Universidad Miguel Hernández de Elche

MÁSTER UNIVERSITARIO EN ROBÓTICA

"Planificación avanzada de trayectorias de un robot móvil binario con estructura paralela"

Trabajo Fin de Máster

Curso 2021-2022

Autor: Mario Pérez Checa

Tutor: Adrián Peidró Vidal

ÍNDICE

Capítulo 1. Introducción.

1.1. Antecedentes del Trabajo Fin de Máster.

Los robots binarios desarrollan su movimiento a partir de actuadores que solamente tienen dos estados, 0 o 1 (binarios). La principal ventaja de los actuadores binarios es la simplicidad en el control, ya que solo es necesario, a lo sumo, detectar si cada uno de los actuadores ha alcanzado alguna de sus posiciones extremas. Este tipo de actuadores también simplifican la planificación de trayectorias, puesto que estos tienen espacios de trabajo discretos, es decir, solo pueden alcanzar posiciones puntuales aisladas.

Debido a esa discretización del espacio de trabajo, en casos donde la precisión es un requerimiento prioritario, el robot debe disponer de un número más elevado de actuadores binarios, que resultan en robots binarios hiper-redundantes, que consisten en concatenar módulos binarios uno tras otro, como si se tratara de una serpiente. Algunos ejemplos de robots binarios hiper-redundantes fueron propuestos por los siguientes investigadores: *Clysdale y Sun (2005)*, *Maeda and Konaka (2014)*, *Tzorakoleftherakis et al. (2016)*, *Sujan and Dubowsky (2004)*, *Miao et al. (2014)*, *Chirikjian et al. (1995-1996-1997)*.

En contraste con los robots hiper-redundantes, también han sido analizados robots binarios más simples con pocos grados de libertad. Algunos de estos robots son los desarrollados por: *Schutz et al. (2010-2013)*, *Zhou (2003)*, *Chen y Yeo (2002)*. Dentro de esta categoría se encuentra el robot Xrobin (ROBot BINario en forma de X), que es el que se desarrolla en este Trabajo Fin de Máster.

En el Grupo de Automatización, Robótica y Visión por Computador (ARVC) de la UMH, durante los últimos años se ha estado trabajando en el desarrollo de un nuevo robot móvil binario con pocos actuadores, pero con una elevada movilidad. Como ya se ha comentado, este robot se llama Xrobin, y se caracteriza por tener dos actuadores binarios dispuestos de forma cruzada.

Como se ha demostrado en trabajos de investigación previos, *Peidró et al. (2015)*, este robot puede realizar transiciones no-singulares, que son transiciones entre distintas soluciones de la cinemática directa para un mismo valor de los actuadores. Gracias a esto, y como se demostró en un Trabajo Fin de Grado previo, *García-Martínez (2020)*,

disponiendo los actuadores binarios de este robot de forma cruzada le permite adoptar el doble de configuraciones que alcanzaría si estos no se cruzasen.

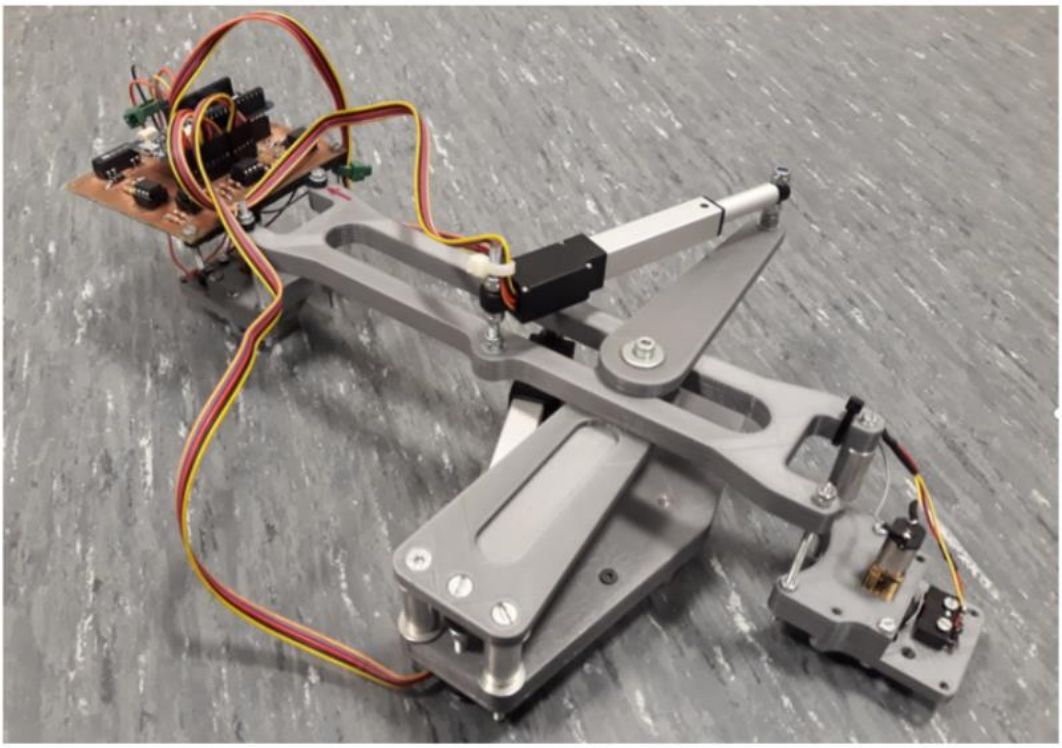

Figura 1: Prototipo del robot Xrobin. Figura extraída de Peidró et al. (2015) con permiso.

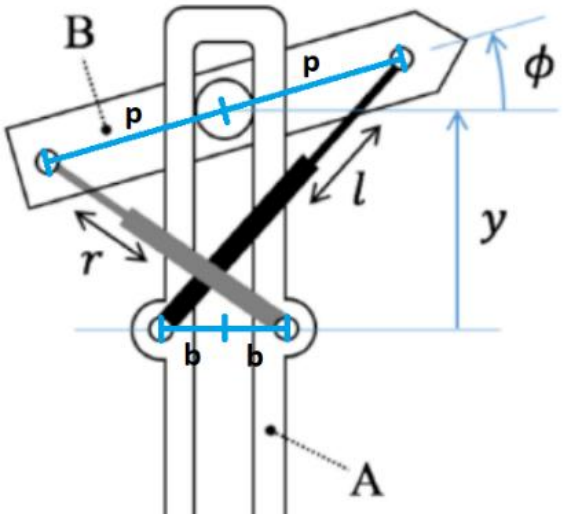

Figura 2: Parámetros del robot.

Las ecuaciones que resuelven la cinemática inversa del robot se encuentran expresadas en función de los parámetros de la Figura 2 y son las siguientes:

$$l = \sqrt{(p * \cos(\phi) + b)^2 + (p * \sin(\phi) + y)^2}$$

$$r = \sqrt{(-p * \cos(\phi) - b)^2 + (-p * \sin(\phi) + y)^2}$$

El robot objeto de este Trabajo Fin de Máster está protegido por patente (*Peidró et al., 2020*), y cuyo prototipo, mostrado en la Figura 1, se desarrolló en un Trabajo Fin de Grado previo a este, *García-Martínez (2020)*. Este prototipo es capaz de realizar movimientos básicos como avanzar, retroceder, girar 90º hacia la derecha, o girar 90º hacia la izquierda; es decir, solo se programaron una serie de movimientos básicos predeterminados. En dicho trabajo no se resolvió la planificación de trayectorias complejas que permitan alcanzar cualquier posición y orientación deseadas, sino que esto se resolvió en (*Pérez, 2021*), el Trabajo Fin de Grado previo a este Trabajo Fin de Máster.

Para realizar los movimientos mencionados anteriormente, se sigue el siguiente principio: el movimiento del robot se basa en pegar uno de sus cuerpos con imanes o ventosas, mover el otro cuerpo y cambiar la adhesión de los cuerpos para repetir el proceso. Un término que se va a utilizar una y otra vez en este Trabajo Fin de Grado es el de "ciclo", por lo que es necesario entender que un ciclo consiste en el movimiento del robot tras dos despegues de imanes o ventosas. Es decir, un cuerpo empieza fijo y el otro comienza a moverse tras su despegue, a continuación, el segundo cuerpo se fija y se despega y comienza a mover el primero. Una vez este primer cuerpo ha finalizado su movimiento, se considera el final del ciclo de movimiento.

Como se ha comentado anteriormente, la planificación de trayectorias de este robot móvil se planteó en (*Pérez, 2021*), y este consistió en buscar la secuencia de configuraciones que el robot debía adoptar para alcanzar la posición y orientación que se le imponga al inicio del movimiento en diversas situaciones. Dichas situaciones fueron las siguientes: alcanzar la posición más cercana a la deseada dentro del espacio de trabajo del robot, alcanzar la mejor combinación de posición y orientación posible dentro del espacio de trabajo del robot, alcanzar la posición más cercana a la deseada fuera del espacio de trabajo del robot mediante solapamiento de espacios de trabajo, alcanzar la posición y orientación más cercanas a las deseadas fuera del espacio de trabajo del robot mediante solapamiento de espacios de trabajo y reorientación final y la consideración de obstáculos, restringiendo tanto puntos como zonas del espacio.

Como se ha comentado anteriormente, este Trabajo Fin de Máster es la continuación de (*Pérez, 2021*), es por ello que, al disponerse de todos los códigos utilizados en dicho trabajo, estos constituirán el punto de partida del presente trabajo, a

partir de los cuales se han desarrollado todos los códigos y principios básicos de este proyecto.

1.2. Objetivos

El objetivo del presente Trabajo Fin de Máster es, en pocas palabras, resolver los trabajos futuros propuestos en (*Pérez, 2021*), los cuales fueron los siguientes: comparar el solapamiento de espacios de trabajo con elevar el número de ciclos de movimiento, cosa que no se pudo hacer en dicho trabajo debido al problema que supone la búsqueda por fuerza bruta; la optimización de la planificación de trayectorias con el fin de reducir o suprimir la circunferencia circunscrita al robot que permite realizar la comprobación de colisiones, o lo que es lo mismo, optimizar la comprobación de colisiones; y finalmente, la optimización del escape de mínimos locales para obstáculos definidos por zonas del espacio. Además, se ha decidido añadir un objetivo extra a esta lista, que es el desarrollo de uno o varios algoritmos que directamente eviten entrar en mínimos locales, para lo que se aplicarán algoritmos que utilizarán el mapa de ocupación del espacio.

1.3. Estructura de la memoria

Con el fin de explicar brevemente cómo se van a cumplir los objetivos planteados anteriormente, en esta sección se pretende plantear un resumen de cómo ha sido estructurada esta memoria. A continuación, se expondrán brevemente los contenidos de los siguientes capítulos.

El capítulo 2 consiste en la resolución de la evasión de obstáculos con mínimos locales, es decir, lo correspondiente con el objetivo planteado como optimizar el escape de mínimos locales para obstáculos definidos como zonas del espacio. Con este fin, se han desarrollado dos algoritmos que permiten cumplir este objetivo: el primero sigue el mismo principio que el expuesto en (*Pérez, 2021*), en el cual se intenta enviar al robot a una posición aleatoria intermedia antes de retomar su destino inicial, pero en este caso ha sido mejorado y optimizado, y el segundo se encarga de hacer al robot moverse en la dirección de la “pared” del obstáculo que le obstruye el movimiento hasta que este pueda rodearlo por completo. Además, en esta sección también se ha planteado el algoritmo que pretende optimizar la comprobación de colisiones del robot con los obstáculos, tal y como se ha planteado en la sección anterior.

En el capítulo 3, se van a plantear los algoritmos de planificación de trayectorias conocido el mapa de ocupación comentados en la sección anterior. Dichos algoritmos son comúnmente conocidos como A* y diagrama de Voronoi. El primero consiste en la búsqueda heurística del camino de menor coste entre dos celdas de una malla que representa el espacio de trabajo, con o sin obstáculos, y el segundo consiste en la búsqueda de los puntos equidistantes a dos obstáculos, de tal forma que estos forman unos caminos cuyos puntos equidistan siempre de sus dos obstáculos más cercanos.

Finalmente, en el capítulo 5 se van a exponer las medidas tomadas para satisfacer el objetivo que queda por cumplir: la búsqueda de un método o algoritmo que sea capaz de encontrar una solución al problema de elevar el número de ciclos de movimiento del robot, evitando el método de fuerza bruta; cabe destacar que una de las principales razones de querer cumplir este objetivo es la de poder comparar el resultado obtenido mediante el algoritmo de solapamiento de espacios de trabajo desarrollado en (*Pérez, 2021*) con un algoritmo de búsqueda que aplique un criterio más fiable que el que se expuso en dicho trabajo. Para ello, se han desarrollado los siguientes algoritmos: dos basados en inteligencia artificial, que son el entrenamiento de una red neuronal y el desarrollo de un algoritmo genético, y uno tratando al robot móvil como un manipulador hiper-redundante de varios módulos binarios.

Capítulo 2. Algoritmos basados en condiciones locales.

En este primer capítulo, se pretende mejorar y optimizar los algoritmos que se emplearon en el proyecto anterior a este, que corresponde con la referencia (*Pérez, 2021*). En concreto, se van a mejorar tres algoritmos principales: la detección de colisiones, el escape de mínimos locales a través de generación aleatoria de destinos intermedios y el aumento de precisión en la posición final del robot para los últimos ciclos de movimiento, cuando este adopta la orientación deseada.

2.1. Algoritmo de comprobación de colisiones.

El primer algoritmo que se va a explicar es la detección de colisiones. En (*Pérez, 2021*), este consistía en la detección de los obstáculos cercanos al robot a un radio determinado, es decir, se circunscribía al robot dentro de una circunferencia que asegurara que este pudiera mover los eslabones dentro de la misma para el número de ciclos de movimiento que se le asignara, generalmente 2. En este caso, se pretende ajustar mucho más la comprobación de colisiones a las áreas por donde se van a mover los eslabones en cada medio ciclo de movimiento. Este es un algoritmo implementado en la mayoría de los anexos, pero en esta explicación solo se hará referencia al expuesto en el Anexo 1, a pesar de que sea igual a los del resto de anexos. Los *scripts* que conforman este algoritmo son: **area_local.m**, **calculo_area_adyac.m**, **inversa.m**, **directa.m**, **calculo_area.m**, **codigo_kdtree.m** y **comprobacion_colisiones.m**. Cabe destacar que este algoritmo solamente se explicará en esta sección, y para el resto de este Trabajo fin de Máster se remitirá a esta cada vez que se emplee.

Lo primero que requiere este algoritmo es la ejecución de la función **area_local.m**, que se encarga de obtener las áreas barridas por los eslabones en cada medio ciclo de movimiento en coordenadas locales y simplificarlas de tal forma que sean óptimas computacionalmente, pero a la vez sean lo más fieles posibles a las reales.

Antes de explicar el código de esta función, es necesario explicar la función **calculo_area_adyac.m**. Como se explicó en (*Pérez, 2021*) y como se ha comentado brevemente en la introducción, el robot se mueve mediante ciclos de movimiento, pero este no es capaz de adoptar cualquier posición en cualquier momento, sino que tiene que moverse solo a configuraciones adyacentes cinemáticamente a la actual. Es por ello que, para calcular el área barrida por los eslabones, es necesario obtener el movimiento del

robot pasando por todas las posibles configuraciones contiguas. Esta función se encarga de tomar los puntos por los que pasa un eslabón entre una configuración y la siguiente; para ello, recibe como parámetros de entrada los índices de las configuraciones adyacentes entre las que se quiere obtener el área recorrida, la posición y orientación del eslabón en el instante que se quiere calcular el área recorrida (para este Trabajo, el origen de coordenadas en todos los casos), y el área calculada para el resto de configuraciones en instantes anteriores. Con estos parámetros, se va a proceder a explicar el código de esta función.

```
 2 -   m = 20;
 3 -   p = 101.31;
 4 -   b = 18.59;
 5 -   Barr_aux = Barr;
 6 -   phi_v = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
 7 -   y_v = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,21.95478428];
 8 -
 9 -   phi_inicial = phi_v(i);
10 -   y_inicial = y_v(i);
11 -   phi_final = phi_v(f);
12 -   y_final = y_v(f);
13 -
14 -   [li,ri] = inversa(phi_inicial,y_inicial);
15 -   [lf,rf] = inversa(phi_final,y_final);
16 -   dl = (lf-li)/(m-1);
17 -   dr = (rf-ri)/(m-1);
18 -
19 -   l = li;
20 -   r = ri;
21 -   phi = phi_inicial;
22 -   y = y_inicial;
```

En primer lugar, se definen los parámetros que se van a emplear en el cálculo del área. Entre las líneas 2-7 se definen los parámetros de diseño del robot, así como se almacena el área calculada en instantes anteriores en una variable auxiliar (línea 5). A continuación, se definen las variables del instante actual del movimiento del eslabón (líneas 9-12), y se define la longitud inicial de los actuadores calculando la cinemática inversa mediante la función **inversa.m**, que se desarrolló en (*Pérez, 2021*) y se encuentra reflejada en el Anexo 1. Posteriormente, se definen los parámetros iniciales (líneas 19-22) que se van a ir actualizando en un posterior bucle.

```
24 -   for i = 1:m
25 -
26 -     [phi,y] = directa(l,r,phi,y);
27 -     l = l+dl;
28 -     r = r+dr;
29 -
30 -     T_BA = [ cos(phi)  -sin(phi)    0;
31 -              sin(phi)   cos(phi)    y;
32 -                 0          0        1];
```

```
33 -      if movB == 1
34 -        T_A = [ cos(ori_0) -sin(ori_0)  pos_0(1);
35 -                sin(ori_0) cos(ori_0)   pos_0(2);
36 -                     0          0           1   ];
37 -        T_B = T_A*T_BA;
38 -        B1 = T_B*[p;0;1];
39 -        B2 = T_B*[-p;0;1];
40 -        Barr_aux = [Barr_aux; B1(1) B1(2) B2(1) B2(2)];
41 -      else
42 -        T_B = [cos(ori_0) -sin(ori_0)  pos_0(1);
43 -               sin(ori_0) cos(ori_0)   pos_0(2);
44 -                    0          0           1   ];
45 -        T_A = T_B*inv(T_BA);
46 -        altura_pasador = 70;
47 -        radio_pasador = b/2;
48 -        a11 = T_A*[-radio_pasador;altura_pasador;1];
49 -        a12 = T_A*[ radio_pasador;altura_pasador;1];
50 -        a21 = T_A*[-radio_pasador;-altura_pasador;1];
51 -        a22 = T_A*[ radio_pasador;-altura_pasador;1];
52 -        Barr_aux = [Barr_aux; a11(1) a11(2) a12(1) a12(2) a21(1) a21(2) a22(1) a22(2)];
53 -      end
54 -    end
55 -
56 -    Barr = Barr_aux;
57 -
58 - end
```

En este bucle, se calculan *m* puntos (línea 24) que se encuentran en la trayectoria del eslabón entre las configuraciones *i* y *j*. Para ello, primero se debe calcular la cinemática directa para conocer la posición y orientación del eslabón en cada instante, ya que, aunque se calcule siempre el área en el origen, se pretende calcular el movimiento de un eslabón en movimiento, por ello se debe calcular la posición y orientación en cada instante, por lo que se debe emplear la función **directa.m**, desarrollada en (*Pérez, 2021*) y reflejada en el anexo 1. Una vez obtenidas posición y orientación (línea 26), y actualizadas las variables de longitud de los actuadores (líneas 27-28) con los incrementos definidos en las líneas 16-17, se procede a calcular las posiciones de los puntos característicos del eslabón que corresponda. Las posiciones calculadas hasta ahora son las del centro de un eslabón con respecto al otro, que se encuentra fijado en el origen, por lo que ahora se deben calcular las posiciones de los extremos del eslabón en movimiento.

En primer lugar, se calcula la matriz de transformación entre los eslabones (líneas 30-32), y se procede a calcular los puntos característicos del eslabón correspondiente. En el caso de que se mueva el eslabón B (línea 33), se calcula la matriz de transformación del eslabón fijo (A) con respecto al origen, (que en este y todos los casos que se expondrán, será la identidad, pero se hace porque se contempla la posibilidad de que el área no se quiera calcular en el origen), y se calcula la transformación entre el eslabón A y B (línea 37) y con ella se obtienen los puntos extremos del eslabón B y se almacenan

en una variable (líneas 38-40). Por el contrario, para el caso en que se mueve el eslabón A (línea 41) el procedimiento es similar, pero esta vez se calculan 4 puntos, ya que el eslabón A es un rectángulo, como se puede apreciar en la Figura 2. Finalmente, se termina la función devolviendo todos los puntos calculados para las configuraciones anteriores y para la actual definida por *i* y *j*. Una vez explicada esta función, se continúa con la explicación de **area_local.m**.

Esta función recibe como parámetros de entrada la posición y orientación donde se va a calcular el área en coordenadas locales; como ya se ha mencionado, para este Trabajo, en todos los casos, se van a calcular las áreas en el origen de coordenadas y para orientación 0 radianes, pero la función está diseñada para poder ser empleada en cualquier posición y orientación.

```
 3 -   v = [1,2,3,4,5,6,7,8,1];
 4 -   BarrB = [];
 5 -   BarrA = [];
 6 -   for t = 1:8
 7 -     BarrB = calculo_area_adyac(v(t),v(t+1),1,P0,phi0,BarrB);
 8 -   end
 9 -   areaB = BarrB;
10 -   v_aux = 101.31*ones(1,20);
11 -   v_aux_p = -50.242:5.2886:50.242;
12 -   v_t = transpose([v_aux; v_aux_p; ones(1,20)]);
13 -   v_t_2 = transpose([-v_aux; v_aux_p; ones(1,20)]);
14 -   areaB = [areaB(40:80,1) areaB(40:80,2);
15 -            v_t(:,1) v_t(:,2);
16 -            areaB(120:160,1) areaB(120:160,2);
17 -            areaB(1:41,3)  areaB(1:41,4);
18 -            v_t_2(:,1) v_t_2(:,2);
19 -            areaB(80:120,3) areaB(80:120,4)];
```

En primer lugar, se calcula el área barrida por el eslabón B entre las 8 configuraciones posibles (líneas 6-9), cuyo resultado es el que se muestra en la Figura 3.a.

Este resultado, a pesar de ser el más exacto posible, no es computacionalmente muy eficiente, puesto que tiene demasiados puntos, por lo que se va a simplificar y se va a tomar como que los límites laterales están definidos por rectas, y las líneas de puntos interiores se van a suprimir, puesto que solamente interesan los límites del área barrida por el eslabón. Estas operaciones se realizan en las líneas 10-13, y el resultado es el que se puede apreciar en la Figura 3.b.

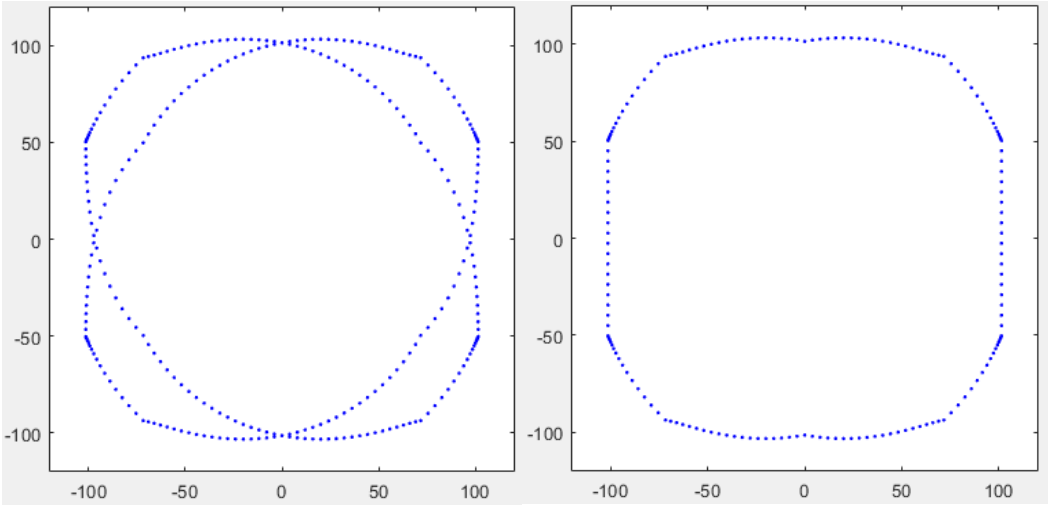

Figura 3: a) área barrida por el eslabón B. b) área de B simplificada.

Para el caso de movimiento del eslabón A, el procedimiento es prácticamente idéntico, y se muestra en el siguiente fragmento de código.

```
20 -   for u = 1:8
21 -     BarrA = calculo_area_adyac(v(u),v(u+1),0,P0,phi0,BarrA);
22 -   end
23 -   areaA = BarrA;
24 -   v_aux_2p = -9.295:3.718:9.295;
25 -   v_aux_3p = 120.242*ones(1,6);
26 -   v_aux_4p = 75.668*ones(1,9);
27 -   v_aux_5p = -22.5656:5.6414:22.5656;
28 -   v_t_p = transpose([v_aux_2p; v_aux_3p; ones(1,6)]);
29 -   v_t_2p = transpose([v_aux_4p; v_aux_5p; ones(1,9)]);
30 -   v_t_3p = transpose([v_aux_2p; -1*v_aux_3p; ones(1,6)]);
31 -   v_t_4p = transpose([-1*v_aux_4p; v_aux_5p; ones(1,9)]);
32 -   areaA = [areaA(52:80,1) areaA(52:80,2);
33 -            v_t_p(:,1) v_t_p(:,2);
34 -            areaA(80:110,3) areaA(80:110,4);
35 -            v_t_2p(:,1) v_t_2p(:,2);
36 -            areaA(1:30,7) areaA(1:30,8);
37 -            v_t_3p(:,1) v_t_3p(:,2);
38 -            areaA(131:160,5) areaA(131:160,6);
39 -            v_t_4p(:,1) v_t_4p(:,2)];
```

El resultado de calcular el área barrida por el eslabón A para las 8 configuraciones posibles se encuentra reflejado en la Figura 4.a. Al igual que en el caso del eslabón B, se ha simplificado dicho resultado aproximando mediante rectas en los límites superior, inferior y laterales, y los puntos interiores también se van a suprimir, lo que da como resultado lo que se muestra en la Figura 4.b.

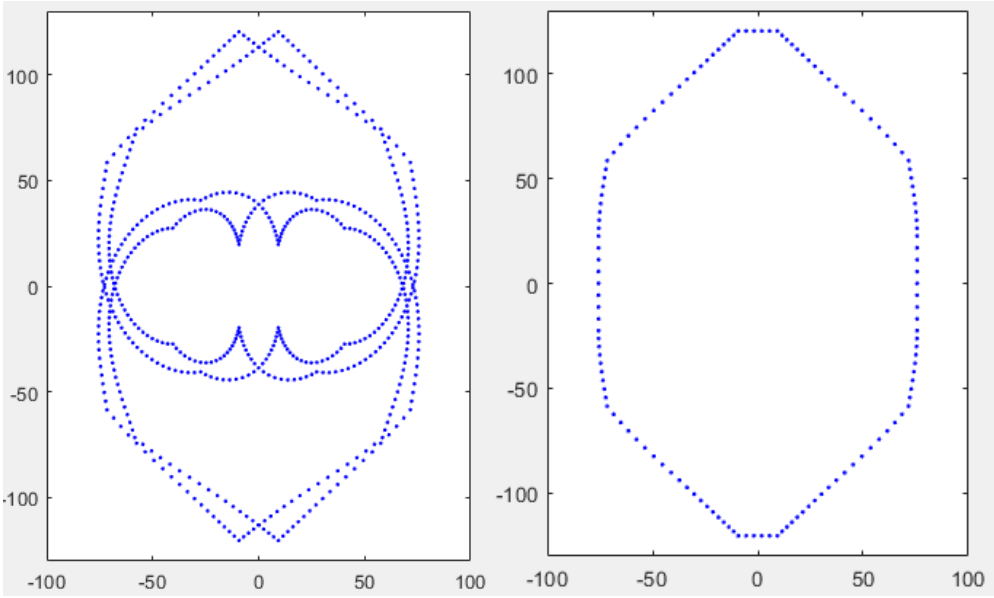

Figura 4: a) área barrida por el eslabón A. b) área de A simplificada.

Con esto, quedaría concluido el cálculo de áreas en coordenadas locales que pueden barrer los eslabones en cada medio ciclo. Con ellas, ya es posible implementar una comprobación de colisiones en cada medio ciclo de movimiento de una forma muy exacta y computacionalmente eficiente. Para ello, primero es necesario explicar las funciones **calculo_area.m** y **codigo_kdtree.m**. La primera es una función muy simple que traslada el área calculada en coordenadas locales a coordenadas globales, dependiendo de la posición y orientación del robot en cada instante de tiempo, y esta se encuentra reflejada en el Anexo 1.

Por otra parte, la función **codigo_kdtree.m** es algo más larga, pero también bastante simple. Esta también es recogida en el Anexo 1. Esta función se encarga de calcular, mediante un K-D tree definido antes de llamar a cualquier función de la comprobación de colisiones, los puntos obstáculo que se encuentran dentro de una ventana del espacio en el instante de tiempo que se requiera. Dicha ventana se define a partir de los puntos máximos y mínimos en *x* y en *y* del área del eslabón correspondiente en coordenadas globales (es decir, a partir del resultado de **calculo_area.m**) en el instante que corresponda. El resultado de esta función es un conjunto de puntos que se encuentran dentro de un rectángulo con las dimensiones de la ventana definida anteriormente, pero dichos puntos no tienen por qué estar dentro del área barrida por el eslabón, por lo que es necesario filtrarlos, y esto se hace en la función **comprobacion_colisiones.m**, la cual se explicará a continuación.

La función **comprobacion_colisiones.m** es la última en lo referente al aumento de precisión en la comprobación de colisiones, por lo que su objetivo es detectar cuándo el robot colisiona o no con los obstáculos a partir de las funciones explicadas hasta ahora.

Esta función recibe como parámetros de entrada la variable que define qué eslabón se mueve (*movB*), la posición y orientación del robot en el instante que se quiere hacer la comprobación de colisiones, la configuración del robot al inicio del movimiento (*indice*), la matriz de transformación entre los eslabones en el instante actual (*T_p*), la celda de matrices de configuraciones del robot (*T*) y el área que barre el eslabón en movimiento (*area*).

```
 2 -   Retrieved = codigo_kdtree(pos_0,area);
 3 -   if length(Retrieved) == 0
 4 -     colision = 0;
 5 -   else
 6 -     Retrieved_L = T_p\[Retrieved(:,1) Retrieved(:,2) ones(length(Retrieved(:,1)),1)]';
 7 -     x_obs_L = Retrieved_L(1,:);
 8 -     y_obs_L = Retrieved_L(2,:);
 9 -     for j = 1:length(x_obs_L)
10 -       if movB == 1
11 -         if x_obs_L(j)^2+y_obs_L(j)^2<=13891.29 && x_obs_L(j)>=-101.31 && ...
12 -           x_obs_L(j)<=101.31 && y_obs_L(j)>=-103.1332 && y_obs_L(j)<=103.1332
13 -           colision = 1;
14 -           break
15 -         else
16 -           colision = 0;
17 -         end
18 -       else
19 -         if 0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 && -0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 ...
20 -            && -0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 && 0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 ...
21 -            && x_obs_L(j)<=75.668 && x_obs_L(j)>=-75.668 && y_obs_L(j)<=120.2 && y_obs_L(j)>=-120.2
22 -           colision = 1;
23 -           break
24 -         else
25 -           colision = 0;
26 -         end
27 -       end
28 -     end
29 -   end
30 -   if movB == 1
31 -     T_rel = T{indice};
32 -     T_new = T_p*T_rel;
33 -   else
34 -     T_rel = T{indice};
35 -     T_new = T_p/T_rel;
36 -   end
37 -   pos_F = [T_new(1,3) T_new(2,3)];
38 -   ori_F = atan2(T_new(2,1),T_new(2,2));
```

El primer paso es obtener los posibles puntos obstáculo que podrían colisionar con el robot, para lo que se emplea la función **codigo_kdtree** explicada anteriormente (línea 2). Una vez se ha obtenido el resultado de dicha función, se comprueba si esta ha

encontrado algún punto que pueda provocar colisión. En caso negativo, se da el valor 0 a la variable *colision* (líneas 3-4), y en caso afirmativo, se comprueba si los puntos obtenidos están realmente dentro del área que barre el eslabón correspondiente. Para ello, se van a tomar los puntos obtenidos en coordenadas locales del robot (línea 6) y se va a comprobar si dichos puntos están dentro de las áreas en coordenadas locales, para lo que se definen dichas áreas como combinación de rectas y circunferencias, con el fin de comprobar si el punto en cuestión se encuentra a un lado o al otro de las rectas, o dentro o fuera de las circunferencias. Dichas rectas y circunferencias aproximan las áreas de los eslabones como se muestra en la Figura 5.

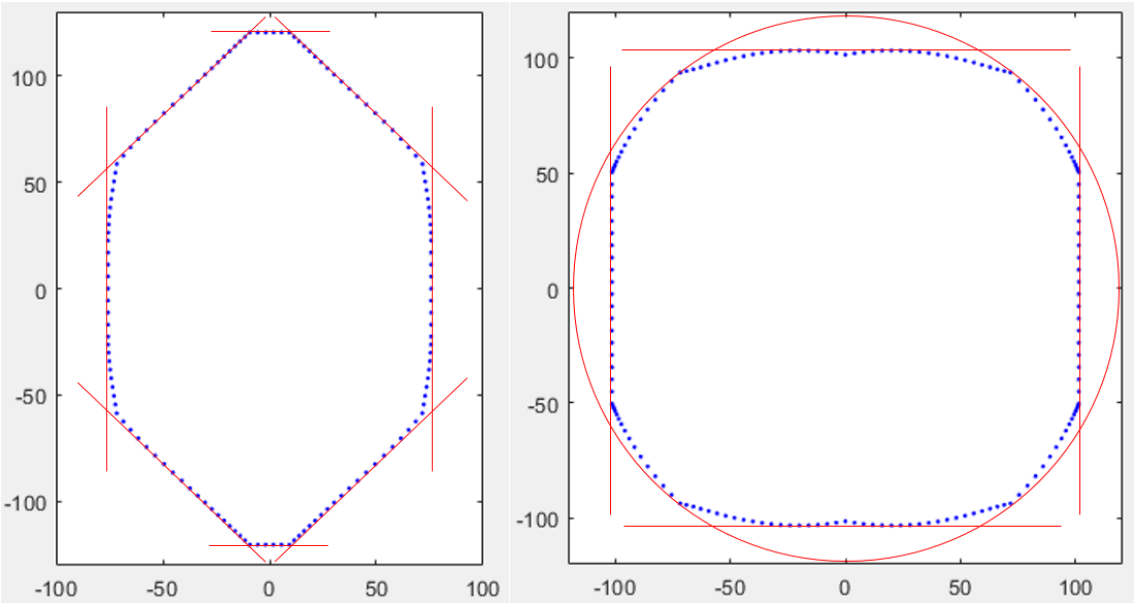

Figura 5: aproximaciones geométricas de las áreas simplificadas.

Dichas aproximaciones se aplican dependiendo de qué eslabón esté en movimiento (línea 10) y si se cumplen todas ellas (líneas 11-13 o 19-21 dependiendo del caso) se da el valor 1 a la variable *colision*, y en caso contrario se le da el valor 0. Finalmente, se calcula la posición que tendría el eslabón móvil al final del medio ciclo de movimiento del que se está haciendo la comprobación de colisiones (líneas 30-38).

2.2. Posicionamiento preciso y orientación mediante 4 ciclos.

Una vez explicada la sección anterior, ya se tiene todo lo necesario que se necesita para implementar la comprobación de colisiones con una precisión mucho mayor que se tenía en (*Pérez, 2021*). La siguiente mejora que se va a explicar del trabajo mencionado es el aumento de precisión en la posición final del robot para los últimos ciclos de movimiento, cuando este adopta la orientación deseada.

En (*Pérez, 2021*) se hizo un algoritmo que permitía al robot colocarse en la orientación deseada tras alcanzar la posición objetivo (o una posición suficientemente cercana) mediante tres ciclos de movimiento. Sin embargo, este algoritmo no presentaba una precisión demasiado grande, ya que el error final en la posición del robot se encontraba generalmente entre 3 y 8 mm. Para mejorar esta precisión, se ha decidido implementar dicha orientación final en 4 ciclos en lugar de 3, lo que hace que la precisión sea mucho mayor, ya que la densidad de puntos entre la nube de puntos del espacio de trabajo aumenta de forma exponencial.

Para ello, es necesario obtener todos los puntos de la nube de puntos del espacio de trabajo de 4 ciclos de movimiento que se encuentra a menos de 10 mm (precisión escogida en posición antes de realizar la orientación final) del origen. Para ello, debido a la cantidad de memoria que es necesaria para ello, ya que se necesita posición, orientación, y secuencia de movimientos de cada punto, se ha tenido que modificar el algoritmo de generación de la nube de puntos para que se almacenen menos resultados. Aun así, el algoritmo tardó varias horas en obtener los resultados, que se almacenaron en un archivo .mat y se trataron para eliminar los puntos que se encontraran a más de 10 mm de distancia del origen de coordenadas de la nube de puntos. Una vez obtenidos los datos que se buscan, se genera un archivo definitivo llamado *resultado_ori_final.mat*, que se carga al inicio del algoritmo de orientación final, llamado **ori_final.m**, y que se encuentra, al igual que la comprobación de colisiones, en muchos de los anexos, pero se concreta su presencia en el Anexo 1.

Los resultados almacenados en dicho archivo .mat presentan la forma representada en la Figura 6. Como puede apreciarse, la nube presenta una gran densidad de puntos en un entorno tan pequeño como lo es el de 10 mm alrededor del origen. Las zonas más conflictivas son las representadas mediante circunferencias rojas, pero estas no rondan 1 mm de radio, es decir, que a priori el error máximo que daría este algoritmo sería de alrededor de 1 mm.

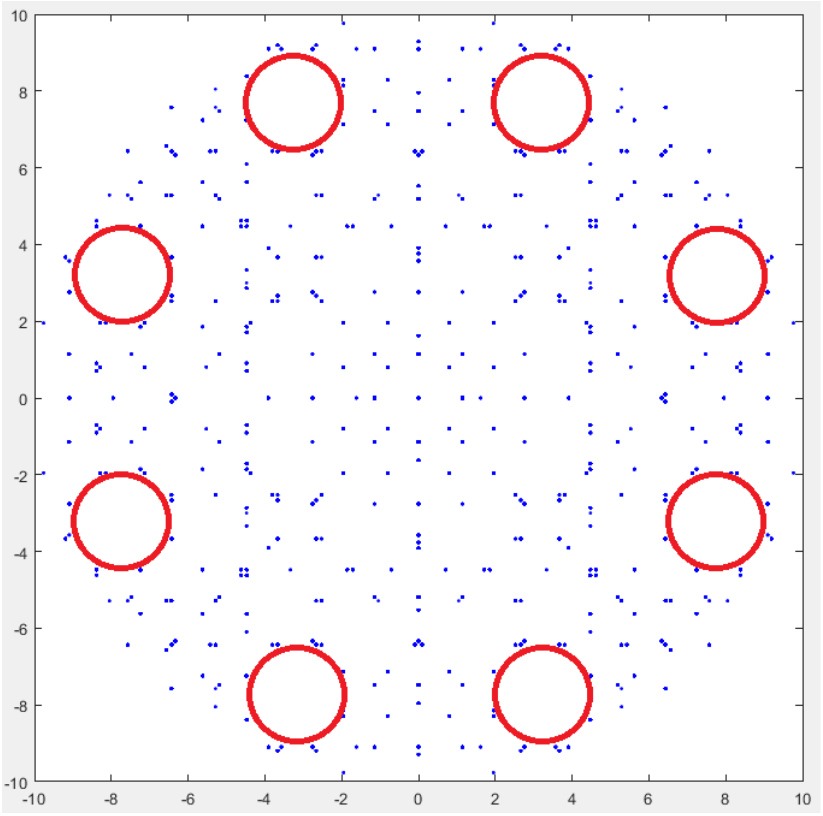

Figura 6: Espacio de trabajo de 4 ciclos a 10 mm del origen.

Una vez se obtiene la nube de puntos, el procedimiento de aplicación del algoritmo es similar al que se tomó en (*Pérez, 2021*). En primer lugar, se calcula la posición y orientación deseada en coordenadas locales del robot, y se buscan todos los puntos de la nube que tienen la orientación calculada. De todos esos puntos, se toma aquel cuya distancia al punto objetivo sea menor, así se obtiene el punto más cercano posible al objetivo con la orientación deseada, y de este se obtiene su secuencia de movimientos para saber qué configuraciones consecutivas debe adoptar el robot para alcanzar dicha pose. Como se ha mencionado anteriormente, el código completo de este algoritmo se encuentra reflejado en el Anexo 1.

2.3. Escape de mínimos locales mediante movimientos aleatorios

En este apartado se pretende explicar la mejora en el rendimiento de convergencia del algoritmo implementado en (*Pérez, 2021*). Para ello, se va a explicar el algoritmo completo, ya que se han realizado un número considerable de cambios al anterior a este. Cabe destacar que los aspectos que no hayan cambiado con respecto al algoritmo desarrollado en (*Pérez, 2021*) también se explicarán, pero es posible que no se entre en tanto detalle como se hizo en dicho trabajo. Las funciones y *scripts* que pertenecen a este

algoritmo corresponden con la totalidad del Anexo 1, y sus nombres son los siguientes: **ws_binario_mp.m**, **fr_2D_malla.m**, **area_local.m**, **calculo_area_adyac.m**, **inversa.m**, **directa.m**, **ciclo_completo.m**, **encontrar_punto.m**, **calculo_area.m**, **comprobacion_colisiones.m**, **codigo_kdtree.m**, **ori_final.m** y **dibujo.m**.

El *script* **ws_binario_mp.m** es donde se llaman al resto de funciones, por lo que primero se explicarán las tareas que desempeña cada una de las funciones y finalmente se profundizará en el *script* mencionado.

Las funciones referentes a la comprobación de colisiones y a la orientación final ya se han explicado anteriormente, por lo que no se volverá a hacer, solamente se mencionará el momento de su llamada en el *script* principal. Por ello, las funciones que quedan por explicar son **fr_2D_malla.m**, **ciclo_completo.m**, **encontrar_punto.m**, y **dibujo.m**.

La primera, **fr_2D_malla.m**, es una función que se encarga de obtener los puntos de la frontera de los obstáculos. Los obstáculos se generan mediante una nube de puntos en el espacio que el robot no puede ocupar, pero es muy ineficiente computacionalmente tomar las nubes completas, por lo que se ha desarrollado esta función, que se encarga de tomar solamente la frontera de estas con el fin de mejorar el rendimiento del resto del algoritmo. Esta función recibe como parámetros de entrada las resoluciones en X y en Y de la nube de puntos obstáculo y sus valores máximos y mínimos en X y en Y.

El primer paso es el cálculo de los parámetros necesarios para el algoritmo a partir de los parámetros de entrada, tales como las dimensiones (*Dx* y *Dy*), el número de filas y columnas (*g* y *h*) del mapa de ocupación, así como un vector que simplemente almacena las resoluciones en X y en Y (*v_paso*) y el radio del K-D tree que se va a emplear posteriormente (*radio*), definido como el mínimo de las resoluciones dividido entre 2.

```
13 -   for i=1:g
14 -     x = (i-1)*paso_x+xmin;
15 -     for j = 1:h
16 -       y = (j-1)*paso_y + ymin;
17 -       pto = [x y];
18 -       Idx = rangesearch(Md,pto,radio);
19 -       Retrieved = zeros(length(Idx{1}),2);
20 -       if isempty(Idx{1}) == 0
21 -         hay_obstaculo(i,j) = 1;
22 -       end
23 -     end
24 -   end
```

Con los datos definidos previamente, se define el bucle que se muestra en las líneas 13-24, que se encarga únicamente de rellenar un mapa de ocupación (*hay_obstaculo*) con un 1 en caso de que haya obstáculo en ese lugar o con un 0 en caso contrario a partir del K-D tree calculado en el *script* principal y tomado como variable global. Para ello, se recorren todos los puntos del mapa y se busca si el K-D tree tiene un punto obstáculo definido en el punto actual. Según el resultado, se define *hay_obstaculo* como corresponda según el criterio comentado anteriormente.

```
26 -  for b = 1:g
27 -    for c = 1:h
28 -
29 -      columna = b;
30 -      fila = c;
31 -      if hay_obstaculo(columna,fila) == 1
32 -        vecinos = zeros(8,2);
33 -        z = 1;
34 -        for i = -1:1
35 -          for j = -1:1
36 -            if (i~=0 || j~=0)
37 -              vecinos(z,1) = columna+i;
38 -              vecinos(z,2) = fila+j;
39 -              z = z+1;
40 -            end
41 -          end
42 -        end
43 -        for k = 1:8
44 -          e = vecinos(k,1);
45 -          d = vecinos(k,2);
46 -          if e>g || d>h || e<1 || d<1
47 -            continue
48 -          end
49 -          if hay_obstaculo(e,d) == 1
50 -            continue
51 -          else
52 -            p = [columna fila];
53 -            frontera = [frontera; [(p(1)-1)*paso_x+xmin,(p(2)-1)*paso_y+ymin]];
54 -          end
55 -        end
56 -      end
57 -    end
58 -  end
```

Finalmente, se recorre el mapa de ocupación completo (líneas 26-27) y se busca un punto donde haya obstáculo (línea 31), y de este se define un vector (*vecinos*) que almacena las posiciones en el mapa de los 8 vecinos del punto obstáculo seleccionado (líneas 32-42). Una vez encontrados los 8 vecinos del punto obstáculo, se busca en ellos si, en primer lugar, alguno de ellos se sale de los límites del mapa (líneas 46-48), y en

segundo lugar, en caso de que no se cumpla la condición anterior, se busca si todos los vecinos forman parte del obstáculo. En caso afirmativo, se descarta el punto, puesto que no pertenece a la frontera, y en caso contrario, como tienen al menos un vecino que no es obstáculo, se considera que sí pertenece a la frontera, por lo que se almacena la posición absoluta del punto en la variable *frontera* (líneas 49-54).

Con esto concluiría la explicación de la función **fr_2D_malla.m**, y ya se tendría definida la frontera del obstáculo. A continuación, sería turno de explicar la función **ciclo_completo.m**. Esta función ya se explicó detenidamente en (*Pérez, 2021*), por lo que no se profundizará tanto en ella. Su función principal es la de calcular el espacio de trabajo discreto del robot para un número N de ciclos de movimiento que definido previamente. Este cálculo se realiza a partir de las 8 configuraciones posibles que el robot puede adoptar, es decir, se buscan las $(8^2)^N$ posibilidades (el 2 de la potencia se debe a que cada ciclo conlleva 2 configuraciones del robot) y se almacenan en una matriz de posiciones y orientaciones. El código de esta función se encuentra reflejado en el Anexo 1.

Al igual que en el caso anterior, la función **encontrar_punto.m** también se desarrolló extensamente en (*Pérez, 2021*). Su única función es la de calcular, a partir de las distancias entre todos los puntos del espacio de trabajo calculado mediante la función anterior y el punto objetivo, el punto más cercano al objetivo. El código de esta función, al igual que para el caso anterior, se puede ver en el Anexo 1.

Finalmente, solamente queda explicar el *script* **dibujo.m**, cuya función es sencillamente la de dibujar el resultado obtenido y mostrarlo por pantalla.

```
 1 - plot(nube_grande(:,1),nube_grande(:,2),'.b');
 2 - set(gca,'DataAspectRatio',[1,1,1]);
 3 - hold on
 4 - axis([xmin xmax ymin ymax]);
 5 -
 6 - for d = 1:size(P_obs_p,1)
 7 -     P = P_obs_p(d,:);
 8 -     plot(P(1),P(2),'.k');
 9 - end
10 - for d = 1:size(P_obs,1)
11 -     P = P_obs(d,:);
12 -     plot(P(1),P(2),'.r');
13 - end
14 -
15 - plot(P_ref(1),P_ref(2),'or')
16 - plot(posicion_inicial(1),posicion_inicial(2),'.g','MarkerSize',25)
17 - plot(puntos_elegidos(:,1),puntos_elegidos(:,2),'*y')
18 - quiver(P_ref(1),P_ref(2),50*cos(phi_ref),50*sin(phi_ref),'r')
19 - plot(P_mas_cercano(1),P_mas_cercano(2),'*m')
20 - quiver(P_mas_cercano(1),P_mas_cercano(2),50*cos(nube(indice_min,3)),50*sin(nube(indice_min,3)),'m')
```

```
21 - plot(P_final(1),P_final(2),'sg')
22 - quiver(P_final(1),P_final(2),50*cos(phi_final),50*sin(phi_final),'g');
```

El resultado mostrado por pantalla es la nube de puntos de los espacios de trabajo solapados que el robot ha recorrido (línea 1), como se definió en (*Pérez, 2021*), las nubes de puntos obstáculo (líneas 6-9) y los puntos de la frontera (10-13), la posición y orientación objetivo (líneas 15 y 18), el punto inicial (línea 16), los puntos aleatorios generados (línea 17), y el resultado obtenido (líneas 19-20).

Con todas estas funciones auxiliares detalladas, ya se puede proceder a la explicación del algoritmo correspondiente a este apartado, reflejado en el Anexo 1. Dicho algoritmo se ha desarrollado en un *script* llamado **ws_binario_mp.m**, y su desarrollo es el siguiente:

```
 3 - global nube
 4 - global global_counter
 5 - global secuencia
 6 - global indice_min
 7 - global secuencia_encontrada
 8 - global P_mas_cercano
 9 - global ori_mas_cercano
10 - global Md
11 - global Md1
12 - global P_obs
13 -
14 - N = 2;
15 - P_ref = [600 500];
16 - phi_ref = pi/4;
17 - P0 = [0,0];
18 - posicion_inicial = P0;
19 - phi0 = 0;
```

Tras definir todas las variables globales necesarias (líneas 3-12), se definen ciertos parámetros que serán necesarios más tarde. Estas variables son: el número de ciclos de movimiento del robot por cada solape de espacios de trabajo (línea 14), la posición y orientación objetivo (líneas 15-16), y la posición y orientación iniciales (líneas 18-19).

```
21 - P_obs = [];
22 - P_obs_p = [];
23 - paso_x = 10;
24 - paso_y = 10;
25 - for xobs = 400-300:paso_x:400+300
26 -   for yobs = 300-50:paso_y:300+50
27 -     if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
28 -       P_obs_p = [P_obs_p;[xobs,yobs]];
29 -     end
30 -   end
31 - end
```

```
for xobs = 300-50:paso_x:300+50
  for yobs = 400-300:paso_y:400+300
    if (xobs-300)^2/50^2 + (yobs-400)^2/300^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end
```

El siguiente paso es la definición de las nubes de puntos obstáculo en el espacio de la tarea. En el caso que se va a mostrar como ejemplo, se han generado dos elipses de puntos, una vertical centrada en (300,400) y una horizontal centrada en (400,300), ambas con radio mayor 300 y radio menor 50. Dicho ejemplo se puede ver representado en la Figura 7, siendo el punto verde el inicio y el rojo el fin:

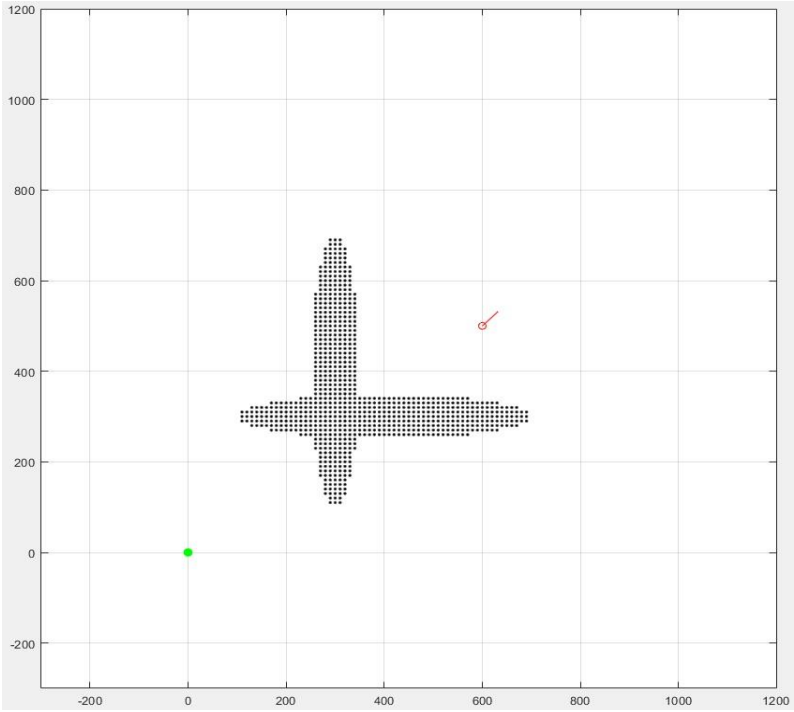

Figura 7: Caso a resolver en este trabajo mediante todos los algoritmos.

```
for i = 1:length(P_obs_p(:,1))
  if norm(P0-P_obs_p(i,:)) < 120.65
    disp('Error: el punto de inicio se encuentra muy cerca de un obstaculo.')
    pausa1 = 1;
    break
  else
    pausa1 = 0;
  end
  if norm(P_ref-P_obs_p(i,:)) < 120.65
    disp('Error: el punto final se encuentra muy cerca de un obstaculo.')
    pausa2 = 1;
    break
  else
    pausa2 = 0;
  end
```

```
56 - end
57 - if pausa1 == 1 || pausa2 == 1
58 -   pause
59 - end
```

Lo siguiente es la comprobación de que tanto el objetivo como el inicio no se encuentran obstruidos por obstáculos, y esto se hace simplemente comprobando que dichos puntos se encuentren a más de 120.65 mm (radio mínimo de la circunferencia que circunscribe al robot) de todos los puntos obstáculos (líneas 42-56). En caso afirmativo, se para el algoritmo (líneas 57-59), y en caso contrario se continúa.

```
61 - Md = KDTreeSearcher(P_obs_p,'Distance','minkowski','P',inf,'BucketSize',1);
62 -
63 - xmin = -300; xmax = 1200; ymin = -300; ymax = 1200;
64 - P_obs = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax);
```

A continuación, se calcula el K-D tree que se empleará para sacar la frontera de los obstáculos (línea 61), se definen los límites del mapa de ocupación (línea 63) y se obtiene la frontera de los obstáculos con la función correspondiente, explicada anteriormente.

```
67 - puntos_dest = [];
68 - puntos_elegidos = [];
69 - Rmax = 50;
70 - x_p=0;
71 - y_p=0;
72 -
73 - Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
74 -
75 - tamanyo = 0;
76 - for i=1:N
77 -   tamanyo = tamanyo + 64^i;
78 - end
79 -
80 - nube = zeros(tamanyo,3);
81 - global_counter = 0;
82 - secuencia = (-1)*ones(tamanyo,2*N);
83 - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
84 -
85 - registro_seq = [];
86 - puntos_encontrados = [];
87 - ori_encontradas = [];
88 - nube_grande = [];
89 -
90 - % Depende de: b, p, rho0, Delta_rho
91 - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
92 - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,21.95478428];
93 -
94 - for i=1:8
95 -  T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
```

```
96 - end
```

A partir de la frontera de los obstáculos, se debe calcular un nuevo K-D tree para trabajar sobre él en lugar de sobre el que almacena las nubes completas de puntos con el fin de mejorar el rendimiento del algoritmo (línea 73). Además, se definen ciertos parámetros que serán necesarios a continuación. Estos son: el radio máximo de generación de puntos aleatorios (*Rmax*, línea 69), el tamaño de la nube de puntos del espacio de trabajo definido por N ciclos (*tamanyo*, líneas 75-78), y una celda de matrices que almacena las 8 matrices de transformación posibles entre el eslabón A y B (*T*, líneas 91-96). Además de definición de parámetros, también se hacen reservas de memoria necesarias posteriormente, que corresponden a las líneas no mencionadas hasta ahora.

A continuación, se procede con la ejecución del algoritmo en sí, el cual consiste en un bucle *while* de alrededor de 180 líneas, por lo que se explicará por trozos.

```
 98 - f_aux = 1;
 99 - f_aux_2 = 1;
100 - ultimo_indice = 1;
101 -
102 - % Áreas que recorren los eslabones
103 - [areaA,areaB] = area_local(P0,phi0);
104 -
105 - while 1
106 -   if norm(P_ref-P0) < 10
107 -     break
108 -   end
109 -
110 -   if f_aux == 1
111 -     global_counter = 0;
112 -     ciclo_completo(P0,phi0,T,1,N);
113 -     f_aux = 0;
114 -     distancias = zeros(size(nube,1),1);
115 -     for i=1:size(nube,1)
116 -       distancias(i) = norm(P_ref-nube(i,1:2));
117 -     end
118 -   end
119 -
120 -   encontrar_punto(distancias)
121 -
122 -   pos_0 = P_mas_cercano;
123 -   ori_0 = ori_mas_cercano;
```

Este algoritmo requiere de la definición de dos variables auxiliares (líneas 98-99) que se usarán como señales o *flags*, es decir, serán binarias y servirán para comprobar que se ha cumplido alguna condición, así como de la definición del índice de la configuración del robot en el instante anterior, y cuyo valor inicial es 1 (línea 100). A

continuación, se realiza el cálculo del área que barre el robot en coordenadas locales (línea 103), tal y como se explicó previamente.

En la línea 105 se inicia el bucle que se encarga de calcular el algoritmo, y justo a continuación se define la condición de ruptura del bucle (líneas 106-113), la cual se cumple cuando la posición actual del robot se encuentra a menos de 10 mm del objetivo. La variable binaria *f_aux* define si ya se ha calculado la nube de puntos actual, por lo que si no se ha calculado previamente, se calcula y se cambia el valor de *f_aux* para que en caso de que el punto encontrado al que moverse provoque colisión, no se vuelva a calcular (líneas 110-118). A continuación, es necesario calcular las distancias entre todos los puntos de la nube y el objetivo (líneas 114-118), para con ellas encontrar el punto más cercano al objetivo (línea 120) y definirlo como la posición actual (líneas 122-123).

```
125 -   for h = 1:N
126 -
127 -     if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
128 -       break
129 -     end
130 -
131 -     indice_inicio = secuencia_encontrada(h);
132 -     indice_fin = secuencia_encontrada(h+N);
133 -
134 -     movB = 1;
135 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
136 -            sin(ori_0)  cos(ori_0) pos_0(2);
137 -                0           0         1  ];
138 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
139 -     [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,indice_inicio,T_p,T,area);
140 -     if colision == 1
141 -       break
142 -     end
143 -
144 -     movB = 0;
145 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
146 -            sin(ori_0)  cos(ori_0) pos_0(2);
147 -                0           0         1  ];
148 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
149 -     [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,indice_fin,T_p,T,area);
150 -     if colision == 1
151 -       break
152 -     end
153 -
154 -   end
```

En las líneas 125-154, se procede a realizar la comprobación de colisiones para la secuencia de movimientos que hacen al robot moverse hasta la posición calculada mediante la función **encontrar_punto.m**. Dicha secuencia se ha almacenado en la

variable *secuencia_encontrada*, y se ha calculado en la función mencionada, y esta almacena las configuraciones que el robot debe adoptar en cada medio ciclo. Para ello, se va a hacer este análisis por cada ciclo, es decir, que se van a tomar dos índices, uno para el primer medio ciclo (línea 131) y otro para el segundo medio ciclo (línea 132).

Para el primer medio ciclo, se define que el eslabón en movimiento es B (línea 134) y se calcula la matriz de transformación entre la posición actual del eslabón fijo (A) con respecto al sistema de coordenadas global (líneas 135-137). A continuación, se realiza la comprobación de colisiones tal y como se ha explicado anteriormente, obteniendo una salida binaria (*colision*) que es verdadera cuando se ha producido una colisión y falsa cuando no. A continuación, se comprueba si hay colisión (líneas 140-142) y se actúa en consecuencia: en caso afirmativo, se rompe el bucle con el fin de penalizar el punto encontrado y obtener otro nuevo, y en caso negativo, se procede a la comprobación del siguiente medio ciclo. El siguiente medio ciclo, es decir, el movimiento del eslabón A, se comprueba de una forma idéntica al anterior, pero tomando como eslabón fijo el eslabón B: se toma el eslabón A como móvil (línea 144), se calcula la matriz de transformación absoluta del eslabón B (líneas 145-147), se realiza la comprobación de colisiones (líneas 150-152) y se actúa en consecuencia de dicha comprobación (líneas 150-152). Como el movimiento hasta alcanzar el objetivo se hace de 2 en 2 ciclos, este bucle permite hacer la comprobación de colisiones de 4 medios ciclos consecutivos, por lo que su resultado influirá directamente en el resto del código.

```
156 -   if colision == 0
157 -     registro_seq = [registro_seq; secuencia_encontrada];
158 -     puntos_encontrados = [puntos_encontrados; P_mas_cercano];
159 -     ori_encontradas = [ori_encontradas; ori_mas_cercano];
160 -     P0 = P_mas_cercano;
161 -     phi0 = ori_mas_cercano;
162 -     f_aux = 1;
163 -     tam_p = length(puntos_encontrados(:,1));
```

```
284 -     nube_grande = [nube_grande;nube];
285 -   else
286 -     distancias(indice_min) = 1000000000;
287 -   end
```

Según el resultado anterior, se cumple lo que se muestra en el código. En caso de que no exista colisión (líneas 156-163 y 284) se actualiza la posición actual por la obtenida, se almacena la nube de puntos del espacio de trabajo (línea 284) y se pasa a la siguiente iteración para buscar otro punto, y en caso de que sí exista (líneas 285-287), se

penaliza el punto encontrado y se busca otro que no esté obstruido por un obstáculo. El algoritmo podría funcionar perfectamente solamente con estas líneas, pero solo para casos donde no existan mínimos locales, por lo que es necesario modificar el algoritmo para que sea capaz de evitar mínimos locales, es por eso que existe un fragmento de código entre las líneas 163-283 que se encarga de esto. Para ello, es necesario almacenar ciertos datos que serán necesarios para el algoritmo de mínimos locales (líneas 157-159).

Dicho algoritmo ya se implementó prácticamente de la misma forma en (*Pérez, 2021*), sin embargo, se volverá a explicar con detenimiento en este Trabajo Fin de Máster debido a que se han hecho algunos cambios con respecto al anterior. Siendo un algoritmo tan largo, se subdividirá en secciones de código para su explicación.

```
164 -     if tam_p > 4
165 -       X_v = puntos_encontrados(tam_p-4:tam_p,1);
166 -       Y_v = puntos_encontrados(tam_p-4:tam_p,2);
167 -       DX = std(X_v);
168 -       DY = std(Y_v);
169 -       if DX < 5 && DY < 5
170 -         Rmax = Rmax + 25;
171 -         tam = length(registro_seq(:,1));
172 -         registro_seq = registro_seq(1:tam-3,:);
173 -         puntos_encontrados = puntos_encontrados(1:tam-3,:);
174 -         ori_encontradas = ori_encontradas(1:tam-3,:);
175 -         P_mas_cercano = puntos_encontrados(tam-3,:);
176 -         ori_mas_cercano = ori_encontradas(tam-3,:);
177 -         P0 = P_mas_cercano;
178 -         phi0 = ori_mas_cercano;
179 -         repulsor = puntos_encontrados(tam-3,:);
```

Este algoritmo toma los últimos 4 puntos alcanzados por el robot (líneas 165-166) y calcula su desviación típica (líneas 167-168), para saber si el robot lleva 4 movimientos desplazándose poco (línea 169), lo que significaría que se ha quedado bloqueado en un mínimo local. Una vez detectado el mínimo local, se incrementa el valor del radio máximo de la generación de puntos aleatorios (línea 170) para que cada vez que no encuentre un punto que le permita escapar del mínimo local, intente buscar otro punto más lejano con el fin de que finalmente encuentre uno que sí le permita escapar del mínimo. A continuación, se eliminan los puntos encontrados que no han producido movimiento, es decir, los que han determinado el mínimo local (líneas 171-176), se actualizan la posición y orientación actual a la de antes de entrar al mínimo local (177-178), y se almacena el punto del mínimo local para su posterior visualización (línea 179).

```
181 -         % Generación de puntos aleatorios
182 -         puntos_destino = [];
```

```
183 -         for g = 1:7
184 -           while 1
185 -             R = Rmax*rand;
186 -             angulo_hacia_destino = atan2(P_ref(2)-P0(2),P_ref(1)-P0(1));
187 -             if rand > 0.5 % Virar hacia la derecha
188 -               angulo = angulo_hacia_destino - 3*pi/4 + pi/2*rand;
189 -             else % Virar hacia la izquierda
190 -                angulo = angulo_hacia_destino + 3*pi/4 - pi/2*rand;
191 -             end
192 -             x_p = P_mas_cercano(1) + R*cos(angulo);
193 -             y_p = P_mas_cercano(2) + R*sin(angulo);
194 -             P_ref_n = [x_p,y_p];
195 -             d_n = zeros(length(P_obs(:,1)),1);
196 -             for d = 1:length(P_obs(:,1))
197 -               d_n(d) = norm(P_ref_n-P_obs(d,:));
198 -             end
199 -             if min(d_n) > 125
200 -               puntos_destino = [puntos_destino;P_ref_n];
201 -               break
202 -             end
203 -           end
204 -         end
205 -
206 -         % Búsqueda del mejor punto generado
207 -         l = length(puntos_destino(:,1));
208 -         dist_repulsor = zeros(l,1);
209 -         dist_ref = zeros(l,1);
210 -         dist_aux = zeros(l,1);
211 -         for k = 1:l
212 -           dist_repulsor(k) = norm(repulsor-puntos_destino(k,:));
213 -           dist_ref(k) = norm(P_ref - puntos_destino(k,:));
214 -           dist_aux(k) = dist_ref(k) - 2*dist_repulsor(k);
215 -         end
216 -         [~,ind_def] = min(dist_aux);
217 -         P_ref_n = puntos_destino(ind_def,:);
218 -         puntos_elegidos = [puntos_elegidos;P_ref_n];
```

El siguiente paso es la generación de un punto aleatorio al que el robot deba moverse para retomar su trayectoria al objetivo con el objetivo de que este destino intermedio le sirva al robot para escapar del mínimo local. En (*Pérez, 2021*) se implementó este sistema generando un único punto y enviando al robot al mismo, pero su ineficiencia ha hecho que se decida cambiar por un sistema que genera 7 puntos (línea 183) y de ellos se tome el mejor posible. Esta implementación no hace que se encuentren puntos correctos al primer intento, pero requiere muchos menos intentos que en el que se desarrolló en (*Pérez, 2021*).

Para generar cada punto, se va a seguir el algoritmo implementado en las líneas 184-203. La idea es generar puntos hasta encontrar uno que no se encuentre obstruido por un obstáculo. Para ello, en primer lugar, se genera la distancia a la que estará el punto

intermedio aleatorio del mínimo local (línea 185), y a continuación se debe generar la dirección en la que se pondrá dicho punto. Dicha orientación se tenderá a generar a los lados del mínimo local con respecto al destino, tal y como se muestra en la Figura 8. Los puntos se generarán en las zonas amarillas de dicha figura, con el fin de que el robot intente rodear el obstáculo que se interpone en el camino hacia el objetivo definido por la línea morada. Destacar que en la Figura 8 los obstáculos son los círculos negros.

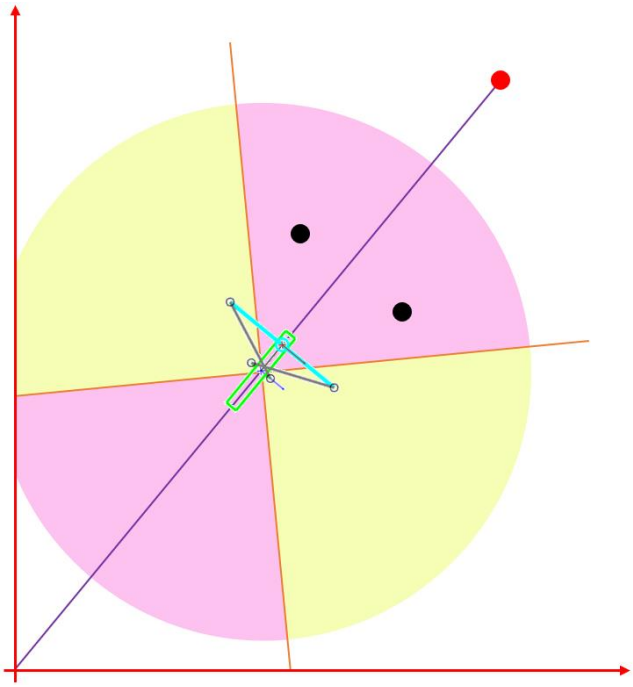

Figura 8: Distribución de la generación de destinos aleatorios.

Para ello, se calcula una orientación aleatoria dentro de las zonas amarillas de la Figura 8 a partir del código situado en las líneas 186-191. Una vez ya se tiene la distancia (*R*) y el ángulo (*angulo*), se calcula el destino aleatorio generado (líneas 192-194) y se comprueba si se encuentra a una distancia de 125 (radio de la circunferencia circunscrita al robot) o más de todos los puntos de la frontera del obstáculo (líneas 195-202). En caso de que dicho punto no se encuentre obstruido por un obstáculo, se toma como bueno y se almacena en el vector generado para ello (línea 200) y se rompe el bucle de la línea 184, y si el punto se encuentra obstruido, continúa ejecutando el bucle iniciado en la línea 184 hasta encontrar un punto no obstruido.

El siguiente paso, como se ha comentado anteriormente, consiste en escoger el mejor de los puntos generados en el bucle anterior (línea 183), para lo que se toman todos los puntos generados y se busca cuál es el que se encuentra a la vez más cerca del objetivo

y más lejos del mínimo local (líneas 207-215). Finalmente, se escoge el mejor y se toma como objetivo intermedio antes de alcanzar el destino (líneas 216-218).

```
220 -       while 1
221 -         if norm(P_ref_n-P0) < 10
222 -           break
223 -         end
224 -         if f_aux_2 == 1
225 -           global_counter = 0;
226 -           ciclo_completo(P0,phi0,T,1,N)
227 -           f_aux_2 = 0;
228 -           distancias_n = zeros(size(nube,1),1);
229 -           for i=1:size(nube,1)
230 -             distancias_n(i) = norm(P_ref_n-nube(i,1:2));
231 -           end
232 -         end
233 -
234 -         encontrar_punto(distancias_n)
235 -
236 -         pos_0 = P_mas_cercano;
237 -         ori_0 = ori_mas_cercano;
238 -
239 -         for h = 1:N
240 -
241 -           if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
242 -             break
243 -           end
244 -
245 -           indice_inicio = secuencia_encontrada(h);
246 -           indice_fin = secuencia_encontrada(h+N);
247 -
248 -           movB = 1;
249 -           T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
250 -                  sin(ori_0)  cos(ori_0) pos_0(2);
251 -                      0           0         1  ];
252 -           area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
253 -           [colision,pos_0,ori_0]=comprobacion_colisiones(movB,pos_0,ori_0,indice_inicio,T_p,T,area);
254 -           if colision == 1
255 -             break
256 -           end
257 -
258 -           movB = 0;
259 -           T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
260 -                  sin(ori_0)  cos(ori_0) pos_0(2);
261 -                       0           0         1  ];
262 -           area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
263 -           [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,indice_fin,T_p,T,area);
264 -           if colision == 1
265 -             break
266 -           end
267 -
268 -         end
269 -
```

```
270 -           if colision == 0
271 -             registro_seq = [registro_seq; secuencia_encontrada];
272 -             puntos_encontrados = [puntos_encontrados; P_mas_cercano];
273 -             ori_encontradas = [ori_encontradas; ori_mas_cercano];
274 -             P0 = P_mas_cercano;
275 -             phi0 = ori_mas_cercano;
276 -             f_aux_2 = 1;
277 -             nube_grande = [nube_grande;nube];
278 -           else
279 -             distancias_n(indice_min) = 1000000000;
280 -           end
281 -         end
```

El siguiente segmento del algoritmo es bastante extenso, pero también bastante similar a lo que se ha explicado hasta ahora, por lo que se expondrá de manera más rápida. Este segmento consiste en alcanzar el nuevo objetivo intermedio, por lo que sus primeras líneas (221-223) consisten en la comprobación de que el robot ha alcanzado el destino intermedio, se calcula la nube de puntos de N ciclos (2 en todos los casos expuestos) y se busca el mejor punto de todos los calculados (líneas 224-237) y se realiza la comprobación de colisiones exactamente igual que para el algoritmo general (líneas 239-268); en caso de que no haya colisión, se toma el punto encontrado como bueno, almacenando los datos necesarios (líneas 270-277), y en caso contrario, se penaliza el punto encontrado y se busca otro (líneas 278-280).

```
282 -         end
283 -       end
284 -       nube_grande = [nube_grande;nube];
285 -     else
286 -       distancias(indice_min) = 1000000000;
287 -     end
288 -
289 - end
290 -
291 - ori_final;
292 - dibujo;
```

Finalmente, se almacenan las nubes de puntos del espacio de trabajo que se ha calculado en cada instante (línea 284), y se aplican tanto el algoritmo de orientación final (línea 291) como el algoritmo de visualización de los resultados (línea 292), ambos explicados anteriormente.

Con esto, se ha concluido la explicación del algoritmo de generación aleatoria de puntos para escapar de mínimos locales. Lo siguiente que se expondrá serán los resultados del algoritmo obtenidos para el ejemplo expuesto en la Figura 7. Como se ha comentado, este algoritmo genera puntos aleatorios, por lo que cada vez que se ejecute el resultado

puede ser distinto, por lo que se van a mostrar solamente dos resultados, los más distintos posibles, ya que el resto son variantes similares de los que se van a exponer en la Figura 9.

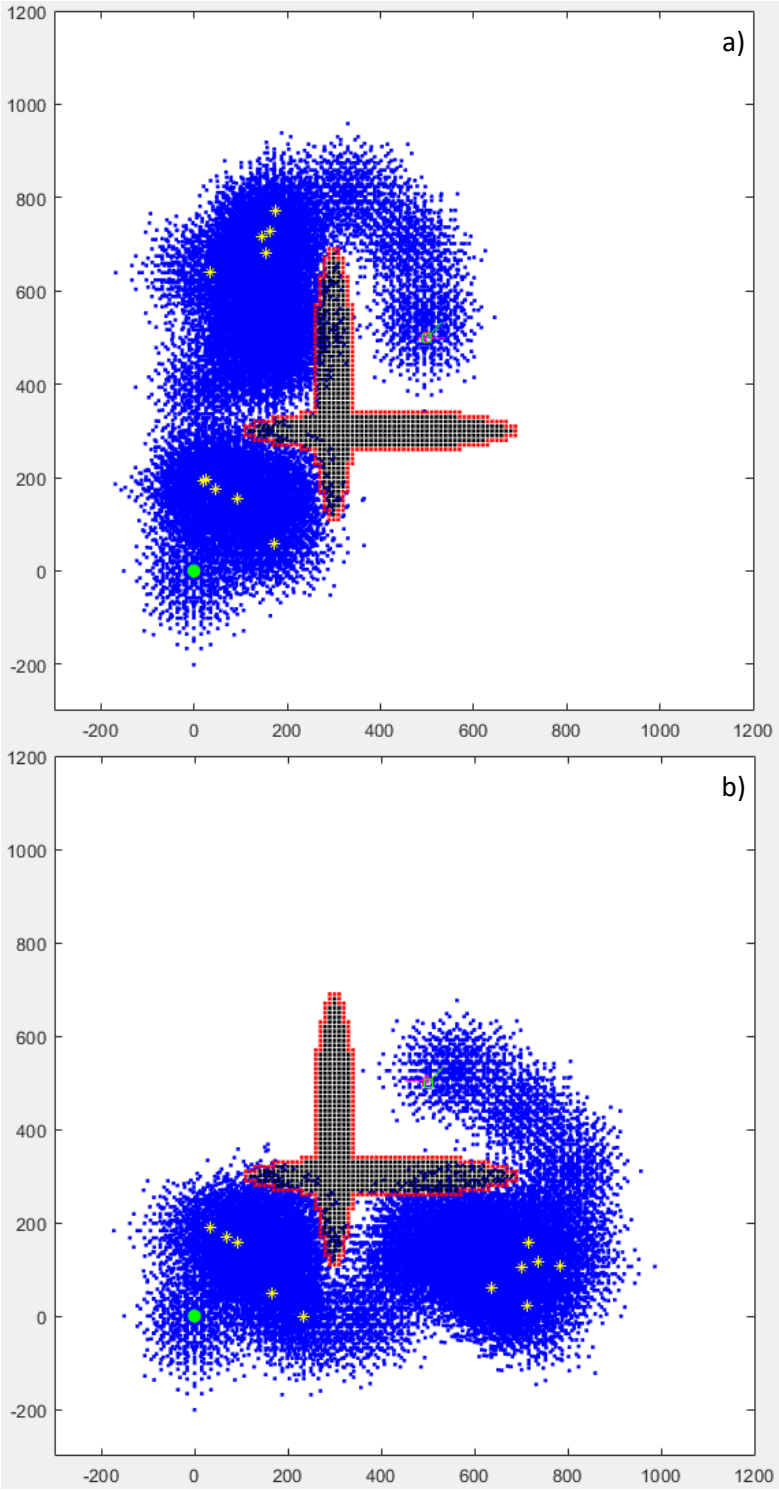

Figura 9: Resultado para el algoritmo de generación de puntos aleatorios. a) rodeando por la izquierda. b) rodeando por la derecha.

Los resultados representados en las Figura 9 tienen el siguiente código de colores: los puntos negros representan la nube de puntos obstáculo, y los rojos su frontera; la nube azul de puntos representa el solapamiento de los espacios de trabajo calculados; el punto verde corresponde con el punto inicial, y el cuadrado y flecha verdes representan el resultado (en posición y orientación) obtenido; los asteriscos amarillos representan los destinos intermedios que el robot ha tomado en la generación aleatoria; finalmente, la circunferencia y flecha rojas representan la posición y orientación objetivo, pero no se pueden apreciar bien debido a que el resultado se ha dibujado encima, y el asterisco y flecha magentas representan el resultado obtenido antes de realizar la orientación final. Como puede apreciarse en las Figura 9, los resultados obtenidos son los más lógicos que se pueden pensar: rodear el obstáculo por la derecha o hacerlo por la izquierda. Finalmente, solo queda dar valores numéricos a los resultados, y estos serán los valores de los errores de posición y el número de ciclos de movimiento que el robot ha realizado en cada caso. Dichos valores son 0.2716 mm de error en posición y 106 ciclos de movimiento (102 para posicionarse y 4 para orientarse) para el caso de la Figura 9.a y 0.3702 mm de error en posición y 118 ciclos de movimiento (114 para posicionarse y 4 para orientarse) para el caso de la Figura 9.b.

2.4. Escape de mínimos locales mediante wall-following.

En este apartado se pretende modificar el algoritmo de evasión de mínimos locales a través de generación aleatoria de destinos intermedios. Concretamente, este pretende eliminar la aleatoriedad que hace que el algoritmo mencionado sea tan poco predecible y tan redundante en ciertas ocasiones. La idea principal es desarrollar el algoritmo de tal forma que haga al robot capaz de rodear los obstáculos que se encuentre tomando como referencia el propio obstáculo, es decir, que detecte la forma que tiene el obstáculo y lo rodee haciendo un “seguimiento de la pared”.

Para ello, se deben aplicar los siguientes scripts y funciones, todos ellos presentes en el Anexo 2: **ws_binario_mp.m**, **fr_2D_malla.m**, **area_local.m**, **calculo_area_adyac.m**, **inversa.m**, **directa.m**, **ciclo_completo.m**, **encontrar_punto.m**, **calculo_area.m**, **comprobacion_colisiones.m**, **codigo_kdtree.m**, **LineFitting.m**, **ori_final.m**, **dibujo.m**. De todos estos, los únicos que cambian con respecto a los desarrollados en la sección 2.2 y reflejados en el Anexo 1 son **ws_binario_mp.m** y **LineFitting.m**, este último no estando en

el caso mencionado. Es por ello que únicamente se explicarán dichos códigos y se referenciará al resto cuando sean llamados.

Primero se explicará la función **LineFitting.m**, ya que esta será llamada en **ws_binario_mp.m** y para ese momento es necesario comprender su funcionamiento. Esta función se encarga de la aproximación lineal de puntos mediante RANSAC, un algoritmo iterativo de ajuste lineal, similar al ajuste por mínimos cuadrados, cuyo objetivo no es encontrar la recta que se encuentre a la mínima distancia posible de todos los puntos, sino la recta que tenga la mayor cantidad de puntos cerca de ella. Un ejemplo de comparación de estos algoritmos es el que se muestra en la Figura 10.

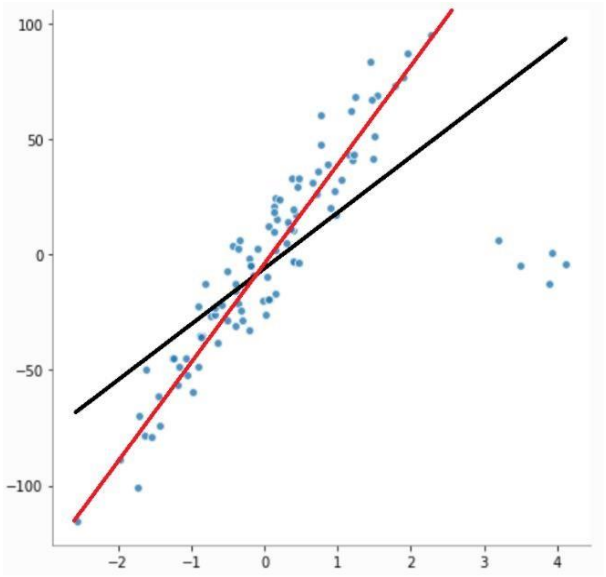

Figura 10: Comparación entre mínimos cuadrados y RANSAC.

La recta que se busca (generada por RANSAC) es la roja, la cual nos permite aproximar los puntos que se le asignen a una recta lo más precisamente posible, aunque se deje puntos fuera de ella. Estos puntos simplemente se consideran como otro obstáculo diferente o como otra pared del mismo obstáculo, pero lo que se necesita es solo una pared del mismo para rodearla. Al igual que en el apartado anterior, el obstáculo será tratado únicamente tomando su frontera, por lo que la recta que aproxime los puntos de dicha frontera será bastante precisa en su aproximación, ya que no se toman los puntos internos del obstáculo.

Esta función recibe como parámetros de entrada los puntos de la frontera del obstáculo obtenidos mediante el cálculo del K-D tree de una ventana determinada centrada en el robot, y su salida son los puntos que se han encontrado como que mejor realizan el ajuste lineal de los puntos.

```
 3 -   tam = length(Retrieved(:,1));
 4 -   iteraciones = 25;
 5 -   contador_prev = 0;
 6 -
 7 -   for i = 1:iteraciones
 8 -     i1 = randi(tam);
 9 -     i2 = randi(tam);
10 -     p1 = Retrieved(i1,:);
11 -     p2 = Retrieved(i2,:);
12 -     v = p2-p1;
13 -     A = v(2)/v(1);
14 -     B = -1;
15 -     C = -B*p1(2)-A*p1(1);
16 -     contador = 0;
17 -     for j = 1:tam
18 -       if Retrieved(j,1) == 0 && Retrieved(j,2) == 0
19 -         continue
20 -       end
21 -       p = Retrieved(j,:);
22 -       d = abs((A*p(1)+B*p(2)+C)/sqrt(A^2+B^2));
23 -       if d < 5 %umbral
24 -         contador = contador+1;
25 -       end
26 -     end
27 -     if contador > contador_prev
28 -       pf1 = p1;
29 -       pf2 = p2;
30 -       contador_prev = contador;
31 -     end
32 -   end
```

Este algoritmo realiza un número de iteraciones determinado (línea 4) en las que toma dos puntos aleatorios de la entrada (líneas 8-11), y calcula la distancia del resto de puntos a la recta formada por dichos 2 puntos. Para ello, se emplea la siguiente ecuación, la cual calcula la distancia entre un punto y una recta a partir de la ecuación general de la misma:

$$D = \frac{|A \cdot x_p + B \cdot y_p + C|}{\sqrt{A^2 + B^2}}$$

Para ello, se calculan los parámetros de la ecuación general de la recta (líneas 12-15), se calcula la distancia entre el resto de los puntos y la recta (líneas 17-22) y se cuenta cuántos de esos puntos se encuentran dentro del umbral establecido como distancia válida de aproximación RANSAC (líneas 23-25). Finalmente, se toman como salida los puntos que han generado un mayor número en el contador de puntos cercanos a la recta (líneas 27-31).

Una vez explicada esta función, lo único que resta de este apartado es la explicación del algoritmo en sí, desarrollado en el *script* **ws_binario_mp.m**. A pesar de

que este algoritmo es bastante similar al anterior hasta el punto de aplicar la nueva evasión de obstáculos, se explicará el código completo, aunque sea con menos detalle que en la sección anterior.

```
  3 - global nube
  4 - global global_counter
  5 - global secuencia
  6 - global indice_min
  7 - global secuencia_encontrada
  8 - global P_mas_cercano
  9 - global ori_mas_cercano
 10 - global Md
 11 - global Md1
 12 - global P_obs
 13 -
 14 - N = 2;
 15 - P_ref = [500 500];
 16 - phi_ref = pi/4;
 17 - P0 = [0,0];
 18 - posicion_inicial = P0;
 19 - phi0 = 0;
 20 -
 21 - P_obs = [];
 22 - P_obs_p = [];
 23 - paso_x = 10;
 24 - paso_y = 10;
 25 - for xobs = 400-300:paso_x:400+300
 26 -   for yobs = 300-50:paso_y:300+50
 27 -     if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
 28 -       P_obs_p = [P_obs_p;[xobs,yobs]];
 29 -     end
 30 -   end
 31 - end
 32 - for xobs = 300-50:paso_x:300+50
 33 -   for yobs = 400-300:paso_y:400+300
 34 -     if (xobs-300)^2/50^2 + (yobs-400)^2/300^2 < 1
 35 -       P_obs_p = [P_obs_p;[xobs,yobs]];
 36 -     end
 37 -   end
 38 - end
 39 -
 40 -
 41 - % Comprobación de si el punto inicial o final se encuentran obstaculizados
 42 - for i = 1:length(P_obs_p(:,1))
 43 -   if norm(P0-P_obs_p(i,:)) < 120.65
 44 -     disp('Error: el punto de inicio se encuentra muy cerca de un obstaculo.')
 45 -     pausa = 1;
 46 -     break
 47 -   else
 48 -     pausa = 0;
 49 -   end
 50 -   if norm(P_ref-P_obs_p(i,:)) < 120.65
 51 -     disp('Error: el punto final se encuentra muy cerca de un obstaculo.')
 52 -     pausa = 1;
 53 -     break
 54 -   else
 55 -     pausa = 0;
 56 -   end
 57 - end
```

```
58 - if pausa == 1
59 -   pause
60 - end
```

Esta parte del código es idéntica al algoritmo de la sección 2.2. Se definen las variables globales (líneas 3-12), las condiciones iniciales y finales (líneas 14-19), y la nube de puntos obstáculo (líneas 21-38), y se ha comprobado que los puntos inicial y final no se encuentran obstruidos por obstáculos (líneas 41-60).

```
62 - Md = KDTreeSearcher(P_obs_p,'Distance','minkowski','P',inf,'BucketSize',1);
63 - 
64 - xmin = -300; xmax = 1200; ymin = -300; ymax = 1200;
65 - P_obs = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax);
66 - 
67 - 
68 - puntos_dest = [];
69 - puntos_elegidos = [];
70 - x_p=0;
71 - y_p=0;
72 - 
73 - Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
74 - 
75 - tamanyo = 0;
76 - for i=1:N
77 -   tamanyo = tamanyo + 64^i;
78 - end
79 - 
80 - nube = zeros(tamanyo,3);
81 - global_counter = 0;
82 - secuencia = (-1)*ones(tamanyo,2*N);
83 - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
84 - 
85 - registro_seq = [];
86 - puntos_encontrados = posicion_inicial;
87 - P_ref_n = posicion_inicial;
88 - ori_encontradas = [];
89 - nube_grande = [];
90 - repulsores = [];
91 - puntos_anteriores = [];
92 - pes = [];
93 - qus = [];
94 - 
95 - % Depende de: b, p, rho0, Delta_rho
96 - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
97 - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,21.95478428];
98 - 
99 - for i=1:8
100 -   T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
101 - end
102 - 
103 - f_aux = 1;
104 - f_aux_2 = 1;
105 - ultimo_indice = 1;
106 - 
107 - % Áreas que recorren los eslabones
108 - [areaA,areaB] = area_local(P0,phi0);
```

Al igual que en el caso anterior, lo siguiente que se debe hacer es calcular el K-D tree del espacio con los obstáculos como nubes de puntos (línea 62), con él calcular la frontera del obstáculo (líneas 64-65), calcular un nuevo K-D tree con el que operar posteriormente (línea 73), y reservar memoria e inicializar variables que se usarán posteriormente (líneas 67-71 y 75-105). Finalmente, se calculan las áreas barridas por los eslabones en cada medio ciclo en coordenadas locales (línea 108). Cabe destacar que en las líneas 92 y 93 se inicializan las variables *pes* y *qus*, que son variables de control que se encargan de almacenar los puntos encontrados por RANSAC con el único fin de poder visualizarlos, no tienen un uso directo en el algoritmo.

```
110 - while 1
111 -   if norm(P_ref-P0) < 10
112 -     break
113 -   end
114 -
115 -   if f_aux == 1
116 -     global_counter = 0;
117 -     ciclo_completo(P0,phi0,T,1,N);
118 -     f_aux = 0;
119 -     distancias = zeros(size(nube,1),1);
120 -     for i=1:size(nube,1)
121 -       distancias(i) = norm(P_ref-nube(i,1:2));
122 -     end
123 -   end
124 -
125 -   encontrar_punto(distancias)
126 -
127 -   pos_0 = P_mas_cercano;
128 -   ori_0 = ori_mas_cercano;
129 -
130 -   for h = 1:N
131 -
132 -     if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
133 -       break
134 -     end
135 -
136 -     indice_ini = secuencia_encontrada(h);
137 -     indice_fin = secuencia_encontrada(h+N);
138 -
139 -     movB = 1;
140 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
141 -            sin(ori_0)  cos(ori_0) pos_0(2);
142 -                0           0         1  ];
143 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
144 -     [colision,pos_0,ori_0]=comprobacion_colisiones(movB,pos_0,ori_0,indice_ini,T_p,T,area);
145 -     if colision == 1
146 -       break
147 -     end
148 -
149 -     movB = 0;
150 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
151 -            sin(ori_0)  cos(ori_0) pos_0(2);
152 -                0           0         1  ];
153 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
```

```
154 -     [colision,pos_0,ori_0]=comprobacion_colisiones(movB,pos_0,ori_0,indice_fin,T_p,T,area);
155 -     if colision == 1
156 -       break
157 -     end
158 -
159 -   end
160 -
161 -   if colision == 0
162 -     registro_seq = [registro_seq; secuencia_encontrada];
163 -     puntos_encontrados = [puntos_encontrados; P_mas_cercano];
164 -     ori_encontradas = [ori_encontradas; ori_mas_cercano];
165 -     P0 = P_mas_cercano;
166 -     phi0 = ori_mas_cercano;
167 -     f_aux = 1;
168 -     tam_p = length(puntos_encontrados(:,1));
```

```
376 -     nube_grande = [nube_grande;nube];
377 -   else
378 -     distancias(indice_min) = 1000000000;
379 -   end
380 -
381 - end
```

El siguiente paso es el inicio del algoritmo en sí. La primera parte, mostrada en el fragmento de código anterior, corresponde con la parte del movimiento en la que no intervienen los mínimos locales, por lo que su desarrollo es idéntico al de la sección 2.2. El procedimiento es el siguiente: se implementa la condición de ruptura del bucle (líneas 111-113), se calcula la nube de puntos del espacio de trabajo del que se quiere buscar el punto donde mover al robot (líneas 115-123), se busca cuál de todos los puntos calculados es el más cercano al objetivo (líneas 125-128) y se realiza la comprobación de colisiones para el número de ciclos definido (en este caso 2) a partir del método de comprobación de colisiones implementado en la sección anterior (líneas 130-159). Una vez se tiene el resultado de la comprobación de colisiones, se actúa en consecuencia: si no hay colisiones (líneas 161-168 y 376) se almacenan los datos calculados como el estado actual de posición y orientación, se almacena la nube de puntos del espacio de trabajo calculado (línea 376) y se pasa a la siguiente iteración, y en caso contrario (líneas 377-379), se penaliza el punto encontrado por estar obstruido (línea 378) y se busca otro que no lo esté.

Hasta aquí es prácticamente idéntico al algoritmo del caso anterior. Como es lógico, si lo que se ha modificado es el algoritmo de escape de mínimos locales, lo que se ha modificado es dicha parte, que es la comprendida entre las líneas 169-375.

```
169 -     if tam_p > 4
170 -       X_v = puntos_encontrados(tam_p-3:tam_p,1);
171 -       Y_v = puntos_encontrados(tam_p-3:tam_p,2);
172 -       DX = std(X_v);
173 -       DY = std(Y_v);
174 -       if DX < 5 && DY < 5
```

```
175 -         flag_destino = 0;
176 -         registro_seq = registro_seq(1:tam_p-3,:);
177 -         puntos_encontrados = puntos_encontrados(1:tam_p-3,:);
178 -         ori_encontradas = ori_encontradas(1:tam_p-3,:);
179 -         P_mas_cercano = puntos_encontrados(tam_p-3,:);
180 -         ori_mas_cercano = ori_encontradas(tam_p-3,:);
181 -         repulsor = puntos_encontrados(tam_p-3,:);
182 -         repulsores = [repulsores;repulsor];
183 -         X_t = [repulsor(1) P_ref_n(1)];
184 -         Y_t = [repulsor(2) P_ref_n(2)];
185 -         D_Xt = std(X_t);
186 -         D_Yt = std(Y_t);
187 -         if D_Xt > 20 && D_Yt > 20
188 -           punto_anterior = P_ref_n;
189 -           puntos_anteriores = [puntos_anteriores;punto_anterior];
190 -         end
```

Para detectar si el robot se ha bloqueado en un mínimo local, se toman los últimos 4 puntos que este ha recorrido (líneas 170-171), se calcula su desviación típica (líneas 172-173) y se comprueba si esta es pequeña (línea 174). En caso afirmativo, se eliminarían los puntos redundantes (líneas 176-180) y se almacenaría el punto donde existe el mínimo local (líneas 181-182). Las líneas 183-190 se encargan de definir si el destino intermedio anteriormente encontrado (por defecto, el origen del movimiento) está lo suficientemente lejos del mínimo local encontrado; esto se utiliza más adelante en el algoritmo, por lo que la razón de su definición se explicará entonces.

```
191 -         while flag_destino == 0
192 -           P0 = P_mas_cercano;
193 -           phi0 = ori_mas_cercano;
194 -           Idx = rangesearch(Md1,repulsor,140);
195 -           Retrieved = zeros(length(Idx{1}),2);
196 -           for i = 1:length(Idx{1})
197 -             Retrieved(i,:) = P_obs(Idx{1}(i),:);
198 -           end
199 -           if isempty(Retrieved)
200 -             continue
201 -           else
202 -             retrieved_points = [];
203 -             for i = 1:length(Retrieved(:,1))
204 -               if norm(Retrieved(i,:)-repulsor) <= 140
205 -                 retrieved_points = [retrieved_points;Retrieved(i,:)];
206 -               end
207 -             end
208 -           end
209 -
210 -           [p,q] = LineFitting(retrieved_points);
```

El bucle iniciado en la línea 191 pretende buscar el siguiente punto de destino intermedio para escapar del mínimo local en el que se encuentra el robot. La idea es, en líneas generales, desplazar al robot en línea recta siguiendo la dirección de la pared del obstáculo hasta encontrar un punto que no se encuentre obstruido por el obstáculo. A

continuación, se añadirá un obstáculo virtual que hará que el robot no pueda pasar por donde ya ha pasado.

Para ello, en primer lugar, se debe encontrar la dirección en la que mandar al robot a rodear el obstáculo, y esta se define tomando los puntos frontera del obstáculo cercanos al robot (líneas 195-207) y con ellos se calculan los dos puntos que mejor ajustan dicha frontera a una línea mediante RANSAC (línea 210).

```
211 -          pes = [pes;p];
212 -          qus = [qus;q];
213 -
214 -          u = p-q;
215 -          distancia_maxima = 0;
216 -          A = u(2)/u(1);
217 -          B = -1;
218 -          C = -B*p(2)-A*p(1);
219 -          for i = 1:length(Retrieved(:,1))
220 -            r = Retrieved(i,:);
221 -            if r(1) == 0 && r(2) == 0
222 -              continue
223 -            else
224 -              d = abs((A*r(1)+B*r(2)+C)/sqrt(A^2+B^2));
225 -              if d > distancia_maxima
226 -                distancia_maxima = d;
227 -                punto_mas_lejano = r;
228 -              end
229 -            end
230 -          end
231 -          if distancia_maxima > 20
232 -            w = repulsor - punto_mas_lejano;
233 -          else
234 -            w = repulsor - punto_anterior;
235 -            line = 1;
236 -          end
237 -
238 -          v = u*(u(1)*w(1)+u(2)*w(2))/norm(u)^2;
239 -          v_unitario = v/norm(v);
```

Una vez se tienen los puntos que aproximan la frontera del obstáculo, se almacenan (líneas 211-212), y se calcula el vector director de la recta que los une (línea 214). Ahora bien, el vector director de la recta tiene una dirección, que es la que se busca, y un sentido, que no se sabe si es el idóneo, por ello, se debe escoger en función de la situación en la que se encuentre el robot. Para ello, se deben distinguir dos situaciones de mínimos locales posibles, que se representan en la Figura 11.

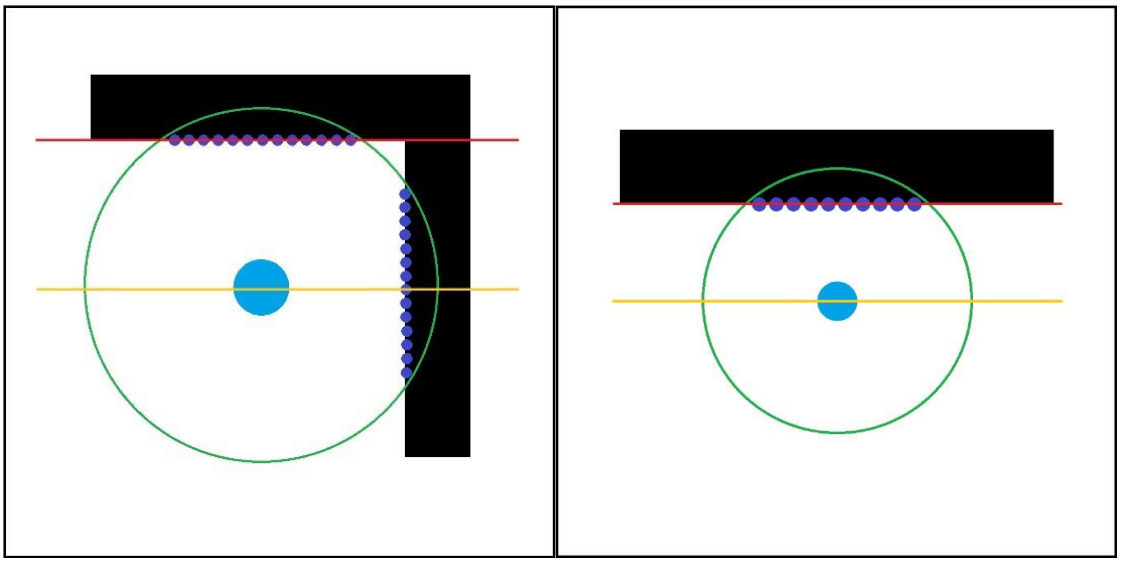

Figura 11: a) Mínimo local provocado por esquina. b) Mínimo local provocado por obstáculo recto.

Para el caso en el que el mínimo local está generado por una esquina, el algoritmo se comporta como se muestra en la Figura 11.a. Mediante el KD-tree se define una circunferencia (verde) que encuentra los puntos de la frontera más cercanos al robot (puntos morados). Con esos puntos, el algoritmo de ajuste lineal genera la recta de dirección de movimiento (línea roja) y, encontrada la dirección, se le debe mandar al robot moverse en dirección paralela a esa línea (dirección de la línea amarilla).

Para el caso en que el mínimo local está generado por un obstáculo con una única pared (en este caso el robot quiere alcanzar el otro lado del obstáculo, Figura 11.b), la recta que ajusta la frontera del obstáculo (línea roja) es mucho más intuitiva de apreciar, ya que prácticamente todos los puntos se encontrarán en la misma dirección, por lo que será similar, si no igual, al ajuste por mínimos cuadrados. Sin embargo, se debe implementar el algoritmo para un caso cualquiera, por lo que se utiliza RANSAC como criterio de ajuste lineal.

Como se ha comentado anteriormente, una vez definido el vector director de la recta que indica la dirección de movimiento del robot, se debe escoger el sentido dependiendo de cuál de las dos situaciones mostradas en la Figura 11 sea en la que se encuentra el robot. Para detectar la situación del robot, en primer lugar, se calcula la distancia máxima a la que se encuentra la recta de todos los puntos del obstáculo obtenidos mediante el K-D tree (variable *Retrieved*) mediante la fórmula de la distancia punto-recta a partir de la ecuación general (líneas 215-230). Una vez obtenida esa distancia máxima, se aplica el siguiente criterio para saber cuál de los dos casos es: si la

distancia es superior a un umbral determinado (línea 225), se considera que el obstáculo tiene dos paredes, o existen dos obstáculos diferentes, es decir, se considera que el robot se encuentra en el caso de la esquina (Figura 11.a), y en caso contrario, se considera que el obstáculo solamente tiene una pared (Figura 11.b).

Una vez identificado en qué caso se encuentra el robot, se define el vector *w* acorde con el siguiente criterio: si el obstáculo tiene dos paredes, el robot deberá moverse alejándose de la pared que no ha detectado el algoritmo de **LineFitting.m**, es decir, alejándose del punto de máxima distancia (línea 232); en cambio, si este se encuentra en el caso en el que solamente tiene una pared, el robot simplemente tenderá a alejarse de la dirección en la que venía, es decir, se alejará de la última posición en la que ha estado (línea 234), con el fin de evitar "volver atrás". El vector *w*, por regla general, nunca va a ser paralelo a la dirección del movimiento (definida por el vector director *u*), por lo que, para obtener correctamente la dirección y sentido del movimiento que debe tomar el robot, se debe realizar una operación más, la cual es obtener la proyección de *w* sobre el vector director de la recta *u* (líneas 238-239). Así, se tendría un vector en la dirección de *u*, pero con el signo que le corresponde a *w*.

```
241 -          R = 100;
242 -          deltaR = 25;
243 -          while 1
244 -            punto_destino = repulsor + v_unitario*R;
245 -            Idx = rangesearch(Md1,punto_destino,200);
246 -            Retrieved = zeros(length(Idx{1}),2);
247 -            for i = 1:length(Idx{1})
248 -              Retrieved(i,:) = P_obs(Idx{1}(i),:);
249 -            end
250 -            if isempty(Retrieved) == 0
251 -              vacia = 1;
252 -              for i = 1:length(Retrieved(:,1))
253 -                if Retrieved(i,1) ~= 0 || Retrieved(i,2) ~= 0
254 -                  vacia = 0;
255 -                  break
256 -                else
257 -                  vacia = 1;
258 -                end
259 -              end
260 -            else
261 -              vacia = 1;
262 -            end
263 -            if vacia == 0
264 -              for i = 1:length(Retrieved(:,1))
265 -                if Retrieved(i,1) ~= 0 || Retrieved(i,2) ~= 0
266 -                  if norm(Retrieved(i,:)-punto_destino) <= 200
267 -                    f_col = 1;
268 -                    break
269 -                  else
270 -                    f_col = 0;
271 -                  end
```

```
272 -                 end
273 -               end
274 -             else
275 -               f_col = 0;
276 -             end
277 -             if f_col == 0
278 -               break
279 -             else
280 -               R = R+deltaR;
281 -             end
282 -           end
```

El bucle mostrado en las líneas 243-282 pretende buscar en línea recta el punto al que el robot debe moverse para evitar el obstáculo. Su objetivo es generar puntos en dicha dirección hasta encontrar uno que no se encuentre cerca de un obstáculo. Para ello, se definen las distancias inicial e incremental que se van a emplear (líneas 241-242), y a continuación se inicia el bucle que va a desarrollar dicho algoritmo.

Para encontrar el punto buscado, se debe obtener el punto con el radio actual (línea 244) y se calcula, mediante el K-D tree, los puntos cercanos al robot (líneas 245-249). Una vez realizado dicho cálculo, se comprueba si el vector que almacena dichos puntos se encuentra vacío, en cuyo caso se define la señal binaria *vacia* como 1, y en caso contrario esta se define como 0 (líneas 250-262). Una vez definida esta variable binaria, se comprueba si los puntos encontrados se encuentran a una distancia radial del centro del robot, ya que el K-D tree ha calculado los puntos en una ventana cuadrada. En caso de que o bien el vector esté vacío o bien que los puntos calculados se encuentren a la distancia radial definida (línea 266), se define la variable binaria *f_col* como 0, y en caso de que los puntos encontrados estén a menor distancia de la definida anteriormente se define la variable *f_col* como 1 (líneas 263-276). Dicha variable se utiliza para establecer la condición de salida del bucle, que consiste en romper el bucle si esta tiene valor 0 o actualizar el valor de la variable R y realizar otra iteración en caso de que sea 1 (líneas 277-281).

Finalmente, una vez finalizado el bucle se ha encontrado el punto que cumple las condiciones buscadas, por lo que solo queda definirlo como destino intermedio y alcanzarlo antes de continuar con el destino impuesto al inicio del algoritmo.

```
284 -           P_ref_n = punto_destino;
285 -           puntos_elegidos = [puntos_elegidos;P_ref_n];
286 -           while 1
287 -             if norm(P_ref_n-P0) < 10
288 -               flag_destino = 1;
289 -               break
290 -             end
```

```
291 -             if f_aux_2 == 1
292 -               global_counter = 0;
293 -               ciclo_completo(P0,phi0,T,1,N)
294 -               f_aux_2 = 0;
295 -               distancias_n = zeros(size(nube,1),1);
296 -               for i=1:size(nube,1)
297 -                 distancias_n(i) = norm(P_ref_n-nube(i,1:2));
298 -               end
299 -             end
300 -
301 -             encontrar_punto(distancias_n)
302 -
303 -             pos_0 = P_mas_cercano;
304 -             ori_0 = ori_mas_cercano;
305 -
306 -             for h = 1:N
307 -
308 -               if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
309 -                 break
310 -               end
311 -
312 -               indice_ini = secuencia_encontrada(h);
313 -               indice_fin = secuencia_encontrada(h+N);
314 -
315 -               movB = 1;
316 -               T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
317 -                      sin(ori_0)  cos(ori_0) pos_0(2);
318 -                           0           0         1  ];
319 -               area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
320 -               [colision,pos_0,ori_0]=comprobacion_colisiones(movB,pos_0,ori_0,indice_ini,T_p,T,area);
321 -               if colision == 1
322 -                 break
323 -               end
324 -
325 -               movB = 0;
326 -               T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
327 -                      sin(ori_0)  cos(ori_0) pos_0(2);
328 -                           0           0         1  ];
329 -               area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
330 -               [colision,pos_0,ori_0]=comprobacion_colisiones(movB,pos_0,ori_0,indice_fin,T_p,T,area);
331 -               if colision == 1
332 -                 break
333 -               end
334 -
335 -             end
336 -
337 -             if colision == 0
338 -               registro_seq = [registro_seq; secuencia_encontrada];
339 -               puntos_encontrados = [puntos_encontrados; P_mas_cercano];
340 -               ori_encontradas = [ori_encontradas; ori_mas_cercano];
341 -               P0 = P_mas_cercano;
342 -               phi0 = ori_mas_cercano;
343 -               f_aux_2 = 1;
344 -               nube_grande = [nube_grande;nube];
345 -             else
346 -               distancias_n(indice_min) = 1000000000;
347 -             end
348 -           end
349 -         end
```

Para cumplir dicho objetivo intermedio, se va a aplicar el mismo algoritmo que se desarrolló en la sección 2.2, que consiste en mover al robot a través del espacio 2D solapando espacios de trabajo de 2 ciclos de movimiento, realizando la comprobación de colisiones en cada medio ciclo y penalizando los puntos que generen colisiones del robot con el entorno (líneas 284-349).

```
351 -          vect_eng = repulsor-P_ref_n;
352 -          u_eng = vect_eng/norm(vect_eng);
353 -          P_inicio_eng = P_ref_n+u_eng*120.65;
354 -          P_final_eng = repulsor;
355 -          alpha = atan2(vect_eng(2),vect_eng(1));
356 -          P1 = P_inicio_eng + 60.325*[cos(alpha-pi/2) sin(alpha-pi/2)];
357 -          P2 = P_inicio_eng + 60.325*[cos(alpha+pi/2) sin(alpha+pi/2)];
358 -          P3 = P_final_eng + 60.325*[cos(alpha+pi/2) sin(alpha+pi/2)];
359 -          P4 = P_final_eng + 60.325*[cos(alpha-pi/2) sin(alpha-pi/2)];
360 -
361 -          resol_rect = 10;
362 -          x12 = P1(1)+(P2(1)-P1(1))*[0:resol_rect/norm(P2-P1):1];
363 -          y12 = P1(2)+(P2(2)-P1(2))*[0:resol_rect/norm(P2-P1):1];
364 -          x23 = P2(1)+(P3(1)-P2(1))*[0:resol_rect/norm(P3-P2):1];
365 -          y23 = P2(2)+(P3(2)-P2(2))*[0:resol_rect/norm(P3-P2):1];
366 -          x34 = P3(1)+(P4(1)-P3(1))*[0:resol_rect/norm(P4-P3):1];
367 -          y34 = P3(2)+(P4(2)-P3(2))*[0:resol_rect/norm(P4-P3):1];
368 -          x41 = P4(1)+(P1(1)-P4(1))*[0:resol_rect/norm(P1-P4):1];
369 -          y41 = P4(2)+(P1(2)-P4(2))*[0:resol_rect/norm(P1-P4):1];
370 -
371 -          P_obs = [P_obs;[x12' y12'];[x23' y23'];[x34' y34'];[x41' y41']];
372 -          Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
```

Como se ha comentado anteriormente en esta sección, este algoritmo requiere de la generación de obstáculos virtuales en ciertos lugares con el fin de que el robot evite retroceder a posiciones anteriores y volver a entrar en el mínimo local del que acaba de salir. Para ello, se buscan los 4 vértices de dicho obstáculo virtual a partir de la posición del repulsor y del punto objetivo intermedio (líneas 351-359). A continuación, se define la frontera de dicho obstáculo virtual a partir de una resolución determinada (líneas 361-369) y se recalcula el K-D tree con los obstáculos antiguos y nuevos (líneas 371-372).

Finalmente, únicamente resta aplicar el algoritmo de orientación final (**ori_final.m**) y realizar la representación gráfica del resultado, idénticamente a la sección 2.2.

Con esto concluiría la explicación del algoritmo de evasión de obstáculos mediante el principio de seguimiento de pared. Lo siguiente que se expondrá serán los resultados de dicho algoritmo para el ejemplo mostrado en la Figura 12. Al igual que en el caso anterior, este algoritmo es capaz de encontrar dos soluciones posibles a este caso, una rodeando por la izquierda al obstáculo y otra por la derecha, por lo que se expondrán

ambas en la Figura 12. Nótese que en este caso también se han mostrado los obstáculos virtuales generados, que son regiones rectangulares que cubren aproximadamente el espacio recorrido por el robot desde el mínimo local hasta el destino intermedio.

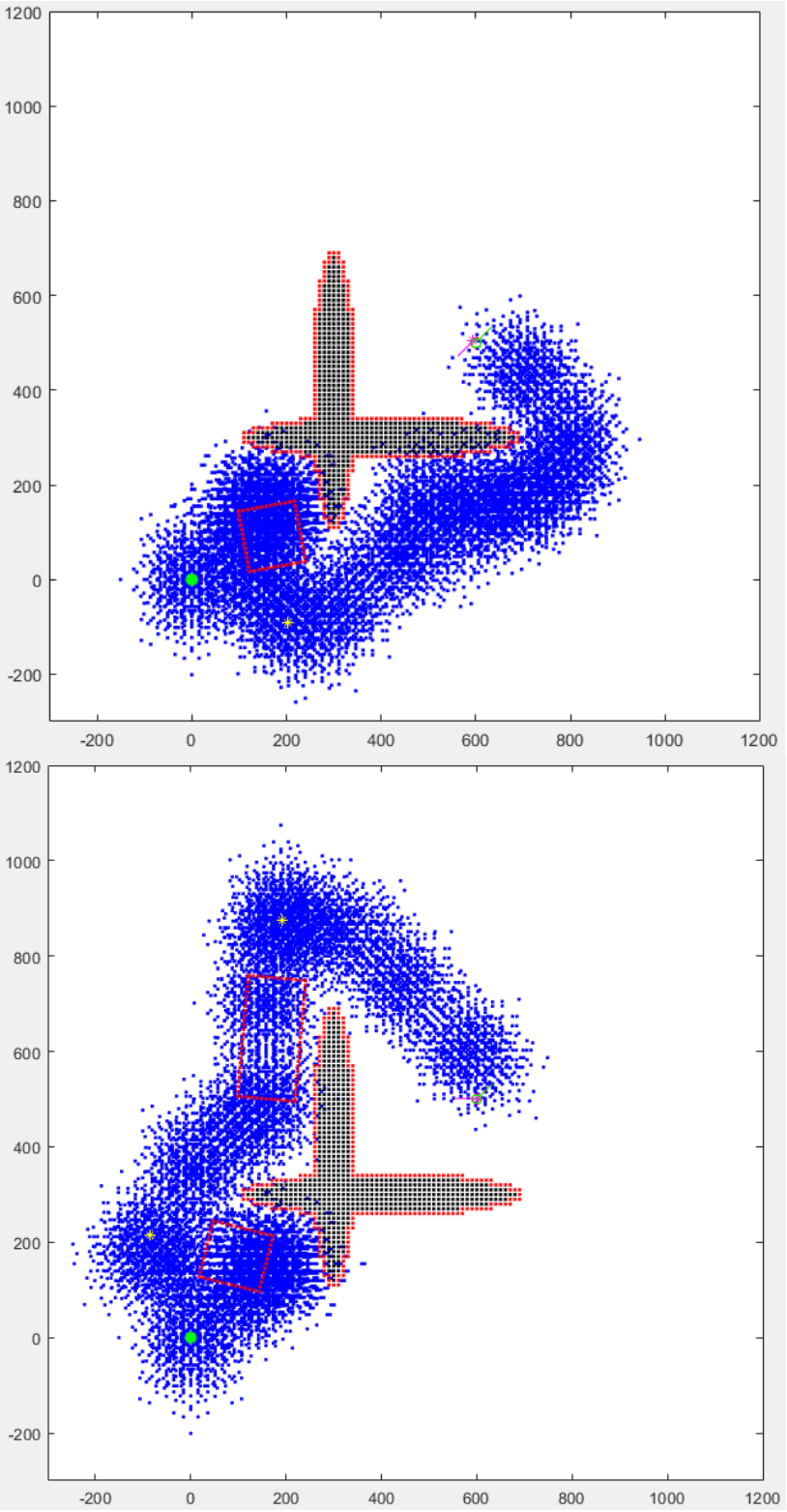

Figura 12: Resultados del algoritmo local dirigido (Wall-following).

Los resultados representados en la Figura 12 tienen el mismo código de colores que en la Figura 9 de la sección 2.2, por lo que solamente queda dar valores numéricos a los resultados obtenidos, y esto únicamente puede hacerse dando el valor absoluto del error de posición y el coste en ciclos de movimiento para cada caso. Dichos valores son 0.3803 mm de error en posición y 32 ciclos de movimiento (28 para posicionarse y 4 para orientarse) para el caso de la Figura 12.a y 0.7325 mm de error en posición y 34 ciclos de movimiento (30 para posicionarse y 4 para orientarse) para el caso de la Figura 12.b. Como puede apreciarse, este método requiere un número de ciclos mucho menor que el de generar destinos intermedios de forma aleatoria.

Capítulo 3. Planificación de trayectorias conocido el mapa.

3.1. Algoritmo A*.

El algoritmo A* es un algoritmo de búsqueda heurística de grafos, el cual encuentra, siempre y cuando se cumplan las condiciones pertinentes, el camino de menor coste entre un nodo origen y un nodo destino dentro de una malla. Un ejemplo de este algoritmo es el que puede verse en la Figura 13:

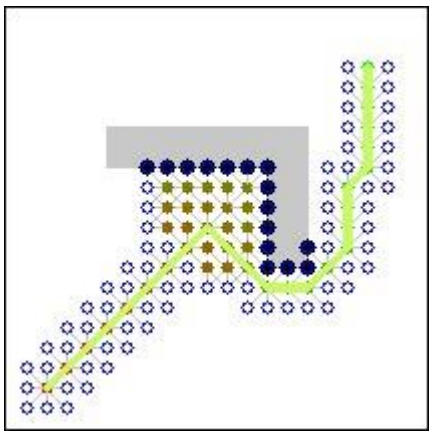

Figura 13: Representación gráfica del algoritmo A.*

Todos los puntos de la Figura 13 son nodos de la malla. De los nodos explorados, se seleccionan aquellos que conducen con un menor coste al objetivo marcado. Este es el principio de funcionamiento para el caso general del algoritmo A*.

No obstante, no es posible resolver el caso que nos ocupa solamente aplicando el caso general, sino que se requiere aplicar restricciones adicionales al algoritmo. A continuación, se explicará el código desarrollado para este apartado, que se encuentra en el Anexo 3 del presente Trabajo Fin de Máster. Las funciones presentes en dicho Anexo, y por tanto las correspondientes a este apartado son las siguientes: **fr_2D_malla.m**, **area_local.m**, **calculo_area_adyac.m**, **inversa.m**, **directa.m**, **ciclo_completo.m**, **encontrar_punto.m**, **calculo_area.m**, **comprobacion_colisiones.m**, **codigo_kdtree.m**, y **ori_final.m** como funciones idénticas a las secciones anteriores, y **ws_binario_mp.m**, **A_estrella.m**, **extractor_destinos.m**, y **dibujo.m** como funciones modificadas de funciones ya explicadas o totalmente nuevas.

En líneas generales, este algoritmo se basa en generar un mapa de ocupación del espacio, que consiste en una rejilla cuyas celdas (o nodos) subdividen dicho espacio en

unidades binarias que adoptan el valor 0 o 1 en función de si esa zona del espacio está libre o si hay un obstáculo en ella, respectivamente.

Para lograr este objetivo, se debe aplicar el algoritmo desarrollado en el script **ws_binario_mp.m** del Anexo 3, pero para ello primero es necesario conocer todas las subfunciones, por lo que antes se procederá a la explicación de **A_estrella.m**, **extractor_destinos.m** y **dibujo.m**.

La función **A_estrella.m** es la que se encarga de hacer la búsqueda heurística a partir del mapa de ocupación y de encontrar la mejor ruta posible hacia el objetivo. Esta función se desarrolló en *Gallego (2020)* a partir del pseudocódigo que se muestra en el Anexo 7, y esta se ha adaptado a las necesidades del presente Trabajo Fin de Máster. Por tanto, se va a proceder a explicar dichas adaptaciones realizadas al código original.

Como se ha comentado anteriormente, esta función consiste en generar un mapa de ocupación del espacio, para lo que se genera una matriz que divide dicho espacio en nodos. A continuación, partiendo del nodo inicial, se exploran los vecinos del mismo y se añaden a la lista de nodos sin explorar. De dicha lista, se selecciona el nodo cuyo coste de alcanzar el nodo objetivo es el menor y se toma como nodo siguiente, y el anterior se añade a la lista de nodos visitados. Posteriormente, se toma el nodo seleccionado como nodo actual, se exploran sus vecinos y se añaden a la lista de nodos sin explorar, y de dicha lista se selecciona el nodo de menor coste. Este proceso se repite sucesivamente hasta alcanzar el nodo destino, lográndose así resolver el caso general del algoritmo A*. Sin embargo, este caso concreto requiere que se cumpla una condición adicional, y es que, a diferencia del caso general, el robot estudiado en este trabajo no puede considerarse en ningún caso puntual, que es como se trata en el caso general, por lo que se ha añadido al algoritmo el siguiente código:

```
67 -     X = (fila-1)*paso_x + xmin;
68 -     Y = (columna-1)*paso_y + ymin;
69 -     radio = 120.65;
70 -     centro = [X Y];
71 -
72 -     Idx = rangesearch(Md1,centro,radio);
73 -     Retrieved = zeros(length(Idx{1}),2);
74 -     for k = 1:length(Idx{1})
75 -       Retrieved(k,:) = P_obs(Idx{1}(k),:);
76 -     end
77 -
78 -     flag_invalido = 0;
79 -     x_Retrieved = Retrieved(:,1);
80 -     y_Retrieved = Retrieved(:,2);
81 -     for w = 1:length(x_Retrieved)
```

```
82 -        p_Retrieved = [x_Retrieved(w) y_Retrieved(w)];
83 -        if norm(centro-p_Retrieved) <= radio
84 -          f_score(fila,columna) = 10^10;
85 -          flag_invalido = 1;
86 -          break
87 -        end
88 -      end
```

Esta condición se comprueba justo antes de seleccionar el nodo siguiente, haciendo que, si *flag_invalido* vale 1, no se elija un nuevo nodo, sino que se explore de nuevo en la lista de no explorados en busca de otro nodo que sí cumpla la restricción.

Para realizar dicha comprobación, simplemente se toma el nodo actual, se calcula su equivalente en posición en las líneas 67 y 68 (los nodos se representan por índices dentro del mapa de ocupación, y sabiendo el tamaño de cada nodo se puede saber la posición absoluta de cada uno) y se comprueba, mediante el K-D tree calculado con la frontera del obstáculo, si dicha posición se encuentra a la distancia mínima requerida (línea 83) del obstáculo. En caso afirmativo, *flag_invalido* valdrá 0, y, en caso contrario, valdrá 1, con las consecuencias explicadas previamente.

Una vez finalizada esta función, esta devuelve el camino de mínimo coste que debe recorrer el robot para alcanzar el objetivo a partir del tamaño de las celdas del mapa y de los límites del mismo. Dicho camino consiste en una serie de puntos consecutivos que unen el punto inicial y final, pero sería poco eficiente tomar todos esos puntos y decirle al robot que los recorra uno a uno, por lo que es necesario realizar otro algoritmo que se encargue de tomar los puntos más significativos de la ruta y hacer al robot alcanzarlos como destinos intermedios antes de alcanzar el objetivo. Dichos puntos se han obtenido mediante un algoritmo de división recursiva, cuyo funcionamiento se representa de forma visual en la Figura 14.

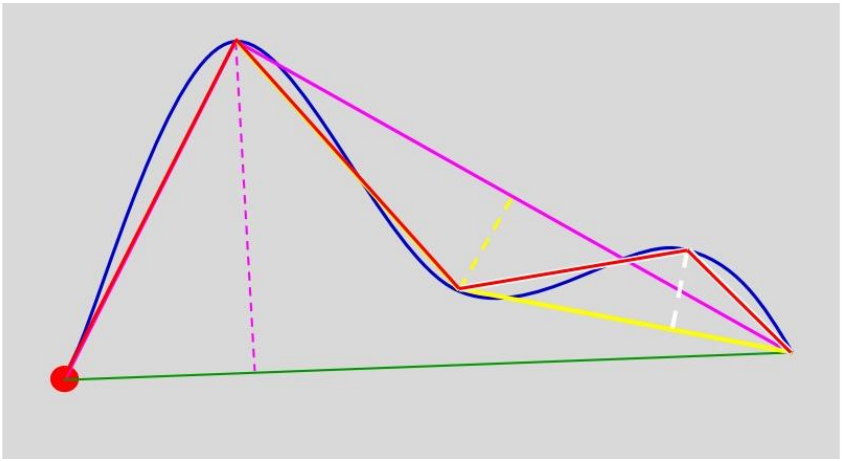

Figura 14: Representación gráfica de la segmentación de la trayectoria.

El objetivo es hallar los vértices de la línea roja teniendo como ruta la línea añil. Dicho algoritmo se ha desarrollado desde cero, y se encuentra en la función **extractor_destinos.m** del Anexo 3. Para ello, se va a desarrollar un algoritmo que busque el punto más lejano de la ruta entre dos puntos dados y determinar si este punto se puede considerar como significativamente dispar a la recta que une dichos puntos. Esta función recibe como parámetros de entrada la ruta de movimiento entre los dos puntos (*route*) y los puntos entre los que se va a calcular la ruta (*P1* y *P2*).

```
 3 -   tam = length(route(:,1));
 4 -   dist_vector = zeros(tam,1);
 5 -
 6 -   v = P1-P2;
 7 -   A = v(2)/v(1);
 8 -   B = -1;
 9 -   C = P1(2)-A*P1(1);
10 -   for i = 1:tam
11 -     Pi = route(i,:);
12 -     d = abs((A*Pi(1)+B*Pi(2)+C)/sqrt(A^2+B^2));
13 -     dist_vector(i) = d;
14 -   end
15 -   [dist_max,ind_max] = max(dist_vector);
16 -   destino = route(ind_max,:);
```

Como se puede apreciar, en dicha función se calculan los parámetros de la ecuación general de la recta que une los puntos mencionados anteriormente (líneas 6-9), y se recorre el segmento de la ruta seleccionado en busca del punto más lejano (líneas 10-15), y se devuelve este valor al algoritmo donde esta función es llamada, para su posterior consideración de si es suficientemente significativo en la ruta o no.

El objetivo de esta función es ir llamando a esta función recursivamente para dividir en segmentos la ruta obtenida en la función **A_estrella.m** con el fin de enviar al robot de un punto a otro consecutivamente con el fin de que este alcance el objetivo inicial.

Antes de explicar el desarrollo completo del algoritmo, queda por definir los cambios realizados en la representación gráfica de los resultados a partir del *script* **dibujo.m**. Estos cambios se basan únicamente en añadir la representación gráfica de la ruta calculada y algún que otro cambio en el código de colores, que se explicará cuando se expliquen los resultados obtenidos.

Como se ha comentado, solamente resta explicar la aplicación del algoritmo en sí, que se desarrolla en el *script* **ws_binario_mp.m** del Anexo 3. Sus primeras líneas son prácticamente idénticas a las de las anteriores secciones, basándose en declaración de

variables globales, inicialización de variables, definición de los obstáculos, el cálculo de su frontera mediante **fr_2D_malla.m**, comprobación de obstrucción de los puntos inicial y final, el cálculo del K-D tree del entorno y el cálculo de las áreas barridas por los eslabones en cada medio ciclo mediante la función **area_local.m**.

```
79 - resol_A_estrella = 10;
80 - route=A_estrella(resol_A_estrella,resol_A_estrella,P0(1),P0(2),P_ref(1),P_ref(2),xmin,xmax,ymin,ymax);
81 - route = [P_ref;route;P0];
82 - puntos_dest = [P_ref;P0];
83 - ind_dest = [1,length(route(:,1))];
84 - umbral = 50;
85 - while 1
86 -
87 -   for i = 1:length(ind_dest)-1
88 -     ind1 = ind_dest(i);
89 -     ind2 = ind_dest(i+1);
90 -     P1 = puntos_dest(i,:);
91 -     P2 = puntos_dest(i+1,:);
92 -     [destino,dist_max,ind_max] = extractor_destinos(route(ind1:ind2,:),P1,P2);
93 -     if dist_max > umbral
94 -       flag_1 = 0;
95 -       break
96 -     else
97 -       flag_1 = 1;
98 -     end
99 -   end
100 -
101 -  if flag_1 == 1
102 -    break
103 -  else
104 -    tam_ind_dest = length(ind_dest);
105 -    ind_act = ind_max + ind1;
106 -    for j = 1:tam_ind_dest
107 -      if ind_dest(j) > ind_act
108 -        ind_dest = [ind_dest(1:j-1) ind_act ind_dest(j:tam_ind_dest)];
109 -        puntos_dest = [puntos_dest(1:j-1,:); destino; puntos_dest(j:tam_ind_dest,:)];
110 -        break
111 -      end
112 -    end
113 -  end
114 -
115 - end
```

Tras lo mencionado anteriormente, el primer paso es la ejecución del algoritmo **A_estrella.m** y obtener la ruta que debe seguir el robot (línea 80). A continuación, se debe segmentar dicha ruta en sus puntos significativos tal y como se ha representado en la Figura 14. Para ello, se debe emplear el código mostrado anteriormente.

En primer lugar, se calcula la ruta a seguir por el robot (línea 80), se añaden a dicha ruta los puntos inicial y final (línea 81) y se inicializa la variable que define los primeros puntos entre los que se va a dividir la ruta, que en todos los casos serán, inicialmente, el punto inicial y el de destino (línea 82). Finalmente, se inicializa la variable

que va a almacenar los índices de todos los puntos entre los que se divida la ruta (línea 83). A continuación, se va a proceder al desarrollo del bucle iniciado en la línea 85 que se encargará de obtener dicha segmentación de la ruta.

Para ello, en primer lugar, se procede a comprobar si todos los puntos de segmentación encontrados hasta el instante actual cumplen con la condición del umbral establecido (líneas 87-99). Dicho umbral representa la distancia máxima entre la curva de la ruta y las rectas que generan los puntos de segmentación encontrados. Para ello, es necesario aplicar la función **extractor_destinos.m** entre todos los puntos encontrados y comprobando si la distancia máxima devuelta por dicha función es mayor al umbral; en caso afirmativo, la variable binaria *flag_1* toma el valor 0 (línea 94), lo que indica que se debe añadir un punto de segmentación entre los dos puntos analizados, y en caso contrario, la variable binaria *flag_1* toma el valor 1 (línea 97), lo que indica que no se ha encontrado ningún punto con una distancia máxima a la recta correspondiente mayor al umbral establecido.

Finalmente, se debe aplicar el criterio mencionado anteriormente en función del valor de la variable *flag_1*. En caso de que esta sea 1, como se ha comentado, significa que no existen segmentos que no cumplan con el umbral establecido, por lo que se rompe el bucle y se da como bien segmentada la ruta (líneas 101-102). En caso contrario, se debe añadir el punto de máxima distancia que no ha cumplido dicho umbral a la variable *puntos_dest* y sus índices a la variable *ind_dest* para su comprobación en la siguiente iteración; para añadir dichos puntos, se debe hacer en orden de los índices de los puntos, por lo que se debe recorrer *ind_dest* y colocar el nuevo punto en la posición que corresponda (líneas 106-112).

```
116 - for i = length(ind_dest):-1:1
117 -   P_ref_actual = puntos_dest(i,:);
118 -   f_aux = 1;
119 -   while 1
120 -     if norm(P_ref_actual-P0) < 10
121 -       break
122 -     end
123 -     if f_aux == 1
124 -       global_counter = 0;
125 -       ciclo_completo(P0,phi0,T,1,N);
126 -       f_aux = 0;
127 -       distancias = zeros(size(nube,1),1);
128 -       for j=1:size(nube,1)
129 -         distancias(j) = norm(P_ref_actual-nube(j,1:2));
130 -       end
131 -     end
132 -     encontrar_punto(distancias)
```

```
133 -     pos_0 = P_mas_cercano;
134 -     ori_0 = ori_mas_cercano;
135 -     for h = 1:N
136 -       if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
137 -         break
138 -       end
139 -       indice_inicio = secuencia_encontrada(h);
140 -       indice_fin = secuencia_encontrada(h+N);
141 -       movB = 1;
142 -       T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
143 -              sin(ori_0)  cos(ori_0) pos_0(2);
144 -                   0           0         1  ];
145 -       area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
146 -       [colision,pos_0,ori_0]=comprobacion_colisiones(movB,pos_0,ori_0,indice_inicio,T_p,T,area);
147 -       if colision == 1
148 -         break
149 -       end
150 -       movB = 0;
151 -       T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
152 -              sin(ori_0)  cos(ori_0) pos_0(2);
153 -                   0           0         1  ];
154 -       area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
155 -       [colision,pos_0,ori_0]=comprobacion_colisiones(movB,pos_0,ori_0,indice_fin,T_p,T,area);
156 -       if colision == 1
157 -         break
158 -       end
159 -     end
160 -     if colision == 0
161 -       registro_seq = [registro_seq; secuencia_encontrada];
162 -       puntos_encontrados = [puntos_encontrados; P_mas_cercano];
163 -       ori_encontradas = [ori_encontradas; ori_mas_cercano];
164 -       P0 = P_mas_cercano;
165 -       phi0 = ori_mas_cercano;
166 -       f_aux = 1;
167 -       nube_grande = [nube_grande;nube];
168 -     else
169 -       distancias(indice_min) = 1000000000;
170 -     end
171 -   end
172 - 
173 - end
174 - ori_final
175 - dibujo
```

Una vez obtenidos dichos puntos, solamente resta que el robot los alcance uno tras otro. Para ello, se sigue el mismo principio que se aplicó en las secciones 2.2 y 2.3, enviando al robot a dichos puntos uno tras otro, pero esta vez sin aplicar el criterio de mínimos locales, ya que se sabe que en este caso no existirán. A pesar de que se sabe que los puntos encontrados en la ruta cumplen todos con el criterio de la distancia mínima con el obstáculo para que el robot no colisione, los trayectos entre puntos no necesariamente los cumplen, puesto que a pesar de que el robot seguirá una trayectoria más o menos rectilínea entre dos puntos consecutivos (tal y como muestra la polilínea roja de la Figura 14), es posible que no sea del todo recta y se aproxime más de la cuenta a un obstáculo, por tanto, se ha implementado la comprobación de colisiones en los trayectos rectilíneos

(líneas 135-173). Finalmente, se aplica el algoritmo de orientación final (línea 174) y se representa gráficamente el resultado (línea 175).

Con esto concluiría la explicación del algoritmo de evasión de obstáculos mediante la aplicación del algoritmo A*. Lo siguiente que se expondrá serán los resultados de dicho algoritmo para el ejemplo mostrado en la Figura 7. Al contrario que los casos anteriores, este algoritmo es capaz de encontrar una solución posible a este caso, que es la que se expone en la Figura 15. En este caso, se debe añadir en el código de colores que el color rojo se utiliza para representar no solo la frontera del obstáculo sino también el camino de puntos encontrados por el algoritmo A*, y los asteriscos amarillos en este caso representan los puntos entre los que se mueve el robot consecutivamente para alcanzar el objetivo, y no los puntos generados para el escape de mínimos locales como en las secciones 2.2 y 2.3.

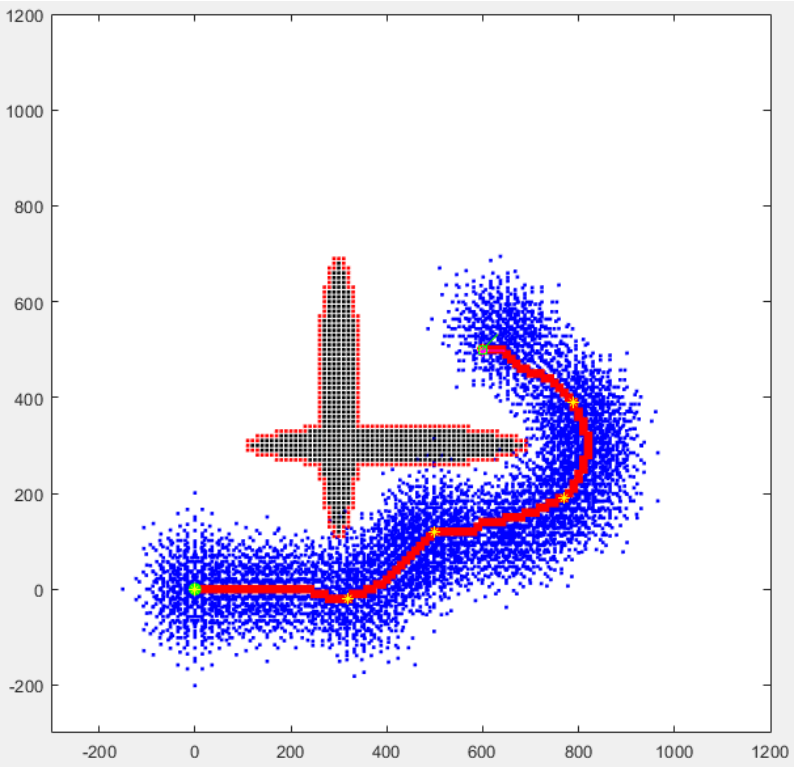

Figura 15: Resultado del algoritmo A.*

Únicamente resta dar valores numéricos a los resultados obtenidos, y esto únicamente puede hacerse dando los valores absolutos de los errores de posición y el coste de ciclos de movimiento para alcanzar el objetivo. En este caso, el algoritmo ha conseguido una precisión de 0.7492 mm, y ha realizado 20 ciclos para posicionarse y 4 para orientarse (24 en total).

3.2. Diagrama de Voronoi.

El diagrama de Voronoi es un algoritmo estructural que, en esencia, se encarga de dividir todo el espacio donde el robot puede moverse (es decir, todo el espacio no ocupado por obstáculos) en regiones cuyos límites equidisten de dos obstáculos, para así hacer que el robot se desplace por dichos límites, siempre lo más alejado posible de los obstáculos. Unos ejemplos sencillos de este algoritmo son los que se ven en la Figura 16.

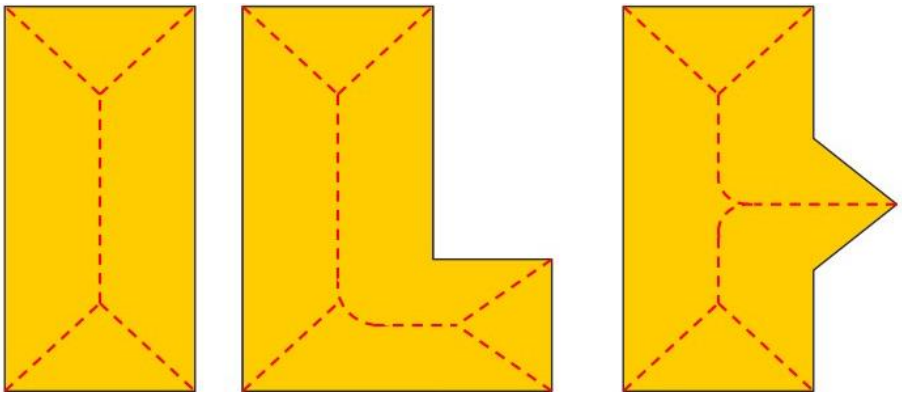

Figura 16: Ejemplos de diagramas de Voronoi.

Como se puede apreciar en estos ejemplos, el algoritmo se encarga de obtener el conjunto de puntos que equidistan de, en estos casos, las paredes de los espacios de movimiento. Si se obliga al robot a moverse por dichos puntos se asegura que este no va a entrar en mínimos locales y que, al alejarse de los obstáculos lo máximo posible, tampoco tendrá conflictos con las colisiones.

El algoritmo que se encarga de extraer estos "caminos" equidistantes a los obstáculos ha sido implementado utilizando la librería *Machine Vision Toolbox for MATLAB*, creada por Peter Corke. Esta es una librería que tiene una gran cantidad de funciones útiles para la visión por computador, pero de todas ellas, para esta aplicación, solamente nos interesa una: *ithin*. Esta función recibe como parámetro de entrada el mapa de ocupación, dividido en celdas, las cuales adoptan el valor de 1 cuando pertenecen al espacio libre y 0 cuando pertenecen al espacio ocupado, y devuelve una matriz con las mismas dimensiones que el mapa de ocupación, pero, esta vez, las celdas que toman el valor 1 son las que pertenecen a los "caminos" del diagrama de Voronoi, y las que tienen valor 0 son las celdas que no pertenecen a los mismos.

Este algoritmo está reflejado en el Anexo 4, y las funciones correspondientes al mismo son las siguientes: **fr_2D_malla.m**, **area_local.m**, **calculo_area_adyac.m**,

inversa.m, **directa.m**, **extractor_destinos.m**, **ciclo_completo.m**, **encontrar_punto.m**, **calculo_area.m**, **comprobacion_colisiones.m**, **codigo_kdtree** y **ori_final.m** como funciones idénticas a las de la sección 3.1 y **ws_binario_mp.m**, **voronoi.m**, **A_estrella.m** y **dibujo.m** como funciones modificadas de las anteriores o completamente nuevas.

Como en todas las secciones anteriores, el *script* que desarrolla el algoritmo perteneciente a esta sección es **ws_binario_mp.m**, pero para poder comprenderlo en su totalidad, primero es necesario explicar el resto de funciones. Es por eso que la primera de estas que se explicará será la llamada **voronoi.m**. Esta función recibe como parámetros de entrada los tamaños de las celdas del mapa de ocupación (*paso_x* y *paso_y*), los límites de dicho mapa (*xmax*, *xmin*, *ymax* e *ymin*) y las posiciones inicial y final del movimiento (*P0* y *P_ref*).

```
  3 -   global Md
  4 -   global P_obs
  5 -
  6 -   Dx = xmax-xmin;
  7 -   Dy = ymax-ymin;
  8 -   g = round((Dx)/paso_x)+1;
  9 -   h = round((Dy)/paso_y)+1;
 10 -   hay_obstaculo = zeros(g,h);
 11 -   frontera = [];
 12 -   caminos_voronoi = [];
 13 -   esquema = [];
 14 -   v_paso = [paso_x,paso_y];
 15 -   radio = min(v_paso)/2;
 16 -
 17 -   r = (P_ref(1)-xmin)/paso_x+1;
 18 -   s = (P_ref(2)-ymin)/paso_y+1;
 19 -   hay_obstaculo(r,s) = 1;
 20 -   r = (P0(1)-xmin)/paso_x+1;
 21 -   s = (P0(2)-ymin)/paso_y+1;
 22 -   hay_obstaculo(r,s) = 1;
 23 -
 24 -   for i=1:g
 25 -     x = (i-1)*paso_x+xmin;
 26 -     for j = 1:h
 27 -       y = (j-1)*paso_y + ymin;
 28 -       pto = [x y];
 29 -       Idx = rangesearch(Md,pto,radio);
 30 -       Retrieved = zeros(length(Idx{1}),2);
 31 -       if isempty(Idx{1}) == 0
 32 -         hay_obstaculo(i,j) = 1;
 33 -       end
 34 -     end
 35 -   end
 36 -   hay_obstaculo(1,:) = 1;
 37 -   hay_obstaculo(g,:) = 1;
 38 -   hay_obstaculo(:,1) = 1;
 39 -   hay_obstaculo(:,h) = 1;
 40 -
 41 -   esq = ithin(1-hay_obstaculo);
 42 -
```

```
43 -    for b = 1:g
44 -      for c = 1:h
45 -        q = [b c];
46 -        if esq(b,c) == 1
47 -          esquema = [esquema; q];
48 -          caminos_voronoi=[caminos_voronoi;[(q(1)-1)*paso_x+xmin,(q(2)-1)*paso_y+ymin]];
49 -        end
50 -      end
51 -    end
```

Esta función pretende calcular el diagrama de Voronoi del mapa del entorno del robot. Para ello, en primer lugar, se deben inicializar las variables que posteriormente se van a utilizar en el algoritmo (líneas 6-15); dichas variables son, en general, los parámetros de la malla (tamaño, tamaño de celda, número de filas y columnas, etc.). Para calcular dicho diagrama, se necesita generar una matriz binaria que contenga 1 en las celdas del espacio libre y 0 en las celdas del espacio ocupado por obstáculos; dicha matriz no se tiene en un inicio, por lo que se debe generar a partir del K-D tree calculado fuera de la función y declarado como variable global.

Dicha matriz corresponde con la variable *hay_obstaculo*, inicializada en la línea 10 como una matriz de ceros con las dimensiones del mapa del entorno. Lo primero que se hace es añadir los puntos inicial y final a la matriz de ocupación (líneas 17-22) definiéndolos como puntos obstáculo, ya que es una forma de que el algoritmo realice un camino que alcance dichos puntos. Una vez definidos los puntos inicial y final en la matriz de ocupación, se debe definir los obstáculos en sí, para lo que se recorre todo el mapa con la misma resolución con la que estén definidos los obstáculos con el fin de encontrar qué puntos del K-D tree se encuentran ocupados y definirlos dentro de la matriz (líneas 24-35). A continuación, se deben definir las paredes del mapa también como obstáculos (líneas 36-39).

El siguiente paso es el cálculo del diagrama de Voronoi a partir de la matriz calculada. Para poder hacer dicho cálculo, es necesario invertir los valores de las celdas de la matriz, y que esta se ha calculado para que se definan con un 1 las celdas ocupadas con obstáculos y viceversa, por lo que se debe aplicar la función *ithin* con la entrada *1-hay_obstáculo* (línea 41). Finalmente, solamente queda convertir la matriz de celdas a variables de posición y orientación (líneas 43-51). El resultado obtenido de esta función es el que se muestra en la Figura 17, mostrado en color rojo.

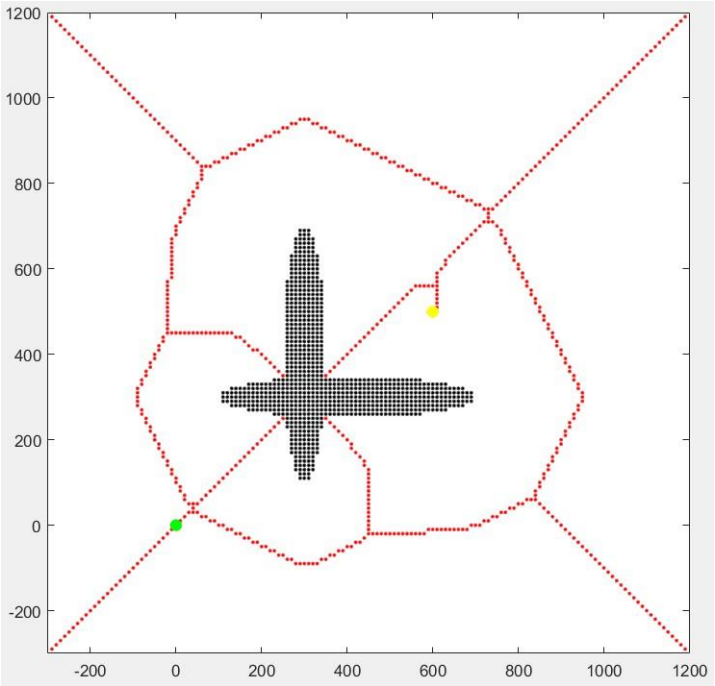

Figura 17: Caminos del diagrama de Voronoi.

Una vez obtenido el diagrama, es necesario recorrer dichos caminos mediante el principio del algoritmo A* con el objetivo de encontrar el camino de menor coste que conecte el punto inicial y final pero esta vez siguiendo los caminos definidos en el diagrama de Voronoi. Para ello, se ha desarrollado la función **A_estrella.m**, que es una variación de la aplicada en la sección 3.1 pero con la siguiente modificación:

```
73 -      vecinos_p = zeros(8,2);
74 -      z = 1;
75 -      for i = -1:1
76 -        for j = -1:1
77 -          if (i~=0 || j~=0)
78 -            vecinos_p(z,1) = fila+i;
79 -            vecinos_p(z,2) = columna+j;
80 -            z = z+1;
81 -          end
82 -        end
83 -      end
84 -
85 -      vecinos = [];
86 -      for i = 1:8
87 -        v = vecinos_p(i,:);
88 -        for j = 1:length(esquema_v)
89 -          if esquema_v(j,:) == v
90 -            vecinos = [vecinos;v];
91 -            break
92 -          end
93 -        end
94 -      end
```

La función es idéntica a la de la sección 3.1 salvo en el segmento de código mostrado. Las líneas 73-83 sí son idénticas a las del algoritmo anterior, y estas representan

los 8 vecinos del punto que se está analizando. La idea es limitar dichos vecinos a los que sí se encuentran dentro del diagrama de Voronoi. Para ello, se ha añadido el código de las líneas 85-94, que se encarga de encontrar cuáles de esos 8 vecinos están dentro del diagrama (en este caso llamado *esquema_v*) con el fin de limitar el siguiente punto al que el robot puede ir para alcanzar el objetivo, ya que la búsqueda se hará entre los vecinos encontrados dentro de la variable *vecinos*.

Antes de explicar el algoritmo en sí, solamente queda por mencionar los cambios realizados en **dibujo.m**. Dichos cambios consisten en añadir la representación gráfica de los caminos de Voronoi, así como el cambio del código de colores adaptado a este caso, el cual se explicará cuando se expongan los resultados obtenidos.

Finalmente, como se ha comentado anteriormente, se procederá a la explicación del *script* **ws_binario_mp.m**, el cual se encarga de la aplicación del algoritmo correspondiente a la presente sección, haciendo las llamadas pertinentes a las funciones cuando se requiera. Dicho *script* comienza exactamente igual que los de las secciones anteriores: inicializando variables, definiendo los obstáculos, determinando si los puntos inicial y final se encuentran obstruidos por obstáculos, calculando la frontera de dichos obstáculos y calculando el K-D tree del entorno tomando dichas fronteras para su posterior uso en el algoritmo. No obstante, este algoritmo requiere que se calcule el diagrama de Voronoi a partir de los obstáculos completos, no de las fronteras, por lo que antes de obtener la frontera se debe ejecutar el algoritmo explicado anteriormente:

```
58 - xmin = -300; xmax = 1200; ymin = -300; ymax = 1200;
59 - [caminos_voronoi,esquema] = voronoi(paso_x,paso_y,xmin,xmax,ymin,ymax,P0,P_ref);
60 - P_obs = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax);
61 - Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
62 - xin=0;yin=0;xfin=600;yfin=500;
63 - route = A_estrella(paso_x,paso_y,xin,yin,xfin,yfin,xmin,xmax,ymin,ymax,esquema);
```

Una vez obtenidos el diagrama de Voronoi (líneas 58-59) tanto en posiciones absolutas (*caminos_voronoi*) como en forma de matriz de celdas binarias (*esquema*), se procede, como se ha mencionado, al cálculo de la frontera de los obstáculos (línea 60). A continuación, se aplica la función **A_estrella.m** explicada anteriormente (línea 63) para encontrar el camino que recorra los caminos de Voronoi y alcance el objetivo con el menor coste posible. Posteriormente, se terminan de inicializar las variables que restan por hacerlo y se calcula el área que barren los eslabones en coordenadas locales mediante la función **area_local.m**.

Una vez obtenida la ruta por la que va a moverse el robot mediante el algoritmo A*, el resto del código es idéntico al de la sección 3.1, en el que, a partir de la ruta de movimiento, se obtienen los puntos característicos de dicha ruta (empleando la función **extractor_destinos.m**) con el fin de enviar al robot de uno a otro consecutivamente, comprobando las colisiones en cada medio ciclo de la misma forma que se ha realizado en las secciones anteriores a esta.

Con esto concluiría la explicación del algoritmo de evasión de obstáculos mediante el diagrama de Voronoi. Lo siguiente que se expondrá serán los resultados de dicho algoritmo para el ejemplo mostrado en la Figura 7. Al igual que en el caso de la sección 3.1, este algoritmo es capaz de encontrar una solución posible a este caso, que es la que se expone en la Figura 18. En este caso, se debe añadir en el código de colores que, al igual que en la sección 3.1, el color rojo se utiliza para representar no solo la frontera del obstáculo sino también el camino de puntos encontrados por el algoritmo A*, los asteriscos amarillos en este caso representan los puntos entre los que se mueve el robot consecutivamente para alcanzar el objetivo, y el color cian se emplea para representar el propio diagrama de Voronoi.

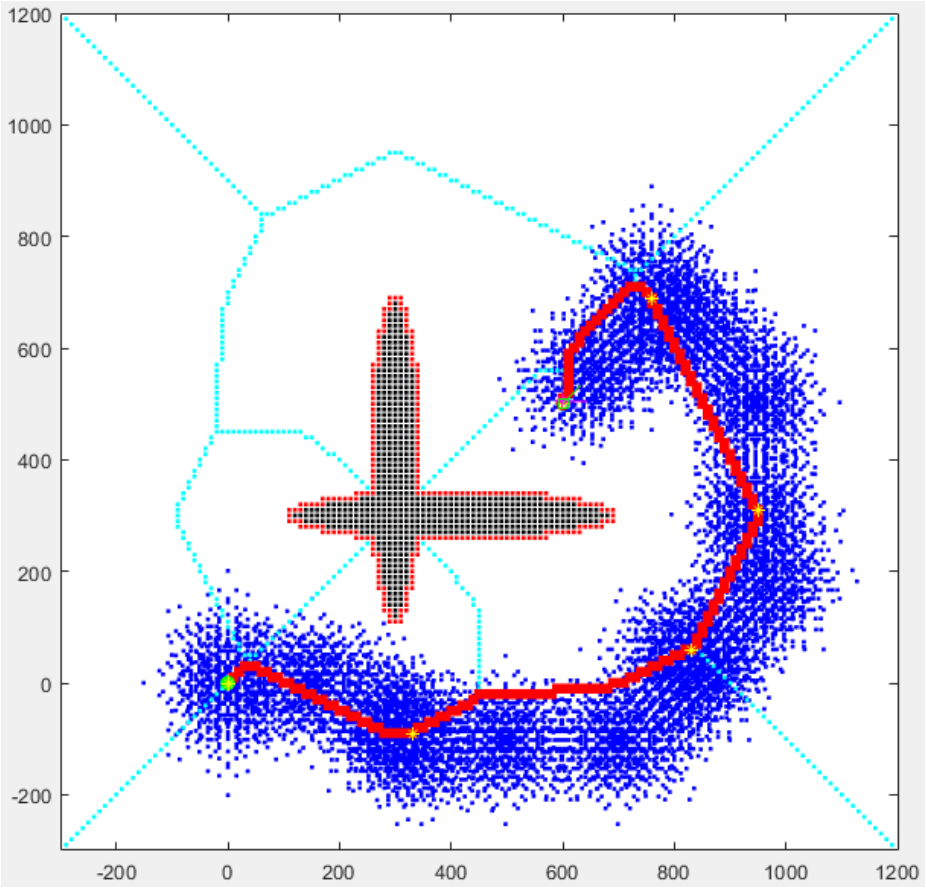

Figura 18: Resultados del diagrama de Voronoi.

Únicamente resta dar valores numéricos a los resultados obtenidos, y esto únicamente puede hacerse dando los valores absolutos de los errores de posición y el coste de ciclos de movimiento para alcanzar el objetivo. En este caso, el algoritmo ha conseguido una precisión de 1.0401 mm, y ha realizado 28 ciclos para posicionarse y 4 para orientarse (32 en total).

Capítulo 4. Aumento del número de ciclos de movimiento.

En este capítulo se van a desarrollar una serie de algoritmos que pretenden mejorar el algoritmo de fuerza bruta para el cálculo del espacio de trabajo con un número de ciclos de movimiento mayor a 3. Para ello, se han decidido desarrollar dos tipos de algoritmos: dos basados en inteligencia artificial, desarrollados en la primera sección de este capítulo, y uno basado en un robot hiper-redundante, desarrollado en la segunda sección de este capítulo.

4.1. Algoritmos basados en inteligencia artificial.

Como se ha comentado anteriormente, en esta sección se pretenden desarrollar dos algoritmos basados en inteligencia artificial. El primero será mediante el entrenamiento de una red neuronal que se encargue de desarrollar dicha tarea, y el segundo consistirá en un algoritmo genético.

4.1.1. Entrenamiento de redes neuronales.

Como se ha comentado, el objetivo de estos algoritmos basados en inteligencia artificial es el desarrollo de los mismos para un número de ciclos fijado y no demasiado elevado, pero con el fin de demostrar que este tipo de algoritmos podrían ser extrapolados a un número de ciclos mayor simplemente aumentando el tiempo de entrenamiento o ejecución de los algoritmos diseñados.

Como se ha comentado, uno de los algoritmos basados en inteligencia artificial que se van a intentar diseñar para este Trabajo Fin de Máster es la aplicación de redes neuronales. Cabe destacar el uso de la palabra “intentar”, ya que, se adelanta que, con el algoritmo de redes neuronales, los resultados obtenidos han sido desfavorables. Este algoritmo se ha desarrollado en Python, en lugar de Matlab como el resto del presente trabajo, y se ha desarrollado para N = 3 ciclos de movimiento.

En primer lugar, van a explicarse los primeros algoritmos planteados basados en redes neuronales, que, si bien fueron descartados, supusieron una ayuda para poder alcanzar las soluciones que sí tenían sentido.

La primera idea que se planteó fue el desarrollo de una regresión, introduciendo todos los datos de entrada (posiciones y orientaciones) y salida (vectores de

configuraciones discretas adoptadas por el robot en cada semiciclo) tal cual se han obtenido a la red. Para este caso, existía un problema con los datos, y es que cuando el robot no completaba los 3 ciclos, también se almacenaban esas posiciones, dejando un -1 en los índices que no se "utilizaban", y se consideró eliminar dichos datos, ya que constituían muy poco porcentaje del total y estos podrían complicar tanto el planteamiento de la red como el entrenamiento de la misma. Tras el tratamiento de los datos, se volvió a entrenar la red con ellos, y la propia función *fit* de Python ha devuelto una precisión final de alrededor del 17%, por lo que se ha descartado esta opción.

A continuación, se planteó que, al tratarse de una cinemática inversa, el robot estaría entrenando que, para alcanzar una posición y orientación determinada, puede hacerlo mediante distintas secuencias de movimiento, por lo que se pensó que esto podría estar confundiendo a la red neuronal, y se decidió volver a tratar los datos para eliminar estas duplicidades y que así la red solamente reciba un dato por cada posición y orientación. Este tratamiento de los datos redujo el número de los mismos desde 266000 aproximadamente hasta 31470. Una vez reducido el número de datos, se entrenó a la red de la misma forma que en el caso anterior con los nuevos datos, obteniendo, de la propia función *fit*, una precisión de aproximadamente un 21%. A pesar de ser una precisión mejor que en el caso anterior, tampoco es un resultado acertado.

Con la reducción de datos de 266000 a 31000, se planteó entrenar a la red como una clasificación, pidiéndole que clasificara los 266000 datos en 31000 clases diferentes, pero debido al número tan reducido de datos por clase (8 datos/clase en promedio), se ha descartado directamente, ya que este número tan bajo haría que no se clasifiquen bien los datos.

El primer método que se ha planteado con la expectativa de obtener un resultado mejor que los planteados anteriormente es el de entrenar 6 redes neuronales (2 por cada ciclo de movimiento) unidireccionales de tipo clasificación, una para cada valor de la salida. Se ha planteado así debido a que se ha considerado que, al tener solamente una salida, las redes serían capaces de converger a soluciones más correctas.

El código que se ha desarrollado para este apartado es el que se muestra a continuación:

```
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense
from sklearn.preprocessing import OneHotEncoder
import random

nube = np.loadtxt('./nube.txt',delimiter=",")
secuencia = np.loadtxt('./secuencia_enteros.txt',delimiter=",")

nube = nube.astype(float)
secuencia = secuencia.astype(int)

v_elim = random.sample(range(31437), 4000)

nube_validacion = []
secuencia_validacion = []
for i in v_elim:
    nube_validacion.append(nube[i, 0:])
    secuencia_validacion.append(secuencia[i, 0:])

nube_validacion = np.array(nube_validacion, dtype=np.float64 )
secuencia_validacion = np.array(secuencia_validacion, dtype=np.int64)

nube = np.delete(nube,(v_elim),axis=0)
secuencia = np.delete(secuencia,(v_elim),axis=0)

# Red 1
model = Sequential()
model.add(Dense(20,input_dim=3,activation='relu'))
model.add(Dense(20,input_dim=20,activation='relu'))
model.add(Dense(20,input_dim=20,activation='relu'))
model.add(Dense(8,activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
col_seq = [row[0] for row in secuencia]
ohe = OneHotEncoder()
col_reshaped = np.array(col_seq).reshape(-1,1)
col = ohe.fit_transform(col_reshaped).toarray()
history = model.fit(nube,col,epochs=10)

# Red 2
model2 = Sequential()
model2.add(Dense(20,input_dim=3,activation='relu'))
model2.add(Dense(20,input_dim=20,activation='relu'))
model2.add(Dense(20,input_dim=20,activation='relu'))
model2.add(Dense(8,activation='softmax'))
model2.summary()
model2.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
col_seq = [row[1] for row in secuencia]
ohe = OneHotEncoder()
col_reshaped = np.array(col_seq).reshape(-1,1)
col = ohe.fit_transform(col_reshaped).toarray()
history = model2.fit(nube,col,epochs=10)

# Red 3
model3 = Sequential()
model3.add(Dense(20,input_dim=3,activation='relu'))
model3.add(Dense(20,input_dim=20,activation='relu'))
model3.add(Dense(20,input_dim=20,activation='relu'))
model3.add(Dense(8,activation='softmax'))
model3.summary()
model3.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
col_seq = [row[2] for row in secuencia]
ohe = OneHotEncoder()
col_reshaped = np.array(col_seq).reshape(-1,1)
col = ohe.fit_transform(col_reshaped).toarray()
history = model3.fit(nube,col,epochs=10)

```

```
# Red 4
model4 = Sequential()
model4.add(Dense(20,input_dim=3,activation='relu'))
model4.add(Dense(20,input_dim=20,activation='relu'))
model4.add(Dense(20,input_dim=20,activation='relu'))
model4.add(Dense(8,activation='softmax'))
model4.summary()
model4.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
col_seq = [row[3] for row in secuencia]
ohe = OneHotEncoder()
col_reshaped = np.array(col_seq).reshape(-1,1)
col = ohe.fit_transform(col_reshaped).toarray()
history = model4.fit(nube,col,epochs=10)

# Red 5
model5 = Sequential()
model5.add(Dense(20,input_dim=3,activation='relu'))
model5.add(Dense(20,input_dim=20,activation='relu'))
model5.add(Dense(20,input_dim=20,activation='relu'))
model5.add(Dense(8,activation='softmax'))
model5.summary()
model5.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
col_seq = [row[4] for row in secuencia]
ohe = OneHotEncoder()
col_reshaped = np.array(col_seq).reshape(-1,1)
col = ohe.fit_transform(col_reshaped).toarray()
history = model5.fit(nube,col,epochs=10)

# Red 6
model6 = Sequential()
model6.add(Dense(20,input_dim=3,activation='relu'))
model6.add(Dense(20,input_dim=20,activation='relu'))
model6.add(Dense(20,input_dim=20,activation='relu'))
model6.add(Dense(8,activation='softmax'))
model6.summary()
model6.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
col_seq = [row[5] for row in secuencia]
ohe = OneHotEncoder()
col_reshaped = np.array(col_seq).reshape(-1,1)
col = ohe.fit_transform(col_reshaped).toarray()
history = model6.fit(nube,col,epochs=10)
# Predicción red 1
prediccion = model.predict(col)
pred = list()
for i in range(len(prediccion)):
    pred.append(np.argmax(prediccion[i]))
col_test = [row[0] for row in secuencia_validacion]
ohe = OneHotEncoder()
col_reshaped_test = np.array(col_test).reshape(-1,1)
col = ohe.fit_transform(col_reshaped_test).toarray()
test = list()
for i in range(len(col)):
    test.append(np.argmax(col[i]))

# Predicción red 2
prediccion2 = model2.predict(col)
pred2 = list()
for i in range(len(prediccion2)):
    pred2.append(np.argmax(prediccion2[i]))
col_test = [row[1] for row in secuencia_validacion]
ohe = OneHotEncoder()
col_reshaped_test = np.array(col_test).reshape(-1,1)
col = ohe.fit_transform(col_reshaped_test).toarray()
test2 = list()
for i in range(len(col)):
    test2.append(np.argmax(col[i]))

# Predicción red 3
prediccion3 = model3.predict(col)
pred3 = list()
for i in range(len(prediccion3)):
    pred3.append(np.argmax(prediccion3[i]))
col_test = [row[2] for row in secuencia_validacion]
ohe = OneHotEncoder()
col_reshaped_test = np.array(col_test).reshape(-1,1)
col = ohe.fit_transform(col_reshaped_test).toarray()
test3 = list()
for i in range(len(col)):
    test3.append(np.argmax(col[i]))
```

```
# Predicción red 4
prediccion4 = model4.predict(col)
pred4 = list()
for i in range(len(prediccion4)):
    pred4.append(np.argmax(prediccion4[i]))
col_test = [row[3] for row in secuencia_validacion]
ohe = OneHotEncoder()
col_reshaped_test = np.array(col_test).reshape(-1,1)
col = ohe.fit_transform(col_reshaped_test).toarray()
test4 = list()
for i in range(len(col)):
    test4.append(np.argmax(col[i]))

# Predicción red 5
prediccion5 = model5.predict(col)
pred5 = list()
for i in range(len(prediccion5)):
    pred5.append(np.argmax(prediccion5[i]))
col_test = [row[4] for row in secuencia_validacion]
ohe = OneHotEncoder()
col_reshaped_test = np.array(col_test).reshape(-1,1)
col = ohe.fit_transform(col_reshaped_test).toarray()
test5 = list()
for i in range(len(col)):
    test5.append(np.argmax(col[i]))

# Predicción red 6
prediccion6 = model6.predict(col)
pred6 = list()
for i in range(len(prediccion6)):
    pred6.append(np.argmax(prediccion6[i]))
col_test = [row[5] for row in secuencia_validacion]
ohe = OneHotEncoder()
col_reshaped_test = np.array(col_test).reshape(-1,1)
col = ohe.fit_transform(col_reshaped_test).toarray()
test6 = list()
for i in range(len(col)):
    test6.append(np.argmax(col[i]))

# Precisión redes
from sklearn.metrics import accuracy_score
a = accuracy_score(pred ,test)
print('Accuracy of net 1 is:', a*100)
a2 = accuracy_score(pred2 ,test2)
print('Accuracy of net 2 is:', a2*100)
a3 = accuracy_score(pred3 ,test3)
print('Accuracy of net 3 is:', a3*100)
a4 = accuracy_score(pred4 ,test4)
print('Accuracy of net 4 is:', a4*100)
a5 = accuracy_score(pred5 ,test5)
print('Accuracy of net 5 is:', a5*100)
a6 = accuracy_score(pred6 ,test6)
print('Accuracy of net 6 is:', a6*100)
```

Para este caso, como se trata de una clasificación, se ha tenido que tratar los datos para que la red los reciba como un vector de 8 números binarios, uno por cada posible valor de la salida. Para ello, se ha utilizado la función *OneHotEnconder* de la librería *sklearn*. Esta función se encarga de convertir un número entero en un vector de números que contiene 0 en todas las posiciones y un 1 en la posición indicada por el número entero. Así, se obtiene de la salida de cada red tomando como solución el índice con el valor de probabilidad más alto.

Para 6 redes independientes, se ha considerado entrenar la red con los dos conjuntos de datos que se tienen: los 266000 datos que corresponden con toda la cinemática inversa del robot, y los 31400 que se obtienen tras eliminar las duplicidades

del primer caso. Esto se debe a que las salidas, aunque no sean directamente dependientes unas de otras, sí existe cierta necesidad de que, para los resultados duplicados, las salidas posibles deban corresponderse entre ellas, es decir, si se toma como referencia la entrada X, y esta entrada tiene 10 soluciones posibles, las redes independientes puede que resulten dar, por ejemplo, la primera red dar la solución número 2, la segunda red dar la solución, número 5, etc. Por ello, si cada entrada solamente tiene una salida posible, las redes independientes sí deben dar la solución correcta, ya que es única (no existen duplicidades).

Los resultados de precisión para el caso en que los datos constituyen 266000 valores diferentes, cuya precisión se espera que sea peor que la segunda variante (31400 datos) son los siguientes:

- Red 1: 26.70 %
- Red 2: 18.69 %
- Red 3: 19.82 %
- Red 4: 20.49 %
- Red 5: 18.99 %
- Red 6: 26.86 %

Para entrenar el segundo caso, simplemente se han modificado las entradas y salidas y los datos de validación (estos últimos pasaron de ser 50000 en el primer caso a 4000 en el segundo). Los resultados de precisión para este caso son los siguientes:

- Red 1: 34.38 %
- Red 2: 25.96 %
- Red 3: 27.32 %
- Red 4: 30.05 %
- Red 5: 24.71 %
- Red 6: 16.36 %

Como se puede apreciar, salvo la red 6, todas han conseguido una mayor precisión que sus correspondientes del caso anterior. Era previsible, puesto que, en este segundo caso, la dependencia entre los valores de salida es mínima, a diferencia del caso anterior, pero aun así el resultado está lejos de ser correcto.

La baja precisión de la red 6 puede deberse simplemente a que en este caso concreto se han tomado unos datos de validación que la perjudican, porque esta está diseñada exactamente igual que las otras, por lo que debería tener una precisión mayor a su homónima del caso anterior.

Se ha probado a cambiar los parámetros de las redes (número de capas, neuronas, métodos de entrenamiento, optimizadores, etc.) y a volver a entrenarlas, y los resultados han sido iguales o peores, por lo que se ha decidido no exponerlos. En el siguiente apartado se plantea una forma diferente de resolver este problema.

El segundo método que se ha planteado con la expectativa de obtener un buen resultado consiste en entrenar una red de tal forma que sus salidas sean las mismas que en el apartado anterior (6 redes x 8 salidas = 48 clases) pero que estas se generen solo mediante una red, lo que hace que el problema que tienen las 6 redes independientes del apartado anterior, que es que las salidas no son 100% independientes unas de otras, se solucione debido a que, al ser solamente una red, las relaciones de dependencia sí se tienen en cuenta. Por ello, este caso sí tiene sentido que se pueda resolver entrenando con los 266000 datos; también podría entrenarse eliminando las duplicidades, pero en este caso solamente afectaría al número de datos de entrenamiento, por lo que probablemente solo empeore el resultado.

El código empleado para este apartado es el que se muestra a continuación:

```
import numpy as np
from keras.models import Sequential
from keras.layers.core import Dense
import random

nube = np.loadtxt('./nube_tot.txt',delimiter=",")
secuencia = np.loadtxt('./secuencia_clas.txt',delimiter=",")

nube = nube.astype(float)
secuencia = secuencia.astype(int)

v_elim = random.sample(range(266176), 10000)

nube_validacion = []
secuencia_validacion = []
for i in v_elim:
    nube_validacion.append(nube[i, 0:])
    secuencia_validacion.append(secuencia[i, 0:])

nube_validacion = np.array(nube_validacion, dtype=np.float64 )
secuencia_validacion = np.array(secuencia_validacion, dtype=np.int64)

nube = np.delete(nube,(v_elim),axis=0)
secuencia = np.delete(secuencia,(v_elim),axis=0)

```

```
model = Sequential()
model.add(Dense(20,input_dim=3,activation='relu'))
model.add(Dense(50,input_dim=20,activation='relu'))
model.add(Dense(20,input_dim=50,activation='relu'))
model.add(Dense(6,input_dim=20,activation='relu'))
model.add(Dense(48,activation='softmax'))
model.summary()
model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
history = model.fit(nube,secuencia,epochs=100)

seq_pred = model.predict(nube_validacion)
col_seq = [row[0:7] for row in seq_pred]
pred = list()
for i in range(len(col_seq)):
    pred.append(np.argmax(col_seq[i]))

col_test = [row[0:7] for row in secuencia_validacion]
test = list()
for i in range(len(col_test)):
    test.append(np.argmax(col_test[i]))

col_seq = [row[8:15] for row in seq_pred]
pred2 = list()
for i in range(len(col_seq)):
    pred.append(np.argmax(col_seq[i]))

col_test = [row[8:15] for row in secuencia_validacion]
test2 = list()
for i in range(len(col_test)):
    test.append(np.argmax(col_test[i]))

col_seq = [row[16:23] for row in seq_pred]
pred3 = list()
for i in range(len(col_seq)):
    pred.append(np.argmax(col_seq[i]))

col_test = [row[16:23] for row in secuencia_validacion]
test3 = list()
for i in range(len(col_test)):
    test.append(np.argmax(col_test[i]))

col_seq = [row[24:31] for row in seq_pred]
pred4 = list()
for i in range(len(col_seq)):
    pred.append(np.argmax(col_seq[i]))

col_test = [row[24:31] for row in secuencia_validacion]
test4 = list()
for i in range(len(col_test)):
    test.append(np.argmax(col_test[i]))

col_seq = [row[31:39] for row in seq_pred]
pred5 = list()
for i in range(len(col_seq)):
    pred.append(np.argmax(col_seq[i]))

col_test = [row[31:39] for row in secuencia_validacion]
test5 = list()
for i in range(len(col_test)):
    test.append(np.argmax(col_test[i]))

col_seq = [row[40:47] for row in seq_pred]
pred6 = list()
for i in range(len(col_seq)):
    pred.append(np.argmax(col_seq[i]))

col_test = [row[40:47] for row in secuencia_validacion]
test6 = list()
for i in range(len(col_test)):
    test.append(np.argmax(col_test[i]))
```

```
from sklearn.metrics import accuracy_score
a = accuracy_score(pred,test)
print('Acuracy is: ',a*100)
a2 = accuracy_score(pred,test)
print('Acuracy is: ',a2*100)
a3 = accuracy_score(pred,test)
print('Acuracy is: ',a3*100)
a4 = accuracy_score(pred,test)
print('Acuracy is: ',a4*100)
a5 = accuracy_score(pred,test)
print('Acuracy is: ',a5*100)
a6 = accuracy_score(pred,test)
print('Acuracy is: ',a5*100)
```

Como se ha mencionado, este diseño se ha basado en la programación de 48 clases diferentes. Estas 48 clases se subdividen en 6 grupos de 8 clases, al igual que en el caso anterior, 6 grupos porque son 6 salidas, y 8 clases por grupo porque la salida puede tomar 8 valores diferentes. Este diseño parecía que sí obtendría resultados al menos coherentes, por ello se ha decidido entrenar dicha red con 100 iteraciones, tardando algo más de una hora cada vez que se quería probar algo diferente.

A pesar de lo alentador que parecía este diseño, los números obtenidos de esta red no han sido para nada buenos. En concreto, este caso presentaba durante las iteraciones valores de la función de pérdidas (*loss*) del orden de 10^{16}, siendo dicho valor más correcto cuanto más bajo es. La precisión final de esta red ha resultado ser del 22.755 %, un valor similar al promedio de las precisiones del mismo caso en el apartado anterior.

Viendo estos resultados tan desalentadores, se ha investigado un poco acerca de los optimizadores del entrenamiento de este tipo de redes neuronales. Para el caso que se acaba de exponer, se ha empleado el optimizador "*adam*", y viendo los resultados, se ha decidido probar a utilizar otro. Se ha escogido para ello el optimizador *Stochastic Gradient Descent* ("*SGD*"), pero los resultados han sido similares: la función de pérdidas directamente tomaba el valor de *nan*, aunque la precisión calculada ha resultado ser mejor, siendo de un 25.027 %.

Como se ha podido observar, los resultados obtenidos en los casos expuestos han dejado mucho que desear. Además de los casos expuestos, se han probado distintas configuraciones de redes, con más y menos neuronas y con más y menos capas ocultas, y los resultados han sido similares. Realmente, se esperaba que con estas configuraciones los datos mejoraran algún ápice, pero no se ha encontrado la manera de hacerlo posible.

Analizando los datos obtenidos y los principios de diseño adoptados, se ha concluido que el método empleado en el apartado 3 tiene un mayor futuro para la clasificación de los datos cuando se eliminan las duplicidades, es decir, cuando los datos

de entrada y salida constituyen soluciones únicas para la cinemática inversa; sin embargo, el segundo algoritmo parece más óptimo para la clasificación de la nube completa de datos. Esto sería así en el hipotético caso de que los resultados fueran buenos, pero tal y como están, estos algoritmos no son para nada adecuados para esta aplicación.

Como conclusión final, se ha considerado que las redes neuronales de tipo clasificación no son adecuadas para el desarrollo de la cinemática inversa de robots de este tipo, o al menos no se ha encontrado la forma de que lo sea. Probablemente, al tratarse de un robot binario, sea más óptimo desarrollar un algoritmo genético para el tratamiento de este problema, ya que los valores de los actuadores del mismo siempre van a ser binarios (1 o 0).

4.1.2. Diseño de un agoritmo genético.

Como se comentó en la introducción de la sección 4.1, otro algoritmo basado en inteligencia artificial que se va a desarrollar para esta aplicación es el diseño de un algoritmo genético que sea capaz de encontrar una solución al problema de aumentar el número de ciclos de movimiento del robot, es decir, al problema producido por la solución de "fuerza bruta" para un gran número de datos.

Como definición general, los algoritmos genéticos son métodos adaptativos que permiten resolver problemas de búsqueda y optimización, y están basados en el proceso genético de los seres vivos (evolución), es decir, en la selección natural y supervivencia de los más fuertes. En otras palabras, los algoritmos genéticos pretenden realizar una analogía directa del principio de la selección natural: los individuos más fuertes o mejor dotados tienen una mayor capacidad de reproducirse, lo que produciría un mayor número de descendientes con sus genes.

El algoritmo desarrollado en esta sección se ha reflejado en el Anexo 5. En concreto, se han desarrollado 3 casos diferentes, cada uno compuesto por 3 *scripts* diferentes y homónimos entre los distintos casos. Dichos *scripts* son: **ws_alg_gen.m**, **child_generation.m** y **reproduccion.m**.

La única función que se mantiene invariante en los 3 casos es **reproduccion.m**, por ello será la primera en ser explicada, y posteriormente se explicarán las otras dos para cada caso independiente. Dicha función recibe como parámetros de entrada el número total de la población (*n_poblacion*), el número de ciclos de movimiento del robot (*N*) para

saber el tamaño de las entradas y las salidas, y los dos vectores que representan los individuos que van a reproducirse (*parent1* y *parent2*). Mediante estos datos, esta función va a devolver como salida el resultado de la reproducción de los individuos de entrada (*child1* y *child2*).

```
 3 -  %   child1 = [parent1(1:3*N) parent2(3*N+1:6*N)];
 4 -  %   child2 = [parent2(1:3*N) parent1(3*N+1:6*N)];
 5 -
 6 -    child1 = zeros(1,6*N);
 7 -    child2 = zeros(1,6*N);
 8 -    for i = 1:6:6*N
 9 -      child1(i:i+5) = [parent1(i:i+2) parent2(i+3:i+5)];
10 -      child2(i:i+5) = [parent2(i:i+2) parent1(i+3:i+5)];
11 -    end
12 -
13 -  %   Tasa de mutación 1/(landa^0.9318*^L^0.4535)
14 -    tasa = 1/(n_poblacion^0.9318*(6*N)^0.4535);
15 -    if rand >= 1-tasa
16 -      mutacion = randi(6*N);
17 -      child1(mutacion) = not(child1(mutacion));
18 -    end
19 -    if rand >= 1-tasa
20 -      mutacion = randi(6*N);
21 -      child2(mutacion) = not(child2(mutacion));
22 -    end
```

El primer paso es hacer la reproducción de los individuos. Esto puede hacerse de dos formas; la primera, comentada en las líneas 3-4, consiste simplemente en dividir ambos vectores por la mitad y combinar la primera mitad de uno con la segunda mitad del otro y viceversa; por otro lado, la segunda forma consiste en combinar ambos vectores según sus índices, es decir, se combinan los términos en posiciones pares del primer vector con los términos en posiciones impares del segundo y viceversa (líneas 6-11).

Finalmente, se va a aplicar una pequeña probabilidad de que uno de los descendientes sufra una mutación en una de sus características. La probabilidad de mutación se ha calculado mediante la ecuación de Schaffer, siendo λ el número de individuos, es decir, *n_poblacion*, y l el número de bits que define cada individuo, es decir, *N*.

$$\textit{Ecuación de Schaffer} \quad \rightarrow \quad p_{mutation} = \frac{1}{\lambda^{0.9318} l^{0.4535}}$$

Una vez se calcula dicha ecuación (línea 14), se aplica el cálculo de probabilidad de mutación para el primer nuevo individuo (líneas 15-18) y a continuación para el segundo (líneas 19-22). En ambos casos, si el cálculo de probabilidad se cumple, se toma un término aleatorio del vector y se invierte.

En lo que respecta a los dos posibles métodos de reproducción mencionados, ambos se han empleado para tomar resultados, y se concretará más sobre su eficacia cuando se muestren los mismos.

A continuación, va a explicarse el primer caso desarrollado para este apartado. Los *scripts* que involucran a este caso son los que tienen un (1) precediendo al nombre del *script* en el Anexo 5. Este caso se basa en el principio de selección natural propiamente dicho, es decir, se calcula cuáles son los individuos más favorables al caso y estos son los que se reproducen. Una vez reproducidos, los descendientes sustituyen a los progenitores dentro de la población y se procede a calcular la siguiente generación. Para ello, en primer lugar, se explicará la función **child_generation.m**, y posteriormente se procederá a explicar el algoritmo en sí, desarrollado en **ws_alg_gen.m**.

La función **child_generation.m** recibe como entradas las siguientes variables: la población completa (*pobl*), la celda que almacena las 8 configuraciones posibles del robot (*T*), el número de ciclos de movimiento del robot (*N*), la posición y orientación objetivo del robot (*P_ref* y *phi_ref*) y el número total de individuos (*n_poblacion*).

```
 3 -   pobl_enteros = zeros(n_poblacion,2*N);
 4 -   contador = 0;
 5 -   for i = 1:3:6*N
 6 -     contador = contador+1;
 7 -     pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
 8 -   end
 9 -   pobl_enteros = pobl_enteros+1;
10 -
11 -   distancias = zeros(n_poblacion,1);
12 -   for l = 1:n_poblacion
13 -     secuencia_encontrada = pobl_enteros(l,:);
14 -     T_encontrado = eye(3);
15 -     for k=1:N
16 -       i = secuencia_encontrada(k);
17 -       j = secuencia_encontrada(k+N);
18 -       if i==-1 || j==-1
19 -         continue
20 -       end
21 -       T_encontrado = T_encontrado*T{i}/T{j};
22 -     end
23 -     v_entrada = [P_ref phi_ref];
24 -     P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
25 -     phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
26 -     v_encontrada = [P_encontrada phi_encontrada];
27 -     ang_dif = abs(v_entrada(3)-v_encontrada(3));
28 -     if 2*pi-ang_dif < ang_dif
29 -       ang_dif = 2*pi-ang_dif;
30 -     end
31 -     v_norm=[v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2) -150/(-1.8379+log(ang_dif))];
32 -     distancias(l) = norm(v_norm);
33 -   end
```

La definición de los vectores que representan a los individuos se explicarán cuando se desarrolle el algoritmo completo, en este momento solamente se debe tener en cuenta que la población por defecto se compone de un vector de 6*N bits, de los que cada 3 bits representan un entero entre 0 y 7; como el cálculo de la posición y orientación necesita un vector de 2*N enteros entre 1 y 8, se va a realizar la conversión, primero, de binario a decimal (líneas 4-8) y a continuación se va a sumar 1 a cada término de la matriz que almacena la población (línea 9). Con la población convertida a enteros, se va a proceder a calcular el vector que va a almacenar el valor que determina cuán favorable es el individuo para el objetivo impuesto.

Para calcular dicho vector, se debe tomar cada individuo en forma de vector de enteros y calcular a qué pose corresponde dicha secuencia de enteros (cada uno representando una configuración que el robot debe adoptar para moverse), como puede verse en las líneas 13-25. A continuación, se pretende calcular dicho vector (*distancias*), que se compone por las normas de cada individuo (línea 32) de un vector de 3 componentes calculado en la línea 31. Las tres componentes de dicho vector son, respectivamente, posición en X del punto correspondiente al individuo, posición en Y del mismo, y una ponderación de la orientación correspondiente a dicho punto. La ponderación pretende permitir comparar la orientación objetivo con la correspondiente al individuo con una orden de magnitud más similar a la de la posición, es decir, que, en lugar de simplemente restar las coordenadas de posición y orientación, las cuales podrían ser las primeras del orden de 100 y las segundas del orden de 1, se pretende que ambas tengan una magnitud similar para que, a la hora de hacer la norma del vector, ambas (posición y orientación) influyan lo mismo. Dicha ponderación se calcula tomando la diferencia absoluta entre las orientaciones que se desean comparar (líneas 27-30) y se hace el cálculo de la ponderación en escala logarítmica (última componente del vector de la línea 31). Más concretamente, los valores máximos y mínimos de la ponderación que se obtienen son los siguientes: para una diferencia angular de 0 rad, la ponderación calculada es 0, y para una diferencia angular de π rad, la ponderación calculada es de 216.3971, y los valores intermedios se encuentran entre estos valores.

```
35 -    % Se reproduce el 25% de la población
36 -    tasa_prom = 0.25;
37 -    n_reproductores = round(n_poblacion*tasa_prom);
38 -    n_reproductores = floor(n_reproductores/2)*2;
39 -    if min(distancias) < 10
40 -      flag = 1;
41 -      child = zeros(n_reproductores,6*N);
```

```
42 -      i_child = 1;
43 -    else
44 -      flag = 0;
45 -      d_max = max(distancias)+10;
46 -      v_pond = round((d_max-distancias)/10);
47 -      v_pond_acumulado = cumsum(v_pond);
48 -      ganadores = randi(v_pond_acumulado(end)-1,[n_reproductores,1]);
49 -      indices_reproductores = zeros(n_reproductores,1);
50 -      for i = 1:n_reproductores
51 -        n = ganadores(i);
52 -        v_pond_acumulado_aux = abs(v_pond_acumulado-n);
53 -        [~,indice] = min(v_pond_acumulado_aux);
54 -        if v_pond_acumulado(indice) >= n
55 -          indices_reproductores(i) = indice;
56 -        else
57 -          indices_reproductores(i) = indice+1;
58 -        end
59 -      end
60 -
61 -      child = zeros(n_reproductores,6*N);
62 -      i_child = indices_reproductores;
63 -      for i = 1:n_reproductores/2
64 -        indice_parent1 = indices_reproductores(i);
65 -        parent1 = pobl(indice_parent1,:);
66 -        indice_parent2 = indices_reproductores(i+n_reproductores/2);
67 -        parent2 = pobl(indice_parent2,:);
68 -        [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion);
69 -        child(i,:) = child1;
70 -        child(i+n_reproductores/2,:) = child2;
71 -      end
72 -    end
```

Lo primero que se va a explicar son las líneas 39-42, y estas se emplean para finalizar el algoritmo una vez se ha encontrado un punto que cumple con una norma pequeña, es decir, cuando un individuo generado antes de finalizar el algoritmo es suficientemente bueno, se toma como solución final y se detiene el proceso. Cabe destacar que se han calculado resultados con y sin esta interrupción, ya que esta no está totalmente ligada al principio de los algoritmos genéticos, se ha añadido para mejorar generalmente los resultados.

Una vez se tiene el vector *distancias*, se debe realizar la reproducción de los individuos que corresponda, en este caso, un 25% de la población total (líneas 36-38); al necesitar que los reproductores sean pares, en caso de que sean impares, se va a restar un reproductor (línea 38). Una vez se tiene el número de individuos que se van a reproducir, se debe seleccionar cuáles lo van a hacer, para lo que se va a aplicar un sistema de ruleta con diferente probabilidad, es decir, todos los individuos van a tener la posibilidad de reproducirse, pero se les dará mayor probabilidad a los más favorables. Para implementar dicho sistema, se va a seguir el principio que se muestra en la Figura 19.

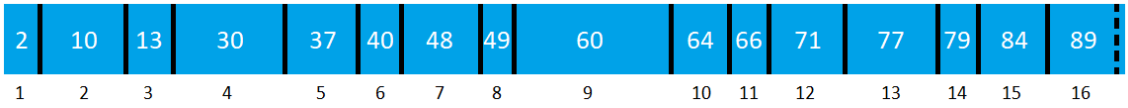

Figura 19: Representación de la ruleta de probabilidad.

El principio a seguir es el siguiente: se debe calcular el tamaño de las celdas mostradas en la Figura 19, para lo que se toma la distancia máxima del vector *distancias* y se le suma 10 (esta suma se hace para que el individuo más desfavorable también tenga posibilidad de reproducirse, línea 45); a continuación, se calcula el vector que almacena el tamaño de las celdas del vector de la Figura 19 (*v_pond*, línea 46) y con dicho vector se calcula su acumulado (*v_pond_acumulado*, línea 47) para que este represente lo mejor posible a la Figura 19, ya que como puede apreciarse esta figura representa una acumulación de números que aumenta más o menos dependiendo del tamaño de la celda; finalmente, se deben generar aleatoriamente los reproductores que corresponda, definida la cantidad en la línea 38, y que se explicará mediante un ejemplo.

Dicho ejemplo es el siguiente: se genera un número aleatorio entre 1 y el último valor del vector de acumulados, que es el tamaño total del vector de la Figura 19, y este resulta ser el 55; dicho número, como puede verse, pertenece a la celda 9, ya que es menor que el valor acumulado correspondiente a dicha celda (60) y mayor al acumulado correspondiente a la celda anterior (49). De la misma forma, se generan los *n_reproductores* individuos que se van a reproducir, definiendo los valores que caigan justo en los cambios de celda como correspondientes a la celda anterior (líneas 48-59).

Además, esta función debe calcular, a partir de los individuos seleccionados, sus descendientes, para lo que se debe aplicar la función **reproduccion.m** entre cada dos individuos (líneas 61-70).

Una vez explicada la función **child_generation.m**, se va a proceder a explicar el algoritmo desarrollado en el primer caso de este apartado. Este caso se basa en seleccionar a los individuos a reproducir mediante un sistema de ruleta con diferente probabilidad para cada individuo, explicado en el desarrollo de la función **child_generation.m**. A continuación, se va a explicar dicho algoritmo desarrollado en el *script* **ws_alg_gen.m**.

```
1 - clear all
2 -
3 - N = 3;
4 - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
5 - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
6 - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,21.95478428];
```

```
 7 -
 8 - for i=1:8
 9 -   T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
10 - end
11 -
12 - P_ref = [50 120];
13 - phi_ref = pi/4;
14 - n_poblacion = 500/64^(3-N);
15 - pobl = round(rand(n_poblacion,6*N));
16 -
17 - for i = 1:1000
18 -   [child,i_child,flag] = child_generation(pobl,T,N,P_ref,phi_ref,n_poblacion);
19 -   if flag == 1
20 -     break
21 -   end
22 -   pobl(i_child,:) = child;
23 - end
```

Las primeras líneas se emplean para calcular las variables necesarias para el algoritmo, todas ellas calculadas en algoritmos previos (líneas 3-13). El siguiente paso es el cálculo de la población inicial, y esta se ha definido como 500 individuos para un espacio de trabajo de 3 ciclos (266000 puntos), aproximadamente un 0.19%, y se ha hecho que sea proporcional dependiendo del número de ciclos (línea 14). Una vez calculada la población total (*n_poblacion*), se calcula la población en sí. Como se ha comentado, cada índice se representa mediante 3 bits, por lo que cada individuo debe tener 6**N* bits, ya que cada ciclo son 2 índices y cada índice son 3 bits; únicamente resta generar las cadenas de bits, que se hace redondeando el resultado de rand, ya que como esta última da un número aleatorio entre 0 y 1, al redondearla dará un 0 o un 1 (línea 15).

El siguiente paso es la aplicación del algoritmo desarrollado en **child_generation.m**, y se ha definido por defecto que se van a calcular 1000 generaciones de la población, es decir, la población se va a reproducir 1000 veces (línea 17). Como se ha comentado anteriormente, se va a implementar un sistema de interrupción si se encuentra un individuo suficientemente adecuado al objetivo, lo que interrumpirá el bucle (líneas 19-21). Además, como se ha mencionado al inicio de la explicación de este caso, en este se generan los descendientes y estos sustituyen a los individuos progenitores (línea 22).

```
25 - pobl_enteros = zeros(n_poblacion,2*N);
26 - contador = 0;
27 - for i = 1:3:6*N
28 -   contador = contador+1;
29 -   pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
30 - end
31 - pobl_enteros = pobl_enteros+1;
32 -
33 - distancias = zeros(n_poblacion,1);
```

```
34 - for l = 1:n_poblacion
35 -   secuencia_encontrada = pobl_enteros(l,:);
36 -   T_encontrado = eye(3);
37 -   for k=1:N
38 -     i = secuencia_encontrada(k);
39 -     j = secuencia_encontrada(k+N);
40 -     if i==-1 || j==-1
41 -       continue
42 -     end
43 -     T_encontrado = T_encontrado*T{i}/T{j};
44 -   end
45 -   v_entrada = [P_ref phi_ref];
46 -   P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
47 -   phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
48 -   v_encontrada = [P_encontrada phi_encontrada];
49 -   ang_dif = abs(v_entrada(3)-v_encontrada(3));
50 -   if 2*pi-ang_dif < ang_dif
51 -     ang_dif = 2*pi-ang_dif;
52 -   end
53 -   v_norm=[v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2) -150/(-1.8379+log(ang_dif))];
54 -   distancias(l) = norm(v_norm);
55 - end
56 -
57 - [~,ind_minimo] = min(distancias);
58 -
59 - sol_final = pobl(ind_minimo,:);
60 - secuencia_encontrada = zeros(1,2*N);
61 - contador = 0;
62 - for i = 1:3:6*N
63 -   contador = contador+1;
64 -   n = bi2de(sol_final(i:i+2));
65 -   secuencia_encontrada(contador) = n;
66 - end
67 - secuencia_encontrada = secuencia_encontrada+1;
68 - T_encontrado = eye(3);
69 - for k=1:N
70 -   i = secuencia_encontrada(k);
71 -   j = secuencia_encontrada(k+N);
72 -   if i==-1 || j==-1
73 -     continue
74 -   end
75 -   T_encontrado = T_encontrado*T{i}/T{j};
76 - end
77 -
78 - P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)]
79 - phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1))
```

Finalmente, se debe buscar, de toda la población final, al individuo más adecuado para el objetivo, para lo que se calcula el vector *distancias* de la misma forma que se hizo en **child_generation.m** (línea 25-55). A continuación, se busca el mínimo de todos ellos (línea 57) y se calcula la solución que obtendría dicho individuo (líneas 59-79), que sería la solución final de este primer caso.

Finalmente, solamente resta presentar los resultados obtenidos por este algoritmo, los cuales van a ser tanto resultados de tiempo como numéricos, con y sin la interrupción previamente explicada. En primer lugar, se van a mostrar los resultados de tiempo

obtenidos para el algoritmo con la interrupción habilitada, por lo que se va a ejecutar el algoritmo 20 veces (10 con la reproducción a la entre las mitades de los individuos y otras 10 con la reproducción mediante cambios de índices pares por impares) y se van a representar los tiempos en la Tabla 1.

Como puede apreciarse, la media total de cada posibilidad es muy similar. El primer caso es ligeramente más rápido que el segundo, por lo que se puede afirmar que ambos métodos son similares en cuanto a rendimiento, aunque el de reproducción mediante combinación de las mitades de cada individuo es algo más eficiente.

Tabla 1: Resultados temporales para N=3 con interrupción.

Reproducción por mitades	Reproducción por combinación de índices pares con impares
6,4841 s	1,0069 s
0,4762 s	1,0827 s
11,0297 s	18,0944 s
1,2166 s	1,8376 s
1,5225 s	6,5922 s
0,9377 s	10,0426 s
0,3871 s	0,456 s
2,0122 s	0,3565 s
0,8356 s	2,512 s
20,2249 s	0,7907 s
Media: 4,5127 s	Media: 4,2772 s

Por otra parte, si se retira la interrupción, el algoritmo tarda el mismo tiempo en converger aproximadamente en todos los intentos que se han hecho, por lo que se van a exponer los tiempos solamente de un intento para cada caso de reproducción:

Reproducción por mitades = 36.5273 s

Reproducción por combinación de índices pares con impares = 39.7708 s

A continuación, van a mostrarse los resultados obtenidos en posición y orientación de 5 intentos realizados para cada combinación posible de lo mostrado hasta ahora: una con y otra sin interrupción para el caso de reproducción por mitades, y una con y otra sin interrupción para el caso de reproducción por combinación de índices pares con impares. Dichos resultados se muestran en la Tabla 2, y en todos ellos se ha establecido como objetivo el punto [50,120] mm con orientación $\pi/4$, para un número de ciclos de N = 3.

Tabla 2: Resultados numéricos para N=3 para el primer caso, donde los reproductores se eligen por el sistema de ruleta.

Con interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
49,0982	121,2950	0,7854	51,0508	122,1038	0,7854
57,4813	115,1996	0,7854	43,9096	122,4388	0,7854
57,4813	116,8172	0,7854	46,5731	127,8639	0,7854
46,5731	116,0084	0,7854	57,4813	116,8172	0,7854
51,0508	114,1537	0,7854	46,5731	116,0084	0,7854
Sin interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
43,9096	122,4388	0,7854	57,4813	116,8172	0,7854
46,5731	127,8639	0,7854	66,5752	114,0557	0,7854
35,5265	114,0557	0,7854	15,5244	96,8151	0,7854
51,0508	94,9604	0,7854	46,5731	116,0084	0,7854
51,0508	101,2929	0,7854	33,5738	108,5321	0,7854

Finalmente, con el fin de conocer las limitaciones computacionales de este algoritmo, se ha intentado hacer que este calcule unos casos más desfavorables. En primer lugar, se ha calculado para $N = 4$ un intento, en lugar de 5. El hacer solamente un intento se debe a que ejecutar este algoritmo para $N = 4$ el tiempo de computación es de entre 40 y 50 minutos si no salta la interrupción, por lo que se ha decidido no hacer más de un intento por cada caso. Estos resultados se muestran en la Tabla 3, con objetivo [150,225] en posición y $\pi/4$ en orientación.

Tabla 3: Resultados numéricos para N=4 con interrupción para el primer caso, donde los reproductores se eligen por el sistema de ruleta.

Con interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
157,6303	220,7331	0,7854	157,6303	220,7331	0,7854
Sin interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
142,1059	220,6351	0,7854	153,1525	203,3945	0,7854

Los últimos resultados que se van a mostrar van a ser, al igual que en el caso anterior, para mostrar las limitaciones del algoritmo, por lo que se va a ejecutar el mismo para 5 ciclos de movimiento, pero solamente una generación, por lo que se podrá extrapolar el tiempo obtenido al número de generaciones que se quiera implementar.

Dichos resultados se muestran a continuación. Cabe destacar que este resultado ha sido obtenido mediante la reproducción por combinación de índices pares e impares, y se ha suprimido la interrupción para conocer el tiempo de ejecución total.

Tiempo de ejecución para una generación de 5 ciclos de movimiento = 1752.054 s = 29.20 min

Con este resultado obtenido para una generación, se puede extrapolar al caso expuesto para 3 y 4 ciclos, es decir, obtener el tiempo de ejecución para el caso de 1000 generaciones, el cual serían aproximadamente 486.68 horas. Dicho resultado, a pesar de demostrar que sería posible obtener un resultado para N=5, cosa que no se había conseguido hasta ahora, no es un resultado que pueda considerarse como bueno, ya que un tiempo de ejecución tan largo hace inviable a casi cualquier algoritmo.

A pesar de que los otros dos casos que se han comentado en el inicio de este apartado no mejoran, o al menos no una cantidad significativa, se van a exponer junto con los resultados obtenidos para N=3 ciclos de movimiento con el fin de mostrar su viabilidad junto a los obtenidos por el algoritmo ya presentado.

El segundo caso que se va a exponer pretende cambiar la forma en la que se eligen los reproductores. En este caso, en lugar de definir un número fijo de reproductores, se le va a dar a cada individuo la probabilidad individual de reproducirse, y esta será mayor o menor dependiendo de la adecuación de cada individuo, es decir, cuanto más adecuado sea el individuo, más posibilidad tendrá de reproducirse. La forma de hacerlo se explicará cuando se muestre el algoritmo desarrollado para ello, lo cual se hará a continuación.

El algoritmo en sí se ha desarrollado en el *script* **ws_alg_gen.m**, al igual que en el caso anterior, por ello, lo primero que se expondrán serán las subfunciones **reproduccion.m** y **child_generation.m**. La primera, como se ha comentado anteriormente, es invariante para los tres casos, por lo que no se volverá a profundizar en ella; sin embargo, la segunda sí que presenta cambios que afectan al desarrollo del algoritmo, por lo que se va a proceder a explicar dichos cambios.

Las primeras 33 líneas de esta función son idénticas a las de su homóloga del caso anterior, y, como recordatorio, estas se encargaban de calcular el vector que almacena las normas de los vectores de tres componentes (dos para posición y una para orientación) entre el objetivo y cada uno de los individuos. Con dicho vector, se pretende evaluar la idoneidad de cada individuo, dándole mayor o menor probabilidad de reproducirse.

```
35 -   dist_norm = distancias/(1.1*max(distancias));
36 -   counter = 0;
37 -   indices_reproductores = zeros(n_poblacion,1);
38 -   for i = 1:n_poblacion
39 -     if dist_norm(i) > rand
40 -       counter = counter+1;
41 -       indices_reproductores(counter) = i;
42 -     end
43 -   end
44 -   n_reproductores = floor(counter/2)*2;
45 -   indices_reproductores = indices_reproductores(1:n_reproductores,:);
46 -
47 -   if min(distancias) < 10
48 -     flag = 1;
49 -     child = zeros(n_reproductores,6*N);
50 -     i_child = 1;
51 -   else
52 -     child = zeros(n_reproductores,6*N);
53 -     i_child = indices_reproductores;
54 -     for i = 1:n_reproductores/2
55 -       indice_parent1 = indices_reproductores(i);
56 -       indice_parent2 = indices_reproductores(i+n_reproductores/2);
57 -       parent1 = pobl(indice_parent1,:);
58 -       parent2 = pobl(indice_parent2,:);
59 -       [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion);
60 -       child(i,:) = child1;
61 -       child(i+n_reproductores/2,:) = child2;
62 -     end
63 -   end
```

Para establecer la probabilidad individual de reproducción mencionada anteriormente, en primer lugar, se calcula un vector que surge de dividir todas las componentes del vector *distancias* entre un factor ligeramente superior que el máximo de dicho vector (línea 35). Con esto, se obtiene un vector con valores entre 0 y 1 (nunca habrá ningún 0 ni ningún 1 gracias a ese ligero aumento del denominador del cociente anterior) con los que se pueden comparar un lanzamiento de *rand* y determinar si se reproduce o no (línea 39). Con cada comparación verdadera, se añade un reproductor a la lista (líneas 38-43), aunque al tener que ser siempre pares los reproductores, se elimina al último encontrado en caso de que el número total sea impar (líneas 44-45).

Finalmente, tras establecer la condición de salida del bucle tal y como se hizo en el caso anterior (líneas 47-50), se realiza la reproducción de los individuos seleccionados aplicando la función **reproduccion.m** dos a dos como en el caso anterior (líneas 51-63), obteniendo a los individuos descendientes y devolviéndolos a la función principal (**ws_alg_gen.m**).

A continuación, se expondrán los resultados obtenidos de este caso para N=3 ciclos de movimiento, al igual que se ha hecho en el caso anterior. Dichos resultados se

muestran en la Tabla 4. Cabe destacar que el objetivo impuesto para estos resultados es el mismo que el caso anterior: [50, 120] en posición y π/4 en orientación.

Tabla 4: Resultados numéricos para N=3 del segundo caso, donde se da probabilidad de reproducción a todos los individuos.

Con interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
46.5731	116.0084	0.7854	43.9096	122.4388	0.7854
57.4813	116.8172	0.7854	51.0508	122.1038	0.7854
51.0508	122.1038	0.7854	49.0982	121.2950	0.7854
49.0982	121.2950	0.7854	46.5731	116.0084	0.7854
43.9096	122.4388	0.7854	51.0508	114.1537	0.7854
Sin interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
57.4813	115.1996	0.7854	35.5265	126.9165	0.7854
35.5265	86.9123	0.7854	49.0982	121.2950	0.7854
46.5731	116.0084	0.7854	51.0508	122.1038	0.7854
49.0982	121.2950	0.7854	46.5731	116.0084	0.7854
34.7176	116.0084	0.7854	51.0508	101.2929	0.7854

Finalmente, únicamente resta exponer el tercer caso correspondiente a este apartado y comentar los resultados obtenidos para los tres. Se ha decidido hacerlo de esta manera ya que, como los resultados son similares en los tres casos, se ha considerado más adecuado comentar los resultados de los tres casos juntos.

Por consiguiente, lo siguiente que se va a explicar serán los algoritmos desarrollados para este tercer caso, que son homólogos a los de los casos anteriores. De hecho, como se ha comentado, la función **reproduccion.m** es idéntica en los tres casos, por lo que no se va a explicar de nuevo. La siguiente función, al igual que en los casos anteriores, es **child_generation.m**, y en este caso, se muestra mucho más sencilla que en los casos anteriores, ya que para este no se ha establecido un criterio de selección de reproductores, sino que toda la población tiene la misma probabilidad de reproducirse, pero se ha hecho una selección de los individuos que se mantienen para la próxima generación, en lugar de simplemente sustituir a los progenitores por los descendientes al finalizar la reproducción, como en los casos anteriores.

```
3 -   % Se reproduce el 25% de la población
4 -   tasa_prom = 0.25;
5 -   n_reproductores = round(n_poblacion*tasa_prom);
6 -   n_reproductores = floor(n_reproductores/2)*2;
```

```
 7 -   indices_reproductores = randi(n_poblacion,[n_reproductores,1]);
 8 -
 9 -   child = zeros(n_reproductores,6*N);
10 -   for i = 1:n_reproductores/2
11 -     indice_parent1 = indices_reproductores(i);
12 -     parent1 = pobl(indice_parent1,:);
13 -     indice_parent2 = indices_reproductores(i+n_reproductores/2);
14 -     parent2 = pobl(indice_parent2,:);
15 -     [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion);
16 -     child(i,:) = child1;
17 -     child(i+n_reproductores/2,:) = child2;
18 -   end
```

En este algoritmo, se toma el 25% de la población (al igual que en el primer caso) de forma completamente aleatoria (líneas 4-7), siempre siendo un número par (línea 6), y se reproducen dos a dos aplicando la función **reproduccion.m** (líneas 9-18) y devolviendo al algoritmo principal los descendientes generados. Dicho algoritmo principal va a explicarse a continuación, y este sí que presenta más modificaciones con respecto a sus homólogos de los casos anteriores.

Las primeras líneas de este *script* (**ws_alg_gen.m**) sí son idénticas a las de los casos anteriores, y en ellas se declaran las variables iniciales, se definen la posición y la orientación objetivos y se genera la población inicial. Es dentro del bucle de las generaciones donde se encuentran los cambios principales de este algoritmo.

```
19 - distancias = 10000000;
20 - for a = 1:1000
21 -
22 -   if min(distancias) < 10
23 -     break
24 -   end
25 -   child = child_generation(pobl,N,n_poblacion);
26 -   pobl = [pobl;child];
27 -   pobl_enteros = zeros(size(pobl,1),2*N);
28 -   contador = 0;
29 -   for i = 1:3:6*N
30 -     contador = contador+1;
31 -     pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
32 -   end
33 -   pobl_enteros = pobl_enteros+1;
34 -
35 -   distancias = zeros(n_poblacion,1);
36 -   for l = 1:size(pobl_enteros,1)
37 -     secuencia_encontrada = pobl_enteros(l,:);
38 -     T_encontrado = eye(3);
39 -     for k=1:N
40 -       i = secuencia_encontrada(k);
41 -       j = secuencia_encontrada(k+N);
42 -       if i==-1 || j==-1
43 -         continue
44 -       end
45 -       T_encontrado = T_encontrado*T{i}/T{j};
46 -     end
47 -     v_entrada = [P_ref phi_ref];
```

```
48 -      P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
49 -      phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
50 -      v_encontrada = [P_encontrada phi_encontrada];
51 -      ang_dif = abs(v_entrada(3)-v_encontrada(3));
52 -      if 2*pi-ang_dif < ang_dif
53 -        ang_dif = 2*pi-ang_dif;
54 -      end
55 -      v_norm=[v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2) -150/(-1.8379+log(ang_dif))];
56 -      distancias(l) = norm(v_norm);
57 -    end
58 -
59 -    indices_max = zeros(size(pobl,1)-n_poblacion,1);
60 -    for j = 1:length(indices_max)
61 -      [~,imax] = max(distancias);
62 -      distancias(imax) = 0;
63 -      indices_max(j) = imax;
64 -    end
65 -    pobl(indices_max,:) = [];
66 -    distancias(indices_max) = [];
67 -
68 - end
```

El primer paso de cada generación es establecer la interrupción del bucle, al igual que en los casos anteriores. En caso de que no se interrumpa el bucle, lo siguiente es realizar la reproducción mediante la función **child_generation.m** (línea 25). Una vez se tienen los descendientes y la población inicial de la generación actual, se juntan ambas en una misma variable (línea 26), teniendo así una única población un 25% más grande que la inicial, de la cual se van a escoger los peores individuos (una cantidad correspondiente a ese 25%) y se van a eliminar de la población con el fin de que los mejores individuos sigan reproduciéndose. Para ello, se va a calcular el vector *distancias* de la misma forma que en los casos anteriores pero esta vez para la matriz que incluye tanto a la población inicial como a la nueva generación (líneas 27-57). Para finalizar cada generación, se debe hacer la selección comentada anteriormente, para lo que se ejecuta un bucle tantas veces como grande sea la matriz de individuos (línea 60), se busca en cada iteración el máximo valor del vector *distancias* (línea 61) y se almacena el índice correspondiente al individuo con dicho valor máximo (línea 63).

Una vez se tienen almacenados todos los índices correspondientes a los valores más grandes de distancias entre cada individuo y el objetivo, estos se utilizan para eliminar los individuos correspondientes a cada uno de ellos de la población total actual (línea 65) para que así la población acabe siendo del mismo tamaño que la original, antes de añadir la nueva generación, pero con individuos más adecuados para cumplir el objetivo impuesto.

```
70 - pobl_enteros = zeros(n_poblacion,2*N);
```

```
contador = 0;
for i = 1:3:6*N
  contador = contador+1;
  pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
end
pobl_enteros = pobl_enteros+1;

distancias = zeros(n_poblacion,1);
for l = 1:n_poblacion
  secuencia_encontrada = pobl_enteros(l,:);
  T_encontrado = eye(3);
  for k=1:N
    i = secuencia_encontrada(k);
    j = secuencia_encontrada(k+N);
    if i==-1 || j==-1
        continue
    end
    T_encontrado = T_encontrado*T{i}/T{j};
  end
  v_entrada = [P_ref phi_ref];
  P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
  phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
  v_encontrada = [P_encontrada phi_encontrada];
  ang_dif = abs(v_entrada(3)-v_encontrada(3));
  if 2*pi-ang_dif < ang_dif
    ang_dif = 2*pi-ang_dif;
  end
  v_norm=[v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2) -150/(-1.8379+log(ang_dif))];
  distancias(l) = norm(v_norm);
end

[~,ind_minimo] = min(distancias);

sol_final = pobl(ind_minimo,:);
secuencia_encontrada = zeros(1,2*N);
contador = 0;
for i = 1:3:6*N
  contador = contador+1;
  n = bi2de(sol_final(i:i+2));
  secuencia_encontrada(contador) = n;
end
secuencia_encontrada = secuencia_encontrada+1;
T_encontrado = eye(3);
for k=1:N
  i = secuencia_encontrada(k);
  j = secuencia_encontrada(k+N);
  if i==-1 || j==-1
    continue
  end
  T_encontrado = T_encontrado*T{i}/T{j};
end

P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)]
phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1))
```

Finalmente, una vez finalizado el bucle de generaciones, tanto por interrupción como por cumplimiento de las 1000 iteraciones, se debe buscar el mejor individuo de todos los generados en la última generación del bucle, para lo que se utiliza el mismo sistema que en los casos anteriores. Primero de todo, se calcula el vector *distancias* para

la última generación (líneas 70-100), de todos los individuos, se busca cuál tiene la mínima distancia (línea 102), se obtiene la solución correspondiente a este individuo (líneas 104-121), y se muestra por pantalla (líneas 123-124).

Al igual que en el caso anterior, se van a mostrar los resultados obtenidos para este caso con N=3 ciclos de movimiento. Dichos resultados se muestran en la Tabla 5. Cabe destacar que el objetivo impuesto para estos resultados es el mismo que el caso anterior: [50, 120] en posición y $\pi/4$ en orientación, como en los casos anteriores.

Tabla 5: Resultados numéricos para N=3 para el tercer caso, donde todos los individuos tienen la misma probabilidad de reproducirse, pero se selecciona a los mejores.

Con interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
46.5731	116.0084	0.7854	46.5731	109.6760	0.7854
49.0982	121.2950	0.7854	57.4813	115.1996	0.7854
34.7176	116.0084	0.7854	43.9096	122.4388	0.7854
46.5731	127.8639	0.7854	49.0982	121.2950	0.7854
51.0508	122.1038	0.7854	46.5731	127.8639	0.7854
Sin interrupción					
Reproducción por mitades			Reproducción por combinación de índices pares con impares		
x(mm)	y(mm)	phi(rad)	x(mm)	y(mm)	phi(rad)
43.9096	122.4388	0.7854	51.0508	114.1537	0.7854
49.0982	121.2950	0.7854	43.9096	122.4388	0.7854
35.5265	114.0557	0.7854	57.4813	116.8172	0.7854
46.5731	127.8639	0.7854	55.5286	130.4869	0.7854
46.5731	116.0084	0.7854	46.5731	104.1529	0.7854

Finalmente, se van a exponer las conclusiones alcanzadas sobre los algoritmos genéticos desarrollados a partir de los resultados obtenidos. Estos tres algoritmos comparten un mismo principio, y es el de implementar una interrupción que salte cuando se ha encontrado un individuo suficientemente adecuado, y a pesar de que este no es un principio propio de los algoritmos genéticos, este mejora en gran medida tanto los resultados obtenidos como los tiempos de ejecución, por ello se ha decidido implementarlo, y considerarlo como mejor opción, a pesar de que también se hayan expuesto los resultados sin aplicar la interrupción.

De los resultados propiamente dichos, se debe comentar que, en todos y cada uno de ellos, para los tres casos, se ha encontrado exactamente la orientación buscada, por lo que se puede afirmar que, si bien el principio adoptado para ponderar la orientación no se ha basado en ningún principio conocido y estudiado, sino que se ha ideado en concreto

para esta operación mediante prueba y error, este ha resultado ser efectivo para esta aplicación.

Hablando exclusivamente de la posición, se puede ver que las soluciones obtenidas son muy variadas, pero ninguna demasiado dispar. De hecho, el error máximo en posición obtenido de entre los tres casos ha sido de 36.1148 mm, y que, si bien no es un resultado bueno en absoluto, que este sea el valor máximo del error de entre 60 intentos (solamente se están comparando los correspondientes a N=3) indica que, a pesar de la aleatoriedad de este algoritmo, este tiende a converger correctamente hacia la solución la mayoría de las veces.

En lo referente a los resultados de tiempo, los tres casos presentan unos tiempos similares (el tercer caso es ligeramente mejor, pero no lo suficiente), y estos son elevados para un número de ciclos elevado (5 o más), por lo que, a pesar de haber conseguido obtener un resultado para un número de ciclos mayor a 3, cosa que hasta ahora era inalcanzable por métodos de fuerza bruta, como se demostró en (*Pérez, 2021*), estos no han sido totalmente satisfactorios, ya que los tiempos de ejecución escalan exponencialmente y esto hace que este algoritmo no sea aplicable para situaciones muy lejanas al origen.

4.2. Algoritmo basado en la equivalencia con robot hiper-redundante.

Como se ha comentado, en este último apartado, se pretende que el robot sea capaz de alcanzar posiciones muy lejanas al origen sin necesidad de solapar espacios de trabajo como en todos los casos de los capítulos 2 y 3. En otras palabras, se va a resolver de forma eficiente y funcional el problema que se lleva intentando resolver desde el principio de la planificación de trayectorias de este robot, iniciado en (*Pérez, 2021*) y al fin logrado en el presente Trabajo Fin de Máster. Este problema ha intentado resolverse por fuerza bruta, mediante el uso de redes neuronales y aplicando un algoritmo genético, y de todas esas maneras, no se ha conseguido ninguna que sea capaz de calcular una solución de forma eficiente para más de 4 ciclos de movimiento, por lo que se ha aplicado el algoritmo que se explicará en esta sección, y este consiste en tratar al robot binario como un robot hiper-redundante constituido por una cadena de módulos binarios.

Como se puede observar en la Figura 20, cada módulo se ha compuesto por una cadena de dos robots móviles, compartiendo el eslabón B (cian) como unión. De esta

forma, el siguiente módulo podría compartir el eslabón A con el módulo anterior y así sucesivamente hasta alcanzar el objetivo. Cabe destacar que el principio aplicado en esta sección se desarrolló en (*Ebert-Uphoff, 1996*), que se basa en el cálculo de la cinemática inversa de robots hiper-redundantes, y este ha sido aplicado al caso correspondiente a este trabajo. Dicho principio se encuentra explicado para el caso general en el artículo mencionado, por lo que en el presente trabajo se mostrará el desarrollo del mismo para el caso que ocupa.

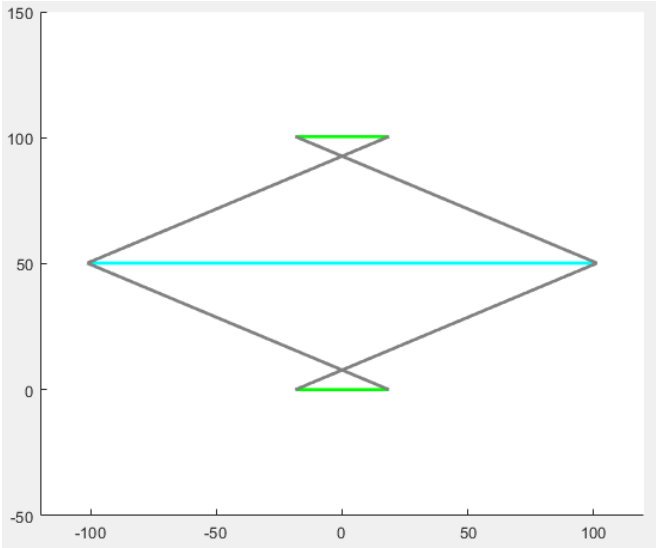

Figura 20: Representación gráfica de un módulo binario.

La unión de módulos binarios como el de la Figura 20 da como resultado una cadena de módulos como la que se muestra en la Figura 21.

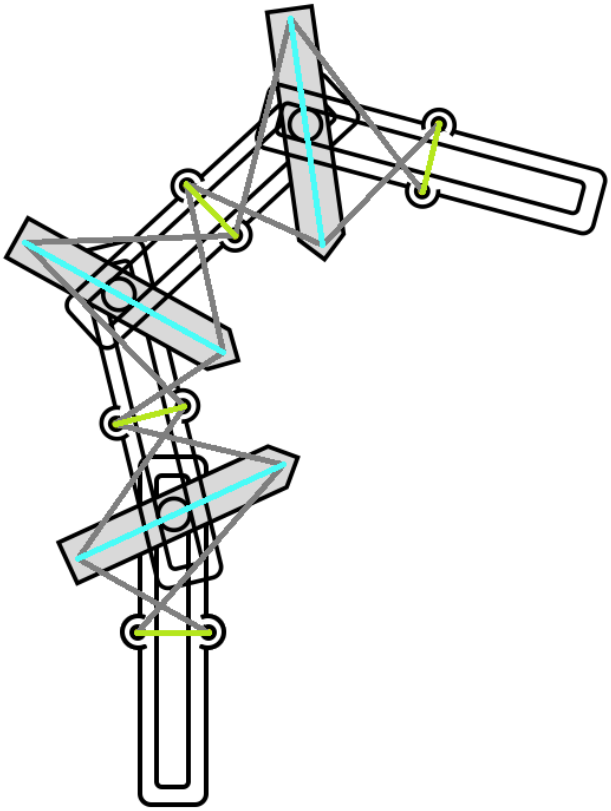

Figura 21: Cadena de módulos binarios.

Los *scripts* y funciones relacionados con el algoritmo desarrollado en este apartado se encuentran reflejados en el Anexo 6, y estas son las siguientes: **ws_alg_hr.m**, **calculo_malla_espacio_trabajo.m**, **convolucion.m**, **comprob_sec.m**, **plot_espacio_trabajo.m**, **plot_robot.m** y **plot_modulo.m**.

De todas ellas, la primera que se va a explicar será la función **convolucion.m**, ya que se trata de la más compleja de todas, pero a su vez es la más primitiva, ya que el resto del algoritmo depende de ella. Esta función se ha desarrollado siguiendo el principio sacado del artículo (*Ebert-Uphoff, 1995*), que se basa en generar de forma eficiente el espacio de trabajo de manipuladores binarios con muchos actuadores. Esta función recibe como parámetros de entrada la matriz de densidad de la iteración anterior (*D*), la cual será explicada más adelante, pero en esencia, almacena la densidad de puntos alcanzables por el espacio de trabajo en la iteración anterior; los límites máximos y mínimos que abarca la matriz *D* de la iteración anterior (*xmax*, *xmin*, *ymax* e *ymin*), y la celda que almacena las matrices de configuraciones del robot (*T*). Por otro lado, las salidas de esta función son las mismas que las entradas pero para la iteración actual (*D_s*, *xmax_s*, *xmin_s*, *ymax_s* e *ymin_s*), acompañadas de los índices correspondientes a la celda central de la

matriz *D_s*. Con estas variables definidas, ya se puede empezar a explicar el algoritmo desarrollado para esta función.

```
 3 -   [f,c] = size(D);
 4 -   deltaX = (xmax-xmin)/c;
 5 -   deltaY = (ymax-ymin)/f;
 6 -
 7 -   esquina1 = [xmin,ymin];
 8 -   esquina2 = [xmax,ymin];
 9 -   esquina3 = [xmin,ymax];
10 -   esquina4 = [xmax,ymax];
11 -   esquinas = [esquina1;esquina2;esquina3;esquina4];
12 -
13 -   nube = zeros(4*64,2);
14 -   global_counter = 1;
15 -   for k = 1:4
16 -     esquina = esquinas(k,:);
17 -     for i=1:8
18 -       for j=1:8
19 -         TM = T{i}/T{j};
20 -         v_p = TM*[esquina(1);esquina(2);1];
21 -         global_counter = global_counter + 1;
22 -         nube(global_counter,:) = [v_p(1),v_p(2)];
23 -       end
24 -     end
25 -   end
26 -
27 -   xmax_s = max(nube(:,1));
28 -   xmin_s = min(nube(:,1));
29 -   ymax_s = max(nube(:,2));
30 -   ymin_s = min(nube(:,2));
31 -
32 -   nx = ceil((xmax_s-xmin_s)/deltaX);
33 -   ny = ceil((ymax_s-ymin_s)/deltaY);
34 -   if floor(nx/2) == nx/2
35 -     nx = nx+1;
36 -   end
37 -   if floor(ny/2) == ny/2
38 -     ny = ny+1;
39 -   end
40 -
41 -   D_s = zeros(ny,nx);
42 -   deltaX_s = (xmax_s-xmin_s)/nx;
43 -   deltaY_s = (ymax_s-ymin_s)/ny;
44 -   i0 = ceil(c/2);
45 -   j0 = ceil(f/2);
46 -   i0_s = ceil(nx/2);
47 -   j0_s = ceil(ny/2);
```

Las primeras líneas de esta función se han empleado para definir variables que posteriormente serán necesarias. En primer lugar, se han definido el número de filas y columnas de la matriz de densidad (línea 3), y las dimensiones de cada celda (líneas 4-5); a continuación, se definen las esquinas de la matriz *D* en coordenadas locales, tomando para ello el número de celdas y el tamaño de las mismas (líneas 7-11). Con dichas esquinas, se pretende buscar el tamaño máximo que tendrá la matriz *D_s*, para lo que se

va a generar una nube de puntos desplazando las 4 esquinas definidas anteriormente a todas las posiciones posibles tras un ciclo de movimiento, tal y como se hace en (*Ebert-Uphoff, 1995*).

En otras palabras, se van a desplazar las 4 esquinas de la matriz D un número de veces igual a las posibilidades de movimiento de un ciclo de movimiento (en este caso, un ciclo de movimiento tiene 64 movimientos posibles), reflejado en las líneas 13-25. Una vez definidas todas las esquinas posibles, se pretende utilizarlas para calcular el tamaño de la nueva malla, que se usará para definir la matriz de densidades *D_s*; para ello, se calcularán los máximos y los mínimos de todos los puntos obtenidos para obtener los límites de dicha malla (líneas 27-30). Finalmente, se definen los parámetros de la nueva matriz de densidades (*D_s*), entre ellos el número de filas y columnas, siendo obligatorio que siempre sean impares (líneas 32-39), las nuevas dimensiones de celda (líneas 42-43), y las celdas centrales de las matrices de densidad (líneas 44-47).

El siguiente paso será la definición del propio contenido de la matriz de densidades, para lo que se ha desarrollado el siguiente bucle:

```
49 -  for n = 1:8
50 -    for m = 1:8
51 -      Tt = T{n}/T{m};
52 -      for i = 1:c
53 -        for j = 1:f
54 -          if D(j,i) > 0
55 -            x = deltaX*(i-i0);
56 -            y = deltaY*(j-j0);
57 -            xy_s = Tt*[x;y;1];
58 -            x_s = xy_s(1);
59 -            y_s = xy_s(2);
60 -            i_s = floor((x_s)/deltaX_s+0.5)+i0_s;
61 -            j_s = floor((y_s)/deltaY_s+0.5)+j0_s;
62 -            D_s(j_s,i_s) = D_s(j_s,i_s)+D(j,i);
63 -          end
64 -        end
65 -      end
66 -    end
67 -  end
```

En dicho bucle, se van a aplicar las ecuaciones empleadas en (*Ebert-Uphoff, 1995*) para el cálculo de la nueva matriz de densidades. Para ello, se van a trasladar los centros de todas las celdas siguiendo el mismo principio que las esquinas en las líneas 15-25 (líneas 55-59), posteriormente se calcula la posición correspondiente al centro de la celda trasladado en la nueva malla definida por la matriz de densidad *D_s* (líneas 60-61), y se añade el valor que contiene la celda trasladada de *D* a la nueva celda correspondiente de *D_s* (línea 62).

Una vez finalizado este bucle, ya se encuentra definida la matriz de densidades para la iteración actual, es decir, para el número de ciclos actual, ya que estas matrices se calculan ciclo a ciclo. El siguiente paso será explicar el resto de funciones, de las cuales la primera será **calculo_malla_espacio_trabajo.m**, una función muy sencilla que se encarga de calcular todas las matrices de densidad necesarias dependiendo del número de ciclos de movimiento que se quieran hacer. Para ello, esta función recibe como parámetros de entrada el índice correspondiente al ciclo que se quiere calcular (*N*), ya que, como se ha comentado anteriormente, estas matrices se calculan ciclo a ciclo, y la celda de configuraciones del robot (*T*).

```
 3 -   deltaX0 = 20;
 4 -   deltaY0 = 20;
 5 -   xmin = -deltaX0/2;
 6 -   xmax = deltaX0/2;
 7 -   ymin = -deltaY0/2;
 8 -   ymax = deltaY0/2;
 9 -   D = 1;
10 -
11 -   for i = 1:N
12 -     [D,xmax,xmin,ymax,ymin,i0_f,j0_f] = convolucion(D,xmax,xmin,ymax,ymin,T);
13 -   end
```

En primer lugar, se definen los parámetros iniciales de la malla y la matriz de densidades para el caso de 0 ciclos de movimiento. Dicha malla se define como una única celda de tamaño 20x20 (líneas 3-4), cuyo centro se encuentra en el centro de la propia celda, por lo que sus límites son la mitad de la celda inicial en longitud (líneas 5-8), y se define la matriz *D*, como se ha comentado, como una única celda, con densidad inicial 1 (línea 9).

A continuación, se calcula la matriz *D* correspondiente al instante actual, para lo que se necesita aplicar un bucle que vaya calculando las matrices de instantes anteriores hasta alcanzar la buscada (líneas 11-13). Con esto, ya se tiene la matriz *D* obtenida para el ciclo actual que se quería calcular, y esto se hará recursivamente en la función principal (**ws_alg_hr.m**) para calcular las matrices D de todos los ciclos desde el 1 hasta el N, pero esto se explicará en su debido momento.

Los siguientes algoritmos que se van a explicar son los que se ejecutan después de obtener los resultados por parte del *script* principal **ws_alg_hr.m**. Estos algoritmos se encargan de comprobar lo correcto que es el resultado y de representarlo gráficamente, y la razón de exponerlos antes que el *script* principal es porque así se ha hecho en el resto

de apartados del presente Trabajo Fin de Máster, por lo que se pretende seguir la misma dinámica.

El primero de ellos que será explicado será **comprob_sec.m**, el cual se encarga de tomar el resultado obtenido por **ws_alg_hr.m**, que será un vector que almacene los índices de las configuraciones correspondientes a la solución encontrada, y transformarlo en un resultado de posición y orientación. Este algoritmo se trata de un *script*, por lo que no tiene parámetros de entrada, sino que tomará los que necesite que se encuentren almacenados en el *Workspace*.

```
1 - T_encontrado = eye(3);
2 - for k=1:N
3 -   i = secuencia(k);
4 -   j = secuencia(k+N);
5 -   if i==-1 || j==-1
6 -     continue
7 -   end
8 -   T_encontrado = T_encontrado*T{i}/T{j};
9 - end
10 - P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)]
11 - phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
```

Este algoritmo consiste simplemente en tomar el vector secuencia y transformarlo en una matriz de transformación que represente la posición y orientación final del robot móvil (o lo que es lo mismo, del efector final del robot hiper-redundante equivalente); para ello, se toman los índices correspondientes a cada configuración consecutiva (líneas 3-4), y va haciendo el producto matricial entre la matriz de transformación de la iteración anterior y las correspondientes a los índices de la iteración actual (línea 8). Finalmente, se toman los resultados en posición y orientación de la matriz final calculada (líneas 10-11).

El próximo algoritmo a explicar será **plot_espacio_trabajo.m**, un *script* que se encarga de dibujar una nube de puntos que represente el espacio de trabajo del robot a partir de la matriz de densidades y las dimensiones de la malla, con el fin de poder apreciar visualmente cuál sería aproximadamente la dimensión y forma del espacio de trabajo para N ciclos de movimiento.

```
1 - [D,i0_f,j0_f,xmax,xmin,ymax,ymin] = calculo_malla_espacio_trabajo(N,T);
2 -
3 - u = unique(D);
4 - u(1) = [];
5 - num_colores = length(u)-1;
6 - delta_col = max(u)/num_colores;
7 - v_aux = [0;delta_col*ones(num_colores,1)];
8 - v_ac = cumsum(v_aux);
```

```
 9 -
10 - pos_col = zeros(size(D,2)*size(D,1),5);
11 - cont = 0;
12 - deltaX = (xmax-xmin)/size(D,2);
13 - deltaY = (ymax-ymin)/size(D,1);
14 -
15 - for i = 1:size(D,2)
16 -   for j = 1:size(D,1)
17 -     if D(j,i) == 0
18 -       continue
19 -     end
20 -     cont = cont+1;
21 -     x = deltaX*(i-i0_f);
22 -     y = deltaY*(j-j0_f);
23 -     esc = floor(1000000*log(D(j,i))/log(max(u)))/1000000;
24 -     color = [esc 0 0];
25 -     pos_col(cont,:) = [x y color];
26 -   end
27 - end
28 -
29 - pos_col(cont+1:end,:) = [];
30 -
31 - figure
32 - x = pos_col(:,1);
33 - y = pos_col(:,2);
34 - color = pos_col(:,3:5);
35 - scatter(x, y, 20, color, 'Filled')
36 -
37 - axis equal
```

En primer lugar, se debe aplicar la función **calculo_malla_espacio_trabajo.m** para calcular los parámetros mencionados anteriormente (matriz de densidad y límites de la malla) para el número de ciclos de movimiento total *N* (línea 1). A continuación, es necesario definir una serie de variables que serán necesarias después (líneas 3-13); la idea de estas variables es, ya que no se va a poder representar el espacio de trabajo como tal porque son demasiados puntos, representar mediante un gradiente de color las zonas con mayor densidad de puntos en el espacio de trabajo real; es por ello que la función de estas variables se explicará cuando estas sean empleadas.

La idea del algoritmo es recorrer todas las celdas de la malla (líneas 15-16), y en todas las que la densidad sea distinta a 0 (líneas 17-19), es decir, en todas aquellas en las que haya algún punto, se toma la posición correspondiente a la celda (líneas 21-22), se calcula la intensidad del color a partir del valor máximo de la densidad (*max(u)*) y en escala logarítmica (líneas 23-24). Finalmente, se almacenan en una matriz las posiciones calculadas y el valor de color correspondiente a cada posición (línea 25). Una vez obtenida una matriz que almacene las posiciones y los colores correspondientes, se usa el comando *scatter* para realizar la representación gráfica del espacio de trabajo (línea 35), que era el objetivo principal de este *script*.

El siguiente y último algoritmo que se va a explicar antes de proceder a la explicación del *script* principal es **plot_robot.m**, que a su vez emplea una subfunción llamada **plot_modulo.m**, por lo que esta será la primera en ser desarrollada.

```
 3 -   b = 18.59;
 4 -   p = 101.31;
 5 -
 6 -   T0 = [cos(phi0) -sin(phi0) P0(1); sin(phi0) cos(phi0) P0(2); 0 0 1];
 7 -   T1 = T0*T{i};
 8 -   T2 = T1/T{j};
 9 -
10 -   P2 = T2(1:2,3);
11 -   phi2 = atan2(T2(2,1),T2(1,1));
12 -
13 -   A01 = T0*[-b;0;1];
14 -   A02 = T0*[b;0;1];
15 -   B11 = T1*[-p;0;1];
16 -   B12 = T1*[p;0;1];
17 -   A21 = T2*[-b;0;1];
18 -   A22 = T2*[b;0;1];
19 -
20 -   plot([A01(1),A02(1)],[A01(2),A02(2)],'g','LineWidth',2)
21 -   hold on
22 -   set(gca,'DataAspectRatio',[1,1,1])
23 -   plot([B11(1),B12(1)],[B11(2),B12(2)],'c','LineWidth',2)
24 -   plot([A21(1),A22(1)],[A21(2),A22(2)],'g','LineWidth',2)
25 -   plot([A01(1),B12(1)],[A01(2),B12(2)],'color',[0.5,0.5,0.5],'LineWidth',2);
26 -   plot([A02(1),B11(1)],[A02(2),B11(2)],'color',[0.5,0.5,0.5],'LineWidth',2);
27 -   plot([B11(1),A22(1)],[B11(2),A22(2)],'color',[0.5,0.5,0.5],'LineWidth',2);
28 -   plot([B12(1),A21(1)],[B12(2),A21(2)],'color',[0.5,0.5,0.5],'LineWidth',2);
```

Esta función se encarga de dibujar un módulo del robot hiper-redundante, compuesto por dos configuraciones del robot móvil, por lo que esta función recibe como parámetros de entrada la celda de matrices con las 8 configuraciones (*T*), la posición y orientación iniciales del módulo que se pretende dibujar, es decir, las finales del módulo anterior (*P0*, *phi0*), y los índices correspondientes a las configuraciones que cada semi-módulo tiene que adoptar (*i* y *j*).

Este algoritmo consiste básicamente en encontrar los puntos correspondientes a las uniones entre los eslabones de la Figura 20 y dibujar entre dichos puntos los eslabones correspondientes, por lo que la esencia de esta función es encontrar dichos puntos. Para ello, se definen algunas variables iniciales, como lo son las dimensiones de los semi-eslabones A y B (líneas 3-4), la matriz de transformación del origen del módulo (línea 6), la matriz de transformación del efector final del primer semi-módulo (línea 7), la matriz de transformación del efector final del módulo (línea 8), y su posición y orientación (líneas 10-11). Con todos estos datos, es posible calcular todos los puntos necesarios

(líneas 13-18), y con estos se puede realizar la representación gráfica del módulo (líneas 20-28). Una vez explicada la función que permite representar gráficamente un módulo, solamente queda explicar el script que permite dibujar todos los módulos del robot hiper-redundante (**plot_robot.m**).

```
 1 - hold on
 2 -
 3 - P0 = [0 0];
 4 - phi0 = 0;
 5 -
 6 - i_v = secuencia(1:N);
 7 - j_v = secuencia(N+1:end);
 8 -
 9 - for i = 1:N
10 -   [P1,phi1] = plot_modulo(T,P0,phi0,i_v(i),j_v(i));
11 -   P0 = P1;
12 -   phi0 = phi1;
13 - end
14 - plot(bee(1),bee(2),'xy','Linewidth',2)
```

Este *script* se encarga de llamar recursivamente a la función descrita previamente, enviándole los índices de configuraciones correspondientes y actualizar las posiciones de cada módulo para enviárselas a la siguiente iteración (líneas 9-13).

Finalmente, únicamente resta explicar el algoritmo principal de esta sección, desarrollado en **ws_alg_hr.m**, el cual se encarga de implementar las funciones de cálculo explicadas como corresponda con el fin de encontrar la solución al problema propuesto con el número de ciclos propuesto. Cabe destacar que este algoritmo no tiene en cuenta la orientación del efector final, puesto que, en todos los algoritmos desarrollados en el presente trabajo, la orientación se ha definido mediante un algoritmo que orientaba al robot utilizando 4 ciclos de movimiento extras, por lo que se ha determinado que no es necesario hacer planificación de orientación teniendo en cuenta que se puede aplicar el mismo principio.

```
 3 - N = 7;
 4 - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
 5 - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,21.95478428];
 6 - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
 7 -
 8 - bee = [400,400];
 9 -
10 - for i=1:8
11 -   T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
12 - end
13 -
14 - deltaX0 = 20;
15 - deltaY0 = 20;
16 - xmin = -deltaX0/2;
17 - xmax = deltaX0/2;
```

```
18 - ymin = -deltaY0/2;
19 - ymax = deltaY0/2;
20 - D = 1;
```

Este algoritmo requiere de la definición de ciertas variables iniciales para poder funcionar, por lo que estas se definen en las primeras líneas, y son: el número de ciclos de movimiento (línea 3), la celda que almacena las configuraciones del robot (líneas 10-12, empleando los parámetros de diseño del robot de las líneas 5-6), la posición objetivo (línea 8), que en este caso se ha definido como *bee* en lugar de *P_ref* como en los casos anteriores debido a que así se ha nombrado en (*Ebert-Uphoff, 1996*), y las dimensiones y parámetros de la malla inicial (líneas 14-20).

```
22 - cell_mallas = {D};
23 - cell_limites = {[xmax,xmin,ymax,ymin]};
24 - cell_centros = {[1 1]};
25 -
26 - for i = 1:N-1
27 -   [D,i0_f,j0_f,xmax,xmin,ymax,ymin] = calculo_malla_espacio_trabajo(i,T);
28 -   cell_mallas{i+1} = D;
29 -   cell_limites{i+1} = [xmax,xmin,ymax,ymin];
30 -   cell_centros{i+1} = [i0_f j0_f];
31 - end
```

A continuación, se deben calcular las matrices de densidad, los límites y los centros de las mallas para todos los valores de número de ciclos entre 1 y *N* (líneas 22-31). Todos estos valores se almacenan en celdas para su posterior utilización.

```
33 - T_ant = eye(3);
34 - secuencia = zeros(1,2*N);
35 -
36 - for k = 1:N
37 -   densidad = zeros(8,8);
38 -   for i = 1:8
39 -     for j = 1:8
40 -       Tt = T{i}/T{j};
41 -       TL = T_ant*Tt;
42 -       bL = TL(1:2,3);
43 -       bu = TL(1:2,1:2)'*(bee'-bL);
44 -       ij_0 = cell_centros{N-(k-1)};
45 -       xy_maxmin = cell_limites{N-(k-1)};
46 -       D = cell_mallas{N-(k-1)};
47 -       deltaX = (xy_maxmin(1)-xy_maxmin(2))/size(D,2);
48 -       deltaY = (xy_maxmin(3)-xy_maxmin(4))/size(D,1);
49 -       i_b = floor((bu(1))/deltaX+0.5)+ij_0(1);
50 -       j_b = floor((bu(2))/deltaY+0.5)+ij_0(2);
51 -       if i_b>=1 && i_b<=size(D,2) && j_b>=1 && j_b<=size(D,1)
52 -         densidad(i,j) = D(j_b,i_b);
53 -       else
54 -         densidad(i,j) = 0;
55 -       end
56 -     end
57 -   end
58 -   maximo = max(max(densidad));
59 -   [f_max,c_max] = find(densidad==maximo);
```

```
60 -    if length(f_max) == 1
61 -      imax = f_max;
62 -      jmax = c_max;
63 -    else
64 -      norma = 100000000000;
65 -      for i = 1:length(f_max)
66 -        T_aux = T_ant*T{f_max(i)}/T{c_max(i)};
67 -        pos_aux = T_aux(1:2,3);
68 -        if norm(bee'-pos_aux) < norma
69 -          norma = norm(bee'-pos_aux);
70 -          imax = f_max(i);
71 -          jmax = c_max(i);
72 -        end
73 -      end
74 -    end
75 -    T_ant = T_ant*T{imax}/T{jmax};
76 -    secuencia(k) = imax;
77 -    secuencia(k+N) = jmax;
78 - end
```

El siguiente paso es generar una matriz 8x8 en la que se almacene la densidad de puntos que tendría cada posible configuración del módulo (cada semi-módulo tiene 8 configuraciones, por lo que hay 64 posibles combinaciones) si se trasladara la malla a la posición correspondiente a cada configuración. Para ello, en cada ciclo de movimiento (línea 36), se debe recorrer cada celda de la matriz 8x8 (líneas 38-39), se calculan las posiciones absoluta y relativa con respecto al objetivo del efector final del módulo actual (líneas 40-43). A continuación, se toman los centros, límites y matriz de densidad de la malla correspondiente al ciclo actual (líneas 44-46), y con ellos se calculan los tamaños de celda (líneas 47-48) y los índices de la malla donde se encuentra el punto de destino con respecto a la posición donde se haya posicionado el centro de la malla, o lo que es lo mismo, la celda que corresponde al destino en coordenadas locales de la malla (líneas 49-50). El siguiente paso es añadir a la matriz 8x8 el valor de densidad correspondiente, el cual, en caso de que los índices encontrados se encuentren dentro de los límites de la malla, dicho valor es el correspondiente al valor encontrado en la celda de los índices calculados, y en caso de que se encuentren fuera de la malla, este valor sería 0 (líneas 51-55).

Una vez se tiene definida la matriz de densidades, se busca en ella el mayor valor máximo (línea 58), y se busca a que celda pertenece (línea 59), ya que los índices de la celda a la que pertenecen corresponden con los índices de las configuraciones que cada semi-módulo del robot hiper-redundante debe adoptar para alcanzar el punto correspondiente a la celda seleccionada.

No obstante, es posible que haya dos celdas con la misma densidad, por lo que se debe tratar debidamente dicha situación. Para ello, se ha definido el caso en el que el comando *find* de la línea 59 solamente haya encontrado una solución, en la cual se define dicha solución como la única válida (líneas 60-62), y el caso contrario (línea 63), en el que se va a aplicar el siguiente criterio: se van a calcular las posiciones en el espacio que corresponderían con cada índice encontrado en la línea 59 (líneas 66-67), y con la posición encontrada para cada índice, se va a calcular la norma del vector entre la misma y el objetivo, buscando la menor de todas con el fin de que el algoritmo tome como siguiente punto el más cercano al destino con mayor densidad (líneas 68-71).

Una vez encontrados los índices correspondientes a las configuraciones que debe adoptar el módulo actual del robot hiper-redundante, (*imax* y *jmax*), se calcula la posición del efector final del módulo actual, o lo que es lo mismo, el inicio del siguiente (línea 75) y se añaden los índices encontrados al vector *secuencia*, que almacena todas las configuraciones de todos los módulos hasta el momento. Antes de finalizar la explicación del algoritmo, cabe destacar que para poder calcular y visualizar correctamente los resultados de esta sección, se deben ejecutar los siguientes *scripts* en el siguiente orden: **ws_alg_hr.m**, **comprob_sec.m**, **plot_espacio_trabajo.m** y **plot_robot.m**.

Con esto concluiría la explicación del algoritmo que trata al robot móvil como un equivalente hiper-redundante, únicamente restaría exponer los resultados obtenidos, y en este caso se expondrán varias situaciones, ya que se deben tener en cuenta una serie de aspectos.

El primer caso que se va a explicar es el mostrado en la Figura 22, el cual es el caso más lógico y sencillo de visualizar. En este caso, se ha determinado un número de ciclos de 7, es decir, 7 módulos del robot hiper-redundante, y con el destino en [400,400]. obteniendo un error en posición de 1.1678 mm. El código de colores para este y todos los casos es el siguiente: la nube de puntos representa el espacio de trabajo, con un color más claro cuanto mayor densidad de puntos hay en esa zona y viceversa, la cruz amarilla representa el destino que se quiere alcanzar, las líneas grises representan los actuadores binarios del robot, y las líneas verde y cian representan los eslabones A y B del robot móvil respectivamente.

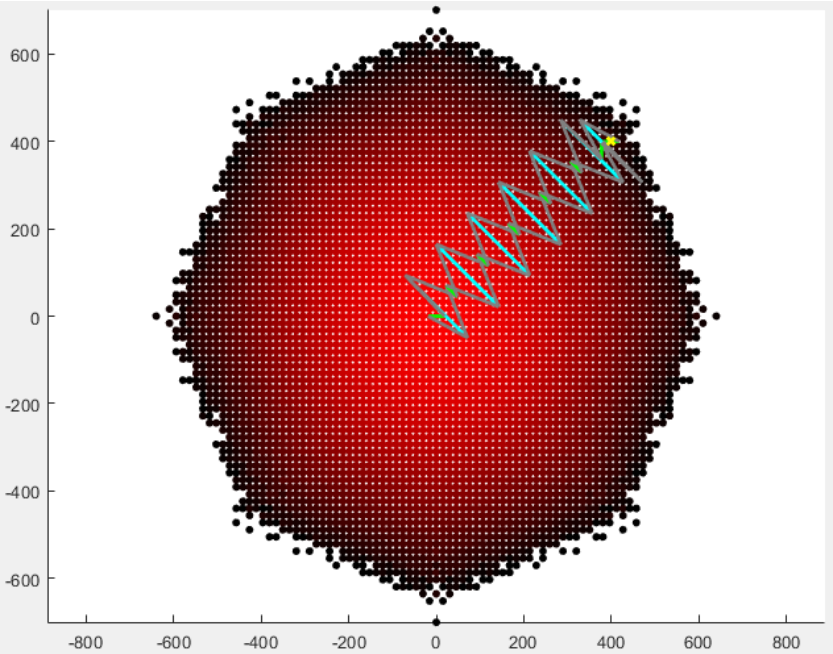

Figura 22: Resultado del algoritmo hiper-redundante para la posición [400,400].

El segundo caso que se va a exponer es el caso en el que se define una posición relativamente cercana al origen, en la cual el robot no necesita realizar todos los ciclos de movimiento definidos al inicio del algoritmo, pero, por la naturaleza del algoritmo (que es que el robot hiper-redundante siempre va a tener el mismo número de módulos y no es reductible) el robot va a tender a plegarse sobre sí mismo en torno al punto objetivo, como puede apreciarse en la Figura 23. En dicha figura, el punto objetivo se ha determinado en [200,200], con un número de módulos de 7 y con un error de posición de 0.5839 mm.

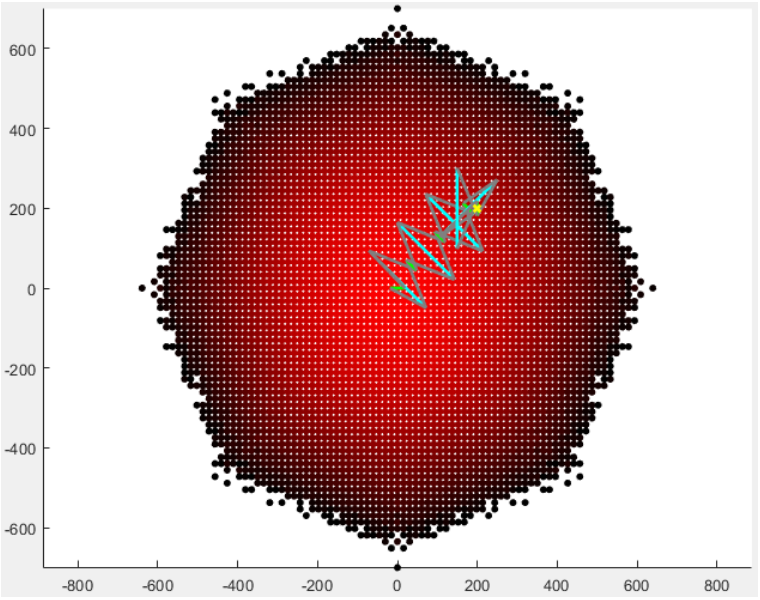

Figura 23: Resultado del algoritmo hiper-redundante para la posición [200,200].

El siguiente caso que se va a exponer es el mostrado en la Figura 24, el cual corresponde con un objetivo fuera del espacio de trabajo. Como puede apreciarse, a pesar de que el manipulador hiper-redundante nunca va a poder alcanzar el objetivo debido a la distancia a la que este se encuentra, este tiende a un resultado coherente, moviéndose en la dirección en la que se encuentra el objetivo, cosa inesperada, ya que en un principio se pensaba que simplemente el algoritmo no convergería. En este caso, no tiene sentido exponer el error en posición, este resultado se ha expuesto para mostrar la tendencia del algoritmo.

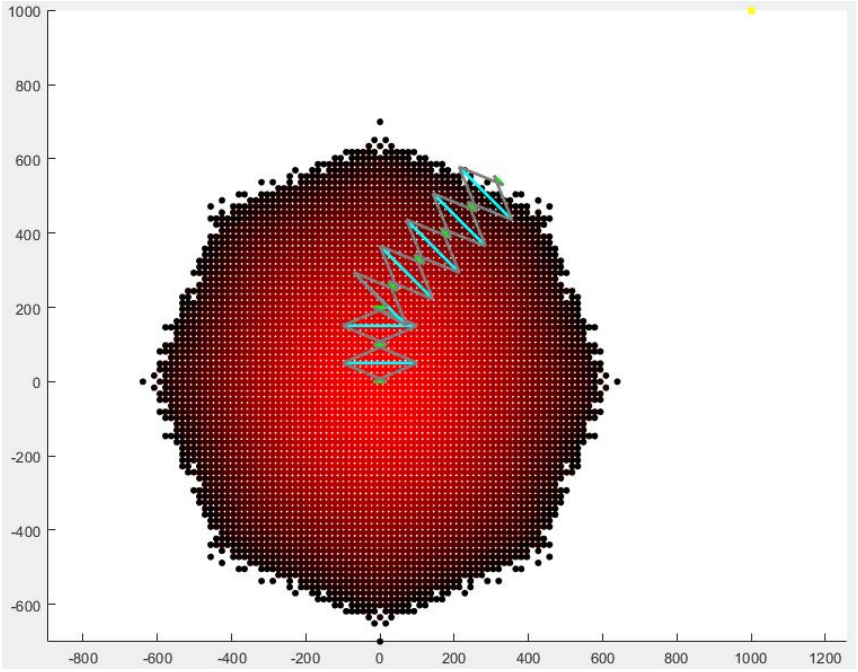

Figura 24: Resultado del algoritmo hiper-redundante para la posición [1000,1000].

Finalmente, el último caso que se va a exponer es el mostrado en la Figura 25, que es el caso en el que el objetivo se sitúa en [0,0], y como se puede apreciar, el robot se pliega sobre sí mismo varias veces sin mover el efector final, obteniendo un error en posición de 0 mm.

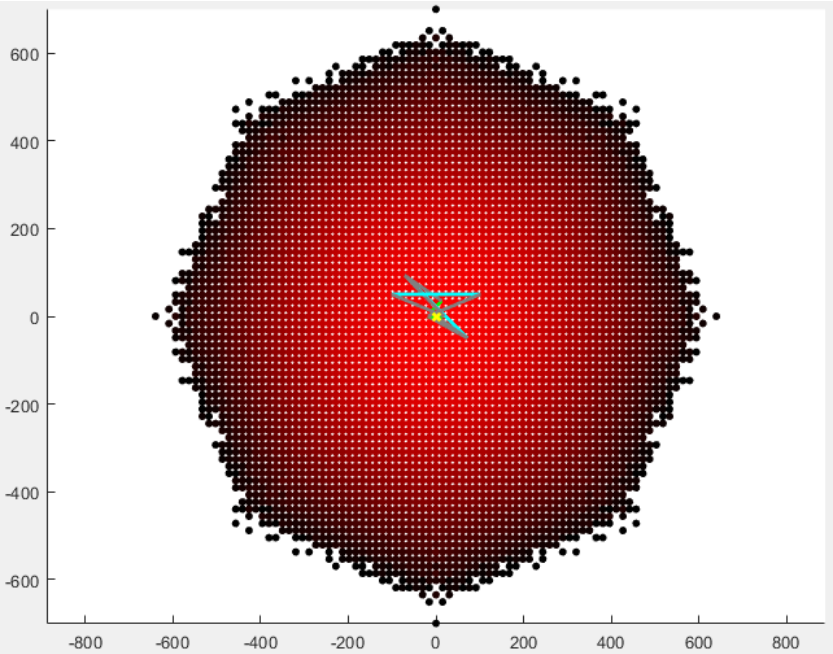

Figura 25: Resultado del algoritmo hiper-redundante para la posición [0,0].

A pesar de los resultados obtenidos en los casos mostrados, se han realizado varias pruebas de ejecución y se han obtenido errores de hasta 8 mm. A pesar de que estos valores no son tan precisos como los obtenidos en los casos mostrados, estos se pueden corregir fácilmente aplicando el algoritmo de orientación final, por lo que, en un principio, no sería necesario desarrollar ningún algoritmo más preciso que este para esta aplicación.

Capítulo 5. Conclusiones y trabajos futuros.

Este Trabajo Fin de Máster se han abordado diversos problemas de planificación de movimientos del robot móvil binario Xrobin. Algunos problemas de panificación de movimientos de dicho robot ya fueron abordados en (*Pérez, 2021*), pero otros problems quedaron pendientes de resolver o mejorar, de modo que el presente Trabajo Fin de Máster pretende resolver algunos de dichos problemas pendientes y además otros nuevos. En primer lugar, se ha decidido optimizar la comprobación de colisiones para que esta sea más efectiva que simplemente circunscribir al robot en una circunferencia y comprobar que esta no colisione con el obstáculo, que fue lo que se hizo en (*Pérez, 2021*). Esto se ha conseguido mediante el cálculo y la simplificación de las áreas barridas por cada eslabón en movimiento completo, y comprobando en cada medio ciclo (es decir, cada movimiento de un eslabón) si estas colisionan con el obsáculo. A pesar de que parezca que este algoritmo puede ralentizar la ejecución, ya que a priori es menos eficiente que comprobar si el obstáculo está o no en una circunferencia, la diferencia en el tiempo de ejecución es mínima, por no hablar de que, al ser menos restrictivo (ya que el área de los eslabones siempre será menor a la circunferencia), será más fácil que este encuentre puntos buenos, por lo que en ocasiones puede que hasta sea más rápido debido a que converge en menos iteraciones.

En segundo lugar, se ha optimizado el algoritmo de escape de mínimos locales mediante generación aleatoria de destinos intermedios diseñado en (*Pérez, 2021*). Este apartado parece ciertamente redundante, ya que prácticamente se ha rehecho casi desde el principio cuando en su día ya se hizo y se descartó, pero ha resultado ser muy diferente en esta ocasión, ya que los resultados obtenidos son mucho más alentadores, lo suficiente como para volver a tomar este algoritmo como una posibilidad viable. Dichos resultados son los siguientes: en (*Pérez, 2021*), se logró alcanzar una solución tras 121 ciclos de movimiento, mientras que en el caso de este trabajo se ha conseguido que este converja en 108 ciclos. Aunque estos resultados no parezcan del todo alejados, cabe destacar que, aunque las situaciones a las que han sido expuestos ambos algoritmos en lo que a obstáculos se refiere han sido bastante similares, el caso del presente trabajo ha sido más restrictivo que su antecesor, y aún así ha conseguido mejores resultados, por no hablar de que el algoritmo desarrollado en (*Pérez, 2021*) no conseguía converger en la mayoría de las situaciones y su tiempo de computación era muchísimo más elevado, cosas que no

pasan con el nuevo algoritmo (también debido a la nueva comprobación de colisiones explicada en el párrafo anterior).

El siguiente algoritmo desarrollado se ha basado en intentar eliminar la aleatoriedad del algoritmo anterior que le quita tanta eficiencia, ya que hace muchos movimientos intentando excapar del mínimo local sin éxito. La idea principal ha sido implementar el principio de *wall-following*, que consiste en que el robot siga la "pared" del obstáculo hasta detectar que lo ha esquivado y entonces retomar su objetivo inicial. Este algoritmo ha resultado ser bastante eficiente para los casos probados, pudiendo converger para el ejemplo expuesto en este trabajo en 32 ciclos de movimiento, que es bastante rápido teniendo en cuenta que es un algoritmo local (es decir, no se conoce el mapa de ocupación).

A continuación, se ha decidido mostrar la otra cara de la moneda: los algoirmtos que trabajan conociendo el mapa de ocupación. Los algoritmos de este tipo desarrollados para este trabajo han sido el algoritmo A* y el diagrama de Voronoi. El primero consiste en buscar entre todo el mapa de ocupación el camino más corto entre el origen y el destino, mientras que el segundo pretende buscar el trayecto más corto pero siguiendo un camino a través del espacio que pretende alejarse lo máximo posible de los obstáculos. De entre estos dos algoritmos, se ha encontrado que el A* es el que hace que el robot haga menos ciclos de movimiento para alcanzar el objetivo, y en cuanto a tiempo de computación ambos andan bastante a la par. No obstante, es cierto que hay posibilidad de que el diagrama de Voronoi pueda ser más idóneo que el A*, sobre todo para casos que conciernan a otros robots, ya que es posible que estos tengan mayor dificultad en atravesar estrecheces (que sería por donde se movería si se usara A*, ya que este busca el camino más corto), y por tanto sería mejor que tomara un camino alejándose de los obstáculos que atravesando dichas estrecheces.

Finalmente, se ha intentado encontrar una forma de poder hacer que el robot pueda moverse sin aplicar el solapamiento de espacios de trabajo y evitando el método de "fuerza bruta" en la búsqueda de puntos. Para ello, en primer lugar, se ha desarrollado un algoritmo basado en redes neuronales, el cual ha resultado ser muy poco satisfactorio en todos los casos en los que se ha intentado, por lo que se ha descartado directamente esta opción. A continuación, se ha diseñado un algoritmo genético para esta tarea, el cual, si bien ha conseguido funcionar tal y como se esperaba de él, este ha resultado no ser suficientemente bueno, ya que no es capaz de calcular de manera eficiente soluciones

para un número de ciclos mayor a 4, por lo que, a pesar de haber tenido resultados correctos, se ha descartado como método para comparar con el solapamiento de espacios de trabajo. Con el fin de solucionar este problema, se ha desarrollado un algoritmo basado en tratar al robot móvil como un robot binario hiper-redundante y aplicar sobre este los principios tomados de *Ebert-Uphoff* (1995) y *Ebert-Uphoff* (1996); a diferencia de los otros dos, este algoritmo sí ha sido capaz de dar resultados correctos de forma eficiente para cualquiera de los números de ciclos probados (se han probado hasta 20 ciclos de movimiento sin una gran pérdida de eficiencia). Una vez encontrado el sistema idóneo para hacer esta comparación, se va a proceder a comparar ambos algoritmos y ver lo eficiente que es la aplicación de solapamiento de espacios de trabajo: con el objetivo en [400,500], ambos algoritmos han logrado alcanzarlo en 8 ciclos de movimiento, por lo que se ha decidido probar para un caso más desfavorable; dicho caso ha sido con el objetivo en [980,980], para el cual el algoritmo de solapamiento de espacios de trabajo ha conseguido converger en 16 ciclos de movimiento, mientras que su homólogo lo ha hecho en 15. Esto demuestra que el algoritmo de solapamiento de espacios de trabajo desarrollado en (*Pérez, 2021*) es bastante bueno teniendo en cuenta que por un pequeño aumento del número de movimientos del robot se consigue una eficiencia mucho mayor en cuanto a velocidad de computación, algo que se lleva intentando demostrar desde que se hizo dicho trabajo y por fin se ha conseguido.

En cuanto a los trabajos futuros, se proponen los siguientes: la incorporación de obstáculos en el algoritmo del equivalente hiper-redundante para evitar colisiones, la construcción de un prototipo totalmente neumático del robot Xrobin para probar en él los algoritmos desarrollados en el presente Trabajo Fin de Máster, en especial los de tipo hiper-redundante, y combinar los algoritmos de planificación de trayectorias desarrollados en el presente trabajo con técnicas de localización y de *mapping* para la exploración de entornos.

ANEXOS

ANEXO 1

ws_binario_mp.m

```
clear all;

global nube
global global_counter
global secuencia
global indice_min
global secuencia_encontrada
global P_mas_cercano
global ori_mas_cercano
global Md
global Md1
global P_obs

N = 2;
P_ref = [600 500];
phi_ref = pi/4;
P0 = [0,0];
posicion_inicial = P0;
phi0 = 0;

P_obs = [];
P_obs_p = [];
paso_x = 10;
paso_y = 10;
for xobs = 400-300:paso_x:400+300
  for yobs = 300-50:paso_y:300+50
    if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end
for xobs = 300-50:paso_x:300+50
  for yobs = 400-300:paso_y:400+300
    if (xobs-300)^2/50^2 + (yobs-400)^2/300^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end

% Comprobación de si el punto inicial o final se encuentran obstaculizados
for i = 1:length(P_obs_p(:,1))
  if norm(P0-P_obs_p(i,:)) < 120.65
    disp('Error: el punto de inicio se encuentra muy cerca de un obstaculo.')
    pausa1 = 1;
    break
  else
    pausa1 = 0;
  end
  if norm(P_ref-P_obs_p(i,:)) < 120.65
    disp('Error: el punto final se encuentra muy cerca de un obstaculo.')
```

```
51 -        pausa2 = 1;
52 -        break
53 -      else
54 -        pausa2 = 0;
55 -      end
56 -    end
57 -    if pausa1 == 1 || pausa2 == 1
58 -      pause
59 -    end
60 -
61 -    Md = KDTreeSearcher(P_obs_p,'Distance','minkowski','P',inf,'BucketSize',1);
62 -
63 -    xmin = -300; xmax = 1200; ymin = -300; ymax = 1200;
64 -    P_obs = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax);
65 -
66 -
67 -    puntos_dest = [];
68 -    puntos_elegidos = [];
69 -    Rmax = 50;
70 -    x_p=0;
71 -    y_p=0;
72 -
73 -    Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
74 -
75 -    tamanyo = 0;
76 -    for i=1:N
77 -      tamanyo = tamanyo + 64^i;
78 -    end
79 -
80 -    nube = zeros(tamanyo,3);
81 -    global_counter = 0;
82 -    secuencia = (-1)*ones(tamanyo,2*N);
83 -    T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
84 -
85 -    registro_seq = [];
86 -    puntos_encontrados = [];
87 -    ori_encontradas = [];
88 -    nube_grande = [];
89 -
90 -    % Depende de: b, p, rho0, Delta_rho
91 -    phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
92 -    y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-
     21.95478428,0,21.95478428];
93 -
94 -    for i=1:8
95 -      T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
96 -    end
97 -
98 -    f_aux = 1;
99 -    f_aux_2 = 1;
100 -   ultimo_indice = 1;
101 -
102 -   % Áreas que recorren los eslabones
103 -   [areaA,areaB] = area_local(P0,phi0);
104 -
```

```
105 - while 1
106 -   if norm(P_ref-P0) < 10
107 -     break
108 -   end
109 -
110 -   if f_aux == 1
111 -     global_counter = 0;
112 -     ciclo_completo(P0,phi0,T,1,N);
113 -     f_aux = 0;
114 -     distancias = zeros(size(nube,1),1);
115 -     for i=1:size(nube,1)
116 -       distancias(i) = norm(P_ref-nube(i,1:2));
117 -     end
118 -   end
119 -
120 -   encontrar_punto(distancias)
121 -
122 -   pos_0 = P_mas_cercano;
123 -   ori_0 = ori_mas_cercano;
124 -
125 -   for h = 1:N
126 -
127 -     if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
128 -       break
129 -     end
130 -
131 -     indice_inicio = secuencia_encontrada(h);
132 -     indice_fin = secuencia_encontrada(h+N);
133 -
134 -     movB = 1;
135 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
136 -            sin(ori_0)  cos(ori_0) pos_0(2);
137 -                0           0         1  ];
138 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
139 -     [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                  indice_inicio,T_p,T,area);
140 -     if colision == 1
141 -       break
142 -     end
143 -
144 -     movB = 0;
145 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
146 -            sin(ori_0)  cos(ori_0) pos_0(2);
147 -                0           0         1  ];
148 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
149 -     [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                  indice_fin,T_p,T,area);
150 -     if colision == 1
151 -       break
152 -     end
153 -
154 -   end
155 -
156 -   if colision == 0
157 -     registro_seq = [registro_seq; secuencia_encontrada];
```

```
158 -       puntos_encontrados = [puntos_encontrados; P_mas_cercano];
159 -       ori_encontradas = [ori_encontradas; ori_mas_cercano];
160 -       P0 = P_mas_cercano;
161 -       phi0 = ori_mas_cercano;
162 -       f_aux = 1;
163 -       tam_p = length(puntos_encontrados(:,1));
164 -       if tam_p > 4
165 -         X_v = puntos_encontrados(tam_p-4:tam_p,1);
166 -         Y_v = puntos_encontrados(tam_p-4:tam_p,2);
167 -         DX = std(X_v);
168 -         DY = std(Y_v);
169 -         if DX < 5 && DY < 5
170 -           Rmax = Rmax + 25;
171 -           tam = length(registro_seq(:,1));
172 -           registro_seq = registro_seq(1:tam-3,:);
173 -           puntos_encontrados = puntos_encontrados(1:tam-3,:);
174 -           ori_encontradas = ori_encontradas(1:tam-3,:);
175 -           P_mas_cercano = puntos_encontrados(tam-3,:);
176 -           ori_mas_cercano = ori_encontradas(tam-3,:);
177 -           P0 = P_mas_cercano;
178 -           phi0 = ori_mas_cercano;
179 -           repulsor = puntos_encontrados(tam-3,:);
180 -
181 -           % Generación de puntos aleatorios
182 -           puntos_destino = [];
183 -           for g = 1:7
184 -             while 1
185 -               R = Rmax*rand;
186 -               angulo_hacia_destino = atan2(P_ref(2)-P0(2),P_ref(1)-P0(1));
187 -               if rand > 0.5 % Virar hacia la derecha
188 -                 angulo = angulo_hacia_destino - 3*pi/4 + pi/2*rand;
189 -               else % Virar hacia la izquierda
190 -               angulo = angulo_hacia_destino + 3*pi/4 - pi/2*rand;
191 -               end
192 -               x_p = P_mas_cercano(1) + R*cos(angulo);
193 -               y_p = P_mas_cercano(2) + R*sin(angulo);
194 -               P_ref_n = [x_p,y_p];
195 -               d_n = zeros(length(P_obs(:,1)),1);
196 -               for d = 1:length(P_obs(:,1))
197 -                 d_n(d) = norm(P_ref_n-P_obs(d,:));
198 -               end
199 -               if min(d_n) > 125
200 -                 puntos_destino = [puntos_destino;P_ref_n];
201 -                 break
202 -               end
203 -             end
204 -           end
205 -
206 -           % Búsqueda del mejor punto generado
207 -           l = length(puntos_destino(:,1));
208 -           dist_repulsor = zeros(l,1);
209 -           dist_ref = zeros(l,1);
210 -           dist_aux = zeros(l,1);
211 -           for k = 1:l
```

```
212 -             dist_repulsor(k) = norm(repulsor-puntos_destino(k,:));
213 -             dist_ref(k) = norm(P_ref - puntos_destino(k,:));
214 -             dist_aux(k) = dist_ref(k) - 2*dist_repulsor(k);
215 -           end
216 -           [~,ind_def] = min(dist_aux);
217 -           P_ref_n = puntos_destino(ind_def,:);
218 -           puntos_elegidos = [puntos_elegidos;P_ref_n];
219 -
220 -           while 1
221 -             if norm(P_ref_n-P0) < 10
222 -               break
223 -             end
224 -             if f_aux_2 == 1
225 -               global_counter = 0;
226 -               ciclo_completo(P0,phi0,T,1,N)
227 -               f_aux_2 = 0;
228 -               distancias_n = zeros(size(nube,1),1);
229 -               for i=1:size(nube,1)
230 -                 distancias_n(i) = norm(P_ref_n-nube(i,1:2));
231 -               end
232 -             end
233 -
234 -             encontrar_punto(distancias_n)
235 -
236 -             pos_0 = P_mas_cercano;
237 -             ori_0 = ori_mas_cercano;
238 -
239 -             for h = 1:N
240 -
241 -               if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
242 -                 break
243 -               end
244 -
245 -               indice_inicio = secuencia_encontrada(h);
246 -               indice_fin = secuencia_encontrada(h+N);
247 -
248 -               movB = 1;
249 -               T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
250 -                      sin(ori_0)  cos(ori_0) pos_0(2);
251 -                          0           0         1  ];
252 -               area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
253 -               [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                             indice_inicio,T_p,T,area);
254 -               if colision == 1
255 -                 break
256 -               end
257 -
258 -               movB = 0;
259 -               T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
260 -                      sin(ori_0)  cos(ori_0) pos_0(2);
261 -                           0           0         1  ];
262 -               area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
263 -               [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                             indice_fin,T_p,T,area);
264 -               if colision == 1
```

```
265 -                 break
266 -               end
267 -
268 -             end
269 -
270 -             if colision == 0
271 -               registro_seq = [registro_seq; secuencia_encontrada];
272 -               puntos_encontrados = [puntos_encontrados; P_mas_cercano];
273 -               ori_encontradas = [ori_encontradas; ori_mas_cercano];
274 -               P0 = P_mas_cercano;
275 -               phi0 = ori_mas_cercano;
276 -               f_aux_2 = 1;
277 -               nube_grande = [nube_grande;nube];
278 -             else
279 -               distancias_n(indice_min) = 1000000000;
280 -             end
281 -           end
282 -         end
283 -       end
284 -       nube_grande = [nube_grande;nube];
285 -     else
286 -       distancias(indice_min) = 1000000000;
287 -     end
288 -
289 - end
290 -
291 - ori_final;
292 - dibujo;
293 -
294 - save('resultados')
```

fr_2D_malla.m

```
 1 - function frontera = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax)
 2 -   global Md
 3 -
 4 -   Dx = xmax-xmin;
 5 -   Dy = ymax-ymin;
 6 -   g = round((Dx)/paso_x)+1;
 7 -   h = round((Dy)/paso_y)+1;
 8 -   hay_obstaculo = zeros(g,h);
 9 -   frontera = [];
10 -   v_paso = [paso_x,paso_y];
11 -   radio = min(v_paso)/2;
12 -
13 -   for i=1:g
14 -     x = (i-1)*paso_x+xmin;
15 -     for j = 1:h
16 -       y = (j-1)*paso_y + ymin;
17 -       pto = [x y];
18 -       Idx = rangesearch(Md,pto,radio);
19 -       Retrieved = zeros(length(Idx{1}),2);
20 -       if isempty(Idx{1}) == 0
21 -         hay_obstaculo(i,j) = 1;
22 -       end
```

```
23 -      end
24 -    end
25 -
26 -    for b = 1:g
27 -      for c = 1:h
28 -
29 -        columna = b;
30 -        fila = c;
31 -        if hay_obstaculo(columna,fila) == 1
32 -          vecinos = zeros(8,2);
33 -          z = 1;
34 -          for i = -1:1
35 -            for j = -1:1
36 -              if (i~=0 || j~=0)
37 -                vecinos(z,1) = columna+i;
38 -                vecinos(z,2) = fila+j;
39 -                z = z+1;
40 -              end
41 -            end
42 -          end
43 -          for k = 1:8
44 -            e = vecinos(k,1);
45 -            d = vecinos(k,2);
46 -            if e>g || d>h || e<1 || d<1
47 -              continue
48 -            end
49 -            if hay_obstaculo(e,d) == 1
50 -              continue
51 -            else
52 -              p = [columna fila];
53 -              frontera = [frontera; [(p(1)-1)*paso_x+xmin,(p(2)-1)*paso_y+ymin]];
54 -            end
55 -          end
56 -        end
57 -      end
58 -    end
59 -
60 - end
```

area_local.m

```
 1 - function [areaA,areaB] = area_local(P0,phi0)
 2 -
 3 -   v = [1,2,3,4,5,6,7,8,1];
 4 -   BarrB = [];
 5 -   BarrA = [];
 6 -   for t = 1:8
 7 -     BarrB = calculo_area_adyac(v(t),v(t+1),1,P0,phi0,BarrB);
 8 -   end
 9 -   areaB = BarrB;
10 -   v_aux = 101.31*ones(1,20);
11 -   v_aux_p = -50.242:5.2886:50.242;
12 -   v_t = transpose([v_aux; v_aux_p; ones(1,20)]);
13 -   v_t_2 = transpose([-v_aux; v_aux_p; ones(1,20)]);
14 -   areaB = [areaB(40:80,1) areaB(40:80,2);
15 -            v_t(:,1) v_t(:,2);
16 -            areaB(120:160,1) areaB(120:160,2);
```

```
17 -            areaB(1:41,3)  areaB(1:41,4);
18 -            v_t_2(:,1) v_t_2(:,2);
19 -            areaB(80:120,3) areaB(80:120,4)];
20 -    for u = 1:8
21 -      BarrA = calculo_area_adyac(v(u),v(u+1),0,P0,phi0,BarrA);
22 -    end
23 -    areaA = BarrA;
24 -    v_aux_2p = -9.295:3.718:9.295;
25 -    v_aux_3p = 120.242*ones(1,6);
26 -    v_aux_4p = 75.668*ones(1,9);
27 -    v_aux_5p = -22.5656:5.6414:22.5656;
28 -    v_t_p = transpose([v_aux_2p; v_aux_3p; ones(1,6)]);
29 -    v_t_2p = transpose([v_aux_4p; v_aux_5p; ones(1,9)]);
30 -    v_t_3p = transpose([v_aux_2p; -1*v_aux_3p; ones(1,6)]);
31 -    v_t_4p = transpose([-1*v_aux_4p; v_aux_5p; ones(1,9)]);
32 -    areaA = [areaA(52:80,1) areaA(52:80,2);
33 -             v_t_p(:,1) v_t_p(:,2);
34 -             areaA(80:110,3) areaA(80:110,4);
35 -             v_t_2p(:,1) v_t_2p(:,2);
36 -             areaA(1:30,7) areaA(1:30,8);
37 -             v_t_3p(:,1) v_t_3p(:,2);
38 -             areaA(131:160,5) areaA(131:160,6);
39 -             v_t_4p(:,1) v_t_4p(:,2)];
40 -
41 - end
```

calculo_area_adyac.m

```
 1 - function Barr = calculo_area_adyac(i,f,movB,pos_0,ori_0,Barr)
 2 -   m = 20;
 3 -   p = 101.31;
 4 -   b = 18.59;
 5 -   Barr_aux = Barr;
 6 -   phi_v = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
 7 -   y_v = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
           21.95478428];
 8 -
 9 -   phi_inicial = phi_v(i);
10 -   y_inicial = y_v(i);
11 -   phi_final = phi_v(f);
12 -   y_final = y_v(f);
13 -
14 -   [li,ri] = inversa(phi_inicial,y_inicial);
15 -   [lf,rf] = inversa(phi_final,y_final);
16 -   dl = (lf-li)/(m-1);
17 -   dr = (rf-ri)/(m-1);
18 -
19 -   l = li;
20 -   r = ri;
21 -   phi = phi_inicial;
22 -   y = y_inicial;
23 -
24 -   for i = 1:m
25 -
26 -     [phi,y] = directa(l,r,phi,y);
27 -     l = l+dl;
28 -     r = r+dr;
29 -
30 -     T_BA = [ cos(phi)  -sin(phi)    0;
31 -              sin(phi)   cos(phi)    y;
32 -                 0          0        1];
33 -     if movB == 1
34 -       T_A = [ cos(ori_0) -sin(ori_0)  pos_0(1);
```

```
35 -                sin(ori_0) cos(ori_0)   pos_0(2);
36 -                     0          0           1   ];
37 -        T_B = T_A*T_BA;
38 -        B1 = T_B*[p;0;1];
39 -        B2 = T_B*[-p;0;1];
40 -        Barr_aux = [Barr_aux; B1(1) B1(2) B2(1) B2(2)];
41 -      else
42 -        T_B = [cos(ori_0) -sin(ori_0)  pos_0(1);
43 -               sin(ori_0) cos(ori_0)   pos_0(2);
44 -                    0          0           1   ];
45 -        T_A = T_B*inv(T_BA);
46 -        altura_pasador = 70;
47 -        radio_pasador = b/2;
48 -        a11 = T_A*[-radio_pasador;altura_pasador;1];
49 -        a12 = T_A*[ radio_pasador;altura_pasador;1];
50 -        a21 = T_A*[-radio_pasador;-altura_pasador;1];
51 -        a22 = T_A*[ radio_pasador;-altura_pasador;1];
52 -        Barr_aux = [Barr_aux; a11(1) a11(2) a12(1) a12(2) a21(1) a21(2)
                      a22(1) a22(2)];
53 -      end
54 -    end
55 -
56 -    Barr = Barr_aux;
57 -
58 - end
```

inversa.m

```
 1 - function [l,r] = inversa(phi,y)
 2 -   b = 18.59;
 3 -   p = 101.31;
 4 -   l = sqrt((p*cos(phi)+b)^2+(y+p*sin(phi))^2);
 5 -   r = sqrt((-p*cos(phi)-b)^2+(y-p*sin(phi))^2);
 6 - end
```

directa.m

```
 1 - function [phi,y] = directa(l,r,phi_p,y_p)
 2 -   b = 18.59;
 3 -   p = 101.31;
 4 -   F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
 5 -        (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
 6 -   J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
 7 -        -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
 8 -   aux = [phi_p;y_p];
 9 -
10 -   for iter = 1:10
11 -     aux = aux - J\F;
12 -     phi_p = aux(1);
13 -     y_p = aux(2);
14 -     J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
15 -          -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
16 -     F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2;
17 -          (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
18 -   end
19 -   phi = aux(1);
20 -   y = aux(2);
21 -
22 - end
```

ciclo_completo.m

```
 1 - function ciclo_completo(pos_0,ori_0,T,n,N)
 2 -   global nube
 3 -   global global_counter
```

```
 4 -   global secuencia
 5 -   v_i=-1*ones(1,N);
 6 -   v_j=-1*ones(1,N);
 7 -   if n<=N
 8 -
 9 -     TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
10 -             sin(ori_0), cos(ori_0),pos_0(2);
11 -                  0 ,      0     ,    1   ];
12 -
13 -     for i=1:8
14 -       TB = TA_0*T{i};
15 -       for j=1:8
16 -         TA = TB*inv(T{j});
17 -         pos = TA(1:2,3)';
18 -         ori = atan2(TA(2,1),TA(1,1));
19 -         global_counter = global_counter + 1;
20 -         nube(global_counter,:) = [pos,ori];
21 -         if n==1
22 -           v_i(1)=i;
23 -           v_j(1)=j;
24 -           secuencia(global_counter,:) = [v_i v_j];
25 -         else
26 -           auxiliar=secuencia(global_counter-1,:);
27 -           auxiliar(n)=i;
28 -           auxiliar(N+n)=j;
29 -           secuencia(global_counter,:) = auxiliar;
30 -         end
31 -         ciclo_completo(pos,ori,T,n+1,N);
32 -       end
33 -     end
34 -   end
35 - end
```

encontrar_punto.m

```
 1 - function encontrar_punto(distancias)
 2 -   global nube
 3 -   global indice_min
 4 -   global secuencia_encontrada
 5 -   global P_mas_cercano
 6 -   global ori_mas_cercano
 7 -   global secuencia
 8 -   [~,indice_min] = min(distancias);
 9 -   P_mas_cercano = nube(indice_min,1:2);
10 -   ori_mas_cercano = nube(indice_min,3);
11 -   secuencia_encontrada = secuencia(indice_min,:);
12 - end
```

calculo_area.m

```
 1 - function area = calculo_area(movB,pos_0,ori_0,T_p,area)
 2 -   area_aux = [area(:,1) area(:,2) ones(length(area),1)];
 3 -   area_aux = transpose(T_p*transpose(area_aux));
 4 -   area = [area_aux(:,1) area_aux(:,2)];
 5 - end
```

comprobacion_colisiones.m

```
 1 - function [colision,pos_F,ori_F] = comprobacion_colisiones(movB,pos_0,ori_0,
                                       indice,T_p,T,area)
 2 -   Retrieved = codigo_kdtree(pos_0,area);
 3 -   if length(Retrieved) == 0
 4 -     colision = 0;
 5 -   else
```

```
 6 -     Retrieved_L = T_p\[Retrieved(:,1) Retrieved(:,2)
                          ones(length(Retrieved(:,1)),1)]';
 7 -     x_obs_L = Retrieved_L(1,:);
 8 -     y_obs_L = Retrieved_L(2,:);
 9 -     for j = 1:length(x_obs_L)
10 -       if movB == 1
11 -         if x_obs_L(j)^2+y_obs_L(j)^2<=13891.29 && x_obs_L(j)>=-101.31 && ...
12 -           x_obs_L(j)<=101.31 && y_obs_L(j)>=-103.1332 && y_obs_L(j)<=103.1332
13 -           colision = 1;
14 -           break
15 -         else
16 -           colision = 0;
17 -         end
18 -       else
19 -         if 0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 &&
                -0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 && ...
20 -            -0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 && 0.9913*x_obs_L(j)-
                129.4137-y_obs_L(j)<=0 && ...
21 -            x_obs_L(j)<=75.668 && x_obs_L(j)>=-75.668 &&
                y_obs_L(j)<=120.2 && y_obs_L(j)>=-120.2
22 -           colision = 1;
23 -           break
24 -         else
25 -           colision = 0;
26 -         end
27 -       end
28 -     end
29 -   end
30 -   if movB == 1
31 -     T_rel = T{indice};
32 -     T_new = T_p*T_rel;
33 -   else
34 -     T_rel = T{indice};
35 -     T_new = T_p/T_rel;
36 -   end
37 -   pos_F = [T_new(1,3) T_new(2,3)];
38 -   ori_F = atan2(T_new(2,1),T_new(2,2));
39 - end
```

codigo_kdtree.m

```
 1 - function Retrieved = codigo_kdtree(pos_0,area)
 2 -   global Md1
 3 -   global P_obs
 4 -   p1 = min(area(:,1));
 5 -   p2 = max(area(:,2));
 6 -   p3 = max(area(:,1));
 7 -   p4 = min(area(:,2));
 8 -   pt = [p1,p2,p3,p4];
 9 -   p = max(pt);
10 -   punto_cuadrado = [p,pos_0(1)];
11 -   centro = pos_0;
12 -   radio = norm(pos_0-punto_cuadrado);
13 -   Idx = rangesearch(Md1,centro,radio);
14 -   Retrieved = zeros(length(Idx{1}),2);
15 -   for i=1:length(Idx{1})
16 -     Retrieved(i,:) = P_obs(Idx{1}(i),:);
17 -   end
18 - end
```

ori_final.m

```
 1 - load resultado_ori_final;
 2 - v_indices = [];
 3 - v_dist = [];
```

```
T_encontrado = eye(3);
phi_dif = phi_ref-nube(indice_min,3);
if phi_dif > pi
  phi_dif = phi_dif-2*pi;
elseif phi_dif < -pi
  phi_dif = phi_dif+2*pi;
end
for x = 1:length(nube_fin(:,1))
  if abs(nube_fin(x,3)-phi_dif) < 0.001
    v_indices = [v_indices;x];
  end
end
for y = 1:size(v_indices)
  v_aux = nube_fin(v_indices(y),1:2);
  theta = atan2(v_aux(2),v_aux(1));
  phi_dif_aux = theta+nube(indice_min,3);
  mod = norm(v_aux);
  P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
           P_mas_cercano(2)+mod*sin(phi_dif_aux)];
  dist = norm(P_aux-P_ref);
  v_dist = [v_dist;dist];
end
[~,ind_tmp] = min(v_dist);
secuencia_final = seq_fin(v_indices(ind_tmp),:);
for k = 1:4
  i = secuencia_final(k);
  j = secuencia_final(k+4);
  if i==-1 || j==-1
    continue
  end
  T_encontrado = T_encontrado*T{i}/T{j};
end
T0 = [ cos(nube(indice_min,3)) -sin(nube(indice_min,3)) nube(indice_min,1);
       sin(nube(indice_min,3))  cos(nube(indice_min,3)) nube(indice_min,2);
                   0                        0                    1        ];

T1 = T0*T_encontrado;
P_final = T1(1:2,3)';
phi_final = atan2(T1(2,1),T1(1,1));
```

dibujo.m

```
plot(nube_grande(:,1),nube_grande(:,2),'.b');
set(gca,'DataAspectRatio',[1,1,1]);
hold on
axis([xmin xmax ymin ymax]);

for d = 1:size(P_obs_p,1)
  P = P_obs_p(d,:);
  plot(P(1),P(2),'.k');
end
for d = 1:size(P_obs,1)
  P = P_obs(d,:);
  plot(P(1),P(2),'.r');
end

plot(P_ref(1),P_ref(2),'or')
plot(posicion_inicial(1),posicion_inicial(2),'.g','MarkerSize',25)
plot(puntos_elegidos(:,1),puntos_elegidos(:,2),'*y')
quiver(P_ref(1),P_ref(2),50*cos(phi_ref),50*sin(phi_ref),'r')
plot(P_mas_cercano(1),P_mas_cercano(2),'*m')
quiver(P_mas_cercano(1),P_mas_cercano(2),
       50*cos(nube(indice_min,3)),50*sin(nube(indice_min,3)),'m')
```

```
21 - plot(P_final(1),P_final(2),'sg')
22 - quiver(P_final(1),P_final(2),50*cos(phi_final),50*sin(phi_final),'g');
```

ANEXO 2

ws_binario_mp.m

```
clear all;

global nube
global global_counter
global secuencia
global indice_min
global secuencia_encontrada
global P_mas_cercano
global ori_mas_cercano
global Md
global Md1
global P_obs

N = 2;
P_ref = [500 500];
phi_ref = pi/4;
P0 = [0,0];
posicion_inicial = P0;
phi0 = 0;

P_obs = [];
P_obs_p = [];
paso_x = 10;
paso_y = 10;
for xobs = 400-300:paso_x:400+300
  for yobs = 300-50:paso_y:300+50
    if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end
for xobs = 300-50:paso_x:300+50
  for yobs = 400-300:paso_y:400+300
    if (xobs-300)^2/50^2 + (yobs-400)^2/300^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end


% Comprobación de si el punto inicial o final se encuentran obstaculizados
for i = 1:length(P_obs_p(:,1))
  if norm(P0-P_obs_p(i,:)) < 120.65
    disp('Error: el punto de inicio se encuentra muy cerca de un obstaculo.')
    pausa = 1;
    break
  else
    pausa = 0;
  end
  if norm(P_ref-P_obs_p(i,:)) < 120.65
    disp('Error: el punto final se encuentra muy cerca de un obstaculo.')
    pausa = 1;
    break
  else
    pausa = 0;
  end
end
```

```
58 - if pausa == 1
59 -   pause
60 - end
61 -
62 - Md = KDTreeSearcher(P_obs_p,'Distance','minkowski','P',inf,'BucketSize',1);
63 -
64 - xmin = -300; xmax = 1200; ymin = -300; ymax = 1200;
65 - P_obs = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax);
66 -
67 -
68 - puntos_dest = [];
69 - puntos_elegidos = [];
70 - x_p=0;
71 - y_p=0;
72 -
73 - Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
74 -
75 - tamanyo = 0;
76 - for i=1:N
77 -   tamanyo = tamanyo + 64^i;
78 - end
79 -
80 - nube = zeros(tamanyo,3);
81 - global_counter = 0;
82 - secuencia = (-1)*ones(tamanyo,2*N);
83 - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
84 -
85 - registro_seq = [];
86 - puntos_encontrados = posicion_inicial;
87 - P_ref_n = posicion_inicial;
88 - ori_encontradas = [];
89 - nube_grande = [];
90 - repulsores = [];
91 - puntos_anteriores = [];
92 - pes = [];
93 - qus = [];
94 -
95 - % Depende de: b, p, rho0, Delta_rho
96 - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
97 - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
        21.95478428];
98 -
99 - for i=1:8
100 -   T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
101 - end
102 -
103 - f_aux = 1;
104 - f_aux_2 = 1;
105 - ultimo_indice = 1;
106 -
107 - % Áreas que recorren los eslabones
108 - [areaA,areaB] = area_local(P0,phi0);
109 -
110 - while 1
111 -   if norm(P_ref-P0) < 10
112 -     break
113 -   end
114 -
115 -   if f_aux == 1
116 -     global_counter = 0;
117 -     ciclo_completo(P0,phi0,T,1,N);
118 -     f_aux = 0;
```

```
119 -     distancias = zeros(size(nube,1),1);
120 -     for i=1:size(nube,1)
121 -       distancias(i) = norm(P_ref-nube(i,1:2));
122 -     end
123 -   end
124 -
125 -   encontrar_punto(distancias)
126 -
127 -   pos_0 = P_mas_cercano;
128 -   ori_0 = ori_mas_cercano;
129 -
130 -   for h = 1:N
131 -
132 -     if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
133 -       break
134 -     end
135 -
136 -     indice_ini = secuencia_encontrada(h);
137 -     indice_fin = secuencia_encontrada(h+N);
138 -
139 -     movB = 1;
140 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
141 -            sin(ori_0)  cos(ori_0) pos_0(2);
142 -                0           0         1  ];
143 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
144 -     [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                  indice_ini,T_p,T,area);
145 -     if colision == 1
146 -       break
147 -     end
148 -
149 -     movB = 0;
150 -     T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
151 -            sin(ori_0)  cos(ori_0) pos_0(2);
152 -                0           0         1  ];
153 -     area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
154 -     [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                  indice_fin,T_p,T,area);
155 -     if colision == 1
156 -       break
157 -     end
158 -
159 -   end
160 -
161 -   if colision == 0
162 -     registro_seq = [registro_seq; secuencia_encontrada];
163 -     puntos_encontrados = [puntos_encontrados; P_mas_cercano];
164 -     ori_encontradas = [ori_encontradas; ori_mas_cercano];
165 -     P0 = P_mas_cercano;
166 -     phi0 = ori_mas_cercano;
167 -     f_aux = 1;
168 -     tam_p = length(puntos_encontrados(:,1));
169 -     if tam_p > 4
170 -       X_v = puntos_encontrados(tam_p-3:tam_p,1);
171 -       Y_v = puntos_encontrados(tam_p-3:tam_p,2);
172 -       DX = std(X_v);
173 -       DY = std(Y_v);
174 -       if DX < 5 && DY < 5
175 -         flag_destino = 0;
176 -         registro_seq = registro_seq(1:tam_p-3,:);
177 -         puntos_encontrados = puntos_encontrados(1:tam_p-3,:);
178 -         ori_encontradas = ori_encontradas(1:tam_p-3,:);
```

```
179 -         P_mas_cercano = puntos_encontrados(tam_p-3,:);
180 -         ori_mas_cercano = ori_encontradas(tam_p-3,:);
181 -         repulsor = puntos_encontrados(tam_p-3,:);
182 -         repulsores = [repulsores;repulsor];
183 -         X_t = [repulsor(1) P_ref_n(1)];
184 -         Y_t = [repulsor(2) P_ref_n(2)];
185 -         D_Xt = std(X_t);
186 -         D_Yt = std(Y_t);
187 -         if D_Xt > 20 && D_Yt > 20
188 -           punto_anterior = P_ref_n;
189 -           puntos_anteriores = [puntos_anteriores;punto_anterior];
190 -         end
191 -         while flag_destino == 0
192 -           P0 = P_mas_cercano;
193 -           phi0 = ori_mas_cercano;
194 -           Idx = rangesearch(Md1,repulsor,140);
195 -           Retrieved = zeros(length(Idx{1}),2);
196 -           for i = 1:length(Idx{1})
197 -             Retrieved(i,:) = P_obs(Idx{1}(i),:);
198 -           end
199 -           if isempty(Retrieved)
200 -             continue
201 -           else
202 -             retrieved_points = [];
203 -             for i = 1:length(Retrieved(:,1))
204 -               if norm(Retrieved(i,:)-repulsor) <= 140
205 -                 retrieved_points = [retrieved_points;Retrieved(i,:)];
206 -               end
207 -             end
208 -           end
209 -
210 -           [p,q] = LineFitting(retrieved_points);
211 -           pes = [pes;p];
212 -           qus = [qus;q];
213 -
214 -           u = p-q;
215 -           distancia_maxima = 0;
216 -           A = u(2)/u(1);
217 -           B = -1;
218 -           C = -B*p(2)-A*p(1);
219 -           for i = 1:length(Retrieved(:,1))
220 -             r = Retrieved(i,:);
221 -             if r(1) == 0 && r(2) == 0
222 -               continue
223 -             else
224 -               d = abs((A*r(1)+B*r(2)+C)/sqrt(A^2+B^2));
225 -               if d > distancia_maxima
226 -                 distancia_maxima = d;
227 -                 punto_mas_lejano = r;
228 -               end
229 -             end
230 -           end
231 -           if distancia_maxima > 20
232 -             w = repulsor - punto_mas_lejano;
233 -           else
234 -             w = repulsor - punto_anterior;
235 -             line = 1;
236 -           end
237 -
238 -           v = u*(u(1)*w(1)+u(2)*w(2))/norm(u)^2;
239 -           v_unitario = v/norm(v);
240 -
```

```
241 -           R = 100;
242 -           deltaR = 25;
243 -           while 1
244 -             punto_destino = repulsor + v_unitario*R;
245 -             Idx = rangesearch(Md1,punto_destino,200);
246 -             Retrieved = zeros(length(Idx{1}),2);
247 -             for i = 1:length(Idx{1})
248 -               Retrieved(i,:) = P_obs(Idx{1}(i),:);
249 -             end
250 -             if isempty(Retrieved) == 0
251 -               vacia = 1;
252 -               for i = 1:length(Retrieved(:,1))
253 -                 if Retrieved(i,1) ~= 0 || Retrieved(i,2) ~= 0
254 -                   vacia = 0;
255 -                   break
256 -                 else
257 -                   vacia = 1;
258 -                 end
259 -               end
260 -             else
261 -               vacia = 1;
262 -             end
263 -             if vacia == 0
264 -               for i = 1:length(Retrieved(:,1))
265 -                 if Retrieved(i,1) ~= 0 || Retrieved(i,2) ~= 0
266 -                   if norm(Retrieved(i,:)-punto_destino) <= 200
267 -                     f_col = 1;
268 -                     break
269 -                   else
270 -                     f_col = 0;
271 -                   end
272 -                 end
273 -               end
274 -             else
275 -               f_col = 0;
276 -             end
277 -             if f_col == 0
278 -               break
279 -             else
280 -               R = R+deltaR;
281 -             end
282 -           end
283 -
284 -           P_ref_n = punto_destino;
285 -           puntos_elegidos = [puntos_elegidos;P_ref_n];
286 -           while 1
287 -             if norm(P_ref_n-P0) < 10
288 -               flag_destino = 1;
289 -               break
290 -             end
291 -             if f_aux_2 == 1
292 -               global_counter = 0;
293 -               ciclo_completo(P0,phi0,T,1,N)
294 -               f_aux_2 = 0;
295 -               distancias_n = zeros(size(nube,1),1);
296 -               for i=1:size(nube,1)
297 -                 distancias_n(i) = norm(P_ref_n-nube(i,1:2));
298 -               end
299 -             end
300 -
301 -             encontrar_punto(distancias_n)
302 -
```

```
303 -             pos_0 = P_mas_cercano;
304 -             ori_0 = ori_mas_cercano;
305 -
306 -             for h = 1:N
307 -
308 -               if secuencia_encontrada(h)==-1 || secuencia_encontrada(h+N)==-1
309 -                 break
310 -               end
311 -
312 -               indice_ini = secuencia_encontrada(h);
313 -               indice_fin = secuencia_encontrada(h+N);
314 -
315 -               movB = 1;
316 -               T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
317 -                      sin(ori_0)  cos(ori_0) pos_0(2);
318 -                           0           0         1  ];
319 -               area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
320 -               [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,
                                             ori_0,indice_ini,T_p,T,area);
321 -               if colision == 1
322 -                 break
323 -               end
324 -
325 -               movB = 0;
326 -               T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
327 -                      sin(ori_0)  cos(ori_0) pos_0(2);
328 -                           0           0         1  ];
329 -               area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
330 -               [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,
                                             ori_0,indice_fin,T_p,T,area);
331 -               if colision == 1
332 -                 break
333 -               end
334 -
335 -             end
336 -
337 -             if colision == 0
338 -               registro_seq = [registro_seq; secuencia_encontrada];
339 -               puntos_encontrados = [puntos_encontrados; P_mas_cercano];
340 -               ori_encontradas = [ori_encontradas; ori_mas_cercano];
341 -               P0 = P_mas_cercano;
342 -               phi0 = ori_mas_cercano;
343 -               f_aux_2 = 1;
344 -               nube_grande = [nube_grande;nube];
345 -             else
346 -               distancias_n(indice_min) = 1000000000;
347 -             end
348 -           end
349 -         end
350 -
351 -         vect_eng = repulsor-P_ref_n;
352 -         u_eng = vect_eng/norm(vect_eng);
353 -         P_inicio_eng = P_ref_n+u_eng*120.65;
354 -         P_final_eng = repulsor;
355 -         alpha = atan2(vect_eng(2),vect_eng(1));
356 -         P1 = P_inicio_eng + 60.325*[cos(alpha-pi/2) sin(alpha-pi/2)];
357 -         P2 = P_inicio_eng + 60.325*[cos(alpha+pi/2) sin(alpha+pi/2)];
358 -         P3 = P_final_eng + 60.325*[cos(alpha+pi/2) sin(alpha+pi/2)];
359 -         P4 = P_final_eng + 60.325*[cos(alpha-pi/2) sin(alpha-pi/2)];
360 -
361 -         resol_rect = 10;
362 -         x12 = P1(1)+(P2(1)-P1(1))*[0:resol_rect/norm(P2-P1):1];
```

```
363 -         y12 = P1(2)+(P2(2)-P1(2))*[0:resol_rect/norm(P2-P1):1];
364 -         x23 = P2(1)+(P3(1)-P2(1))*[0:resol_rect/norm(P3-P2):1];
365 -         y23 = P2(2)+(P3(2)-P2(2))*[0:resol_rect/norm(P3-P2):1];
366 -         x34 = P3(1)+(P4(1)-P3(1))*[0:resol_rect/norm(P4-P3):1];
367 -         y34 = P3(2)+(P4(2)-P3(2))*[0:resol_rect/norm(P4-P3):1];
368 -         x41 = P4(1)+(P1(1)-P4(1))*[0:resol_rect/norm(P1-P4):1];
369 -         y41 = P4(2)+(P1(2)-P4(2))*[0:resol_rect/norm(P1-P4):1];
370 -
371 -         P_obs = [P_obs;[x12' y12'];[x23' y23'];[x34' y34'];[x41' y41']];
372 -         Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,
                    'BucketSize',1);
373 -
374 -       end
375 -     end
376 -     nube_grande = [nube_grande;nube];
377 -   else
378 -     distancias(indice_min) = 1000000000;
379 -   end
380 -
381 - end
382 -
383 - ori_final;
384 - dibujo;
385 -
386 - save('resultados')
```

fr_2D_malla.m

```
  1 - function frontera = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax)
  2 -   global Md
  3 -
  4 -   Dx = xmax-xmin;
  5 -   Dy = ymax-ymin;
  6 -   g = round((Dx)/paso_x)+1;
  7 -   h = round((Dy)/paso_y)+1;
  8 -   hay_obstaculo = zeros(g,h);
  9 -   frontera = [];
 10 -   v_paso = [paso_x,paso_y];
 11 -   radio = min(v_paso)/2;
 12 -
 13 -   for i=1:g
 14 -     x = (i-1)*paso_x+xmin;
 15 -     for j = 1:h
 16 -       y = (j-1)*paso_y + ymin;
 17 -       pto = [x y];
 18 -       Idx = rangesearch(Md,pto,radio);
 19 -       Retrieved = zeros(length(Idx{1}),2);
 20 -       if isempty(Idx{1}) == 0
 21 -         hay_obstaculo(i,j) = 1;
 22 -       end
 23 -     end
 24 -   end
 25 -
 26 -   for b = 1:g
 27 -     for c = 1:h
 28 -
 29 -       columna = b;
 30 -       fila = c;
 31 -       if hay_obstaculo(columna,fila) == 1
 32 -         vecinos = zeros(8,2);
 33 -         z = 1;
 34 -         for i = -1:1
 35 -           for j = -1:1
```

```
            if (i~=0 || j~=0)
              vecinos(z,1) = columna+i;
              vecinos(z,2) = fila+j;
              z = z+1;
            end
          end
        end
        for k = 1:8
          e = vecinos(k,1);
          d = vecinos(k,2);
          if e>g || d>h || e<1 || d<1
            continue
          end
          if hay_obstaculo(e,d) == 1
            continue
          else
            p = [columna fila];
            frontera = [frontera; [(p(1)-1)*paso_x+xmin,(p(2)-1)*paso_y+ymin]];
          end
        end
      end
    end
  end

end
```

area_local.m

```
function [areaA,areaB] = area_local(P0,phi0)

  v = [1,2,3,4,5,6,7,8,1];
  BarrB = [];
  BarrA = [];
  for t = 1:8
    BarrB = calculo_area_adyac(v(t),v(t+1),1,P0,phi0,BarrB);
  end
  areaB = BarrB;
  v_aux = 101.31*ones(1,20);
  v_aux_p = -50.242:5.2886:50.242;
  v_t = transpose([v_aux; v_aux_p; ones(1,20)]);
  v_t_2 = transpose([-v_aux; v_aux_p; ones(1,20)]);
  areaB = [areaB(40:80,1) areaB(40:80,2);
           v_t(:,1) v_t(:,2);
           areaB(120:160,1) areaB(120:160,2);
           areaB(1:41,3)  areaB(1:41,4);
           v_t_2(:,1) v_t_2(:,2);
           areaB(80:120,3) areaB(80:120,4)];
  for u = 1:8
    BarrA = calculo_area_adyac(v(u),v(u+1),0,P0,phi0,BarrA);
  end
  areaA = BarrA;
  v_aux_2p = -9.295:3.718:9.295;
  v_aux_3p = 120.242*ones(1,6);
  v_aux_4p = 75.668*ones(1,9);
  v_aux_5p = -22.5656:5.6414:22.5656;
  v_t_p = transpose([v_aux_2p; v_aux_3p; ones(1,6)]);
  v_t_2p = transpose([v_aux_4p; v_aux_5p; ones(1,9)]);
  v_t_3p = transpose([v_aux_2p; -1*v_aux_3p; ones(1,6)]);
  v_t_4p = transpose([-1*v_aux_4p; v_aux_5p; ones(1,9)]);
  areaA = [areaA(52:80,1) areaA(52:80,2);
           v_t_p(:,1) v_t_p(:,2);
           areaA(80:110,3) areaA(80:110,4);
```

```
35 -            v_t_2p(:,1) v_t_2p(:,2);
36 -            areaA(1:30,7) areaA(1:30,8);
37 -            v_t_3p(:,1) v_t_3p(:,2);
38 -            areaA(131:160,5) areaA(131:160,6);
39 -            v_t_4p(:,1) v_t_4p(:,2)];
40 -
41 - end
```

calculo_area_adyac.m

```
 1 - function Barr = calculo_area_adyac(i,f,movB,pos_0,ori_0,Barr)
 2 -   m = 20;
 3 -   p = 101.31;
 4 -   b = 18.59;
 5 -   Barr_aux = Barr;
 6 -   phi_v = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
 7 -   y_v = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
           21.95478428];
 8 -
 9 -   phi_inicial = phi_v(i);
10 -   y_inicial = y_v(i);
11 -   phi_final = phi_v(f);
12 -   y_final = y_v(f);
13 -
14 -   [li,ri] = inversa(phi_inicial,y_inicial);
15 -   [lf,rf] = inversa(phi_final,y_final);
16 -   dl = (lf-li)/(m-1);
17 -   dr = (rf-ri)/(m-1);
18 -
19 -   l = li;
20 -   r = ri;
21 -   phi = phi_inicial;
22 -   y = y_inicial;
23 -
24 -   for i = 1:m
25 -
26 -     [phi,y] = directa(l,r,phi,y);
27 -     l = l+dl;
28 -     r = r+dr;
29 -
30 -     T_BA = [ cos(phi)  -sin(phi)    0;
31 -              sin(phi)   cos(phi)    y;
32 -                 0          0        1];
33 -     if movB == 1
34 -       T_A = [ cos(ori_0) -sin(ori_0)  pos_0(1);
35 -               sin(ori_0) cos(ori_0)   pos_0(2);
36 -                    0          0           1   ];
37 -       T_B = T_A*T_BA;
38 -       B1 = T_B*[p;0;1];
39 -       B2 = T_B*[-p;0;1];
40 -       Barr_aux = [Barr_aux; B1(1) B1(2) B2(1) B2(2)];
41 -     else
42 -       T_B = [cos(ori_0) -sin(ori_0)  pos_0(1);
43 -              sin(ori_0) cos(ori_0)   pos_0(2);
44 -                   0          0           1   ];
45 -       T_A = T_B*inv(T_BA);
46 -       altura_pasador = 70;
47 -       radio_pasador = b/2;
48 -       a11 = T_A*[-radio_pasador;altura_pasador;1];
49 -       a12 = T_A*[ radio_pasador;altura_pasador;1];
50 -       a21 = T_A*[-radio_pasador;-altura_pasador;1];
51 -       a22 = T_A*[ radio_pasador;-altura_pasador;1];
52 -       Barr_aux = [Barr_aux;
```

```
                    a11(1) a11(2) a12(1) a12(2) a21(1) a21(2) a22(1) a22(2)];
53 -    end
54 -  end
55 -
56 -  Barr = Barr_aux;
57 -
58 - end
```

inversa.m

```
1 - function [l,r] = inversa(phi,y)
2 -   b = 18.59;
3 -   p = 101.31;
4 -   l = sqrt((p*cos(phi)+b)^2+(y+p*sin(phi))^2);
5 -   r = sqrt((-p*cos(phi)-b)^2+(y-p*sin(phi))^2);
6 - end
```

directa.m

```
 1 - function [phi,y] = directa(l,r,phi_p,y_p)
 2 -   b = 18.59;
 3 -   p = 101.31;
 4 -   F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
 5 -        (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
 6 -   J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
 7 -        -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
 8 -   aux = [phi_p;y_p];
 9 -
10 -   for iter = 1:10
11 -     aux = aux - J\F;
12 -     phi_p = aux(1);
13 -     y_p = aux(2);
14 -     J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
15 -          -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
16 -     F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2;
17 -          (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
18 -   end
19 -   phi = aux(1);
20 -   y = aux(2);
21 -
22 - end
```

ciclo_completo.m

```
 1 - function ciclo_completo(pos_0,ori_0,T,n,N)
 2 -   global nube
 3 -   global global_counter
 4 -   global secuencia
 5 -   v_i=-1*ones(1,N);
 6 -   v_j=-1*ones(1,N);
 7 -   if n<=N
 8 -
 9 -     TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
10 -             sin(ori_0), cos(ori_0),pos_0(2);
11 -                  0 ,      0     ,    1   ];
12 -
13 -     for i=1:8
14 -       TB = TA_0*T{i};
15 -       for j=1:8
16 -         TA = TB*inv(T{j});
17 -         pos = TA(1:2,3)';
18 -         ori = atan2(TA(2,1),TA(1,1));
19 -         global_counter = global_counter + 1;
20 -         nube(global_counter,:) = [pos,ori];
```

```
21 -          if n==1
22 -            v_i(1)=i;
23 -            v_j(1)=j;
24 -            secuencia(global_counter,:) = [v_i v_j];
25 -          else
26 -            auxiliar=secuencia(global_counter-1,:);
27 -            auxiliar(n)=i;
28 -            auxiliar(N+n)=j;
29 -            secuencia(global_counter,:) = auxiliar;
30 -          end
31 -          ciclo_completo(pos,ori,T,n+1,N);
32 -        end
33 -      end
34 -    end
35 - end
```

encontrar_punto.m

```
 1 - function encontrar_punto(distancias)
 2 -   global nube
 3 -   global indice_min
 4 -   global secuencia_encontrada
 5 -   global P_mas_cercano
 6 -   global ori_mas_cercano
 7 -   global secuencia
 8 -   [~,indice_min] = min(distancias);
 9 -   P_mas_cercano = nube(indice_min,1:2);
10 -   ori_mas_cercano = nube(indice_min,3);
11 -   secuencia_encontrada = secuencia(indice_min,:);
12 - end
```

calculo_area.m

```
1 - function area = calculo_area(movB,pos_0,ori_0,T_p,area)
2 -   area_aux = [area(:,1) area(:,2) ones(length(area),1)];
3 -   area_aux = transpose(T_p*transpose(area_aux));
4 -   area = [area_aux(:,1) area_aux(:,2)];
5 - end
```

comprobacion_colisiones.m

```
 1 - function [colision,pos_F,ori_F] = comprobacion_colisiones(movB,pos_0,ori_0,
                                       indice,T_p,T,area)
 2 -   Retrieved = codigo_kdtree(pos_0,area);
 3 -   if length(Retrieved) == 0
 4 -     colision = 0;
 5 -   else
 6 -     Retrieved_L = T_p\[Retrieved(:,1) Retrieved(:,2)
                          ones(length(Retrieved(:,1)),1)]';
 7 -     x_obs_L = Retrieved_L(1,:);
 8 -     y_obs_L = Retrieved_L(2,:);
 9 -     for j = 1:length(x_obs_L)
10 -       if movB == 1
11 -         if x_obs_L(j)^2+y_obs_L(j)^2<=13891.29 && x_obs_L(j)>=-101.31 && ...
12 -           x_obs_L(j)<=101.31 && y_obs_L(j)>=-103.1332 && y_obs_L(j)<=103.1332
13 -           colision = 1;
14 -           break
15 -         else
16 -           colision = 0;
17 -         end
18 -       else
19 -         if 0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 &&
               -0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 && ...
20 -           -0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 &&
               0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 && ...
```

```
21 -            x_obs_L(j)<=75.668 && x_obs_L(j)>=-75.668 &&
                y_obs_L(j)<=120.2 && y_obs_L(j)>=-120.2
22 -            colision = 1;
23 -            break
24 -          else
25 -            colision = 0;
26 -          end
27 -        end
28 -      end
29 -    end
30 -    if movB == 1
31 -      T_rel = T{indice};
32 -      T_new = T_p*T_rel;
33 -    else
34 -      T_rel = T{indice};
35 -      T_new = T_p/T_rel;
36 -    end
37 -    pos_F = [T_new(1,3) T_new(2,3)];
38 -    ori_F = atan2(T_new(2,1),T_new(2,2));
39 - end
```

codigo_kdtree.m

```
 1 - function Retrieved = codigo_kdtree(pos_0,area)
 2 -   global Md1
 3 -   global P_obs
 4 -   p1 = min(area(:,1));
 5 -   p2 = max(area(:,2));
 6 -   p3 = max(area(:,1));
 7 -   p4 = min(area(:,2));
 8 -   pt = [p1,p2,p3,p4];
 9 -   p = max(pt);
10 -   punto_cuadrado = [p,pos_0(1)];
11 -   centro = pos_0;
12 -   radio = norm(pos_0-punto_cuadrado);
13 -   Idx = rangesearch(Md1,centro,radio);
14 -   Retrieved = zeros(length(Idx{1}),2);
15 -   for i=1:length(Idx{1})
16 -     Retrieved(i,:) = P_obs(Idx{1}(i),:);
17 -   end
18 - end
```

LineFitting.m

```
 1 - function [pf1,pf2] = LineFitting(Retrieved)
 2 -
 3 -   tam = length(Retrieved(:,1));
 4 -   iteraciones = 25;
 5 -   contador_prev = 0;
 6 -
 7 -   for i = 1:iteraciones
 8 -     i1 = randi(tam);
 9 -     i2 = randi(tam);
10 -     p1 = Retrieved(i1,:);
11 -     p2 = Retrieved(i2,:);
12 -     v = p2-p1;
13 -     A = v(2)/v(1);
14 -     B = -1;
15 -     C = -B*p1(2)-A*p1(1);
16 -     contador = 0;
17 -     for j = 1:tam
18 -       if Retrieved(j,1) == 0 && Retrieved(j,2) == 0
19 -         continue
```

```
      end
      p = Retrieved(j,:);
      d = abs((A*p(1)+B*p(2)+C)/sqrt(A^2+B^2));
      if d < 5 %umbral
        contador = contador+1;
      end
    end
    if contador > contador_prev
      pf1 = p1;
      contador_prev = contador;
    end
    contador_prev = contador;
  end

end
```

ori_final.m

```
load resultado_ori_final;
v_indices = [];
v_dist = [];
T_encontrado = eye(3);
phi_dif = phi_ref-nube(indice_min,3);
if phi_dif > pi
  phi_dif = phi_dif-2*pi;
elseif phi_dif < -pi
  phi_dif = phi_dif+2*pi;
end
for x = 1:length(nube_fin(:,1))
  if abs(nube_fin(x,3)-phi_dif) < 0.001
    v_indices = [v_indices;x];
  end
end
for y = 1:size(v_indices)
  v_aux = nube_fin(v_indices(y),1:2);
  theta = atan2(v_aux(2),v_aux(1));
  phi_dif_aux = theta+nube(indice_min,3);
  mod = norm(v_aux);
  P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
           P_mas_cercano(2)+mod*sin(phi_dif_aux)];
  dist = norm(P_aux-P_ref);
  v_dist = [v_dist;dist];
end
[~,ind_tmp] = min(v_dist);
secuencia_final = seq_fin(v_indices(ind_tmp),:);
for k = 1:4
  i = secuencia_final(k);
  j = secuencia_final(k+4);
  if i==-1 || j==-1
    continue
  end
  T_encontrado = T_encontrado*T{i}/T{j};
end
T0 = [ cos(nube(indice_min,3)) -sin(nube(indice_min,3)) nube(indice_min,1);
       sin(nube(indice_min,3))  cos(nube(indice_min,3)) nube(indice_min,2);
                    0                        0                    1        ];

T1 = T0*T_encontrado;
P_final = T1(1:2,3)';
phi_final = atan2(T1(2,1),T1(1,1));
```

dibujo.m

```
plot(nube_grande(:,1),nube_grande(:,2),'.b');
set(gca,'DataAspectRatio',[1,1,1]);
hold on
axis([xmin xmax ymin ymax]);

for d = 1:size(P_obs_p,1)
    P = P_obs_p(d,:);
    plot(P(1),P(2),'.k');
end
for d = 1:size(P_obs,1)
    P = P_obs(d,:);
    plot(P(1),P(2),'.r');
end

plot(P_ref(1),P_ref(2),'or')
plot(posicion_inicial(1),posicion_inicial(2),'.g','MarkerSize',25)
plot(puntos_elegidos(:,1),puntos_elegidos(:,2),'*y')
quiver(P_ref(1),P_ref(2),50*cos(phi_ref),50*sin(phi_ref),'r')
plot(P_mas_cercano(1),P_mas_cercano(2),'*m')
quiver(P_mas_cercano(1),P_mas_cercano(2),50*cos(nube(indice_min,3)),
       50*sin(nube(indice_min,3)),'m')
plot(P_final(1),P_final(2),'sg')
quiver(P_final(1),P_final(2),50*cos(phi_final),50*sin(phi_final),'g');
```

ANEXO 3

ws_binario_mp.m

```
clear all;
global nube
global global_counter
global secuencia
global indice_min
global secuencia_encontrada
global P_mas_cercano
global ori_mas_cercano
global Md
global Md1
global P_obs
N = 2;
P_ref = [600 500];
phi_ref = pi/4;
P0 = [0,0];
posicion_inicial = P0;
phi0 = 0;
P_obs = [];
P_obs_p = [];
paso_x = 10;
paso_y = 10;
for xobs = 400-300:paso_x:400+300
  for yobs = 300-50:paso_y:300+50
    if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end
for xobs = 300-50:paso_x:300+50
  for yobs = 400-300:paso_y:400+300
    if (xobs-300)^2/50^2 + (yobs-400)^2/300^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end
% Comprobación de si el punto inicial o final se encuentran obstaculizados
for i = 1:length(P_obs_p(:,1))
  if norm(P0-P_obs_p(i,:)) < 120.65
    disp('Error: el punto de inicio se encuentra muy cerca de un obstaculo.')
    pausa1 = 1;
    break
  else
  pausa1 = 0;
  end
  if norm(P_ref-P_obs_p(i,:)) < 120.65
    disp('Error: el punto final se encuentra muy cerca de un obstaculo.')
    pausa2 = 1;
    break
  else
    pausa2 = 0;
  end
end
if pausa1 == 1 || pausa2 == 1
  pause
end
Md = KDTreeSearcher(P_obs_p,'Distance','minkowski','P',inf,'BucketSize',1);
xmin = -300; xmax = 1200; ymin = -300; ymax = 1200;
```

```
P_obs = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax);
Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
tamanyo = 0;
for i=1:N
  tamanyo = tamanyo + 64^i;
end
nube = zeros(tamanyo,3);
global_counter = 0;
secuencia = (-1)*ones(tamanyo,2*N);
T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
registro_seq = [];
nube_grande = [];
puntos_encontrados = [];
ori_encontradas = [];
% Depende de: b, p, rho0, Delta_rho
phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
     21.95478428];
for i=1:8
  T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
end
[areaA,areaB] = area_local(P0,phi0);
resol_A_estrella = 10;
route = A_estrella(resol_A_estrella,resol_A_estrella,P0(1),P0(2),P_ref(1),
        P_ref(2),xmin,xmax,ymin,ymax);
route = [P_ref;route;P0];
puntos_dest = [P_ref;P0];
ind_dest = [1,length(route(:,1))];
umbral = 50;
while 1

  for i = 1:length(ind_dest)-1
    ind1 = ind_dest(i);
    ind2 = ind_dest(i+1);
    P1 = puntos_dest(i,:);
    P2 = puntos_dest(i+1,:);
    [destino,dist_max,ind_max] = extractor_destinos(route(ind1:ind2,:),P1,P2);
    if dist_max > umbral
      flag_1 = 0;
      break
    else
      flag_1 = 1;
    end
  end

  if flag_1 == 1
    break
  else
    tam_ind_dest = length(ind_dest);
    ind_act = ind_max + ind1;
    for j = 1:tam_ind_dest
      if ind_dest(j) > ind_act
        ind_dest = [ind_dest(1:j-1) ind_act ind_dest(j:tam_ind_dest)];
        puntos_dest = [puntos_dest(1:j-1,:); destino;
                       puntos_dest(j:tam_ind_dest,:)];
        break
      end
    end
  end

end
for i = length(ind_dest):-1:1
  P_ref_actual = puntos_dest(i,:);
```

```
118 -   f_aux = 1;
119 -   while 1
120 -     if norm(P_ref_actual-P0) < 10
121 -       break
122 -     end
123 -     if f_aux == 1
124 -       global_counter = 0;
125 -       ciclo_completo(P0,phi0,T,1,N);
126 -       f_aux = 0;
127 -       distancias = zeros(size(nube,1),1);
128 -       for j=1:size(nube,1)
129 -         distancias(j) = norm(P_ref_actual-nube(j,1:2));
130 -       end
131 -     end
132 -     encontrar_punto(distancias)
133 -     pos_0 = P_mas_cercano;
134 -     ori_0 = ori_mas_cercano;
135 -     for h = 1:N
136 -       if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
137 -         break
138 -       end
139 -       indice_inicio = secuencia_encontrada(h);
140 -       indice_fin = secuencia_encontrada(h+N);
141 -       movB = 1;
142 -       T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
143 -              sin(ori_0)  cos(ori_0) pos_0(2);
144 -                   0           0         1  ];
145 -       area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
146 -       [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                     indice_inicio,T_p,T,area);
147 -       if colision == 1
148 -         break
149 -       end
150 -       movB = 0;
151 -       T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
152 -              sin(ori_0)  cos(ori_0) pos_0(2);
153 -                   0           0         1  ];
154 -       area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
155 -       [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                     indice_fin,T_p,T,area);
156 -       if colision == 1
157 -         break
158 -       end
159 -     end
160 -     if colision == 0
161 -       registro_seq = [registro_seq; secuencia_encontrada];
162 -       puntos_encontrados = [puntos_encontrados; P_mas_cercano];
163 -       ori_encontradas = [ori_encontradas; ori_mas_cercano];
164 -       P0 = P_mas_cercano;
165 -       phi0 = ori_mas_cercano;
166 -       f_aux = 1;
167 -       nube_grande = [nube_grande;nube];
168 -     else
169 -       distancias(indice_min) = 1000000000;
170 -     end
171 -   end
172 -
173 - end
174 - ori_final
175 - dibujo
176 - save('resultados')
```

fr_2D_malla.m

```
function frontera = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax)
  global Md

  Dx = xmax-xmin;
  Dy = ymax-ymin;
  g = round((Dx)/paso_x)+1;
  h = round((Dy)/paso_y)+1;
  hay_obstaculo = zeros(g,h);
  frontera = [];
  v_paso = [paso_x,paso_y];
  radio = min(v_paso)/2;

  for i=1:g
    x = (i-1)*paso_x+xmin;
    for j = 1:h
      y = (j-1)*paso_y + ymin;
      pto = [x y];
      Idx = rangesearch(Md,pto,radio);
      Retrieved = zeros(length(Idx{1}),2);
      if isempty(Idx{1}) == 0
        hay_obstaculo(i,j) = 1;
      end
    end
  end

  for b = 1:g
    for c = 1:h

      columna = b;
      fila = c;
      if hay_obstaculo(columna,fila) == 1
        vecinos = zeros(8,2);
        z = 1;
        for i = -1:1
          for j = -1:1
            if (i~=0 || j~=0)
              vecinos(z,1) = columna+i;
              vecinos(z,2) = fila+j;
              z = z+1;
            end
          end
        end
        for k = 1:8
          e = vecinos(k,1);
          d = vecinos(k,2);
          if e>g || d>h || e<1 || d<1
            continue
          end
          if hay_obstaculo(e,d) == 1
            continue
          else
            p = [columna fila];
            frontera = [frontera; [(p(1)-1)*paso_x+xmin,(p(2)-1)*paso_y+ymin]];
          end
        end
      end
    end
  end

```

```
60 - end
```

area_local.m

```
 1 - function [areaA,areaB] = area_local(P0,phi0)
 2 -
 3 -   v = [1,2,3,4,5,6,7,8,1];
 4 -   BarrB = [];
 5 -   BarrA = [];
 6 -   for t = 1:8
 7 -     BarrB = calculo_area_adyac(v(t),v(t+1),1,P0,phi0,BarrB);
 8 -   end
 9 -   areaB = BarrB;
10 -   v_aux = 101.31*ones(1,20);
11 -   v_aux_p = -50.242:5.2886:50.242;
12 -   v_t = transpose([v_aux; v_aux_p; ones(1,20)]);
13 -   v_t_2 = transpose([-v_aux; v_aux_p; ones(1,20)]);
14 -   areaB = [areaB(40:80,1) areaB(40:80,2);
15 -            v_t(:,1) v_t(:,2);
16 -            areaB(120:160,1) areaB(120:160,2);
17 -            areaB(1:41,3)  areaB(1:41,4);
18 -            v_t_2(:,1) v_t_2(:,2);
19 -            areaB(80:120,3) areaB(80:120,4)];
20 -   for u = 1:8
21 -     BarrA = calculo_area_adyac(v(u),v(u+1),0,P0,phi0,BarrA);
22 -   end
23 -   areaA = BarrA;
24 -   v_aux_2p = -9.295:3.718:9.295;
25 -   v_aux_3p = 120.242*ones(1,6);
26 -   v_aux_4p = 75.668*ones(1,9);
27 -   v_aux_5p = -22.5656:5.6414:22.5656;
28 -   v_t_p = transpose([v_aux_2p; v_aux_3p; ones(1,6)]);
29 -   v_t_2p = transpose([v_aux_4p; v_aux_5p; ones(1,9)]);
30 -   v_t_3p = transpose([v_aux_2p; -1*v_aux_3p; ones(1,6)]);
31 -   v_t_4p = transpose([-1*v_aux_4p; v_aux_5p; ones(1,9)]);
32 -   areaA = [areaA(52:80,1) areaA(52:80,2);
33 -            v_t_p(:,1) v_t_p(:,2);
34 -            areaA(80:110,3) areaA(80:110,4);
35 -            v_t_2p(:,1) v_t_2p(:,2);
36 -            areaA(1:30,7) areaA(1:30,8);
37 -            v_t_3p(:,1) v_t_3p(:,2);
38 -            areaA(131:160,5) areaA(131:160,6);
39 -            v_t_4p(:,1) v_t_4p(:,2)];
40 -
41 - end
```

calculo_area_adyac.m

```
 1 - function Barr = calculo_area_adyac(i,f,movB,pos_0,ori_0,Barr)
 2 -   m = 20;
 3 -   p = 101.31;
 4 -   b = 18.59;
 5 -   Barr_aux = Barr;
 6 -   phi_v = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
 7 -   y_v = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
           21.95478428];
 8 -
 9 -   phi_inicial = phi_v(i);
10 -   y_inicial = y_v(i);
11 -   phi_final = phi_v(f);
12 -   y_final = y_v(f);
13 -
14 -   [li,ri] = inversa(phi_inicial,y_inicial);
```

```
15 -   [lf,rf] = inversa(phi_final,y_final);
16 -   dl = (lf-li)/(m-1);
17 -   dr = (rf-ri)/(m-1);
18 -
19 -   l = li;
20 -   r = ri;
21 -   phi = phi_inicial;
22 -   y = y_inicial;
23 -
24 -   for i = 1:m
25 -
26 -     [phi,y] = directa(l,r,phi,y);
27 -     l = l+dl;
28 -     r = r+dr;
29 -
30 -     T_BA = [ cos(phi)  -sin(phi)    0;
31 -              sin(phi)   cos(phi)    y;
32 -                 0          0        1];
33 -     if movB == 1
34 -       T_A = [ cos(ori_0) -sin(ori_0)  pos_0(1);
35 -               sin(ori_0) cos(ori_0)   pos_0(2);
36 -                   0          0           1   ];
37 -       T_B = T_A*T_BA;
38 -       B1 = T_B*[p;0;1];
39 -       B2 = T_B*[-p;0;1];
40 -       Barr_aux = [Barr_aux; B1(1) B1(2) B2(1) B2(2)];
41 -     else
42 -       T_B = [cos(ori_0) -sin(ori_0)  pos_0(1);
43 -              sin(ori_0) cos(ori_0)   pos_0(2);
44 -                  0          0           1   ];
45 -       T_A = T_B*inv(T_BA);
46 -       altura_pasador = 70;
47 -       radio_pasador = b/2;
48 -       a11 = T_A*[-radio_pasador;altura_pasador;1];
49 -       a12 = T_A*[ radio_pasador;altura_pasador;1];
50 -       a21 = T_A*[-radio_pasador;-altura_pasador;1];
51 -       a22 = T_A*[ radio_pasador;-altura_pasador;1];
52 -       Barr_aux = [Barr_aux;
                   a11(1) a11(2) a12(1) a12(2) a21(1) a21(2) a22(1) a22(2)];
53 -     end
54 -   end
55 -
56 -   Barr = Barr_aux;
57 -
58 - end
```

inversa.m

```
1 - function [l,r] = inversa(phi,y)
2 -   b = 18.59;
3 -   p = 101.31;
4 -   l = sqrt((p*cos(phi)+b)^2+(y+p*sin(phi))^2);
5 -   r = sqrt((-p*cos(phi)-b)^2+(y-p*sin(phi))^2);
6 - end
```

directa.m

```
1 - function [phi,y] = directa(l,r,phi_p,y_p)
2 -   b = 18.59;
3 -   p = 101.31;
4 -   F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
5 -        (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
6 -   J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
```

```
 7 -         -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
 8 -   aux = [phi_p;y_p];
 9 -
10 -   for iter = 1:10
11 -     aux = aux - J\F;
12 -     phi_p = aux(1);
13 -     y_p = aux(2);
14 -     J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
15 -          -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
16 -     F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2;
17 -          (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
18 -   end
19 -   phi = aux(1);
20 -   y = aux(2);
21 -
22 - end
```

A_estrella.m

```
 1 - function route =
A_estrella(paso_x,paso_y,xin,yin,xfin,yfin,xmin,xmax,ymin,ymax)
 2 -   global P_obs
 3 -   global Md1
 4 -   route = [];
 5 -   Dx = xmax-xmin;
 6 -   Dy = ymax-ymin;
 7 -   g = round((Dx)/paso_x)+1;
 8 -   h = round((Dy)/paso_y)+1;
 9 -
10 -   ClosedSet = zeros(g,h);
11 -   OpenSet = zeros(g,h);
12 -   xf = round((xfin-xmin)/paso_x)+1;
13 -   yf = round((yfin-ymin)/paso_y)+1;
14 -   l = round((xin-xmin)/paso_x)+1;
15 -   m = round((yin-ymin)/paso_y)+1;
16 -   OpenSet(l,m) = 1;
17 -   Came_From = cell(g,h);
18 -   for i = 1:g
19 -     for j = 1:h
20 -       Came_From{i,j} = [-1,-1];
21 -     end
22 -   end
23 -   g_score = ClosedSet;
24 -   for i = 1:g
25 -     for j = 1:h
26 -       g_score(i,j) = 100000000;
27 -     end
28 -   end
29 -   g_score(l,m) = 0;
30 -   f_score = 100000000*ones(g,h);
31 -   d = sqrt((xf-l)^2+(yf-m)^2);
32 -   f_score(l,m) = d;
33 -   while (sum(sum(OpenSet))~=0)
34 -     min_value = 100000000;
35 -     for i = 1:g
36 -       for j = 1:h
37 -         if OpenSet(i,j) == 1
38 -           if f_score(i,j) < min_value
39 -             min_value = f_score(i,j);
40 -             fila = i;
41 -             columna = j;
42 -           end
43 -         end
```

```
 44 -       end
 45 -     end
 46 -     if (fila == xf && columna == yf)
 47 -       [path] = reconstruct_path(Came_From,xf,yf,l,m);
 48 -       for i = 2:length(path)-1
 49 -         p = path{1,i};
 50 -         route = [route;(p(1)-1)*paso_x+xmin (p(2)-1)*paso_y+ymin];
 51 -       end
 52 -       return
 53 -     end
 54 -     OpenSet(fila,columna) = 0;
 55 -     ClosedSet(fila,columna) = 1;
 56 -     vecinos = zeros(8,2);
 57 -     z = 1;
 58 -     for i = -1:1
 59 -       for j = -1:1
 60 -         if (i~=0 || j~=0)
 61 -           vecinos(z,1) = fila+i;
 62 -           vecinos(z,2) = columna+j;
 63 -           z = z+1;
 64 -         end
 65 -       end
 66 -     end
 67 -     X = (fila-1)*paso_x + xmin;
 68 -     Y = (columna-1)*paso_y + ymin;
 69 -     radio = 120.65;
 70 -     centro = [X Y];
 71 -
 72 -     Idx = rangesearch(Md1,centro,radio);
 73 -     Retrieved = zeros(length(Idx{1}),2);
 74 -     for k = 1:length(Idx{1})
 75 -       Retrieved(k,:) = P_obs(Idx{1}(k),:);
 76 -     end
 77 -
 78 -     flag_invalido = 0;
 79 -     x_Retrieved = Retrieved(:,1);
 80 -     y_Retrieved = Retrieved(:,2);
 81 -     for w = 1:length(x_Retrieved)
 82 -       p_Retrieved = [x_Retrieved(w) y_Retrieved(w)];
 83 -       if norm(centro-p_Retrieved) <= radio
 84 -         f_score(fila,columna) = 10^10;
 85 -         flag_invalido = 1;
 86 -         break
 87 -       end
 88 -     end
 89 -
 90 -     if flag_invalido == 0
 91 -       for i = 1:8
 92 -         a = vecinos(i,1);
 93 -         b = vecinos(i,2);
 94 -         if a<=0 || a>g || b<=0 || b>h
 95 -           continue
 96 -         end
 97 -         if ClosedSet(a,b) == 1
 98 -           continue
 99 -         end
100 -         tentative_g_score = g_score(fila,columna)+
                                  sqrt((fila-a)^2+(columna-b)^2);
101 -         if OpenSet(a,b) == 0
102 -           OpenSet(a,b) = 1;
103 -         elseif tentative_g_score >= g_score(a,b)
104 -           continue
```

```
105 -         end
106 -         Came_From{a,b} = [fila,columna];
107 -         g_score(a,b) = tentative_g_score;
108 -         f_score(a,b) = g_score(a,b)+sqrt((a-xf)^2+(b-yf)^2);
109 -       end
110 -     end
111 -   end
112 - end
113 - function [path] = reconstruct_path(Came_From,xf,yf,l,m)
114 -   [gen,g] = size(Came_From);
115 -   gen = (gen*gen);
116 -   nodo = [xf,yf];
117 -   paso = cell(1,gen);
118 -   for i = 1:gen
119 -     if (nodo ~= [-1,-1])
120 -       paso{1,i} = nodo;
121 -       nodo = Came_From{nodo(1),nodo(2)};
122 -     end
123 -   end
124 -   cantidad = 1;
125 -   while (paso{1,cantidad}(1)~=l || paso{1,cantidad}(2)~=m)
126 -     cantidad = cantidad + 1;
127 -   end
128 -   path = cell(1,cantidad);
129 -   for i = 1:cantidad
130 -     path{1,i} = paso{1,i};
131 -   end
132 - end
```

extractor_destinos.m

```
1 - function [destino,dist_max,ind_max] = extractor_destinos(route,P1,P2)
2 -
3 -   tam = length(route(:,1));
4 -   dist_vector = zeros(tam,1);
5 -
6 -   v = P1-P2;
7 -   A = v(2)/v(1);
8 -   B = -1;
9 -   C = P1(2)-A*P1(1);
10 -  for i = 1:tam
11 -    Pi = route(i,:);
12 -    d = abs((A*Pi(1)+B*Pi(2)+C)/sqrt(A^2+B^2));
13 -    dist_vector(i) = d;
14 -  end
15 -  [dist_max,ind_max] = max(dist_vector);
16 -  destino = route(ind_max,:);
17 -
18 - end
```

ciclo_completo.m

```
1 - function ciclo_completo(pos_0,ori_0,T,n,N)
2 -   global nube
3 -   global global_counter
4 -   global secuencia
5 -   v_i=-1*ones(1,N);
6 -   v_j=-1*ones(1,N);
7 -   if n<=N
8 -
9 -     TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
10 -            sin(ori_0), cos(ori_0),pos_0(2);
11 -                 0 ,      0     ,    1   ];
```

```
12 -
13 -      for i=1:8
14 -        TB = TA_0*T{i};
15 -        for j=1:8
16 -          TA = TB*inv(T{j});
17 -          pos = TA(1:2,3)';
18 -          ori = atan2(TA(2,1),TA(1,1));
19 -          global_counter = global_counter + 1;
20 -          nube(global_counter,:) = [pos,ori];
21 -          if n==1
22 -            v_i(1)=i;
23 -            v_j(1)=j;
24 -            secuencia(global_counter,:) = [v_i v_j];
25 -          else
26 -            auxiliar=secuencia(global_counter-1,:);
27 -            auxiliar(n)=i;
28 -            auxiliar(N+n)=j;
29 -            secuencia(global_counter,:) = auxiliar;
30 -          end
31 -          ciclo_completo(pos,ori,T,n+1,N);
32 -        end
33 -      end
34 -    end
35 - end
```

encontrar_punto.m

```
 1 - function encontrar_punto(distancias)
 2 -   global nube
 3 -   global indice_min
 4 -   global secuencia_encontrada
 5 -   global P_mas_cercano
 6 -   global ori_mas_cercano
 7 -   global secuencia
 8 -   [~,indice_min] = min(distancias);
 9 -   P_mas_cercano = nube(indice_min,1:2);
10 -   ori_mas_cercano = nube(indice_min,3);
11 -   secuencia_encontrada = secuencia(indice_min,:);
12 - end
```

calculo_area.m

```
1 - function area = calculo_area(movB,pos_0,ori_0,T_p,area)
2 -   area_aux = [area(:,1) area(:,2) ones(length(area),1)];
3 -   area_aux = transpose(T_p*transpose(area_aux));
4 -   area = [area_aux(:,1) area_aux(:,2)];
5 - end
```

comprobacion_colisiones.m

```
 1 - function [colision,pos_F,ori_F] =
     comprobacion_colisiones(movB,pos_0,ori_0,indice,T_p,T,area)
 2 -   Retrieved = codigo_kdtree(pos_0,area);
 3 -   if length(Retrieved) == 0
 4 -     colision = 0;
 5 -   else
 6 -     Retrieved_L = T_p\[Retrieved(:,1) Retrieved(:,2)
                          ones(length(Retrieved(:,1)),1)]';
 7 -     x_obs_L = Retrieved_L(1,:);
 8 -     y_obs_L = Retrieved_L(2,:);
 9 -     for j = 1:length(x_obs_L)
10 -       if movB == 1
11 -         if x_obs_L(j)^2+y_obs_L(j)^2<=13891.29 && x_obs_L(j)>=-101.31 && ...
12 -           x_obs_L(j)<=101.31 && y_obs_L(j)>=-103.1332 && y_obs_L(j)<=103.1332
```

```
13 -            colision = 1;
14 -            break
15 -          else
16 -            colision = 0;
17 -          end
18 -        else
19 -          if 0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 &&
                 -0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 && ...
20 -             -0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 &&
                 0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 && ...
21 -             x_obs_L(j)<=75.668 && x_obs_L(j)>=-75.668 &&
                 y_obs_L(j)<=120.2 && y_obs_L(j)>=-120.2
22 -            colision = 1;
23 -            break
24 -          else
25 -            colision = 0;
26 -          end
27 -        end
28 -      end
29 -    end
30 -    if movB == 1
31 -      T_rel = T{indice};
32 -      T_new = T_p*T_rel;
33 -    else
34 -      T_rel = T{indice};
35 -      T_new = T_p/T_rel;
36 -    end
37 -    pos_F = [T_new(1,3) T_new(2,3)];
38 -    ori_F = atan2(T_new(2,1),T_new(2,2));
39 - end
```

codigo_kdtree.m

```
 1 - function Retrieved = codigo_kdtree(pos_0,area)
 2 -   global Md1
 3 -   global P_obs
 4 -   p1 = min(area(:,1));
 5 -   p2 = max(area(:,2));
 6 -   p3 = max(area(:,1));
 7 -   p4 = min(area(:,2));
 8 -   pt = [p1,p2,p3,p4];
 9 -   p = max(pt);
10 -   punto_cuadrado = [p,pos_0(1)];
11 -   centro = pos_0;
12 -   radio = norm(pos_0-punto_cuadrado);
13 -   Idx = rangesearch(Md1,centro,radio);
14 -   Retrieved = zeros(length(Idx{1}),2);
15 -   for i=1:length(Idx{1})
16 -     Retrieved(i,:) = P_obs(Idx{1}(i),:);
17 -   end
18 - end
```

ori_final.m

```
 1 - load resultado_ori_final;
 2 - v_indices = [];
 3 - v_dist = [];
 4 - T_encontrado = eye(3);
 5 - phi_dif = phi_ref-nube(indice_min,3);
 6 - if phi_dif > pi
 7 -   phi_dif = phi_dif-2*pi;
 8 - elseif phi_dif < -pi
 9 -   phi_dif = phi_dif+2*pi;
10 - end
```

```
11 - for x = 1:length(nube_fin(:,1))
12 -   if abs(nube_fin(x,3)-phi_dif) < 0.001
13 -     v_indices = [v_indices;x];
14 -   end
15 - end
16 - for y = 1:size(v_indices)
17 -   v_aux = nube_fin(v_indices(y),1:2);
18 -   theta = atan2(v_aux(2),v_aux(1));
19 -   phi_dif_aux = theta+nube(indice_min,3);
20 -   mod = norm(v_aux);
21 -   P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
                P_mas_cercano(2)+mod*sin(phi_dif_aux)];
22 -   dist = norm(P_aux-P_ref);
23 -   v_dist = [v_dist;dist];
24 - end
25 - [~,ind_tmp] = min(v_dist);
26 - secuencia_final = seq_fin(v_indices(ind_tmp),:);
27 - for k = 1:4
28 -   i = secuencia_final(k);
29 -   j = secuencia_final(k+4);
30 -   if i==-1 || j==-1
31 -     continue
32 -   end
33 -   T_encontrado = T_encontrado*T{i}/T{j};
34 - end
35 - T0 = [ cos(nube(indice_min,3)) -sin(nube(indice_min,3)) nube(indice_min,1);
36 -        sin(nube(indice_min,3))  cos(nube(indice_min,3)) nube(indice_min,2);
37 -                    0                        0                    1        ];
38 -
39 - T1 = T0*T_encontrado;
40 - P_final = T1(1:2,3)';
41 - phi_final = atan2(T1(2,1),T1(1,1));
```

dibujo.m

```
 1 - plot(nube_grande(:,1),nube_grande(:,2),'.b');
 2 - set(gca,'DataAspectRatio',[1,1,1]);
 3 - hold on
 4 - axis([xmin xmax ymin ymax]);
 5 - for d = 1:size(P_obs_p,1)
 6 -   P = P_obs_p(d,:);
 7 -   plot(P(1),P(2),'.k');
 8 - end
 9 - for d = 1:size(P_obs,1)
10 -   P = P_obs(d,:);
11 -   plot(P(1),P(2),'.r');
12 - end
13 - plot(route(:,1),route(:,2),'.r','MarkerSize',15)
14 - plot(posicion_inicial(1),posicion_inicial(2),'.g','MarkerSize',25)
15 - plot(P_ref(1),P_ref(2),'or');
16 - quiver(P_ref(1),P_ref(2),50*cos(phi_ref),50*sin(phi_ref),'r');
17 - plot(puntos_dest(:,1),puntos_dest(:,2),'*y')
18 - plot(P_mas_cercano(1),P_mas_cercano(2),'*m')
19 - quiver(P_mas_cercano(1),P_mas_cercano(2),50*cos(nube(indice_min,3)),
        50*sin(nube(indice_min,3)),'m');
20 - plot(P_final(1),P_final(2),'sg')
21 - quiver(P_final(1),P_final(2),50*cos(phi_final),50*sin(phi_final),'g')
```

ANEXO 4

ws_binario_mp.m

```
clear all;
global nube
global global_counter
global secuencia
global indice_min
global secuencia_encontrada
global P_mas_cercano
global ori_mas_cercano
global Md
global Md1
global P_obs
N = 2;
P_ref = [600 500];
phi_ref = pi/4;
P0 = [0,0];
posicion_inicial = P0;
phi0 = 0;
P_obs = [];
P_obs_p = [];
paso_x = 10;
paso_y = 10;
for xobs = 400-300:paso_x:400+300
  for yobs = 300-50:paso_y:300+50
    if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end
for xobs = 300-50:paso_x:300+50
  for yobs = 400-300:paso_y:400+300
    if (xobs-300)^2/50^2 + (yobs-400)^2/300^2 < 1
      P_obs_p = [P_obs_p;[xobs,yobs]];
    end
  end
end
% Comprobación de si el punto inicial o final se encuentran obstaculizados
for i = 1:length(P_obs_p(:,1))
  if norm(P0-P_obs_p(i,:)) < 120.65
    disp('Error: el punto de inicio se encuentra muy cerca de un obstaculo.')
    pausa1 = 1;
    break
  else
    pausa1 = 0;
  end
  if norm(P_ref-P_obs_p(i,:)) < 120.65
    disp('Error: el punto final se encuentra muy cerca de un obstaculo.')
    pausa2 = 1;
    break
  else
    pausa2 = 0;
  end
end
if pausa1 == 1 || pausa2 == 1
  pause
end
P_obs = P_obs_p;
Md = KDTreeSearcher(P_obs_p,'Distance','minkowski','P',inf,'BucketSize',1);
```

```
xmin = -300; xmax = 1200; ymin = -300; ymax = 1200;
[caminos_voronoi,esquema] = voronoi(paso_x,paso_y,xmin,xmax,ymin,ymax,
                            P0,P_ref);
P_obs = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax);
Md1 = KDTreeSearcher(P_obs,'Distance','minkowski','P',inf,'BucketSize',1);
xin=0;yin=0;xfin=600;yfin=500;
route = A_estrella(paso_x,paso_y,xin,yin,xfin,yfin,xmin,xmax,ymin,ymax,
        esquema);
tamanyo = 0;
for i=1:N
  tamanyo = tamanyo + 64^i;
end
nube = zeros(tamanyo,3);
global_counter = 0;
secuencia = (-1)*ones(tamanyo,2*N);
T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
registro_seq = [];
nube_grande = [];
puntos_encontrados = [];
ori_encontradas = [];
% Depende de: b, p, rho0, Delta_rho
phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
     21.95478428];
for i=1:8
  T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
end
[areaA,areaB] = area_local(P0,phi0);
puntos_dest = [P_ref;P0];
ind_dest = [1,length(route(:,1))];
umbral = 60;
while 1

  for i = 1:length(ind_dest)-1
    ind1 = ind_dest(i);
    ind2 = ind_dest(i+1);
    P1 = puntos_dest(i,:);
    P2 = puntos_dest(i+1,:);
    [destino,dist_max,ind_max] = extractor_destinos(route(ind1:ind2,:),P1,P2);
    if dist_max > umbral
      flag_1 = 0;
      break
    else
      flag_1 = 1;
    end
  end

  if flag_1 == 1
    break
  else
    tam_ind_dest = length(ind_dest);
    ind_act = ind_max + ind1;
    for j = 1:tam_ind_dest
      if ind_dest(j) > ind_act
        ind_dest = [ind_dest(1:j-1) ind_act ind_dest(j:tam_ind_dest)];
        puntos_dest = [puntos_dest(1:j-1,:); destino;
                       puntos_dest(j:tam_ind_dest,:)];
        break;
      end
    end
  end

end
```

```
117 - for i = length(ind_dest):-1:1
118 -   P_ref_actual = puntos_dest(i,:);
119 -   f_aux = 1;
120 -   while 1
121 -     if norm(P_ref_actual-P0) < 10
122 -       break
123 -     end
124 -     if f_aux == 1
125 -       global_counter = 0;
126 -       ciclo_completo(P0,phi0,T,1,N);
127 -       f_aux = 0;
128 -       distancias = zeros(size(nube,1),1);
129 -       for j=1:size(nube,1)
130 -         distancias(j) = norm(P_ref_actual-nube(j,1:2));
131 -       end
132 -     end
133 -     encontrar_punto(distancias)
134 -     pos_0 = P_mas_cercano;
135 -     ori_0 = ori_mas_cercano;
136 -     for h = 1:N
137 -       if secuencia_encontrada(h) == -1 || secuencia_encontrada(h+N) == -1
138 -         break
139 -       end
140 -       indice_inicio = secuencia_encontrada(h);
141 -       indice_fin = secuencia_encontrada(h+N);
142 -       movB = 1;
143 -       T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
144 -              sin(ori_0)  cos(ori_0) pos_0(2);
145 -                   0           0         1  ];
146 -       area = calculo_area(movB,pos_0,ori_0,T_p,areaB);
147 -       [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                     indice_inicio,T_p,T,area);
148 -       if colision == 1
149 -         break
150 -       end
151 -       movB = 0;
152 -       T_p = [cos(ori_0) -sin(ori_0) pos_0(1);
153 -              sin(ori_0)  cos(ori_0) pos_0(2);
154 -                   0           0         1  ];
155 -       area = calculo_area(movB,pos_0,ori_0,T_p,areaA);
156 -       [colision,pos_0,ori_0] = comprobacion_colisiones(movB,pos_0,ori_0,
                                     indice_fin,T_p,T,area);
157 -       if colision == 1
158 -         break
159 -       end
160 -     end
161 -     if colision == 0
162 -       registro_seq = [registro_seq; secuencia_encontrada];
163 -       puntos_encontrados = [puntos_encontrados; P_mas_cercano];
164 -       ori_encontradas = [ori_encontradas; ori_mas_cercano];
165 -       P0 = P_mas_cercano;
166 -       phi0 = ori_mas_cercano;
167 -       f_aux = 1;
168 -       nube_grande = [nube_grande;nube];
169 -     else
170 -       distancias(indice_min) = 1000000000;
171 -     end
172 -   end
173 -
174 - end
175 - ori_final;
176 - dibujo;
```

```
177 - save('resultados')
```

voronoi.m

```
 1 - function [caminos_voronoi,esquema] = voronoi(paso_x,paso_y,xmin,xmax,ymin,ymax,
                                                  P0,P_ref)
 2 -
 3 -   global Md
 4 -   global P_obs
 5 -
 6 -   Dx = xmax-xmin;
 7 -   Dy = ymax-ymin;
 8 -   g = round((Dx)/paso_x)+1;
 9 -   h = round((Dy)/paso_y)+1;
10 -   hay_obstaculo = zeros(g,h);
11 -   frontera = [];
12 -   caminos_voronoi = [];
13 -   esquema = [];
14 -   v_paso = [paso_x,paso_y];
15 -   radio = min(v_paso)/2;
16 -
17 -   r = (P_ref(1)-xmin)/paso_x+1;
18 -   s = (P_ref(2)-ymin)/paso_y+1;
19 -   hay_obstaculo(r,s) = 1;
20 -   r = (P0(1)-xmin)/paso_x+1;
21 -   s = (P0(2)-ymin)/paso_y+1;
22 -   hay_obstaculo(r,s) = 1;
23 -
24 -   for i=1:g
25 -     x = (i-1)*paso_x+xmin;
26 -     for j = 1:h
27 -       y = (j-1)*paso_y + ymin;
28 -       pto = [x y];
29 -       Idx = rangesearch(Md,pto,radio);
30 -       Retrieved = zeros(length(Idx{1}),2);
31 -       if isempty(Idx{1}) == 0
32 -         hay_obstaculo(i,j) = 1;
33 -       end
34 -     end
35 -   end
36 -   hay_obstaculo(1,:) = 1;
37 -   hay_obstaculo(g,:) = 1;
38 -   hay_obstaculo(:,1) = 1;
39 -   hay_obstaculo(:,h) = 1;
40 -
41 -   esq = ithin(1-hay_obstaculo);
42 -
43 -   for b = 1:g
44 -     for c = 1:h
45 -       q = [b c];
46 -       if esq(b,c) == 1
47 -         esquema = [esquema; q];
48 -         caminos_voronoi = [caminos_voronoi;
                               [(q(1)-1)*paso_x+xmin,(q(2)-1)*paso_y+ymin]];
49 -       end
50 -     end
51 -   end
52 -
53 -
54 - end
```

fr_2D_malla.m

```
function frontera = fr_2D_malla(paso_x,paso_y,xmin,xmax,ymin,ymax)
  global Md

  Dx = xmax-xmin;
  Dy = ymax-ymin;
  g = round((Dx)/paso_x)+1;
  h = round((Dy)/paso_y)+1;
  hay_obstaculo = zeros(g,h);
  frontera = [];
  v_paso = [paso_x,paso_y];
  radio = min(v_paso)/2;

  for i=1:g
    x = (i-1)*paso_x+xmin;
    for j = 1:h
      y = (j-1)*paso_y + ymin;
      pto = [x y];
      Idx = rangesearch(Md,pto,radio);
      Retrieved = zeros(length(Idx{1}),2);
      if isempty(Idx{1}) == 0
        hay_obstaculo(i,j) = 1;
      end
    end
  end

  for b = 1:g
    for c = 1:h

      columna = b;
      fila = c;
      if hay_obstaculo(columna,fila) == 1
        vecinos = zeros(8,2);
        z = 1;
        for i = -1:1
          for j = -1:1
            if (i~=0 || j~=0)
              vecinos(z,1) = columna+i;
              vecinos(z,2) = fila+j;
              z = z+1;
            end
          end
        end
        for k = 1:8
          e = vecinos(k,1);
          d = vecinos(k,2);
          if e>g || d>h || e<1 || d<1
            continue
          end
          if hay_obstaculo(e,d) == 1
            continue
          else
            p = [columna fila];
            frontera = [frontera; [(p(1)-1)*paso_x+xmin,(p(2)-1)*paso_y+ymin]];
          end
        end
      end
    end
  end

```

```
60 - end
```

A_estrella.m

```
 1 - function route =
     A_estrella(paso_x,paso_y,xin,yin,xfin,yfin,xmin,xmax,ymin,ymax,
                     esquema)
 2 -
 3 -   global P_obs
 4 -   global Md1
 5 -
 6 -   route = [];
 7 -
 8 -   Dx = xmax-xmin;
 9 -   Dy = ymax-ymin;
10 -
11 -   g = round((Dx)/paso_x)+1;
12 -   h = round((Dy)/paso_y)+1;
13 -
14 -   ClosedSet = zeros(g,h);
15 -   OpenSet = zeros(g,h);
16 -
17 -   xf = round((xfin-xmin)/paso_x)+1;
18 -   yf = round((yfin-ymin)/paso_y)+1;
19 -   l = round((xin-xmin)/paso_x)+1;
20 -   m = round((yin-ymin)/paso_y)+1;
21 -
22 -   esquema_v = [esquema;xf yf;l m];
23 -
24 -   OpenSet(l,m) = 1;
25 -   Came_From = cell(g,h);
26 -
27 -   for i = 1:g
28 -     for j = 1:h
29 -       Came_From{i,j} = [-1,-1];
30 -     end
31 -   end
32 -
33 -   g_score = ClosedSet;
34 -
35 -   for i = 1:g
36 -     for j = 1:h
37 -       g_score(i,j) = 100000000;
38 -     end
39 -   end
40 -
41 -   g_score(l,m) = 0;
42 -   f_score = 100000000*ones(g,h);
43 -   d = sqrt((xf-l)^2+(yf-m)^2);
44 -    _score(l,m) = d;
45 -
46 -   while (sum(sum(OpenSet))~=0)
47 -
48 -     min_value = 100000000;
49 -     for i = 1:g
50 -       for j = 1:h
51 -         if OpenSet(i,j) == 1
52 -           if f_score(i,j) < min_value
53 -             min_value = f_score(i,j);
54 -             fila = i;
55 -             columna = j;
56 -           end
57 -         end
```

```
  58 -       end
  59 -     end
  60 -
  61 -     if (fila == xf && columna == yf)
  62 -       [path] = reconstruct_path(Came_From,xf,yf,l,m);
  63 -       for i = 2:length(path)-1
  64 -         p = path{1,i};
  65 -         route = [route;(p(1)-1)*paso_x+xmin (p(2)-1)*paso_y+ymin];
  66 -       end
  67 -       return
  68 -     end
  69 -
  70 -     OpenSet(fila,columna) = 0;
  71 -     ClosedSet(fila,columna) = 1;
  72 -
  73 -     vecinos_p = zeros(8,2);
  74 -     z = 1;
  75 -     for i = -1:1
  76 -       for j = -1:1
  77 -         if (i~=0 || j~=0)
  78 -           vecinos_p(z,1) = fila+i;
  79 -           vecinos_p(z,2) = columna+j;
  80 -           z = z+1;
  81 -         end
  82 -       end
  83 -     end
  84 -
  85 -     vecinos = [];
  86 -     for i = 1:8
  87 -       v = vecinos_p(i,:);
  88 -       for j = 1:length(esquema_v)
  89 -         if esquema_v(j,:) == v
  90 -           vecinos = [vecinos;v];
  91 -           break
  92 -         end
  93 -       end
  94 -     end
  95 -
  96 -     X = (fila-1)*paso_x + xmin;
  97 -     Y = (columna-1)*paso_y + ymin;
  98 -
  99 -     radio = 120.65;
 100 -     centro = [X Y];
 101 -
 102 -     Idx = rangesearch(Md1,centro,radio);
 103 -     Retrieved = zeros(length(Idx{1}),2);
 104 -     for k = 1:length(Idx{1})
 105 -       Retrieved(k,:) = P_obs(Idx{1}(k),:);
 106 -     end
 107 -
 108 -     flag_invalido = 0;
 109 -     x_Retrieved = Retrieved(:,1);
 110 -     y_Retrieved = Retrieved(:,2);
 111 -     for w = 1:length(x_Retrieved)
 112 -       p_Retrieved = [x_Retrieved(w) y_Retrieved(w)];
 113 -       if norm(centro-p_Retrieved) <= radio
 114 -         f_score(fila,columna) = 10^10;
 115 -         flag_invalido = 1;
 116 -         break
 117 -       end
 118 -     end
 119 -
```

```
120 -     if flag_invalido == 0
121 -       for i = 1:length(vecinos(:,1))
122 -
123 -         a = vecinos(i,1);
124 -         b = vecinos(i,2);
125 -         if a<=0 || a>g || b<=0 || b>h
126 -           continue
127 -         end
128 -         if ClosedSet(a,b) == 1
129 -           continue
130 -         end
131 -
132 -         tentative_g_score = g_score(fila,columna)+
                                  sqrt((fila-a)^2+(columna-b)^2);
133 -
134 -         if OpenSet(a,b) == 0
135 -           OpenSet(a,b) = 1;
136 -         elseif tentative_g_score >= g_score(a,b)
137 -           continue
138 -         end
139 -         Came_From{a,b} = [fila,columna];
140 -         g_score(a,b) = tentative_g_score;
141 -         f_score(a,b) = g_score(a,b)+sqrt((a-xf)^2+(b-yf)^2);
142 -
143 -       end
144 -     end
145 -
146 -   end
147 -
148 - end
149 -
150 - function [path] = reconstruct_path(Came_From,xf,yf,l,m)
151 -
152 -   [gen,g] = size(Came_From);
153 -   gen = (gen*gen);
154 -   nodo = [xf,yf];
155 -   paso = cell(1,gen);
156 -
157 -   for i = 1:gen
158 -     if (nodo ~= [-1,-1])
159 -       paso{1,i} = nodo;
160 -       nodo = Came_From{nodo(1),nodo(2)};
161 -     end
162 -   end
163 -
164 -   cantidad = 1;
165 -   while (paso{1,cantidad}(1)~=l || paso{1,cantidad}(2)~=m)
166 -     cantidad = cantidad + 1;
167 -   end
168 -
169 -   path = cell(1,cantidad);
170 -   for i = 1:cantidad
171 -     path{1,i} = paso{1,i};
172 -   end
173 -
174 - end
```

area_local.m

```
  1 - function [areaA,areaB] = area_local(P0,phi0)
  2 -
  3 -   v = [1,2,3,4,5,6,7,8,1];
  4 -   BarrB = [];
```

```
  BarrA = [];
  for t = 1:8
    BarrB = calculo_area_adyac(v(t),v(t+1),1,P0,phi0,BarrB);
  end
  areaB = BarrB;
  v_aux = 101.31*ones(1,20);
  v_aux_p = -50.242:5.2886:50.242;
  v_t = transpose([v_aux; v_aux_p; ones(1,20)]);
  v_t_2 = transpose([-v_aux; v_aux_p; ones(1,20)]);
  areaB = [areaB(40:80,1) areaB(40:80,2);
           v_t(:,1) v_t(:,2);
           areaB(120:160,1) areaB(120:160,2);
           areaB(1:41,3)  areaB(1:41,4);
           v_t_2(:,1) v_t_2(:,2);
           areaB(80:120,3) areaB(80:120,4)];
  for u = 1:8
    BarrA = calculo_area_adyac(v(u),v(u+1),0,P0,phi0,BarrA);
  end
  areaA = BarrA;
  v_aux_2p = -9.295:3.718:9.295;
  v_aux_3p = 120.242*ones(1,6);
  v_aux_4p = 75.668*ones(1,9);
  v_aux_5p = -22.5656:5.6414:22.5656;
  v_t_p = transpose([v_aux_2p; v_aux_3p; ones(1,6)]);
  v_t_2p = transpose([v_aux_4p; v_aux_5p; ones(1,9)]);
  v_t_3p = transpose([v_aux_2p; -1*v_aux_3p; ones(1,6)]);
  v_t_4p = transpose([-1*v_aux_4p; v_aux_5p; ones(1,9)]);
  areaA = [areaA(52:80,1) areaA(52:80,2);
           v_t_p(:,1) v_t_p(:,2);
           areaA(80:110,3) areaA(80:110,4);
           v_t_2p(:,1) v_t_2p(:,2);
           areaA(1:30,7) areaA(1:30,8);
           v_t_3p(:,1) v_t_3p(:,2);
           areaA(131:160,5) areaA(131:160,6);
           v_t_4p(:,1) v_t_4p(:,2)];

end
```

calculo_area_adyac.m

```
function Barr = calculo_area_adyac(i,f,movB,pos_0,ori_0,Barr)
  m = 20;
  p = 101.31;
  b = 18.59;
  Barr_aux = Barr;
  phi_v = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
  y_v = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
         21.95478428];

  phi_inicial = phi_v(i);
  y_inicial = y_v(i);
  phi_final = phi_v(f);
  y_final = y_v(f);

  [li,ri] = inversa(phi_inicial,y_inicial);
  [lf,rf] = inversa(phi_final,y_final);
  dl = (lf-li)/(m-1);
  dr = (rf-ri)/(m-1);

  l = li;
  r = ri;
  phi = phi_inicial;
  y = y_inicial;
```

```
23 -
24 -   for i = 1:m
25 -
26 -     [phi,y] = directa(l,r,phi,y);
27 -     l = l+dl;
28 -     r = r+dr;
29 -
30 -     T_BA = [ cos(phi)  -sin(phi)    0;
31 -              sin(phi)   cos(phi)    y;
32 -                 0          0        1];
33 -     if movB == 1
34 -       T_A = [ cos(ori_0) -sin(ori_0)  pos_0(1);
35 -               sin(ori_0) cos(ori_0)   pos_0(2);
36 -                    0          0           1   ];
37 -       T_B = T_A*T_BA;
38 -       B1 = T_B*[p;0;1];
39 -       B2 = T_B*[-p;0;1];
40 -       Barr_aux = [Barr_aux; B1(1) B1(2) B2(1) B2(2)];
41 -     else
42 -       T_B = [cos(ori_0) -sin(ori_0)  pos_0(1);
43 -              sin(ori_0) cos(ori_0)   pos_0(2);
44 -                   0          0           1   ];
45 -       T_A = T_B*inv(T_BA);
46 -       altura_pasador = 70;
47 -       radio_pasador = b/2;
48 -       a11 = T_A*[-radio_pasador;altura_pasador;1];
49 -       a12 = T_A*[ radio_pasador;altura_pasador;1];
50 -       a21 = T_A*[-radio_pasador;-altura_pasador;1];
51 -       a22 = T_A*[ radio_pasador;-altura_pasador;1];
52 -       Barr_aux = [Barr_aux;
                  a11(1) a11(2) a12(1) a12(2) a21(1) a21(2) a22(1) a22(2)];
53 -     end
54 -   end
55 -
56 -   Barr = Barr_aux;
57 -
58 - end
```

inversa.m

```
1 - function [l,r] = inversa(phi,y)
2 -   b = 18.59;
3 -   p = 101.31;
4 -   l = sqrt((p*cos(phi)+b)^2+(y+p*sin(phi))^2);
5 -   r = sqrt((-p*cos(phi)-b)^2+(y-p*sin(phi))^2);
6 - end
```

directa.m

```
1 - function [phi,y] = directa(l,r,phi_p,y_p)
2 -   b = 18.59;
3 -   p = 101.31;
4 -   F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
5 -        (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
6 -   J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
7 -        -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
8 -   aux = [phi_p;y_p];
9 -
10 -  for iter = 1:10
11 -    aux = aux - J\F;
12 -    phi_p = aux(1);
13 -    y_p = aux(2);
14 -    J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
```

```
          -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
    F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2;
         (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
  end
  phi = aux(1);
  y = aux(2);

end
```

extractor_destinos.m

```
function [destino,dist_max,ind_max] = extractor_destinos(route,P1,P2)

  tam = length(route(:,1));
  dist_vector = zeros(tam,1);

  v = P1-P2;
  A = v(2)/v(1);
  B = -1;
  C = P1(2)-A*P1(1);
  for i = 1:tam
    Pi = route(i,:);
    d = abs((A*Pi(1)+B*Pi(2)+C)/sqrt(A^2+B^2));
    dist_vector(i) = d;
  end
  [dist_max,ind_max] = max(dist_vector);
  destino = route(ind_max,:);

end
```

ciclo_completo.m

```
function ciclo_completo(pos_0,ori_0,T,n,N)
  global nube
  global global_counter
  global secuencia
  v_i=-1*ones(1,N);
  v_j=-1*ones(1,N);
  if n<=N

    TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
            sin(ori_0), cos(ori_0),pos_0(2);
                 0 ,      0     ,    1   ];

    for i=1:8
      TB = TA_0*T{i};
      for j=1:8
        TA = TB*inv(T{j});
        pos = TA(1:2,3)';
        ori = atan2(TA(2,1),TA(1,1));
        global_counter = global_counter + 1;
        nube(global_counter,:) = [pos,ori];
        if n==1
          v_i(1)=i;
          v_j(1)=j;
          secuencia(global_counter,:) = [v_i v_j];
        else
          auxiliar=secuencia(global_counter-1,:);
          auxiliar(n)=i;
          auxiliar(N+n)=j;
          secuencia(global_counter,:) = auxiliar;
        end
        ciclo_completo(pos,ori,T,n+1,N);
```

```
32 -        end
33 -      end
34 -    end
35 - end
```

encontrar_punto.m

```
 1 - function encontrar_punto(distancias)
 2 -   global nube
 3 -   global indice_min
 4 -   global secuencia_encontrada
 5 -   global P_mas_cercano
 6 -   global ori_mas_cercano
 7 -   global secuencia
 8 -   [~,indice_min] = min(distancias);
 9 -   P_mas_cercano = nube(indice_min,1:2);
10 -   ori_mas_cercano = nube(indice_min,3);
11 -   secuencia_encontrada = secuencia(indice_min,:);
12 - end
```

calculo_area.m

```
1 - function area = calculo_area(movB,pos_0,ori_0,T_p,area)
2 -   area_aux = [area(:,1) area(:,2) ones(length(area),1)];
3 -   area_aux = transpose(T_p*transpose(area_aux));
4 -   area = [area_aux(:,1) area_aux(:,2)];
5 - end
```

comprobacion_colisiones.m

```
 1 - function [colision,pos_F,ori_F] = comprobacion_colisiones(movB,pos_0,ori_0,
                                       indice,T_p,T,area)
 2 -   Retrieved = codigo_kdtree(pos_0,area);
 3 -   if length(Retrieved) == 0
 4 -     colision = 0;
 5 -   else
 6 -     Retrieved_L = T_p\[Retrieved(:,1) Retrieved(:,2)
                          ones(length(Retrieved(:,1)),1)]';
 7 -     x_obs_L = Retrieved_L(1,:);
 8 -     y_obs_L = Retrieved_L(2,:);
 9 -     for j = 1:length(x_obs_L)
10 -       if movB == 1
11 -         if x_obs_L(j)^2+y_obs_L(j)^2<=13891.29 && x_obs_L(j)>=-101.31 && ...
12 -           x_obs_L(j)<=101.31 && y_obs_L(j)>=-103.1332 && y_obs_L(j)<=103.1332
13 -           colision = 1;
14 -           break
15 -         else
16 -           colision = 0;
17 -         end
18 -       else
19 -         if 0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 &&
                -0.9913*x_obs_L(j)+129.4137-y_obs_L(j)>=0 && ...
20 -            -0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 &&
                0.9913*x_obs_L(j)-129.4137-y_obs_L(j)<=0 && ...
21 -            x_obs_L(j)<=75.668 && x_obs_L(j)>=-75.668 &&
                y_obs_L(j)<=120.2 && y_obs_L(j)>=-120.2
22 -           colision = 1;
23 -           break
24 -         else
25 -           colision = 0;
26 -         end
27 -       end
28 -     end
29 -   end
30 -   if movB == 1
31 -     T_rel = T{indice};
```

```
32 -      T_new = T_p*T_rel;
33 -    else
34 -      T_rel = T{indice};
35 -      T_new = T_p/T_rel;
36 -    end
37 -    pos_F = [T_new(1,3) T_new(2,3)];
38 -    ori_F = atan2(T_new(2,1),T_new(2,2));
39 - end
```

codigo_kdtree.m

```
 1 - function Retrieved = codigo_kdtree(pos_0,area)
 2 -   global Md1
 3 -   global P_obs
 4 -   p1 = min(area(:,1));
 5 -   p2 = max(area(:,2));
 6 -   p3 = max(area(:,1));
 7 -   p4 = min(area(:,2));
 8 -   pt = [p1,p2,p3,p4];
 9 -   p = max(pt);
10 -   punto_cuadrado = [p,pos_0(1)];
11 -   centro = pos_0;
12 -   radio = norm(pos_0-punto_cuadrado);
13 -   Idx = rangesearch(Md1,centro,radio);
14 -   Retrieved = zeros(length(Idx{1}),2);
15 -   for i=1:length(Idx{1})
16 -     Retrieved(i,:) = P_obs(Idx{1}(i),:);
17 -   end
18 - end
```

ori_final.m

```
 1 - load resultado_ori_final;
 2 - v_indices = [];
 3 - v_dist = [];
 4 - T_encontrado = eye(3);
 5 - phi_dif = phi_ref-nube(indice_min,3);
 6 - if phi_dif > pi
 7 -   phi_dif = phi_dif-2*pi;
 8 - elseif phi_dif < -pi
 9 -   phi_dif = phi_dif+2*pi;
10 - end
11 - for x = 1:length(nube_fin(:,1))
12 -   if abs(nube_fin(x,3)-phi_dif) < 0.001
13 -     v_indices = [v_indices;x];
14 -   end
15 - end
16 - for y = 1:size(v_indices)
17 -   v_aux = nube_fin(v_indices(y),1:2);
18 -   theta = atan2(v_aux(2),v_aux(1));
19 -   phi_dif_aux = theta+nube(indice_min,3);
20 -   mod = norm(v_aux);
21 -   P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
                P_mas_cercano(2)+mod*sin(phi_dif_aux)];
22 -   dist = norm(P_aux-P_ref);
23 -   v_dist = [v_dist;dist];
24 - end
25 - [~,ind_tmp] = min(v_dist);
26 - secuencia_final = seq_fin(v_indices(ind_tmp),:);
27 - for k = 1:4
28 -   i = secuencia_final(k);
29 -   j = secuencia_final(k+4);
30 -   if i==-1 || j==-1
```

```
31 -     continue
32 -   end
33 -   T_encontrado = T_encontrado*T{i}/T{j};
34 - end
35 - T0 = [ cos(nube(indice_min,3)) -sin(nube(indice_min,3)) nube(indice_min,1);
36 -        sin(nube(indice_min,3))  cos(nube(indice_min,3)) nube(indice_min,2);
37 -                    0                          0                    1        ];
38 -
39 - T1 = T0*T_encontrado;
40 - P_final = T1(1:2,3)';
41 - phi_final = atan2(T1(2,1),T1(1,1));
```

dibujo.m

```
 1 - plot(caminos_voronoi(:,1),caminos_voronoi(:,2),'ok')
 2 - set(gca,'DataAspectRatio',[1,1,1]);
 3 - hold on
 4 - axis([xmin xmax ymin ymax]);
 5 -
 6 - for d = 1:size(P_obs_p,1)
 7 -   P = P_obs_p(d,:);
 8 -   plot(P(1),P(2),'.k');
 9 - end
10 - for d = 1:size(P_obs,1)
11 -   P = P_obs(d,:);
12 -   plot(P(1),P(2),'.r');
13 - end
14 -
15 - plot(nube_grande(:,1),nube_grande(:,2),'.b');
16 - plot(route(:,1),route(:,2),'.r','MarkerSize',15)
17 - plot(posicion_inicial(1),posicion_inicial(2),'.g','MarkerSize',25)
18 - plot(puntos_dest(:,1),puntos_dest(:,2),'*y')
19 - plot(P_ref(1),P_ref(2),'or');
20 - quiver(P_ref(1),P_ref(2),50*cos(phi_ref),50*sin(phi_ref),'r');
21 - plot(P_mas_cercano(1),P_mas_cercano(2),'*m');
22 - quiver(P_mas_cercano(1),P_mas_cercano(2), 50*cos(nube(indice_min,3)),
          50*sin(nube(indice_min,3)),'m');
23 - plot(P_final(1),P_final(2),'sg');
24 - quiver(P_final(1),P_final(2),50*cos(phi_final),50*sin(phi_final),'g');
```

ANEXO 5

(1) ws_alg_gen.m

```
clear all

N = 3;
T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-
21.95478428,0,21.95478428];

for i=1:8
  T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
end

P_ref = [50 120];
phi_ref = pi/4;
n_poblacion = 500/64^(3-N);
pobl = round(rand(n_poblacion,6*N));

for i = 1:1000
  [child,i_child,flag] = child_generation(pobl,T,N,P_ref,phi_ref,n_poblacion);
  if flag == 1
    break
  end
  pobl(i_child,:) = child;
end

pobl_enteros = zeros(n_poblacion,2*N);
contador = 0;
for i = 1:3:6*N
  contador = contador+1;
  pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
end
pobl_enteros = pobl_enteros+1;

distancias = zeros(n_poblacion,1);
for l = 1:n_poblacion
  secuencia_encontrada = pobl_enteros(l,:);
  T_encontrado = eye(3);
  for k=1:N
    i = secuencia_encontrada(k);
    j = secuencia_encontrada(k+N);
    if i==-1 || j==-1
      continue
    end
    T_encontrado = T_encontrado*T{i}/T{j};
  end
  v_entrada = [P_ref phi_ref];
  P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
  phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
  v_encontrada = [P_encontrada phi_encontrada];
  ang_dif = abs(v_entrada(3)-v_encontrada(3));
  if 2*pi-ang_dif < ang_dif
    ang_dif = 2*pi-ang_dif;
  end
  v_norm = [v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2)
            -150/(-1.8379+log(ang_dif))];
  distancias(l) = norm(v_norm);
end

```

```
57 - [~,ind_minimo] = min(distancias);
58 -
59 - sol_final = pobl(ind_minimo,:);
60 - secuencia_encontrada = zeros(1,2*N);
61 - contador = 0;
62 - for i = 1:3:6*N
63 -   contador = contador+1;
64 -   n = bi2de(sol_final(i:i+2));
65 -   secuencia_encontrada(contador) = n;
66 - end
67 - secuencia_encontrada = secuencia_encontrada+1;
68 - T_encontrado = eye(3);
69 - for k=1:N
70 -   i = secuencia_encontrada(k);
71 -   j = secuencia_encontrada(k+N);
72 -   if i==-1 || j==-1
73 -     continue
74 -   end
75 -   T_encontrado = T_encontrado*T{i}/T{j};
76 - end
77 -
78 - P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)]
79 - phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1))
```

(1) child_generation.m

```
 1 - function [child,i_child,flag] = child_generation(pobl,T,N,P_ref,phi_ref,
                                     n_poblacion)
 2 -
 3 -   pobl_enteros = zeros(n_poblacion,2*N);
 4 -   contador = 0;
 5 -   for i = 1:3:6*N
 6 -     contador = contador+1;
 7 -     pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
 8 -   end
 9 -   pobl_enteros = pobl_enteros+1;
10 -
11 -   distancias = zeros(n_poblacion,1);
12 -   for l = 1:n_poblacion
13 -     secuencia_encontrada = pobl_enteros(l,:);
14 -     T_encontrado = eye(3);
15 -     for k=1:N
16 -       i = secuencia_encontrada(k);
17 -       j = secuencia_encontrada(k+N);
18 -       if i==-1 || j==-1
19 -          continue
20 -       end
21 -       T_encontrado = T_encontrado*T{i}/T{j};
22 -     end
23 -     v_entrada = [P_ref phi_ref];
24 -     P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
25 -     phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
26 -     v_encontrada = [P_encontrada phi_encontrada];
27 -     ang_dif = abs(v_entrada(3)-v_encontrada(3));
28 -     if 2*pi-ang_dif < ang_dif
29 -       ang_dif = 2*pi-ang_dif;
30 -     end
31 -     v_norm = [v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2)
                  -150/(-1.8379+log(ang_dif))];
32 -     distancias(l) = norm(v_norm);
33 -   end
34 -
35 -   % Se reproduce el 25% de la población
```

```
  tasa_prom = 0.25;
  n_reproductores = round(n_poblacion*tasa_prom);
  n_reproductores = floor(n_reproductores/2)*2;
  if min(distancias) < 10
    flag = 1;
    child = zeros(n_reproductores,6*N);
    i_child = 1;
  else
    flag = 0;
    d_max = max(distancias)+10;
    v_pond = round((d_max-distancias)/10);
    v_pond_acumulado = cumsum(v_pond);
    ganadores = randi(v_pond_acumulado(end)-1,[n_reproductores,1]);
    indices_reproductores = zeros(n_reproductores,1);
    for i = 1:n_reproductores
      n = ganadores(i);
      v_pond_acumulado_aux = abs(v_pond_acumulado-n);
      [~,indice] = min(v_pond_acumulado_aux);
      if v_pond_acumulado(indice) >= n
        indices_reproductores(i) = indice;
      else
        indices_reproductores(i) = indice+1;
      end
    end

    child = zeros(n_reproductores,6*N);
    i_child = indices_reproductores;
    for i = 1:n_reproductores/2
      indice_parent1 = indices_reproductores(i);
      parent1 = pobl(indice_parent1,:);
      indice_parent2 = indices_reproductores(i+n_reproductores/2);
      parent2 = pobl(indice_parent2,:);
      [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion);
      child(i,:) = child1;
      child(i+n_reproductores/2,:) = child2;
    end
  end

end
```

(1) reproduccion.m

```
function [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion)

%   child1 = [parent1(1:3*N) parent2(3*N+1:6*N)];
%   child2 = [parent2(1:3*N) parent1(3*N+1:6*N)];

  child1 = zeros(1,6*N);
  child2 = zeros(1,6*N);
  for i = 1:6:6*N
    child1(i:i+5) = [parent1(i:i+2) parent2(i+3:i+5)];
    child2(i:i+5) = [parent2(i:i+2) parent1(i+3:i+5)];
  end

%   Tasa de mutación 1/(landa^0.9318*^L^0.4535)
  tasa = 1/(n_poblacion^0.9318*(6*N)^0.4535);
  if rand >= 1-tasa
    mutacion = randi(6*N);
    child1(mutacion) = not(child1(mutacion));
  end
  if rand >= 1-tasa
    mutacion = randi(6*N);
```

```
    child2(mutacion) = not(child2(mutacion));
  end

end
```

(2) ws_alg_gen.m

```
clear all

N = 3;
T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
     21.95478428];

for i=1:8
   T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
end

% Datos de entrada
P_ref = [50 120];
phi_ref = pi/4;
n_poblacion = 500/64^(3-N);

% Generación de la población
pobl = round(rand(n_poblacion,6*N));

for i = 1:1000
    [child,i_child,flag] = child_generation(pobl,T,N,P_ref,phi_ref,n_poblacion);
    if flag == 1
        break
    end
    pobl(i_child,:) = son;
end

pobl_enteros = zeros(n_poblacion,2*N);
contador = 0;
for i = 1:3:6*N
    contador = contador+1;
    pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
end
pobl_enteros = pobl_enteros+1;

distancias = zeros(n_poblacion,1);
for l = 1:n_poblacion
    secuencia_encontrada = pobl_enteros(l,:);
    T_encontrado = eye(3);
    for k=1:N
        i = secuencia_encontrada(k);
        j = secuencia_encontrada(k+N);
        if i==-1 || j==-1
            continue
        end
        T_encontrado = T_encontrado*T{i}/T{j};
    end
    v_entrada = [P_ref phi_ref];
    P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
    phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
    v_encontrada = [P_encontrada phi_encontrada];
    ang_dif = abs(v_entrada(3)-v_encontrada(3));
    if 2*pi-ang_dif < ang_dif
```

```
54 -        ang_dif = 2*pi-ang_dif;
55 -    end
56 -    v_norm = [v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2)
                  -150/(-1.8379+log(ang_dif))];
57 -    distancias(l) = norm(v_norm);
58 - end
59 -
60 - [~,ind_minimo] = min(distancias);
61 -
62 - sol_final = pobl(ind_minimo,:);
63 - secuencia_encontrada = zeros(1,2*N);
64 - contador = 0;
65 - for i = 1:3:6*N
66 -     contador = contador+1;
67 -     n = bi2de(sol_final(i:i+2));
68 -     secuencia_encontrada(contador) = n;
69 - end
70 - secuencia_encontrada = secuencia_encontrada+1;
71 - T_encontrado = eye(3);
72 - for k=1:N
73 -     i = secuencia_encontrada(k);
74 -     j = secuencia_encontrada(k+N);
75 -     if i==-1 || j==-1
76 -         continue
77 -     end
78 -     T_encontrado = T_encontrado*T{i}/T{j};
79 - end
80 -
81 - P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)]
82 - phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1))
```

(2) child_generation.m

```
1 - function [child,i_child,flag] = child_generation(pobl,T,N,P_ref,phi_ref,
                                    n_poblacion)
2 -
3 -   pobl_enteros = zeros(n_poblacion,2*N);
4 -   contador = 0;
5 -   for i = 1:3:6*N
6 -     contador = contador+1;
7 -     pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
8 -   end
9 -   pobl_enteros = pobl_enteros+1;
10 -
11 -  distancias = zeros(n_poblacion,1);
12 -  for l = 1:n_poblacion
13 -    secuencia_encontrada = pobl_enteros(l,:);
14 -    T_encontrado = eye(3);
15 -    for k=1:N
16 -      i = secuencia_encontrada(k);
17 -      j = secuencia_encontrada(k+N);
18 -      if i==-1 || j==-1
19 -        continue
20 -      end
21 -      T_encontrado = T_encontrado*T{i}/T{j};
22 -    end
23 -    v_entrada = [P_ref phi_ref];
24 -    P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
25 -    phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
26 -    v_encontrada = [P_encontrada phi_encontrada];
27 -    ang_dif = abs(v_entrada(3)-v_encontrada(3));
28 -    if 2*pi-ang_dif < ang_dif
29 -      ang_dif = 2*pi-ang_dif;
```

```
30 -      end
31 -      v_norm = [v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2)
                    -150/(-1.8379+log(ang_dif))];
32 -      distancias(l) = norm(v_norm);
33 -    end
34 -
35 -    dist_norm = distancias/(1.1*max(distancias));
36 -    counter = 0;
37 -    indices_reproductores = zeros(n_poblacion,1);
38 -    for i = 1:n_poblacion
39 -      if dist_norm(i) > rand
40 -        counter = counter+1;
41 -        indices_reproductores(counter) = i;
42 -      end
43 -    end
44 -    n_reproductores = floor(counter/2)*2;
45 -    indices_reproductores = indices_reproductores(1:n_reproductores,:);
46 -
47 -    if min(distancias) < 10
48 -      flag = 1;
49 -      child = zeros(n_reproductores,6*N);
50 -      i_child = 1;
51 -    else
52 -      child = zeros(n_reproductores,6*N);
53 -      i_child = indices_reproductores;
54 -      for i = 1:n_reproductores/2
55 -        indice_parent1 = indices_reproductores(i);
56 -        indice_parent2 = indices_reproductores(i+n_reproductores/2);
57 -        parent1 = pobl(indice_parent1,:);
58 -        parent2 = pobl(indice_parent2,:);
59 -        [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion);
60 -        child(i,:) = child1;
61 -        child(i+n_reproductores/2,:) = child2;
62 -      end
63 -    end
64 -
65 - end
```

(2) reproduccion.m

```
 1 - function [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion)
 2 -
 3 -   child1 = [parent1(1:3*N) parent2(3*N+1:6*N)];
 4 -   child2 = [parent2(1:3*N) parent1(3*N+1:6*N)];
 5 -
 6 - %   child1 = zeros(1,6*N);
 7 - %   child2 = zeros(1,6*N);
 8 - %   for i = 1:6:6*N
 9 - %     child1(i:i+5) = [parent1(i:i+2) parent2(i+3:i+5)];
10 - %     child2(i:i+5) = [parent2(i:i+2) parent1(i+3:i+5)];
11 - %   end
12 -
13 - %   Tasa de mutación 1/(landa^0.9318*^L^0.4535)
14 -   tasa = 1/(n_poblacion^0.9318*(6*N)^0.4535);
15 -   if rand >= 1-tasa
16 -     mutacion = randi(6*N);
17 -     child1(mutacion) = not(child1(mutacion));
18 -   end
19 -   if rand >= 1-tasa
20 -     mutacion = randi(6*N);
21 -     child2(mutacion) = not(child2(mutacion));
22 -   end
23 -
```

```
24 - end
```

(3) ws_alg_gen.m

```
 1 - clear all
 2 -
 3 - N = 4;
 4 - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
 5 - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
 6 - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
       21.95478428];
 7 -
 8 - for i=1:8
 9 -   T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
10 - end
11 -
12 - % Datos de entrada
13 - P_ref = [150 225];
14 - phi_ref = pi/4;
15 - n_poblacion = 500/64^(3-N);
16 -
17 - % Generación de la población
18 - pobl = round(rand(n_poblacion,6*N));
19 - distancias = 10000000;
20 - for a = 1:1000
21 -
22 -   if min(distancias) < 10
23 -     break
24 -   end
25 -   child = child_generation(pobl,N,n_poblacion);
26 -   pobl = [pobl;child];
27 -   pobl_enteros = zeros(size(pobl,1),2*N);
28 -   contador = 0;
29 -   for i = 1:3:6*N
30 -     contador = contador+1;
31 -     pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
32 -   end
33 -   pobl_enteros = pobl_enteros+1;
34 -
35 -   distancias = zeros(n_poblacion,1);
36 -   for l = 1:size(pobl_enteros,1)
37 -     secuencia_encontrada = pobl_enteros(l,:);
38 -     T_encontrado = eye(3);
39 -     for k=1:N
40 -       i = secuencia_encontrada(k);
41 -       j = secuencia_encontrada(k+N);
42 -       if i==-1 || j==-1
43 -         continue
44 -       end
45 -       T_encontrado = T_encontrado*T{i}/T{j};
46 -     end
47 -     v_entrada = [P_ref phi_ref];
48 -     P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
49 -     phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
50 -     v_encontrada = [P_encontrada phi_encontrada];
51 -     ang_dif = abs(v_entrada(3)-v_encontrada(3));
52 -     if 2*pi-ang_dif < ang_dif
53 -       ang_dif = 2*pi-ang_dif;
54 -     end
55 -     v_norm = [v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2)
                 -150/(-1.8379+log(ang_dif))];
```

```
 56 -     distancias(l) = norm(v_norm);
 57 -   end
 58 -
 59 -   indices_max = zeros(size(pobl,1)-n_poblacion,1);
 60 -   for j = 1:length(indices_max)
 61 -     [~,imax] = max(distancias);
 62 -     distancias(imax) = 0;
 63 -     indices_max(j) = imax;
 64 -   end
 65 -   pobl(indices_max,:) = [];
 66 -   distancias(índices_max) = [];
 67 -
 68 - end
 69 -
 70 - pobl_enteros = zeros(n_poblacion,2*N);
 71 - contador = 0;
 72 - for i = 1:3:6*N
 73 -   contador = contador+1;
 74 -   pobl_enteros(:,contador) = bi2de(pobl(:,i:i+2));
 75 - end
 76 - pobl_enteros = pobl_enteros+1;
 77 -
 78 - distancias = zeros(n_poblacion,1);
 79 - for l = 1:n_poblacion
 80 -   secuencia_encontrada = pobl_enteros(l,:);
 81 -   T_encontrado = eye(3);
 82 -   for k=1:N
 83 -     i = secuencia_encontrada(k);
 84 -     j = secuencia_encontrada(k+N);
 85 -     if i==-1 || j==-1
 86 -         continue
 87 -     end
 88 -     T_encontrado = T_encontrado*T{i}/T{j};
 89 -   end
 90 -   v_entrada = [P_ref phi_ref];
 91 -   P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)];
 92 -   phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
 93 -   v_encontrada = [P_encontrada phi_encontrada];
 94 -   ang_dif = abs(v_entrada(3)-v_encontrada(3));
 95 -   if 2*pi-ang_dif < ang_dif
 96 -     ang_dif = 2*pi-ang_dif;
 97 -   end
 98 -   v_norm = [v_entrada(1)-v_encontrada(1) v_entrada(2)-v_encontrada(2)
                -150/(-1.8379+log(ang_dif))];
 99 -   distancias(l) = norm(v_norm);
100 - end
101 -
102 - [~,ind_minimo] = min(distancias);
103 -
104 - sol_final = pobl(ind_minimo,:);
105 - secuencia_encontrada = zeros(1,2*N);
106 - contador = 0;
107 - for i = 1:3:6*N
108 -   contador = contador+1;
109 -   n = bi2de(sol_final(i:i+2));
110 -   secuencia_encontrada(contador) = n;
111 - end
112 - secuencia_encontrada = secuencia_encontrada+1;
113 - T_encontrado = eye(3);
114 - for k=1:N
115 -   i = secuencia_encontrada(k);
116 -   j = secuencia_encontrada(k+N);
```

```
117 -   if i==-1 || j==-1
118 -     continue
119 -   end
120 -   T_encontrado = T_encontrado*T{i}/T{j};
121 - end
122 -
123 - P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)]
124 - phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1))
```

(3) child_generation.m

```
  1 - function child = child_generation(pobl,N,n_poblacion)
  2 -
  3 -   % Se reproduce el 25% de la población
  4 -   tasa_prom = 0.25;
  5 -   n_reproductores = round(n_poblacion*tasa_prom);
  6 -   n_reproductores = floor(n_reproductores/2)*2;
  7 -   indices_reproductores = randi(n_poblacion,[n_reproductores,1]);
  8 -
  9 -   child = zeros(n_reproductores,6*N);
 10 -   for i = 1:n_reproductores/2
 11 -     indice_parent1 = indices_reproductores(i);
 12 -     parent1 = pobl(indice_parent1,:);
 13 -     indice_parent2 = indices_reproductores(i+n_reproductores/2);
 14 -     parent2 = pobl(indice_parent2,:);
 15 -     [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion);
 16 -     child(i,:) = child1;
 17 -     child(i+n_reproductores/2,:) = child2;
 18 -   end
 19 -
 20 - end
```

(3) reproduccion.m

```
  1 - function [child1,child2] = reproduccion(parent1,parent2,N,n_poblacion)
  2 -
  3 -   child1 = [parent1(1:3*N) parent2(3*N+1:6*N)];
  4 -   child2 = [parent2(1:3*N) parent1(3*N+1:6*N)];
  5 -
  6 - %   child1 = zeros(1,6*N);
  7 - %   child2 = zeros(1,6*N);
  8 - %   for i = 1:6:6*N
  9 - %     child1(i:i+5) = [parent1(i:i+2) parent2(i+3:i+5)];
 10 - %     child2(i:i+5) = [parent2(i:i+2) parent1(i+3:i+5)];
 11 - %   end
 12 -
 13 - %   Tasa de mutación 1/(landa^0.9318*^L^0.4535)
 14 -   tasa = 1/(n_poblacion^0.9318*(6*N)^0.4535);
 15 -   if rand >= 1-tasa
 16 -     mutacion = randi(6*N);
 17 -     child1(mutacion) = not(child1(mutacion));
 18 -   end
 19 -   if rand >= 1-tasa
 20 -     mutacion = randi(6*N);
 21 -     child2(mutacion) = not(child2(mutacion));
 22 -   end
 23 -
 24 - end
```

ANEXO 6

ws_alg_hr.m

```
clear all

N = 7;
phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,0,
     21.95478428];
T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};

bee = [400,400];

for i=1:8
  T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
end

deltaX0 = 20;
deltaY0 = 20;
xmin = -deltaX0/2;
xmax = deltaX0/2;
ymin = -deltaY0/2;
ymax = deltaY0/2;
D = 1;

cell_mallas = {D};
cell_limites = {[xmax,xmin,ymax,ymin]};
cell_centros = {[1 1]};

for i = 1:N-1
  [D,i0_f,j0_f,xmax,xmin,ymax,ymin] = calculo_malla_espacio_trabajo(i,T);
  cell_mallas{i+1} = D;
  cell_limites{i+1} = [xmax,xmin,ymax,ymin];
  cell_centros{i+1} = [i0_f j0_f];
end

T_ant = eye(3);
secuencia = zeros(1,2*N);

for k = 1:N
  densidad = zeros(8,8);
  for i = 1:8
    for j = 1:8
      Tt = T{i}/T{j};
      TL = T_ant*Tt;
      bL = TL(1:2,3);
      bu = TL(1:2,1:2)'*(bee'-bL);
      ij_0 = cell_centros{N-(k-1)};
      xy_maxmin = cell_limites{N-(k-1)};
      D = cell_mallas{N-(k-1)};
      deltaX = (xy_maxmin(1)-xy_maxmin(2))/size(D,2);
      deltaY = (xy_maxmin(3)-xy_maxmin(4))/size(D,1);
      i_b = floor((bu(1))/deltaX+0.5)+ij_0(1);
      j_b = floor((bu(2))/deltaY+0.5)+ij_0(2);
      if i_b>=1 && i_b<=size(D,2) && j_b>=1 && j_b<=size(D,1)
        densidad(i,j) = D(j_b,i_b);
      else
        densidad(i,j) = 0;
      end
    end
```

```
57 -    end
58 -    maximo = max(max(densidad));
59 -    [f_max,c_max] = find(densidad==maximo);
60 -    if length(f_max) == 1
61 -      imax = f_max;
62 -      jmax = c_max;
63 -    else
64 -      norma = 100000000000;
65 -      for i = 1:length(f_max)
66 -        T_aux = T_ant*T{f_max(i)}/T{c_max(i)};
67 -        pos_aux = T_aux(1:2,3);
68 -        if norm(bee'-pos_aux) < norma
69 -          norma = norm(bee'-pos_aux);
70 -          imax = f_max(i);
71 -          jmax = c_max(i);
72 -        end
73 -      end
74 -    end
75 -    T_ant = T_ant*T{imax}/T{jmax};
76 -    secuencia(k) = imax;
77 -    secuencia(k+N) = jmax;
78 - end
```

calculo_malla_espacio_trabajo.m

```
1 - function [D,i0_f,j0_f,xmax,xmin,ymax,ymin] = calculo_malla_espacio_trabajo(N,T)
2 -
3 -    deltaX0 = 20;
4 -    deltaY0 = 20;
5 -    xmin = -deltaX0/2;
6 -    xmax = deltaX0/2;
7 -    ymin = -deltaY0/2;
8 -    ymax = deltaY0/2;
9 -    D = 1;
10 -
11 -   for i = 1:N
12 -     [D,xmax,xmin,ymax,ymin,i0_f,j0_f] = convolucion(D,xmax,xmin,ymax,ymin,T);
13 -   end
14 -
15 - end
```

convolucion.m

```
1 - function [D_s,xmax_s,xmin_s,ymax_s,ymin_s,i0_s,j0_s] = convolucion(D,xmax,xmin,
                                                            ymax,ymin,T)
2 -
3 -    [f,c] = size(D);
4 -    deltaX = (xmax-xmin)/c;
5 -    deltaY = (ymax-ymin)/f;
6 -
7 -    esquina1 = [xmin,ymin];
8 -    esquina2 = [xmax,ymin];
9 -    esquina3 = [xmin,ymax];
10 -   esquina4 = [xmax,ymax];
11 -   esquinas = [esquina1;esquina2;esquina3;esquina4];
12 -
13 -   nube = zeros(4*64,2);
14 -   global_counter = 1;
15 -   for k = 1:4
16 -     esquina = esquinas(k,:);
17 -     for i=1:8
18 -       for j=1:8
19 -         TM = T{i}/T{j};
```

```
20 -          v_p = TM*[esquina(1);esquina(2);1];
21 -          global_counter = global_counter + 1;
22 -          nube(global_counter,:) = [v_p(1),v_p(2)];
23 -        end
24 -      end
25 -    end
26 -
27 -    xmax_s = max(nube(:,1));
28 -    xmin_s = min(nube(:,1));
29 -    ymax_s = max(nube(:,2));
30 -    ymin_s = min(nube(:,2));
31 -
32 -    nx = ceil((xmax_s-xmin_s)/deltaX);
33 -    ny = ceil((ymax_s-ymin_s)/deltaY);
34 -    if floor(nx/2) == nx/2
35 -      nx = nx+1;
36 -    end
37 -    if floor(ny/2) == ny/2
38 -      ny = ny+1;
39 -    end
40 -
41 -    D_s = zeros(ny,nx);
42 -    deltaX_s = (xmax_s-xmin_s)/nx;
43 -    deltaY_s = (ymax_s-ymin_s)/ny;
44 -    i0 = ceil(c/2);
45 -    j0 = ceil(f/2);
46 -    i0_s = ceil(nx/2);
47 -    j0_s = ceil(ny/2);
48 -
49 -    for n = 1:8
50 -      for m = 1:8
51 -        Tt = T{n}/T{m};
52 -        for i = 1:c
53 -          for j = 1:f
54 -            if D(j,i) > 0
55 -              x = deltaX*(i-i0);
56 -              y = deltaY*(j-j0);
57 -              xy_s = Tt*[x;y;1];
58 -              x_s = xy_s(1);
59 -              y_s = xy_s(2);
60 -              i_s = floor((x_s)/deltaX_s+0.5)+i0_s;
61 -              j_s = floor((y_s)/deltaY_s+0.5)+j0_s;
62 -              D_s(j_s,i_s) = D_s(j_s,i_s)+D(j,i);
63 -            end
64 -          end
65 -        end
66 -      end
67 -    end
68 -
69 - end
```

comprob_sec.m

```
1 - T_encontrado = eye(3);
2 - for k=1:N
3 -   i = secuencia(k);
4 -   j = secuencia(k+N);
5 -   if i==-1 || j==-1
6 -     continue
7 -   end
8 -   T_encontrado = T_encontrado*T{i}/T{j};
9 - end
```

```
10 - P_encontrada = [T_encontrado(1,3) T_encontrado(2,3)]
11 - phi_encontrada = atan2(T_encontrado(2,1),T_encontrado(1,1));
```

plot_espacio_trabajo.m

```
 1 - [D,i0_f,j0_f,xmax,xmin,ymax,ymin] = calculo_malla_espacio_trabajo(N,T);
 2 -
 3 - u = unique(D);
 4 - u(1) = [];
 5 - num_colores = length(u)-1;
 6 - delta_col = max(u)/num_colores;
 7 - v_aux = [0;delta_col*ones(num_colores,1)];
 8 - v_ac = cumsum(v_aux);
 9 -
10 - pos_col = zeros(size(D,2)*size(D,1),5);
11 - cont = 0;
12 - deltaX = (xmax-xmin)/size(D,2);
13 - deltaY = (ymax-ymin)/size(D,1);
14 -
15 - for i = 1:size(D,2)
16 -   for j = 1:size(D,1)
17 -     if D(j,i) == 0
18 -       continue
19 -     end
20 -     cont = cont+1;
21 -     x = deltaX*(i-i0_f);
22 -     y = deltaY*(j-j0_f);
23 -     esc = floor(1000000*log(D(j,i))/log(max(u)))/1000000;
24 -     color = [esc 0 0];
25 -     pos_col(cont,:) = [x y color];
26 -   end
27 - end
28 -
29 - pos_col(cont+1:end,:) = [];
30 -
31 - figure
32 - x = pos_col(:,1);
33 - y = pos_col(:,2);
34 - color = pos_col(:,3:5);
35 - scatter(x, y, 20, color, 'Filled')
36 -
37 - axis equal
```

plot_robot.m

```
 1 - hold on
 2 -
 3 - P0 = [0 0];
 4 - phi0 = 0;
 5 -
 6 - i_v = secuencia(1:N);
 7 - j_v = secuencia(N+1:end);
 8 -
 9 - for i = 1:N
10 -   [P1,phi1] = plot_modulo(T,P0,phi0,i_v(i),j_v(i));
11 -   P0 = P1;
12 -   phi0 = phi1;
13 - end
14 - plot(bee(1),bee(2),'xy','Linewidth',2)
```

plot_modulo.m

```
 1 - function [P2,phi2] = plot_modulo(T,P0,phi0,i,j)
 2 -
```

```
  b = 18.59;
  p = 101.31;

  T0 = [cos(phi0) -sin(phi0) P0(1); sin(phi0) cos(phi0) P0(2); 0 0 1];
  T1 = T0*T{i};
  T2 = T1/T{j};

  P2 = T2(1:2,3);
  phi2 = atan2(T2(2,1),T2(1,1));

  A01 = T0*[-b;0;1];
  A02 = T0*[b;0;1];
  B11 = T1*[-p;0;1];
  B12 = T1*[p;0;1];
  A21 = T2*[-b;0;1];
  A22 = T2*[b;0;1];

  plot([A01(1),A02(1)],[A01(2),A02(2)],'g','LineWidth',2)
  hold on
  set(gca,'DataAspectRatio',[1,1,1])
  plot([B11(1),B12(1)],[B11(2),B12(2)],'c','LineWidth',2)
  plot([A21(1),A22(1)],[A21(2),A22(2)],'g','LineWidth',2)
  plot([A01(1),B12(1)],[A01(2),B12(2)],'color',[0.5,0.5,0.5],'LineWidth',2);
  plot([A02(1),B11(1)],[A02(2),B11(2)],'color',[0.5,0.5,0.5],'LineWidth',2);
  plot([B11(1),A22(1)],[B11(2),A22(2)],'color',[0.5,0.5,0.5],'LineWidth',2);
  plot([B12(1),A21(1)],[B12(2),A21(2)],'color',[0.5,0.5,0.5],'LineWidth',2);

end
```

ANEXO 7

Pseudocódigo de la función A*

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path
// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather than a hash-
    // set.
    openSet := {start}
    // For node n, cameFrom[n] is the node immediately preceding it on the cheapest
    // path from start
    // to n currently known.
    cameFrom := an empty map
    // For node n, gScore[n] is the cost of the cheapest path from start to n currently
    // known.
    gScore := map with default value of Infinity
    gScore[start] := 0
    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best
guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)
    while openSet is not empty
        // This operation can occur in O(Log(N)) time if openSet is a min-heap or a
        // priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)
        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through
            // current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one. Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)
    // Open set is empty but goal was never reached
    return failure
```

Referencias bibliográficas

R. Clysdale, Q. Sun, Motion planning for planar binary robots in a reduced workspace, in: IEEE 2005 International Conference Mechatronics and Automation, Vol. 1, IEEE, 2005, pp. 388–393. doi:10.1109/ICMA.2005.1626578.

I.-M. Chen, S. H. Yeo, Locomotion and navigation of a Planar Walker based on binary actuation, in: Proceedings of the 2002 IEEE International Conference on Robotics and Automation, Vol. 1, IEEE, 2002, pp. 329–334. doi: 10.1109/ROBOT.2002.1013382.

G. S. Chirikjian, Inverse kinematics of binary manipulators using a continuum model, Journal of Intelligent and Robotic Systems 19 (1) (1997) 5–22. doi:10.1023/A:1007942530293.

I. Ebert-uphoff, G. S. Chirikjian, Actuated Hyper-Redundant Manipulators Using Workspace Densities, in: Proceedings of IEEE International Conference on Robotics and Automation (1996). doi:10.1109/robot.1996.503586.

I. Ebert-uphoff, G. S. Chirikjian, Efficient workspace generation for binary manipulators with many actuators, Journal of Robotic Systems 12 (6) (1995) 383–400. doi:10.1002/rob.4620120605.

J. Gallego Pagán, Desarrollo de algoritmos para la navegación de un robot modular con un único actuador en entornos conocidos, Universidad Miguel Hernández de Elche (2020).

A. García Martínez, Desarrollo de un robot móvil paralelo con actuación binaria, Universidad Miguel Hernández de Elche (2020).

D. S. Lees, G. S. Chirikjian, An efficient method for computing the forward kinematics of binary manipulators, in: Proceedings of IEEE International Conference on Robotics and Automation, Vol. 2, IEEE, 1996, pp. 1012–1017. doi: 10.1109/ROBOT.1996.506841.

K. Maeda, E. Konaka, Ellipsoidal outer-approximation of workspace of binary manipulator for inverse kinematics solution, in: 2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics, IEEE, 2014, pp. 1331– 1336. doi:10.1109/AIM.2014.6878267.

Y. Miao, F. Gao, Y. Zhang, Gait fitting for snake robots with binary actuators, Science China Technological Sciences 57 (1) (2014) 181–191. doi:10.1007/s11431-013-5405-0.

A. Peidró Vidal, J.M. Marín López, M. Ballesta Galdeano, O. Reinoso García, L. Payá Castelló, y L.M. Jiménez García. "ROBOT MÓVIL DESPLAZABLE EN UN PLANO". Patente nacional número ES2853898. Fecha de prioridad: 17.03.2020. Fecha de concesión: 04.02.2022. Titular: UNIVERSIDAD MIGUEL HERNÁNDEZ.

A. Peidro, J. María Marín, A. Gil, O. Reinoso, Performing nonsingular transitions between assembly modes in analytic parallel manipulators by enclosing quadruple solutions, Journal of Mechanical Design 137 (12) (2015) 122302. doi: 10.1115/1.4031653

D. Schutz, A. Raatz, J. Hesselbach, The development of a reconfigurable parallel robot with binary actuators, in: Advances in Robot Kinematics: Motion in Man and Machine, Springer, 2010, pp. 225–232. doi:10.1007/ 978-90-481-9262-5_24.

D. Schutz, A. Raatz, J. Hesselbach, Adapted task configuration of a reconfigurable binary parallel robot with PRRRP structure, Robotica 31 (2) (2013) 285–293. doi:10.1017/S0263574712000240.

V. A. Sujan, S. Dubowsky, Design of a lightweight hyper-redundant deployable binary manipulator, Journal of Mechanical Design 126 (1) (2004) 29–39. doi:10.1115/1.1637647.

E. Tzorakoleftherakis, A. Mavrommati, A. Tzes, Design and implementation of a binary redundant manipulator with cascaded modules, Journal of Mechanisms and Robotics 8 (1) (2016) 011002. doi:10.1115/1.4030372.

Y. Zhou, On the planar stability of rigid-link binary walking robots, Robotica 21 (6) (2003) 667–675. doi:10.1017/ S0263574703005162.