos ampliados para los distintos radios, dos vectores, q0 y qF, que contienen las configuraciones inicial y final respectivamente, y el tiempo, T, que requiere la función *moveAbsJ*. Esta función se utiliza en este caso, no para una simulación, sino para crear los vectores de valores de cada una de las articulaciones que definen la trayectoria. En este caso, este parámetro T no define la duración de la simulación, pues no se está simulando; pero sí determina el número de elementos de los vectores que devuelve *moveAbsJ*, ya que esta función crea un nuevo punto cada 5 ms como se ha explicado anteriormente. Por tanto, el parámetro T determina el número de puntos generados al muestrear las trayectorias entre nodos.

Dentro de esta función, en primer lugar, se llama a *moveAbsJ*, a la que se le pasan las configuraciones final e inicial, el parámetro T y el número 0 para indicar que el tiempo de inicio

de la trayectoria es cero. En este caso, este último dato es arbitrario, pues, como ya se ha indicado, en este caso no se va a simular y el tiempo es irrelevante. En segundo lugar, se llama a la función *Trayectoria_xyz* a la que se le pasa la segunda columna de las matrices devueltas por *moveAbsJ*, ya que la primera columna son los instantes de tiempo y la segunda los valores articulares. *Trayectoria_xyz* devuelve una sucesión de puntos, que definen las trayectorias de los diez puntos del robot que se han de comprobar. A continuación, se comprueba, para cada una de las trayectorias de los puntos del robot, si entran o no en alguno de los obstáculos debidamente ampliados en cada caso. Para ello, se usa la función *contains*, pasando como parámetro la trayectoria del punto correspondiente. Ahora, este método devuelve un vector con ceros y unos lógicos ya que se le está pasando un vector de puntos, es decir una ruta, y no un solo punto. Para cada configuración muestreada en la trayectoria se añade un cero al vector de salida si no está contenido en el poliedro y un uno si sí lo está. Es preciso, por tanto, recorrer el vector resultante en busca de unos lógicos que indiquen colisión. Esto se lleva a cabo mediante el comando *find*. Esta instrucción, tras recibir como parámetro el vector obtenido como resultado de la función *contains*, devuelve un nuevo vector con los índices del vector de entrada cuyos elementos son distintos de cero. En caso de que todos los elementos sean nulos, devuelve un vector vacío. Así pues, si el vector devuelto por el comando *find* está vacío no se produce ningún choque en la trayectoria, y si no es así, el robot colisiona en algún punto de la ruta. De este modo se verifica si una unión entre dos nodos está o no libre de colisiones.

4.4. Implementación del algoritmo PRM

Para programar este método en Matlab se crea un *script* donde, en primer lugar, se indica el número de nodos aleatorios que se van a generar. En segundo lugar, se declaran los nodos inicial y final, entre los que se va a buscar una trayectoria. Seguidamente, se representa el entorno con los obstáculos presentes en él y se añaden al gráfico los puntos cartesianos correspondientes a los nodos de partida y de destino. Para esto último se utilizan dos funciones creadas con anterioridad. La primera de ellas, denominada *Cinemática_Directa*, recibe los valores articulares del robot en radianes y devuelve la matriz de transformación entre el sistema de la base y el del extremo. Puesto que esta función recibe los datos en radianes, pero se han definido en grados, es necesario realizar una conversión. Para tal efecto se ha implementado la función *DegToRad*, que tiene por entrada un dato en grados y lo devuelve en radianes. La segunda función empleada, *DibujarPuntos6D*, dibuja la posición y orientación del efector final en el espacio cartesiano. Para ello, necesita la matriz de transformación, la longitud de los ejes que va a dibujar para representar la orientación, el símbolo con el que se representa el punto (asterisco, punto, círculo, cuadrado, etc.) y el grosor que se desea utilizar.

La Figura 40 muestra un ejemplo de la representación del entorno, de un nodo inicial y un nodo final. En este caso, se ha utilizado un asterisco para el punto inicial y un cuadrado para el final. Para cada nodo, se representan las poses (posición y orientación) del sistema de referencia del extremo del robot. El eje rojo es x_6, el verde, y_6, y el azul, z_6.

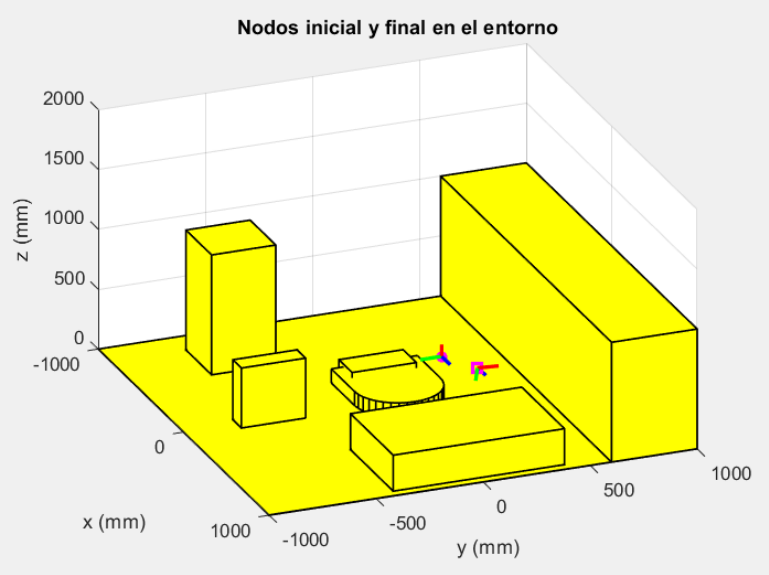


Figura 40. Representación del entorno con los nodos inicial y final para el algoritmo PRM

A continuación, se generan tantos nodos aleatorios como se indique al comienzo del *script*. Se ha creado para este fin la función *generar_puntos_aleatorios_q*, que tiene por entradas el número de puntos que se desea generar y dos vectores, de seis elementos cada uno, con los

valores mínimos en el primer caso y máximos en el segundo que pueden tomar las articulaciones. El resultado es una matriz de seis columnas y tantas filas como nodos se generen.

Después, se llama a la función *Delimita_Cfree*, que tiene por entradas la matriz de puntos aleatorios y todos los obstáculos ampliados para los distintos puntos tomados a lo largo de robot con el fin de verificar las colisiones. Los nodos se clasifican según estén en C_{free} o en C_{obs}. Para averiguar si un nodo está en colisión o no usa la función *Colision_Configuracion_Obstaculos*, explicada en el apartado 4.3. Este método tiene dos salidas: *Ptos_free*, que es una matriz con los nodos en C_{free}, y *Ptos_Obs*, que es una matriz con los nodos en C_{obs}.

Se procede entonces a representar los puntos en C_{free} obtenidos. Para ello, se recorren las filas de la matriz *Ptos_free* y, se utilizan nuevamente las funciones *Cinemática_Directa*, para pasar del espacio de configuraciones al cartesiano, y *DibujarPuntos6D* para graficar los puntos. En este caso, estos se dibujan como círculos. La Figura 41 muestra un ejemplo de la representación descrita.

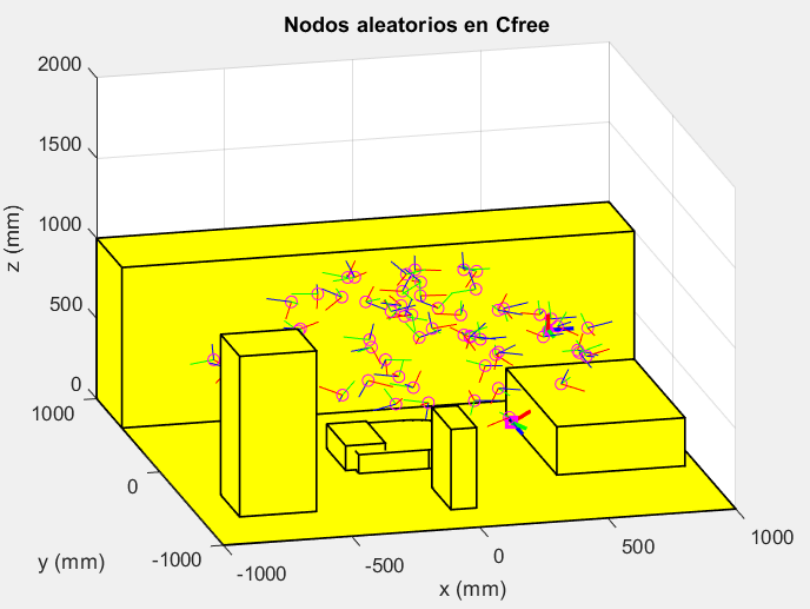


Figura 41. Representación del entorno con los nodos aleatorios generados mediante el algoritmo PRM

Tras obtener los nodos aleatorios y descartar los que se hallan en colisión, se utiliza la función *generar_rectas_matriz_costes*, que tiene por entradas la matriz *Ptos_free* y los obstáculos ampliados. Su papel consiste en recorrer todos los puntos y tratar de unirlos con los demás. En este caso, en lugar de tratar de unir los nodos con sus *k* vecinos más cercanos, se opta por tratar de unir todos con todos, lo que permite obtener rutas con menor coste, aunque conlleva un mayor tiempo de ejecución. Para cada pareja de puntos, comprueba, mediante la función *Colision_Curva_MoveAbsJ_Obstaculos*, si existe o no un impacto en la trayectoria que une dos nodos. Si no se produce colisión, se añade la recta a una matriz llamada *rectas* que contiene las parejas de puntos que pueden unirse. Además, se crea una matriz de costes. Se trata de una matriz cuadrada con tantas filas y columnas como nodos. En cada casilla se guarda el coste de

unir el nodo correspondiente a la fila con el correspondiente a la columna. Si no pueden unirse debido a la presencia de obstáculos, su valor es infinito. La matriz se inicializa con todos sus elementos iguales a infinito y si al recorrer las parejas de puntos se determina que una unión es posible se sustituye el valor de la fila y columna que corresponda por el coste de la recta entre los nodos en el espacio de configuraciones. Cabe destacar que la matriz de costes es simétrica respecto de su diagonal, lo que reduce los cálculos a la mitad.

El cálculo del coste entre nodos se calcula como la distancia euclídea en el espacio de configuraciones de seis dimensiones entre ellos. En este trabajo, se asigna además una ponderación distinta a la variación de cada una de las articulaciones, ya que no todas ellas contribuyen de igual modo al movimiento en el espacio cartesiano del extremo. Así, mientras que los movimientos de los tres primeros grados de libertad afectan en gran medida a la posición del extremo, los cambios de las tres últimas articulaciones provocan variaciones más pequeñas. Concretamente se ha aplicado un factor igual a 5 para q_1, 4 para q_2, 3,5 para q_3, 0,5 para q_4, 0,25 para q_5 y 0 para q_6. Se desprecia la última articulación en el cálculo del coste, ya que esta no afecta a la posición del extremo, sino solo a su orientación y, por ser el último eslabón cilíndrico, no influye en las colisiones.

Hasta aquí llega la primera fase del método, la de aprendizaje. Se pasa ahora a la fase de consulta, en la que, en primer lugar, se añaden los nodos inicial y final al mapa.

Para añadir el primer nodo, se ordenan los puntos en C_{free} de menor a mayor coste respecto al nodo inicial. Con este fin se utiliza la función *Ordenar_ptos_6D*, cuyas entradas son un punto y una lista de puntos. Calcula los costes de la ruta entre el punto y todos los puntos de la lista y devuelve un vector con los puntos de la lista ordenados de menor a mayor coste, otro vector con los costes también en orden y un tercer vector con los índices de los elementos del vector de entrada ordenados también por distancia. Para la obtención de los costes se utiliza también la ponderación. Pasando como parámetros el nodo inicial y los puntos contenidos en *Ptos_free* se ordenan los nodos de menor a mayor coste desde el nodo inicial.

Una vez se tienen los nodos en C_{free} en orden por coste hasta el nodo de partida, se recorren los puntos ordenados de menos a más coste hasta poder unir el nodo de inicio con alguno de ellos sin colisionar o hasta recorrer todos los nodos. Si se halla una conexión posible, se guarda el valor del índice del nodo con el que se ha unido. Del mismo modo se procede con el nodo final.

En este momento, se utiliza la función *Dijkstra* para obtener el camino óptimo mediante el algoritmo que lleva el mismo nombre. Se le pasan como entradas la matriz de costes y los índices de los nodos con los que se han unido el inicial y el de destino. Dado que los nodos de partida y meta ya se han unido al de menor coste de los generados en la fase de aprendizaje, esta función tomará como punto de partida aquel que se ha unido al nodo inicial y como nodo objetivo, aquel que se ha unido al final. Se devuelve el camino como una sucesión de números, que son los índices de los nodos que componen la ruta, y el coste de la trayectoria encontrada. Si el valor del coste es infinito, significa que no se ha encontrado un trayecto, lo cual se indica con un mensaje de error. Al coste devuelto por esta función se le suman los costes de añadir los nodos inicial y final al mapa.

Si se ha hallado una ruta, se procede a representarla. En primer lugar, se recorren los nodos del camino y se vuelven a graficar como círculos con bordes gruesos, para destacarlos sobre el resto de los nodos. Acto seguido, se recorren los nodos de la trayectoria excepto el último y se dibuja la trayectoria entre el nodo correspondiente y el siguiente. Esto último se lleva a cabo utilizando la función *dibujar_trayectoria*, cuyas entradas son dos nodos en el espacio de configuraciones. Dicha función, calcula la trayectoria para pasar de un nodo a otro mediante *moveAbsJ*, obtiene la ruta del extremo en coordenadas cartesianas a través de la cinemática directa y la dibuja. La Figura 42 muestra un ejemplo de una trayectoria obtenida mediante el algoritmo PRM.

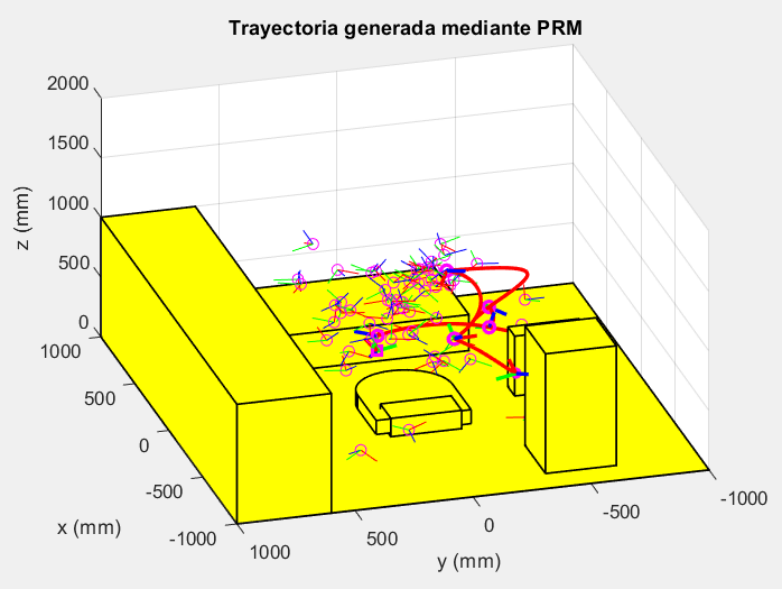


Figura 42. Representación de una trayectoria generada mediante el algoritmo PRM

4.5. Implementación del algoritmo RRT básico

Para desarrollar este método se han creado dos *scripts* en Matlab. En el primero de ellos, *inicializar_RRT*, se representa el entorno con sus obstáculos y se declaran las variables necesarias empezando por el árbol que se pretende crear y siguiendo con los valores articulares máximos y mínimos. Para los nodos, se ha definido una clase, llamada *nodo_conf_6D*, cuyas propiedades son su número, nombre, el número del nodo predecesor, el nodo predecesor, los seis valores articulares y su coste. De vuelta al *script* para inicializar, se declaran los nodos inicial y final como objetos de la clase *nodo_conf_6D*. El nodo inicial tiene por nombre "inicio", por número el cero y su coste es nulo. Se definen además sus valores articulares. El nodo final se llama "objetivo" y de declaran también sus valores para las seis articulaciones.

Utilizando de nuevo las funciones *Cinematica_Directa* y *DibujarPuntos6D*, ya explicadas en el apartado 4.4, se representan las configuraciones inicial y final. Los sistemas representan la pose del extremo. Se ha elegido el símbolo del asterisco para el inicio y un cuadrado para el objetivo. Se muestra un ejemplo del gráfico obtenido en la Figura 43.

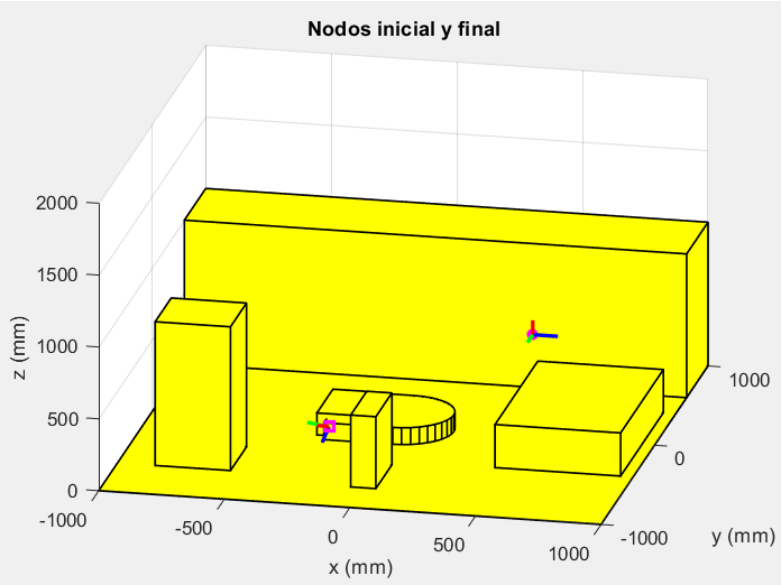


Figura 43. Representación del entorno con los nodos inicial y final para el algoritmo RRT

A continuación, se crea una lista con los puntos, que en un principio tan solo contiene los valores articulares del nodo inicial. Se establece un número máximo de iteraciones, el valor de la distancia entre nodos, el valor del parámetro T para la función *moveAbsJ* dentro de *Trayectoria_xyz* para comprobar las colisiones, y el valor del sesgo que se utiliza en la generación de los nodos aleatorios. El sesgo es un porcentaje de la configuración de destino que se suma a las coordenadas de los nodos aleatorios para encaminarlos hacia la meta. Nuevamente, T no establece un tiempo de simulación, pues en este caso no se está simulando, sino que determina

el número de puntos en los que se muestrea la trayectoria para verificar si colisiona. Finalmente, se añade el nodo de inicio al árbol.

El segundo *script* es el que genera el árbol de configuraciones, y lleva por nombre *crear_arbol_configuraciones_RRT*. En él, se entra en un bucle que se repite hasta que se alcance el número máximo de iteraciones o se llegue al nodo de destino.

Se llama a la función *crear_nodo_conf_RRT*, cuyos parámetros de entrada son el árbol, la longitud de los ejes de los puntos para la representación, la lista de puntos, la distancia entre nodos, el valor del parámetro T de *moveAbsJ*, los valores máximos y mínimos de las articulaciones, los valores articulares del nodo de destino, el sesgo y los obstáculos ampliados. La función entra en un bucle que se repite hasta que la configuración creada pueda añadirse al árbol sin dar lugar a una colisión. Se llama a la función *crear_configuracion_aleatoria_RRT_6D_OBS*, que se explicará más adelante, para crear un nodo aleatorio. Mediante la función *Ordenar_ptos_6D*, ya presentada en el apartado 4.4, se ordenan las configuraciones de la lista de puntos de menor a mayor distancia para unirlas con el nuevo nodo. Cabe destacar que se vuelve a emplear la ponderación descrita en 4.4 para las articulaciones en el cálculo de la distancia o coste entre nodos.

Se crea un nodo candidato a unirse al árbol en la recta entre el nodo aleatorio y el nodo más cercano de la lista, situado a la distancia establecida del nodo más próximo. Se comprueba, mediante la función *Colision_Curva_MoveAbsJ_Obstaculos*, si se produce colisión en la ruta entre el candidato a nuevo nodo y el más cercano. En caso negativo, se devuelve el nodo candidato, se guarda el más cercano como su predecesor y se calcula su coste como la suma del coste de su predecesor y el del camino entre este último y el nuevo nodo. Se representa también el nuevo nodo mediante *Cinematica_Directa* y *DibujarPuntos6D* utilizando círculos, y la unión entre los nodos con la función *dibujar_trayectoria*. Si en cambio existe colisión, se busca en la lista de nodos ordenados por distancia el siguiente más cercano y se trata nuevamente de unirlo, repitiendo el proceso hasta que se logre una conexión sin obstáculos o se hayan recorrido todos los nodos. Si sucede esto último se genera un nuevo nodo y se repite el procedimiento.

La función *crear_configuracion_aleatoria_RRT_6D_OBS*, mencionada anteriormente, recibe los valores articulares máximos y mínimos, el sesgo, el objetivo y los obstáculos ampliados. Con estos datos, crea una configuración aleatoria con el sesgo determinado hacia el objetivo. A continuación, comprueba, mediante la función *Colision_Configuracion_Obstaculos* si la configuración generada está o no en colisión y, si es así, repite el procedimiento hasta obtener una configuración en C_{free}.

Prosiguiendo con el *script* que crea el árbol, una vez obtenido el nuevo nodo, se añade el nodo al árbol y sus valores articulares, a la lista de puntos. Finalmente, se comprueba si la distancia entre el nuevo nodo y el final es menor que la estipulada. En caso de que así sea, comprueba mediante *Colision_Curva_MoveAbsJ_Obstaculos* si se produce o no colisión al unir el nodo creado con la meta y, si no se produce un choque, se añade el nodo final al árbol, se guarda el último nodo creado como su predecesor, se calcula el coste del nodo final y se representa la ruta entre ambos nodos. Tras alcanzar el nodo de destino finaliza el proceso.

Después, ya fuera del bucle, en caso de que se haya alcanzado el nodo de destino antes de llegar al número máximo de iteraciones, se calcula el camino mediante la función *Trayectoria_RRT_6D*. Esta última recibe el nodo de partida y el de destino y recorre el árbol del final al inicio leyendo los nodos predecesores para devolver así la sucesión de nodos que llevan del primer nodo al último.

A continuación, se vuelven a representar los puntos que componen la trayectoria con un grosor mayor, y las rutas entre nodos en color rojo y también con más grosor, para resaltar la trayectoria sobre el resto del árbol. La Figura 44 muestra un ejemplo de una trayectoria obtenida mediante este método.

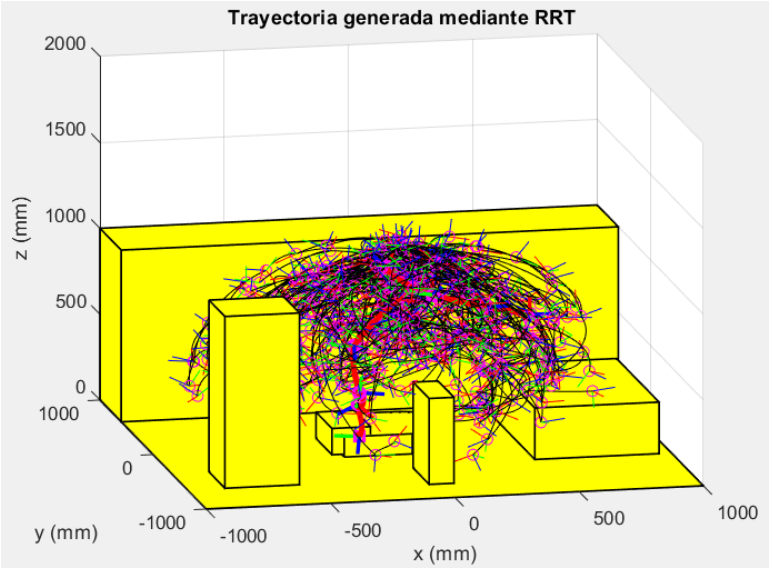


Figura 44. Representación de un árbol y una trayectoria generados mediante el algoritmo RRT

Se dibuja nuevamente la trayectoria en un nuevo gráfico, sin el resto de los nodos, para poder apreciarla mejor, como se ve en la Figura 45.

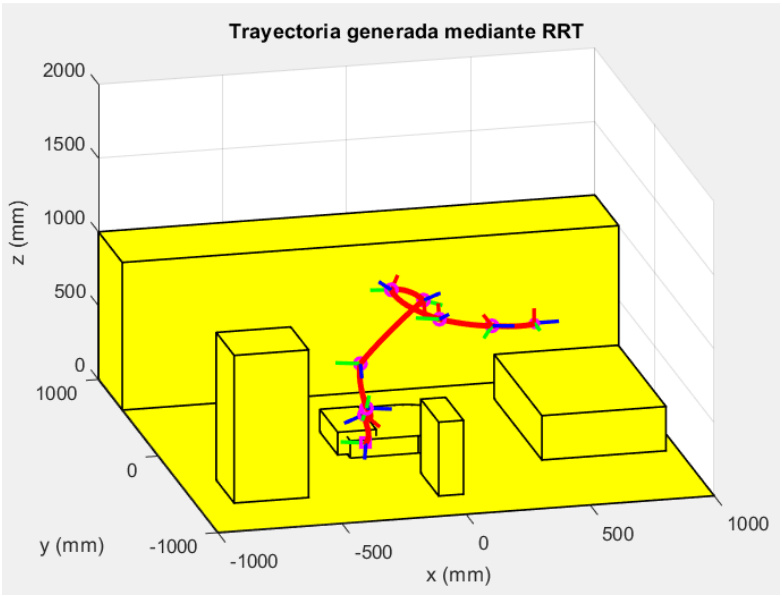


Figura 45. Representación de una trayectoria generada mediante el algoritmo RRT

4.6. Implementación del algoritmo RRT*

Este algoritmo se desarrolla en dos *scripts*, al igual que ocurre con el método RRT básico. El primero de ellos, se utiliza para inicializar las variables y el entorno. Se denomina *inicializar_RRT_estrella*, y es exactamente igual que el programa creado para inicializar el algoritmo RRT básico. En cuanto al segundo fichero, llamado *crear_arbol_configuraciones_RRT_estrella* es análogo al que crea el árbol de RRT básico con la única diferencia de que ahora la función que crea el nuevo nodo es *crear_nodo_conf_RRT_estrella*. Es en esta función donde reside la diferencia con el procedimiento anterior.

La función recibe las mismas entradas que la creada para generar los nodos con RRT básico, es decir, el árbol, la longitud de los ejes de los puntos para la representación, la lista de puntos, la distancia máxima entre nodos, el valor del parámetro T de *moveAbsJ*, los valores máximos y mínimos de las articulaciones, los valores articulares del nodo de destino, el sesgo y los obstáculos ampliados.

La función entra en un bucle que se repite hasta que la configuración creada pueda añadirse al árbol sin dar lugar a una colisión. La creación del nodo candidato a añadirse es igual que en RRT, es decir, la función *crear_configuracion_aleatoria_RRT_6D_OBS* genera un nodo aleatorio y, mediante la función *Ordenar_ptos_6D*, se ordenan las configuraciones de la lista de puntos de menor a mayor distancia para unirlas con el nuevo nodo. Se usa, por supuesto, la ponderación para el cálculo de esta. Después, se crea un nodo candidato a unirse al árbol en la recta entre el nodo aleatorio y el nodo de la lista que tiene menor coste hasta él.

Ahora, la función varía con respecto a la anterior. En vez de intentar unir el nodo candidato con el más cercano, se vuelve a utilizar la función *Ordenar_ptos_6D*, esta vez para ordenar los nodos según su distancia hasta el nodo candidato, y no hasta la configuración aleatoria.

Se recorre el vector con los nodos de menos a más distancia al candidato hasta que esta supere el valor de un radio establecido. Se busca de entre los nodos situados dentro del radio el de menor coste total desde el inicio que pueda unirse al candidato sin colisionar, lo cual se comprueba mediante la fusión *Colision_Curva_MoveAbsJ_Obstaculos*. Una vez hallado tal nodo, se guarda como el predecesor del candidato.

Es importante destacar que, en este caso, el coste que se revisa no es el existente entre el nodo candidato y el nodo del árbol con el que se conecta, sino el coste total para llegar a los nodos cercanos al candidato desde el nodo inicial. Es decir, el radio hace referencia a la distancia entre el candidato y sus vecinos, pero el coste total cuyo mínimo se busca, se refiere a la distancia total que debe recorrerse en el árbol para llegar del inicio a los nodos vecinos al candidato.

Finalmente, se devuelve el candidato, se guarda su predecesor, se calcula su coste y se representan tanto el nodo como la trayectoria entre el punto anterior y él.

Las Figuras 46 y 47 permiten visualizar la generación de trayectorias mediante este método.

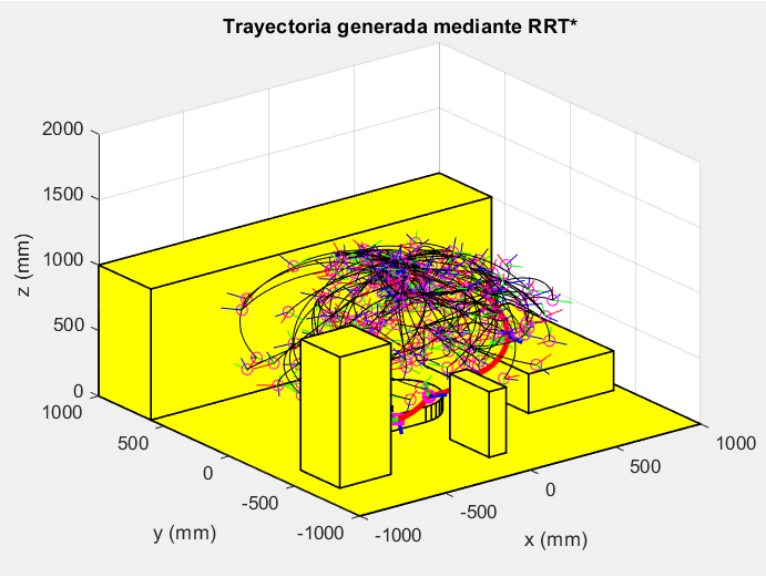


*Figura 46. Representación de un árbol y una trayectoria generados mediante el algoritmo RRT**

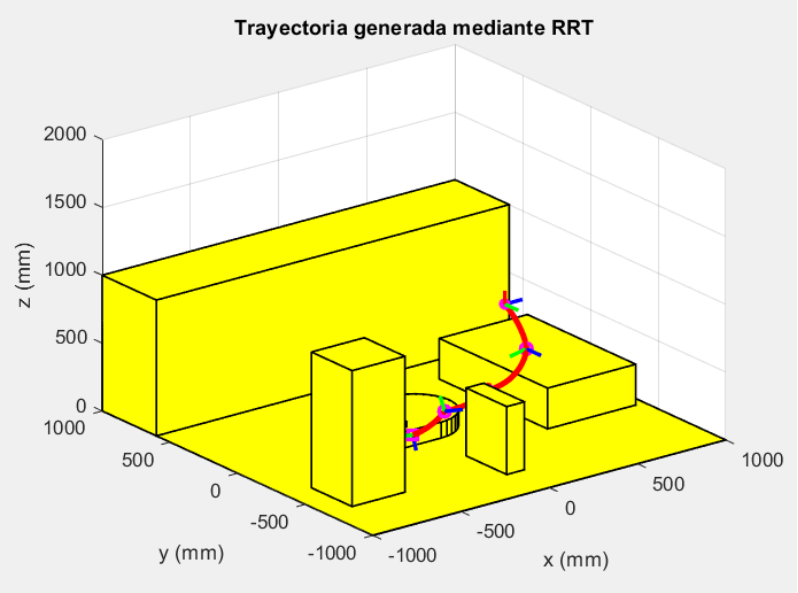


*Figura 47. Representación de una trayectoria generada mediante el algoritmo RRT**

Puede observarse cómo la Figura 47 muestra un camino mucho menos costoso que el de la Figura 45, obtenido con RRT básico, que zigzaguea mucho más dando lugar a un coste superior. Comparando estas dos figuras se observa cómo la variante RRT* reduce el coste de la trayectoria.

4.7. Implementación de la simulación en Simscape Multibody

Para la simulación de las trayectorias, se genera un *script* en el que, en primer lugar, se declaran seis matrices inicialmente vacías, una para cada articulación. En segundo lugar, se asigna al parámetro T, un valor de 5 segundos, que será la duración de la trayectoria entre cada nodo y el siguiente.

Después, comienza un bucle en el que se crea una trayectoria utilizando la función *moveAbsJ* entre un nodo y el siguiente. El bucle se repite para todos los nodos desde el primero hasta el penúltimo. En cada iteración, se obtienen seis matrices de dos columnas, devueltas por *moveAbsJ*, donde la primera columna contiene los valores de tiempo, con un incremento de 5 ms ente ellos, y la segunda, el valor de la articulación correspondiente para cada instante.

Estas matrices se añaden como nuevas filas a las matrices creadas primeramente vacías antes del bucle. Además, en cada iteración, se incrementa en uno el valor del parámetro *num* de *moveAbsJ*, que determina el tiempo inicial y que, en la primera iteración vale cero. De este modo, en cada iteración, el tiempo de inicio de la nueva trayectoria es un periodo posterior al de la repetición anterior. Esto conlleva que, la trayectoria entre los nodos dos y tres comienza en tiempo igual a 5 segundos, tiempo que tarda en simularse el camino entre el nodo uno y el dos. El trayecto entre el nodo tres y cuatro empieza a los diez segundos y así sucesivamente. De este modo, se concatenan las trayectorias entre nodos, de tal forma que el resultado final al acabar el bucle es una única matriz para cada articulación. La primera columna contiene los instantes de tiempo de 5 en 5 ms empezando por cero y acabando en un tiempo igual al periodo de cada trayectoria (5 segundos) multiplicado por el número de trayectorias entre nodos, es decir, el número de nodos menos una unidad. La segunda columna contiene todos los valores articulares de la trayectoria desde el nodo inicial hasta el final pasando por todos los nodos intermedios de la ruta creada.

Finalmente, mediante la función *datosBrazo* se cargan los datos del archivo *DataFile* y se llama a la simulación en Simulink que se describe a continuación y que automáticamente abre Simscape Multibody.

Por otro lado, en Simulink, se ha tomado el modelo del brazo importado en Simscape debidamente modificado, para permitir la simulación, como se detalla en el apartado 4.2. Se añaden, además, nuevos sólidos, precedidos de las correspondientes translaciones, para visualizar también los obstáculos. Además, se modifican los bloques de las articulaciones para que se les pueda indicar su valor a través de una entrada. Tras seleccionar esta opción, los bloques muestran una nueva entrada llamada q. Esta requiere de una matriz con una columna de instantes de tiempo y otra de valores articulares, por lo que se ha organizado la información en el *script* de este modo, como ya se ha explicado.

A continuación, se añaden al esquema seis bloques del tipo *From Workspace*, y en cada uno de ellos se establece como dato una de las matrices creadas en el script con todos los valores de tiempo y articulaciones para cada junta de revolución. Cada uno de estos bloques se conecta con la nueva entrada del modelo de la articulación correspondiente colocando entre ellos un

convertidor *Simulink-PS*, que convierte una señal de Simulink en una señal física. La Figura 48 muestra el esquema generado en Simulink.

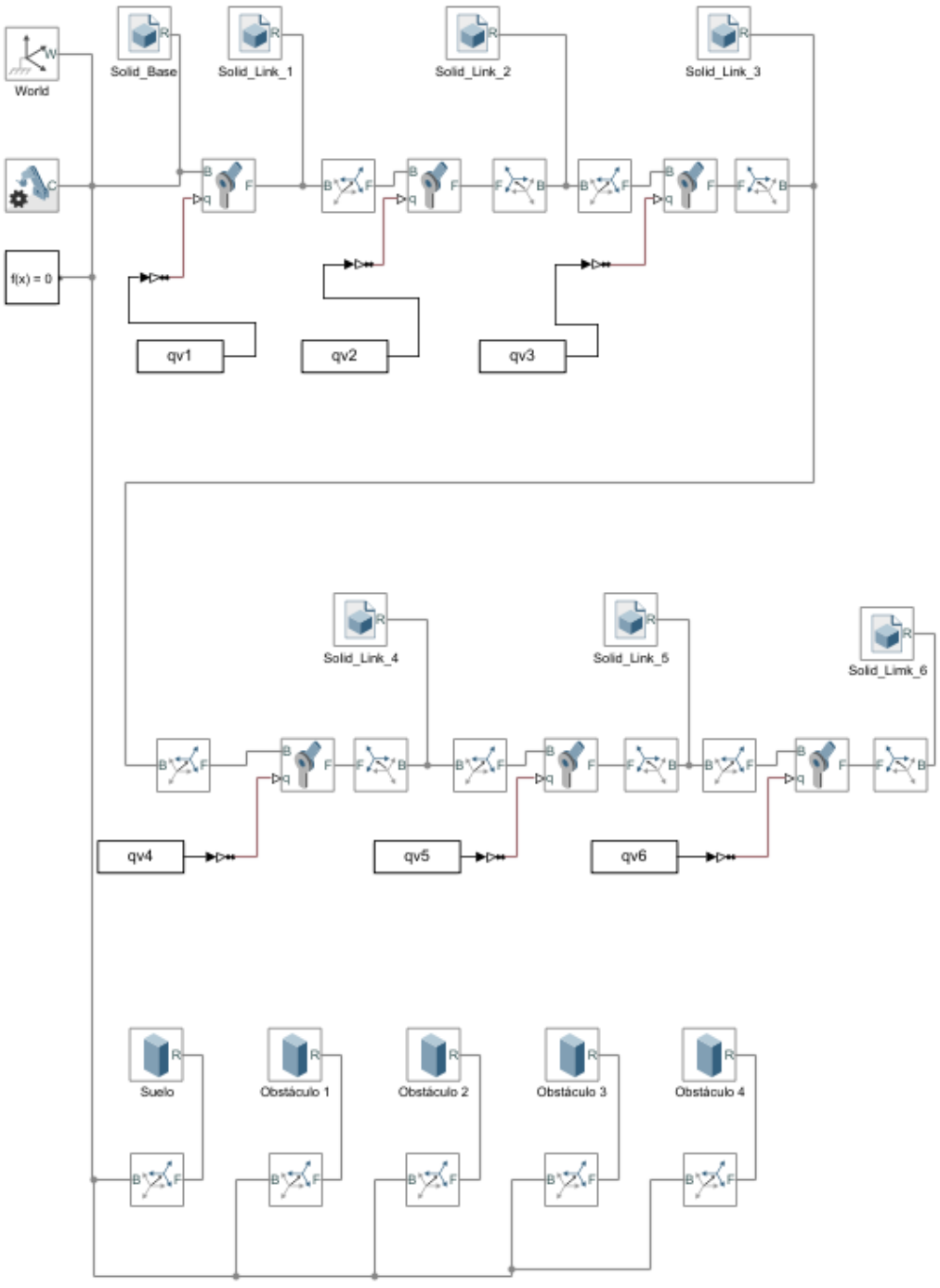


Figura 48. Modelo en Simulink para la simulación de las trayectorias generadas

5. Resultados

En este apartado se muestran y se comentan los resultados obtenidos mediante los algoritmos implementados. Se realiza además un estudio comparativo de ellos resaltando las ventajas e inconvenientes de una técnica frente a la otra.

5.1. Resultados obtenidos mediante PRM

Con el fin de analizar los resultados obtenidos mediante la generación de trayectorias con el método PRM se realizan una serie de ejecuciones variando el número de puntos generados. Para las distintas pruebas llevadas a cabo con diferentes cantidades de nodos se estudia la probabilidad de éxito, el coste del camino obtenido y el tiempo de ejecución que se requiere. La probabilidad de éxito se calcula como el cociente entre las pruebas exitosas y las totales, y el coste de la trayectoria en grados es el devuelto por la función Dijkstra al que se le suman los costes de añadir los nodos inicial y final al mapa. Para todas las pruebas se utilizan un mismo nodo inicial y final, y el mismo entorno, con los obstáculos mostrados en el apartado 4.3.

A continuación, las Tablas 2 y 3 muestran los resultados obtenidos mediante el algoritmo PRM con distintas cantidades de nodos aleatorios generados. Para cada número de puntos se realizan veinte ejecuciones.

Número de nodos					
25		50		100	
Coste (°)	Tiempo (s)	Coste (°)	Tiempo (s)	Coste (°)	Tiempo (s)
1370,40	2,19	1190,20	6,38	1313,60	16,77
Fallo	1,67	1217,50	3,66	916,38	19,90
1286,60	1,75	925,63	3,43	890,31	18,38
1512,50	1,59	1318,20	5,80	803,39	18,40
Fallo	1,37	1064,10	3,06	896,54	21,00
706,78	0,95	919,08	5,72	1249,80	17,33
837,67	1,50	841,18	4,11	1072,00	16,68
Fallo	0,95	1146,00	3,86	974,36	15,34
Fallo	1,10	797,56	2,72	1108,90	19,21
775,05	1,24	1180,80	4,54	856,34	17,00
Fallo	1,43	1118,90	4,07	1057,90	17,09
Fallo	1,73	Fallo	4,97	830,50	22,21
Fallo	1,64	883,65	6,23	747,04	28,11
Fallo	1,81	1000,20	6,96	1365,30	24,67
1348,80	1,00	1059,10	5,34	1012,60	17,77
1335,80	1,21	Fallo	5,38	861,43	19,57
1328,60	0,78	1039,60	4,98	902,04	22,21
Fallo	2,29	1232,50	4,70	1110,70	16,26
814,57	0,63	846,83	5,47	823,61	20,41
Fallo	1,40	882,74	6,08	1040,80	20,29

Tabla 2. Resultados obtenidos mediante PRM con 25, 50 y 100 nodos

Número de nodos			
150		200	
Coste (°)	Tiempo (s)	Coste (°)	Tiempo (s)
682,88	43,37	622,53	84,75
719,71	47,99	969,85	84,38
711,56	54,94	1111,60	78,64
1110,00	44,89	947,73	69,30
788,25	50,51	855,25	67,97
707,71	39,55	941,85	84,95
918,31	39,43	703,41	80,85
905,52	34,53	888,12	62,63
1081,70	32,99	752,53	78,53
815,35	41,05	802,30	109,85
1266,40	38,88	973,39	85,51
890,31	38,88	681,65	83,23
896,54	44,61	642,25	82,12
1245,00	43,51	927,08	78,68
1092,10	43,82	1118,60	67,38
1052,00	43,11	739,30	87,41
837,84	58,34	923,58	95,32
1012,40	34,77	645,16	86,99
770,36	41,00	879,74	72,01
691,17	38,83	915,95	84,05

Tabla 3. Resultados obtenidos mediante PRM con 150 y 200 nodos

Con los datos obtenidos, se calcula la probabilidad de éxito y los valores promedio del coste (sin tener en cuenta los fallos) y de tiempo para cada cantidad de puntos generados. Estos resultados figuran en la Tabla 4.

Número de nodos	Probabilidad de éxito (%)	Coste medio (°)	Tiempo medio (s)
25	50,00	1131,68	1,41
50	90,00	1036,88	4,87
100	100,00	991,68	19,43
150	100,00	909,76	42,75
200	100,00	852,09	81,23

Tabla 4. Valores medios obtenidos mediante PRM variando el número de nodos

Los tres resultados calculados pueden representarse de forma gráfica frente al número de nodos como se aprecia en los Gráficos 1, 2 y 3.

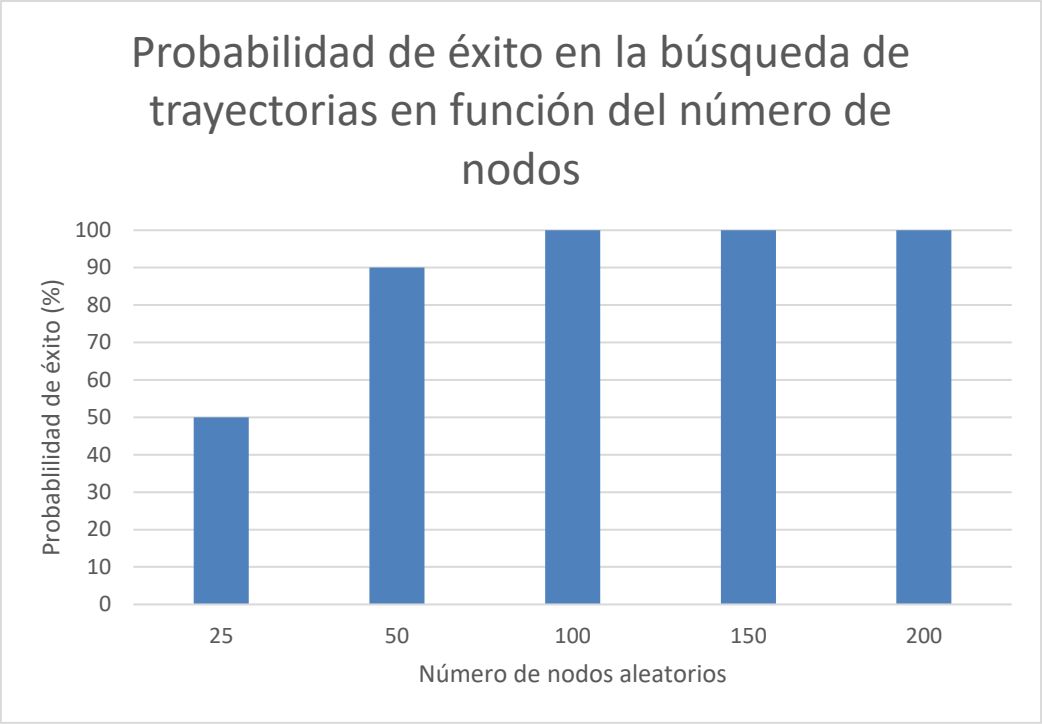


Gráfico 1. Probabilidad de éxito en la búsqueda de trayectorias para distintas cantidades de nodos con el algoritmo PRM

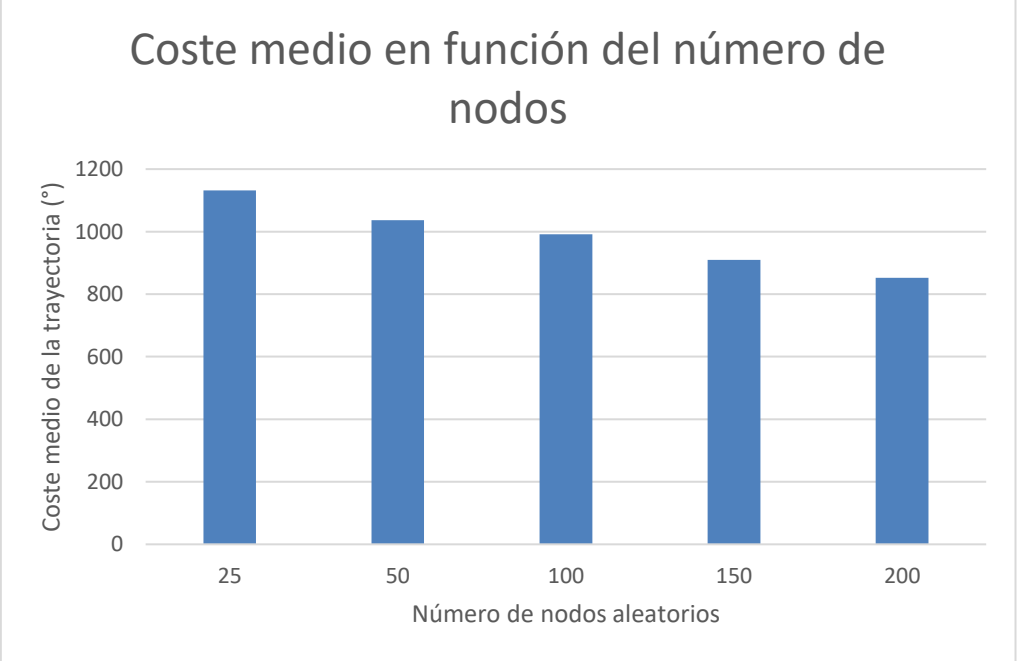


Gráfico 2. Coste medio de la trayectoria en función del número de nodos con el algoritmo PRM

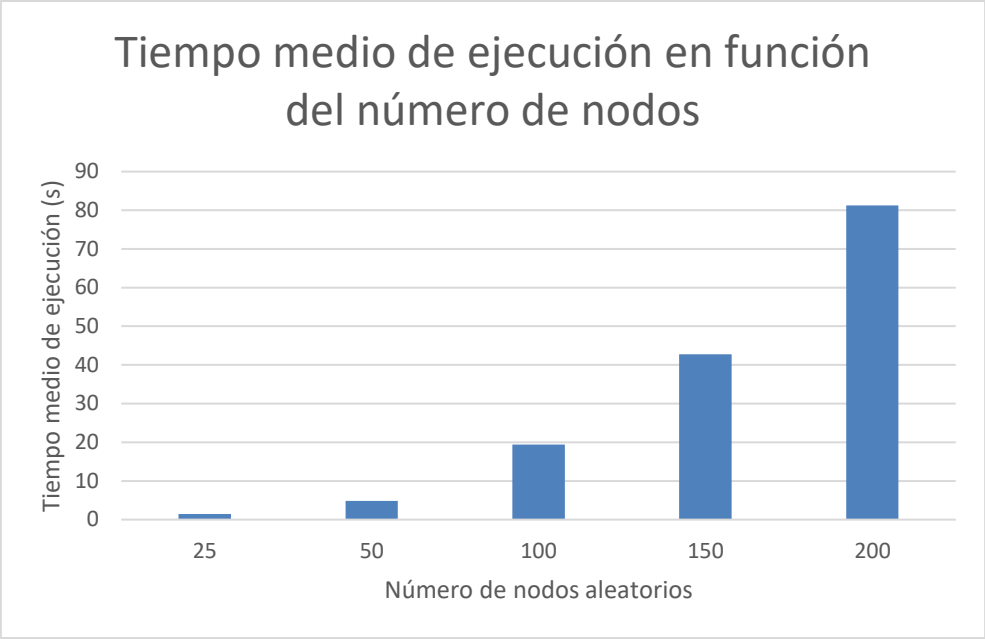


Gráfico 3. Tiempo medio de ejecución en función del número de nodos con el algoritmo PRM

Tal y como muestran los valores obtenidos, para un número pequeño de nodos generados, como 25, no está garantizado que se encuentre un camino entre los nodos inicial y final. Ello es debido a que, los pocos nodos generados pueden ubicare en el espacio de configuraciones de tal forma que ninguno de ellos sea alcanzable desde el nodo inicial o desde el final sin interferir con los obstáculos. También puede darse el caso de que, aun pudiendo unir a algún otro nodo tanto el inicial como el final, no sea posible llegar de uno a otro a través del mapa creado en la fase de aprendizaje. Como es lógico, un mayor número de nodos disminuye las posibilidades de que estas situaciones de bloqueo se den, puesto que más nodos implican más conexiones que se pueden intentar, lo que aumenta las probabilidades de que alguna de ellas sea posible. Así, según el Gráfico 1, con 25 nodos, la mitad de las pruebas fracasa; mientras que con 50, la probabilidad de éxito asciende hasta el 90 %. Para todas las ejecuciones llevadas a cabo con 100 nodos o más, se encuentra una trayectoria en la totalidad de los casos.

Por otro lado, en el Gráfico 2, se aprecia una mejora del coste de la ruta conforme aumenta el número de nodos aleatorio, ya que un mayor número de nodos abarca más configuraciones, lo que contribuye al aumento de las posibilidades de dar con una ruta que se acerque más a la óptima.

Finalmente, como es natural, el coste de ejecución aumenta siguiendo una curva cuya pendiente es creciente, como se observa en el Gráfico 3. Ello es debido a que, para cada nodo debe evaluarse su posible conexión con el resto. El número de comprobaciones es igual al sumatorio de todos los números entre la unidad y el número de nodos menos uno; según la igualdad:

$$Número\ de\ comprobaciones = \sum_{i=1}^{N-1} i$$

donde *N* es el número de nodos en C_{free}.

Dado que esta expresión no aumenta de manera lineal, sino que su pendiente crece con el incremento del número de nodos, del mismo modo, el tiempo de ejecución aumenta con el número de nodos de manera no lineal.

5.2. Resultados obtenidos mediante RRT básico

El análisis de los resultados obtenidos mediante el algoritmo RRT se aborda a través de dos comparaciones. Por un lado, se realizan veinte ejecuciones con distintos valores del sesgo para estudiar su efecto sobre el número de nodos necesarios para alcanzar la meta, así como el coste y tiempo de ejecución. El sesgo es un porcentaje del valor final al que se desea llegar que se añade al valor aleatorio generado con el fin de dirigir los puntos hacia ese destino. Generalmente, un pequeño sesgo contribuye a alcanzar el destino con menos nodos, aunque un sesgo excesivamente grande puede dar lugar a que muchos de los puntos aleatorios sobrepasen su valor dificultando llegar al objetivo. Por otra parte se analiza la influencia de la distancia.

Las Tablas 5 y 6 muestran los valores obtenidos en las ejecuciones llevadas a cabo para el análisis del efecto del sesgo. Los valores medios figuran en la Tabla 7. Para estas pruebas se ha elegido una distancia de 100°, y un valor de 0.5 segundos para el parámetro T, de manera que, en cada trayectoria se generen y comprueben 100 puntos, tantos como el valor de la distancia elegido.

Valor del sesgo (%)					
0			5		
Nodos	Coste (°)	Tiempo (s)	Nodos	Coste (°)	Tiempo (s)
784	881,68	16,82	268	1093,10	6,70
969	990,82	21,34	96	875,79	1,99
152	1098,90	3,27	356	957,65	7,23
977	1655,90	20,94	385	854,40	8,89
681	1798,40	13,23	197	1129,70	3,57
382	1170,50	7,43	176	886,38	3,20
970	1053,90	21,29	1015	1554,70	22,02
776	1154,00	15,87	682	1276,50	13,52
556	1156,60	10,81	999	1389,90	21,87
315	896,46	6,90	191	798,23	4,03
493	1270,40	9,34	852	1173,70	17,68
377	1198,30	6,93	750	1566,90	15,48
369	1598,00	6,79	604	2088,50	11,80
512	1672,50	9,63	20	1287,50	3,88
477	1177,60	9,03	180	984,57	3,30
606	1292,80	12,33	622	877,12	12,00
431	1665,50	8,20	385	1494,70	7,32
241	790,22	4,36	1399	1079,80	32,32
1426	1074,50	32,97	287	1258,20	5,37
122	1297,00	2,46	506	1346,70	10,26

Tabla 5. Resultados obtenidos mediante RRT básico con sesgos del 0 % y el 5 %

Valor del sesgo (%)					
10			15		
Nodos	Coste (°)	Tiempo (s)	Nodos	Coste (°)	Tiempo (s)
563	1321,70	14,38	1151	1386,80	27,60
32	1077,80	0,96	955	1272,60	20,78
644	1179,20	16,39	310	978,89	6,66
514	798,16	13,53	955	1066,50	19,45
808	1070,20	19,15	113	1495,90	1,94
723	1566,10	16,75	687	1174,30	13,60
632	2171,50	14,81	227	1499,00	4,18
952	1066,80	23,00	359	990,70	6,51
130	1282,60	2,33	820	1249,20	16,32
257	1293,40	4,79	883	1070,70	18,88
1363	1077,70	30,55	327	853,56	5,97
279	1464,00	5,31	813	1599,00	16,61
495	1451,80	9,50	685	1383,20	13,68
442	1092,90	12,76	1174	763,65	25,48
424	1283,20	9,32	516	1184,10	9,65
505	1092,10	10,99	1012	1145,00	20,91
248	1392,90	5,10	512	1097,40	9,72
1128	1154,40	25,83	276	1298,30	4,94
1434	922,26	40,16	1860	1775,90	44,47
715	1173,50	16,54	424	1477,20	9,73

Tabla 6. Resultados obtenidos mediante RRT básico con sesgos del 10 % y el 15 %

Sesgo (%)	Número medio de nodos	Coste medio (°)	Tiempo de ejecución medio (s)
0	580,80	1244,70	12,00
5	498,50	1198,70	10,62
10	614,40	1246,61	14,61
15	702,95	1238,09	14,85

Tabla 7. Valores medios obtenidos mediante RRT básico variando el sesgo

Los resultados de los nodos y del coste expuestos en la Tabla 7 aparecen representados en los Gráficos 4 y 5 para una mejor visualización de estos.

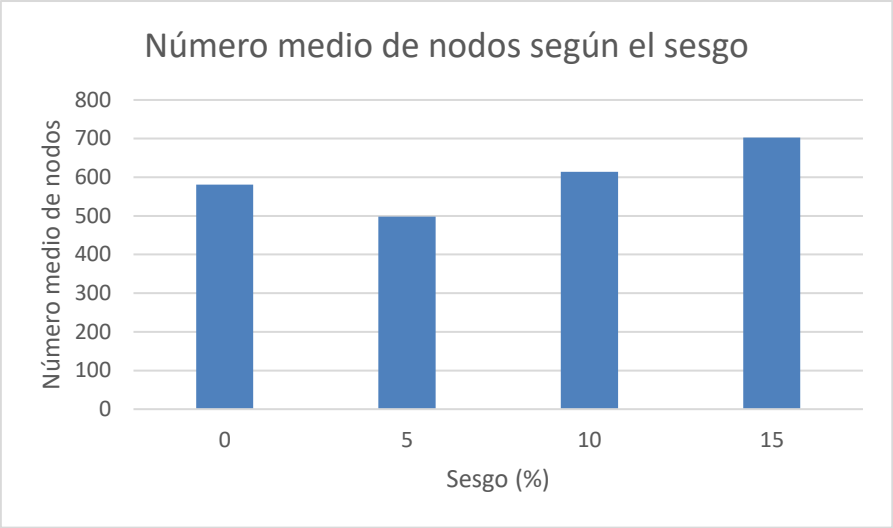


Gráfico 4. Número medio de nodos según el sesgo con el algoritmo RRT básico

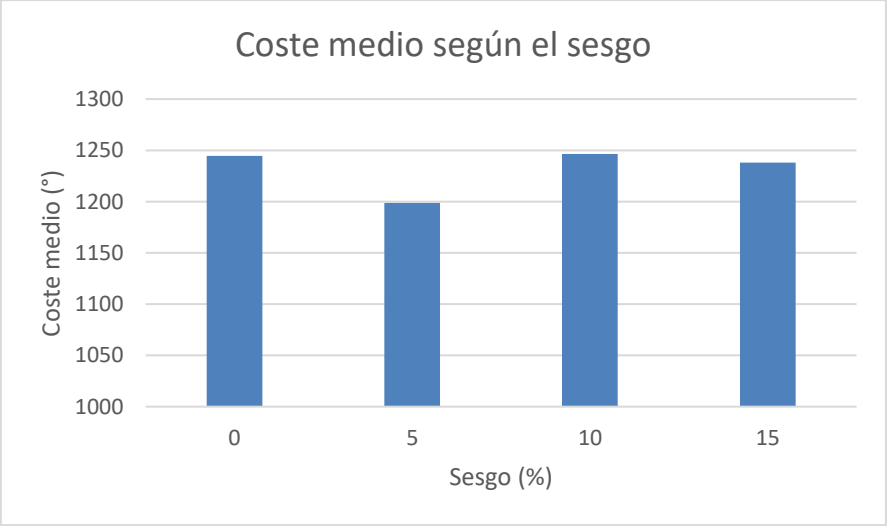


Gráfico 5. Coste medio de la trayectoria según el sesgo con el algoritmo RRT básico

Los resultados obtenidos, visibles en el Gráfico 4, muestran que un sesgo del 5 % reduce el número de nodos necesarios para alcanzar el nodo final en un 15 % aproximadamente. Sin embargo, un sesgo del 10 % dificulta la búsqueda de la meta, no solo con respecto al 5 % sino también con respecto a un sesgo nulo, o lo que es equivalente, a la aleatoriedad total. Con un sesgo del 15 %, se requiere también una cantidad media de nodos mayor.

En cuanto al coste, este no varía excesivamente al cambiar el sesgo, aunque sí puede decirse que es ligeramente más bajo con un sesgo del 5 % como indica el Gráfico 5. Esto se debe a que, con este sesgo, los nodos se encaminan hacia la meta, pero sin sobrepasarla como ocurre con sesgos mayores, lo que puede dar pie a un menor coste, ya que se dirige en cierta medida el árbol hacia el nodo deseado.

Por otro lado, el sesgo afecta al tiempo total, pero no al tiempo por nodo. Es decir, si el tiempo medio para un sesgo del 5 % es menor que en el resto de los casos, se debe únicamente a que el número de nodos es menor. Para demostrar esta afirmación se calcula el tiempo medio por nodo para cada uno de los sesgos, como el cociente entre el tiempo de ejecución medio y el número medio de nodos. Los resultados figuran en la Tabla 8.

Sesgo (%)	Tiempo de ejecución medio por nodo (ms/nodo)
0	20,7
5	21,3
10	23,8
15	21,1

Tabla 8. Tiempo medio de ejecución por nodo variando el sesgo con el algoritmo RRT básico

Los valores son similares e indican un tiempo de ejecución algo superior a 20 ms por nodo. Con ello queda demostrado que el sesgo solo influye en el tiempo de ejecución a través de sus efectos sobre el número de nodos.

Se procede ahora a estudiar el efecto de la distancia elegida sobre el número de nodos necesarios, el coste de la trayectoria y el tiempo de ejecución. Es importante destacar que se ha mantenido como criterio para la selección del parámetro T, que este sea tal que los puntos muestreados en la trayectoria sean iguales al valor de la distancia seleccionada. Ello implica que ahora, al variar la distancia y, por tanto, T, sí se producen cambios en el tiempo de ejecución por nodo, ya que en la verificación de colisiones a lo largo de la trayectoria se comprueban más o menos puntos según el coste mínimo que se establezca.

Se han realizado veinte ejecuciones con distancias de 100°, 200°, 300°, 400° y 500°, con valores para T de 0,5; 1; 1,5; 2 y 2,5, que dan lugar a 100, 200, 300, 400 y 500 puntos a lo largo de las trayectorias respectivamente. Todas las pruebas se han llevado a cabo utilizando un sesgo del 5 %, por ser el más rápido según el estudio anterior. Los resultados se recogen en las Tablas 9, 10 y 11, y los valores medios en la Tabla 12.

Distancia (°)					
100 (T = 0,5 s)			200 (T = 1 s)		
Nodos	Coste (°)	Tiempo (s)	Nodos	Coste (°)	Tiempo (s)
268	1093,1	6,704822	37	1157,60	1,93
96	875,7923	1,992992	28	1569,00	1,32
356	957,6549	7,23157	43	1582,50	1,57
385	854,3974	8,885939	30	1132,50	1,09
197	1129,7	3,565842	37	1179,40	1,55
176	886,3784	3,200553	45	1792,20	1,29
1015	1554,7	22,018854	62	1172,50	2,06
682	1276,5	13,524884	27	967,01	0,86
999	1389,9	21,873216	147	1278,90	5,29
191	798,2278	4,03294	100	1126,20	3,96
852	1173,7	17,676392	57	1094,30	1,94
750	1566,9	15,479112	45	940,29	1,63
604	2088,5	11,801426	75	1597,60	2,48
20	1287,5	3,87684	31	1198,50	1,00
180	984,5688	3,301529	18	1129,80	0,55
622	877,1189	12,001017	141	1979,40	5,31
385	1494,7	7,317602	60	1779,80	2,03
1399	1079,8	32,31755	73	1392,00	2,56
287	1258,2	5,370921	109	1572,30	3,43
506	1346,7	10,264918	32	1718,50	1,12

Tabla 9. Resultados obtenidos mediante RRT básico con distancias de 100° y 200°

Distancia (°)					
300 (T = 1,5 s)			400 (T = 2 s)		
Nodos	Coste (°)	Tiempo (s)	Nodos	Coste (°)	Tiempo (s)
42	1912,60	2,89	22	2242,50	2,65
10	1444,50	0,69	11	1908,10	2,02
69	2005,90	4,51	26	2656,40	2,93
16	798,45	1,67	54	3779,50	6,92
8	896,40	0,59	5	1576,90	0,90
9	1391,90	0,59	9	1910,00	1,61
146	2036,50	9,66	10	2347,30	1,01
18	1636,10	1,28	12	2209,30	1,24
21	1654,10	1,00	13	1929,30	1,51
81	1444,20	5,20	11	1494,00	1,09
29	2090,20	1,73	32	1543,00	2,27
22	2946,90	1,40	8	1335,30	1,52
35	1345,30	1,82	17	2379,70	1,59
7	1768,10	0,31	12	1833,80	0,81
104	2617,30	6,21	11	13331,00	1,17
18	2990,10	0,89	11	1969,20	1,54
69	1967,30	4,27	23	234,10	2,25
25	1766,50	1,66	14	1939,20	1,02
23	1460,60	1,42	44	2558,70	4,63
62	2073,90	4,75	15	1527,70	1,79

Tabla 10. Resultados obtenidos mediante RRT básico con distancias de 300° y 400°

Distancia (°)		
500 (T = 2,5 s)		
Nodos	Coste (°)	Tiempo (s)
15	2477,10	3,99
10	2376,80	1,79
19	2452,60	3,67
28	3247,30	4,73
16	1669,60	7,37
5	1771,20	2,50
29	3966,40	7,17
25	3280,90	5,83
29	1631,50	4,65
8	2497,30	0,95
7	1696,10	4,78
19	3224,20	3,42
17	2596,80	3,94
10	2962,10	2,14
14	3221,80	3,32
18	2758,10	3,62
18	2844,10	2,81
25	3849,60	4,38
57	3348,20	10,51
23	4856,40	5,22

Tabla 11. Resultados obtenidos mediante RRT básico con una distancia de 500°

Distancia (°)	Número medio de nodos	Coste medio (°)	Tiempo de ejecución medio (s)
100	498,50	1198,70	10,62
200	59,85	1368,01	2,15
300	40,70	1812,34	2,63
400	18,00	2535,25	2,02
500	19,60	2836,41	4,34

Tabla 12. Valores medios obtenidos mediante RRT básico variando la distancia

Los datos referentes al número medio de nodos y al coste medio de la Tabla 12 se presentan de forma mucho más visual en los Gráficos 6 y 7.

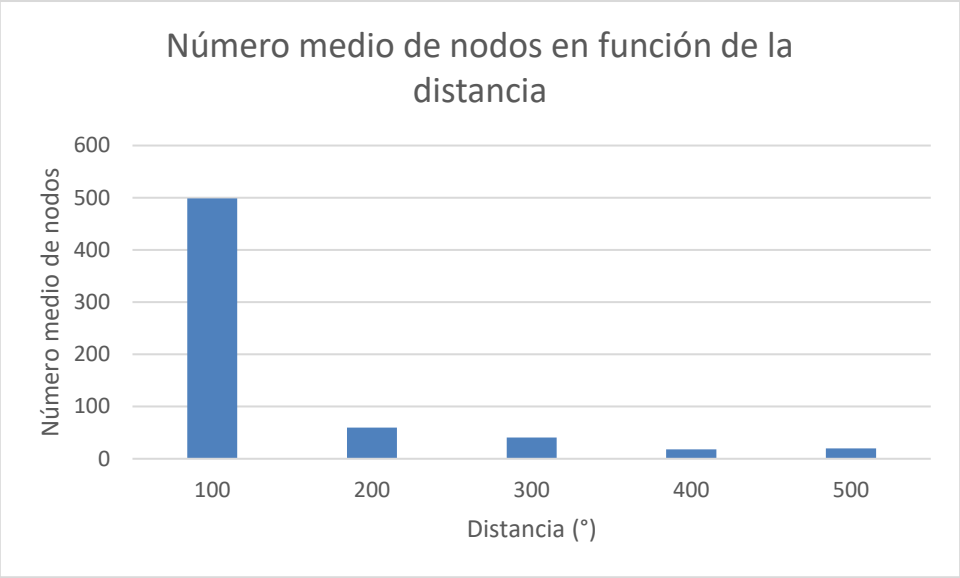


Gráfico 6. Número medio de nodos en función de la distancia mínima con el algoritmo RRT básico

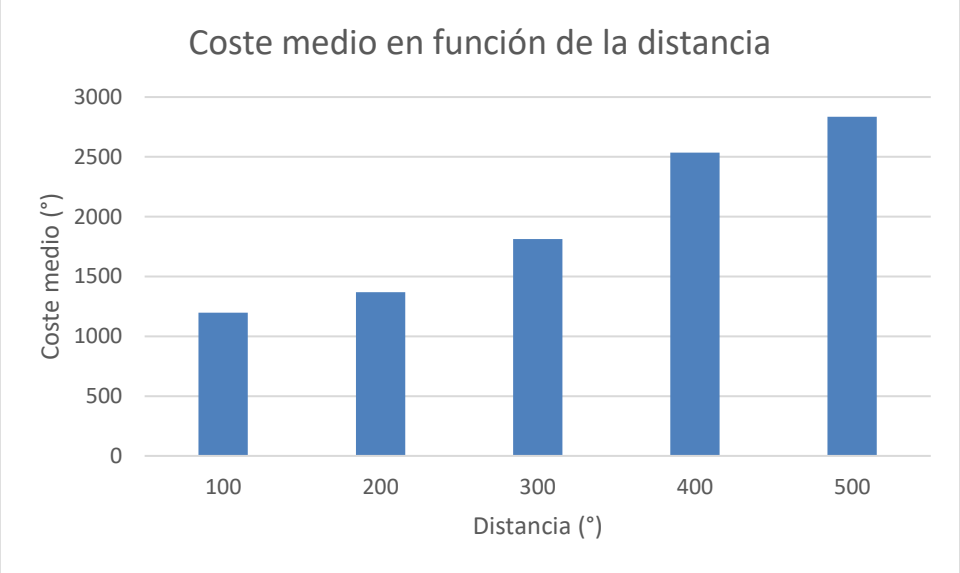


Gráfico 7. Coste medio de la trayectoria en función de la distancia mínima con el algoritmo RRT básico

En el Gráfico 6 se puede apreciar cómo el aumento de la distancia conlleva un descenso en el número de nodos necesarios para llegar al destino. Esto es especialmente notable entre una distancia de 100° y otra de 200°. El aumento de la distancia incrementa el avance de cada nodo y conlleva mayores posibilidades de que el nodo final quede lo bastante próximo como para ser unido. Para distancias mayores de 200° la disminución es menos severa y, para distancias muy grandes, como es el caso de 500°, el número de nodos se estanca o incluso aumenta. Esto es debido a que, al aumentar la distancia, llega un punto en que la posibilidad de que el camino entre dos nodos esté totalmente libre es lo bastante pequeña como para sobreponerse a la mayor probabilidad existente de encontrar antes el nodo si la distancia es mayor. Esto puede

retrasar la generación de nodos que puedan unirse o dar lugar a nodos que pasen de largo del objetivo o que se alejen de él por la gran longitud de las ramas, dificultando su alcance.

En cuanto al coste, tal y como se ve en el Gráfico 7, aumenta notablemente al incrementar la distancia, ya que mayores distancias entre los nodos implican que el camino a recorrer entre dos nodos sea mayor. Si la trayectoria entre el nodo inicial y el final fuese una recta en el espacio de configuraciones, la distancia no afectaría al coste. Sin embargo, esto no es así, sino que las ramificaciones del árbol suelen dar lugar a una ruta que zigzaguea en el espacio de configuraciones, lo que provoca que las grandes distancias entre nodos tiendan a aumentar el coste.

Por último, con el fin de estudiar el efecto del parámetro T sobre los tiempos de ejecución se calcula el tiempo de ejecución medio por nodo para cada caso, tal y como se ve en la Tabla 13 y en el Gráfico 8. Se observa claramente el aumento de tiempo de ejecución, fruto de un mayor muestreo de la trayectoria entre dos puntos y las mayores posibilidades de hallar uniones no posibles que obliguen a repetir la generación de algunos nodos.

T (s)	Tiempo de ejecución medio por nodo (ms/nodo)
0,5	21,3
1	35,9
1,5	64,5
2	112,4
2,5	221,4

Tabla 13. Tiempo medio de ejecución por nodo variando T con el algoritmo RRT básico

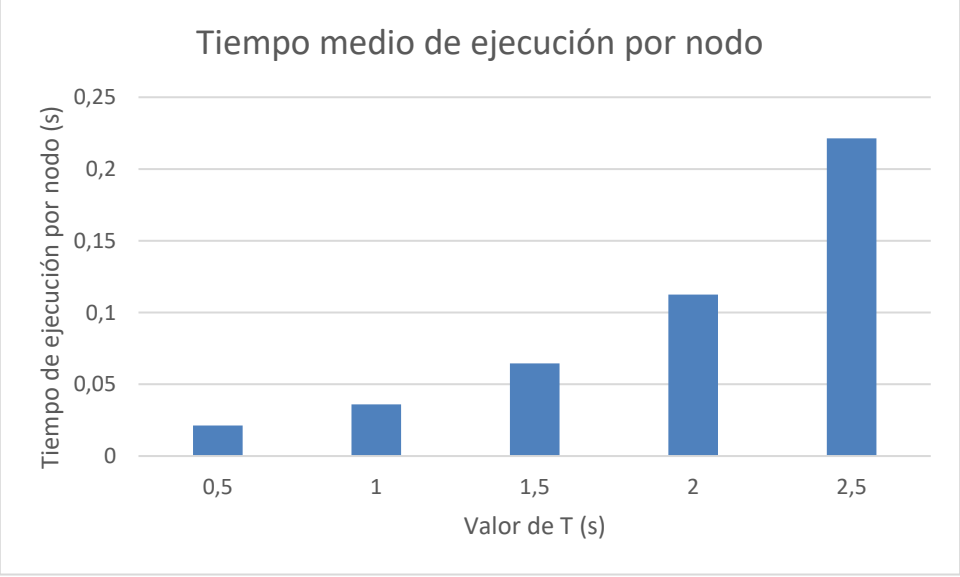


Gráfico 8. Tiempo medio de ejecución por nodo variando el parámetro T con RRT básico

5.3. Resultados obtenidos mediante RRT*

A continuación, se realiza un estudio de los resultados obtenidos con el algoritmo RRT*. Dado que la influencia del sesgo y de la distancia mínima ya se ha analizado en el apartado 5.2, en este caso, se observará el efecto de la variación del radio en el que se buscan nodos cercanos para conectar el nuevo nodo con aquel de los vecinos más próximos que tenga un menor coste desde el inicio. Las Tablas 14 y 15 muestran los nodos necesarios pare encontrar una ruta, el coste de dicha trayectoria y el tiempo computacional para veinte ejecuciones con distintos valores del radio. En todos los casos, la distancia es de 100°, con un valor de 0,5 segundos para el parámetro T y un sesgo del 5 %.

Radio (°)					
150			200		
Nodos	Coste (°)	Tiempo (s)	Nodos	Coste (°)	Tiempo (s)
362	867,09	30,53	899	839,84	99,23
109	1082,20	6,47	816	774,78	79,47
631	787,39	47,30	764	774,27	71,81
197	1018,90	8,50	354	1046,70	25,78
176	850,91	7,62	1481	593,86	222,95
1015	825,19	76,87	430	674,77	29,47
682	960,16	40,62	307	831,34	20,33
999	885,51	68,59	616	545,15	55,57
191	772,80	8,23	459	657,62	31,51
852	976,08	60,06	315	680,34	22,17
480	820,05	23,53	334	645,55	22,55
272	982,35	11,98	532	811,60	43,42
658	754,13	36,72	534	774,81	46,25
163	875,82	6,17	208	875,38	11,67
180	919,35	7,49	507	942,36	39,47
622	722,15	34,27	466	747,08	38,14
130	958,85	5,03	586	720,17	49,89
256	852,17	12,26	78	881,37	3,45
1684	739,35	171,58	333	1139,00	23,71
506	930,31	27,00	118	708,22	6,08

Tabla 14. Resultados obtenidos mediante RRT con radios de 150° y 200°*

Radio (°)					
250			300		
Nodos	Coste (°)	Tiempo (s)	Nodos	Coste (°)	Tiempo (s)
965	766,67	184,48	392	731,52	66,77
444	640,43	45,09	506	748,53	88,95
605	696,68	89,07	246	980,20	28,55
266	972,29	23,51	385	726,30	58,10
1009	821,71	174,37	197	711,95	19,31
356	687,35	33,39	176	739,41	14,17
139	589,88	9,57	1015	651,61	274,78
1375	714,72	282,40	682	825,50	125,68
175	1044,50	11,26	999	748,06	283,13
295	745,72	26,32	191	614,00	18,30
1013	653,83	171,38	852	772,65	202,39
424	921,33	42,87	480	780,56	76,44
331	766,73	30,86	885	787,94	213,22
90	679,29	5,96	64	759,51	3,97
342	773,92	32,42	142	628,26	13,10
972	692,79	167,44	180	839,68	16,07
192	731,44	17,24	622	631,11	97,22
395	774,30	40,62	130	664,59	10,69
198	805,51	16,25	256	659,03	25,75
421	671,36	42,83	1684	610,61	770,13

Tabla 15. Resultados obtenidos mediante RRT con radios de 250° y 300°*

Los valores medios del número de nodos, coste y tiempo de ejecución para cada valor del radio se muestran en la Tabla 16 y en los Gráficos 9, 10 y 11.

Radio (°)	Número medio de nodos	Coste medio (°)	Tiempo de ejecución medio (s)
150	508,25	879,04	34,54
200	506,85	783,21	47,15
250	500,35	757,52	72,37
300	504,20	730,55	110,74

Tabla 16. Valores medios obtenidos mediante RRT variando el radio*

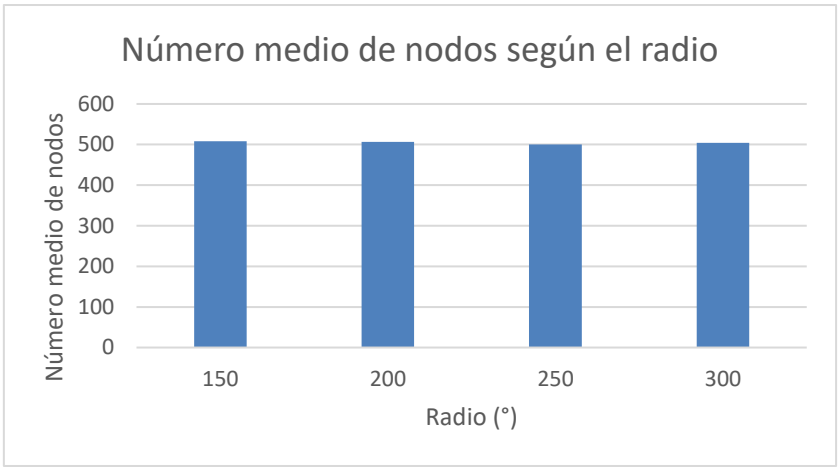


*Gráfico 9. Número medio de nodos según el radio con el algoritmo RRT**

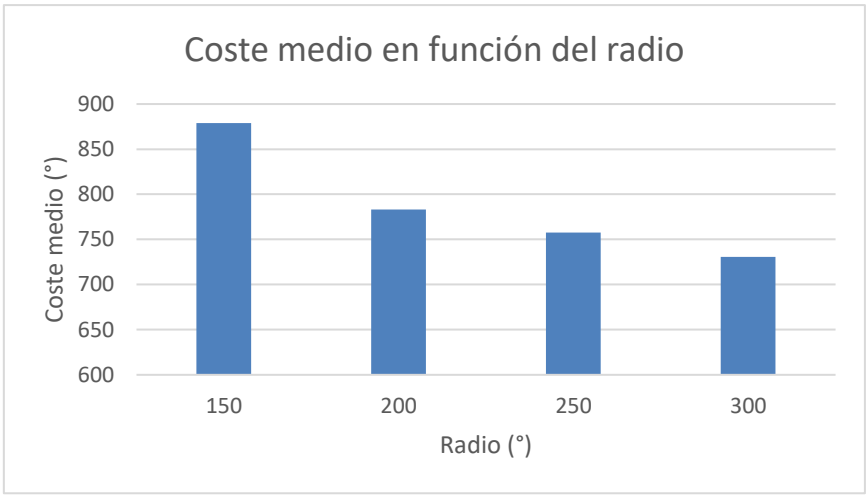


*Gráfico 10. Coste medio de la trayectoria según el radio con el algoritmo RRT**

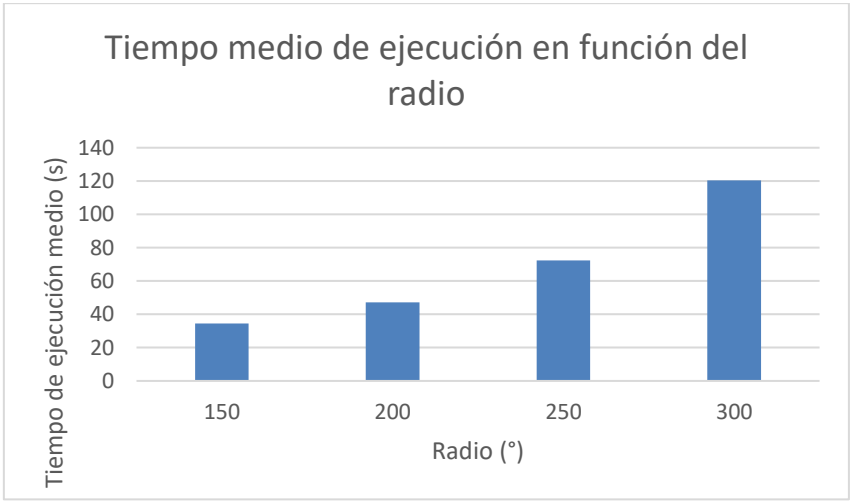


*Gráfico 11. Tiempo medio de ejecución según el radio con el algoritmo RRT**

Se observa en el Gráfico 9 que el radio no afecta en absoluto al número medio de nodos necesarios para hallar el camino. Esto es lógico, pues el radio no afecta a la generación de los nodos ni a la distancia a la que deben estar del nodo final para alcanzarlo; sino que solo determina con qué nodo del árbol se conecta el nuevo punto.

Un mayor radio, sí que influye en el coste, como muestra el Gráfico 10, ya que, de este modo, se revisa un mayor número de vecinos al incrementarse el radio, lo que aumenta las posibilidades de encontrar nodos con menor coste desde el inicio. Sin embargo, el mayor número de iteraciones del bucle que lee los costes de los nodos cercanos provoca un notable incremento en el tiempo de ejecución, lo cual se aprecia en el Gráfico 11. Además, el aumento del tiempo con el número de nodos no es lineal, ya que conforme aumenta el número de nodos, las iteraciones de dicho bucle también se incrementan. Los Gráficos 12, 13, 14 y 15, permiten ver el aumento del tiempo con el número de nodos para cada radio, que puede aproximarse a una curva polinómica de segundo grado en todos los casos.

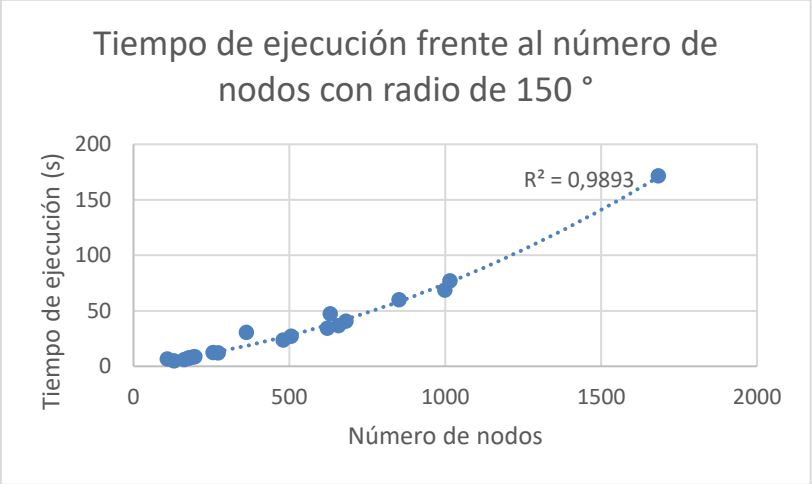


*Gráfico 12. Tiempo de ejecución frente al número de nodos con radio de 150° con el algoritmo RRT**

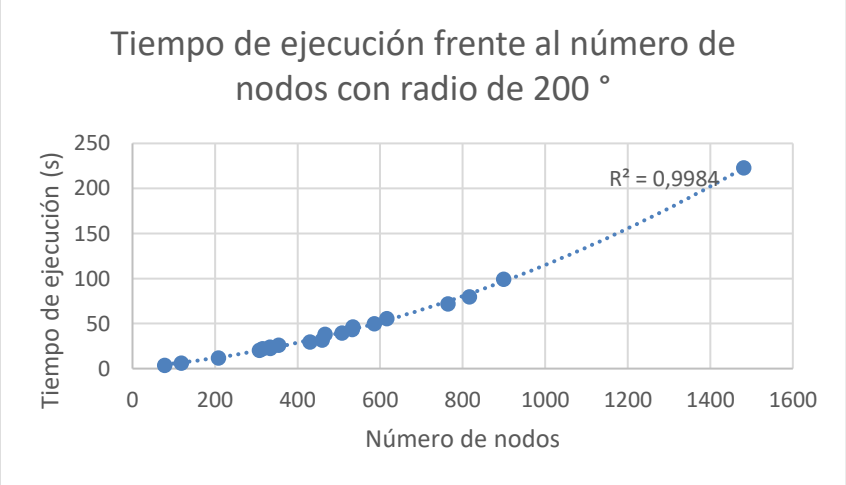


*Gráfico 13. Tiempo de ejecución frente al número de nodos con radio de 200° con el algoritmo RRT**

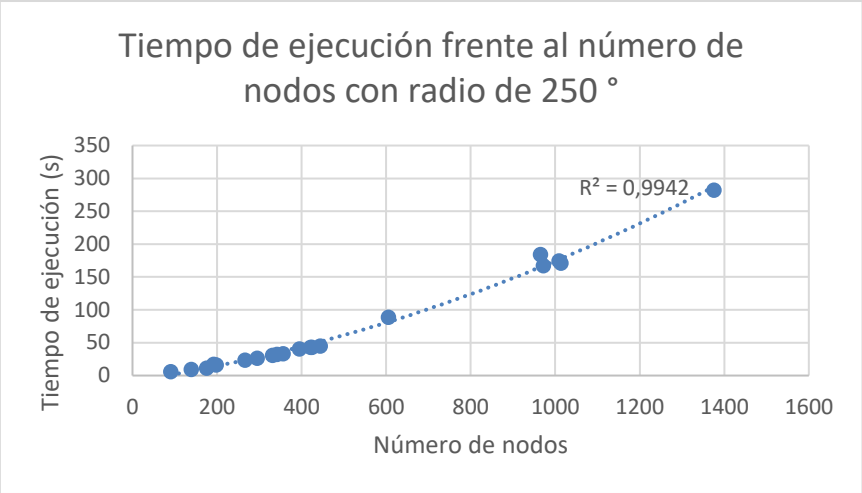


*Gráfico 14. Tiempo de ejecución frente al número de nodos con radio de 250° con el algoritmo RRT**

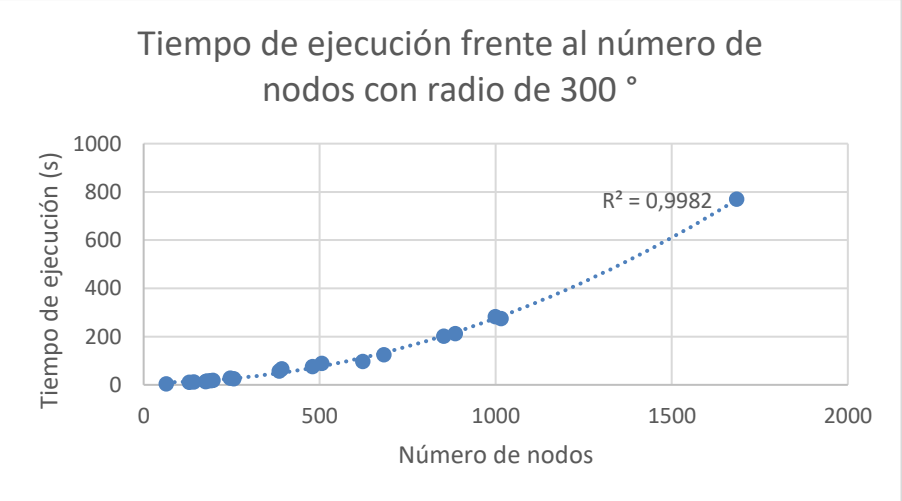


*Gráfico 15. Tiempo de ejecución frente al número de nodos con radio de 300° con el algoritmo RRT**

5.4. Análisis comparativo

Se procede ahora a comparar los resultados obtenidos con los algoritmos empleados para la generación de trayectorias, es decir, PRM, RRT básico y RRT*.

En primer lugar, se realiza una comparación entre el método RRT básico y PRM. Los resultados se muestran en la Tabla 17 y en los Gráficos 16, 17, 18 y 19.

Algoritmo	Sesgo (%)	Distancia mínima (°)	Número medio de nodos	Coste medio (°)	Tiempo de ejecución medio (s)	Probabilidad de éxito (%)
PRM	-----	-----	25	1131,68	1,41	50,00
	-----	-----	50	1036,88	4,87	90,00
	-----	-----	100	991,68	19,43	100,00
	-----	-----	150	909,76	42,75	100,00
	-----	-----	200	852,09	81,23	100,00
RRT	5	100	498,5	1198,7	10,62	100,00
	5	200	59,85	1368,01	2,15	100,00
	5	300	40,70	1812,34	2,63	100,00
	5	400	18,00	2535,25	2,02	100,00
	5	500,00	19,60	2836,41	4,34	100,00
	0	100,00	580,80	1244,70	12,00	100,00
	10	100,00	614,40	1246,61	14,61	100,00
	15	100,00	702,95	1238,09	14,85	100,00

Tabla 17. Comparación entre los resultados obtenidos con PRM y RRT básico

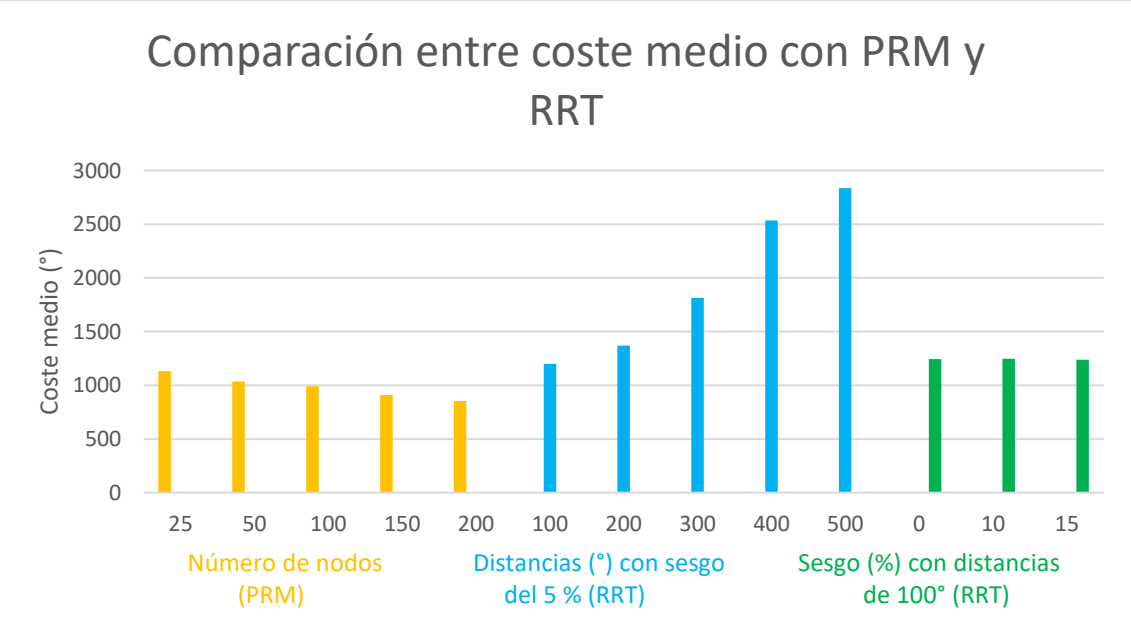


Gráfico 16. Comparación entre coste medio con PRM y RRT

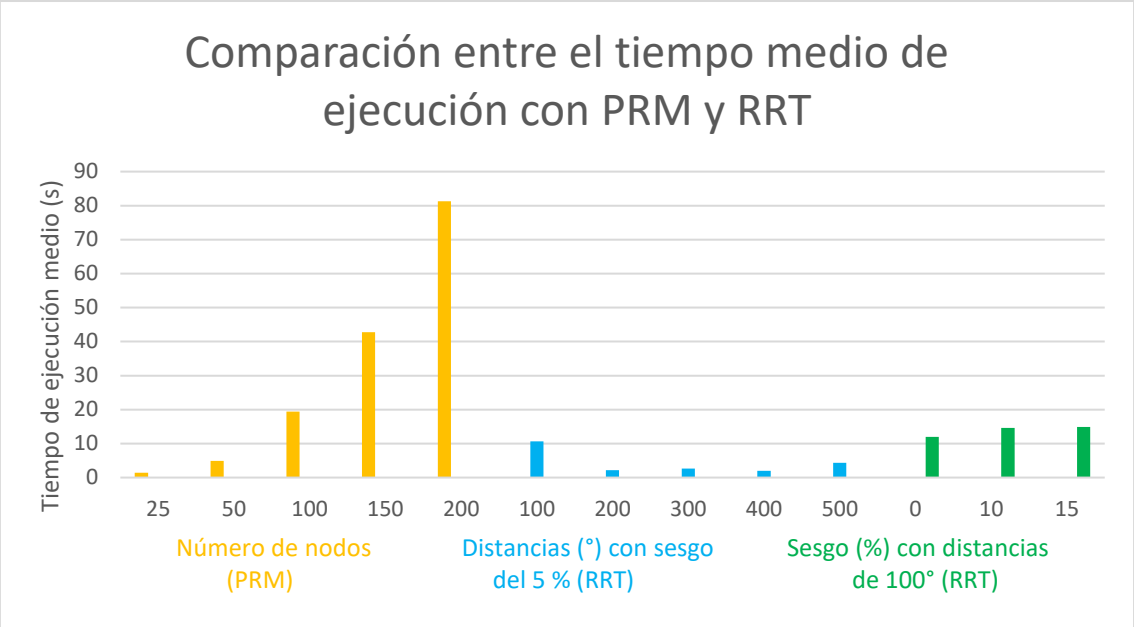


Gráfico 17. Comparación entre tiempo medio de ejecución con PRM y RRT

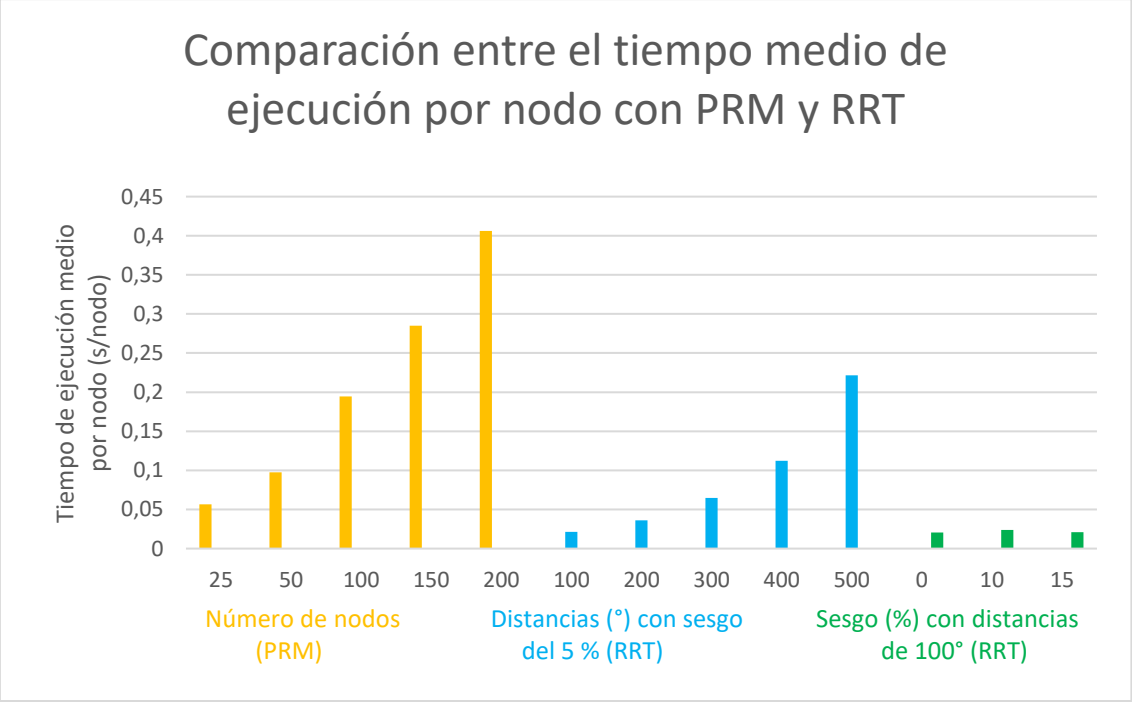


Gráfico 18. Comparación entre tiempo medio de ejecución por nodo con PRM y RRT

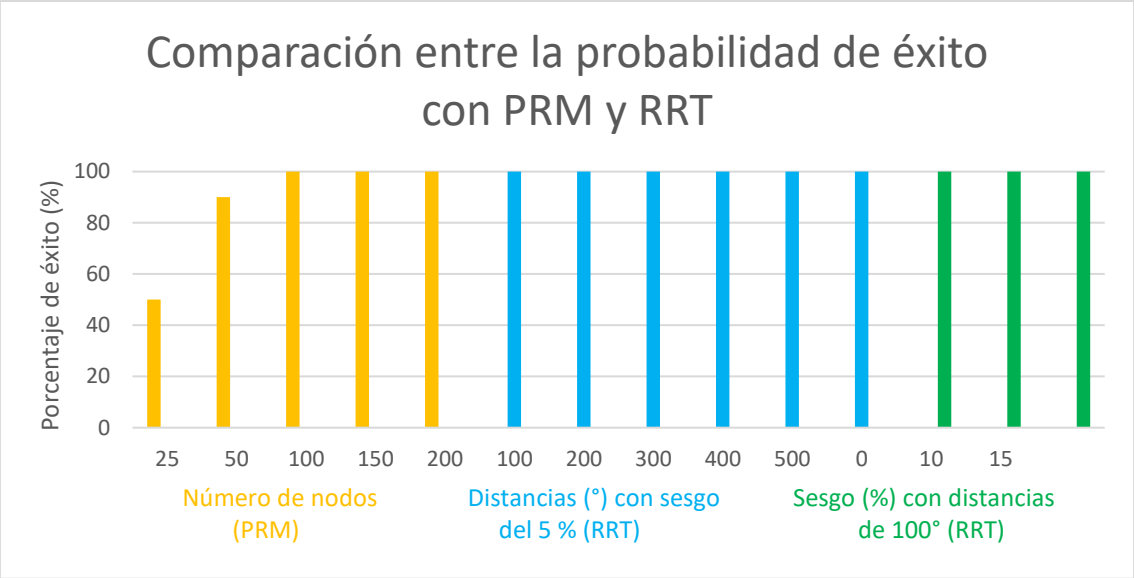


Gráfico 19. Comparación entre la probabilidad de éxito con PRM y RRT

Se aprecia claramente en la Tabla 17 y en el Gráfico 16 que los costes obtenidos mediante PRM, incluso con tan solo 25 nodos, son menores que los obtenidos mediante RRT. Las ramificaciones de RRT en el espacio de configuraciones dan lugar a trayectorias con un coste mucho mayor que PRM, que busca un camino óptimo mediante el algoritmo Dijkstra entre los nodos del mapa. Por otro lado, tal y como se ve en el Gráfico 19, la probabilidad de éxito de RRT es, en todos los casos, del 100 %, mientras que en PRM para 25 y 50 nodos existe un porcentaje de pruebas fallidas. El algoritmo PRM genera un cierto número de puntos y trata de unirlos sin entrar en colisión, mientras que RRT genera un nuevo nodo en cada iteración hasta hallar el camino o alcanzar un límite de iteraciones, por lo que RRT siempre hallará un camino, si este existe, cuando las iteraciones tiendan a infinito. En cuanto al tiempo de ejecución, se observa en el Gráfico 17 cómo es mucho mayor en PRM que en RRT, pues PRM comprueba la posibilidad de unir un nodo con todos los demás y RRT únicamente con el más cercano, lo que supone una carga computacional mucho menor. Así pues, si se utilizan en PRM 100 nodos o más, para garantizar con una cierta seguridad que se encuentre un camino, el tiempo requerido por RRT es menor en todos los casos estudiados. El Gráfico 18, que muestra el tiempo de ejecución por nodo permite ver cómo para un número de nodos en PRM que asegure encontrar un camino, el tiempo por nodo es superior que en RRT, salvo para un número de puntos de muestreo en la trayectoria de 500°, que supera a PRM con 100 nodos.

Por otro lado, se compara el algoritmo RRT básico con su variación RRT*. Los resultados de los que se parte se muestran en la Tabla 18 y en los Gráficos 20, 21 y 22. Para esta comparación únicamente se ha utilizado el valor obtenido mediante RRT con una distancia de 100° y un sesgo del 5 %, ya que son los parámetros utilizados en las pruebas del algoritmo RRT*.

Algoritmo	Radio (°)	Número medio de nodos	Coste medio (°)	Tiempo de ejecución medio (s)	Porcentaje de éxito (%)
RRT	-----	498,50	1198,70	10,62	100,00
RRT*	150	508,25	879,04	34,54	100,00
	200	506,85	783,21	47,15	100,00
	250	500,35	757,52	72,37	100,00
	300	504,2	730,55	120,34	100,00

*Tabla 18. Comparación entre los resultados obtenidos con RRT básico y RRT**

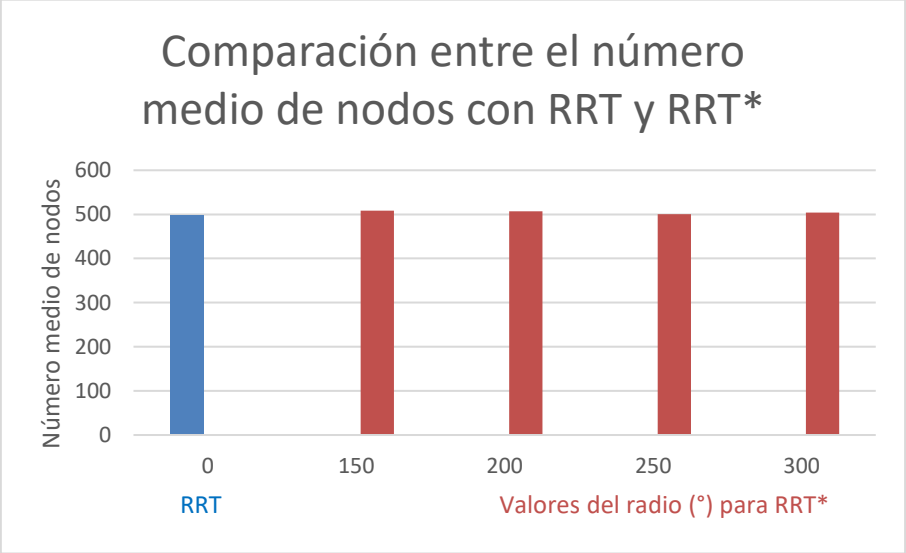


*Gráfico 20. Comparación entre el número medio de nodos con RRT y RRT**

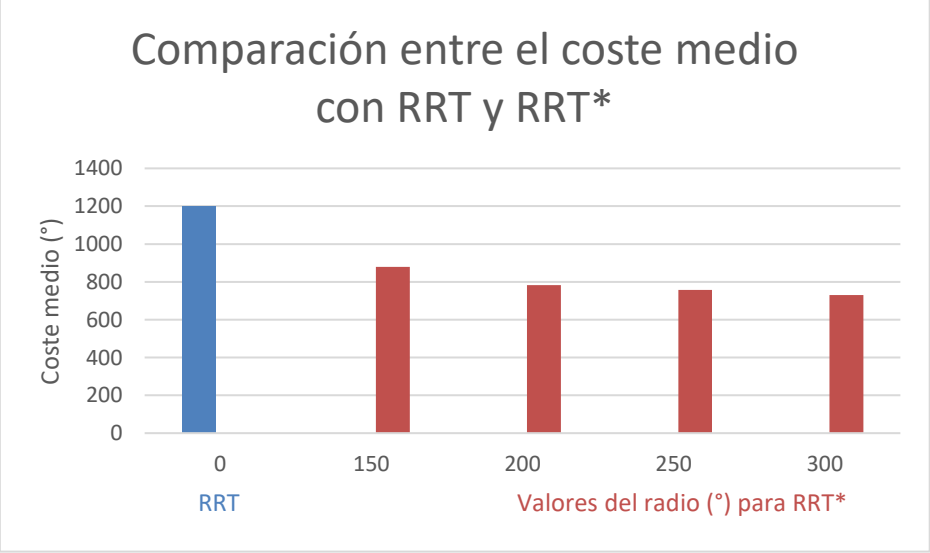


*Gráfico 21. Comparación entre el coste medio con RRT y RRT**

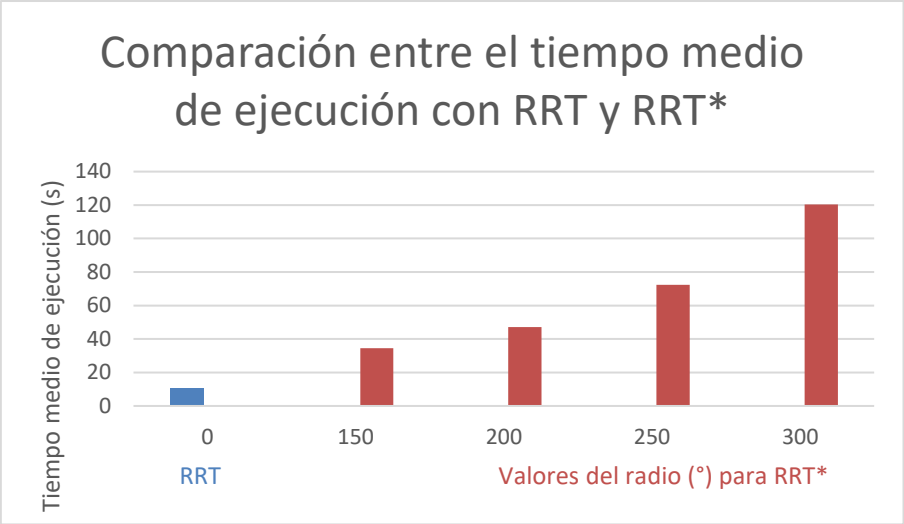


*Gráfico 22. Comparación entre el tiempo medio de ejecución con RRT y RRT**

Se observa en la Tabla 18 y en el Gráfico 21 un claro descenso en el valor del coste al emplear la técnica RRT* frente al coste obtenido con RRT clásico. La búsqueda del nodo con menor coste desde el inicio para unirlo al nuevo nodo generado en lugar de unirlo directamente al más cercano logra reducir de forma significativa el coste de la ruta. Además, conforme se aumenta el radio disminuye aún más el coste. Sin embargo, ello es a costa de incrementar el tiempo de ejecución del programa como muestra el Gráfico 22, ya que conlleva recorrer los nodos dentro de radio establecido en busca de aquel con un menor coste respecto del nodo de partida. El porcentaje de éxito no varía y es del 100 %, pues al igual que el método RRT básico, RRT* continúa generando nodos hasta llegar al destino sin colisionar con los obstáculos. Por otro lado, la generación de los nodos es igual que con RRT, solo cambia el nodo del árbol al que se une, por lo que, tal y como muestran los resultados en el Gráfico 20, el número medio de nodos necesarios para alcanzar la meta no presenta variaciones, y se mantiene en una cifra próxima a los 500 nodos con ambos algoritmos para los valores de sesgo y distancia elegidos.

Finalmente, se pueden comparar también los resultados de PRM y RRT*, que figuran en la Tabla 19 y en los Gráficos 23, 24, 25 y 26.

Algoritmo	Sesgo (%)	Distancia mínima (°)	Radio (°)	Número medio de nodos	Coste medio (°)	Tiempo de ejecución medio (s)	Probabilidad de éxito (%)
PRM	-----	-----	-----	25	1131,68	1,41	50
	-----	-----	-----	50	1036,88	4,87	90
	-----	-----	-----	100	991,68	19,43	100
	-----	-----	-----	150	909,76	42,75	100
	-----	-----	-----	200	852,09	81,23	100
RRT *	5	100	150	508,25	879,04	34,54	100,00
	5	100	200	506,85	783,21	47,15	100,00
	5	100	250	500,35	757,52	72,37	100,00
	5	100	300	504,2	730,55	120,34	100,00

*Tabla 19. Comparación entre los resultados obtenidos con PRM y RRT**

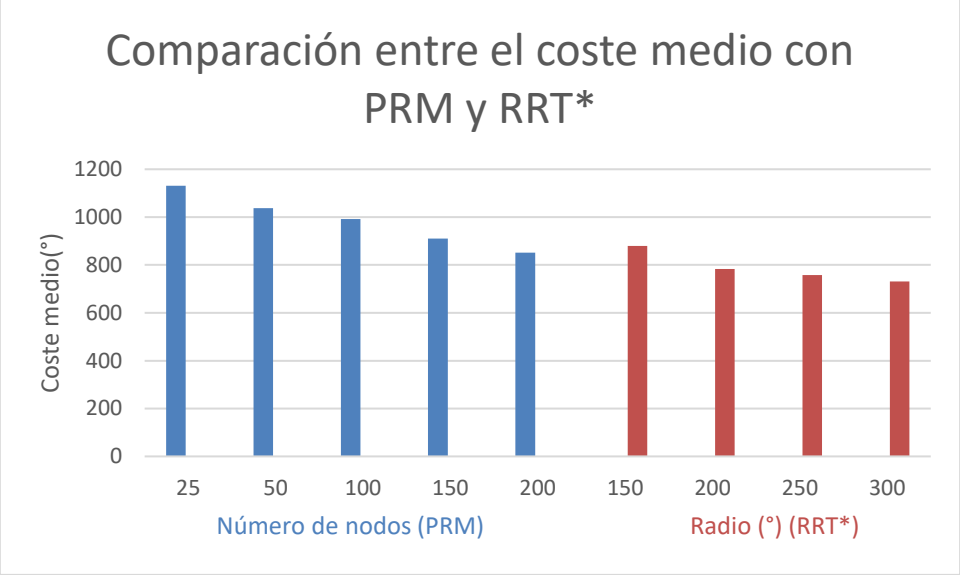


*Gráfico 23. Comparación entre el coste medio con PRM y RRT**

En este caso, como señalan los datos de la Tabla 19 y el Gráfico 23, los costes obtenidos difieren menos que con RRT. Se puede apreciar que, para un número de nodos de hasta 150° con PRM el coste es ahora mayor que con RRT*, para radios superiores a 150°. El tiempo de ejecución por nodo, visible en el Gráfico 25, sigue siendo mayor con el algoritmo PRM, pues la media de nodos con RRT* se sitúa en torno a un valor de 500, aunque RRT* requiere de un tiempo de ejecución superior a RRT básico, y dicho tiempo se incrementa al aumentar el radio de búsqueda. El tiempo total necesario en este caso es mayor en uno u otro método según los nodos establecidos en PRM y el radio de RRT*, como se observa en el Gráfico 24. En cuanto a la probabilidad de éxito, con RRT* se alcanza un camino en todas las pruebas, lo que con PRM solo ocurre con 100 nodos o más, como se ve en el Gráfico 26.

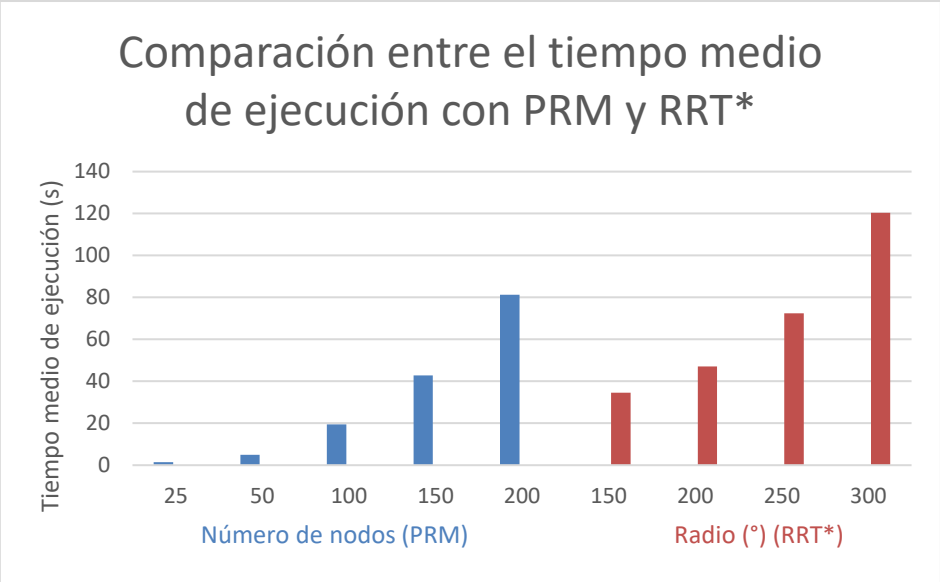


*Gráfico 24. Comparación entre el tiempo de ejecución con PRM y RRT**

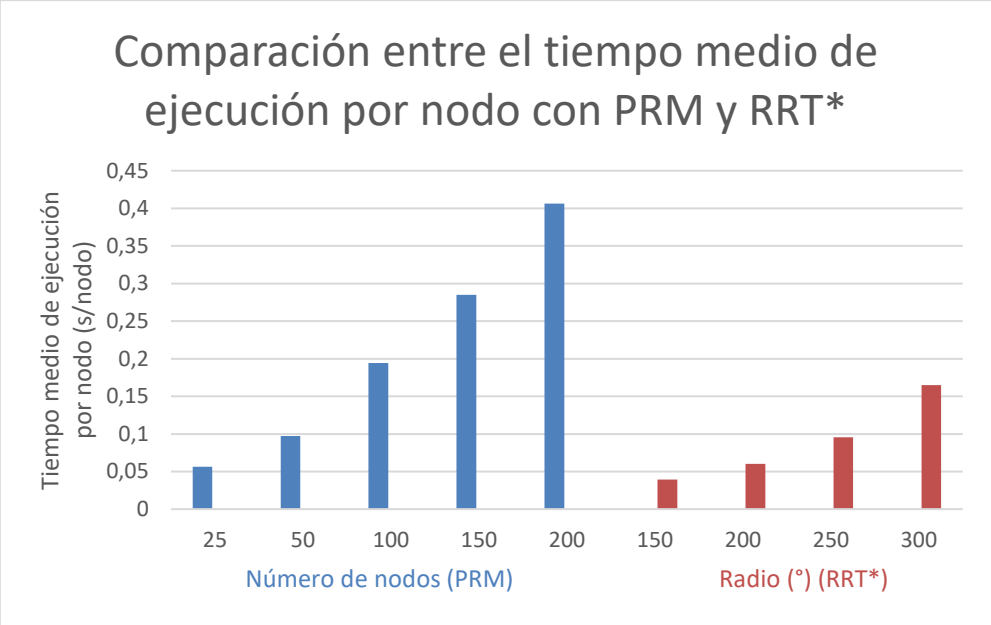


*Gráfico 25. Comparación entre el tiempo medio de ejecución por nodo con PRM y RRT**

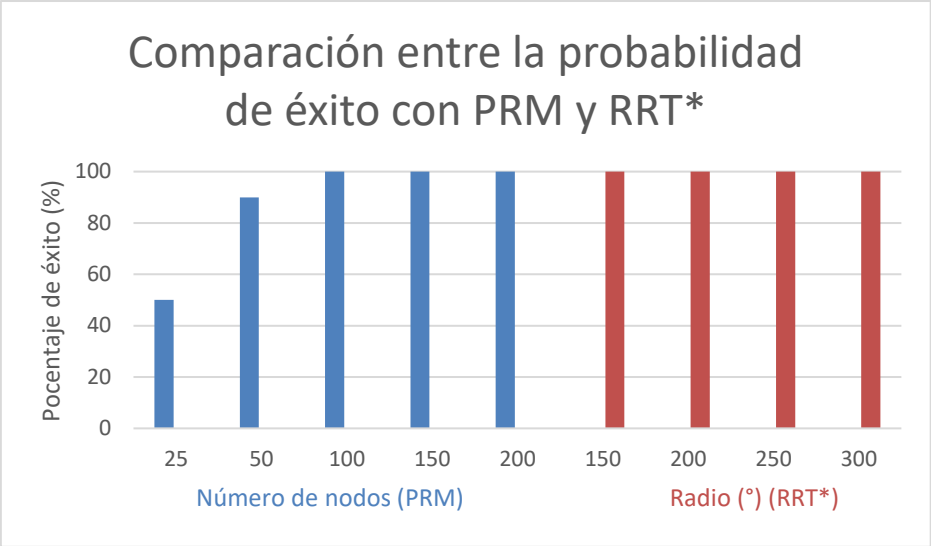


*Gráfico 26. Comparación entre la probabilidad de éxito con PRM y RRT**

En definitiva, los resultados recogidos muestran que el algoritmo PRM, con un número de nodos lo bastante alto como para garantizar una alta probabilidad de éxito en la búsqueda de una trayectoria, da lugar a rutas con costes menores que RRT en su versión básica. Por otro lado, RRT, encuentra el camino en todos los casos, si este existe, cuando sus nodos tienden a infinito y con un menor tiempo de ejecución por nodo. Si bien es cierto, que este algoritmo da lugar a rutas con un coste muy elevado, una de sus variaciones, RRT*, permite reducir su coste logrando, para radios lo bastante grandes, costes similares e incluso menores que obtenidos con PRM, aunque ello supone aumentar considerablemente su carga computacional con respecto a RRT básico, hasta tal punto que, dependiendo del caso, puede llegar a ser más lento que el método PRM.

6. Conclusiones

A continuación, se exponen las conclusiones a las que se ha llegado tras la realización del presente proyecto.

En primer lugar, se ha desarrollado un software en Matlab que permite generar trayectorias a partir de los algoritmos PRM (*Probablilistic Roadmap*) y RRT (*Rapidly-exploring Random Tree*) que crean nodos en el espacio de configuraciones para alcanzar un punto final partiendo de una posición inicial. Además, con el fin de corregir uno de los defectos del método RRT, concretamente el alto coste de sus trayectorias, se ha implementado también la técnica RRT*. Las pruebas realizadas demuestran que se consigue alcanzar el punto final evitando todos los obstáculos con los tres métodos utilizados, excepto con PRM y cantidades reducidas de nodos, caso en el que no siempre se encuentra una ruta.

En segundo lugar, se ha logrado aplicar el software diseñado, así como los algoritmos de planificación de trayectorias elegidos al modelo IRB 140 de ABB. Las funciones implementadas logran descartar aquellas configuraciones que provocan el choque de los eslabones del robot con otros eslabones y con los elementos del entorno. De ese modo, los nodos generados que se hallan en C_{obs} son descartados y no se tienen en cuenta en la búsqueda de rutas entre configuraciones.

También se han implementado con éxito funciones que permiten al robot sortear los obstáculos situados en su área de trabajo descartando aquellas conexiones entre nodos cuya trayectoria implica una colisión, ya sea del robot consigo mismo o con alguno de los obstáculos presentes en su área de trabajo. El método elegido se basa en pasar del espacio de configuraciones al cartesiano mediante la cinemática directa y comprobar si las trayectorias correspondientes al paso entre una configuración y otra chocan con algún elemento del entorno. De este modo, no es necesario representar los obstáculos en el espacio articular de 6 dimensiones, lo que simplifica enormemente los cálculos y rebaja la carga computacional. Las herramientas proporcionadas por Multi-Parametric Toolbox han sido de gran ayuda en esta tarea.

Además, se ha desarrollado un entorno para simular y visualizar las trayectorias generadas y los movimientos del robot, lo que permite una mayor comprensión de los resultados obtenidos y la verificación del correcto funcionamiento del sistema. Para ello se ha utilizado Simscape Multibody, una extensión de Matlab que permite importar moldeos de CAD de elementos mecánicos, teniendo en cuenta, no solo sus características geométricas sino también su masa e inercia; y realizar simulaciones de su funcionamiento. Así se ha podido implementar una simulación del movimiento del robot en su entorno utilizando los modelos CAD 3D proporcionados por el fabricante del brazo.

Tras implementar el código necesario para la planificación de las trayectorias, se ha llevado a cabo un análisis comparativo de los algoritmos empleados. Se ha llegado a la conclusión de que PRM obtiene rutas con menores costes que RRT, cuyos costes, muchos más altos, se reducen considerablemente con RRT*, que llega incluso a obtener costes menores que PRM para el número de nodos establecido en las pruebas realizadas. Otro aspecto para destacar es la mayor rapidez de RRT frente a PRM, y también frente RRT*, pues este último mejora los costes de RRT,

pero requiere para ello más cálculos y por consiguiente más tiempo. La comparación también apunta a que PRM puede dar lugar, especialmente con pocos nodos y configuraciones de partida y de destino próximas a varios obstáculos a probabilidades relativamente elevadas de que no se encuentre un camino, aunque éste exista; mientras que RRT (y su variante RTT*) siempre hallan el camino, si este existe, cuando el número de sus nodos tiende a infinito. La preferencia de uno u otro algoritmo vendrá dada por un compromiso entre garantía de éxito, rapidez y coste.

Finalmente, cabe destacar que las funciones implementadas son extrapolables a otros tipos de brazos robóticos y entornos, realizando los ajustes pertinentes, por lo que pueden ser aplicadas a distintos manipuladores robóticos utilizados en la industria en diferentes áreas de trabajo.

Por otra parte, conviene resaltar que la mayor dificultad encontrada en el desarrollo del proyecto ha sido la verificación de la ausencia de colisiones en el trayecto que une dos nodos en el espacio de configuraciones. Ello supone trabajar en un espacio de seis dimensiones, lo cual conlleva cálculos complejos y de un alto coste computacional. Con el fin de simplificar este problema se ha optado por utilizar la cinemática directa para pasar del espacio de configuraciones al espacio cartesiano, lo que reduce el número de dimensiones del espacio de trabajo a tres. Para comprobar las colisiones se muestrea la trayectoria en el espacio cartesiano y se compruebe que ninguno de los puntos que la definen se halla dentro de un obstáculo.

Es cierto que el desarrollo de un método en el espacio de configuraciones de seis dimensiones que determine si, para cierta combinación de valores articulares, se produce un impacto sin necesidad de pasar al espacio cartesiano ni de realizar un muestreo a lo largo de la trayectoria sería más preciso y, además, no requeriría de la toma de ciertos puntos a lo largo de los eslabones del robot ni de la ampliación de los obstáculos; sino simplemente de una serie de cálculos trigonométricos y cinemáticos. No obstante, y como ya se ha mencionado, ello conllevaría un incremento significativo de la complejidad de los cálculos y del tiempo de ejecución por suponer trabajar en un espacio con un alto número de dimensiones. La alternativa elegida, mucho más sencilla en términos de computación, permite igualmente la obtención de trayectorias que alcanzan el objetivo sin colisionar, por lo que es también válida para la resolución del problema. Sin embargo, la implementación de funciones que verifiquen las colisiones en el espacio de configuraciones N-dimensional de un robot queda como trabajo futuro para otros proyectos.

Otro aspecto para tener en cuenta en el ámbito de posibles trabajos en el futuro es el desarrollo de sistemas de control para las articulaciones del robot, que permitan el seguimiento de la trayectoria con unas ciertas especificaciones y corrigiendo las perturbaciones causadas por la acción de la gravedad o las fuerzas de inercia de los eslabones. En este proyecto se ha considerado que la implementación de dichos sistemas queda fuera de su ámbito de estudio, centrándose únicamente en la obtención de trayectorias libres de obstáculos entre dos configuraciones.

Con todo, y pese a las dificultades encontradas a lo largo del desarrollo del trabajo, se han logrado todos los objetivos perseguidos al obtener una serie de funciones que, mediante los métodos PRM y RRT, generan y simulan una trayectoria que lleva al robot de un punto de partida a otro de destino evitando cualquier colisión en su movimiento.

7. Bibliografía

[1] A. Barrientos, *Fundamentos de robótica*, 2ª ed. Madrid: McGraw-Hill, Interamericana de España, 2007.

[2] S. B. Niku, *Introduction to robotics: analysis, systems, applications*. Upper Saddle River: Prentice Hall, 2001.

[3] T. Yoshikawa, *Foundations of robotics: Analysis and control*. Cambridge, Massachusetts: MIT Press, 1990.

[4] M. Kim, D.-K. Han, J.-H. Park, y J.-S. Kim, «Motion Planning of Robot Manipulators for a Smoother Path Using a Twin Delayed Deep Deterministic Policy Gradient with Hindsight Experience Replay», *Appl. Sci.*, vol. 10, n.º 2, pp. 575-, 2020, doi: 10.3390/app10020575.

[5] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, 1st ed. 2011. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-20144-8.

[6] L. Sciavicco, *Modelling and control of robot manipulators*, 2nd ed. London: Springer, 2002.

[7] G. Carbone y F. Gomez-Bravo, *Motion and Operation Planning of Robotic Systems: Background and Practical Approaches*, 1st ed. 2015. Cham: Springer International Publishing, 2015. doi: 10.1007/978-3-319-14705-5.

[8] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, y Y. Xia, «Survey of Robot 3D Path Planning Algorithms», *J. Control Sci. Eng.*, vol. 2016, pp. 1-22, 2016, doi: 10.1155/2016/7426913.

[9] M. Šeda, «Roadmap methods vs. cell decomposition in robot motion planning», en *Proceedings of the 6th WSEAS international conference on signal processing, robotics and automation*, 2007, pp. 127-132.

[10] H. Choset *et al.*, *Principles of robot motion: theory, algorithms, and implementation*. Cambridge, Mass: A Bradford Book, 2005.

[11] N. A. Melchior y R. G. Simmons, «Particle RRT for Path Planning with Uncertainty.», en *ICRA*, 2007, pp. 1617-1624.

[12] A. T. Khan, S. Li, S. Kadry, y Y. Nam, «Control Framework for Trajectory Planning of Soft Manipulator Using Optimized RRT Algorithm», *IEEE Access*, vol. 8, pp. 171730-171743, 2020, doi: 10.1109/ACCESS.2020.3024630.

[13] H. Zhang, Y. Wang, J. Zheng, y J. Yu, «Path Planning of Industrial Robot Based on Improved RRT Algorithm in Complex Environments», *IEEE Access*, vol. 6, pp. 53296-53306, 2018, doi: 10.1109/ACCESS.2018.2871222.

[14] S. Kaden y U. Thomas, «Optimizing Mobility of Robotic Arms in Collision-free Motion Planning», *J. Intell. Robot. Syst.*, vol. 102, n.º 2, 2021, doi: 10.1007/s10846-021-01407-0.

[15] «Simscape Multibody». https://es.mathworks.com/products/simscape-multibody.html (accedido 6 de junio de 2022).

[16] R. S. Robeva y M. Macauley, *Algebraic and combinatorial computational biology*. London, United Kingdom: Academic Press is an imprint of Elsevier, 2019.

[17] C. D. Toth, J. O'Rourke, y J. E. Goodman, *Handbook of discrete and computational geometry*. CRC press, 2017.

[18] «MPT3 Wiki | Main / HomePage». https://www.mpt3.org/ (accedido 6 de junio de 2022).

[19] «Datos del IRB 140», *Robotics*. https://new.abb.com/products/robotics/es/robots-industriales/irb-140/datos (accedido 6 de junio de 2022).

[20] «IRB 140 CAD», *Robotics*. https://new.abb.com/products/robotics/es/robots-industriales/irb-140/cad (accedido 6 de junio de 2022).

[21] «Install the Simscape Multibody Link Plugin - MATLAB & Simulink - MathWorks España». https://es.mathworks.com/help/physmod/smlink/ug/installing-and-linking-simmechanics-link-software.html (accedido 6 de junio de 2022).

8. Pliego de Condiciones

8.1. Objeto

El presente documento se refiere a la implementación de un software desarrollado en Matlab para la simulación y generación de trayectorias mediante los algoritmos PRM y RRT, tanto en su forma básica como en su variante RRT*, para el brazo robótico IRB 140 de ABB. Los movimientos planificados deben permitir alcanzar una configuración final partiendo de otra inicial evitando los obstáculos presentes en el entorno. Quedan excluidos de este proyecto el diseño del robot, pues se emplea un modelo de la empresa ABB ya utilizado en la industria, y la implementación de los sistemas de control de las articulaciones del brazo.

8.2. Especificaciones de ejecución

El primer paso, antes de proceder a la generación de trayectorias, es la ejecución del *script* en el que se define el entorno de trabajo, es decir, los obstáculos y sus ampliaciones según el volumen de las distintas piezas del robot para la verificación de las colisiones.

Si se desea utilizar el algoritmo PRM, se debe utilizar el script creado para PRM, indicando el número de nodos que se desea crear en la variable *N_puntos*. Deben introducirse también los valores del nodo inicial y de destino. Se recomienda un valor mínimo de 100 nodos. Si el script devuelve error por no haber hallado un camino se ha de ejecutar de nuevo, preferiblemente con un número de nodos superior. Para visualizar la simulación debe emplearse el *script* creado para la simulación de trayectorias generadas mediante PRM.

Si, por el contrario, se desean aplicar los métodos RRT o RRT*, debe ejecutarse el *script* para inicializar RRT (o RRT*), donde se deben indicar las configuraciones inicial y final, la distancia, el sesgo y el número máximo de iteraciones. Se recomiendan valores de entre 100° y 300° para la distancia, un sesgo del 5 % y un máximo de 3000 iteraciones. Se aconseja muestrear las trayectorias utilizando un número de puntos igual a la distancia seleccionada. Posteriormente se debe ejecutar el *script* que genera el árbol mediante RRT o RRT* según el caso. En este último caso, se recomienda un valor del radio superior a la distancia para una reducción significativa del coste. Para visualizar la simulación se utiliza el *script* creado para la simulación de trayectorias generadas mediante RRT o RRT*.

Si se desea alterar el entorno, deben redefinirse los obstáculos como poliedros en el *script* donde se declaran, con sus respectivas ampliaciones y se han de añadir al modelo de Simulink para que aparezcan en la simulación. Las nuevas comprobaciones necesarias en la verificación de colisiones entre los nuevos elementos del entorno y el robot han de añadirse en las funciones *Colision_Configuracion_Obstaculos* y *Colision_Curva_MoveAbsJ_Obstaculos*. Del mismo modo, si se añade una herramienta al robot debe tomarse un nuevo punto en su extremo y añadirse las comprobaciones necesarias y las ampliaciones correspondientes para los obstáculos.

8.3. Prueba de servicio

Si se altera el entrono o se añade una herramienta, es preciso realizar una prueba de servicio para comprobar el correcto funcionamiento del *software* tras las modificaciones sufridas.

Para ello, se han de ejecutar al menos cinco veces cada uno de los tres métodos implementados con distintas configuraciones de inicio y de destino, todas ellas cercanas a los nuevos obstáculos añadidos. Tras cada ejecución se debe simular la trayectoria generada para validarla, comprobando de manera visual que no se produce ninguna colisión con ningún elemento del entorno.

9. Presupuesto

A continuación, se muestra el presupuesto del proyecto que se ha llevado a cabo, incluyendo los gastos de personal (Tabla 20), de equipos y de licencias de *software* (Tabla 21) y el coste total (Tabla 22).

Costes de personal			
Descripción	Coste unitario (€/h)	Horas (h)	Coste (€)
Documentación	10,00	30	300,00
Programación de los algoritmos	20,00	150	3000,00
Recogida y análisis de resultados	15,00	25	375,00
Redacción de la memoria	10,00	80	800,00
Reuniones con el tutor	20,00	15	300,00
TOTAL		300	**4775,00**

Tabla 20. Costes de personal

Costes de equipos y licencias			
Descripción	Coste unitario (€/ud)	Unidades (ud)	Coste (€)
Licencia de Matlab y Simulink para estudiantes	69,00	1	69,00
Simscape Multibody	7,00	1	7,00
Ordenador portátil Lenovo Intel Core i7 con 8 GB de RAM	749,99	1	749,99
Licencia de Microsoft Office 365	13,90	1	13,90
TOTAL			**839,89**

Tabla 21. Costes de equipos y licencias

Coste total	
Descripción	Coste (€)
Costes de personal	4.775,00 €
Costes de equipos y licencias	839,89 €
TOTAL SIN IVA	**5.614,89 €**
IVA (21 %)	1.179,13 €
TOTAL (IVA INCLUIDO)	**6.794,02 €**

Tabla 22. Coste total

El coste total del proyecto, con IVA incluido, es de seis mil setecientos noventa y cuatro euros y dos céntimos.

10. Anexos

Índice de anexos

10.1. Código de Matlab

10.1.1. Función Cinematica_Directa

```
function [T_06] = Cinematica_Directa(q1,q2,q3,q4,q5,q6)
% Devuelve la matriz de transformación entre el sistema fijo de la base y
% el del extremo a partir del valor de las articulaciones.
%% Parámetros Denavit-Hartenberg
theta_1=q1;
theta_2=q2-pi/2;
theta_3=q3;
theta_4=q4;
theta_5=q5;
theta_6=q6;

d_1=352;
d_2=0;
d_3=0;
d_4=380;
d_5=0;
d_6=65;

a_1=70;
a_2=360;
a_3=0;
a_4=0;
a_5=0;
a_6=0;

alpha_1=-pi/2;
alpha_2=0;
alpha_3=-pi/2;
alpha_4=pi/2;
alpha_5=-pi/2;
alpha_6=0;

%% Matrices de transformación
T_01=MatrizTransformacion(theta_1,d_1,a_1,alpha_1);
T_12=MatrizTransformacion(theta_2,d_2,a_2,alpha_2);
T_23=MatrizTransformacion(theta_3,d_3,a_3,alpha_3);
T_34=MatrizTransformacion(theta_4,d_4,a_4,alpha_4);
T_45=MatrizTransformacion(theta_5,d_5,a_5,alpha_5);
T_56=MatrizTransformacion(theta_6,d_6,a_6,alpha_6);

T_02=T_01*T_12;
T_03=T_02*T_23;
T_04=T_03*T_34;
T_05=T_04*T_45;
T_06=T_05*T_56;

end
```

10.1.2. Función Colision_Configuracion_Obstaculos

```
function [res] = Colision_Configuracion_Obstaculos(q,Suelo_a6,Env_1_Base_a6,
Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,
Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,
Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,
Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)
% Comprueba si existe colisión en  una configuración q
% Se deben pasar además los obstáculos ampliados para cada uno de los
% puntos del robot que se comprueban

res=0;

% Comprobar que los valores de las articulaciones están dentro de los
% límites

if    q(1)<-180||q(1)>180||q(2)<-90||q(2)>110||q(3)<-230||q(3)>50||q(4)<-180||q(4)
>180||q(5)<-115||q(5)>115||q(6)<-180||q(6)>180
    res=1;
end


% Paso de grados a radianes
q=q*pi/180;

% Vectores de puntos del robot en coordenadas cartesianas
[x6,y6,z6,x45,y45,z45,x3,y3,z3,x41,y41,z41,x42,y42,z42,x43,y43,z43,x21,y21,z21,x22,
y22,z22,x23,y23,z23,x24,y24,z24]   =   Trayectoria_xyz(q(1),q(2),q(3),q(4),q(5),q(6));


%% Colisiones con punto en el origen del sistema 6 ampliando obstáculos
%% Suelo
if res==0
res=Suelo_a6.contains([x6;y6;z6]);
%% Envolvente 1 de la Base
if res==0
res=Env_1_Base_a6.contains([x6;y6;z6]);
end
%% Envolvente 2 de la Base
if res==0
res=Env_2_Base_a6.contains([x6;y6;z6]);
end
%% Obstáculo 1
if res==0
res=Obstaculo1_a6.contains([x6;y6;z6]);
end
%% Obstáculo 2
if res==0
res=Obstaculo2_a6.contains([x6;y6;z6]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a6.contains([x6;y6;z6]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a6.contains([x6;y6;z6]);
end
%% Link 1
```

```
if res==0
res=Choque_link1_a6.contains([x6;y6;z6]);
end

%% Colisiones con punto en el origen de los sistemas 4 y 5 ampliando obstáculos
%% Suelo
if res==0
res=Suelo_a4.contains([x45;y45;z45]);
end
%% Envolvente 1 de la Base
if res==0
res=Env_1_Base_a4.contains([x45;y45;z45]);
end
%% Envolvente 2 de la Base
if res==0
res=Env_2_Base_a4.contains([x45;y45;z45]);
end
%% Obstáculo 1
if res==0
res=Obstaculo1_a4.contains([x45;y45;z45]);
end
%% Obstáculo 2
if res==0
res=Obstaculo2_a4.contains([x45;y45;z45]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a4.contains([x45;y45;z45]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a4.contains([x45;y45;z45]);
end
%% Link 1
if res==0
res=Choque_link1_a4.contains([x45;y45;z45]);
end

%% Colisiones con punto en el origen de los sistemas 2 y 3 ampliando obstáculos
%% Obstáculo 1
if res==0
res=Obstaculo1_a2.contains([x3;y3;z3]);
end
%% Obstáculo 2
if res==0
res=Obstaculo2_a2.contains([x3;y3;z3]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a2.contains([x3;y3;z3]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a2.contains([x3;y3;z3]);
end

%% Colisiones con punto 41 (en el eslabón 4) ampliando obstáculos
%% Obstáculo 2
```

```
if res==0
res=Obstaculo2_a4_int.contains([x41;y41;z41]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a4_int.contains([x41;y41;z41]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a4_int.contains([x41;y41;z41]);
end

%% Colisiones con punto 42 (en el eslabón 4) ampliando obstáculos
%% Obstáculo 2
if res==0
res=Obstaculo2_a4_int.contains([x42;y42;z42]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a4_int.contains([x42;y42;z42]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a4_int.contains([x42;y42;z42]);
end

%% Colisiones con punto 43 (en el eslabón 4) ampliando obstáculos
%% Obstáculo 2
if res==0
res=Obstaculo2_a4_int.contains([x43;y43;z43]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a4_int.contains([x43;y43;z43]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a4_int.contains([x43;y43;z43]);
end

%% Colisiones con punto 21 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
res=Obstaculo2_a2_int.contains([x21;y21;z21]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a2_int.contains([x21;y21;z21]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a2_int.contains([x21;y21;z21]);
end

%% Colisiones con punto 22 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
res=Obstaculo2_a2_int.contains([x22;y22;z22]);
```

```
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a2_int.contains([x22;y22;z22]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a2_int.contains([x22;y22;z22]);
end

%% Colisiones con punto 23 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
res=Obstaculo2_a2_int.contains([x23;y23;z23]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a2_int.contains([x23;y23;z23]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a2_int.contains([x23;y23;z23]);
end

%% Colisiones con punto 24 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
res=Obstaculo2_a2_int.contains([x24;y24;z24]);
end
%% Obstáculo 3
if res==0
res=Obstaculo3_a2_int.contains([x24;y24;z24]);
end
%% Obstáculo 4
if res==0
res=Obstaculo4_a2_int.contains([x24;y24;z24]);
end
end
```

10.1.3. Función Colision_Curva_MoveAbsJ_Obstaculos

```
function [res] = Colision_Curva_MoveAbsJ_Obstaculos(q0,qF,T,Suelo_a6,Env_1_Base_a6,
Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,
Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,
Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,
Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)
% Comprueba si existe colisión en la trayectoria entre una configuración
% inicial (q0) y una final (qF). Se deben pasar además los obstáculos
% ampliados para cada uno de los puntos del robot que se comprueban y una
% parámetro T, para la función Trayectoria_xyz

% Paso de grados a radianes
q0=q0*pi/180; qF=qF*pi/180;

% Vectores con los valores de las articualciones a lo largo de la
% trayectoria
[qv1,qv2,qv3,qv4,qv5,qv6]=moveAbsJ(q0,qF,T,0);

%Vectores de puntos del robot en coordenadas cartesianas
[x6,y6,z6,x45,y45,z45,x3,y3,z3,x41,y41,z41,x42,y42,z42,x43,y43,z43,x21,y21,z21,x22,
y22,z22,x23,y23,z23,x24,y24,z24] = Trayectoria_xyz(qv1(:,2),qv2(:,2),qv3(:,2),qv4
(:,2),qv5(:,2),qv6(:,2));

%% Colisiones con punto en el origen del sistema 6 ampliando obstáculos
res=0;
%% Suelo
vectorSuelo=Suelo_a6.contains([x6;y6;z6]);
vectorColisionSuelo=find(vectorSuelo);
N=size(vectorColisionSuelo);

if N(2)>0
    res=1;
end
%% Envolvente 1 de la Base
if res==0
vectoreE1Base=Env_1_Base_a6.contains([x6;y6;z6]);
vectorColisionE1Base=find(vectoreE1Base);
N=size(vectorColisionE1Base);

if N(2)>0
    res=1;
end
end
%% Envolvente 2 de la Base
if res==0
vectorE2Base=Env_2_Base_a6.contains([x6;y6;z6]);
vectorColsionE2Base=find(vectorE2Base);
N=size(vectorColsionE2Base);

if N(2)>0
    res=1;
end
end
%% Obstáculo 1
if res==0
vectorObs1=Obstaculo1_a6.contains([x6;y6;z6]);
vectorColsionObs1=find(vectorObs1);
```

```
N=size(vectorColsionObs1);

if N(2)>0
    res=1;
end
end
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a6.contains([x6;y6;z6]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a6.contains([x6;y6;z6]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a6.contains([x6;y6;z6]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end
%% Link 1
if res==0
vectorLink1=Choque_link1_a6.contains([x6;y6;z6]);
vectorColsionLink1=find(vectorLink1);
N=size(vectorColsionLink1);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto en el origen de los sistemas 4 y 5 ampliando obstáculos
%% Suelo
if res==0
vectorSuelo=Suelo_a4.contains([x45;y45;z45]);
vectorColisionSuelo=find(vectorSuelo);
N=size(vectorColisionSuelo);

if N(2)>0
    res=1;
end
end
```

```
%% Envolvente 1 de la Base
if res==0
vectoreE1Base=Env_1_Base_a4.contains([x45;y45;z45]);
vectorColisionE1Base=find(vectoreE1Base);
N=size(vectorColisionE1Base);

if N(2)>0
    res=1;
end
end
%% Envolvente 2 de la Base
if res==0
vectorE2Base=Env_2_Base_a4.contains([x45;y45;z45]);
vectorColsionE2Base=find(vectorE2Base);
N=size(vectorColsionE2Base);

if N(2)>0
    res=1;
end
end
%% Obstáculo 1
if res==0
vectorObs1=Obstaculo1_a4.contains([x45;y45;z45]);
vectorColsionObs1=find(vectorObs1);
N=size(vectorColsionObs1);

if N(2)>0
    res=1;
end
end
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a4.contains([x45;y45;z45]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a4.contains([x45;y45;z45]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a4.contains([x45;y45;z45]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
```

```
end
end
%% Link 1
if res==0
vectorLink1=Choque_link1_a4.contains([x45;y45;z45]);
vectorColsionLink1=find(vectorLink1);
N=size(vectorColsionLink1);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto en el origen de los sistemas 2 y 3 ampliando obstáculos
%% Obstáculo 1
if res==0
vectorObs1=Obstaculo1_a2.contains([x3;y3;z3]);
vectorColsionObs1=find(vectorObs1);
N=size(vectorColsionObs1);

if N(2)>0
    res=1;
end
end
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a2.contains([x3;y3;z3]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a2.contains([x3;y3;z3]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a2.contains([x3;y3;z3]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto 41 (en el eslabón 4) ampliando obstáculos
%% Obstáculo 2
if res==0
```

```
vectorObs2=Obstaculo2_a4_int.contains([x41;y41;z41]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a4_int.contains([x41;y41;z41]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a4_int.contains([x41;y41;z41]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto 42 (en el eslabón 4) ampliando obstáculos
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a4_int.contains([x42;y42;z42]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a4_int.contains([x42;y42;z42]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a4_int.contains([x42;y42;z42]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
```

```
end
end

%% Colisiones con punto 43 (en el eslabón 4) ampliando obstáculos
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a4_int.contains([x43;y43;z43]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a4_int.contains([x43;y43;z43]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a4_int.contains([x43;y43;z43]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto 21 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a2_int.contains([x21;y21;z21]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a2_int.contains([x21;y21;z21]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
```

```
vectorObs4=Obstaculo4_a2_int.contains([x21;y21;z21]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto 22 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a2_int.contains([x22;y22;z22]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a2_int.contains([x22;y22;z22]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a2_int.contains([x22;y22;z22]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto 23 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a2_int.contains([x23;y23;z23]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a2_int.contains([x23;y23;z23]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);
```

```
if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a2_int.contains([x23;y23;z23]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end

%% Colisiones con punto 24 (en el eslabón 2) ampliando obstáculos
%% Obstáculo 2
if res==0
vectorObs2=Obstaculo2_a2_int.contains([x24;y24;z24]);
vectorColsionObs2=find(vectorObs2);
N=size(vectorColsionObs2);

if N(2)>0
    res=1;
end
end
%% Obstáculo 3
if res==0
vectorObs3=Obstaculo3_a2_int.contains([x24;y24;z24]);
vectorColsionObs3=find(vectorObs3);
N=size(vectorColsionObs3);

if N(2)>0
    res=1;
end
end
%% Obstáculo 4
if res==0
vectorObs4=Obstaculo4_a2_int.contains([x24;y24;z24]);
vectorColsionObs4=find(vectorObs4);
N=size(vectorColsionObs4);

if N(2)>0
    res=1;
end
end
end
```

10.1.4. Función datosBrazo

```
function [smiData] = datosBrazo()
% Carga los datos del brazo para la simulación

% Simscape(TM) Multibody(TM) version: 7.1

% This is a model data file derived from a Simscape Multibody Import XML file using the smimport function.
% The data in this file sets the block parameter values in an imported Simscape Multibody model.
% For more information on this file, see the smimport function help page in the Simscape Multibody documentation.
% You can modify numerical values, but avoid any other changes to this file.
% Do not add code to this file. Do not edit the physical units shown in comments.

%%%VariableName:smiData


%============= RigidTransform =============%

%Initialize the RigidTransform structure array by filling in null values.
smiData.RigidTransform(7).translation = [0.0 0.0 0.0];
smiData.RigidTransform(7).angle = 0.0;
smiData.RigidTransform(7).axis = [0.0 0.0 0.0];
smiData.RigidTransform(7).ID = '';

%Translation Method - Cartesian
%Rotation Method - Arbitrary Axis
smiData.RigidTransform(1).translation = [0 0 0]; % mm
smiData.RigidTransform(1).angle = 0; % rad
smiData.RigidTransform(1).axis = [0 0 0];
smiData.RigidTransform(1).ID = 'RootGround[IRB140_-_M2004C_LINK1-1]';

%Translation Method - Cartesian
%Rotation Method - Arbitrary Axis
smiData.RigidTransform(2).translation = [0 0 0]; % mm
smiData.RigidTransform(2).angle = 0; % rad
smiData.RigidTransform(2).axis = [0 0 0];
smiData.RigidTransform(2).ID = 'RootGround[IRB140_-_M2004C_LINK3-1]';

%Translation Method - Cartesian
%Rotation Method - Arbitrary Axis
smiData.RigidTransform(3).translation = [0 0 0]; % mm
smiData.RigidTransform(3).angle = 0; % rad
smiData.RigidTransform(3).axis = [0 0 0];
smiData.RigidTransform(3).ID = 'RootGround[IRB140_-_M2004C_BASE-1]';

%Translation Method - Cartesian
%Rotation Method - Arbitrary Axis
smiData.RigidTransform(4).translation = [0 0 0]; % mm
smiData.RigidTransform(4).angle = 0; % rad
smiData.RigidTransform(4).axis = [0 0 0];
smiData.RigidTransform(4).ID = 'RootGround[IRB140_-_M2004C_LINK2-1]';

%Translation Method - Cartesian
%Rotation Method - Arbitrary Axis
smiData.RigidTransform(5).translation = [0 0 0]; % mm
smiData.RigidTransform(5).angle = 0; % rad
smiData.RigidTransform(5).axis = [0 0 0];
```

```
smiData.RigidTransform(5).ID  = 'RootGround[IRB140_-_M2004C_LINK4-1]';

%Translation Method - Cartesian
%Rotation Method - Arbitrary Axis
smiData.RigidTransform(6).translation = [0 0 0]; % mm
smiData.RigidTransform(6).angle = 0; % rad
smiData.RigidTransform(6).axis = [0 0 0];
smiData.RigidTransform(6).ID = 'RootGround[IRB140_-_M2004C_LINK5-1]';

%Translation Method - Cartesian
%Rotation Method - Arbitrary Axis
smiData.RigidTransform(7).translation = [0 0 0]; % mm
smiData.RigidTransform(7).angle = 0; % rad
smiData.RigidTransform(7).axis = [0 0 0];
smiData.RigidTransform(7).ID = 'RootGround[IRB140_-_M2004C_LINK6-1]';


%============= Solid =============%
%Center of Mass (CoM) %Moments of Inertia (MoI) %Product of Inertia (PoI)

%Initialize the Solid structure array by filling in null values.
smiData.Solid(7).mass = 0.0;
smiData.Solid(7).CoM = [0.0 0.0 0.0];
smiData.Solid(7).MoI = [0.0 0.0 0.0];
smiData.Solid(7).PoI = [0.0 0.0 0.0];
smiData.Solid(7).color = [0.0 0.0 0.0];
smiData.Solid(7).opacity  =  0.0;
smiData.Solid(7).ID = '';

%Inertia Type - Custom
%Visual Properties - Simple
smiData.Solid(1).mass = 16.287755696601998; % kg
smiData.Solid(1).CoM = [28.105273519553084 43.814591293645414 263.40402176423112];
% mm
smiData.Solid(1).MoI = [239763.08889101553 217382.94278782036 215279.59384058954];
% kg*mm^2
smiData.Solid(1).PoI = [-32460.24566868474 -24709.081027452092
-698.68992333262452];  % kg*mm^2
smiData.Solid(1).color = [0.8901960784313725 0.41176470588235292
0.12549019607843137];
smiData.Solid(1).opacity = 1;
smiData.Solid(1).ID  = 'IRB140_-_M2004C_LINK1*:*Predeterminado';

%Inertia Type - Custom
%Visual Properties - Simple
smiData.Solid(2).mass = 8.0307275402001661; % kg
smiData.Solid(2).CoM = [89.584077364104729 7.5078139538129589 706.32196409286962];
% mm
smiData.Solid(2).MoI = [32281.919308269764 94134.447971653353 86243.960368176864];
% kg*mm^2
smiData.Solid(2).PoI = [363.12165513126956 1893.0409867465703 3018.1264759250676];
% kg*mm^2
smiData.Solid(2).color = [0.8901960784313725 0.41176470588235292
0.12549019607843137];
smiData.Solid(2).opacity = 1;
smiData.Solid(2).ID  = 'IRB140_-_M2004C_LINK3*:*Predeterminado';

%Inertia Type - Custom
```

```
%Visual Properties - Simple
smiData.Solid(3).mass = 12.409016159487051; % kg
smiData.Solid(3).CoM = [-82.039325168842566 0.40202864022401569 59.92473434822454];
% mm
smiData.Solid(3).MoI = [104502.17600625273 167277.84258504011 246486.76406597739];
% kg*mm^2
smiData.Solid(3).PoI = [843.00681015731163 7342.8465567048279 -10.052082753011042];
% kg*mm^2
smiData.Solid(3).color = [0.8901960784313725 0.41176470588235292
0.12549019607843137];
smiData.Solid(3).opacity = 1;
smiData.Solid(3).ID  = 'IRB140_-_M2004C_BASE*:*Predeterminado';

%Inertia Type - Custom
%Visual Properties - Simple
smiData.Solid(4).mass = 7.5803742448443803; % kg
smiData.Solid(4).CoM = [60.272232866162291 -92.433716544259397 550.28801247688648];
% mm
smiData.Solid(4).MoI = [155736.57840372325 131499.06285406949 44937.400606996707];
% kg*mm^2
smiData.Solid(4).PoI = [17977.259799324627 -1829.2494088037963 516.10004314501316];
% kg*mm^2
smiData.Solid(4).color = [0.8901960784313725 0.41176470588235292
0.12549019607843137];
smiData.Solid(4).opacity = 1;
smiData.Solid(4).ID  = 'IRB140_-_M2004C_LINK2*:*Predeterminado';

%Inertia Type - Custom
%Visual Properties - Simple
smiData.Solid(5).mass = 1.7702535337978553; % kg
smiData.Solid(5).CoM = [381.56768070650725 -1.2122512768642943 710.56671529737207];
% mm
smiData.Solid(5).MoI = [3458.860795040453 7274.1725518497315 8147.9589101452411];
% kg*mm^2
smiData.Solid(5).PoI = [3.0728812037748474 -58.991965804051041 204.64630008952696];
% kg*mm^2
smiData.Solid(5).color = [0.8901960784313725 0.41176470588235292
0.12549019607843137];
smiData.Solid(5).opacity = 1;
smiData.Solid(5).ID  = 'IRB140_-_M2004C_LINK4*:*Predeterminado';

%Inertia Type - Custom
%Visual Properties - Simple
smiData.Solid(6).mass = 0.14555277547051354; % kg
smiData.Solid(6).CoM = [448.6176145859244 0.61691241678137698 711.83816363655342];
% mm
smiData.Solid(6).MoI = [71.483038538417432 112.13448276901782 68.837706950697012];
% kg*mm^2
smiData.Solid(6).PoI = [-0.010630848682244265 0.41107005061240870
-0.12391749873160389];  % kg*mm^2
smiData.Solid(6).color = [0.8901960784313725 0.41176470588235292
0.12549019607843137];
smiData.Solid(6).opacity = 1;
smiData.Solid(6).ID  = 'IRB140_-_M2004C_LINK5*:*Predeterminado';

%Inertia Type - Custom
%Visual Properties - Simple
smiData.Solid(7).mass = 0.025865432871949739; % kg
```

```
smiData.Solid(7).CoM = [502.05943284509937 2.3878507882271841e-07
711.77496714351128];  % mm
smiData.Solid(7).MoI = [5.9822143548355609 6.1747932732778752 6.2921914507959009];
% kg*mm^2
smiData.Solid(7).PoI = [6.7562637062982023e-07 0.045824448440177437
3.330919068594962e-07];  % kg*mm^2
smiData.Solid(7).color = [0.79607843137254897 0.82352941176470584
0.93333333333333335];
smiData.Solid(7).opacity = 1;
smiData.Solid(7).ID  = 'IRB140_-_M2004C_LINK6*:*Predeterminado';


end
```

10.1.5. Función DegToRad

```
function [solucion] = DegToRad(angulo)
% Pasa de grados a radianes
solucion=angulo*pi/180; end
```

10.1.6. Función dibujar_trayectoria

```
function [] = dibujar_trayectoria(q0,qF,color,width)
% Dibuja una trayectoria en el espacio cartesiano a partir de la
% configuración inicial (q0), la configuración final (qF), el color de la
% línea (color) y su grosor (width)

% Paso de grados a radianes
q0=q0*pi/180; qF=qF*pi/180;

% Vectores de puntos con los valores de las articulaciones a lo largo de la
% trayectoria
[qv1,qv2,qv3,qv4,qv5,qv6]=moveAbsJ(q0,qF,1,1);

%% Parámetros Denavit-Hartenberg
theta_1=qv1(:,2);
theta_2=qv2(:,2)-pi/2;
theta_3=qv3(:,2);
theta_4=qv4(:,2);
theta_5=qv5(:,2);
theta_6=qv6(:,2);

d_1=352;
d_2=0;
d_3=0;
d_4=380;
d_5=0;
d_6=65;

a_1=70;
a_2=360;
a_3=0;
a_4=0;
a_5=0;
a_6=0;

alpha_1=-pi/2;
alpha_2=0;
alpha_3=-pi/2;
alpha_4=pi/2;
alpha_5=-pi/2;
alpha_6=0;

%% Vectores para alamecenar los puntos de la trayectoria del extremo en
coordendadas cartesianas
x6=[];
y6=[];
z6=[];

%% Matrices de transformación
N=size(qv1,1);

for i=1:1:N

T_01=MatrizTransformacion(theta_1(i),d_1,a_1,alpha_1);
T_12=MatrizTransformacion(theta_2(i),d_2,a_2,alpha_2);
T_23=MatrizTransformacion(theta_3(i),d_3,a_3,alpha_3);
T_34=MatrizTransformacion(theta_4(i),d_4,a_4,alpha_4);
T_45=MatrizTransformacion(theta_5(i),d_5,a_5,alpha_5);
```

```
T_56=MatrizTransformacion(theta_6(i),d_6,a_6,alpha_6);

T_02=T_01*T_12;
T_03=T_02*T_23;
T_04=T_03*T_34;
T_05=T_04*T_45;
T_06=T_05*T_56;

%% Valores x, y, z del extremo
x6=[x6,T_06(1,4)];
y6=[y6,T_06(2,4)];
z6=[z6,T_06(3,4)];
end

%% Representación
plot3(x6,y6,z6,'color',color,'LineWidth',width);

end
```

10.1.7. Función DibujarPuntos6D

```
function [] = DibujarPuntos6D(T,long,signo,n)
% Dibuja el punto del extremo con sus ejes x,y,z de longitud long a partir
% de dicha longiutd, la matriz de transformación (T), el tipo de punto
% (signo) y el grosor para los ejes (n)

%% Coordenadas x y z obtenidas de la matriz de transformación
tras=T(1:3,4);
Punto=T(1:3,4);

%% Extremos de los ejes x, y, z de longitud long sistuados en el origen
Punto1x=[long 0 0];
Punto1y=[0 long 0];
Punto1z=[0 0 long];

%% Submatriz de rotación obtenida de la matriz de transformación
Rotacion=T(1:3,1:3);

%% Extremos de los ejes x, y, z de longitud long aplicando la rotación y traslación
dadas por la matriz de transformación
Punto2x=Rotacion*Punto1x'+tras;
Punto2y=Rotacion*Punto1y'+tras;
Punto2z=Rotacion*Punto1z'+tras;

%% Representación
% Punto donde se sitúa el extremo
plot3(Punto(1),Punto(2),Punto(3),signo,'color','m','lineWidth',n);
% Eje x
X=[Punto(1);Punto2x(1)];
Y=[Punto(2);Punto2x(2)];
Z=[Punto(3);Punto2x(3)];
line(X,Y,Z,'Color','red','lineWidth',n);
%Eje Y
X=[Punto(1);Punto2y(1)];
Y=[Punto(2);Punto2y(2)];
Z=[Punto(3);Punto2y(3)];
line(X,Y,Z,'Color','green','lineWidth',n);
%Eje Z
X=[Punto(1);Punto2z(1)];
Y=[Punto(2);Punto2z(2)];
Z=[Punto(3);Punto2z(3)];
line(X,Y,Z,'Color','blue','lineWidth',n);
end
```

10.1.8. Función MatrizTransformacion

```
function [Matriz] = MatrizTransformacion(theta,d,a,alpha)
% Devuelve la matriz de transformación a partir de los parámetros
% Denavit-Hartenberg
Matriz=[cos(theta) -sin(theta) 0 0; sin(theta) cos(theta) 0 0;0 0 1 0;0 0 0 1]*[1 0
0 a;0 1 0 0;0 0 1 d;0 0 0 1]*[1 0 0 0;0 cos(alpha) -sin(alpha) 0;0 sin(alpha) cos
(alpha) 0;0 0 0 1];
end
```

10.1.9. Función moveAbsJ

```
function [q1,q2,q3,q4,q5,q6] = moveAbsJ(q0,qF,T,num)
% Devuelve una trayectoria cúbica coordinando el movimento absoluto de los
% ejes a partir de una configuración inicial (q0) otra fianl (qT) un periodo
% (T)-calcula una configuración cada 5 ms, por lo que a mayor T, mayor
% número de configuraciones intermedias- y el tiempo inicial al comenzar la
% trayectoria para la simulación(num)

%% Cálculo de los coeficientes de la trayectoria cúbica
a=[];
b=[];
c=[];
d=[];
for i=1:6
    a(i)=-2*(qF(i)-q0(i))/T^3;
    b(i)=3*(qF(i)-q0(i))/T^2;
    c(i)=0;
    d(i)=q0(i);
end

%% Configuraciones a lo largo de la trayectoria
t = (0:0.005:T)';
q1=[t+num*T,a(1)*t.^3+b(1)*t.^2+c(1)*t+d(1)];
q2=[t+num*T,a(2)*t.^3+b(2)*t.^2+c(2)*t+d(2)];
q3=[t+num*T,a(3)*t.^3+b(3)*t.^2+c(3)*t+d(3)];
q4=[t+num*T,a(4)*t.^3+b(4)*t.^2+c(4)*t+d(4)];
q5=[t+num*T,a(5)*t.^3+b(5)*t.^2+c(5)*t+d(5)];
q6=[t+num*T,a(6)*t.^3+b(6)*t.^2+c(6)*t+d(6)];
end
```

10.1.10. *Script* para declarar los obstáculos

```
%% Cálculo y representación de los obstáculos sin ampliar y ampliados para los distintos puntos del robot

intervaloBase=10;
intervaloLink1=10;

%% Envolvente 1 de la base
j=0;
x=NaN(1,180/intervaloBase+1);
y=NaN(1,180/intervaloBase+1);
for i=-90:intervaloBase:90
    j=j+1;
    x(j)=201*cosd(i);
    y(j)=201*sind(i);
end

vertices=[-274 201 0;-274 -201 0;x' y' zeros(180/intervaloBase+1,1);-274 201 117;-274 -201 117;x' y' 117.*ones(180/intervaloBase+1,1)];
Env_1_Base=Polyhedron(vertices);

%% Envolvente 2 de la base
t_2_base=[-235 0 153/2];

d_2_base=[155 300 153];

v_e2_base=zeros(8,4);
v_e2_base(1,1:4)=[d_2_base(1) d_2_base(2) -d_2_base(3) 2]/2;
v_e2_base(2,1:4)=[d_2_base(1) -d_2_base(2) -d_2_base(3) 2]/2;
v_e2_base(3,1:4)=[-d_2_base(1) d_2_base(2) -d_2_base(3) 2]/2;
v_e2_base(4,1:4)=[-d_2_base(1) -d_2_base(2) -d_2_base(3) 2]/2;
v_e2_base(5,1:4)=[d_2_base(1) d_2_base(2) d_2_base(3) 2]/2;
v_e2_base(6,1:4)=[d_2_base(1) -d_2_base(2) d_2_base(3) 2]/2;
v_e2_base(7,1:4)=[-d_2_base(1) d_2_base(2) d_2_base(3) 2]/2;
v_e2_base(8,1:4)=[-d_2_base(1) -d_2_base(2) d_2_base(3) 2]/2;

Trans_e2_base=[1 0 0 t_2_base(1);0 1 0 t_2_base(2);0 0 1 t_2_base(3);0 0 0 1];
v_e2_base_t=zeros(4,8);
for i=1:8
    v_e2_base_t(1:4,i)=Trans_e2_base*(v_e2_base(i,1:4))';
end

v_e2_base=zeros(8,3);
for i=1:8
   v_e2_base(i,1:3)=v_e2_base_t(1:3,i);
end

Env_2_Base=Polyhedron(v_e2_base);

%% Obstáculo 1
t_obs_1=[0 600+400/2 1000/2];

d_obs_1=[2000 400 1000];

v_obs_1=zeros(8,4);
v_obs_1(1,1:4)=[d_obs_1(1) d_obs_1(2) -d_obs_1(3) 2]/2;
v_obs_1(2,1:4)=[d_obs_1(1) -d_obs_1(2) -d_obs_1(3) 2]/2;
v_obs_1(3,1:4)=[-d_obs_1(1) d_obs_1(2) -d_obs_1(3) 2]/2;
v_obs_1(4,1:4)=[-d_obs_1(1) -d_obs_1(2) -d_obs_1(3) 2]/2;
```

```
v_obs_1(5,1:4)=[d_obs_1(1) d_obs_1(2) d_obs_1(3) 2]/2;
v_obs_1(6,1:4)=[d_obs_1(1) -d_obs_1(2) d_obs_1(3) 2]/2;
v_obs_1(7,1:4)=[-d_obs_1(1) d_obs_1(2) d_obs_1(3) 2]/2;
v_obs_1(8,1:4)=[-d_obs_1(1) -d_obs_1(2) d_obs_1(3) 2]/2;

Trans_obs_1=[1 0 0 t_obs_1(1);0 1 0 t_obs_1(2);0 0 1 t_obs_1(3);0 0 0 1];
v_obs_1_t=zeros(4,8);
for i=1:8
    v_obs_1_t(1:4,i)=Trans_obs_1*(v_obs_1(i,1:4))';
end

v_obs_1=zeros(8,3);
for i=1:8
   v_obs_1(i,1:3)=v_obs_1_t(1:3,i);
end

Obstaculo1=Polyhedron(v_obs_1);

%% Obstáculo 2
t_obs_2=[700 0 300/2];

d_obs_2=[500 800 300];

v_obs_2=zeros(8,4);
v_obs_2(1,1:4)=[d_obs_2(1) d_obs_2(2) -d_obs_2(3) 2]/2;
v_obs_2(2,1:4)=[d_obs_2(1) -d_obs_2(2) -d_obs_2(3) 2]/2;
v_obs_2(3,1:4)=[-d_obs_2(1) d_obs_2(2) -d_obs_2(3) 2]/2;
v_obs_2(4,1:4)=[-d_obs_2(1) -d_obs_2(2) -d_obs_2(3) 2]/2;
v_obs_2(5,1:4)=[d_obs_2(1) d_obs_2(2) d_obs_2(3) 2]/2;
v_obs_2(6,1:4)=[d_obs_2(1) -d_obs_2(2) d_obs_2(3) 2]/2;
v_obs_2(7,1:4)=[-d_obs_2(1) d_obs_2(2) d_obs_2(3) 2]/2;
v_obs_2(8,1:4)=[-d_obs_2(1) -d_obs_2(2) d_obs_2(3) 2]/2;

Trans_obs_2=[1 0 0 t_obs_2(1);0 1 0 t_obs_2(2);0 0 1 t_obs_2(3);0 0 0 1];
v_obs_1_t=zeros(4,8);
for i=1:8
    v_obs_2_t(1:4,i)=Trans_obs_2*(v_obs_2(i,1:4))';
end

v_obs_2=zeros(8,3);
for i=1:8
   v_obs_2(i,1:3)=v_obs_2_t(1:3,i);
end

Obstaculo2=Polyhedron(v_obs_2);

%% Obstáculo 3
t_obs_3=[-700 -500 1000/2];

d_obs_3=[300 300 1000];

v_obs_3=zeros(8,4);
v_obs_3(1,1:4)=[d_obs_3(1) d_obs_3(2) -d_obs_3(3) 2]/2;
v_obs_3(2,1:4)=[d_obs_3(1) -d_obs_3(2) -d_obs_3(3) 2]/2;
v_obs_3(3,1:4)=[-d_obs_3(1) d_obs_3(2) -d_obs_3(3) 2]/2;
v_obs_3(4,1:4)=[-d_obs_3(1) -d_obs_3(2) -d_obs_3(3) 2]/2;
v_obs_3(5,1:4)=[d_obs_3(1) d_obs_3(2) d_obs_3(3) 2]/2;
v_obs_3(6,1:4)=[d_obs_3(1) -d_obs_3(2) d_obs_3(3) 2]/2;
```

```
v_obs_3(7,1:4)=[-d_obs_3(1) d_obs_3(2) d_obs_3(3) 2]/2;
v_obs_3(8,1:4)=[-d_obs_3(1) -d_obs_3(2) d_obs_3(3) 2]/2;

Trans_obs_3=[1 0 0 t_obs_3(1);0 1 0 t_obs_3(2);0 0 1 t_obs_3(3);0 0 0 1];
v_obs_3_t=zeros(4,8);
for i=1:8
    v_obs_3_t(1:4,i)=Trans_obs_3*(v_obs_3(i,1:4))';
end

v_obs_3=zeros(8,3);
for i=1:8
   v_obs_3(i,1:3)=v_obs_3_t(1:3,i);
end

Obstaculo3=Polyhedron(v_obs_3);

%% Obstáculo 4
t_obs_4=[0 -600 500/2];

d_obs_4=[100 300 500];

v_obs_4=zeros(8,4);
v_obs_4(1,1:4)=[d_obs_4(1) d_obs_4(2) -d_obs_4(3) 2]/2;
v_obs_4(2,1:4)=[d_obs_4(1) -d_obs_4(2) -d_obs_4(3) 2]/2;
v_obs_4(3,1:4)=[-d_obs_4(1) d_obs_4(2) -d_obs_4(3) 2]/2;
v_obs_4(4,1:4)=[-d_obs_4(1) -d_obs_4(2) -d_obs_4(3) 2]/2;
v_obs_4(5,1:4)=[d_obs_4(1) d_obs_4(2) d_obs_4(3) 2]/2;
v_obs_4(6,1:4)=[d_obs_4(1) -d_obs_4(2) d_obs_4(3) 2]/2;
v_obs_4(7,1:4)=[-d_obs_4(1) d_obs_4(2) d_obs_4(3) 2]/2;
v_obs_4(8,1:4)=[-d_obs_4(1) -d_obs_4(2) d_obs_4(3) 2]/2;

Trans_obs_4=[1 0 0 t_obs_4(1);0 1 0 t_obs_4(2);0 0 1 t_obs_4(3);0 0 0 1];
v_obs_4_t=zeros(4,8);
for i=1:8
    v_obs_4_t(1:4,i)=Trans_obs_4*(v_obs_4(i,1:4))';
end

v_obs_4=zeros(8,3);
for i=1:8
   v_obs_4(i,1:3)=v_obs_4_t(1:3,i);
end

Obstaculo4=Polyhedron(v_obs_4);

%% Choque link 1
j=0;
x=NaN(1,360/intervaloLink1+1);
y=NaN(1,360/intervaloLink1+1);
for  i=-180:intervaloLink1:180
    j=j+1;
    x(j)=230*cosd(i);
    y(j)=230*sind(i);
end

vertices=[x' y' 117.*ones(360/intervaloLink1+1,1);x' y' (117+150).*ones
(360/intervaloLink1+1,1)];
Choque_link1=Polyhedron(vertices);
```

```
%% Suelo
dist=1000;
Suelo=Polyhedron([dist dist 0;dist -dist 0;-dist dist 0;-dist -dist 0;dist dist
-1000;dist -dist -1000;-dist dist -1000;-dist -dist -1000]);



%% AMPLIACIÓN PARA EL LINK 6

a6=30; % margen para ampliar

%% Envolvente 1 de la base
j=0;
x=NaN(1,180/intervaloBase+1);
y=NaN(1,180/intervaloBase+1);
for  i=-90:intervaloBase:90
    j=j+1;
    x(j)=(201+a6)*cosd(i);
    y(j)=(201+a6)*sind(i);
end

vertices=[-274-a6 201+a6 0;-274-a6 -201-a6 0;x' y' zeros(180/intervaloBase+1,1);
-274-a6 201+a6 117+a6;-274-a6 -201-a6 117+a6;x' y' (117+a6).*ones
(180/intervaloBase+1,1)];
Env_1_Base_a6=Polyhedron(vertices);

%% Suelo
Suelo_a6=Polyhedron([dist dist a6;dist -dist a6;-dist dist a6;-dist -dist a6;dis
dist -1000;dist -dist -1000;-dist dist -1000;-dist -dist -1000]);

%% Choque link 1
j=0;
x=NaN(1,360/intervaloLink1+1);
y=NaN(1,360/intervaloLink1+1);
for  i=-180:intervaloLink1:180
    j=j+1;
    x(j)=(230+a6)*cosd(i);
    y(j)=(230+a6)*sind(i);
end

vertices=[x' y' 117.*ones(360/intervaloLink1+1,1);x' y' (117+150+a6).*ones
(360/intervaloLink1+1,1)];
Choque_link1_a6=Polyhedron(vertices);

%% Envolvente 2 de la base
a6=2*a6;

v_e2_base=zeros(8,4);
v_e2_base(1,1:4)=[d_2_base(1)+a6 d_2_base(2)+a6 -d_2_base(3)-a6 2]/2;
v_e2_base(2,1:4)=[d_2_base(1)+a6 -d_2_base(2)-a6 -d_2_base(3)-a6 2]/2;
v_e2_base(3,1:4)=[-d_2_base(1)-a6 d_2_base(2)+a6 -d_2_base(3)-a6 2]/2;
v_e2_base(4,1:4)=[-d_2_base(1)-a6 -d_2_base(2)-a6 -d_2_base(3)-a6 2]/2;
v_e2_base(5,1:4)=[d_2_base(1)+a6 d_2_base(2)+a6 d_2_base(3)+a6 2]/2;
v_e2_base(6,1:4)=[d_2_base(1)+a6 -d_2_base(2)-a6 d_2_base(3)+a6 2]/2;
v_e2_base(7,1:4)=[-d_2_base(1)-a6 d_2_base(2)+a6 d_2_base(3)+a6 2]/2;
v_e2_base(8,1:4)=[-d_2_base(1)-a6 -d_2_base(2)-a6 d_2_base(3)+a6 2]/2;

Trans_e2_base=[1 0 0 t_2_base(1);0 1 0 t_2_base(2);0 0 1 t_2_base(3);0 0 0 1];
```

```
v_e2_base_t=zeros(4,8);
for i=1:8
    v_e2_base_t(1:4,i)=Trans_e2_base*(v_e2_base(i,1:4))';
end

v_e2_base=zeros(8,3);
for i=1:8
   v_e2_base(i,1:3)=v_e2_base_t(1:3,i);
end

Env_2_Base_a6=Polyhedron(v_e2_base);

%% Obstáculo 1
v_obs_1=zeros(8,4);
v_obs_1(1,1:4)=[d_obs_1(1)+a6 d_obs_1(2)+a6 -d_obs_1(3)-a6 2]/2;
v_obs_1(2,1:4)=[d_obs_1(1)+a6 -d_obs_1(2)-a6 -d_obs_1(3)-a6 2]/2;
v_obs_1(3,1:4)=[-d_obs_1(1)-a6 d_obs_1(2)+a6 -d_obs_1(3)-a6 2]/2;
v_obs_1(4,1:4)=[-d_obs_1(1)-a6 -d_obs_1(2)-a6 -d_obs_1(3)-a6 2]/2;
v_obs_1(5,1:4)=[d_obs_1(1)+a6 d_obs_1(2)+a6 d_obs_1(3)+a6 2]/2;
v_obs_1(6,1:4)=[d_obs_1(1)+a6 -d_obs_1(2)-a6 d_obs_1(3)+a6 2]/2;
v_obs_1(7,1:4)=[-d_obs_1(1)-a6 d_obs_1(2)+a6 d_obs_1(3)+a6 2]/2;
v_obs_1(8,1:4)=[-d_obs_1(1)-a6 -d_obs_1(2)-a6 d_obs_1(3)+a6 2]/2;

Trans_obs_1=[1 0 0 t_obs_1(1);0 1 0 t_obs_1(2);0 0 1 t_obs_1(3);0 0 0 1];
v_obs_1_t=zeros(4,8);
for i=1:8
    v_obs_1_t(1:4,i)=Trans_obs_1*(v_obs_1(i,1:4))';
end

v_obs_1=zeros(8,3);
for i=1:8
   v_obs_1(i,1:3)=v_obs_1_t(1:3,i);
end

Obstaculo1_a6=Polyhedron(v_obs_1);

%% Obstáculo 2
v_obs_2=zeros(8,4);
v_obs_2(1,1:4)=[d_obs_2(1)+a6 d_obs_2(2)+a6 -d_obs_2(3)-a6 2]/2;
v_obs_2(2,1:4)=[d_obs_2(1)+a6 -d_obs_2(2)-a6 -d_obs_2(3)-a6 2]/2;
v_obs_2(3,1:4)=[-d_obs_2(1)-a6 d_obs_2(2)+a6 -d_obs_2(3)-a6 2]/2;
v_obs_2(4,1:4)=[-d_obs_2(1)-a6 -d_obs_2(2)-a6 -d_obs_2(3)-a6 2]/2;
v_obs_2(5,1:4)=[d_obs_2(1)+a6 d_obs_2(2)+a6 d_obs_2(3)+a6 2]/2;
v_obs_2(6,1:4)=[d_obs_2(1)+a6 -d_obs_2(2)-a6 d_obs_2(3)+a6 2]/2;
v_obs_2(7,1:4)=[-d_obs_2(1)-a6 d_obs_2(2)+a6 d_obs_2(3)+a6 2]/2;
v_obs_2(8,1:4)=[-d_obs_2(1)-a6 -d_obs_2(2)-a6 d_obs_2(3)+a6 2]/2;

Trans_obs_2=[1 0 0 t_obs_2(1);0 1 0 t_obs_2(2);0 0 1 t_obs_2(3);0 0 0 1];
v_obs_2_t=zeros(4,8);
for i=1:8
    v_obs_2_t(1:4,i)=Trans_obs_2*(v_obs_2(i,1:4))';
end

v_obs_2=zeros(8,3);
for i=1:8
   v_obs_2(i,1:3)=v_obs_2_t(1:3,i);
end
```

```
Obstaculo2_a6=Polyhedron(v_obs_2);

%% Obstáculo 3
v_obs_3=zeros(8,4);
v_obs_3(1,1:4)=[d_obs_3(1)+a6 d_obs_3(2)+a6 -d_obs_3(3)-a6 2]/2;
v_obs_3(2,1:4)=[d_obs_3(1)+a6 -d_obs_3(2)-a6 -d_obs_3(3)-a6 2]/2;
v_obs_3(3,1:4)=[-d_obs_3(1)-a6 d_obs_3(2)+a6 -d_obs_3(3)-a6 2]/2;
v_obs_3(4,1:4)=[-d_obs_3(1)-a6 -d_obs_3(2)-a6 -d_obs_3(3)-a6 2]/2;
v_obs_3(5,1:4)=[d_obs_3(1)+a6 d_obs_3(2)+a6 d_obs_3(3)+a6 2]/2;
v_obs_3(6,1:4)=[d_obs_3(1)+a6 -d_obs_3(2)-a6 d_obs_3(3)+a6 2]/2;
v_obs_3(7,1:4)=[-d_obs_3(1)-a6 d_obs_3(2)+a6 d_obs_3(3)+a6 2]/2;
v_obs_3(8,1:4)=[-d_obs_3(1)-a6 -d_obs_3(2)-a6 d_obs_3(3)+a6 2]/2;

Trans_obs_3=[1 0 0 t_obs_3(1);0 1 0 t_obs_3(2);0 0 1 t_obs_3(3);0 0 0 1];
v_obs_3_t=zeros(4,8);
for i=1:8
    v_obs_3_t(1:4,i)=Trans_obs_3*(v_obs_3(i,1:4))';
end

v_obs_3=zeros(8,3);
for i=1:8
   v_obs_3(i,1:3)=v_obs_3_t(1:3,i);
end

Obstaculo3_a6=Polyhedron(v_obs_3);

%% Obstáculo 4
v_obs_4=zeros(8,4);
v_obs_4(1,1:4)=[d_obs_4(1)+a6 d_obs_4(2)+a6 -d_obs_4(3)-a6 2]/2;
v_obs_4(2,1:4)=[d_obs_4(1)+a6 -d_obs_4(2)-a6 -d_obs_4(3)-a6 2]/2;
v_obs_4(3,1:4)=[-d_obs_4(1)-a6 d_obs_4(2)+a6 -d_obs_4(3)-a6 2]/2;
v_obs_4(4,1:4)=[-d_obs_4(1)-a6 -d_obs_4(2)-a6 -d_obs_4(3)-a6 2]/2;
v_obs_4(5,1:4)=[d_obs_4(1)+a6 d_obs_4(2)+a6 d_obs_4(3)+a6 2]/2;
v_obs_4(6,1:4)=[d_obs_4(1)+a6 -d_obs_4(2)-a6 d_obs_4(3)+a6 2]/2;
v_obs_4(7,1:4)=[-d_obs_4(1)-a6 d_obs_4(2)+a6 d_obs_4(3)+a6 2]/2;
v_obs_4(8,1:4)=[-d_obs_4(1)-a6 -d_obs_4(2)-a6 d_obs_4(3)+a6 2]/2;

Trans_obs_4=[1 0 0 t_obs_4(1);0 1 0 t_obs_4(2);0 0 1 t_obs_4(3);0 0 0 1];
v_obs_4_t=zeros(4,8);
for i=1:8
    v_obs_4_t(1:4,i)=Trans_obs_4*(v_obs_4(i,1:4))';
end

v_obs_4=zeros(8,3);
for i=1:8
   v_obs_4(i,1:3)=v_obs_4_t(1:3,i);
end

Obstaculo4_a6=Polyhedron(v_obs_4);



%% AMPLIACIÓN PARA EL LINK 4-5

a4=80; % margen para ampliar

%% Envolvente 1 de la base
j=0;
```

```
x=NaN(1,180/intervaloBase+1);
y=NaN(1,180/intervaloBase+1);
for i=-90:intervaloBase:90
    j=j+1;
    x(j)=(201+a4)*cosd(i);
    y(j)=(201+a4)*sind(i);
end

vertices=[-274-a4 201+a4 0;-274-a4 -201-a4 0;x' y' zeros(180/intervaloBase+1,1);
-274-a4 201+a4 117+a4;-274-a4 -201-a4 117+a4;x' y' (117+a4).*ones
(180/intervaloBase+1,1)];
Env_1_Base_a4=Polyhedron(vertices);
%% Suelo
Suelo_a4=Polyhedron([dist dist a4;dist -dist a4;-dist dist a4;-dist -dist a4;dis
dist -1000;dist -dist -1000;-dist dist -1000;-dist -dist -1000]);

%% Choque link 1
j=0;
x=NaN(1,360/intervaloLink1+1);
y=NaN(1,360/intervaloLink1+1);
for  i=-180:intervaloLink1:180
    j=j+1;
    x(j)=(230+a4)*cosd(i);
    y(j)=(230+a4)*sind(i);
end

vertices=[x' y' 117.*ones(360/intervaloLink1+1,1);x' y' (117+150+a4).*ones
(360/intervaloLink1+1,1)];
Choque_link1_a4=Polyhedron(vertices);

%% Envolvente 2 de la base
a4=2*a4;

v_e2_base=zeros(8,4);
v_e2_base(1,1:4)=[d_2_base(1)+a4 d_2_base(2)+a4 -d_2_base(3)-a4 2]/2;
v_e2_base(2,1:4)=[d_2_base(1)+a4 -d_2_base(2)-a4 -d_2_base(3)-a4 2]/2;
v_e2_base(3,1:4)=[-d_2_base(1)-a4 d_2_base(2)+a4 -d_2_base(3)-a4 2]/2;
v_e2_base(4,1:4)=[-d_2_base(1)-a4 -d_2_base(2)-a4 -d_2_base(3)-a4 2]/2;
v_e2_base(5,1:4)=[d_2_base(1)+a4 d_2_base(2)+a4 d_2_base(3)+a4 2]/2;
v_e2_base(6,1:4)=[d_2_base(1)+a4 -d_2_base(2)-a4 d_2_base(3)+a4 2]/2;
v_e2_base(7,1:4)=[-d_2_base(1)-a4 d_2_base(2)+a4 d_2_base(3)+a4 2]/2;
v_e2_base(8,1:4)=[-d_2_base(1)-a4 -d_2_base(2)-a4 d_2_base(3)+a4 2]/2;

Trans_e2_base=[1 0 0 t_2_base(1);0 1 0 t_2_base(2);0 0 1 t_2_base(3);0 0 0 1];
v_e2_base_t=zeros(4,8);
for i=1:8
    v_e2_base_t(1:4,i)=Trans_e2_base*(v_e2_base(i,1:4))';
end

v_e2_base=zeros(8,3);
for i=1:8
   v_e2_base(i,1:3)=v_e2_base_t(1:3,i);
end

Env_2_Base_a4=Polyhedron(v_e2_base);

%% Obstáculo 1
v_obs_1=zeros(8,4);
```

```
v_obs_1(1,1:4)=[d_obs_1(1)+a4 d_obs_1(2)+a4 -d_obs_1(3)-a4 2]/2;
v_obs_1(2,1:4)=[d_obs_1(1)+a4 -d_obs_1(2)-a4 -d_obs_1(3)-a4 2]/2;
v_obs_1(3,1:4)=[-d_obs_1(1)-a4 d_obs_1(2)+a4 -d_obs_1(3)-a4 2]/2;
v_obs_1(4,1:4)=[-d_obs_1(1)-a4 -d_obs_1(2)-a4 -d_obs_1(3)-a4 2]/2;
v_obs_1(5,1:4)=[d_obs_1(1)+a4 d_obs_1(2)+a4 d_obs_1(3)+a4 2]/2;
v_obs_1(6,1:4)=[d_obs_1(1)+a4 -d_obs_1(2)-a4 d_obs_1(3)+a4 2]/2;
v_obs_1(7,1:4)=[-d_obs_1(1)-a4 d_obs_1(2)+a4 d_obs_1(3)+a4 2]/2;
v_obs_1(8,1:4)=[-d_obs_1(1)-a4 -d_obs_1(2)-a4 d_obs_1(3)+a4 2]/2;

Trans_obs_1=[1 0 0 t_obs_1(1);0 1 0 t_obs_1(2);0 0 1 t_obs_1(3);0 0 0 1];
v_obs_1_t=zeros(4,8);
for i=1:8
    v_obs_1_t(1:4,i)=Trans_obs_1*(v_obs_1(i,1:4))';
end

v_obs_1=zeros(8,3);
for i=1:8
   v_obs_1(i,1:3)=v_obs_1_t(1:3,i);
end

Obstaculo1_a4=Polyhedron(v_obs_1);
%% Obstáculo 2
v_obs_2=zeros(8,4);
v_obs_2(1,1:4)=[d_obs_2(1)+a4 d_obs_2(2)+a4 -d_obs_2(3)-a4 2]/2;
v_obs_2(2,1:4)=[d_obs_2(1)+a4 -d_obs_2(2)-a4 -d_obs_2(3)-a4 2]/2;
v_obs_2(3,1:4)=[-d_obs_2(1)-a4 d_obs_2(2)+a4 -d_obs_2(3)-a4 2]/2;
v_obs_2(4,1:4)=[-d_obs_2(1)-a4 -d_obs_2(2)-a4 -d_obs_2(3)-a4 2]/2;
v_obs_2(5,1:4)=[d_obs_2(1)+a4 d_obs_2(2)+a4 d_obs_2(3)+a4 2]/2;
v_obs_2(6,1:4)=[d_obs_2(1)+a4 -d_obs_2(2)-a4 d_obs_2(3)+a4 2]/2;
v_obs_2(7,1:4)=[-d_obs_2(1)-a4 d_obs_2(2)+a4 d_obs_2(3)+a4 2]/2;
v_obs_2(8,1:4)=[-d_obs_2(1)-a4 -d_obs_2(2)-a4 d_obs_2(3)+a4 2]/2;

Trans_obs_2=[1 0 0 t_obs_2(1);0 1 0 t_obs_2(2);0 0 1 t_obs_2(3);0 0 0 1];
v_obs_2_t=zeros(4,8);
for i=1:8
    v_obs_2_t(1:4,i)=Trans_obs_2*(v_obs_2(i,1:4))';
end

v_obs_2=zeros(8,3);
for i=1:8
   v_obs_2(i,1:3)=v_obs_2_t(1:3,i);
end

Obstaculo2_a4=Polyhedron(v_obs_2);

%% Obstáculo 3
v_obs_3=zeros(8,4);
v_obs_3(1,1:4)=[d_obs_3(1)+a4 d_obs_3(2)+a4 -d_obs_3(3)-a4 2]/2;
v_obs_3(2,1:4)=[d_obs_3(1)+a4 -d_obs_3(2)-a4 -d_obs_3(3)-a4 2]/2;
v_obs_3(3,1:4)=[-d_obs_3(1)-a4 d_obs_3(2)+a4 -d_obs_3(3)-a4 2]/2;
v_obs_3(4,1:4)=[-d_obs_3(1)-a4 -d_obs_3(2)-a4 -d_obs_3(3)-a4 2]/2;
v_obs_3(5,1:4)=[d_obs_3(1)+a4 d_obs_3(2)+a4 d_obs_3(3)+a4 2]/2;
v_obs_3(6,1:4)=[d_obs_3(1)+a4 -d_obs_3(2)-a4 d_obs_3(3)+a4 2]/2;
v_obs_3(7,1:4)=[-d_obs_3(1)-a4 d_obs_3(2)+a4 d_obs_3(3)+a4 2]/2;
v_obs_3(8,1:4)=[-d_obs_3(1)-a4 -d_obs_3(2)-a4 d_obs_3(3)+a4 2]/2;

Trans_obs_3=[1 0 0 t_obs_3(1);0 1 0 t_obs_3(2);0 0 1 t_obs_3(3);0 0 0 1];
v_obs_3_t=zeros(4,8);
```

```
for i=1:8
    v_obs_3_t(1:4,i)=Trans_obs_3*(v_obs_3(i,1:4))';
end

v_obs_3=zeros(8,3);
for i=1:8
   v_obs_3(i,1:3)=v_obs_3_t(1:3,i);
end

Obstaculo3_a4=Polyhedron(v_obs_3);

%% Obstáculo 4
v_obs_4=zeros(8,4);
v_obs_4(1,1:4)=[d_obs_4(1)+a4 d_obs_4(2)+a4 -d_obs_4(3)-a4 2]/2;
v_obs_4(2,1:4)=[d_obs_4(1)+a4 -d_obs_4(2)-a4 -d_obs_4(3)-a4 2]/2;
v_obs_4(3,1:4)=[-d_obs_4(1)-a4 d_obs_4(2)+a4 -d_obs_4(3)-a4 2]/2;
v_obs_4(4,1:4)=[-d_obs_4(1)-a4 -d_obs_4(2)-a4 -d_obs_4(3)-a4 2]/2;
v_obs_4(5,1:4)=[d_obs_4(1)+a4 d_obs_4(2)+a4 d_obs_4(3)+a4 2]/2;
v_obs_4(6,1:4)=[d_obs_4(1)+a4 -d_obs_4(2)-a4 d_obs_4(3)+a4 2]/2;
v_obs_4(7,1:4)=[-d_obs_4(1)-a4 d_obs_4(2)+a4 d_obs_4(3)+a4 2]/2;
v_obs_4(8,1:4)=[-d_obs_4(1)-a4 -d_obs_4(2)-a4 d_obs_4(3)+a4 2]/2;

Trans_obs_4=[1 0 0 t_obs_4(1);0 1 0 t_obs_4(2);0 0 1 t_obs_4(3);0 0 0 1];
v_obs_4_t=zeros(4,8);
for i=1:8
    v_obs_4_t(1:4,i)=Trans_obs_4*(v_obs_4(i,1:4))';
end

v_obs_4=zeros(8,3);
for i=1:8
   v_obs_4(i,1:3)=v_obs_4_t(1:3,i);
end

Obstaculo4_a4=Polyhedron(v_obs_4);



%% AMPLIACIÓN PARA EL LINK 2-3 (solo parte trasera del link 3)

a2=170; %margen para ampliar
a2=a2*2;

%% Obstáculo 1
v_obs_1=zeros(8,4);
v_obs_1(1,1:4)=[d_obs_1(1)+a2 d_obs_1(2)+a2 -d_obs_1(3)-a2 2]/2;
v_obs_1(2,1:4)=[d_obs_1(1)+a2 -d_obs_1(2)-a2 -d_obs_1(3)-a2 2]/2;
v_obs_1(3,1:4)=[-d_obs_1(1)-a2 d_obs_1(2)+a2 -d_obs_1(3)-a2 2]/2;
v_obs_1(4,1:4)=[-d_obs_1(1)-a2 -d_obs_1(2)-a2 -d_obs_1(3)-a2 2]/2;
v_obs_1(5,1:4)=[d_obs_1(1)+a2 d_obs_1(2)+a2 d_obs_1(3)+a2 2]/2;
v_obs_1(6,1:4)=[d_obs_1(1)+a2 -d_obs_1(2)-a2 d_obs_1(3)+a2 2]/2;
v_obs_1(7,1:4)=[-d_obs_1(1)-a2 d_obs_1(2)+a2 d_obs_1(3)+a2 2]/2;
v_obs_1(8,1:4)=[-d_obs_1(1)-a2 -d_obs_1(2)-a2 d_obs_1(3)+a2 2]/2;

Trans_obs_1=[1 0 0 t_obs_1(1);0 1 0 t_obs_1(2);0 0 1 t_obs_1(3);0 0 0 1];
v_obs_1_t=zeros(4,8);
for i=1:8
    v_obs_1_t(1:4,i)=Trans_obs_1*(v_obs_1(i,1:4))';
end
```

```
v_obs_1=zeros(8,3);
for i=1:8
   v_obs_1(i,1:3)=v_obs_1_t(1:3,i);
end

Obstaculo1_a2=Polyhedron(v_obs_1);
%% Obstáculo 2
v_obs_2=zeros(8,4);
v_obs_2(1,1:4)=[d_obs_2(1)+a2 d_obs_2(2)+a2 -d_obs_2(3)-a2 2]/2;
v_obs_2(2,1:4)=[d_obs_2(1)+a2 -d_obs_2(2)-a2 -d_obs_2(3)-a2 2]/2;
v_obs_2(3,1:4)=[-d_obs_2(1)-a2 d_obs_2(2)+a2 -d_obs_2(3)-a2 2]/2;
v_obs_2(4,1:4)=[-d_obs_2(1)-a2 -d_obs_2(2)-a2 -d_obs_2(3)-a2 2]/2;
v_obs_2(5,1:4)=[d_obs_2(1)+a2 d_obs_2(2)+a2 d_obs_2(3)+a2 2]/2;
v_obs_2(6,1:4)=[d_obs_2(1)+a2 -d_obs_2(2)-a2 d_obs_2(3)+a2 2]/2;
v_obs_2(7,1:4)=[-d_obs_2(1)-a2 d_obs_2(2)+a2 d_obs_2(3)+a2 2]/2;
v_obs_2(8,1:4)=[-d_obs_2(1)-a2 -d_obs_2(2)-a2 d_obs_2(3)+a2 2]/2;

Trans_obs_2=[1 0 0 t_obs_2(1);0 1 0 t_obs_2(2);0 0 1 t_obs_2(3);0 0 0 1];
v_obs_2_t=zeros(4,8);
for i=1:8
    v_obs_2_t(1:4,i)=Trans_obs_2*(v_obs_2(i,1:4))';
end

v_obs_2=zeros(8,3);
for i=1:8
   v_obs_2(i,1:3)=v_obs_2_t(1:3,i);
end

Obstaculo2_a2=Polyhedron(v_obs_2);

%% Obstáculo 3
v_obs_3=zeros(8,4);
v_obs_3(1,1:4)=[d_obs_3(1)+a2 d_obs_3(2)+a2 -d_obs_3(3)-a2 2]/2;
v_obs_3(2,1:4)=[d_obs_3(1)+a2 -d_obs_3(2)-a2 -d_obs_3(3)-a2 2]/2;
v_obs_3(3,1:4)=[-d_obs_3(1)-a2 d_obs_3(2)+a2 -d_obs_3(3)-a2 2]/2;
v_obs_3(4,1:4)=[-d_obs_3(1)-a2 -d_obs_3(2)-a2 -d_obs_3(3)-a2 2]/2;
v_obs_3(5,1:4)=[d_obs_3(1)+a2 d_obs_3(2)+a2 d_obs_3(3)+a2 2]/2;
v_obs_3(6,1:4)=[d_obs_3(1)+a2 -d_obs_3(2)-a2 d_obs_3(3)+a2 2]/2;
v_obs_3(7,1:4)=[-d_obs_3(1)-a2 d_obs_3(2)+a2 d_obs_3(3)+a2 2]/2;
v_obs_3(8,1:4)=[-d_obs_3(1)-a2 -d_obs_3(2)-a2 d_obs_3(3)+a2 2]/2;

Trans_obs_3=[1 0 0 t_obs_3(1);0 1 0 t_obs_3(2);0 0 1 t_obs_3(3);0 0 0 1];
v_obs_3_t=zeros(4,8);
for i=1:8
    v_obs_3_t(1:4,i)=Trans_obs_3*(v_obs_3(i,1:4))';
end

v_obs_3=zeros(8,3);
for i=1:8
   v_obs_3(i,1:3)=v_obs_3_t(1:3,i);
end

Obstaculo3_a2=Polyhedron(v_obs_3);

%% Obstáculo 4
v_obs_4=zeros(8,4);
v_obs_4(1,1:4)=[d_obs_4(1)+a2 d_obs_4(2)+a2 -d_obs_4(3)-a2 2]/2;
```

```
v_obs_4(2,1:4)=[d_obs_4(1)+a2 -d_obs_4(2)-a2 -d_obs_4(3)-a2 2]/2;
v_obs_4(3,1:4)=[-d_obs_4(1)-a2 d_obs_4(2)+a2 -d_obs_4(3)-a2 2]/2;
v_obs_4(4,1:4)=[-d_obs_4(1)-a2 -d_obs_4(2)-a2 -d_obs_4(3)-a2 2]/2;
v_obs_4(5,1:4)=[d_obs_4(1)+a2 d_obs_4(2)+a2 d_obs_4(3)+a2 2]/2;
v_obs_4(6,1:4)=[d_obs_4(1)+a2 -d_obs_4(2)-a2 d_obs_4(3)+a2 2]/2;
v_obs_4(7,1:4)=[-d_obs_4(1)-a2 d_obs_4(2)+a2 d_obs_4(3)+a2 2]/2;
v_obs_4(8,1:4)=[-d_obs_4(1)-a2 -d_obs_4(2)-a2 d_obs_4(3)+a2 2]/2;

Trans_obs_4=[1 0 0 t_obs_4(1);0 1 0 t_obs_4(2);0 0 1 t_obs_4(3);0 0 0 1];
v_obs_4_t=zeros(4,8);
for i=1:8
    v_obs_4_t(1:4,i)=Trans_obs_4*(v_obs_4(i,1:4))';
end

v_obs_4=zeros(8,3);
for i=1:8
   v_obs_4(i,1:3)=v_obs_4_t(1:3,i);
end

Obstaculo4_a2=Polyhedron(v_obs_4);



%% Ampliación para 4_int (tres puntos intermedios del link 4)

a4_int=85; % margen para ampliar
a4_int=a4_int*2;

%% Obstáculo 2
v_obs_2=zeros(8,4);
v_obs_2(1,1:4)=[d_obs_2(1)+a4_int d_obs_2(2)+a4_int -d_obs_2(3)-a4_int 2]/2;
v_obs_2(2,1:4)=[d_obs_2(1)+a4_int -d_obs_2(2)-a4_int -d_obs_2(3)-a4_int 2]/2;
v_obs_2(3,1:4)=[-d_obs_2(1)-a4_int d_obs_2(2)+a4_int -d_obs_2(3)-a4_int 2]/2;
v_obs_2(4,1:4)=[-d_obs_2(1)-a4_int -d_obs_2(2)-a4_int -d_obs_2(3)-a4_int 2]/2;
v_obs_2(5,1:4)=[d_obs_2(1)+a4_int d_obs_2(2)+a4_int d_obs_2(3)+a4_int 2]/2;
v_obs_2(6,1:4)=[d_obs_2(1)+a4_int -d_obs_2(2)-a4_int d_obs_2(3)+a4_int 2]/2;
v_obs_2(7,1:4)=[-d_obs_2(1)-a4_int d_obs_2(2)+a4_int d_obs_2(3)+a4_int 2]/2;
v_obs_2(8,1:4)=[-d_obs_2(1)-a4_int -d_obs_2(2)-a4_int d_obs_2(3)+a4_int 2]/2;

Trans_obs_2=[1 0 0 t_obs_2(1);0 1 0 t_obs_2(2);0 0 1 t_obs_2(3);0 0 0 1];
v_obs_2_t=zeros(4,8);
for i=1:8
    v_obs_2_t(1:4,i)=Trans_obs_2*(v_obs_2(i,1:4))';
end

v_obs_2=zeros(8,3);
for i=1:8
   v_obs_2(i,1:3)=v_obs_2_t(1:3,i);
end

Obstaculo2_a4_int=Polyhedron(v_obs_2);

%% Obstáculo 3
v_obs_3=zeros(8,4);
v_obs_3(1,1:4)=[d_obs_3(1)+a4_int d_obs_3(2)+a4_int -d_obs_3(3)-a4_int 2]/2;
v_obs_3(2,1:4)=[d_obs_3(1)+a4_int -d_obs_3(2)-a4_int -d_obs_3(3)-a4_int 2]/2;
v_obs_3(3,1:4)=[-d_obs_3(1)-a4_int d_obs_3(2)+a4_int -d_obs_3(3)-a4_int 2]/2;
v_obs_3(4,1:4)=[-d_obs_3(1)-a4_int -d_obs_3(2)-a4_int -d_obs_3(3)-a4_int 2]/2;
```

```
v_obs_3(5,1:4)=[d_obs_3(1)+a4_int d_obs_3(2)+a4_int d_obs_3(3)+a4_int 2]/2;
v_obs_3(6,1:4)=[d_obs_3(1)+a4_int -d_obs_3(2)-a4_int d_obs_3(3)+a4_int 2]/2;
v_obs_3(7,1:4)=[-d_obs_3(1)-a4_int d_obs_3(2)+a4_int d_obs_3(3)+a4_int 2]/2;
v_obs_3(8,1:4)=[-d_obs_3(1)-a4_int -d_obs_3(2)-a4_int d_obs_3(3)+a4_int 2]/2;

Trans_obs_3=[1 0 0 t_obs_3(1);0 1 0 t_obs_3(2);0 0 1 t_obs_3(3);0 0 0 1];
v_obs_3_t=zeros(4,8);
for i=1:8
    v_obs_3_t(1:4,i)=Trans_obs_3*(v_obs_3(i,1:4))';
end

v_obs_3=zeros(8,3);
for i=1:8
   v_obs_3(i,1:3)=v_obs_3_t(1:3,i);
end

Obstaculo3_a4_int=Polyhedron(v_obs_3);

%% Obstáculo 4
v_obs_4=zeros(8,4);
v_obs_4(1,1:4)=[d_obs_4(1)+a4_int d_obs_4(2)+a4_int -d_obs_4(3)-a4_int 2]/2;
v_obs_4(2,1:4)=[d_obs_4(1)+a4_int -d_obs_4(2)-a4_int -d_obs_4(3)-a4_int 2]/2;
v_obs_4(3,1:4)=[-d_obs_4(1)-a4_int d_obs_4(2)+a4_int -d_obs_4(3)-a4_int 2]/2;
v_obs_4(4,1:4)=[-d_obs_4(1)-a4_int -d_obs_4(2)-a4_int -d_obs_4(3)-a4_int 2]/2;
v_obs_4(5,1:4)=[d_obs_4(1)+a4_int d_obs_4(2)+a4_int d_obs_4(3)+a4_int 2]/2;
v_obs_4(6,1:4)=[d_obs_4(1)+a4_int -d_obs_4(2)-a4_int d_obs_4(3)+a4_int 2]/2;
v_obs_4(7,1:4)=[-d_obs_4(1)-a4_int d_obs_4(2)+a4_int d_obs_4(3)+a4_int 2]/2;
v_obs_4(8,1:4)=[-d_obs_4(1)-a4_int -d_obs_4(2)-a4_int d_obs_4(3)+a4_int 2]/2;

Trans_obs_4=[1 0 0 t_obs_4(1);0 1 0 t_obs_4(2);0 0 1 t_obs_4(3);0 0 0 1];
v_obs_4_t=zeros(4,8);
for i=1:8
    v_obs_4_t(1:4,i)=Trans_obs_4*(v_obs_4(i,1:4))';
end

v_obs_4=zeros(8,3);
for i=1:8
   v_obs_4(i,1:3)=v_obs_4_t(1:3,i);
end

Obstaculo4_a4_int=Polyhedron(v_obs_4);

%% Ampliación para int_2 (4 esferas del link 2)

a2_int=115; % margen para ampliar
a2_int=a2_int*2;

%% Obstáculo 2
v_obs_2=zeros(8,4);
v_obs_2(1,1:4)=[d_obs_2(1)+a2_int d_obs_2(2)+a2_int -d_obs_2(3)-a2_int 2]/2;
v_obs_2(2,1:4)=[d_obs_2(1)+a2_int -d_obs_2(2)-a2_int -d_obs_2(3)-a2_int 2]/2;
v_obs_2(3,1:4)=[-d_obs_2(1)-a2_int d_obs_2(2)+a2_int -d_obs_2(3)-a2_int 2]/2;
v_obs_2(4,1:4)=[-d_obs_2(1)-a2_int -d_obs_2(2)-a2_int -d_obs_2(3)-a2_int 2]/2;
v_obs_2(5,1:4)=[d_obs_2(1)+a2_int d_obs_2(2)+a2_int d_obs_2(3)+a2_int 2]/2;
v_obs_2(6,1:4)=[d_obs_2(1)+a2_int -d_obs_2(2)-a2_int d_obs_2(3)+a2_int 2]/2;
v_obs_2(7,1:4)=[-d_obs_2(1)-a2_int d_obs_2(2)+a2_int d_obs_2(3)+a2_int 2]/2;
v_obs_2(8,1:4)=[-d_obs_2(1)-a2_int -d_obs_2(2)-a2_int d_obs_2(3)+a2_int 2]/2;
```

```
Trans_obs_2=[1 0 0 t_obs_2(1);0 1 0 t_obs_2(2);0 0 1 t_obs_2(3);0 0 0 1];
v_obs_2_t=zeros(4,8);
for i=1:8
    v_obs_2_t(1:4,i)=Trans_obs_2*(v_obs_2(i,1:4))';
end

v_obs_2=zeros(8,3);
for i=1:8
   v_obs_2(i,1:3)=v_obs_2_t(1:3,i);
end

Obstaculo2_a2_int=Polyhedron(v_obs_2);

%% Obstáculo 3
v_obs_3=zeros(8,4);
v_obs_3(1,1:4)=[d_obs_3(1)+a2_int d_obs_3(2)+a2_int -d_obs_3(3)-a2_int 2]/2;
v_obs_3(2,1:4)=[d_obs_3(1)+a2_int -d_obs_3(2)-a2_int -d_obs_3(3)-a2_int 2]/2;
v_obs_3(3,1:4)=[-d_obs_3(1)-a2_int d_obs_3(2)+a2_int -d_obs_3(3)-a2_int 2]/2;
v_obs_3(4,1:4)=[-d_obs_3(1)-a2_int -d_obs_3(2)-a2_int -d_obs_3(3)-a2_int 2]/2;
v_obs_3(5,1:4)=[d_obs_3(1)+a2_int d_obs_3(2)+a2_int d_obs_3(3)+a2_int 2]/2;
v_obs_3(6,1:4)=[d_obs_3(1)+a2_int -d_obs_3(2)-a2_int d_obs_3(3)+a2_int 2]/2;
v_obs_3(7,1:4)=[-d_obs_3(1)-a2_int d_obs_3(2)+a2_int d_obs_3(3)+a2_int 2]/2;
v_obs_3(8,1:4)=[-d_obs_3(1)-a2_int -d_obs_3(2)-a2_int d_obs_3(3)+a2_int 2]/2;

Trans_obs_3=[1 0 0 t_obs_3(1);0 1 0 t_obs_3(2);0 0 1 t_obs_3(3);0 0 0 1];
v_obs_3_t=zeros(4,8);
for i=1:8
    v_obs_3_t(1:4,i)=Trans_obs_3*(v_obs_3(i,1:4))';
end

v_obs_3=zeros(8,3);
for i=1:8
   v_obs_3(i,1:3)=v_obs_3_t(1:3,i);
end

Obstaculo3_a2_int=Polyhedron(v_obs_3);

%% Obstáculo 4
v_obs_4=zeros(8,4);
v_obs_4(1,1:4)=[d_obs_4(1)+a2_int d_obs_4(2)+a2_int -d_obs_4(3)-a2_int 2]/2;
v_obs_4(2,1:4)=[d_obs_4(1)+a2_int -d_obs_4(2)-a2_int -d_obs_4(3)-a2_int 2]/2;
v_obs_4(3,1:4)=[-d_obs_4(1)-a2_int d_obs_4(2)+a2_int -d_obs_4(3)-a2_int 2]/2;
v_obs_4(4,1:4)=[-d_obs_4(1)-a2_int -d_obs_4(2)-a2_int -d_obs_4(3)-a2_int 2]/2;
v_obs_4(5,1:4)=[d_obs_4(1)+a2_int d_obs_4(2)+a2_int d_obs_4(3)+a2_int 2]/2;
v_obs_4(6,1:4)=[d_obs_4(1)+a2_int -d_obs_4(2)-a2_int d_obs_4(3)+a2_int 2]/2;
v_obs_4(7,1:4)=[-d_obs_4(1)-a2_int d_obs_4(2)+a2_int d_obs_4(3)+a2_int 2]/2;
v_obs_4(8,1:4)=[-d_obs_4(1)-a2_int -d_obs_4(2)-a2_int d_obs_4(3)+a2_int 2]/2;

Trans_obs_4=[1 0 0 t_obs_4(1);0 1 0 t_obs_4(2);0 0 1 t_obs_4(3);0 0 0 1];
v_obs_4_t=zeros(4,8);
for i=1:8
    v_obs_4_t(1:4,i)=Trans_obs_4*(v_obs_4(i,1:4))';
end

v_obs_4=zeros(8,3);
for i=1:8
   v_obs_4(i,1:3)=v_obs_4_t(1:3,i);
end
```

```
Obstaculo4_a2_int=Polyhedron(v_obs_4);

%% Representaciones

%% Sin ampliar
figure
plot(Suelo)
hold on
plot(Env_1_Base)
plot(Env_2_Base)
plot(Obstaculo1)
plot(Obstaculo2)
plot(Obstaculo3)
plot(Obstaculo4)
plot(Choque_link1)
hold off
title('Obstáculos')
xlim([-1000 1000])
ylim([-1000 1000])
zlim([0 2000])
xlabel('x')
ylabel('y')
zlabel('z')

%% Ampliado para el extremo
figure
plot(Suelo_a6)
hold on
plot(Env_1_Base_a6)
plot(Env_2_Base_a6)
plot(Obstaculo1_a6)
plot(Obstaculo2_a6)
plot(Obstaculo3_a6)
plot(Obstaculo4_a6)
plot(Choque_link1_a6)
hold off
title('Obstáculos ampliados para el eslabón 6')
a6=a6/2;
xlim([-1000-a6 1000+a6])
ylim([-1000-a6 1000+a6])
zlim([0 2000+a6])
xlabel('x')
ylabel('y')
zlabel('z')

%% Ampliado para el punto de muñeca
figure
plot(Suelo_a4)
hold on
plot(Env_1_Base_a4)
plot(Env_2_Base_a4)
plot(Obstaculo1_a4)
plot(Obstaculo2_a4)
plot(Obstaculo3_a4)
plot(Obstaculo4_a4)
plot(Choque_link1_a4)
hold off
```

```
title('Obstáculos ampliados para el eslabón 4')
a4=a4/2;
xlim([-1000-a4 1000+a4])
ylim([-1000-a4 1000+a4])
zlim([0 2000+a4])
xlabel('x')
ylabel('y')
zlabel('z')

%% Ampliado para para la articualción 3
figure
plot(Obstaculo1_a2)
hold on
plot(Obstaculo2_a2)
plot(Obstaculo3_a2)
plot(Obstaculo4_a2)
hold off
title('Obstáculos ampliados para la articualción 3')
a2=a2/2;
xlim([-1000-a2 1000+a2])
ylim([-1000-a2 1000+a2])
zlim([0 2000+a2])
xlabel('x')
ylabel('y')
zlabel('z')

%% Ampliado para el eslabón 4
figure
plot(Obstaculo2_a4_int)
hold on
plot(Obstaculo3_a4_int)
plot(Obstaculo4_a4_int)
hold off
title('Obstáculos ampliados para el eslabón 3')
a4_int=a4_int/2;
xlim([-1000-a4_int 1000+a4_int])
ylim([-1000-a4_int 1000+a4_int])
zlim([0 2000+a4_int])
xlabel('x')
ylabel('y')
zlabel('z')

%% Ampliado para el eslabón 2
figure
plot(Obstaculo2_a2_int)
hold on
plot(Obstaculo3_a2_int)
plot(Obstaculo4_a2_int)
hold off
title('Obstáculos ampliados para el eslabón 2')
a2_int=a2_int/2;
xlim([-1000-a2_int 1000+a2_int])
ylim([-1000-a2_int 1000+a2_int])
zlim([0 2000+a2_int])
xlabel('x')
ylabel('y')
zlabel('z')
```

10.1.11. Función Ordenar_ptos_6D

```
function  [puntos,distancias,num]  =  Ordenar_ptos_6D(Punto,lista_puntos)
% Ordena los puntos de una lista según la proximidad  a Punto. Devuelve el
% vector con los putnos ordenados,otro con sus distancias y otro con sus
% índices

N=size(lista_puntos,1);

% Obtenención de las distancias:
for i=1:N
d(i)=sqrt(5*(Punto(1)-lista_puntos(i,1))^2+4*(Punto(2)-lista_puntos(i,2))^2+3.5*
(Punto(3)-lista_puntos(i,3))^2+0.5*(Punto(4)-lista_puntos(i,4))^2+0.25*(Punto(5)-
lista_puntos(i,5))^2+0*(Punto(6)-lista_puntos(i,6))^2);
end

[distancias,num]=sort(d); % ordena las distancias y devuelve la distancia y el
índice (num)

for i=1:N
    puntos(i,:)=lista_puntos(num(i),:);
end
end
```

10.1.12. Función Trayectoria_xyz

```
function [x6,y6,z6,x45,y45,z45,x3,y3,z3,x41,y41,z41,x42,y42,z42,x43,y43,z43,x21,
y21,z21,x22,y22,z22,x23,y23,z23,x24,y24,z24] = Trayectoria_xyz(q1,q2,q3,q4,q5,q6)
% Deuvuelve la trayectoria en coordenadas cartesianas de ciertos puntos del
% robot a partir de un vector con la trayectoria de las articulaciones
%% Parámetros Denavit-Hartenberg
theta_1=q1;
theta_2=q2-pi/2;
theta_3=q3;
theta_4=q4;
theta_5=q5;
theta_6=q6;

d_1=352;
d_2=0;
d_3=0;
d_4=380;
d_5=0;
d_6=65;

a_1=70;
a_2=360;
a_3=0;
a_4=0;
a_5=0;
a_6=0;

alpha_1=-pi/2;
alpha_2=0;
alpha_3=-pi/2;
alpha_4=pi/2;
alpha_5=-pi/2;
alpha_6=0;


d_41=155;
d_42=235;
d_43=315;

d_21=-115;
d_22=-115;
d_23=-115;
d_24=-15;
a_21=-360;
a_22=-260;
a_23=-160;
a_24=-160;

%% Vectores vacíos para almacenar las salidas (inicialmente vacíos)
x6=[];
y6=[];
z6=[];

x45=[];
y45=[];
z45=[];

x3=[];
y3=[];
```

```
z3=[];

x41=[];
y41=[];
z41=[];

x42=[];
y42=[];
z42=[];

x43=[];
y43=[];
z43=[];

x21=[];
y21=[];
z21=[];

x22=[];
y22=[];
z22=[];

x23=[];
y23=[];
z23=[];

x24=[];
y24=[];
z24=[];
%% Matrices de transformación
N=size(q1,1);

for i=1:1:N

T_01=MatrizTransformacion(theta_1(i),d_1,a_1,alpha_1);
T_12=MatrizTransformacion(theta_2(i),d_2,a_2,alpha_2);
T_23=MatrizTransformacion(theta_3(i),d_3,a_3,alpha_3);
T_34=MatrizTransformacion(theta_4(i),d_4,a_4,alpha_4);
T_45=MatrizTransformacion(theta_5(i),d_5,a_5,alpha_5);
T_56=MatrizTransformacion(theta_6(i),d_6,a_6,alpha_6);

T_02=T_01*T_12;
T_03=T_02*T_23;
T_04=T_03*T_34;
T_05=T_04*T_45;
T_06=T_05*T_56;


% Valores x, y, z del extremo
x6=[x6,T_06(1,4)];
y6=[y6,T_06(2,4)];
z6=[z6,T_06(3,4)];

% Valores x, y, z del punto de muñeca
x45=[x45,T_05(1,4)];
y45=[y45,T_05(2,4)];
z45=[z45,T_05(3,4)];
```

```
% Valores x, y, z del origen del origen de los sistemas 2 y 3
x3=[x3,T_02(1,4)];
y3=[y3,T_02(2,4)];
z3=[z3,T_02(3,4)];

% Valores x,y,z a lo largo del eslabón 4
T41=T_03*MatrizTransformacion(theta_4(i),d_41,a_4,alpha_4);
T42=T_03*MatrizTransformacion(theta_4(i),d_42,a_4,alpha_4);
T43=T_03*MatrizTransformacion(theta_4(i),d_43,a_4,alpha_4);

x41=[x41,T41(1,4)];
y41=[y41,T41(2,4)];
z41=[z41,T41(3,4)];

x42=[x42,T42(1,4)];
y42=[y42,T42(2,4)];
z42=[z42,T42(3,4)];

x43=[x43,T43(1,4)];
y43=[y43,T43(2,4)];
z43=[z43,T43(3,4)];

% Valores x,y,z a lo largo del eslabón 2
T21=T_01*MatrizTransformacion(theta_2(i),d_21,a_21,alpha_2);
T22=T_01*MatrizTransformacion(theta_2(i),d_22,a_22,alpha_2);
T23=T_01*MatrizTransformacion(theta_2(i),d_23,a_23,alpha_2);
T24=T_01*MatrizTransformacion(theta_2(i),d_24,a_24,alpha_2);

x21=[x21,T21(1,4)];
y21=[y21,T21(2,4)];
z21=[z21,T21(3,4)];

x22=[x22,T22(1,4)];
y22=[y22,T22(2,4)];
z22=[z22,T22(3,4)];

x23=[x23,T23(1,4)];
y23=[y23,T23(2,4)];
z23=[z23,T23(3,4)];

x24=[x24,T24(1,4)];
y24=[y24,T24(2,4)];
z24=[z24,T24(3,4)];
end

end
```

10.1.13. Función Delimita_Cfree

```
function [Ptos_free,Ptos_Obst] = Delimita_Cfree(Ptos,Suelo_a6,Env_1_Base_a6,
Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,
Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,
Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,
Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)
% Clasifica los puntos en el espacio de configtuaciones en puntos libres de
% colisión (Ptos_free) y puntos en colisión (Ptos_Obst). Las entradas son
% los puntos que se van a clasificar y los obstáculos
N=size(Ptos,1);   % Numero de puntos.
Ptos_free=[];
Ptos_Obst=[];

for i=1:N
if Colision_Configuracion_Obstaculos(Ptos(i,:),Suelo_a6,Env_1_Base_a6,
Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,
Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,
Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,
Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)==0
    Ptos_free=[Ptos_free;Ptos(i,:)]; %  Puntos  libres  de  colisión
else
    Ptos_Obst=[Ptos_Obst;Ptos(i,:)]; %  Puntos  en  colisión
end
end

end
```

10.1.14. Función Dijkstra

```
function  [Camino,  Coste]=Dijkstra(matriz_costes,nodo_inicial,nodo_final)
% Devuelve el camino óptimo y su coste a partir de la matriz de costes y los
% nodos inicial y final mediante el algoritmo Dijkstra

% Tamaño de la matriz
N=size(matriz_costes,1);
% Se añade el primer nodo a la lista cerrados
cerrados= nodo_inicial;
% La listas abiertos, costes y caminos están vacías
abiertos=[];
costes=[];
caminos=[];

% Inicializar abiertos, caminos, coste:
j=1;
for i=1:N
      if i~=nodo_inicial % si el nodo no es el inical
            abiertos=[abiertos;i]; % en la lista están todos los nodos execpto el ↙
inicial
              caminos=[caminos;nodo_inicial,i]; % los caminos posibles son todos ↙
entre el nodo inicial y el resto
            costes=[costes;matriz_costes(nodo_inicial,i)]; % los costes de cada ↙
camino se obtienen de la matriz
            if i==nodo_final % si es el nodo final
               i_final=j; % i_final es j
            end
            j=j+1; % si no es el nodo final se incrementa j
      else % si el nodo es el inical
            caminos=[caminos;nodo_inicial,-1]; % se añade -1
            costes(i)=0; % el coste es 0
      end
end

   % Bucle:

   indice=1; % i se inicializa el índice a 1
   menor=nodo_inicial; % se inicializa menro con el valor del índice del ↙
nodo_inicial

   % Bucle: mientras no se haya recorrido toda la matriz y menor sea
   % distinto del nodo final
   while indice<=N && menor~=nodo_final

   % Se obtiene el nodo de menor coste de abiertos

   M=length(abiertos);
   coste_min=Inf;

   for i=1:M
      if costes(abiertos(i))<coste_min
          menor=abiertos(i);
         coste_min=costes(abiertos(i));
      end
   end

  % Se extrae menor de abiertos y se añade a cerrados

   temp_abiertos=[];
```

```
        for i=1:M
               if abiertos(i)~=menor
                 temp_abiertos=[temp_abiertos,abiertos(i)];
               end
        end
        abiertos=temp_abiertos;
        cerrados=[cerrados;menor];


        % Se comparan los costes y se actualizan

        caminos=[caminos  -ones(N,1)]; % Se añade un -1 en todas las filas de caminos

        % Recorre abiertos
        for i=1:M-1
           n=abiertos(i);
           % si el coste del nodo es mayor que el mínimo más el de menor
           if costes(n)>coste_min+matriz_costes(menor,n)
               % se actualiza su coste
              costes(n)=coste_min+matriz_costes(menor,n);
              k=1;
              aux=[];
              % mientras el coste entre menor y el nodo k sea distinto de -1
              while caminos(menor,k)~=-1
                 aux(k)=caminos(menor,k); % actualiza el coste
                 k=k+1;
              end
              aux=[aux n];

              caminos(n,1:k)=aux; % actualiza los caminos
              % añade un -1 desde el valor siguiente a k hasta el final en
              % caminos
              b=size(caminos,2);
              % escribe -1 en los nuevos caminos
              caminos(n,k+1:b)=-ones(1,size(k+1-b,2));
           end
        end
        indice=indice+1; % se incrementa el índice

    end


     % Se eliminan los -1's:

    Camino_aux=caminos(nodo_final,:);

    i=1;
    while Camino_aux(i)~=-1
        Camino(i)=Camino_aux(i);
        i=i+1;
    end

    % devuelve el coste del nodo final
   Coste=costes(nodo_final);
end
```

10.1.15. Función generar_puntos_aleatorios_q

```
function [Puntos] = generar_puntos_aleatorios_q(N_puntos,q_min,q_max)
% Genera N puntos entre qmin y qmax de las dimensiones de los vectoes qmin
% y qmax introducidos
N=length(q_min);

for i=1:N_puntos
    for j=1:N

     q(j)=(q_max(j)-q_min(j))*rand+q_min(j);
    end

     Puntos(i,:)=q;

end

end
```

10.1.16. Función generar_rectas_matriz_costes

```
function [rectas,matriz_costes]=generar_rectas_matriz_costes(Puntos,Suelo_a6,
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)
% Genera la matriz de costes y las rectas entre los puntos a partir de los
% puntos y los obstáculos ampliados
N=size(Puntos,1);
rectas=[];
matriz_costes=Inf*ones(N); % Se inicililiza la matriz de costes a valores
infinitos.

for i=1:N-1 % Para todos los puntos

    for j=i+1:N % Para el resto de puntos aún sin comprobar

        recta=[Puntos(i,:),Puntos(j,:)]; % Rectas formadas por los dos vértices.

        % si no hay colisión entre ambos putnos
        if Colision_Curva_MoveAbsJ_Obstaculos(Puntos(i,:),Puntos(j,:),0.5,Suelo_a6,
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)==0
                rectas=[rectas;recta]; % se añade la recta a rectas
                % Se calcula el coste y se añade a la matriz
                matriz_costes(i,j)=sqrt(5*(Puntos(i,1)-Puntos(j,1))^2+4*(Puntos(i,
2)-Puntos(j,2))^2+3.5*(Puntos(i,3)-Puntos(j,3))^2+0.5*(Puntos(i,4)-Puntos(j,4)
^2+0.25*(Puntos(i,5)-Puntos(j,5))^2);
                % La matriz es simétrica respecto a su diaganal, por tanto:
                matriz_costes(j,i)= matriz_costes(i,j);
        end
    end


end

end
```

10.1.17. *Script* para la ejecución del algoritmo PRM

```
tic
% Número de puntos aleatorios
N_puntos=150;

% Configuración inicial y final

start=[-160 60 -90 0 0 0];
goal=[0 50 40 -20 -30 50];

% longitud de los ejes de los puntos para la representación
long=100;

% Representación del entorno
figure
plot(Suelo,'color','y')
hold on
plot(Env_1_Base,'color','y')
plot(Env_2_Base,'color','y')
plot(Obstaculo1,'color','y')
plot(Obstaculo2,'color','y')
plot(Obstaculo3,'color','y')
plot(Obstaculo4,'color','y')

title('Trayectoria generada mediante PRM')
xlim([-1000 1000])
ylim([-1000 1000])
zlim([0 2000])
xlabel('x (mm)')
ylabel('y (mm)')
zlabel('z (mm)')

% Cálculo del punto de incio en el espacio cartesiano
Trans=Cinematica_Directa(DegToRad(start(1)),DegToRad(start(2)),DegToRad(start(3)),
DegToRad(start(4)),DegToRad(start(5)),DegToRad(start(6)));
% Representación del punto inicial
DibujarPuntos6D(Trans,long,'*',2);

% Cálculo del punto final en el espacio cartesiano
Trans=Cinematica_Directa(DegToRad(goal(1)),DegToRad(goal(2)),DegToRad(goal(3)),
DegToRad(goal(4)),DegToRad(goal(5)),DegToRad(goal(6)));
% Representación del punto final
DibujarPuntos6D(Trans,long,'s',2);

%% Generar puntos aleatorios
q_min=[-180 -90 -230 -180 -115 -180];
q_max=[180 110 50 180 115 180];
Puntos=generar_puntos_aleatorios_q(N_puntos,q_min,q_max);


%% Descarte de los puntos que están en colisión
[Ptos_free,Ptos_Obs] = Delimita_Cfree(Puntos,Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,
Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,
Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,Obstaculo4_a4,
Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,Choque_link1_a6,
Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,Obstaculo4_a2_int,
Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int);
```

```
%% Representación de los puntos en Cfree
N=size(Ptos_free,1);
for i=1:1:N

    Trans=Cinematica_Directa(DegToRad(Ptos_free(i,1)),DegToRad(Ptos_free(i,2)),
DegToRad(Ptos_free(i,3)),DegToRad(Ptos_free(i,4)),DegToRad(Ptos_free(i,5)),DegToRad
(Ptos_free(i,6)));

    DibujarPuntos6D(Trans,long,'o',0.5);

end

%% Cálculo de las rectas y de la matriz de costes

[rectas,matriz_costes]=generar_rectas_matriz_costes(Ptos_free,Suelo_a6,
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int);

%% Se añaden el nodo inicial y el final
% Se conecta el inicial al más cercano sin colision
[puntosCercaStart,costeStart,num]=Ordenar_ptos_6D(start,Ptos_free);
i=1;
conectado_Start=0;
while i<=N && conectado_Start==0
    if Colision_Curva_MoveAbsJ_Obstaculos(start,puntosCercaStart(i,:),0.5,Suelo_a6,
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)==0
        indiceStart=num(i);
        conectado_Start=1;
        coste1=costeStart(i);
    end
    i=i+1;
end
% Se conecta el final al más cercano sin colision
[puntosCercaGoal,costeGoal,num]=Ordenar_ptos_6D(goal,Ptos_free);
i=1;
conectado_Goal=0;
while i<=N && conectado_Goal==0
    if Colision_Curva_MoveAbsJ_Obstaculos(puntosCercaGoal(i,:),goal,0.5,Suelo_a6,
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)==0
        indiceGoal=num(i);
        conectado_Goal=1;
        coste2=costeGoal(i);
    end
    i=i+1;
end

%% Camino más corto
if conectado_Start==1 && conectado_Goal==1
   [Camino,    Coste]=Dijkstra(matriz_costes,indiceStart,indiceGoal);
   % Coste total
   Coste=Coste+coste1+coste2;
```

```
else %si no se han podido conectar los nodos fianl e inicial
   f = msgbox("Error añadiendo los nodos incial y final"); % Mensje de error
end
%% Si el coste es infinito no ha enctontrado un camino
if Coste==Inf
 f = msgbox("Error. Camino no encontrado"); %Mensje de error
elseif conectado_Start==1 && conectado_Goal==1
%% Dibujar Camino

N=length(Camino);
puntos=[];
for i=1:N
    puntos(i,:)=[Ptos_free(Camino(i),:)];
end

% Se añaden los nodos inicial y final a la trayectoria
puntos=[start;puntos;goal];

N=size(puntos,1);

for i=2:N-1
   Trans=Cinematica_Directa(DegToRad(puntos(i,1)),DegToRad(puntos(i,2)),DegToRad
(puntos(i,3)),DegToRad(puntos(i,4)),DegToRad(puntos(i,5)),DegToRad(puntos(i,6)));

   DibujarPuntos6D(Trans,long,'o',2);
end

for  i=1:N-1
   dibujar_trayectoria(puntos(i,:),puntos(i+1,:),'r',2);
end
hold off

end

toc
```

10.1.18. *Script* para la simulación de la trayectoria generada mediante PRM

```
% Simula la trayectoria calculada mediante PRM

T=5; % 5 segundos por desplazamiento entre dos nodos

qv1=[];
qv2=[];
qv3=[];
qv4=[];
qv5=[];
qv6=[];

for i=1:1:N-1
    [qv1_i,qv2_i,qv3_i,qv4_i,qv5_i,qv6_i] = moveAbsJ([DegToRad(puntos(i,1))
DegToRad(puntos(i,2)) DegToRad(puntos(i,3)) DegToRad(puntos(i,4)) DegToRad(puntos
(i,5)) DegToRad(puntos(i,6))],[DegToRad(puntos(i+1,1)) DegToRad(puntos(i+1,2))
DegToRad(puntos(i+1,3)) DegToRad(puntos(i+1,4)) DegToRad(puntos(i+1,5)) DegToRad
(puntos(i+1,6))],T,i-1);
    qv1=[qv1;qv1_i];
    qv2=[qv2;qv2_i];
    qv3=[qv3;qv3_i];
    qv4=[qv4;qv4_i];
    qv5=[qv5;qv5_i];
    qv6=[qv6;qv6_i];
end
T_sim=(N-1)*T; % duración de la simulación

smiData=datosBrazo(); % datos para la simulación

sim('simulacion_brazo_obstaculos.slx') % ejecuta el modelo de Simulink en Simscape
```

10.1.19. *Script* para crear el árbol de configuraciones mediante RRT

```
i=1;

final=0;
tic

% hasta que se llegue al final o a las iteraciones máximas
while i< max_iteraciones && final==0

    % Crear un nuevo nodo;

   nuevo_nodo=crear_nodo_conf_RRT(arbol,long,lista_puntos,min_distancia,T,q1_min,
q1_max,q2_min,q2_max,q3_min,q3_max,q4_min,q4_max,q5_min,q5_max,q6_min,q6_max,[goal.
q1,goal.q2,goal.q3,goal.q4,goal.q5,goal.q6],sesgo,i,Suelo_a6,Env_1_Base_a6,
Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,
Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,
Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,
Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)


% Añadirlo al árbol:

arbol=[arbol nuevo_nodo];

% Poner al día la lista de puntos:
lista_puntos=[lista_puntos;[nuevo_nodo.q1 nuevo_nodo.q2 nuevo_nodo.q3 nuevo_nodo.q4
nuevo_nodo.q5 nuevo_nodo.q6]];

% Verificar si el nodo nuevo está suficientemente cerca del nodo objetivo.

if sqrt(5*(goal.q1-nuevo_nodo.q1)^2+4*(goal.q2-nuevo_nodo.q2)^2+3.5*(goal.q3-
nuevo_nodo.q3)^2+0.5*(goal.q4-nuevo_nodo.q4)^2+0.25*(goal.q5-nuevo_nodo.q5)^2)
<min_distancia
   % verificar si se puede unir sin colisionar
    if Colision_Curva_MoveAbsJ_Obstaculos([nuevo_nodo.q1,nuevo_nodo.q2,nuevo_nodo.
q3,nuevo_nodo.q4,nuevo_nodo.q5,nuevo_nodo.q6],[goal.q1,goal.q2,goal.q3,goal.q4
goal.q5,goal.q6],T,Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6
Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4
Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2
Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4
Obstaculo2_a2_int,Obstaculo3_a2_int,Obstaculo4_a2_int,Obstaculo2_a4_int
Obstaculo3_a4_int,Obstaculo4_a4_int)==0 % Verificar que no hay obstáculos en medio
        arbol=[arbol goal];
        goal.nodo_predecesor=nuevo_nodo;
        goal_numero=i+1;
        goal.coste=nuevo_nodo.coste+sqrt(5*(goal.q1-nuevo_nodo.q1)^2+4*(goal.q2-
nuevo_nodo.q2)^2+3.5*(goal.q3-nuevo_nodo.q3)^2+0.5*(goal.q4-nuevo_nodo.q4)^2+0.25*
(goal.q5-nuevo_nodo.q5)^2);
        final=1;
        dibujar_trayectoria([nuevo_nodo.q1,nuevo_nodo.q2,nuevo_nodo.q3,nuevo_nodo.
q4,nuevo_nodo.q5,nuevo_nodo.q6],[goal.q1,goal.q2,goal.q3,goal.q4,goal.q5,goal.
q6],'k',0.5);
    end
end
i=i+1;
end
toc
```

```
if final==1
% Se calcula el camino
puntos=Trayectoria_RRT_6D(goal,start);

% Representación de los puntos y trayectorias
N=size(puntos,2);

for i=2:1:N-1
    Trans=Cinematica_Directa(DegToRad(puntos(1,i).q1),DegToRad(puntos(1,i).q2),
DegToRad(puntos(1,i).q3),DegToRad(puntos(1,i).q4),DegToRad(puntos(1,i).q5),DegToRad
(puntos(1,i).q6));

    DibujarPuntos6D(Trans,long,'o',2);
end

for i=1:1:N-1
    dibujar_trayectoria([puntos(1,i).q1,puntos(1,i).q2,puntos(1,i).q3,puntos(1,i).
q4,puntos(1,i).q5,puntos(1,i).q6],[puntos(1,i+1).q1,puntos(1,i+1).q2,puntos(1,i+1).
q3,puntos(1,i+1).q4,puntos(1,i+1).q5,puntos(1,i+1).q6],'r',3);
end

%% Representación de la trayectoria (sin el resto del árbol)
figure
plot(Suelo,'color','y')
hold on
plot(Env_1_Base,'color','y')
plot(Env_2_Base,'color','y')
plot(Obstaculo1,'color','y')
plot(Obstaculo2,'color','y')
plot(Obstaculo3,'color','y')
plot(Obstaculo4,'color','y')
title('Trayectoria generada mediante RRT')
xlim([-1000 1000])
ylim([-1000 1000])
zlim([0 2000])
xlabel('x (mm)')
ylabel('y (mm)')
zlabel('z (mm)')

% Punto inicial en coordenadas cartesianas
Trans=Cinematica_Directa(DegToRad(start.q1),DegToRad(start.q2),DegToRad(start.q3),
DegToRad(start.q4),DegToRad(start.q5),DegToRad(start.q6));
% Representación del punto inicial
DibujarPuntos6D(Trans,long,'*',2);
% Punto final en coordenadas cartesianas
Trans=Cinematica_Directa(DegToRad(goal.q1),DegToRad(goal.q2),DegToRad(goal.q3),
DegToRad(goal.q4),DegToRad(goal.q5),DegToRad(goal.q6));
% Representación del punto final
DibujarPuntos6D(Trans,long,'s',2);

for i=2:1:N-1
    Trans=Cinematica_Directa(DegToRad(puntos(1,i).q1),DegToRad(puntos(1,i).q2),
DegToRad(puntos(1,i).q3),DegToRad(puntos(1,i).q4),DegToRad(puntos(1,i).q5),DegToRad
(puntos(1,i).q6));

    DibujarPuntos6D(Trans,long,'o',2);
end
```

```
for i=1:1:N-1
    dibujar_trayectoria([puntos(1,i).q1,puntos(1,i).q2,puntos(1,i).q3,puntos(1,i).
q4,puntos(1,i).q5,puntos(1,i).q6],[puntos(1,i+1).q1,puntos(1,i+1).q2,puntos(1,i+1).
q3,puntos(1,i+1).q4,puntos(1,i+1).q5,puntos(1,i+1).q6],'r',3);
end
end
```

10.1.20. Función crear_configuracion_aleatoria_RRT_6D_OBS

```
function [conf] = crear_configuracion_aleatoria_RRT_6D_OBS(q1_min,q1_max,q2_min,
q2_max,q3_min,q3_max,q4_min,q4_max,q5_min,q5_max,q6_min,q6_max,objetivo,sesgo,
Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)
% Crea un punto aleatorio en los límites dados y con el sesgo dado hacia el
% objetivo.

colision=1;

while colision==1 % mientras la configuración esté en colisión

q1=(q1_max-q1_min)*rand+q1_min+sesgo*objetivo(1);
q2=(q2_max-q2_min)*rand+q2_min+sesgo*objetivo(2);
q3=(q3_max-q3_min)*rand+q3_min+sesgo*objetivo(3);
q4=(q4_max-q4_min)*rand+q4_min+sesgo*objetivo(4);
q5=(q5_max-q5_min)*rand+q5_min+sesgo*objetivo(5);
q6=(q6_max-q6_min)*rand+q6_min+sesgo*objetivo(6);

q=[q1 q2 q3 q4 q5 q6];

% comprueba si está en colisión
colision=Colision_Configuracion_Obstaculos(q,Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,
Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,
Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,Obstaculo4_a4,
Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,Choque_link1_a6,
Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,Obstaculo4_a2_int,
Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int);
end

conf=[q1,q2,q3,q4,q5,q6];
end
```

10.1.21. Función crear_nodo_conf_RRT

```
function [nuevo_nodo] = crear_nodo_conf_RRT(arbol,long,lista_puntos,longitud,T
q1_min,q1_max,q2_min,q2_max,q3_min,q3_max,q4_min,q4_max,q5_min,q5_max,q6_min
q6_max,objetivo,sesgo,numero,Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,
Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,
Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2
Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4
Obstaculo2_a2_int,Obstaculo3_a2_int,Obstaculo4_a2_int,Obstaculo2_a4_int
Obstaculo3_a4_int,Obstaculo4_a4_int)
% Crea un nuevo nodo

Condicion=1;  % Condicion de que se pueda unir con algunos vecino sin intersectar
nigun obstáculo.
corta=1;

while Condicion ==1   % Hasta encontrar un punto que se pueda conectar con algún
vecino

% Nueva configuración aleatoria
configuracion=crear_configuracion_aleatoria_RRT_6D_OBS(q1_min,q1_max,q2_min,q2_max,
q3_min,q3_max,q4_min,q4_max,q5_min,q5_max,q6_min,q6_max,objetivo,sesgo,Suelo_a6
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int);

% Se busca el vecino más próximo:

% Ordenar los puntos según su distancia al nuevo nodo.

[puntos,  distancias,num]  =  Ordenar_ptos_6D(configuracion,lista_puntos);


M=size(puntos,1);

i=1;
while i<=M   && corta==1 % recorre el árbol mientras la recta corte

q_near=puntos(i,:);
numero_predecesor=num(i);
d=distancias(i);
numero_predecesor=num(i)-1;

previo=arbol(num(i));

Vector=[configuracion(1)-q_near(1),configuracion(2)-q_near(2),configuracion(3)-
q_near(3),configuracion(4)-q_near(4),configuracion(5)-q_near(5),configuracion(6)-
q_near(6)];

% Calcular el candidato a q_new:
Vector_unitario=Vector/d;
q_new_candidato=q_near+Vector_unitario*longitud;


corta=0; % se supone en principio que no colisiona

%% VERIFICAR QUE LA RECTA NO COLISIONA
```

```
                if Colision_Curva_MoveAbsJ_Obstaculos(q_new_candidato,[previo.q1,
previo.q2,previo.q3,previo.q4,previo.q5,previo.q6],T,Suelo_a6,Env_1_Base_a6,
Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,
Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,
Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,
Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)==0
                    % % % Añadir el nuevo nodo:
                    q_new=q_new_candidato;
                    nuevo_nodo=nodo_conf_6D;
                    nuevo_nodo.nombre=sprintf('nodo%04d',numero);
                    nuevo_nodo.numero=numero;
                    sprintf('nodo%04d',numero);
                    nuevo_nodo.predecesor=numero_predecesor;
                    nuevo_nodo.nodo_predecesor=previo;
                    nuevo_nodo.q1=q_new(1);
                    nuevo_nodo.q2=q_new(2);
                    nuevo_nodo.q3=q_new(3);
                    nuevo_nodo.q4=q_new(4);
                    nuevo_nodo.q5=q_new(5);
                    nuevo_nodo.q6=q_new(6);
                    nuevo_nodo.coste=previo.coste+sqrt(5*(previo.q1-nuevo_nodo.q1
^2+4*(previo.q2-nuevo_nodo.q2)^2+3.5*(previo.q3-nuevo_nodo.q3)^2+0.5*(previo.q4-
nuevo_nodo.q4)^2+0.25*(previo.q5-nuevo_nodo.q5)^2);
                    Condicion=0;
                    % Prepresentación del nodo y de la trayectoria de unión
                    % con el nodo anterior
                    Trans=Cinematica_Directa(DegToRad(nuevo_nodo.q1),DegToRad
(nuevo_nodo.q2),DegToRad(nuevo_nodo.q3),DegToRad(nuevo_nodo.q4),DegToRad
(nuevo_nodo.q5),DegToRad(nuevo_nodo.q6));

                    DibujarPuntos6D(Trans,long,'o',0.5);

                    dibujar_trayectoria([previo.q1,previo.q2,previo.q3,previo.q4,
previo.q5,previo.q6],[nuevo_nodo.q1,nuevo_nodo.q2,nuevo_nodo.q3,nuevo_nodo.q4,
nuevo_nodo.q5,nuevo_nodo.q6],'k',0.5);
                else
                    corta=1;
                end
                 i=i+1;
end


end

end
```

10.1.22. *Script* para inicializar RRT (también válido para RRT*)

```
% Longitud de los ejes para la represetación de los puntos
long=100;

% Representación del entorno
figure
plot(Suelo,'color','y')
hold on
plot(Env_1_Base,'color','y')
plot(Env_2_Base,'color','y')
plot(Obstaculo1,'color','y')
plot(Obstaculo2,'color','y')
plot(Obstaculo3,'color','y')
plot(Obstaculo4,'color','y')
title('Trayectoria generada mediante RRT')
xlim([-1000 1000])
ylim([-1000 1000])
zlim([0 2000])
xlabel('x (mm)')
ylabel('y (mm)')
zlabel('z (mm)')


% Variables globales
global arbol
global q1_min;
global q1_max;
global q2_min;
global q2_max;
global q3_min;
global q3_max;
global q4_min;
global q4_max;
global q5_min;
global q5_max;
global q6_min;
global  q6_max;

% Máximos y mínimos de las articulaciones
q1_min=-180;
q1_max=180;
q2_min=-90;
q2_max=110;
q3_min=-230;
q3_max=50;
q4_min=-180;
q4_max=180;
q5_min=-115;
q5_max=115;
q6_min=-180;
q6_max=180;

% Crear el nodo de inicio:
start=nodo_conf_6D;
start.nombre='inicio';
start.numero=0;
start.q1=-160;
start.q2=60;
start.q3=-90;
```

```
start.q4=0;
start.q5=0;
start.q6=0;
start.coste=0;

% Crear el nodo objetivo:
goal=nodo_conf_6D;
goal.nombre='objetivo';
goal.q1=0;
goal.q2=50;
goal.q3=40;
goal.q4=-20;
goal.q5=-30;
goal.q6=50;

% Punto inicial en coordenadas cartesianas
Trans=Cinematica_Directa(DegToRad(start.q1),DegToRad(start.q2),DegToRad(start.q3),
DegToRad(start.q4),DegToRad(start.q5),DegToRad(start.q6));
% Representación del punto inicial
DibujarPuntos6D(Trans,long,'*',2);
% Punto final en coordenadas cartesianas
Trans=Cinematica_Directa(DegToRad(goal.q1),DegToRad(goal.q2),DegToRad(goal.q3),
DegToRad(goal.q4),DegToRad(goal.q5),DegToRad(goal.q6));
% Representación del punto final
DibujarPuntos6D(Trans,long,'s',2);

lista_puntos=[start.q1,start.q2,start.q3,start.q4,start.q5,start.q6]; %
Inicializar la lista de nodos del árbol.



max_iteraciones=3000;  % Número máximo de iteraciones despues del cual se para la
búsqueda:
min_distancia=100;  % Distacia del objetivo a partir de la cual para la busqueda;
T=0.5;  % Valor de T para la función Trayectoria_xyz
sesgo=0.05;


arbol=nodo_conf_6D;
arbol=start;
```

10.1.23. Definición de la clase nodo_conf_6D

```
classdef  nodo_conf_6D
    properties
        numero;
        nombre;
        predecesor;
        nodo_predecesor;
        q1;
        q2;
        q3;
        q4;
        q5;
        q6;
        coste;
    end
 end
```

10.1.24. *Script* para la simulación de trayectorias generadas mediante RRT (o RRT*)

```
% Simulación de la trayectoria generada mediante RRT (también válido con
% RRT*)
qv1=[];
qv2=[];
qv3=[];
qv4=[];
qv5=[];
qv6=[];

T=5;

for i=1:1:N-1
    [qv1_i,qv2_i,qv3_i,qv4_i,qv5_i,qv6_i] = moveAbsJ([DegToRad(puntos(1,i).q1)
DegToRad(puntos(1,i).q2) DegToRad(puntos(1,i).q3) DegToRad(puntos(1,i).q4) DegToRad
(puntos(1,i).q5) DegToRad(puntos(1,i).q6)],[DegToRad(puntos(1,i+1).q1) DegToRad
(puntos(1,i+1).q2) DegToRad(puntos(1,i+1).q3) DegToRad(puntos(1,i+1).q4) DegToRad
(puntos(1,i+1).q5) DegToRad(puntos(1,i+1).q6)],T,i-1);
    qv1=[qv1;qv1_i];
    qv2=[qv2;qv2_i];
    qv3=[qv3;qv3_i];
    qv4=[qv4;qv4_i];
    qv5=[qv5;qv5_i];
    qv6=[qv6;qv6_i];
end
T_sim=(N-1)*T; % duración de la simulación

smiData=datosBrazo(); % datos para la simulación


 sim('simulacion_brazo_obstaculos.slx') % ejecuta el modelo de Simulink en Simscape
```

10.1.25. Función Trayectoria_RRT_6D

```
function [puntos] = Trayectoria_RRT_6D(objetivo,inicio)
 %calcula la trayctoria RRT a recooriendo el árbol desde el nodo final hasta
% el inical
puntos=objetivo;
p=objetivo;
final=0;
while final==0
 Previo=p.nodo_predecesor;
 if Previo.q1==inicio.q1 && Previo.q2==inicio.q2 && Previo.q3==inicio.q3 && Previo.
q4==inicio.q4 && Previo.q5==inicio.q5 && Previo.q6==inicio.q6
    final=1;
 end
puntos=[Previo puntos];
p=Previo;
end
end
```

10.1.26. *Script* para crear el árbol de configuraciones mediante RRT*

```
i=1;

final=0;
tic

% hasta que se llegue al final o a las iteraciones máximas

while i< max_iteraciones && final==0

    % Crear un nuevo nodo;


    nuevo_nodo=crear_nodo_conf_RRT_estrella(arbol,long,lista_puntos,min_distancia,
T,q1_min,q1_max,q2_min,q2_max,q3_min,q3_max,q4_min,q4_max,q5_min,q5_max,q6_min,
q6_max,[goal.q1,goal.q2,goal.q3,goal.q4,goal.q5,goal.q6],sesgo,i,Suelo_a6,
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)



% Añadirlo al árbol:

arbol=[arbol nuevo_nodo];

% Poner al día la lista de puntos:
lista_puntos=[lista_puntos;[nuevo_nodo.q1 nuevo_nodo.q2 nuevo_nodo.q3 nuevo_nodo.q4
nuevo_nodo.q5 nuevo_nodo.q6]];



% Verificar si el nodo nuevo está suficientemente cerca del nodo objetivo.

if sqrt(5*(goal.q1-nuevo_nodo.q1)^2+4*(goal.q2-nuevo_nodo.q2)^2+3.5*(goal.q3-
nuevo_nodo.q3)^2+0.5*(goal.q4-nuevo_nodo.q4)^2+0.25*(goal.q5-nuevo_nodo.q5)^2
<min_distancia
   % verificar si se puede unir sin colisionar
    if Colision_Curva_MoveAbsJ_Obstaculos([nuevo_nodo.q1,nuevo_nodo.q2,nuevo_nodo.
q3,nuevo_nodo.q4,nuevo_nodo.q5,nuevo_nodo.q6],[goal.q1,goal.q2,goal.q3,goal.q4
goal.q5,goal.q6],T,Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6
Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4
Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2
Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4
Obstaculo2_a2_int,Obstaculo3_a2_int,Obstaculo4_a2_int,Obstaculo2_a4_int
Obstaculo3_a4_int,Obstaculo4_a4_int)==0 % Verificar que no hay obstáculos en medio
        arbol=[arbol goal];
        goal.nodo_predecesor=nuevo_nodo;
        goal_numero=i+1;
        goal.coste=nuevo_nodo.coste+sqrt(5*(goal.q1-nuevo_nodo.q1)^2+4*(goal.q2-
nuevo_nodo.q2)^2+3.5*(goal.q3-nuevo_nodo.q3)^2+0.5*(goal.q4-nuevo_nodo.q4)^2+0.25*
(goal.q5-nuevo_nodo.q5)^2);
        final=1;
        dibujar_trayectoria([nuevo_nodo.q1,nuevo_nodo.q2,nuevo_nodo.q3,nuevo_nodo.
q4,nuevo_nodo.q5,nuevo_nodo.q6],[goal.q1,goal.q2,goal.q3,goal.q4,goal.q5,goal.
q6],'k',0.5);
    end
```

```
end

i=i+1;

end
toc

if final==1
% Se calcula el camino
puntos=Trayectoria_RRT_6D(goal,start);

% Representación de los puntos y trayectorias
N=size(puntos,2);

for i=2:1:N-1
    Trans=Cinematica_Directa(DegToRad(puntos(1,i).q1),DegToRad(puntos(1,i).q2),
DegToRad(puntos(1,i).q3),DegToRad(puntos(1,i).q4),DegToRad(puntos(1,i).q5),DegToRad
(puntos(1,i).q6));

    DibujarPuntos6D(Trans,long,'o',2);
end

for i=1:1:N-1
    dibujar_trayectoria([puntos(1,i).q1,puntos(1,i).q2,puntos(1,i).q3,puntos(1,i).
q4,puntos(1,i).q5,puntos(1,i).q6],[puntos(1,i+1).q1,puntos(1,i+1).q2,puntos(1,i+1).
q3,puntos(1,i+1).q4,puntos(1,i+1).q5,puntos(1,i+1).q6],'r',3);
end


%% Representación de la trayectoria (sin el resto del árbol)
figure
plot(Suelo,'color','y')
hold on
plot(Env_1_Base,'color','y')
plot(Env_2_Base,'color','y')
plot(Obstaculo1,'color','y')
plot(Obstaculo2,'color','y')
plot(Obstaculo3,'color','y')
plot(Obstaculo4,'color','y')
title('Trayectoria generada mediante RRT')
xlim([-1000 1000])
ylim([-1000 1000])
zlim([0 2000])
xlabel('x (mm)')
ylabel('y (mm)')
zlabel('z (mm)')

% Punto inicial en coordenadas cartesianas
Trans=Cinematica_Directa(DegToRad(start.q1),DegToRad(start.q2),DegToRad(start.q3),
DegToRad(start.q4),DegToRad(start.q5),DegToRad(start.q6));
% Representación del punto inicial
DibujarPuntos6D(Trans,long,'*',2);
% Punto final en coordenadas cartesianas
Trans=Cinematica_Directa(DegToRad(goal.q1),DegToRad(goal.q2),DegToRad(goal.q3),
DegToRad(goal.q4),DegToRad(goal.q5),DegToRad(goal.q6));
% Representación del punto final
DibujarPuntos6D(Trans,long,'s',2);
```

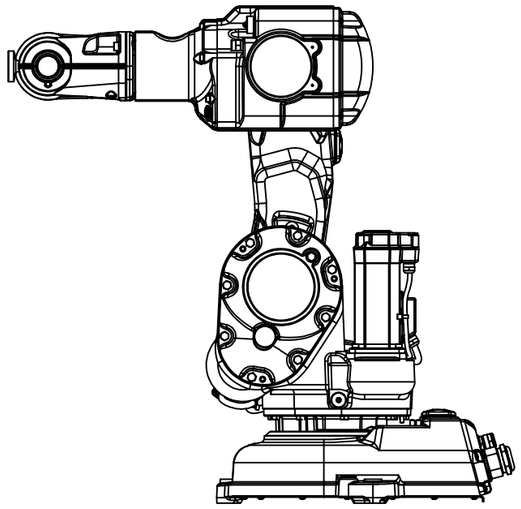
```
for i=2:1:N-1
    Trans=Cinematica_Directa(DegToRad(puntos(1,i).q1),DegToRad(puntos(1,i).q2),
DegToRad(puntos(1,i).q3),DegToRad(puntos(1,i).q4),DegToRad(puntos(1,i).q5),DegToRad
(puntos(1,i).q6));

    DibujarPuntos6D(Trans,long,'o',2);
end

for i=1:1:N-1
    dibujar_trayectoria([puntos(1,i).q1,puntos(1,i).q2,puntos(1,i).q3,puntos(1,i).
q4,puntos(1,i).q5,puntos(1,i).q6],[puntos(1,i+1).q1,puntos(1,i+1).q2,puntos(1,i+1).
q3,puntos(1,i+1).q4,puntos(1,i+1).q5,puntos(1,i+1).q6],'r',3);
end

end
```

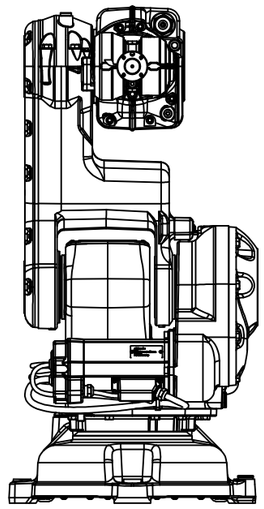
10.1.27. Función crear_nodo_conf_RRT_estrella

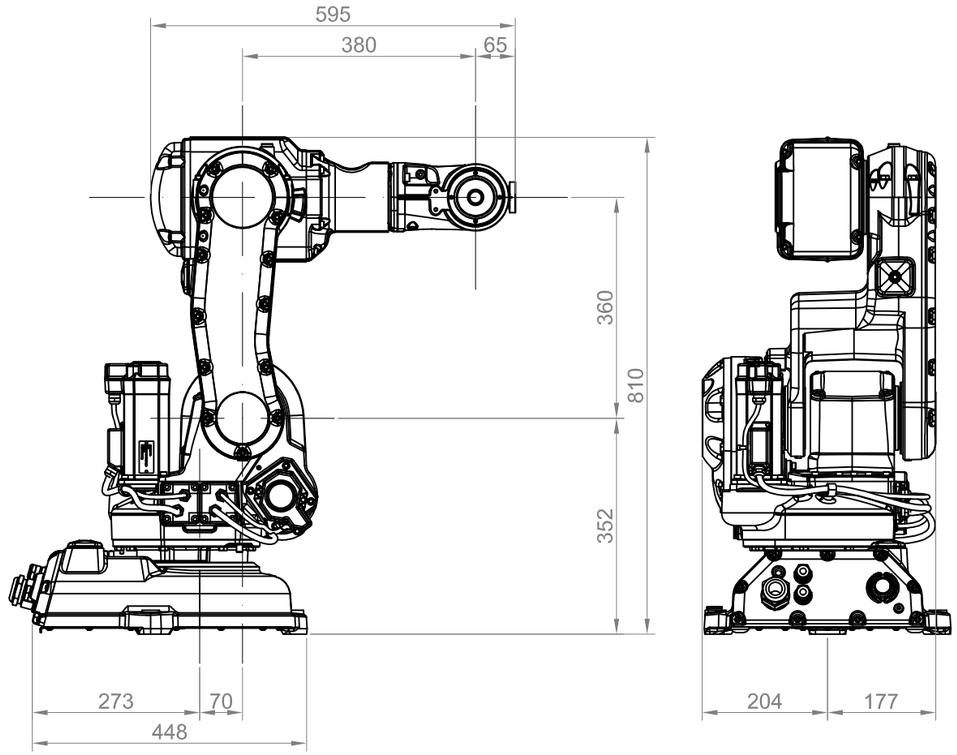
```
function [nuevo_nodo] = crear_nodo_conf_RRT_estrella(arbol,long,lista_puntos,
longitud,T,q1_min,q1_max,q2_min,q2_max,q3_min,q3_max,q4_min,q4_max,q5_min,q5_max,
q6_min,q6_max,objetivo,sesgo,numero,Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,
Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,
Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,Obstaculo4_a4,
Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,Choque_link1_a6,
Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,Obstaculo4_a2_int,
Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)
%Crea un nuevo nodo

Condicion=1;  % Condicion de que se pueda unir con algunos vecino sin intersectar
nigun obstáculo.
corta=1;


while Condicion ==1   % Hasta encontrar un punto que se pueda conectar con algún
vecino

% Nueva configuración aleatoria
configuracion=crear_configuracion_aleatoria_RRT_6D_OBS(q1_min,q1_max,q2_min,q2_max,
q3_min,q3_max,q4_min,q4_max,q5_min,q5_max,q6_min,q6_max,objetivo,sesgo,Suelo_a6,
Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,
Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,
Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,
Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int);


% Se busca el vecino más próximo:

% Ordenar los puntos según su distancia al nuevo nodo.

[puntos,  distancias,num]  =  Ordenar_ptos_6D(configuracion,lista_puntos);


M=size(puntos,1);

i=1;
while i<=M   && corta==1 % recorre el árbol mientras la recta corte

q_near=puntos(i,:);
numero_predecesor=num(i);
d=distancias(i);
numero_predecesor=num(i)-1;

previo=arbol(num(i));

Vector=[configuracion(1)-q_near(1),configuracion(2)-q_near(2),configuracion(3)-
q_near(3),configuracion(4)-q_near(4),configuracion(5)-q_near(5),configuracion(6)-
q_near(6)];

% Calcular el candidato a q_new:
Vector_unitario=Vector/d;
q_new_candidato=q_near+Vector_unitario*longitud;

corta=0;
% Se ordenan los puntos por cercanía a q_new_candidato
```

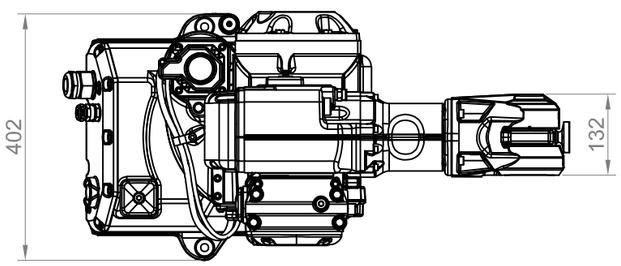
```
[puntosP, distanciasP,numP] = Ordenar_ptos_6D(q_new_candidato,lista_puntos);
radio=300; % busca cercanos en un radio de 150 grados en el espacio de
configuraciones
cont=1;
min_coste=Inf;
M2=size(puntosP,1);


while distanciasP(cont)<radio  &&  cont<M2 % distancias menores que el radio
    if arbol(numP(cont)).coste<min_coste % si el coste es menor que el mínimo
        candidato_previo=arbol(numP(cont)); % actualiza el coste
        % verifica si se pueden unir
        if Colision_Curva_MoveAbsJ_Obstaculos(q_new_candidato,[candidato_previo.q1,
candidato_previo.q2,candidato_previo.q3,candidato_previo.q4,candidato_previo.q5,
candidato_previo.q6],T,Suelo_a6,Env_1_Base_a6,Env_2_Base_a6,Obstaculo1_a6,
Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,Env_1_Base_a4,Env_2_Base_a4,
Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,Obstaculo4_a4,Obstaculo1_a2,
Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,Choque_link1_a6,Choque_link1_a4,
Obstaculo2_a2_int,Obstaculo3_a2_int,Obstaculo4_a2_int,Obstaculo2_a4_int,
Obstaculo3_a4_int,Obstaculo4_a4_int)==0
            previo=candidato_previo;
            numero_predecesor=numP(cont)-1;
            min_coste=arbol(numP(cont)).coste;
        end
    end
    cont=cont+1;
end



%% VERIFICAR QUE LA RECTA NO COLISIONA
                if Colision_Curva_MoveAbsJ_Obstaculos(q_new_candidato,[previo.q1,
previo.q2,previo.q3,previo.q4,previo.q5,previo.q6],T,Suelo_a6,Env_1_Base_a6,
Env_2_Base_a6,Obstaculo1_a6,Obstaculo2_a6,Obstaculo3_a6,Obstaculo4_a6,Suelo_a4,
Env_1_Base_a4,Env_2_Base_a4,Obstaculo1_a4,Obstaculo2_a4,Obstaculo3_a4,
Obstaculo4_a4,Obstaculo1_a2,Obstaculo2_a2,Obstaculo3_a2,Obstaculo4_a2,
Choque_link1_a6,Choque_link1_a4,Obstaculo2_a2_int,Obstaculo3_a2_int,
Obstaculo4_a2_int,Obstaculo2_a4_int,Obstaculo3_a4_int,Obstaculo4_a4_int)==0
                    % % % Añadir el nuevo nodo:
                    q_new=q_new_candidato;
                    nuevo_nodo=nodo_conf_6D;
                    nuevo_nodo.nombre=sprintf('nodo%04d',numero);
                    nuevo_nodo.numero=numero;
                    sprintf('nodo%04d',numero);
                    nuevo_nodo.predecesor=numero_predecesor;
                    nuevo_nodo.nodo_predecesor=previo;
                    nuevo_nodo.q1=q_new(1);
                    nuevo_nodo.q2=q_new(2);
                    nuevo_nodo.q3=q_new(3);
                    nuevo_nodo.q4=q_new(4);
                    nuevo_nodo.q5=q_new(5);
                    nuevo_nodo.q6=q_new(6);
                    nuevo_nodo.coste=previo.coste+sqrt(5*(previo.q1-nuevo_nodo.q1
^2+4*(previo.q2-nuevo_nodo.q2)^2+3.5*(previo.q3-nuevo_nodo.q3)^2+0.5*(previo.q4-
nuevo_nodo.q4)^2+0.25*(previo.q5-nuevo_nodo.q5)^2);
                    Condicion=0;
                    % Prepresentación del nodo y de la trayectoria de unión
```

```
                    % con el nodo anterior
                    Trans=Cinematica_Directa(DegToRad(nuevo_nodo.q1),DegToRad
(nuevo_nodo.q2),DegToRad(nuevo_nodo.q3),DegToRad(nuevo_nodo.q4),DegToRad
(nuevo_nodo.q5),DegToRad(nuevo_nodo.q6));

                    DibujarPuntos6D(Trans,long,'o',0.5);

                    dibujar_trayectoria([previo.q1,previo.q2,previo.q3,previo.q4,
previo.q5,previo.q6],[nuevo_nodo.q1,nuevo_nodo.q2,nuevo_nodo.q3,nuevo_nodo.q4,
nuevo_nodo.q5,nuevo_nodo.q6],'k',0.5);
                else
                     corta=1;
                end
               i=i+1;
end


end

end
```









PROYECTO: Simulación de métodos de planificación de movimiento de robots
TITULAR: Universitat Politènica de València

Fecha: 05/05/2022

Escala
1:10

Autor: Luis Martínez Martínez

Plano:
IRB 140

Plano Nº
01