UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

PLANIFICACIÓN DE MOVIMIENTOS DE UN ROBOT MÓVIL PARALELO CON ACTUADORES BINARIOS

TRABAJO FIN DE GRADO

Julio - 2021

AUTOR: Mario Pérez Checa

DIRECTOR: Adrián Peidró Vidal

ÍNDICE

Capítulo 1. Introducción. 4

1.1. Antecedentes del Trabajo Fin de Grado. 4

1.2. Objetivos. 6

1.3. Estructura de la memoria. 7

1.4. Punto de partida. 8

Capítulo 2. Alcance de posición deseada. 10

Capítulo 3. Alcance de posición y orientación deseadas. 15

3.1. Método de selección de punto óptimo. 15

3.2. Método de pesos relativos. 21

3.3. Comparativa de métodos. 24

Capítulo 4. Alcance de posición deseada mediante solapamiento de espacios de trabajo. 26

Capítulo 5. Alcance de posición y orientación deseadas mediante solapamiento de espacios de trabajo. 32

Capítulo 6. Animación del robot. 42

6.1. Cinemática inversa y directa. 43

6.2. Animación entre dos configuraciones adyacentes. 45

6.3. Obtención de la secuencia entre dos configuraciones no-adyacentes. 53

6.4. Animar medio ciclo y un ciclo completo. 55

6.5. Animación de una trayectoria multi-ciclo. 57

6.6. Ejemplo final 59

Capítulo 7. Consideración de obstáculos. 63

7.1. Obstáculos puntuales. 63

7.2. Obstáculos no puntuales. 79

Capítulo 8. Conclusiones y trabajos futuros. **85**

ANEXOS **88**

Anexo 0 .... **89**

Anexo 1 .... **91**

Anexo 2 .... **94**

Anexo 3 .... **100**

Anexo 4 .... **104**

Anexo 5 .... **112**

Anexo 6 .... **121**

Anexo 7 .... **132**

Referencias bibliográficas **143**

Capítulo 1. Introducción.

1.1. Antecedentes del Trabajo Fin de Grado.

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 [1], Maeda and Konaka, 2014 [2], Tzorakoleftherakis et al., 2016 [3], Sujan and Dubowsky, 2004 [4], Miao et al., 2014 [5], Chirikjian et al., 1995-1996-1997 [6-7-8].

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 [9-10], Zhou, 2003 [11], Chen y Yeo, 2002 [12]. 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 Grado.

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 [13], 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 [14], disponiendo los actuadores binarios de

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

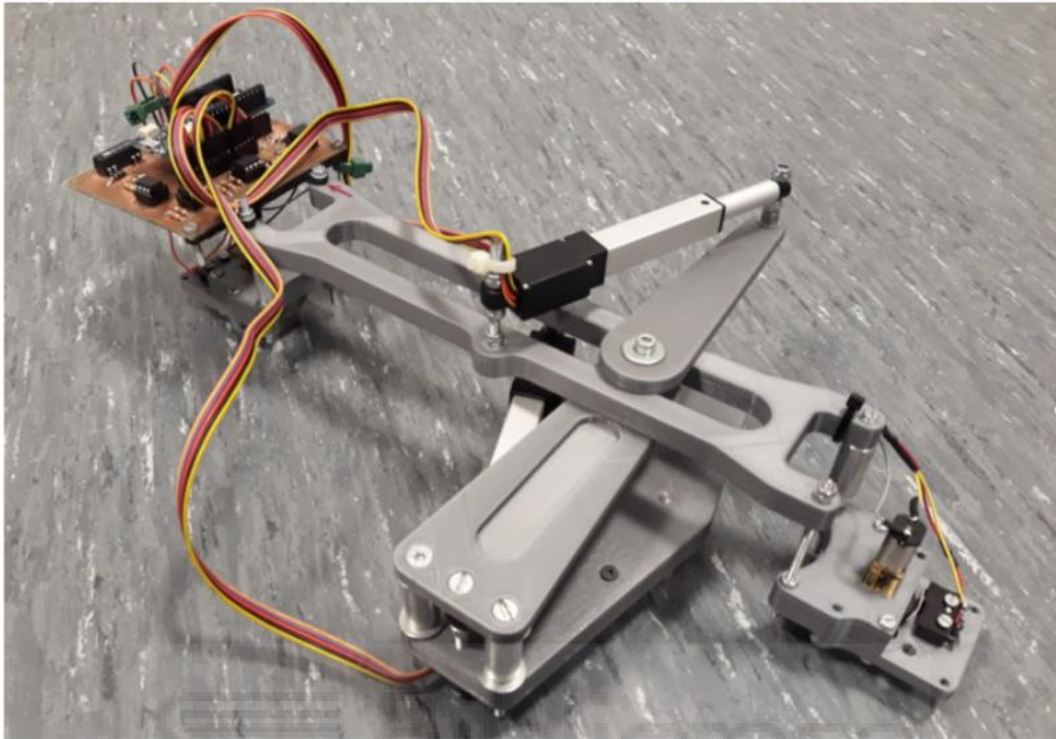


Figura 1: Prototipo del robot Xrobin. Figura extraída de [13] con permiso.

biblioteca
UNIVERSITAS Miguel Hernández

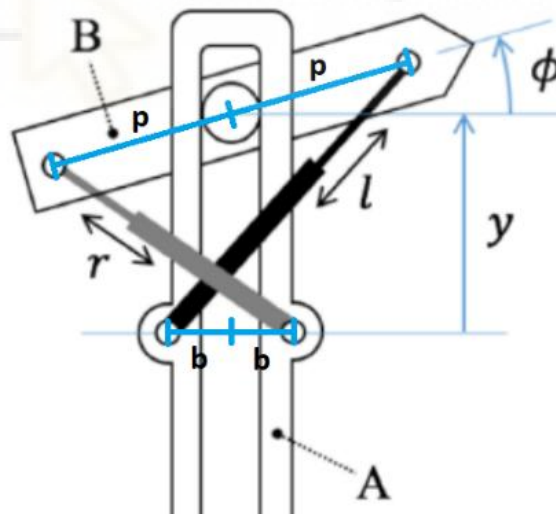


Figura 2: Parámetros del robot.

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 Grado se encuentra en proceso de protección por patente por parte de los profesores del grupo ARVC, cuyo prototipo, mostrado en la Figura 1, se desarrolló en un Trabajo Fin de Grado previo a este, García-Martínez, 2020 [14]. 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 previo no se resolvió la planificación de trayectorias complejas que permitan alcanzar cualquier posición y orientación deseadas, que es lo que se aborda en este Trabajo Fin de Grado.

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.

1.2. Objetivos.

El objetivo de este Trabajo Fin de Grado consiste en la resolución de la planificación de trayectorias del robot móvil binario explicado en la sección 1.1 de este proyecto. Para este fin, se han desarrollado una serie de métodos diferentes que pretenden dar solución a este problema considerando la posición, la orientación, el número de ciclos de trabajo que se requieren para alcanzar una referencia (número de despegues de imanes) así como posibles obstáculos en el camino. El resultado que se espera obtener es elaborar un algoritmo que sea capaz de conducir al robot a través del espacio 2D hasta cualquier punto que se le pida alcanzar y con la orientación que se le requiera. Con el fin de comprobar que este objetivo se cumple, se realizará un algoritmo que anime el movimiento del robot con tal de comprobar si este funciona correctamente.

Biblioteca UNIVERSITAS Miguel Hernández

1.3. Estructura de la memoria.

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 planificación de trayectorias del robot para alcanzar de la forma más precisa posible una posición determinada dentro del espacio de trabajo. Esto se consigue mediante la comparativa de todos los puntos posibles del espacio de trabajo con el punto de referencia y escogiendo el más cercano a este.

En el capítulo 3 se amplía el problema del capítulo 2, imponiendo restricciones de orientación al robot para que así no solamente llegue lo más cerca posible al punto deseado, sino también con una orientación lo más parecida posible a la deseada. Este problema se ha abordado mediante dos métodos diferentes: el primero consiste en la selección del punto más óptimo de entre unos cuantos aplicando unos parámetros de diseño predefinidos, y el segundo se basa en dar un peso relativo a la orientación con respecto a la posición, haciendo que se priorice una u otra dependiendo de la importancia que se le haya dado a cada una.

El capítulo 4 pretende resolver la planificación de trayectorias del robot de manera que este sea capaz de alcanzar cualquier punto del espacio 2D libre de obstáculos. Esto conlleva que el robot deba salirse del espacio de trabajo, para lo que se ha planteado un algoritmo que consiste en solapar espacios de trabajo hasta que este alcance el punto deseado.

El capítulo 5 se basa en ampliar la solución dada en el capítulo 4 pero no solamente en términos de posición, sino también en términos de orientación. Para ello, se ha elaborado un algoritmo que se encarga de, una vez alcanzada una solución en posición, orientar al robot de forma que consiga alcanzar la orientación deseada.

El capítulo 6 consiste en animar el movimiento del robot a través del espacio con el fin de comprobar si este desarrolla el movimiento de forma correcta. Para ello, se ha realizado un código que muestra una secuencia de fotogramas que muestran el movimiento del robot tal y como si se tratase de un vídeo.

Finalmente, el capítulo 7 tratará sobre la consideración de obstáculos en la trayectoria del robot y cómo evitar los mínimos locales. Con este fin, se ha desarrollado un algoritmo que se encarga de que el robot evite los obstáculos y que, en caso de encontrarse con un mínimo local, intente salir de él dirigiéndose a una posición aleatoria que le permita retomar su camino hasta la referencia.

1.4. Punto de partida.

Para la resolución de la planificación de trayectorias del robot objeto del presente Trabajo Fin de Grado, se han propuesto una serie de experimentos que planean hallar la solución más óptima posible al problema planteado.

Se parte de un código, facilitado por el director de este trabajo, que se encuentra reflejado en el Anexo 0, y consta de dos *scripts* elementales: **ws_binario_mp.m** y **ciclo_completo.m**. Dicho código se encarga de calcular los puntos correspondientes al espacio de trabajo del robot, almacenándolos en una variable global llamada *nube*, que será una de las variables más utilizadas a lo largo de todo el código perteneciente a este proyecto.

El *script* **ws_binario_mp.m** se encarga de asignar los parámetros de diseño del robot a las variables correspondientes *phi* e *y*, de crear un objeto tipo *cell* o *celda*, que actúa como matriz de matrices y almacena entre $T\{1\}$ y $T\{8\}$ las 8 posibles transformaciones entre los cuerpos A y B de este robot (correspondientes a las 8 posibles combinaciones de *phi* e *y*, que resultan de actuar el robot de forma binaria, es decir, cuando cada uno de sus actuadores lineales está totalmente retraído o totalmente extendido, y se muestran en la Figura 3), y de llamar a la función **ciclo_completo.m**, que se ocupa de calcular los puntos pertenecientes al espacio de trabajo del robot, calculando recursivamente todos los valores de T_B, que representa la posición y orientación absolutas del eslabón B para el movimiento del primer medio ciclo, y posteriormente los valores de T_A para obtener la posición y orientación absolutas de A para el movimiento del segundo medio ciclo.

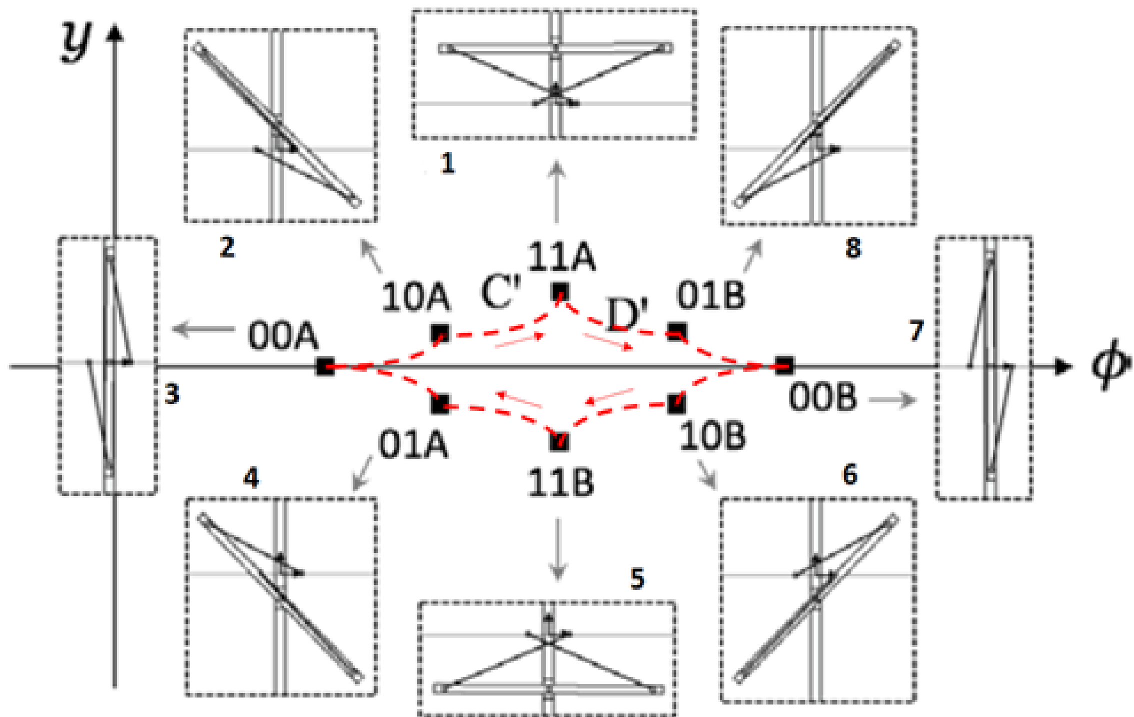


Figura 3: Configuraciones posibles del robot.

Este es el código sobre el que se va a desarrollar todo el trabajo. En los próximos apartados se expondrán una serie de experimentos que pretenden resolver el problema planteado, empezando desde el caso más sencillo posible hasta llegar a resolver la planificación de trayectorias en el plano, pudiendo llegar a una posición y orientación finales deseadas evadiendo obstáculos del entorno, exponiendo sus ventajas y sus inconvenientes.

Capítulo 2. Alcance de posición deseada.

La primera situación que se plantea es la de encontrar el punto más cercano a una referencia predefinida que el robot puede alcanzar en un número de ciclos determinado, sin tener en cuenta la orientación en la que este se encuentre al finalizar el recorrido. A pesar de que se haya diseñado para poder realizar un número de ciclos cualquiera, los experimentos se realizarán con N=3 ciclos como máximo, puesto que, a partir de dicho número, el ordenador con el que se realizan las computaciones no es capaz de calcular un número tan elevado de puntos (por ejemplo, para el caso de N=4 ciclos, es necesario calcular más de 17 millones de puntos). En la mayoría de los casos se trabajará con N=2 ciclos, puesto que se considera que es el más idóneo a nivel de rendimiento de la máquina y es suficientemente preciso.

Para resolver este primer problema, se ha elaborado un *script* que calcule la distancia entre la posición de referencia que se le pida alcanzar al robot y todas las posiciones posibles que el mismo puede alcanzar, y de todas las distancias calculadas, encuentre el índice correspondiente al punto de mínima distancia dentro de la matriz que almacena todas las posiciones y orientaciones.

Dicho *script*, llamado **encontrar_punto.m**, se encuentra en el Anexo 1, acompañado de los scripts originales (**ws_binario_mp.m** y **ciclo_completo.m)** con ciertas modificaciones que se comentarán posteriormente. La función principal de este código se basa en calcular la distancia entre el punto de referencia predefinido y todos los puntos (los almacenados en la variable *nube*) que el robot puede alcanzar tras *N* ciclos.

```
15  -   for i=1:size(nube,1)
16  -       distancia = norm(P_ref-nube(i,1:2));
17  -       if distancia < distancia_minima
18  -           indice_min = i;
19  -           distancia_minima = distancia;
20  -           P_mas_cercano = nube(i,1:2);
21  -       end
22  -   end
```

En cada iteración del bucle, se calcula la distancia como la norma entre el punto de referencia y el actual de cálculo, y dicha distancia se almacena en una variable (línea 16) y se actualiza, junto con el índice y el punto, cuando se encuentra un valor menor de

distancia al almacenado anteriormente (líneas 17-21), lo que hace que al final del bucle se haya obtenido el punto de mínima distancia y su índice dentro de la variable *nube*.

Hasta aquí se podría dar por completado el *script* que resuelve esta situación, pero con el fin de comprobar si el cálculo se ha realizado correctamente, se han modificado **ws_binario_mp.m** y **ciclo_completo.m** para que se almacenen en una matriz las secuencias que el robot sigue para alcanzar cada punto. En el primero, únicamente se ha creado la variable global *secuencia* donde almacenar los valores correspondientes y se ha añadido el llamamiento a los otros dos scripts para que la ejecución sea automática. En el segundo, se han tenido que añadir varias líneas para poder almacenar los índices de las secuencias. Cada secuencia se compone de los valores que toman *i* (pose relativa entre B y A elegida para el primer medio ciclo) y *j* (pose relativa entre A y B elegida para el segundo medio ciclo) para alcanzar cada una de las posibles soluciones, por lo que cada secuencia tendrá un tamaño del doble del número de ciclos, el primer medio vector almacenará los valores de *i* mientras que el segundo almacenará los valores de *j*.

```
29 -  if n==1
30 -      v_i(1)=i;
31 -      v_j(1)=j;
32 -      secuencia(global_counter,:) = [v_i v_j];
33 -  else
34 -      Auxiliar = secuencia(global_counter-1,:);
35 -      auxiliar(n)=i;
36 -      auxiliar(N+n)=j;
37 -      secuencia(global_counter,:) = auxiliar;
38 -  end
```

Biblioteca
UNIVERSITAS Miguel Hernández

Lo que se ha hecho para que los valores se almacenen correctamente es comprobar si es el primer valor de la secuencia, en cuyo caso se almacena como un vector de valores -1 salvo los primeros valores de *i* y *j* (líneas 3-32), que serán los que correspondan en cada momento, y en caso de no ser el primer índice almacenado, se copia el vector *secuencia* anterior y se añade el siguiente índice (líneas 34-38).

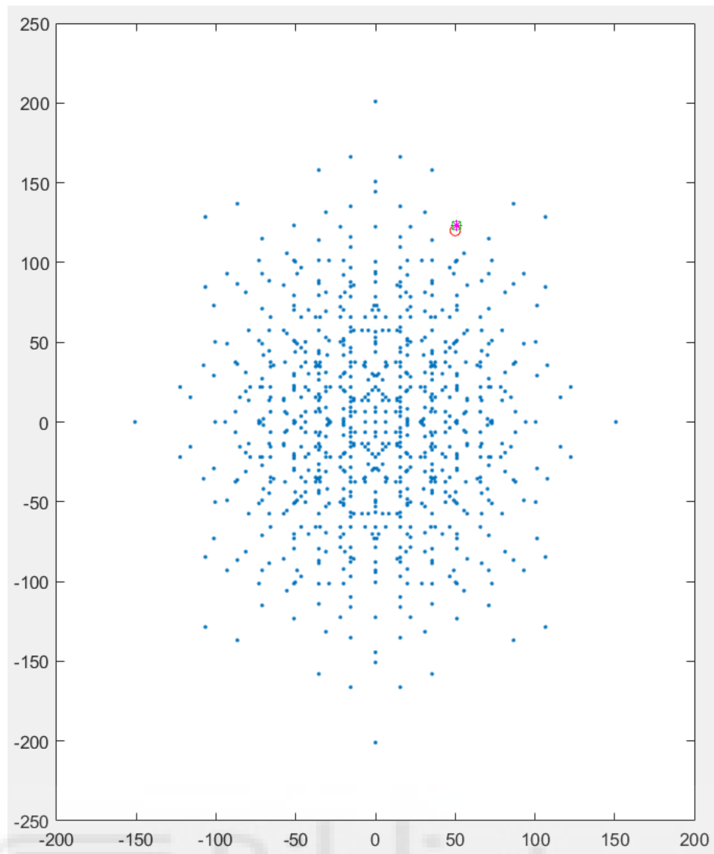
Los valores de cada secuencia corresponden con los índices de los bucles que se encargan de calcular los puntos, y como las secuencias se almacenan en la misma posición que las posiciones y orientaciones, el punto de mínima distancia tendrá el mismo índice en la matriz de secuencias, por lo que sabiendo la secuencia que han seguido *i* y *j*,

podemos hallar la posición final del robot y compararla con la obtenida mediante el algoritmo de calcular la distancia mínima.

```
28  -   for k=1:N
29  -       i = secuencia_encontrada(k);
30  -       j = secuencia_encontrada(k+N);
31  -       if i==-1 || j==-1
32  -           continue
33  -       end
34  -       T_encontrado = T_encontrado*T{i}*inv(T{j});
35  -   end
```

El cálculo de la posición a partir de la secuencia también se realiza en **encontrar_punto.m**, y esta consiste en realizar la misma operación que se realiza al calcular la nube de puntos, pero sólo para los valores de *i* y *j* almacenados en la secuencia definida por el índice que corresponde con el punto de distancia mínima. Para ello, se recorre el vector de la secuencia de forma que se tomen correctamente los valores de *i* y *j* (líneas 29-30), y en cada iteración se aplica la ecuación que aparece en la línea 34. Al finalizar el bucle, *T_encontrado* contendrá los valores del punto más cercano a la referencia, así como su orientación, que se utilizará en cálculos de otros experimentos más adelante.

Lo que se puede ver a continuación son unas imágenes del resultado de este primer experimento donde se puede apreciar la nube de puntos en azul, el punto de referencia en forma de un círculo rojo, y las soluciones obtenidas mediante el algoritmo de distancia mínima y el algoritmo de la secuencia se han representado en forma de cuadrado verde y asterisco magenta respectivamente. Los ejes que representan las posiciones se encuentran en milímetros, y el punto de referencia se ha establecido en (120,50).



Biblioteca UNIVERSITAS Miguel Hernández

Figura 4: Solución del primer experimento.

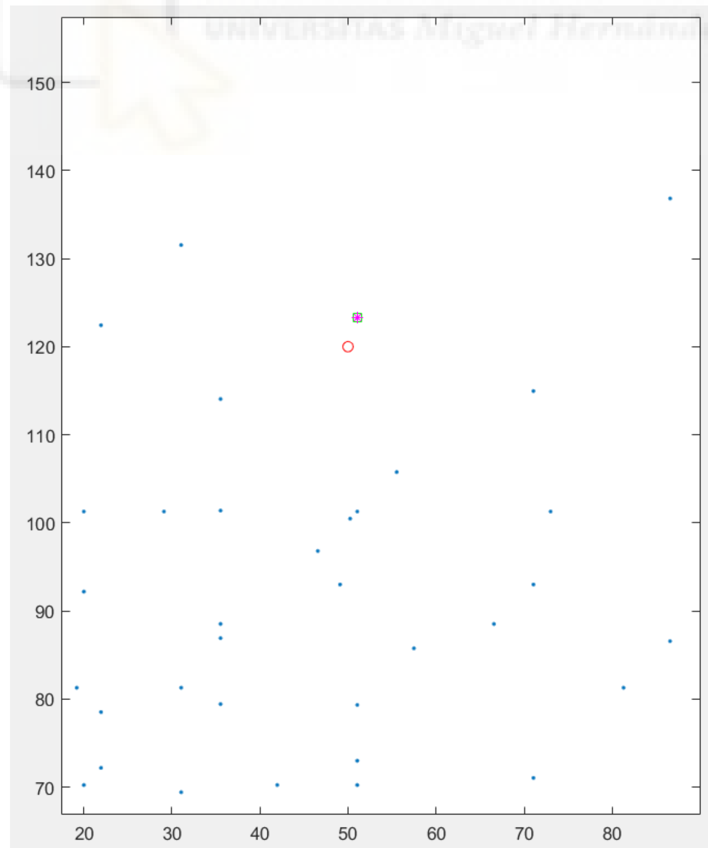


Figura 5: Ampliación de la Figura 4.

El error en posición que se ha obtenido, teniendo en cuenta que, para más clarividencia se ha elegido un punto de la periferia del espacio de trabajo con tal de que se aprecie correctamente en las imágenes, es de 3.4134 mm, habiéndose encontrado el punto más cercano en (51.0508, 123.2476), con un coste de 2 ciclos de movimiento del robot.

Biblioteca
UNIVERSITAS Miguel Hernández

Capítulo 3. Alcance de posición y orientación deseadas.

El segundo problema que se ha planteado es que el robot alcance una posición cercana a la de referencia, pero teniendo en cuenta la orientación, por lo que hay que ponderar cuánto importa la orientación con respecto a la posición. En casos en los que la orientación no tenga demasiada relevancia, se le restará importancia a esta y se priorizará la posición, y en los casos donde la orientación sea muy importante, se debe intentar encontrar una posición no muy desfavorable para poder tener una orientación al menos similar a la deseada. Este último caso es un problema, puesto que pueden existir situaciones en las que la posición sea muy desfavorable por darle demasiada importancia a la orientación y ello provoque que las soluciones propuestas no sean las mejores. Por ello, se han planteado dos soluciones posibles para este problema, atendiendo a distintos criterios de diseño.

3.1. Método de selección de punto óptimo.

La primera solución que se plantea para este problema consiste en, a grandes rasgos, determinar un rango de puntos admisibles y elegir el más óptimo entre ellos. Para ello, contamos con los *scripts* que se encuentran en el Anexo 2, que se compone de **ws_binario_mp.m** y **ciclo_competo.m**, que son iguales a sus homónimos del experimento anterior, **encontrar_punto**, que también es el mismo que en el caso anterior pero se ha modificado para que se tenga en cuenta la orientación a la hora de hacer los cálculos, y **calculo_phi_dif.m** que es un *script* nuevo que calcula la diferencia angular entre dos orientaciones.

El primer código que se explicará será **calculo_phi_dif.m**, que como se acaba de mencionar, consiste en una función que calcula la diferencia entre dos ángulos, y su función principal es que devuelva la diferencia entre dos ángulos de entrada dentro del intervalo [-π,π]. El objetivo es obtener la menor diferencia angular que existe entre dos ángulos dados, que equivale a encontrar el menor ángulo a recorrer para conectar dos puntos del círculo unidad.

```
3   -  phi_dif_tmp = phi_ref-resultado(3);
4
5   -  if phi_dif_tmp > pi
6   -      phi_dif_tmp = phi_dif_tmp-2*pi;
```

```
7   -  end
8   -  if phi_dif_tmp < -pi
9   -      phi_dif_tmp = phi_dif_tmp+2*pi;
10  -  end
11
12  -  phi_dif = abs(phi_dif_tmp);
```

Las entradas de esta función son el ángulo de referencia que se quiere alcanzar en la posición final del robot (*phi_ref*) y la fila de la variable *nube* correspondiente a cada momento, que se guarda en la variable *resultado*, cuya tercera columna corresponde con la orientación. Como dichos ángulos de entrada siempre van a estar entre [-π,π], la diferencia sin aplicar este algoritmo entre ellos siempre va a dar un resultado entre [-2π,2π], por lo que es necesario transformar los valores mayores a π en valores entre [-π,0] (líneas 5-6) y los valores menores a -π en valores entre [0,π] (líneas 8-9). Al aplicar estas condiciones, en la salida *phi_dif* siempre habrá un valor entre [-π,π], que se devolverá a la función principal.

En cuanto a las modificaciones realizadas en el *script* **encontrar_punto**, estas son considerables teniendo en cuenta que se ha cambiado por completo la forma de calcular el punto, ya que no sirve descartar un valor cuando otro está más cerca de la referencia, sino que al tener que considerar la orientación, no se puede descartar una solución tan fácilmente.

```
18  -  for u = 1:size(nube,1)
19  -      v_distancia(u) = norm(P_ref-nube(u,1:2));
20  -  end
21
22     % Cálculo del punto considerando la orientación
23  -  while 1
24  -      cont=cont+1;
25  -      [minimo,indice_min] = min(v_distancia);
26  -      resultado = nube(indice_min,:);
27  -      phi_dif = calculo_phi_dif(phi_ref,resultado);
28  -      if v_distancia(indice_min) < radio
29  -          if abs(phi_dif) <= pi/4
30  -              break
31  -          else
32  -              res_aux = [res_aux;resultado];
33  -              dist_aux = [dist_aux;minimo];
34  -              ind_aux = [ind_aux;indice_min];
35  -              v_distancia(indice_min) = 9999;
36             end
```

```
37 -      else
38 -          if cont==1
39 -              break
40 -          else
41 -              [tmp,indice_tmp] = min(dist_aux);
42 -              resultado = res_aux(indice_tmp);
43 -              indice_min = ind_aux(indice_tmp);
44 -              break
45 -          end
46 -      end
47 - end
```

En primer lugar, se crea un vector *v_distancia* que almacena todas las distancias entre el punto de referencia y todos los puntos del espacio de trabajo (líneas 18-20), para posteriormente escoger de entre las opciones más cercanas a la referencia cuál satisface el criterio de orientación que se aplique a cada caso.

La idea principal de esta solución es determinar una circunferencia, con centro en el punto de referencia, donde se encuentren todas las opciones admisibles en cuanto a posición, definida por la variable *radio* (definida por defecto en 20 mm) y dentro de dichas opciones admisibles se localice la más cercana a la referencia que cumpla un criterio de orientación predefinido en la línea 29, el cual por defecto se ha establecido que una solución válida tendría una diferencia máxima de $\pm\pi/4$ con la referencia (las diferencias deben ser múltiplos de $\pi/4$ debido a la actuación binaria). Se ha establecido como criterio que, si no se encuentra ninguna solución válida dentro de la circunferencia de puntos admisibles, se tome como solución el más cercano en posición, sea cual sea su orientación, puesto que se considera que un punto fuera de la circunferencia admisible es una solución muy desfavorable en posición, y por tanto se escoge la más cercana. En caso de no querer tomar este criterio, lo más recomendable es aumentar el radio de la circunferencia.

Para lograr este objetivo, se ha establecido un bucle *while* (línea 23) que se sigue ejecutando hasta que se encuentra la solución final.

En primer lugar, se busca el punto de mínima distancia con la referencia y su índice dentro de la variable *v_distancia*, y por tanto de *nube* (línea 25). Posteriormente se almacena en una variable independiente llamada *resultado* (línea 26) y se hace llamar a la función **calculo_phi_dif** (línea 27). Tras ella, se pueden dar las siguientes situaciones:

- Si el punto analizado se encuentra dentro de la circunferencia admisible, se comprueba si se cumple o no el criterio de diferencia angular (líneas 28-36).

- En cambio, si se encuentra fuera de la circunferencia admisible, se establecen los criterios pertinentes, que se explicarán a continuación, y se interrumpe el bucle (líneas 37-45).

En el primer caso, se puede dar que la comparación entre la diferencia angular del punto analizado sea verdadera, lo que interrumpe el bucle y da como solución final la variable *resultado* (líneas 29-30); y también puede darse que dicha comparación sea falsa, en cuyo caso se almacenan las variables *resultado*, *distancia* e *indice_min* en variables auxiliares para acceder a ellas si es necesario (líneas 31-36), puesto que inmediatamente después de almacenarlas se descartan, modificando la variable general *v_distancia* para que en la posición del punto actual se almacene el valor 9999 (línea 35) y así se descarte como valor mínimo en las siguientes iteraciones. Al darse este caso, no se da el segundo, por lo que el bucle procede a realizar otro ciclo hasta o bien encontrar un punto que cumpla la comparación de diferencia angular, o bien un punto que no esté dentro de la circunferencia admisible. Si en una de estas iteraciones se ha encontrado un punto que cumpla con la diferencia angular predefinida, se habrá encontrado un punto que cumple tanto con el criterio de la diferencia angular como con el criterio de la circunferencia admisible, por lo que se considerará una solución buena.

En el segundo caso, se ha llegado a un punto que está fuera de la circunferencia, por lo que, en caso de que sea la primera iteración del bucle (que se determina con la variable *cont*), este se interrumpirá y no se obtendrá ningún punto como válido (líneas 38-39), y en caso de que no sea la primera iteración, se aplicará el criterio que se ha descrito anteriormente, y es el de determinar que la solución final será el punto más cercano a la referencia, sin importar la orientación (líneas 40-45), puesto que ya se considera que se ha obtenido una solución muy desfavorable en posición. Para ello, se debe recurrir a las variables auxiliares debido a que en la variable general *v_distancia* se han ido descartando los valores ya usados, por lo que se deben utilizar las auxiliares. Con este fin, se debe obtener el índice del mínimo valor de la variable auxiliar que almacena las distancias (*dist_aux*), y con dicho índice, obtener los valores correspondientes de las

variables auxiliares de resultado (*res_aux*) e índice (*ind_aux*), y devolverlos como resultados finales (líneas 41-43).

Cuando ya hemos obtenido un resultado a partir del código anterior, debemos comprobar si este es correcto, y para ello debemos aplicar el mismo algoritmo que se utilizó para calcular la posición del punto a través de la secuencia, y para ello se ha obtenido, aparte de la solución final, su índice dentro de la variable *nube*, y por tanto de *secuencia*, por lo que disponemos de la secuencia que el robot ha seguido para alcanzar la solución (línea 49, mostrada en el siguiente código).

```
49 - secuencia_encontrada = secuencia(indice_min,:);
50
51   % Dibujo de la circunferencia de puntos admisibles
52 - th = 0:pi/50:2*pi;
53 - xunit = radio*cos(th)+P_ref(1);
54 - yunit = radio*sin(th)+P_ref(2);
55 - h = plot(xunit,yunit);
56
57   % Utilizamos la secuencia obtenida para hallar el punto correcto
58 - for k=1:N
59 -     i = secuencia_encontrada(k);
60 -     j = secuencia_encontrada(k+N);
61 -     if i==-1 || j==-1
62 -         continue
63 -     end
64 -     T_encontrado = T_encontrado*T{i}*inv(T{j});
65 - end
```

Como ya se ha comentado, el algoritmo empleado, que se encuentra entre las líneas 58 y 65, ya se ha empleado en otro experimento, por lo que no se incidirá de nuevo en ello, pero sí se mencionará que se ha hecho un pequeño código, alojado entre las líneas 52 y 55, que se encarga de la representación gráfica de la circunferencia de puntos admisibles con el fin de que sea más visual a la hora de comprobar si el resultado es correcto.

A continuación, se muestran unas imágenes de los resultados obtenidos a partir de este algoritmo. El punto de referencia se ha colocado en el punto (90,70), representado en forma de círculo rojo, y la orientación de referencia se ha puesto a pi/2, representada mediante una flecha roja. Los criterios de diferencia angular máxima y radio de la circunferencia admisible se han dejado por defecto, a $\pm\pi/4$ y 20 mm respectivamente. La

circunferencia se ha representado en color naranja, con centro el punto de referencia. Las soluciones obtenidas mediante el algoritmo principal (la variable *resultado*) y el algoritmo de la secuencia se han representado en forma de cuadrado verde y asterisco magenta respectivamente, y la orientación de la solución se encuentra como una flecha negra.

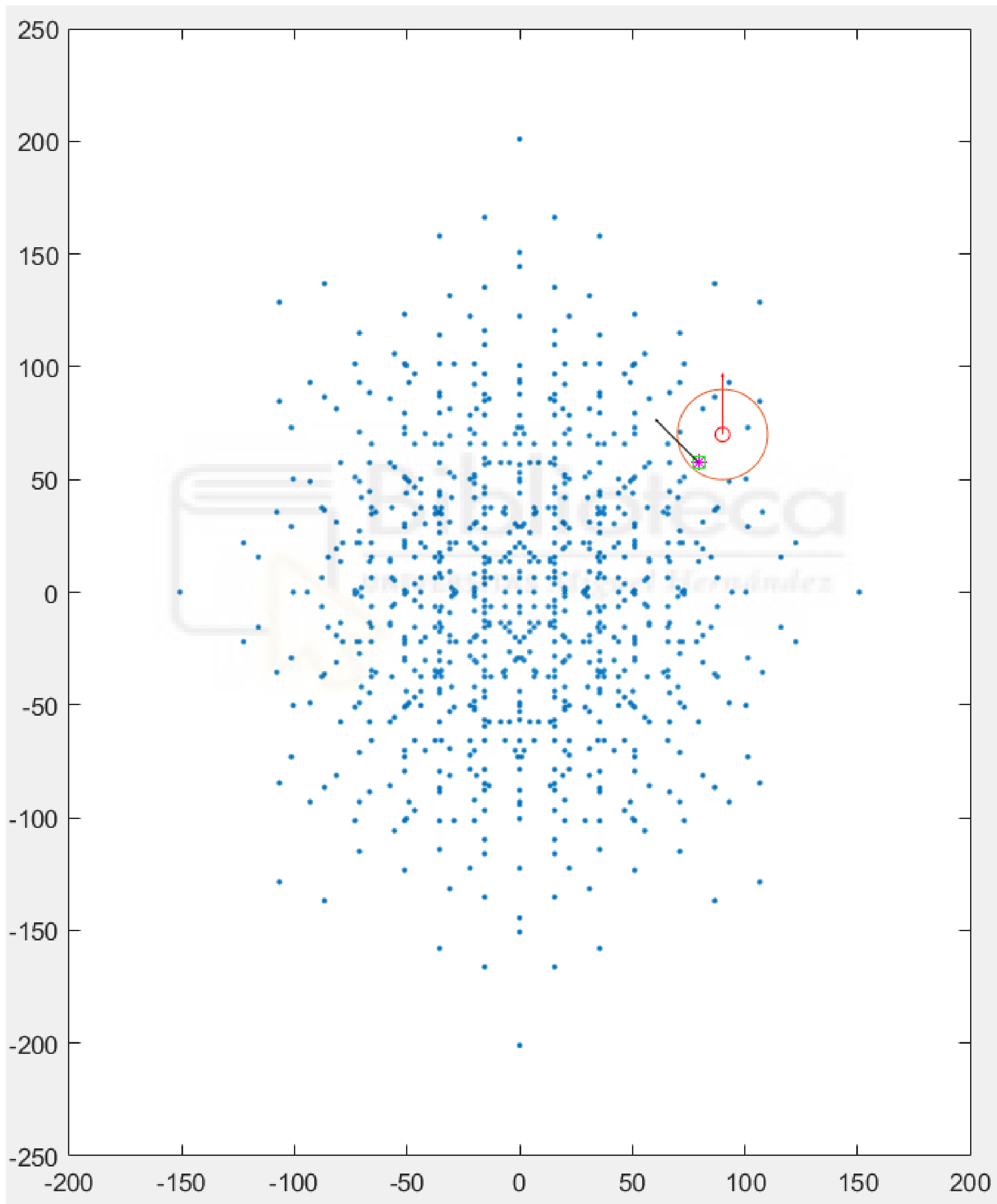


Figura 6: Solución obtenida mediante el método de selección de punto óptimo.

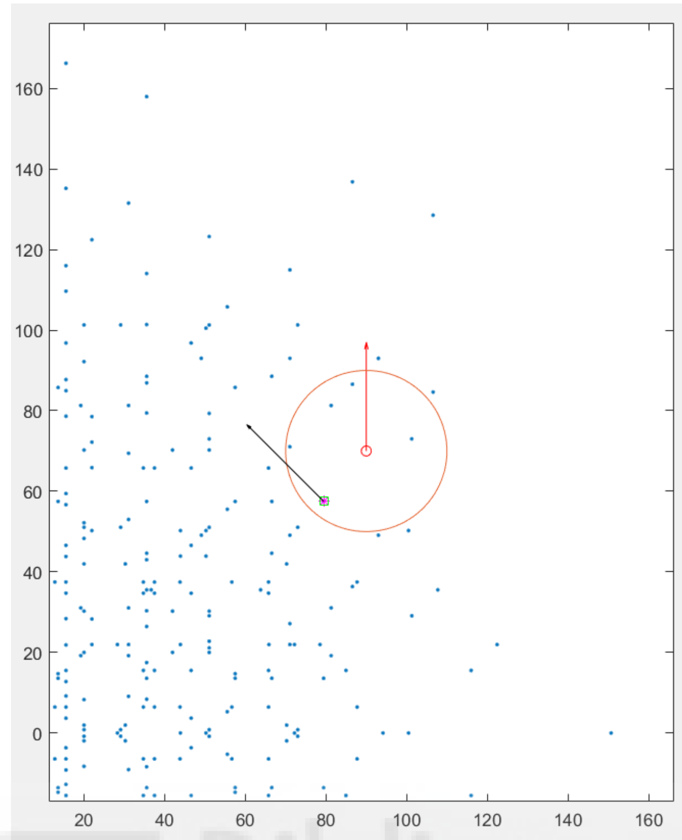


Figura 7: Ampliación de la Figura 6.

Como se puede apreciar, la solución encontrada es correcta, puesto que el punto encontrado se encuentra dentro de la circunferencia admisible y la diferencia de orientaciones es de π/4, por lo que cumple con los parámetros establecidos. La solución obtenida se encuentra en el punto (79.4360, 57.4813), por lo tanto, el error en posición es de 16.3804 mm, y como ya se ha dicho el error en orientación es de π/4, con un coste de 2 ciclos de movimiento del robot.

Una vez más, se ha tomado un punto de la periferia del espacio de trabajo para mayor clarividencia, por lo que el error no es elevado.

3.2. Método de pesos relativos.

La segunda solución que se plantea es la de dar un peso relativo a la orientación con respecto a la posición, y para ello es necesario que puedan ser comparadas ambas de alguna manera. Para ello, lo que se ha planteado hacer es suponer que existe un factor que relaciona milímetros con radianes y darle un valor mayor o menor a ese factor de modo que represente una mayor o menor importancia de la orientación con respecto de la

posición. En esta situación necesitamos calcular el espacio de trabajo para operar sobre la nube de puntos, además del *script* de **calculo_phi_dif.m**. Dicho *script* y el principal de esta solución, que se llama **computacion_distancia_w.m**, así como los que calculan el espacio de trabajo se encuentran en el Anexo 2.

```
7   -  for f=1:size(nube,1)
8   -      Pos_actual = nube(f,1:2);
9   -      resultado = nube(f,:);
10  -      phi_dif = calculo_phi_dif(phi_ref,resultado);
11  -      eq = norm(P_ref-Pos_actual) + w*abs(phi_dif);
12  -      registro_eq = [registro_eq; eq];
13  -  end
14
15     % Se busca el mínimo
16  -  [min_eq,ind_eq] = min(registro_eq);
```

El grueso de esta solución se puede ver en el código mostrado. La ecuación que permite representar el peso relativo entre la posición y la orientación es la que se encuentra en la línea 11. En dicha ecuación, *P_ref* y *Pos_actual* son las posiciones de referencia y actual respectivamente, y al hacer el módulo de ambas se está calculando la distancia entre dichas posiciones; *phi_dif* es la diferencia entre las orientaciones actual y de referencia, cuyo valor siempre estará en el intervalo $[-\pi,\pi]$, para lo que se usará **calculo_phi_dif.m** como se ha mencionado anteriormente. Es el factor w el que representa la importancia de la orientación con respecto a la posición. Cuando se ha calculado el valor de la ecuación para todos los puntos (líneas 7-13), se busca el mínimo (línea 16), que es el que representa la opción más óptima para esta solución.

Las siguientes imágenes muestran los resultados obtenidos de dar a w los valores de 0.1, 10, 1000 y 100000. La referencia (posición y orientación deseadas) se ha predefinido en el punto (90,70) y con orientación pi/2, y se encuentra representada con un círculo rojo, mientras que la solución encontrada se representa como un asterisco magenta.

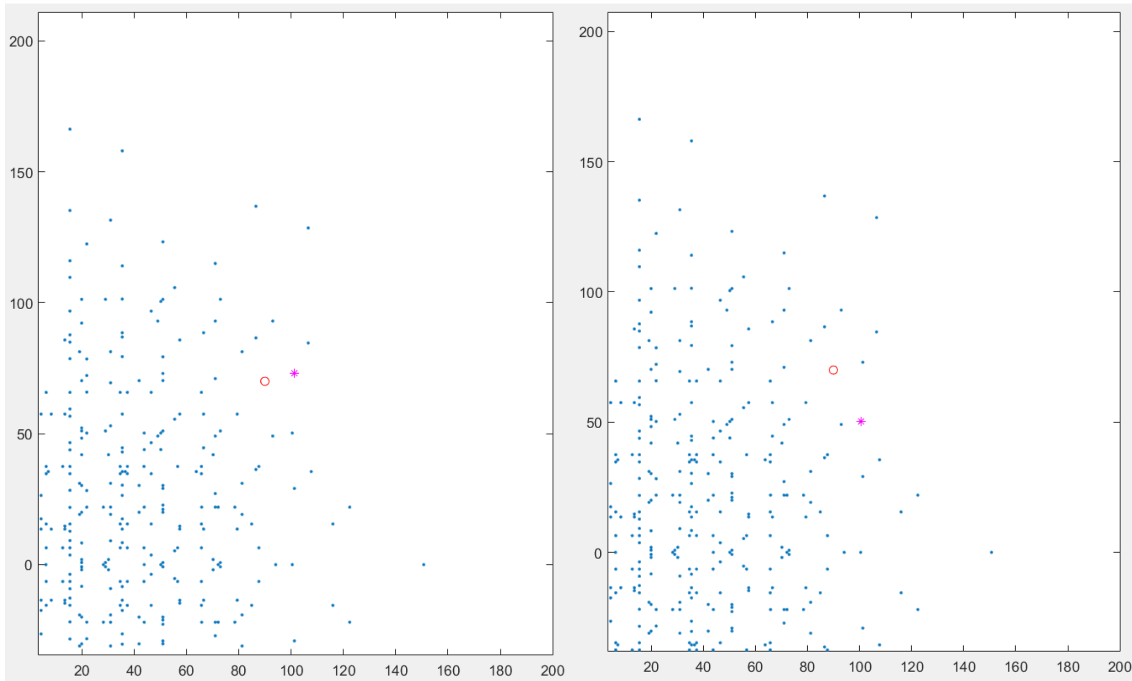


Figura 8: Solución para w = 0.1. Figura 9: Solución para w = 10.

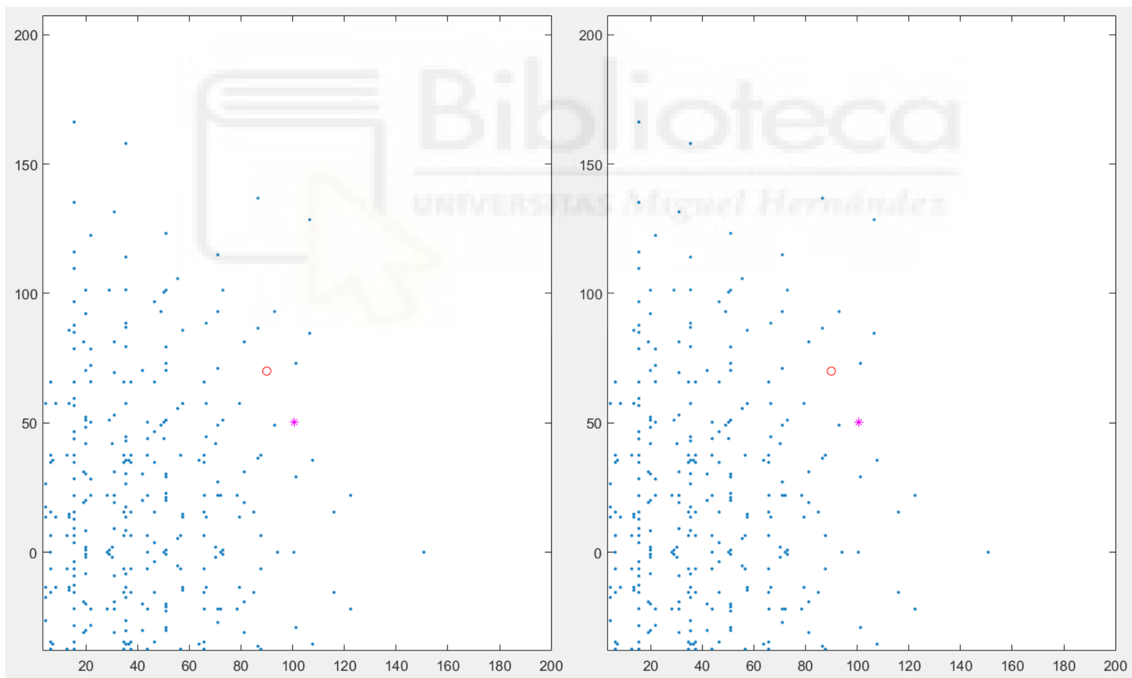


Figura 10: Solución para w = 1000. *Figura 11: Solución para w = 100000.*

Biblioteca UNIVERSITAS Miguel Hernández

Para todos estos casos, se ha aplicado un coste de 2 ciclos de movimiento del robot. En cuanto a los errores de posición y orientación, el valor de $w = 0.1$ tiene un error en posición de 11.6860 mm y de π rad en orientación, y el resto de valores aplicados de w dan un error de 22.3672 mm en posición y 0 rad en orientación. Se debe tener en cuenta que se ha elegido un punto de la periferia del espacio de trabajo del robot para mayor claridad.

Como se puede apreciar, esta solución puede llegar a ser buena para un sistema que no requiera alta precisión, puesto que la diferencia en posición no es demasiado elevada teniendo en cuenta los pesos que se le han dado a la orientación, pero existe la posibilidad de que este algoritmo pueda dar soluciones de posición muy desfavorables sin poder controlarlo si no se aplica un criterio de ponderación diferente.

Es por esto por lo que se ha planteado este problema de dos formas diferentes, atendiendo a dos criterios diferentes de resolución para poder obtener resultados diferentes en función de la aplicación a la que se vaya a someter el robot, puesto que la computación de esta solución es mucho más rápida, y puede que este sea un requisito fundamental en ciertas aplicaciones.

3.3. Comparativa de métodos.

A continuación, se realizará una comparativa entre las dos soluciones planteadas a este problema, expuestas en las secciones 3.1 y 3.2 de este capítulo. Para ello, se va a profundizar un poco en cuáles son los puntos positivos y negativos de cada solución.

La primera solución presentada tiene como principal ventaja que permite controlar el error en posición de manera muy sencilla, puesto que, por definición, nunca será mayor al radio de la circunferencia de puntos admisibles, y dicho valor es parametrizable, por lo que se puede modificar de forma muy sencilla. Además, permite regular la precisión que se quiere tener en cuanto al error de orientación, puesto que se puede implantar manualmente su valor máximo. En cambio, su principal inconveniente es que, en caso de no encontrar un punto válido dentro de la circunferencia admisible, la solución que encuentra puede tener un error en orientación demasiado desfavorable. Adicionalmente, cabe mencionar que este código es más lento puesto que requiere de mayor nivel de computación que su competidor.

Por otra parte, la segunda solución presenta como principales ventajas un menor nivel de computación puesto que es un código más rápido, y además no presenta tanta complejidad de parametrización, puesto que solamente requiere determinar cuál va a ser la importancia de la orientación en el cálculo, cosa que depende exclusivamente de la aplicación, por lo que no presenta un problema. Por el contrario, esta opción presenta un inconveniente muy grande, que consiste en no ser capaz de controlar su error de posición,

y esto puede hacer que en ciertas situaciones se dé el caso de que su error de posición es muy desfavorable cuando se le dé mucha importancia a la orientación que prácticamente obligue al algoritmo a encontrar un punto que cumpla exactamente con la orientación de referencia y este no sea capaz de encontrarlo.

En definitiva, a no ser que sea estrictamente necesario que el algoritmo sea lo más rápido posible, la mejor opción para este problema es la primera solución propuesta, puesto que su principal inconveniente, que es el caso mencionado en que el error en orientación es muy desfavorable, se puede enmendar con cierta simplicidad aumentando el radio de la circunferencia de puntos admisibles. Es más, en caso de que sea imprescindible que el error de orientación sea nulo, que sería equivalente a darle un peso relativo muy elevado a la orientación en la segunda opción, se podría conseguir muy fácilmente determinando que la diferencia de orientaciones sea 0 obligatoriamente; para ello debería darse al radio un valor bastante grande, para que así el algoritmo se limite a encontrar el punto más cercano a la referencia que tenga la orientación buscada, siendo así ambos algoritmos equivalentes en esta situación.

Hablando puramente de números, los errores de posición y orientación de la primera opción son 16.3804 mm y $\pi/4$ respectivamente, mientras que la segunda opción presenta unos errores de posición y orientación de 11.6860 mm y π respectivamente para w=0.1 y de 22.3672 mm y 0 rad respectivamente para el resto de valores de w. Se ha de tener en cuenta que estos números no representan un caso general, puesto que solamente representan un ejemplo, pudiendo cambiar totalmente en otros casos, pero a pesar de ello, analizando los datos numéricos se aprecia que el resultado más conciso sigue siendo el de la primera opción, puesto que es el resultado con errores más equilibrados y controlados.

Capítulo 4. Alcance de posición deseada mediante solapamiento de espacios de trabajo.

El tercer problema que se plantea es que el robot pueda alcanzar una solución en posición válida para cualquier punto del espacio 2D (para este experimento no se tendrá en cuenta la orientación). A priori, este problema parece fácil de solucionar, puesto que bastaría con aumentar el número de ciclos de trabajo del robot y así este podría llegar más lejos, pudiendo así alcanzar cualquier punto que se le pida. No obstante, como ya se explicó al exponer el primer experimento, imponer un número mayor de N=3 ciclos de trabajo supone una capacidad de computación demasiado elevada (calculando más de 17 millones de puntos para N=4 y aumentando de forma exponencial conforme aumenta N).

Es por esto por lo que se ha planteado otra forma para que el robot pueda llegar a cualquier punto del espacio 2D, y esta consiste, en pocas palabras, en solapar espacios de trabajo de N=2 o N=3 ciclos hasta que el robot consiga encontrar una solución válida. Antes de profundizar más en ello, se van a comentar los *scripts* que se van a utilizar en esta solución, así como cuáles de ellos serán modificados.

Los *scripts* que se van a utilizar en la resolución de este problema se encuentran en el Anexo 3, y son únicamente **ws_binario_mp.m**, **ciclo_completo.m** y **encontrar_punto.m**. El único de ellos que se va a encontrar intacto es el segundo, puesto que de este solamente se necesita el cálculo del espacio de trabajo para las condiciones iniciales que se le impongan y el cálculo de las secuencias para alcanzar cada punto. Por otra parte, **encontrar_punto.m** va a guardar todas las secuencias en la variable *registro_seq*, y además va a recortarse, puesto que, ahora, calcular el punto a partir del algoritmo de la secuencia es redundante, ya que se obtiene el mismo resultado que en el algoritmo de distancia mínima por lo que uno de ellos no es necesario, y como el primero requiere que antes se calcule el segundo puesto que necesita la variable *indice_min* para encontrar la secuencia, se ha decidido prescindir de él. Por último, es en **ws_binario_mp.m** donde se van a realizar las modificaciones principales para calcular esta solución, y dichas modificaciones se comentarán a continuación.

```
25 - P0 = [0,0];
26 - phi0 = 0;
27
28 - while 1
```

```
29 -      if norm(P_ref-P0) < 10
30 -          break
31 -      end
32        % Script que calcula el espacio de trabajo
33 -      ciclo_completo(P0,phi0,T,1,N)
34        % Script que encuentra el punto deseado
35 -      encontrar_punto
36        % Actualización de condiciones iniciales
37 -      P0 = P_mas_cercano;
38 -      phi0 = nube(indice_min,3);
39 -  end
```

Una vez que se ha calculado la celda *T* a partir de las variables *phi* e *y* (tal y como en el resto de los experimentos, puesto que se trata de parte del código que pertenece al punto de partida de este trabajo) en otras soluciones simplemente se llamaba a las funciones correspondientes una vez, puesto que nos encontrábamos en situaciones donde se tomaban puntos dentro del espacio de trabajo. Sin embargo, en este caso es necesario llamar a las funciones más de una vez, ya que como se ha dicho anteriormente, el principio básico de esta solución es el de superponer espacios de trabajo.

Lo primero que se debe hacer es definir las condiciones iniciales del robot, que por defecto se pondrán en (0,0) con orientación 0 rad (líneas 25-26); esto se hace porque estas condiciones van a cambiar para cada vez que se calcule el espacio de trabajo, por lo que antes de definir el bucle que haga al robot recorrer el espacio es necesario definir las condiciones iniciales.

A continuación, se define el bucle. Este se va a interrumpir cuando la posición encontrada se encuentre a una distancia menor a la que se defina en la línea 29. Se debe tener cuidado con cuánta precisión se le quiere dar al robot, puesto que si se le da un valor muy bajo de distancia puede pasar que el algoritmo no encuentre una solución y la ejecución nunca termine. Por defecto se ha establecido que una distancia de 10 mm máxima es una solución válida, y es un valor que se considera preciso y con el que el robot se maneja bastante bien.

Una vez establecida la condición de interrupción del bucle, simplemente queda hacer las llamadas necesarias. En primer lugar, se llama a la función **ciclo_completo.m** (línea 33), a la que, aparte de los parámetros iniciales usuales, se le debe mandar las condiciones iniciales del robot en cada momento, que son o bien las iniciales absolutas

en caso de ser la primera iteración o bien las finales del ciclo anterior en caso de no serlo. Posteriormente, se llama a **encontrar_punto.m** (línea 35) para encontrar el punto más cercano al objetivo para poder mandárselo al siguiente ciclo como condición inicial en caso de no ser la última iteración, y en caso de serlo, para encontrar el punto buscado. Finalmente, solo queda actualizar las variables *phi0* y *P0* (líneas 37-38) para poder hacer la siguiente iteración.

A continuación, se mostrarán unas imágenes que muestran el resultado de este experimento. A *N* se le ha dado un valor de 2 para que la nube de puntos pueda apreciarse claramente. El punto de referencia se ha colocado en (500,800) y se ha representado mediante un círculo rojo, y la solución encontrada se ha representado con un asterisco magenta.

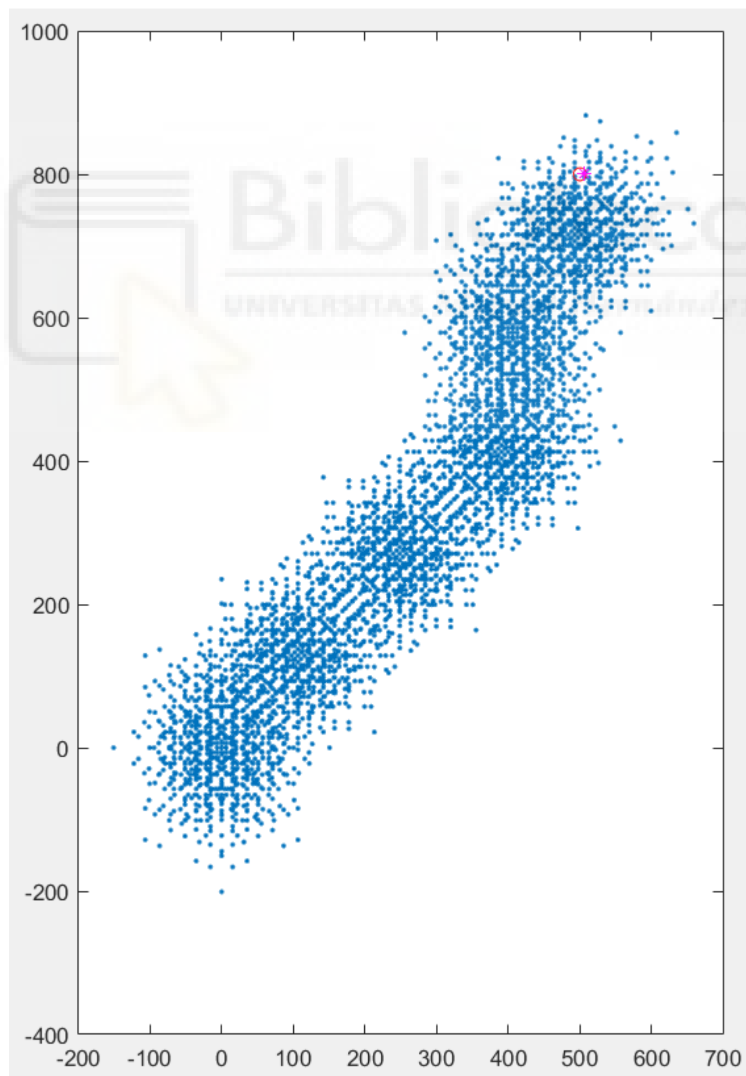


Figura 12: Espacios de trabajo solapados hasta alcanzar una solución lejana, partiendo del origen de coordenadas.

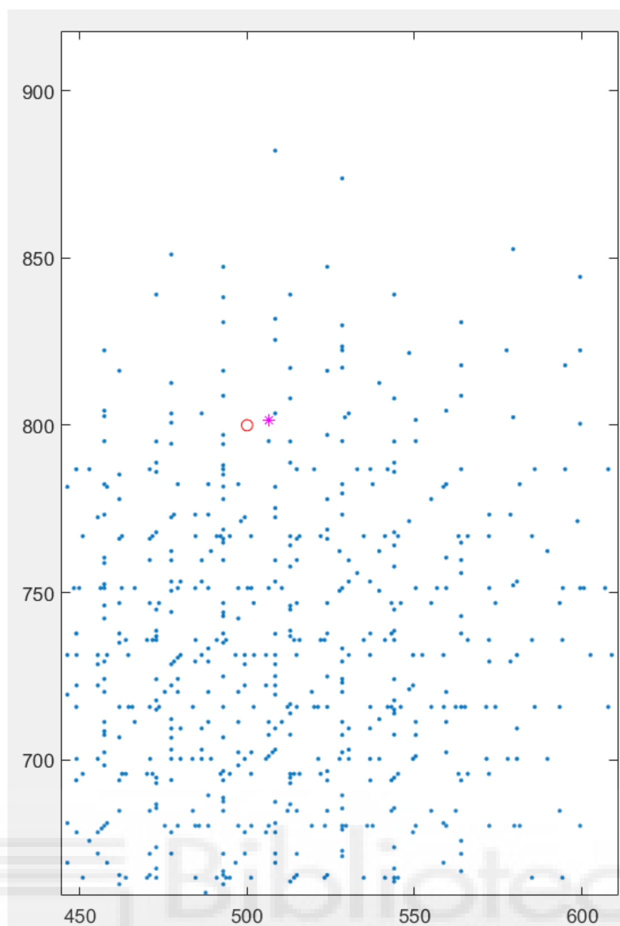


Figura 13. Ampliación de la Figura 12.

La Figura 13 muestra de forma visual la precisión del algoritmo, pero la que realmente representa el principio básico de este experimento es la Figura 12, donde se ve el recorrido del espacio de trabajo del robot en el espacio 2D.

En cuanto al error de este algoritmo, es fácil saber que numéricamente tendrá un valor menor a 10 mm, puesto que se ha puesto como criterio, lo que hace que sea así para cualquier punto que se le imponga. En concreto, este caso ha dado una solución con un error de 6.6558 mm en posición; el error en orientación es irrelevante, puesto que no se ha tenido en cuenta para este experimento.

El factor que empieza a tomar importancia para los experimentos que requieren de más de una ejecución del espacio de trabajo es el coste de ciclos de trabajo que el robot necesita para alcanzar la solución óptima, puesto que ya no es solamente el número N que se le haya impuesto al principio, sino que hay que multiplicarlo por el número de veces que se haya ejecutado el algoritmo. En este caso concreto, el algoritmo se ha ejecutado 6

veces, y N se ha definido como 2 ciclos, por lo que el robot ha realizado 12 ciclos de trabajo para alcanzar esta posición.

Realizar una comparación entre el resultado obtenido a través del método empleado y el que se obtendría si simplemente se aumentara el valor de N hasta que el robot alcanzara la referencia es prácticamente imposible debido a la falta de recursos de computación de los que carecen la mayoría de los ordenadores. Para poner en contexto este argumento, se le ha pedido a Matlab que guarde espacio en memoria para almacenar los puntos que se requieren para representar un espacio de trabajo de 5 y 6 ciclos, y este ha sido el resultado: para N=5 ciclos, el ordenador requeriría de 24.4 GB de memoria RAM para poder almacenar los puntos del espacio de trabajo, que consisten en dos columnas para las coordenadas X e Y y una para la orientación, cosa que a pesar de no ser imposible, ya es poco común que un ordenador común tenga esta capacidad de memoria RAM, pero si calculamos este valor para N=6 ciclos, este asciende a 1.5238 TB, un valor que ya resulta excesivo teniendo en cuenta que el módulo de memoria RAM más grande en el mercado es de 128 GB DDR4, desarrollado por SK Hynix.

Con estos datos se puede ver que elevar el valor de N para hacer cualquier cálculo resulta un método nada adecuado; como se ha demostrado, para N=6 ya resulta imposible de calcular, por lo que teniendo en cuenta que en el ejemplo expuesto en la Figura 12, que no es un caso para nada desproporcionado en cuanto a requerimientos de distancia, el resultado final de N es 12, se demuestra definitivamente que elevar el valor de N es inviable.

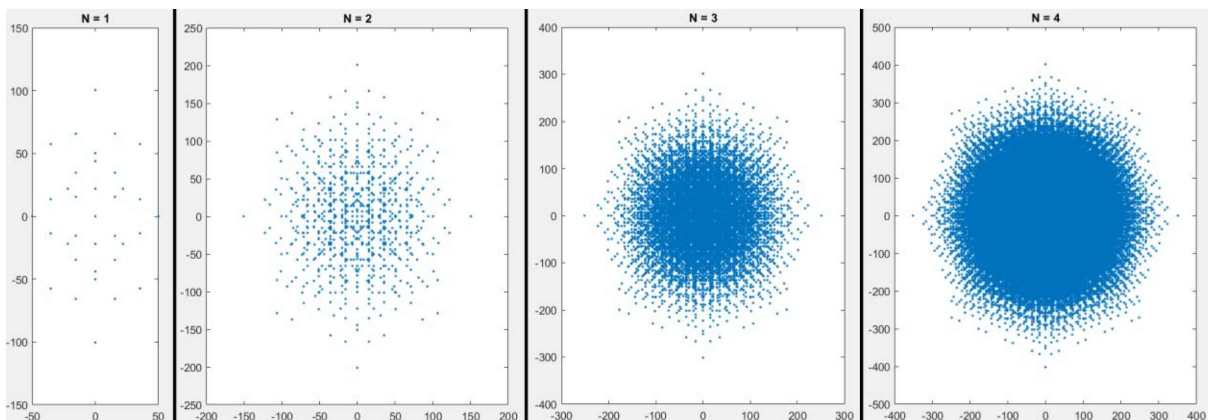


Figura 14: Espacios de trabajo entre los valores N=1 y N=4.

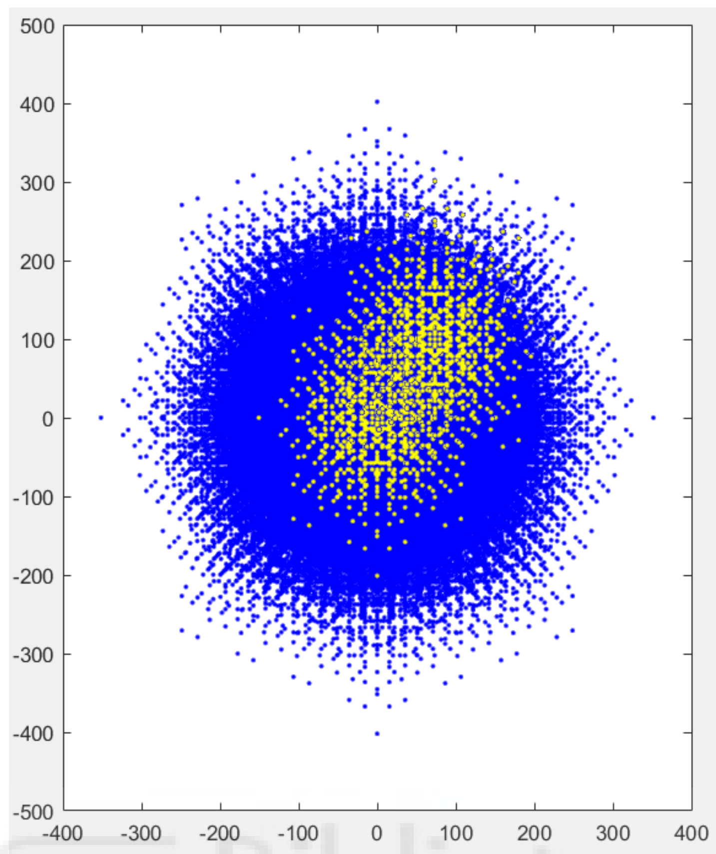


Figura 15: Comparativa del espacio de trabajo de N=4 (azul) con dos espacios solapados de N=2 (amarillo).

La única comparativa con precisión que se puede hacer se muestra en la Figura 15, que consiste en la comparación de un movimiento para *N*=4 con dos movimientos para *N*=2, la cual resulta ser bastante similar, ya que aparentemente parece que en dos movimientos con *N*=2 se puede llegar casi al mismo resultado que en un movimiento con *N*=4, es decir, que los puntos amarillos alcanzan la parte de la zona azul con más puntos; sin embargo, esta comparación resulta no ser del todo aclaratoria, puesto que a pesar de no saberse con exactitud si en casos de *N* mayores a este el método de solapamiento de espacios de trabajo perderá rango efectivo, se estima que sí sucederá debido a que el robot no viaja en línea recta, sino que hace pequeños cambios de dirección, tal y como se ve en la Figura 12.

Debido a todo lo expuesto anteriormente, se concluye que no es posible hacer una comparativa precisa entre ambos métodos (solapamiento de espacios de trabajo y cálculo de un único espacio de trabajo con muchos ciclos). Para poder hacerse, se debería crear un algoritmo que fuera capaz de comparar ambos métodos de forma precisa y con un criterio correcto, cosa que queda fuera del alcance de este Trabajo Fin de Grado.

Capítulo 5. Alcance de posición y orientación deseadas mediante solapamiento de espacios de trabajo.

Este experimento pretende resolver por completo la planificación de trayectorias de este robot en espacios libres de obstáculos. Este es el experimento que va a dar el resultado considerado como el mejor de todos los métodos analizados para la resolución de este Trabajo Fin de Grado, puesto que se va a buscar que llegue lo más cerca posible a cualquier punto del espacio 2D libre de obstáculos con cualquier orientación deseada (siempre que esta sea un múltiplo de 45º). Para esto, en este punto del desarrollo del algoritmo, se debe decidir qué se va a tomar como más importante, precisión o rapidez.

La decisión que se ha tomado es la de calcular la opción más precisa posible, puesto que se considera que este robot va a tener más aplicaciones que requieran precisión antes que rapidez, y, además, el robot en sí ya se mueve lentamente, por lo que no va a desarrollar la tarea de forma rápida igualmente.

Una vez se ha decidido que el objetivo va a ser la mayor precisión posible, se ha planteado la siguiente forma de resolución: alcanzar primero el punto más cercano posible, con el algoritmo desarrollado en el experimento anterior, y a partir de ese punto, orientar el robot hasta que alcance la orientación de referencia. Esta última orientación se hará tras tres ciclos de movimiento, tal y como se mostrará posteriormente.

Antes de analizar el código, se aclarará qué *scripts* son necesarios para resolver este experimento, cuáles van a ser iguales al experimento anterior y cuáles se van a ver modificados. Dichos *scripts* se encuentran en el Anexo 4.

El primer *script* que se encuentra en el Anexo es **ws_binario_mp.m**, y este es exactamente igual al del experimento anterior, salvo las últimas líneas, a las que se han añadido unas instrucciones que permiten la correcta ejecución del algoritmo, y que para profundizar en ellas primero se debe explicar el cometido del último script, por lo que se comentarán posteriormente. El segundo es **ciclo_completo.m**, y este sí se mantiene exactamente igual que en el caso anterior. El tercero se trata de **encontrar_punto.m**, y lo único que cambia es que se eliminan las líneas que se encargaban de ejecutar y configurar los *plots*, puesto que todo esto se hará en el nuevo *script*. El cuarto es **calculo_phi_dif.m**, y es el mismo de todos los experimentos donde se ha usado. Finalmente, tenemos el *script*

nuevo de este experimento, que se llama **ori_final.m**, cuyo cometido principal es el de orientar el robot, el cual partirá desde el punto calculado como más cercano en posición con orientación cualquiera y se colocará con la orientación que se le haya pedido, y además realizará una pequeña corrección en posición si es posible. Complementariamente, está el *script* que se encarga de hacer el dibujo, que en este caso se ha hecho aparte por razones que se explicarán posteriormente.

Ahora que ya se ha explicado brevemente cuál es el propósito de este último *script*, ya se pueden mostrar los cambios en **ws_binario_mp.m** para que se calcule correctamente la solución.

```
42  -  phi_pi=0;
43
44  -  if phi_ref == pi || phi_ref == -pi
45  -      if phi0+phi_ref < 0.001
46  -          phi_pi = 1;
47  -      else
48  -          phi_pi = 0;
49  -      end
50  -  end
51
52  -  if phi0 ~= phi_ref && phi_pi == 0
53  -      flag = 1;
54  -      ori_final
55  -      dibujo
56  -  elseif phi0 == phi_ref || phi_pi == 1
57  -      flag = 0;
58  -      dibujo
59  -  end
```

Como ya se ha dicho en otras ocasiones, las orientaciones almacenadas en la variable *nube* siempre son valores entre $[-\pi, \pi]$, y esto hace que haya algunos elementos que tengan almacenado el valor $-\pi$ y otros que tengan el valor π, pero en la práctica son el mismo valor. Con el fin de identificar cuándo el ángulo de referencia y el del robot al final de la búsqueda toman estos valores, sea cual sea el signo de cada uno de ellos, se ha implementado un pequeño código que se encuentra entre las líneas 42 y 50 y que se encarga precisamente de eso, de devolver una variable llamada *phi_pi* que indique cuándo *phi_ref* ha tomado el valor π y *phi0* ha tomado el valor $-\pi$ o viceversa.

Lo que se pretende con este cambio en **ws_binario_mp.m** es que, si el robot ha llegado a la posición final y coincide con que esa orientación es la deseada, se ahorre el

ejecutar **ori_final.m** puesto que lo único que se conseguiría sería una pequeña corrección en posición, cosa que ya no interesa tanto puesto que le supondría hacer tres ciclos de movimiento más para solamente moverse unos pocos milímetros en el mejor caso. Sin embargo, para el correcto funcionamiento de este código, era necesario identificar el caso mencionado en el párrafo anterior, puesto que, si no se tiene en cuenta, al imponer la condición de que solo se ejecute **ori_final.m** cuando el ángulo final del robot y el de referencia no sean el mismo (línea 52), este estaría ejecutándose cuando dichos ángulos valgan -π y π indistintamente, siendo el mismo valor en la práctica. En caso de que no deba ejecutarse **ori_final.m**, simplemente realizan los pertinentes dibujos de las soluciones (líneas 56-58).

Ahora es momento de profundizar en **ori_final.m**, que es al fin y al cabo el *script* principal de este experimento. Para ello, se debe mostrar y explicar el contenido de la siguiente imagen:

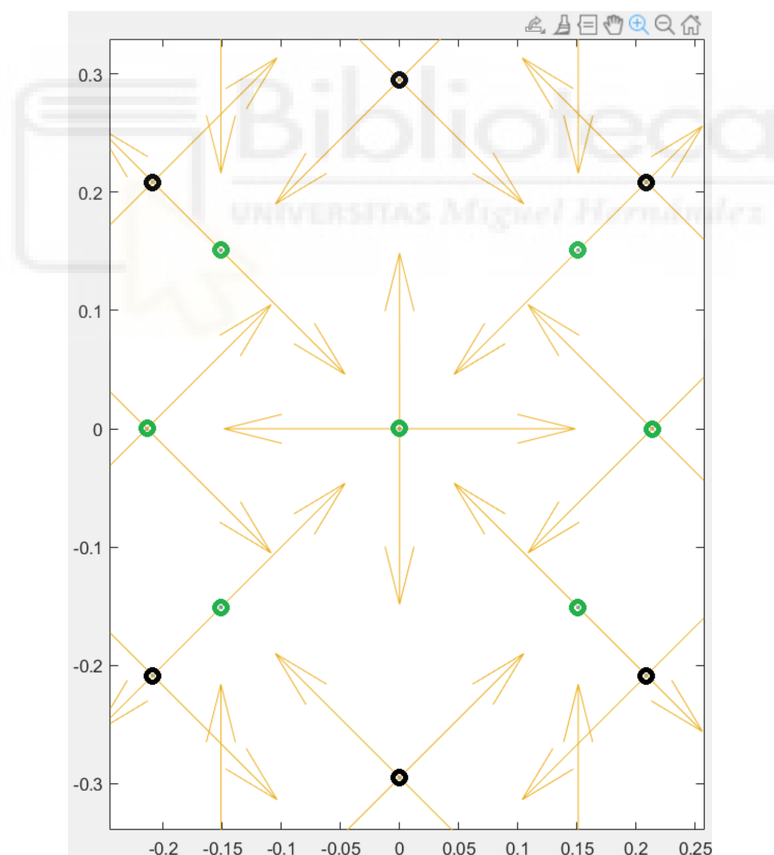


Figura 16: Puntos del espacio de trabajo para N=3 cercanos al origen.

La imagen representa los puntos del espacio de trabajo del robot con *N*=3 más cercanos al origen. Lo que se pretende es tomar una serie de estos puntos que entre todos

abarquen todas las posibles orientaciones del robot con el fin de que este pueda, tomando como origen de coordenadas el punto más cercano en el algoritmo anterior (desde el cual se realizarán los 3 ciclos de orientación final), moverse a uno de esos puntos tomados que tenga la orientación deseada, y así orientarse como se le haya pedido. Se han resaltado todos los puntos cercanos al origen, y se ha decidido que los que se tomarán serán únicamente los marcados en verde puesto que se encuentran bastante cercanos a (0,0) y permiten colocar al robot en todas las orientaciones posibles.

En total son 7 puntos, pero como cada punto tiene distintas orientaciones posibles, en realidad se trata de 20 combinaciones diferentes entre las que el algoritmo tiene que decidir cuál es la más óptima y posteriormente mover el robot a la misma. Para esto último, es necesario saber la secuencia de movimientos del robot, y la forma de obtención de esta ha sido utilizar el código empleado en el experimento 2, primera parte, el cual se ha ejecutado 20 veces, con N=3, imponiendo como referencia las diferentes combinaciones de posiciones y orientaciones posibles. Finalmente, se ha decidido crear un objeto *celda* o *cell* que almacene en la primera columna de elementos la posición, en la segunda la orientación y en la tercera la secuencia, tal y como se muestra a continuación:

```
 6 - puntos_seq = {  [0 0], 0, [1 -1 -1 1 -1 -1];
 7 -                 [0 0], pi/2, [1 7 8 1 2 7];
 8 -                 [0 0], pi, [1 1 3 1 1 7];
 9 -                 [0 0], -pi/2, [1 3 4 1 6 3];
10 -                 [0.151 0.151], pi/4, [5 1 5 3 2 7];
11 -                 [0.151 0.151], -3*pi/4, [5 1 5 3 2 3];
12 -                 [0.151 -0.151], 3*pi/4, [1 1 5 3 8 3];
13 -                 [0.151 -0.151], -pi/4, [1 1 5 3 8 7];
14 -                 [-0.151 0.151], 3*pi/4, [5 5 1 3 6 3];
15 -                 [-0.151 0.151], -pi/4, [5 5 1 3 6 7];
16 -                 [-0.151 -0.151], pi/4, [1 5 1 3 4 7];
17 -                 [-0.151 -0.151], -3*pi/4, [1 5 1 3 4 3];
18 -                 [0.2135 0], pi/4, [3 8 7 5 1 5];
19 -                 [0.2135 0], 3*pi/4, [3 2 3 5 1 5];
20 -                 [0.2135 0], -pi/4, [3 2 7 5 1 1];
21 -                 [0.2135 0], -3*pi/4, [3 8 3 5 1 1];
22 -                 [-0.2135 0], pi/4, [3 6 7 1 5 1];
23 -                 [-0.2135 0], 3*pi/4, [3 4 3 1 5 1];
24 -                 [-0.2135 0], -pi/4, [3 4 7 1 5 5];
25 -                 [-0.2135 0], -3*pi/4, [3 6 3 1 5 5]};
```

Una vez definida esta celda, se puede proceder al cálculo del punto más óptimo para cada caso, pero antes es necesario realizar el siguiente cálculo:

```
27      % Cálculo de la diferencia de ángulos
28  -   phi_dif = phi_ref-nube(indice_min,3);
29  -   if phi_dif > pi
30  -       phi_dif = phi_dif-2*pi;
31  -   elseif phi_dif < -pi
32  -       phi_dif = phi_dif+2*pi;
33  -   end
```

Se debe tener en consideración que tanto el valor de la orientación del robot antes de orientarse como el de la orientación de referencia vienen dados en función del sistema de referencia global. No obstante, las orientaciones que se almacenan en la celda son relativas al sistema de referencia del robot antes de orientarse, es decir, como si el robot partiese de una orientación nula antes de hacer los 3 ciclos finales, por lo que realmente la orientación que se debe lograr con los tres últimos ciclos que orientan el robot es la diferencia entre las dos orientaciones absolutas. Sin embargo, en este caso no sirve usar la función **calculo_phi_dif.m**, puesto que no funciona correctamente, probablemente debido a que dicha función devuelve la diferencia angular en valor absoluto, y en este caso es imprescindible el signo; por ello es necesario aplicar el código. Se debe calcular la diferencia absoluta entre los ángulos y posteriormente transformarlo en el valor que buscamos entre [-π, π]. Esto se consigue aplicando los cálculos que se realizan entre las líneas 29 y 33.

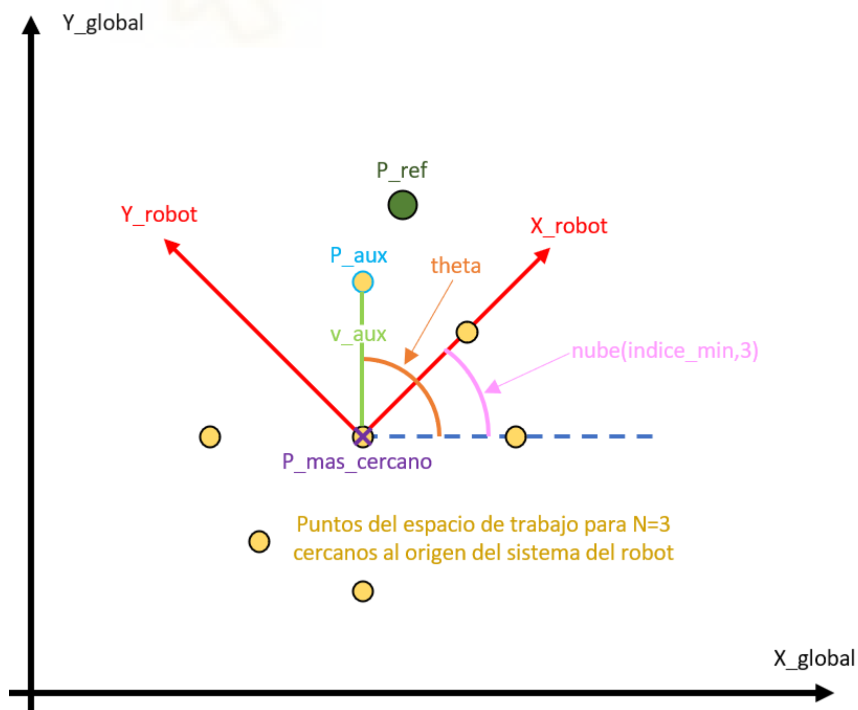


Figura 17: Representación de las variables del código de la orientación final.

En la Figura 17 se muestran diferenciadas por colores las variables que se utilizan en el código que se muestra a continuación, el cual se encarga de encontrar el punto idóneo para orientar el robot en los últimos 3 ciclos. Con esta figura, se pretende dar clarividencia al próximo código.

```
35      % Puntos que coinciden en orientación con la de referencia
36  -   for x = 1:20
37  -       if abs(puntos_seq{x,2}-phi_dif) < 0.001
38  -           v_indices = [v_indices;x];
39  -       end
40  -   end
41
42      % Cálculo de la distancia
43  -   for y = 1:size(v_indices)
44  -       v_aux = puntos_seq{v_indices(y),1};
45  -       theta = atan2(v_aux(2),v_aux(1));
46  -       phi_dif_aux = theta+nube(indice_min,3);
47  -       mod = norm(v_aux);
48
49  -       P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
50                   P_mas_cercano(2)+mod*sin(phi_dif_aux)];
51  -       dist = norm(P_aux-P_ref);
52  -       v_dist = [v_dist;dist];
53  -   end
54
55  -   [min_tmp,ind_tmp] = min(v_dist);
56  -   secuencia_final = puntos_seq{v_indices(ind_tmp),3};
```

La diferencia de orientaciones calculada corresponde con el valor relativo de la orientación de referencia con respecto al sistema de referencia del robot antes de orientarse, que ya es el que corresponde con los valores guardados en la celda *puntos_seq*, por lo que ya se pueden comparar. Precisamente esto es lo que se hace en las líneas 36-40, se recorre la celda y se guardan en un vector *v_indices* todos los índices de los puntos cuya orientación coincide con *phi_dif*.

Cualquiera de los puntos encontrados cumple con la orientación de referencia, pero como podemos escoger entre varias opciones, se va a elegir la que mejor corrección de posición haga. Este objetivo se logra mediante el código que se encuentra entre las líneas 43 y 53, y su funcionamiento se concretará a continuación.

El código se trata de un bucle que cada iteración calcula un punto, por lo que solo se explicará una iteración. El principal problema de este código es que no vale con

simplemente calcular la distancia entre los puntos según sus coordenadas, puesto que se estaría calculando en el sistema de referencia relativo al robot, y este se debe calcular para el sistema global, por lo que las coordenadas se deben traducir al sistema global.

Para ello, en primer lugar, se extrae el punto que se quiere comparar de la celda (línea 44) y se calcula su ángulo con respecto al sistema relativo del robot (línea 45). Posteriormente se calcula el ángulo de ese punto escogido, pero con respecto al sistema de referencia global (línea 46), y el módulo del punto con respecto al robot (línea 47). A continuación, se calculan las coordenadas del punto escogido con respecto al sistema global (línea 49) y se calcula la distancia entre ese punto y el de referencia (línea 51). Finalmente se almacena dicha distancia en un vector que las almacena todas para posteriormente utilizarlo.

Cuando ya se ha calculado el vector de distancias, se debe buscar cuál de todos los puntos tiene la distancia mínima (línea 55) y con su índice obtener la secuencia que lleva al robot a ese punto (línea 56).

Una vez seleccionado el punto y obtenido su secuencia, se calcula su matriz de transformación mediante el algoritmo ya empleado en otros experimentos, el cual se mostrará a continuación, pero no se incidirá en él, puesto que ya se ha explicado en otra ocasión.

```
59 -  for k=1:3
60 -      i = secuencia_final(k);
61 -      j = secuencia_final(k+3);
62 -      if i==-1 || j==-1
63 -          continue
64 -      end
65 -      T_encontrado = T_encontrado*T{i}*inv(T{j});
66 -  end
```

En este caso, puesto que se trata del experimento que resuelve la planificación de trayectorias planteada para este trabajo, sí se profundizará un poco en cómo se dibuja el resultado mediante los comandos *plot* y *quiver*. Todo este proceso viene descrito en el *script* **dibujo.m**, pero para poder dibujar correctamente la solución en caso de que se ejecute **ori_final.m**, primero se debe obtener la posición y orientación finales.

Para ello, lo primero que se debe hacer es el cálculo de la matriz *T0*, que representa la matriz de posición y orientación del sistema de coordenadas del robot con respecto al sistema global, y se calcula de la siguiente manera:

```
68  -  ang = nube(indice_min,3)
69  -  x = nube(indice_min,1)
70  -  y = nube(indice_min,2);
71  -  T0 = [ cos(ang)  -sin(ang)   x;
72  -         sin(ang)   cos(ang)   y;
73  -            0          0       1];
74
75  -  T1 = T0*T_encontrado;
76  -  P_final = T1(1:2,3)';
77  -  phi_final = atan2(T1(2,1),T1(1,1));
```

T_encontrado es la matriz que representa la transformación desde el sistema de referencia del robot antes de orientarse hasta después de haberse orientado, por lo que al multiplicarla por *T0* (línea 75) se obtiene la matriz que contiene los valores de posición y orientación del punto final con respecto al sistema de coordenadas global. Por tanto, con esta matriz llamada *T1* se puede obtener la posición y la orientación del punto final.

Posteriormente, se aplica el método de representación mostrado a continuación, que se encuentra en el *script* **dibujo.m**.

```
1      % Dibujo
2   -  plot(nube(:,1),nube(:,2),'.');
3   -  set(gca,'DataAspectRatio',[1,1,1]);
4   -  hold on
5   -  plot(P_ref(1),P_ref(2),'or');
6   -  quiver(P_ref(1),P_ref(2),cos(phi_ref),sin(phi_ref),'r');
7   -  plot(P_mas_cercano(1),P_mas_cercano(2),'*m');
8   -  quiver(P_mas_cercano(1),P_mas_cercano(2),cos(nube(indice_min,3)),
9   -         sin(nube(indice_min,3)),'m');
10  -  if flag == 1
11  -    plot(P_final(1),P_final(2),'sg');
12  -    quiver(P_final(1),P_final(2),cos(phi_final),sin(phi_final),'g');
13  -  end
```

El *script* **ciclo_completo.m** guarda los puntos de *nube* en la posición marcada por *global_counter*, y nunca se reinicia el valor de este último valor, por lo que la variable *nube* va a seguir almacenando todos los valores de las nubes de puntos consecutivas, así

que simplemente se debe dibujar *nube* con el comando *plot* (línea 2) y se podrán apreciar todas las nubes de puntos superpuestas.

La línea 3 consiste simplemente en una instrucción que fuerza la gráfica del *plot* tenga sus ejes de coordenadas proporcionados, es decir, que un valor de 10 en la coordenada X tenga la misma longitud que un valor de 10 en la coordenada Y.

Finalmente, se dibujan los tres puntos característicos del cálculo y sus orientaciones: la referencia, la situación del robot antes de orientarse y la situación del robot en el punto final (líneas 5-13). El primero se ha representado con un círculo y flecha rojos, el segundo con asterisco y flecha magentas y el tercero con cuadrado y flecha verdes. La condición de *flag* (línea 10) simplemente representa si se ha ejecutado **ori_final.m**, ya que no es posible hacer el tercer conjunto de *plots* si este no se ha ejecutado.

Las siguientes imágenes muestran el resultado de este algoritmo para una referencia de (400,500) en posición y π/4 en orientación.

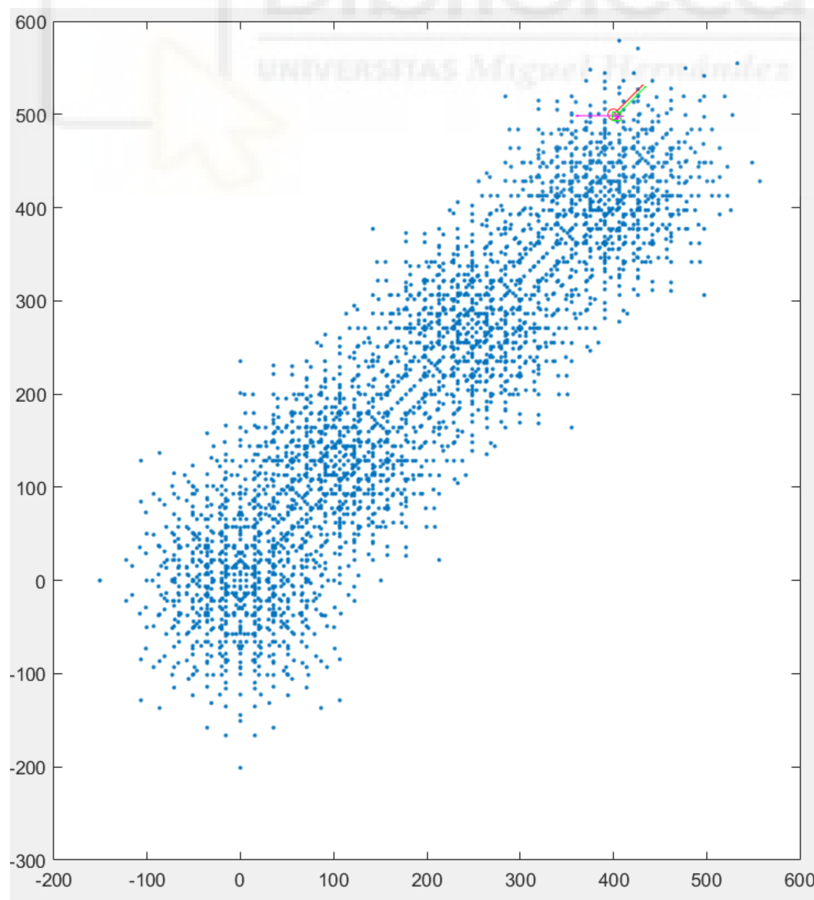


Figura 18: Solución con espacios de trabajo solapados y orientación final.

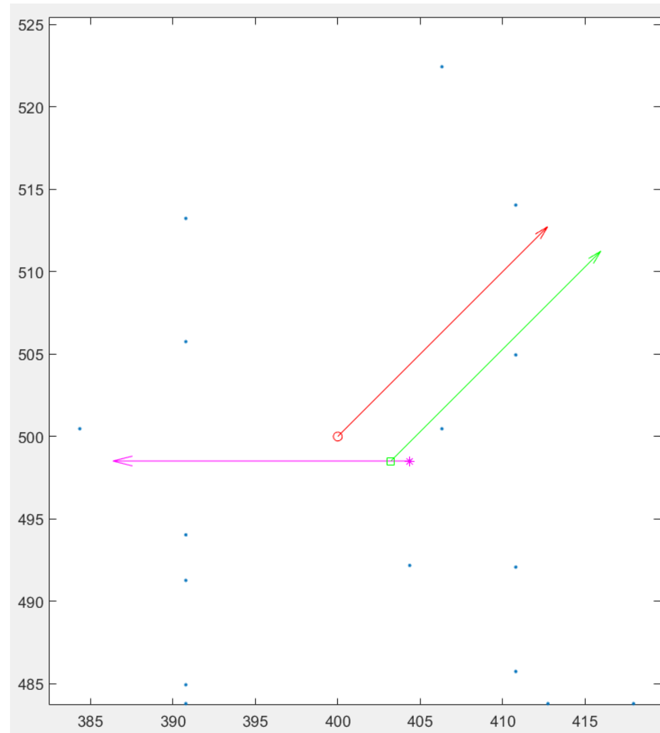


Figura 19: Ampliación de la Figura 18.

En la Figura 19 se puede ver perfectamente la precisión del algoritmo, pero es la Figura 18 la que hace que se pueda ver más claramente el propósito del algoritmo, y además nos permite saber el coste de ciclos de movimiento hace el robot. Concretamente, en este caso el robot hace 8 ciclos de movimiento para acercarse al punto y 3 para orientarse, por lo que en total hace 11 ciclos de movimiento. A pesar de no poder compararse este método con la resolución que se hubiera obtenido elevando el número de ciclos (N), se puede concluir que la velocidad de ejecución del movimiento raramente va a ser un criterio fundamental de diseño, puesto que el robot de por sí es bastante lento, por lo que el coste de ciclos no resulta ser un valor tan relevante.

En cuanto a los errores numéricos en posición y orientación, evidentemente está claro que el error en orientación va a ser nulo en todos los casos, pero el error en posición va a tener un valor en prácticamente todos los casos. Para este resultado en concreto, el error en posición es de 3.5452 mm. Es un error considerado pequeño, puesto que se le ha pedido al robot recorrer una distancia aproximada de 640 mm, lo que supone un error aproximado del 0.55% de la distancia total.

Capítulo 6. Animación del robot.

Hasta ahora, todos los casos resueltos en este trabajo se han mostrado en forma de imágenes que mostraban, en pocas palabras, una serie de nubes de puntos que representaban el espacio o los espacios de trabajo del robot acompañadas de unos puntos y unas flechas que representaban la referencia que se quería alcanzar y el resultado que se había obtenido en cada caso. Sin embargo, se ha decidido hacer un algoritmo que permite realizar una animación del robot moviéndose a través del espacio 2D, puesto que es una forma muy visual de ver cómo se mueve el robot, conocer sus limitaciones y comprobar que todos los experimentos realizados anteriormente son correctos.

Todo el código correspondiente a la animación del robot se encuentra íntegro en el Anexo 5, y consiste en los siguientes *scripts*: **animación_completa.m**, **anim_ciclo_completo.m**, **con_adyacentes.m**, **calculo_secuencia.m**, **anim_adyacentes.m**, **inversa.m** y **directa.m**. Las tareas que todos ellos ejercen para la animación del robot se explicarán a continuación, y se hará desde los *scripts* más "simples" hasta los más "elaborados", es decir, primero se explicarán los que no dependen de otros para funcionar y posteriormente se irán introduciendo los que dependan de los inmediatamente inferiores. Esta estructura jerárquica se ve representada en el siguiente diagrama de flujo:

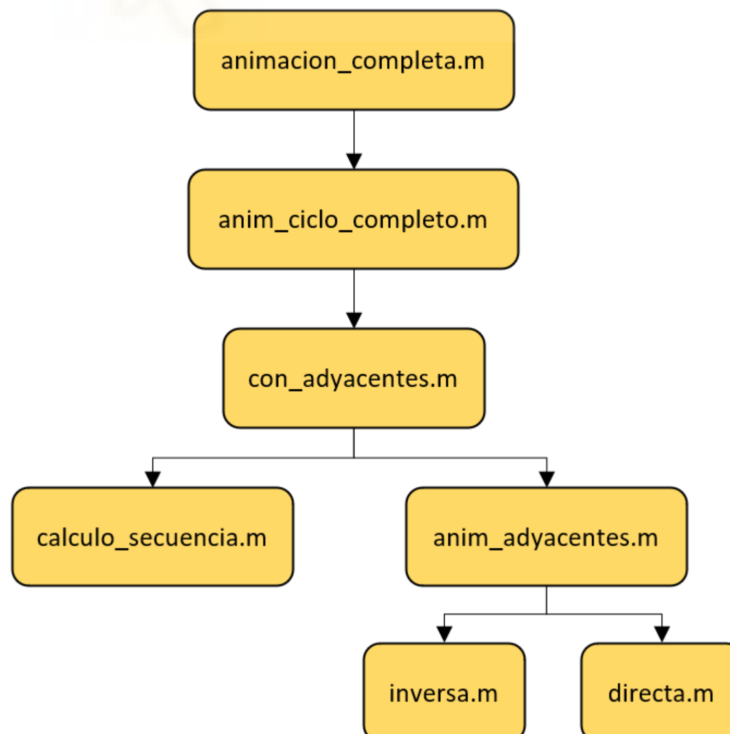


Figura 20: Estructura jerárquica de los scripts de animación.

6.1. Cinemática inversa y directa.

Los primeros que se explicarán serán los de **inversa.m** y **directa.m**, que como sus propios nombres indican, sus funciones principales son las de calcular las cinemáticas inversa y directa respectivamente.

Se empezará por **inversa.m**, el cual consiste en calcular los valores de longitud de los actuadores binarios, que se llamarán *l* para la longitud del actuador apoyado al lado izquierdo del centro del eslabón A y *r* para la longitud del actuador del lado derecho. Los valores de *b* y *p* son constantes para todos los casos puesto que son medidas de diseño del robot. A continuación, se muestra una imagen donde se puede apreciar más claramente qué representan los valores de *b*, *p*, *l*, *r*, *phi* e *y*.

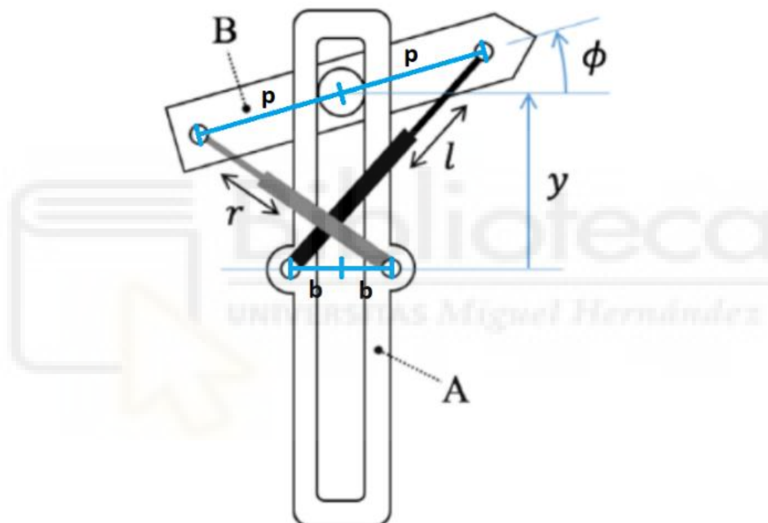


Figura 21. Representación de los parámetros de diseño del robot

```
1  -  function [l,r] = inversa(phi,y)
2
3  -  b = 18.59;
4  -  p = 101.31;
5
6  -  l = sqrt((p*cos(phi)+b)^2+(y+p*sin(phi))^2);
7  -  r = sqrt((-p*cos(phi)-b)^2+(y-p*sin(phi))^2);
8
9  -  end
```

El código de esta función es muy simple. Lo único que se debe hacer es calcular *l* y *r* mediante las ecuaciones que se pueden ver en las líneas 6 y 7, las cuales dependen únicamente de las cuatro variables que se representan en la Figura 21. Las variables *phi*

e *y* dependen del momento en el que se encuentre el robot, y estas son las que hacen que *l* y *r* varíen con el tiempo. Estos valores de phi e y son los que se encontraban en los vectores con su mismo nombre que se han estado utilizando a lo largo de todo el trabajo, a partir de los cuales se ha creado la celda *T* de matrices con las distintas configuraciones posibles del robot.

Por otra parte, el código de la función **directa.m** es más complejo, puesto que por lo general el cálculo de la cinemática directa en robots paralelos es más complicado que la inversa, y este caso no es una excepción.

```
1  - function [phi,y] = directa(l,r,phi_p,y_p)
2
3  - b = 18.59;
4  - p = 101.31;
5
6  - F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
7  -      (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
8
9  - J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
10 -      -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
11
12 - aux = [phi_p;y_p];
13
14 - for iter = 1:10
15 -     aux = aux - inv(J)*F;
16 -     phi_p = aux(1);
17 -     y_p = aux(2);
18 -     J=[-2*p*(b*sin(phi_p)-y_p*cos(phi_p)),2*(y_p+p*sin(phi_p));
19 -        -2*p*(b*sin(phi_p)+y_p*cos(phi_p)),2*(y_p-p*sin(phi_p))];
20 -     F=[(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
21 -         (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
22 - end
23
24 - phi = aux(1);
25 - y = aux(2);
26
27 - end
```

Se ha decidido que la mejor forma de calcular la cinemática directa en este caso es por el método de Newton, que consiste en hacer un bucle que calcule repetidamente la solución siguiente para que en cada iteración del bucle el resultado obtenido sea más próximo al buscado.

$$\begin{bmatrix} phi \\ y \end{bmatrix}_{k+1} = \begin{bmatrix} phi \\ y \end{bmatrix}_{k} + (J)^{-1} \cdot F(phi_k, y_k)$$

La matriz F es una matriz 2x1 que almacena las ecuaciones de l y r utilizadas en la cinemática inversa igualadas a 0, como se puede ver en las líneas 6-7. En cambio, J representa la matriz Jacobiana de F, una matriz 2x2 que almacena las derivadas parciales de los elementos de F con respecto a sus variables *phi* e *y* (líneas 9-10). Como se ha dicho, esta función consiste en hacer un bucle que repita la ecuación anterior un número de veces apropiado para obtener una precisión aceptable (se ha considerado que 10 iteraciones es un número más que suficiente para esta tarea), o hasta que la diferencia entre *phi_y_{k+1}* y *phi_y_{k}* sea despreciable. Además de calcular la ecuación en cada iteración, es muy importante actualizar los valores de F y J para cada repetición, puesto que si no se hace el resultado sería incorrecto. Finalmente, se devuelven los valores de *phi* e *y* finales (líneas 24-25), que son el objetivo principal del cálculo de la cinemática directa.

6.2. Animación entre dos configuraciones adyacentes.

Los *scripts* descritos anteriormente son utilizados en la función **anim_adyacentes.m**, que es el código principal de la animación, pero antes de continuar con esta, es necesario explicar cómo se mueve el robot, así como qué significan los números que se almacenan en las variables de tipo secuencia.

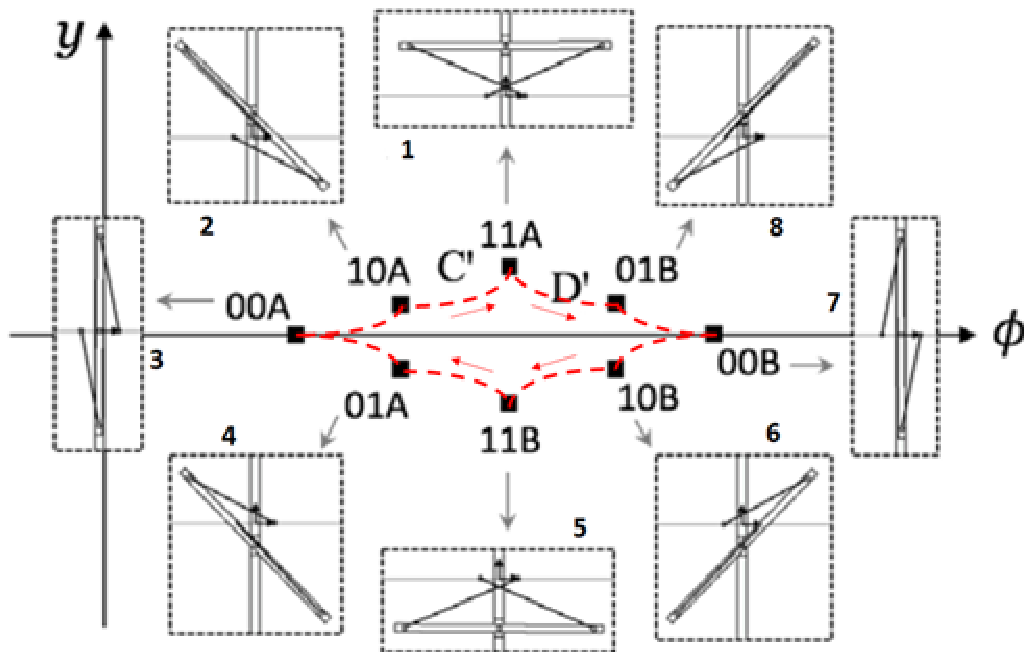


Figura 22: Configuraciones del robot

El robot solo tiene 8 posibles configuraciones mostradas en la Figura 22, a las cuales no puede acceder arbitrariamente. Este solo es capaz de alcanzar una configuración a partir de otra adyacente a ella, por ejemplo, desde la configuración 5, el robot solo se puede mover a las contiguas, la 4 o la 6. Sin embargo, en la variable *registro_seq*, que es la que almacena todas las secuencias desde el inicio hasta alcanzar la posición más cercana, sin orientarse, al igual que en la variable *secuencia_final*, que es la que almacena la secuencia de orientación, se almacenan los índices de las configuraciones que el robot tiene que alcanzar para logra el objetivo, pero no se tiene en cuenta la adyacencia de estos, y para poder animar el robot, por ejemplo entre las configuraciones 3 y 6, este pase contiguamente por 3, 4, 5 y 6. De este cometido se encarga la función **calculo_secuencia.m**, que devuelve la secuencia de configuraciones adyacentes que el robot debe seguir, tanto en sentido ascendente como descendente, dependiendo de cuál sea el camino más corto.

La función **anim_adyacentes.m** solamente funciona cuando sus entradas son dos índices contiguos de la secuencia, puesto que el objetivo prioritario de esta es el de animar simplemente el movimiento entre dos configuraciones adyacentes, por lo que es totalmente necesario que primero se ejecute la función anteriormente mencionada sobre los índices que se reciban de la variable *registro_seq*. Ahora que ya se comprende la importancia de que **anim_adyacentes.m** reciba como entrada dos índices correlativos, se procederá a la explicación tanto de esta función como de la mencionada en el párrafo anterior.

El primer código que se explicará será el de **anim_adyacentes.m**, y debido a que es el código principal de la animación, es bastante extenso, por lo que se desglosará en partes para que su desarrollo sea más claro. La idea principal de este código es ir animando el robot a base de diferenciales del movimiento, es decir, que lo que se pretende es buscar la configuración inicial y la final del movimiento e ir modificando la posición y orientación de los eslabones progresivamente mediante diferenciales de variables, empezando desde la posición inicial hasta alcanzar la final, creando una sucesión de imágenes que simula ser un vídeo del robot moviéndose.

A continuación, se muestra un dibujo en el que se ve la representación esquemática que se va a hacer del robot, representada en los mismos colores con los que se verá en la animación, así se podrá hacer referencia a elementos de esta imagen en los

párrafos posteriores. Se debe destacar que este dibujo no tiene las medidas reales, solamente es orientativo, el robot con las medidas correctas se verá en la animación y en una imagen al final de este apartado.

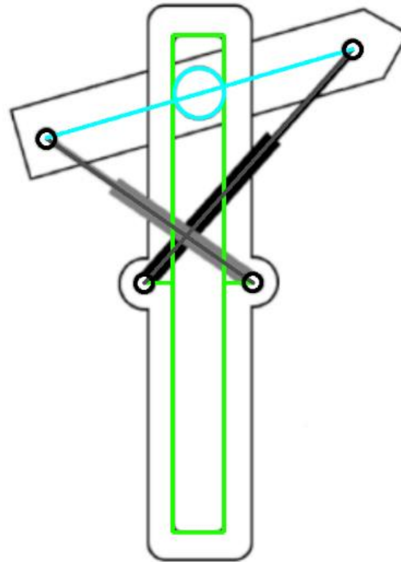


Figura 23: Dibujo esquemático del robot

Biblioteca

Antes de nada, se debe aclarar que las variables de entrada son los índices adyacentes (*índice_inicio* e *indice_fin*), una variable booleana que indica cuál es el cuerpo en movimiento (*movB*), y las variables que indican la posición y orientación absolutas del cuerpo que no está en movimiento. Los vectores *phi_v* e *y_v* son los vectores que almacenan los valores de los posibles *phi* e *y* para cada configuración, ya usados anteriormente y llamados simplemente *phi* e *y*, pero estos han sido renombrados para que se distingan más claramente de otras variables que representan esos mismos parámetros pero sin ser vectores.

```
13  -  y_inicial = y_v(indice_inicio);
14  -  phi_inicial = phi_v(indice_inicio);
15  -  y_final = y_v(indice_fin);
16  -  phi_final = phi_v(indice_fin);
17
18  -  [li,ri] = inversa(phi_inicial,y_inicial);
19  -  [lf,rf] = inversa(phi_final,y_final);
20  -  dl = (lf-li)/(m-1);
21  -  dr = (rf-ri)/(m-1);
22
23  -  l = li;
24  -  r = ri;
25  -  phi = phi_inicial;
```

```
26 - y = y_inicial;
```

Las primeras líneas de este código son bastante simples. En primer lugar, se identifican las configuraciones inicial y final a través de sus valores de *phi* e *y* correspondientes (líneas 13-16). A continuación, se establecen las longitudes iniciales de los actuadores mediante la función **inversa.m** (líneas 18-19) y se calculan los diferenciales de longitud *dl* y *dr* (líneas 20-21); dichos diferenciales representan la cantidad en la que aumenta o disminuye la longitud de los actuadores en cada instante de la simulación (*m* representa el número de diferenciales en los que se divide el recorrido). Únicamente resta dar los valores iniciales a las variables que van a ser actualizadas en próximos pasos del algoritmo (líneas 23-26).

```
28      % pasador de B
29 -    N_pasador = 30;
30 -    radio_pasador = b/2;
31 -    d_ang = 2*pi/(N_pasador-1);
32 -    ang = 0;
33 -    pasador = zeros(N_pasador,2);
34 -    for i=1:N_pasador
35 -        pasador(i,1) = radio_pasador*cos(ang);
36 -        pasador(i,2) = radio_pasador*sin(ang);
37 -        ang = ang + d_ang;
38 -    end
```

Antes de empezar con los cálculos y los *plots*, se han realizado las operaciones adecuadas para crear una matriz llamada *pasador* que almacene en dos columnas las coordenadas x e y de un círculo que va a representar el pasador del robot, correspondiéndose en la Figura 23 con el círculo de color cian. El principio de cálculo de los puntos del círculo es el mismo que se utilizó para dibujar la circunferencia de puntos admisibles que se introdujo en la sección 3.1 del Capítulo 2, por lo que no se incidirá demasiado en ello; solamente decir que, en el otro caso, el círculo se dibujaba en el momento del cálculo y en este caso se deben guardar las coordenadas en la matriz *pasador* puesto que el círculo debe moverse, por lo que hay que dibujarlo al mismo tiempo que el robot.

La siguiente parte del código consiste en un bucle *for* que se encarga de hacer los cálculos necesarios para la animación y de ejecutar la misma, por lo que se trata de un bucle de más de 80 líneas, así que este se va a dividir en secciones, pero se debe tener en cuenta que todo lo que se va a ver a continuación está dentro del mismo bucle.

```
40  -  for i = 1:m
41
42  -     [phi,y] = directa(l,r,phi,y);
43  -     l = l+dl;
44  -     r = r+dr;
45  -     T_BA = [ cos(phi)   -sin(phi)     0;
46  -              sin(phi)    cos(phi)     y;
47  -                 0           0         1];
48
49  -     if movB == 1
50  -         T_A = [ cos(ori_0) -sin(ori_0)   pos_0(1);
51  -                 sin(ori_0) cos(ori_0)    pos_0(2);
52  -                     0          0            1    ];
53  -         T_B = T_A*T_BA;
54  -     else
55  -         T_B = [ cos(ori_0) -sin(ori_0)   pos_0(1);
56  -                 sin(ori_0) cos(ori_0)    pos_0(2);
57  -                     0          0            1    ];
58  -         T_A = T_B*inv(T_BA);
59  -     end
```

Lo primero que se define es el número de iteraciones del bucle (línea 40). El número *m* se ha establecido por defecto en 20, es decir, que desde la configuración inicial a la configuración final (se recuerda que son configuraciones adyacentes) el movimiento se representará con 20 fotogramas.

Entre las líneas 42-59 se muestran los cálculos principales de la animación, que básicamente consisten en calcular las matrices que almacenan las posiciones y orientaciones de los centros de los eslabones. Lo primero que se hace es calcular, mediante la cinemática directa, los valores de *phi* e *y* para los actuales *l* y *r* (línea 42), e inmediatamente después se actualizan los valores de *l* y *r* añadiéndoles sus respectivos diferenciales *dl* y *dr* (líneas 43-44) para la próxima iteración. Una vez conocidos los valores de *phi* e *y* actuales, ya se puede calcular la matriz de transformación *T_BA*, que representa el sistema de coordenadas del eslabón B con respecto al de A (líneas 45-47).

Las líneas que vienen a continuación ya dependen de cuál de los dos eslabones está en movimiento, como se indica en el *if* de la línea 49. Si *movB* es 1, significa que el eslabón fijo es A, por lo que su posición y orientación no van a variar, así que lo que se calcula es *T_B*, la matriz que representa el sistema de coordenadas de B con respecto al global, y esta se obtiene multiplicando la matriz que representa al sistema A en coordenadas globales (*T_A*, calculada en las líneas 50-52), que es constante porque A está

fijo, y la matriz de B con respecto de A (*T_BA*), como se puede ver en la línea 53. Por el contrario, si *movB* es 0, significa que es el eslabón B el que está fijo, por lo que mediante el producto de su matriz (*T_B*, calculada en las líneas 55-57), que es constante, y la inversa de la matriz que relaciona ambos sistemas (*T_BA*) se obtiene *T_A*, que representa al sistema de coordenadas de A con respecto al global.

```
61 -     pos_A = [T_A(1,3) T_A(2,3)];
62 -     ori_A = atan2(T_A(2,1),T_A(2,2));
63 -     pos_B = [T_B(1,3) T_B(2,3)];
64 -     ori_B = atan2(T_B(2,1),T_B(2,2));
65
66       %Puntos de los actuadores
67 -     B1 = T_B*[p;0;1];
68 -     B2 = T_B*[-p;0;1];
69 -     A1 = T_A*[-b;0;1];
70 -     A2 = T_A*[b;0;1];
71
72       % Puntos de la ranura
73 -     altura_pasador = 70;
74 -     a11 = T_A*[-radio_pasador;altura_pasador;1];
75 -     a12 = T_A*[ radio_pasador;altura_pasador;1];
76 -     a21 = T_A*[-radio_pasador;-altura_pasador;1];
77 -     a22 = T_A*[ radio_pasador;-altura_pasador;1];
```

Una vez calculadas las matrices *T_A* y *T_B*, ya sea o bien para el caso en el que se mueve B o bien el caso en el que se mueve A, se pueden obtener todos los puntos necesarios para realizar la simulación. Lo primero que se debe situar es el centro de los eslabones y sus orientaciones (líneas 61-64); estos son los puntos más importantes para poder realizar la animación del robot, a partir de ellos se obtienen el resto de los puntos.

Los actuadores van a ser representados a partir de sus vértices, los cuáles se calculan en las líneas 67-70, multiplicando las matrices de representación del sistema correspondiente a cada eslabón por los puntos (±p,0) para el eslabón B y (±b,0) para el A. Esto hace básicamente que se encuentren los puntos antes mencionados en el sistema de coordenadas del eslabón correspondiente pero representados en el sistema de coordenadas global.

Finalmente, entre las líneas 73-77 se encuentran los cálculos pertinentes para calcular los cuatro vértices de la ranura del pasador, que se encuentra representada en la Figura 23 como un rectángulo verde. Se ha definido aproximadamente la altura del

pasador como 70 mm, puesto que no se dispone de ese dato; el valor que se puede tomar como referencia para realizar una aproximación es el de *y* máxima, que es de ±50.24201358 mm. Para obtener estos puntos, se hace igual que en el caso de los actuadores: se multiplica la matriz, en este caso, *T_A*, por los puntos que se quieren buscar en el sistema de coordenadas de A, que son todas las combinaciones posibles de (±*radio_pasador*; ±*altura_pasador*), siendo radio_pasador el radio de la circunferencia que representa al pasador, definido en la línea 30 mostrado hace dos fragmentos de código.

```
79   -    plot(nube(:,1),nube(:,2),'.y');
80   -    hold on
81
82        % Cuerpo A
83   -    plot(pos_A(1),pos_A(2),'xb')
84   -    axis([-600,600,-600,600])
85   -    set(gca,'DataAspectRatio',[1,1,1])
86   -    quiver(pos_A(1),pos_A(2),esc*cos(ori_A),esc*sin(ori_A),'b');
87   -    plot([a11(1),a12(1)],[a11(2),a12(2)],'g','LineWidth',2);
88   -    plot([a11(1),a21(1)],[a11(2),a21(2)],'g','LineWidth',2);
89   -    plot([a22(1),a12(1)],[a22(2),a12(2)],'g','LineWidth',2);
90   -    plot([a22(1),a21(1)],[a22(2),a21(2)],'g','LineWidth',2);
91   -    plot([ (a11(1)+a21(1))/2,A1(1) ],[ (a11(2)+a21(2))/2,A1(2) ],
92   -           'g','LineWidth',2);
93   -    plot([ (a22(1)+a12(1))/2,A2(1) ],[ (a22(2)+a12(2))/2,A2(2) ],
94   -           'g','LineWidth',2);
95   -    plot([pos_B(1),pos_A(1)],[pos_B(2),pos_A(2)],'k');
96
97        % Cuerpo B
98   -    plot([B1(1),B2(1)],[B1(2),B2(2)],'c','LineWidth',3);
99   -    plot(pos_B(1)+pasador(:,1),pos_B(2)+pasador(:,2),'c',
100           'LineWidth',2);
101  -    plot(pos_B(1),pos_B(2),'xr');
102  -    quiver(pos_B(1),pos_B(2),esc*cos(ori_B),esc*sin(ori_B),'r');
103
104       % Actuador r
105  -    plot([B2(1),A2(1)],[B2(2),A2(2)],'color',[0.5,0.5,0.5],
106           'LineWidth',2);
107
108       % Actuador l
109  -    plot([B1(1),A1(1)],[B1(2),A1(2)],'color',[0.5,0.5,0.5],
110           'LineWidth',2);
111
112       % Articulaciones
113  -    plot(B1(1),B1(2),'ok'); plot(B2(1),B2(2),'ok');
114  -    plot(A1(1),A1(2),'ok'); plot(A2(1),A2(2),'ok');
```

Con todos los puntos ya calculados, lo único que queda es representarlos mediante el comando *plot*. Se ha decidido representar la nube de puntos que representa los espacios de trabajo superpuestos para que se pueda apreciar en la animación, aunque no es necesario, y se puede omitir para ver solo el robot comentando la línea correspondiente (línea 79).

Para la representación de cuerpo A (líneas 82-95), que se representa en color verde en la Figura 23, y que básicamente consiste en la ranura del pasador, simplemente se deben representar los cuatro segmentos definidos por los puntos calculados anteriormente definidos como *a11*, *a12*, *a21* y *a22* (líneas 87-90), y los pequeños segmentos que unen los puntos de apoyo de los actuadores con la ranura (líneas 91-94), que se hacen con el fin de representar un único cuerpo sólido. Además, en esta sección también se ha representado la posición y orientación A, que representa el centro del eslabón A (líneas 83 y 86), se han configurado los ejes de representación (línea 84), se ha ejecutado el comando que representa los ejes con la misma proporción (línea 85), y se ha representado un segmento que une los puntos A y B (línea 95).

El cuerpo B es algo más sencillo de representar (representado en color cian en la Figura 23), puesto que únicamente se debe dibujar la línea que representa al eslabón (línea 98), el círculo del pasador (99-100) y la posición y orientación del punto B, centro del eslabón (líneas 101-102).

Finalmente, los actuadores *l* y *r* se representan como segmentos entre los puntos correspondientes de A y B, (dibujados en gris en la Figura 23); *l* se encuentra entre el punto izquierdo de A y el derecho de B (líneas 109-110) y r se encuentra entre el punto derecho de A y el izquierdo de B (líneas 105-106). Como añadido, se han representado las articulaciones de los actuadores con los eslabones (líneas 113-114).

```
116 -    pause(0.05)
117 -    if i ~= m
118 -        clf
119 -    end
120 -
121 -    if movB == 1
122 -        pos_F = pos_B;
123 -        ori_F = ori_B;
124 -    else
125 -        pos_F = pos_A;
```

```
126 -         ori_F = ori_A;
127 -     end
128
129 - end
```

Para concluir con el *script* **anim_adyacentes.m**, se añaden las líneas mostradas arriba. Lo primero que se ve es el tiempo entre fotograma y fotograma (línea 116), que a pesar de ser bajo, la capacidad del ordenador hace que no pueda funcionar tan rápido, por lo que básicamente se ha puesto a la máxima velocidad de animación posible, que es una velocidad aceptable que permite ver el movimiento con claridad. Las líneas 117-119 sirven para eliminar lo dibujado anteriormente, para que no muestre todos los instantes del robot a la vez, sino que solo se muestre el actual. Por último, se asignan los valores de *pos_F* y *ori_F*, que son los valores que devuelve la función, dependiendo de cuál sea el eslabón que se está moviendo (líneas 121-127). El *end* de la línea 129 es el que cierra el bucle *for* que contiene todo lo explicado hasta ahora.

6.3. Obtención de la secuencia entre dos configuraciones no-adyacentes.

Antes de explicar el funcionamiento del algoritmo anterior, se mencionó otro llamado **calculo_secuencia.m**, del que se explicó brevemente su funcionamiento y sobre el cual se profundizará a continuación. Como ya se dijo en su momento, este *script* se encarga de devolver la sucesión de índices correlativos que el robot debe seguir para alcanzar una configuración determinada empezando desde otra no adyacente. Las configuraciones mencionadas se pueden ver en la Figura 22.

Este código se trata de una función que simplemente recibe los índices inicial y final y devuelve la secuencia de adyacentes, y su principio de funcionamiento se muestra a continuación. Al tratarse de un código de aproximadamente 40 líneas de código, se subdividirá en apartados para una mayor claridad en la explicación.

```
5  -     v = [1,2,3,4,5,6,7,8];
6  -     v_triplicado = [v v v];
7
8  -     for t = 9:16
9  -         if v_triplicado(t) == indice_inicio
10 -             break
11 -         end
12 -     end
```

```
13
14 -     cont_asc = t;
15 -     cont_desc = t;
16 -     cont = 1;
```

Se debe aclarar que el objetivo no es solamente devolver la secuencia de índices adyacentes, sino devolver la más corta posible, puesto que al ser el 1 y el 8 adyacentes como se mostró previamente en la Figura 22, las rutas de movimiento pueden ser más cortas si el número del índice decrece en lugar de aumentar. Por ejemplo, del 1 al 6 resulta ser más corto cuando se realiza la secuencia 1-8-7-6 que 1-2-3-4-5-6.

En este fragmento de código lo que se hace básicamente es triplicar un vector que va desde 1 hasta 8 (líneas 5-6) y encontrar la posición del índice de inicio en el vector de en medio de los tres introducidos en el triplicado. Así, se podrá recorrer dicho vector en ambas direcciones para encontrar el índice final. Adicionalmente, se inicializan los contadores de las líneas 14-15, que corresponden con los que van a almacenar la posición de los índices dentro de *v_triplicado*, por eso se les da el valor inicial de *t*, que es donde se almacena la posición del índice inicial en el vector central, y el contador de la línea 16, que servirá para almacenar los índices en vectores independientes.

```
18 -     while 1
19 -         ind_antihorario(cont) = v_triplicado(cont_asc);
20 -         ind_horario(cont) = v_triplicado(cont_desc);
21 -         if v_triplicado(cont_asc) == indice_fin
22 -             ascendente = 1;
23 -             break
24 -         end
25 -         if v_triplicado(cont_desc) == indice_fin
26 -             ascendente = 0;
27 -             break
28 -         end
29 -         cont_asc = cont_asc+1;
30 -         cont_desc = cont_desc-1;
31 -         cont = cont+1;
32 -     end
33
34 -     if ascendente == 1
35 -         secuencia_animacion = ind_antihorario(1:cont);
36 -     else
37 -         secuencia_animacion = ind_horario(1:cont);
38 -     end
```

Precisamente lo comentado hace un instante es de lo que se encarga el bucle que se encuentra entre las líneas 18-32. En primer lugar, se almacenan los índices actuales en las posiciones definidas con los contadores *cont_asc* y *cont_desc* de los vectores independientes (líneas 19-20). A continuación, se comprueba si el índice analizado en ese momento corresponde con el final, en cuyo caso se interrumpe el bucle y se define mediante el indicador *ascendente* si la secuencia avanza en sentido positivo o negativo (líneas 21-28). En caso contrario, se actualiza el valor de los contadores (líneas 29-31) y se repite el proceso hasta que se encuentre el índice final en cualquiera de los dos sentidos.

Una vez encontrado e interrumpido el bucle, se asigna a la salida de la función el vector secuencia correspondiente según corresponda en función del indicador *ascendente* (líneas 34-38). Cabe destacar que en caso de que en ambos sentidos el tamaño del vector sea el mismo, cosa que es posible, se envíe por defecto la secuencia ascendente.

6.4. Animar medio ciclo y un ciclo completo.

Cuando se ha calculado la secuencia de índices adyacentes con **calculo_secuencia.m** y ya se puede animar entre todos ellos con **anim_adyacentes.m**, el siguiente paso es hacer un algoritmo que se encargue de animar medio ciclo de movimiento, es decir, desde la configuración inicial a la final, pasando por todas las intermedias. Esto se consigue con **con_adyacentes.m**.

```
4    -  secuencia_animacion = calculo_secuencia(indice_inicio,indice_fin);
5
6    -  if length(secuencia_animacion) == 1
7    -      [pos_F,ori_F] = anim_adyacentes(movB,secuencia_animacion,
8    -                          secuencia_animacion,pos_0,ori_0);
9    -  else
10   -      for c = 1:length(secuencia_animacion)-1
11   -          [pos_F,ori_F] = anim_adyacentes(movB,secuencia_animacion(c),
12   -                          secuencia_animacion(c+1),pos_0,ori_0);
13   -      end
14   -  end
```

Esta función recibe como parámetros de entrada los índices inicial y final, la variable que indica qué eslabón se mueve y la posición y orientación al inicio del proceso. Con ellos, simplemente se encarga de llamar a las funciones anteriores como corresponda.

Lo primero que se debe hacer, como es lógico, es llamar a la función **calculo_secuencia.m** (línea 4), puesto que es necesario obtener la variable *secuencia_animacion* antes de ejecutar ningún tipo de animación. Según el algoritmo empleado, la animación falla cuando la secuencia que se debe animar es solamente de tamaño 1, es decir, los índices inicial y final son el mismo (líneas 6-8), por lo que se ha establecido que el robot haga una pausa en ese momento, puesto que está previsto que el robot se mueva en ese lapso de tiempo, pero no lo hace, por tanto, se ha respetado ese tiempo. En caso contrario, se recorre el vector *secuencia_animación* y se le envían sus parámetros contiguos a **anim_adyacentes.m**, junto con *movB*, *pos_0* y *ori_0* (líneas 9-14). Al finalizar este bucle, la función devuelve los últimos valores de *pos_F* y *ori_F*, que se utilizarán en *scripts* de escalones superiores.

Hasta ahora ya es posible animar medio ciclo de trabajo, y el siguiente paso es animar un ciclo completo. Para esto, se ha hecho la función **anim_ciclo_completo.m**. Un ciclo completo necesita tres índices de configuraciones, este empieza en un índice, hace medio ciclo hasta el segundo índice y hace el segundo medio ciclo hasta el tercero, por lo que a esta función se le tienen que enviar como entrada estos tres índices (*indice_inicio*, *indice_medio*, *índice_fin*); además, también recibe la posición y orientación al inicio del ciclo.

```
4  - movB = 1;
5  - [pos_F1,ori_F1] = con_adyacentes(indice_inicio,indice_medio,
6  -                      movB,pos_0,ori_0);
7  - movB = 0;
8  - [pos_F,ori_F] = con_adyacentes(indice_medio,indice_fin,movB,
9  -                    pos_F1,ori_F1);
```

Esta función es tan sencilla como lo que se aprecia arriba. En primer lugar, se configura *movB* para que el primer medio ciclo se mueva el eslabón B y se le envía a la función **con_adyacentes.m**, junto con los índices inicial e intermedio, que son los que corresponden al primer medio ciclo, y la posición y orientación del robot al inicio del ciclo (líneas 4-6). A continuación, se cambia el valor de *movB* de tal forma que en el segundo medio ciclo sea el eslabón A el que se mueve, y se le envía a la misma función junto con los índices que le corresponden (intermedio y final) y la posición y orientación del robot al final del primer medio ciclo (líneas 7-9). Finalmente, se devuelven los valores de la posición y orientación finales para las funciones jerárquicamente superiores.

6.5. Animación de una trayectoria multi-ciclo.

Una vez ya se ha configurado la animación de un ciclo completo, nos hallamos en el último paso, que es el de animar todos los ciclos, cosa que se hace en el script **animacion_completa.m**. Este se divide en dos partes principales: la parte que ejecuta la animación desde el inicio hasta que el robot llega a la posición más cercana posible antes de orientarse y la parte en la que se orienta.

```
5   - ultimo_indice = 1;
6   - for h = 1:size(registro_seq(:,1),1)
7   -     seq_actual = registro_seq(h,:);
8   -     seq_actual = [ultimo_indice,seq_actual];
9   -     for i = 2:(N+1)
10  -         if seq_actual(i)==-1 || seq_actual(i+N)==-1
11  -             break
12  -         end
13  -         indice_inicio = ultimo_indice;
14  -         indice_medio = seq_actual(i);
15  -         indice_fin = seq_actual(i+N);
16  -         [pos_F,ori_F] = anim_ciclo_completo(indice_inicio,
17  -                         indice_medio,indice_fin,pos_0,ori_0);
18  -         ultimo_indice = indice_fin;
19  -         pos_0 = pos_F;
20  -         ori_0 = ori_F;
21  -     end
22  - end
```

La primera parte consiste básicamente en ir tomando los índices correspondientes de *registro_seq*, que era una matriz que almacenaba los vectores de las secuencias de cada ciclo, e ir enviándoselos a **anim_ciclo_competo.m** como corresponda.

Antes de nada, es necesario conocer cuál es la función de la variable *ultimo_indice*. Esta variable corresponde con el índice que representa la configuración con la que acabó el robot en el ciclo anterior, o en caso de la línea 5, el inicial. Esta variable es necesaria enviarla siempre en el inicio de cada ciclo a la función, por lo que se ha decidido añadirla directamente al vector de secuencia en cada caso. Con este concepto claro, se puede empezar con la explicación en profundidad del bucle de ejecución.

Dicho bucle tiene el mismo número de iteraciones que número de secuencias tenga guardadas la matriz *registro_seq* (línea 6). Dentro del mismo, lo primero que se hace es

extraer la secuencia que toque en cada momento (línea 7) y añadirle la variable *ultimo_indice* al inicio (línea 8).

A continuación, en las líneas 9-21 se encuentra un bucle secundario que se encarga de recorrer cada vector secuencia; cabe destacar que lo primero que se hace en este bucle es comprobar si el vector tipo secuencia está lleno, o si por el contrario tiene términos iguales a -1, en cuyo caso se interrumpe el bucle (líneas 10-12). El funcionamiento de este consiste en, en primer lugar, obtener los valores de los índices que se le van a enviar a la función **anim_ciclo_completo.m** (líneas 13-15); indice_inicio corresponde con el índice en el que el robot acabó el ciclo anterior, e indice_medio e indice_fin corresponden con el valor de *i* y el de *j* respectivamente de la iteración actual. Posteriormente, se ejecuta la función **anim_ciclo_completo.m** enviándole como entradas los tres índices y la posición y orientación del robot al inicio del ciclo (líneas 16-17). Además, es imprescindible que se actualicen los valores de *ultimo_indice* con el índice final del ciclo y los valores de *pos_0* y *ori_0* con la posición y orientación finales del ciclo (líneas 18-20).

```
24  -  if flag == 1
25  -      seq_actual = secuencia_final;
26  -      seq_actual = [ultimo_indice,seq_actual];
27  -      for z = 2:4
28  -          if seq_actual(z)==-1 || seq_actual(z+3)==-1
29  -              break
30  -          end
31  -          indice_inicio = ultimo_indice;
32  -          indice_medio = seq_actual(z);
33  -          indice_fin = seq_actual(z+3);
34  -          [pos_F,ori_F] = anim_ciclo_completo(indice_inicio,
35  -                          indice_medio,indice_fin,pos_0,ori_0);
36  -          ultimo_indice = indice_fin;
37  -          pos_0 = pos_F;
38  -          ori_0 = ori_F;
39  -      end
40  -  end
```

La segunda parte de este script tiene el mismo principio de funcionamiento que la primera, pero solamente se ejecuta para un vector tipo secuencia. Se recuerda que la parte final del proceso consistía en orientar el robot, y que esto siempre se hacía tras 3 ciclos de trabajo y que la señal *flag* (línea 24) indicaba cuándo era necesaria la orientación, es

decir, cuándo no coincidía la orientación del robot antes de orientarse con la de referencia. Una vez recordado estos conceptos, se procederá a la explicación de este último paso.

Esta parte consiste, básicamente, en hacer lo mismo que en la primera, pero solo una vez. Por consiguiente, lo primero es obtener la secuencia sobre la que se va a trabajar, almacenada en *secuencia_final* (línea 25), y añadirle el índice que representa la configuración final del robot en el paso anterior (línea 26). Posteriormente, se ejecuta el mismo bucle que en la primera parte, pero para 3 ciclos fijos, por lo que lo primero es comprobar si existen índices iguales a -1 (líneas 28-30), y una vez comprobado, comenzar a asignar los valores de *indice_inicio*, *indice_medio* e *indice_fin* (líneas 31-33). Finalmente, se llama a la función **anim_ciclo_completo.m** enviándole los mismos parámetros que en la primera parte (líneas 34-35) y se actualizan los valores de *ultimo_indice*, *pos_0* y *ori_0* para próximas iteraciones.

6.6. Ejemplo final.

A continuación, se van a mostrar una serie de fotogramas que consisten en capturas de pantalla de la ejecución del *script* de animación para el ejemplo expuesto en el capítulo 5, que corresponde con una referencia de (400, 500) en posición y π.

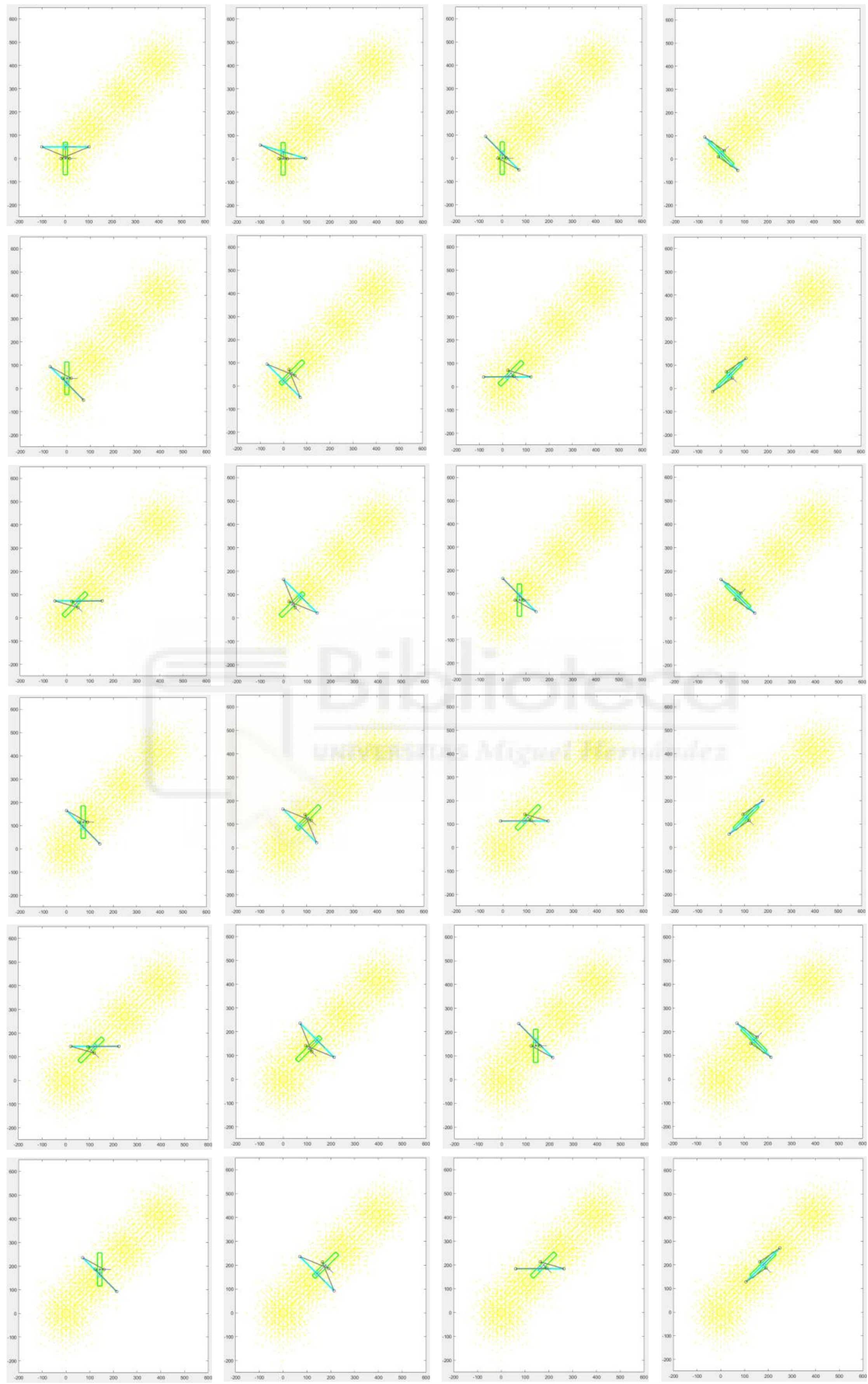


Figura 24: Secuencia de fotogramas de la animación.

Biblioteca
UNIVERSITAS Miguel Hernández

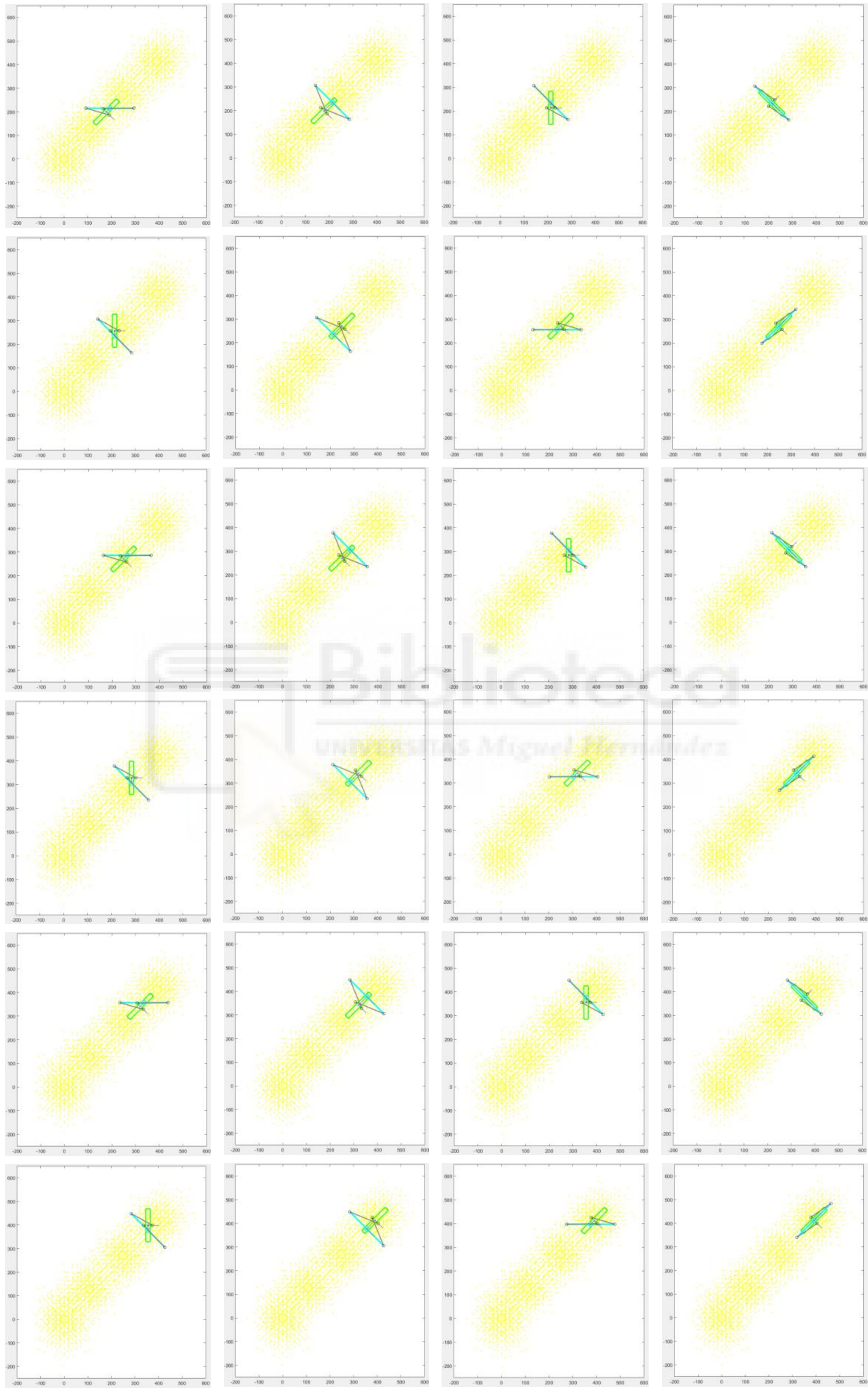


Figura 25: Continuación de la Figura 24.

Biblioteca
UNIVERSITAS Miguel Hernández

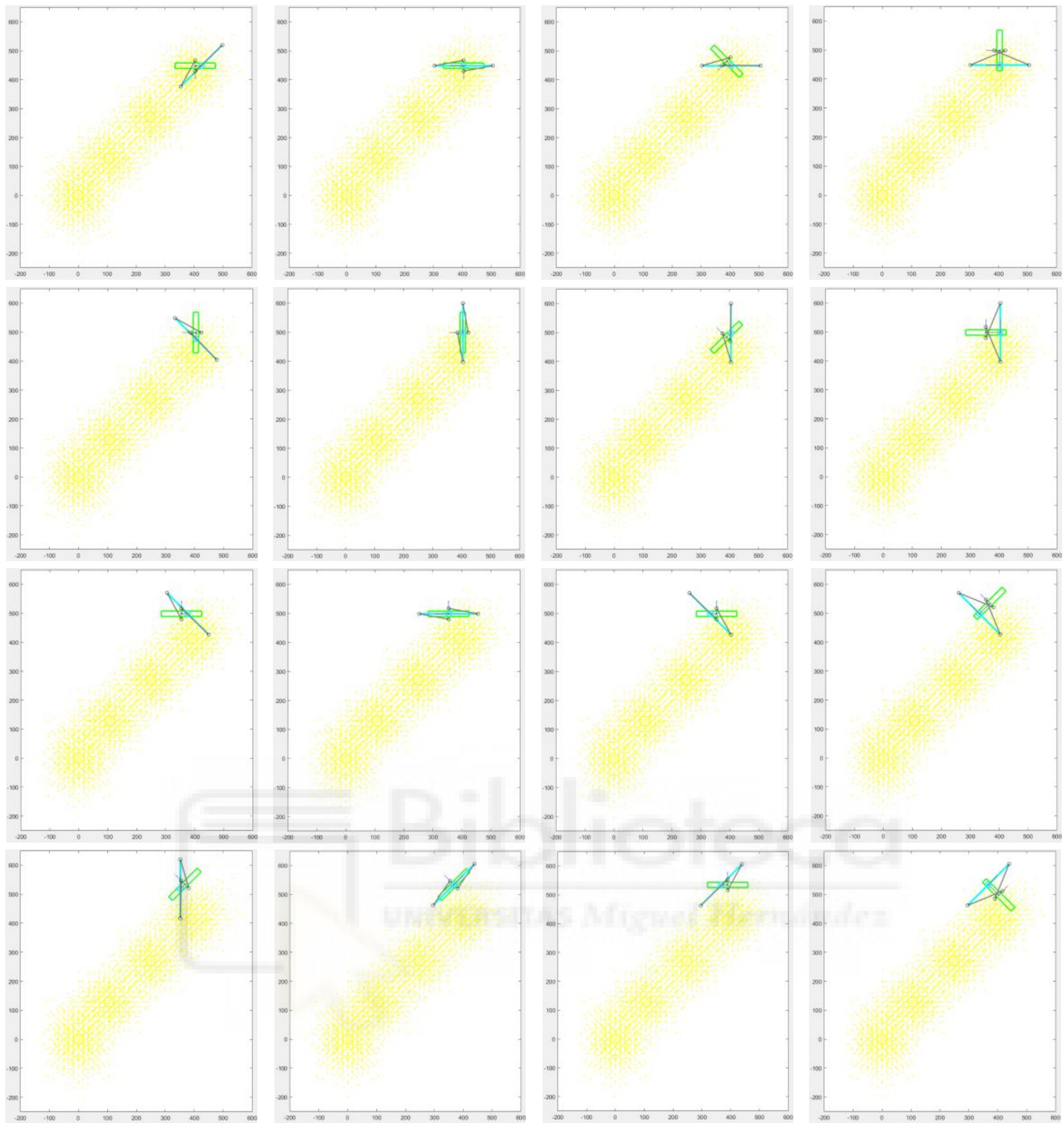


Figura 26: Continuación de la Figura 25.

Capítulo 7. Consideración de obstáculos.

En este último capítulo se pretende, a partir del desarrollo de la planificación de trayectorias del robot en espacios libres de obstáculos, intentar que el robot sea capaz de llegar al punto de referencia evadiendo obstáculos dispuestos a lo largo del espacio 2D. Para ello, en resumen, se pretende inscribir al robot dentro de una circunferencia sobre la que se calculará si los obstáculos inciden en ella.

Para el correcto desarrollo de este algoritmo, lo primero que se va a resolver es la situación donde los obstáculos sean simplemente puntos en el espacio que el robot deba esquivar, y posteriormente se añadirá la opción de que dichos obstáculos consistan en superficies "prohibidas" que pueden discretizarse y aproximarse por una serie de nubes de puntos.

7.1. Obstáculos puntuales.

Como se acaba de comentar, este problema se comenzará abordando con la premisa de que los obstáculos son simplemente puntos aislados. Para lograr el resultado más correcto posible, se ha definido una circunferencia alrededor del robot dentro de la cual se va a comprobar que no haya ningún punto obstáculo. No obstante, como se explicará más en profundidad en los siguientes párrafos, este algoritmo no va a dar el resultado más óptimo posible, puesto que solamente se comprueban las colisiones con los obstáculos para las poses de inicio y fin, y no para las maniobras intermedias que el robot realiza para moverse entre dichas poses. Es decir, podría ocurrir que, a pesar de no colisionar para las poses extremas, alguna pose intermedia sí que provoque alguna colisión, pero comprobar colisiones en cada instante de tiempo complicaría mucho el algoritmo, y esto se queda fuera del alcance de este Trabajo Fin de Grado.

Antes de empezar con la explicación del algoritmo, es necesario mencionar qué *scripts* van a intervenir en este desarrollo. Todos ellos se encuentran en el Anexo 6, y son los siguientes: **ws_binario_mp.m**, que es el *script* que más cambios recibe para la resolución de este problema, puesto que es donde se toman la mayoría de las medidas para el correcto desarrollo del movimiento del robot; **ciclo_completo.m**, en el cual simplemente se han eliminado un par de líneas que se encargaban de acortar las variables *nube* y *secuencia* porque estas reducían el rendimiento del algoritmo, y en este caso es importante debido al hecho de que este requiere más recursos de computación que los

anteriores; **encontrar_punto.m**, cuya función es similar a su homónimo de otros capítulos, pero se ha tenido que modificar por cuestiones de rendimiento que se especificarán posteriormente; al *script* **dibujo.m** simplemente se le ha añadido lo necesario para dibujar los obstáculos, y finalmente, **calculo_phi_dif.m** y **ori_final.m** sí que se mantienen idénticos a sus anteriores versiones.

Los primeros cambios que se explicarán serán los de **encontrar_punto.m**, debido simplemente a que son menos extensos que los de **ws_binario_mp.m**.

La primera modificación sustancial que se le ha realizado ha sido el de cambiar su estructura de *script* a función, cuya entrada es el vector de distancias que se calcula en **ws_binario_mp.m**, y la razón de ello es que en este caso se debe ejecutar en dos situaciones diferentes con dos vectores de distancias diferentes que no pueden ser sobrescritos uno sobre el otro, por lo que ha sido necesario que sea una función con un vector de distancias como entrada. Por otra parte, para que todas las variables que se usan dentro de la función y se necesitan fuera de ella y viceversa, es necesario declarar como globales las siguientes: *nube*, *indice_min*, *secuencia_encontrada*, *P_mas_cercano*, *ori_mas_cercano* y *secuencia*.

```
12     % Búsqueda del punto más cercano al de referencia
13  -  [distancia_minima,indice_min] = min(distancias);
14  -  P_mas_cercano = nube(indice_min,1:2);
15  -  ori_mas_cercano = nube(indice_min,3);
16
17     % Obtención de la secuencia de i y j del punto encontrado
18  -  secuencia_encontrada = secuencia(indice_min,:);
```

Una vez definidas las variables globales, el objetivo de esta función es encontrar el índice del punto de menor distancia (línea 13) y obtener el punto de *nube* que corresponde con ese índice (línea 14). A pesar de que este es el objetivo principal de esta función, también se obtienen la orientación y la secuencia (líneas 15 y 18 respectivamente) del punto encontrado para posteriores operaciones.

A continuación, se procederán a comentar todos los cambios realizados en el *script* **ws_binario_mp.m**. Además de la declaración de las variables pertinentes como globales, los añadidos de la primera parte del *script*, que es básicamente donde se declaran e inicializan variables son los siguientes:

```
17  -  radio_robot = 150;
18
19     %Obstáculos
20  -  P_obs = [300 300;200 375];
21  -  Rmax = 50;
22  -  x_p = 0;
23  -  y_p = 0;
```

Dichos cambios consisten básicamente en la definición del vector de obstáculos (línea 20) y del valor *Rmax* (línea 21), cuya función se explicará más adelante, la inicialización de *x_p* e *y_p* que son variables auxiliares que se utilizarán en la definición de un punto intermedio en la trayectoria, y el más importante, la definición del radio de la circunferencia circunscrita al robot (línea 17). Dicha relevancia reside en que este va a ser el valor que se defina para que el robot no colisione con el obstáculo, y, como se ha comentado en la introducción de este capítulo, no se puede saber cuál es el valor más óptimo para este parámetro.

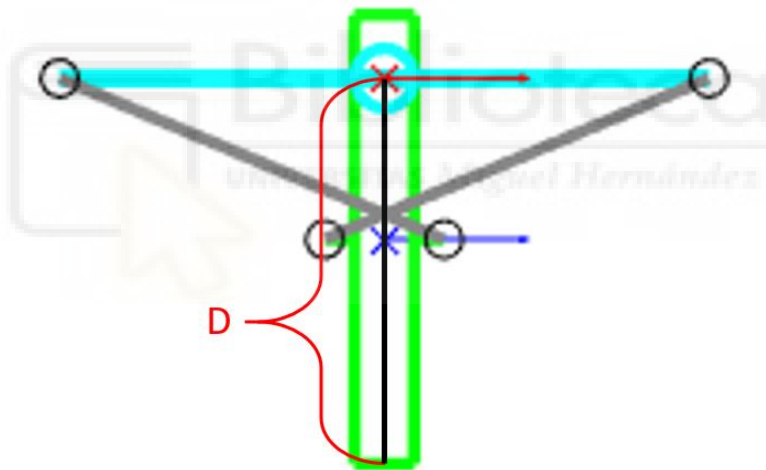


Figura 27: Representación de la distancia mínima al obstáculo.

El parámetro *D* de la Figura 27 representa el radio mínimo que debe tener la circunferencia circunscrita al robot, puesto que cuando el cuerpo en movimiento sea el cuerpo A, la circunferencia se centrará en el centro del eslabón B, por lo que el radio mínimo va a ser, para las dimensiones que se han predefinido de este robot, 70 mm correspondientes a la mitad del eslabón A y 50.242 mm correspondientes a la distancia entre A y B en el instante de la figura, que va a ser el máximo en todos los casos; en total, 120.242 mm. Sin embargo, al tratarse de un algoritmo que solamente comprueba la distancia a los obstáculos en el inicio y el fin de cada conjunto de ciclos, es necesario mayorar esta distancia para obtener un resultado que haga que el robot no colisione en los

instantes intermedios entre el principio y el fin del conjunto de ciclos. Se ha de tener cuidado con el valor que se le da a este radio, puesto que un radio muy elevado haría que ningún punto posible del espacio cumpla con la condición. La mayoración que se ha decidido hacer es la de un 25% aproximadamente, resultando en dar el valor de 150 mm a este radio (línea 17). En un algoritmo más sofisticado, la mayoración que se haría podría ser mucho menor o incluso llegar a ser prácticamente nula, pero ya sería un algoritmo demasiado complejo que se queda fuera del alcance de este Trabajo Fin de Grado.

Después de estos cambios, las siguientes líneas son idénticas a las de otros experimentos, teniendo como función reservar memoria e inicializar variables que ya se usaron en otros experimentos, como *nube* o *secuencia* entre otras, asignar los valores de los vectores *phi* e *y*, y crear T, que se trata de la celda de matrices con las configuraciones del robot, exactamente igual que en el resto de los capítulos. De esta parte en adelante, se ha configurado el algoritmo que permite resolver el problema planteado. Cabe destacar que el bucle *while* que se verá al inicio del código mostrado a continuación es un bucle que ocupa desde la línea 54 hasta la 167, pero que se mostrará dividido en secciones para una mejor explicación del mismo.

```
54   -  while 1
55   -    if norm(P_ref-P0) < 10
56   -      break
57   -    end
58
59   -    if f_aux == 1
60   -      global_counter = 0;
61   -      ciclo_completo(P0,phi0,T,1,N)
62   -      f_aux = 0;
63   -      distancias = zeros(size(nube,1),1);
64   -      for i=1:size(nube,1)
65   -        distancias(i) = norm(P_ref-nube(i,1:2));
66   -      end
67   -    end
68
69   -    encontrar_punto(distancias)
70
71   -    for d = 1:size(P_obs,1)
72   -      if norm(P_mas_cercano-P_obs(d,:)) > radio_robot
73   -        flag_conf = 1;
74   -      else
75   -        flag_conf = 0;
76   -        break;
```

```
77  -       end
78  -     end
```

Lo primero que se configura en el bucle es la condición de ruptura de este (líneas 55-57), que como en los capítulos 4 y 5, esta se cumple cuando el robot se ha acercado a la referencia a una distancia que se considera suficientemente pequeña, establecida por defecto en 10 mm.

La idea general de este algoritmo es, a grandes rasgos, buscar el punto más cercano al objetivo y comprobar si este se encuentra lo suficientemente lejos de un obstáculo como para ser válido, y en caso de no ser así, descartarlo y buscar otro, por lo que en este caso la ejecución de **ciclo_completo.m** se debe hacer solamente una vez y a partir de esa nube de puntos, encontrar el punto correcto, y para saber si se debe ejecutar o no dicha función, se ha creado la variable *f_aux*, que será 1 en caso de requerirse la ejecución de la función (línea 59), y 0 en caso contrario. Esta variable booleana se inicializa en 1 fuera del bucle y dentro de este va adoptando el valor que necesita en cada momento. La ejecución de **ciclo_completo.m** se ve acompañada de otra acción que se debe hacer simultánea a ella, por ello, el *if* que se ejecuta solo cuando es la primera iteración del cálculo (cuando f_aux=1) ocupa las líneas 59-67. Estas líneas incluyen una inicialización de *global_counter*, (línea 60) permitiendo así que las nubes de puntos calculadas no se acumulen dentro de la variable *nube* y así se deban hacer menos cálculos, puesto que hasta ahora se estaban almacenando todos los espacios de trabajo solapados dentro de *nube* y, en este caso, se requiere un rendimiento lo más óptimo posible, por lo que las nubes no se pueden acumular. Después de esto, se ejecuta la función **ciclo_commpleto.m** (línea 61) y se actualiza el parámetro *f_aux* (línea 62), para a continuación hacer, con la nube calculada, el cálculo del vector *distancias* (líneas 63-66), que será el que se le envíe como entrada a la función **encontrar_punto.m** (línea 69).

La siguiente sección de código se encarga de comprobar que el punto definido como más cercano al objetivo se encuentre a una distancia suficiente de todos los obstáculos. Para ello, se ha creado un bucle (líneas 71-78) que comprueba si todos los puntos obstáculos se encuentran a más de la distancia definida como *radio_robot* del punto encontrado como más cercano (*P_mas_cercano*). En caso de cumplirse esa distancia para todos los obstáculos, se define la variable *flag_conf* como 1, y en caso de

que al menos uno de los obstáculos no cumpla esta distancia con el punto, se define *flag_conf* como 0. A partir de esta variable, el algoritmo actuará de una forma u otra.

El caso en que *flag_conf* es 1 ocupa las líneas 80-162, por lo que antes de explicar esta parte se explicará el caso en que *flag_conf* es 0, que se representa con el *else* que se muestra a continuación:

```
163 -   else
164 -     distancias(indice_min) = 1000000000;
165 -   end
```

Como se acaba de explicar, que *flag_conf* sea 0 quiere decir, en pocas palabras, que el punto encontrado no es válido, por lo que se penaliza el valor de la distancia mínima encontrada (línea 164). Esta es la única acción que debe hacerse, puesto que, al volver a ejecutarse el bucle, ya no se calculará de nuevo el vector *distancias*, por lo que la siguiente búsqueda será del siguiente punto más cercano.

A continuación, se procederá a la explicación del caso en que *flag_conf* es 1, el cual constituye la mayor parte de este *script* debido a un caso concreto que se comentará posteriormente.

```
80  -   if flag_conf == 1
81  -     registro_seq = [registro_seq; secuencia_encontrada];
82  -     puntos_encontrados = [puntos_encontrados; P_mas_cercano];
83  -     ori_encontradas = [ori_encontradas; ori_mas_cercano];
84  -     P0 = P_mas_cercano;
85  -     phi0 = nube(indice_min,3);
86  -     f_aux = 1;
```

El caso más simple posible es en el que el robot es capaz de esquivar los obstáculos sin problema, para lo cual únicamente se necesitan las líneas que se acaban de mostrar (salvo la 82 y la 83, que son variables que se usan en el caso especial mencionado anteriormente y que se explicará en los siguientes párrafos) más las mostradas hasta ahora. Para ello, simplemente se almacena la secuencia encontrada mediante la función **encontrar_punto.m** en la matriz *registro_seq* (línea 81), se actualizan las variables *P0* y *phi0* (líneas 84-85), que son la posición y la orientación con las que el robot termina el movimiento y por tanto son las entradas de **ciclo_competo.m** en la siguiente iteración, y se actualiza el valor de *f_aux* (línea 86) para que en la siguiente iteración del bucle *while* se ejecuten las líneas 59-67, explicadas anteriormente. Para el caso que nos ocupa, en este

instante debemos almacenar en matrices las posiciones y orientaciones de los puntos que el robot ha ido encontrando (líneas 82-83), calculadas mediante la función **encontrar_punto.m**, que serán utilizadas más tarde.

Como se acaba de comentar, aquí acabaría el algoritmo si el robot no encontrara problemas para evitar los obstáculos y llegar al objetivo, pero existe la posibilidad de que este encuentre un mínimo local del cual no sea capaz de salir y se quede deambulando a su alrededor indefinidamente, por lo que se ha tenido que implementar un algoritmo que detecte cuándo este se ha atascado y lo saque del problema. El sistema que se ha decidido implementar para solventar este problema consiste, en pocas palabras, en detectar cuándo el robot se ha atascado y enviarlo a un punto intermedio aleatorio para que desde este intente alcanzar la referencia inicial.

```
87   -      tam_p = length(puntos_encontrados(:,1));
88   -      if tam_p > 4
89   -        X_v = puntos_encontrados(tam_p-4:tam_p,1);
90   -        Y_v = puntos_encontrados(tam_p-4:tam_p,2);
91   -        DX = std(X_v);
92   -        DY = std(Y_v);
93   -        if DX < 5 && DY < 5
94   -          Rmax = Rmax + 25;
95   -          tam = length(registro_seq(:,1));
96   -          registro_seq = registro_seq(1:tam-4,:);
97   -          puntos_encontrados = puntos_encontrados(1:tam-4,:);
98   -          ori_encontradas = ori_encontradas(1:tam-4,:);
99   -          P_mas_cercano = puntos_encontrados(tam-4,:);
100  -          P0 = P_mas_cercano;
101  -          phi0 = ori_encontradas(tam-4,:);
```

El primer paso para lograr este objetivo es detectar cuándo el robot se ha atascado, y para ello se debe utilizar la variable *puntos_encontrados*, mencionada anteriormente, que se encarga de almacenar los puntos que el robot va encontrando. El objetivo de esta variable es extraer los últimos cinco puntos encontrados (líneas 89-90) y ver si son lo suficientemente cercanos como para que el robot pueda estar atascado, cosa que se comprueba calculando las desviaciones típicas de sus coordenadas X e Y (líneas 91-92); el caso que nos atañe, que es aquel en el cual el robot se encuentre moviéndose poco o nada alrededor de un mínimo local, se detecta cuando las desviaciones típicas de ambas componentes de su posición son pequeñas o nulas, por ello, en la línea 93 se ha definido la condición de que ambas deben ser menores a 5 mm para que se aplique este método.

Rmax (línea 94) es una variable que determina la distancia máxima a la que va a estar el punto aleatorio del robot. Su valor inicial es 50 mm, y se ha definido que cada vez que no se encuentre un punto intermedio válido, esta distancia aumente en 25, siendo 75 su valor en el primer intento. La idea principal de este aumento progresivo es que el robot sea capaz de salir de los atascamientos que pueda llegar a experimentar, y pueden existir casos en los que estos sean demasiado restrictivos para el robot y no consiga alcanzar la nueva referencia intermedia, por lo que aumentando el radio en cada iteración se intenta que, dándole al robot puntos más lejanos, este tenga más posibilidades de superar las dificultades que le suponen los mínimos locales.

Como se ha comentado anteriormente, el algoritmo detecta cuándo el robot se ha movido poco o nada en los últimos cinco puntos, por lo que el primer paso para un movimiento mínimamente eficiente es que esos cinco puntos sean considerados como uno solo, por lo que se eliminan las últimas cuatro secuencias de la matriz *registro_seq* (líneas 95-96). En este instante, las variables *puntos_encontrados* y *ori_encontradas* van a ejercer la función para la que han sido creadas, y esta consiste en poder actualizar la posición y la orientación del robot en el momento de eliminar las secuencias de *registro_seq*, puesto que estas pueden ser diferentes en los cinco puntos en los que se ha detectado la desviación típica como pequeña, y lo que interesa es que el robot se encuentre como se encontraba en el primero de esos cinco puntos, que es el que se ha almacenado como bueno antes de ejecutar el método de referencia aleatoria. Para ello, al igual que con *registro_seq*, se eliminan los últimos cuatro valores de dichos vectores (líneas 97-98) y se actualizan las variables *P_mas_cercano* (línea 99), *P0* (línea 100) y *phi0* (línea 101).

```
102 -        while 1
103 -          R = Rmax*rand;
104 -          angulo_hacia_destino=atan2(P_ref(2)-P0(2),P_ref(1)-P0(1));
105 -          if rand > 0.5 % Virar hacia la derecha
106 -            angulo = angulo_hacia_destino - pi/2 + pi/2*rand;
107 -          else % Virar hacia la izquierda
108 -            angulo = angulo_hacia_destino + pi/2 + pi/2*rand;
109 -          end
110 -          x_p = P_mas_cercano(1) + R*cos(angulo);
111 -          y_p = P_mas_cercano(2) + R*sin(angulo);
112 -          P_ref_n = [x_p,y_p];
113 -          for d = 1:size(P_obs,1)
114 -            if norm(P_ref_n-P_obs(d,:)) > radio_robot
115 -              flag_conf_3 = 1;
116 -            else
```

```
117 -                   flag_conf_3 = 0;
118 -                   break;
119 -                 end
120 -               end
121 -               if flag_conf_3 == 1
122 -                 break;
123 -               end
124 -             end
```

El siguiente paso es generar la nueva referencia aleatoria, y como es obvio, la mejor idea no es generar un punto aleatorio en el espacio y e imponérselo como nueva referencia al robot, sino que es necesario tener en cuenta ciertos aspectos.

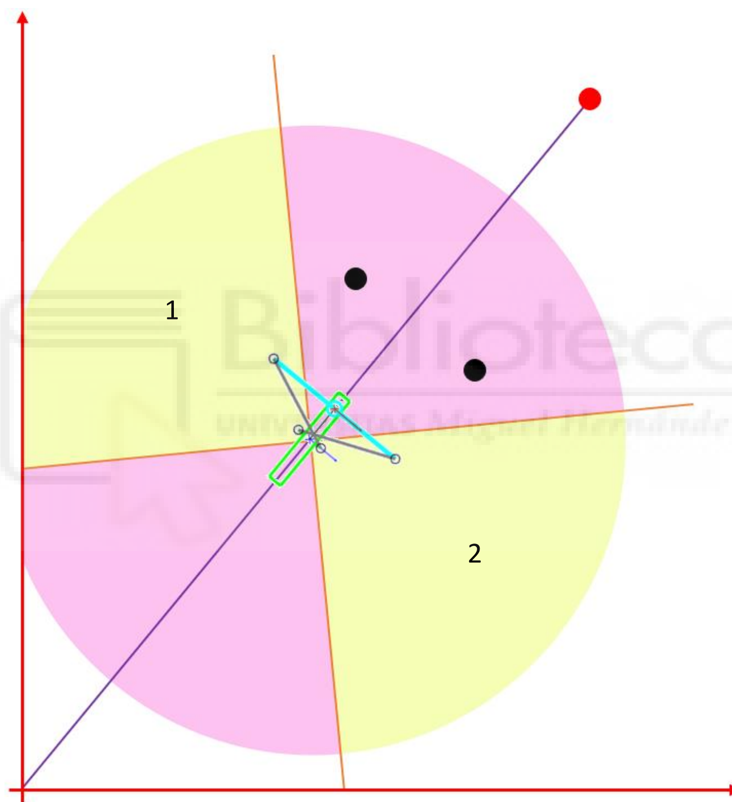


Figura 28: División de las zonas de puntos aleatorios.

La Figura 28 muestra una situación de mínimo local para el robot. El objetivo del movimiento se encuentra representado mediante un círculo rojo, los ejes de coordenadas mediante flechas rojas y los obstáculos mediante dos círculos negros. La línea morada une el centro del robot con la referencia y las líneas naranjas delimitan los sectores circulares que se explican a continuación.

En este caso, lo que debe hacer el robot es generar un punto aleatorio al que dirigirse antes de retomar su camino hacia la referencia. La idea que se ha implantado en

este algoritmo es que el robot genere los puntos aleatorios en las zonas pintadas en amarillo, puesto que las zonas rosas hacen que este o bien no salga del mínimo local o bien salga y vuelva a entrar en él. Es posible que haya situaciones en las que este algoritmo dé problemas, como por ejemplo que haya obstáculos en las zonas amarillas y no sea capaz de evitarlos, pero si no se aplica este método el algoritmo es demasiado lento, por lo que se ha decidido implementar para obtener un mayor rendimiento.

Para implementar esta condición, se ha calculado el ángulo que forma la recta que une el centro del robot con la referencia y el eje de coordenadas X (línea 104), que en el caso de la Figura 28 corresponde con la línea morada. A continuación, se determina si el ángulo se encontrará a la izquierda de la línea (zona 1 de la Figura 28) o a la derecha (zona 2 de la Figura 28), para lo que se determina un número aleatorio entre y 1 (línea 105) y según sea su valor mayor o menor a 0.5, se determina si el robot vira a derecha o izquierda respectivamente. Una vez determinado, se genera el ángulo a la derecha (línea 106) o a la izquierda (línea 108) según corresponda.

Con este ángulo y el radio generado también aleatoriamente en la línea 103, se generan las coordenadas del punto de referencia intermedio (líneas 110 y 111) para a continuación ser unificadas (línea 112).

Como última consideración, se debe tener en cuenta que este nuevo punto no puede ser próximo a un obstáculo, ya que el robot nunca sería capaz de alcanzarlo, por lo que se hace un pequeño bucle que comprueba que este nuevo punto se encuentra suficientemente lejos de todos los obstáculos (líneas 113-120), el cual ya se ha utilizado anteriormente (líneas 71-78) por lo que no se incidirá en él. En caso de que este punto sea válido, flag_conf_3 tendría valor 1 y se rompería el bucle *while* iniciado en la línea 102 gracias al *if* definido en las líneas 121-123. Sin embargo, en caso contrario, se volverá a ejecutar el bucle *while* y se volverá a generar otro punto aleatorio hasta que se encuentre uno que cumpla todas las condiciones impuestas.

La siguiente parte del código sigue el mismo principio que el código principal pero para el punto de referencia intermedio, por lo que se explicará con menos detalle debido a que sería en cierto modo redundante. Se incidirá y explicará en profundidad las partes que no aparezcan o sean diferentes a la idea general del script.

```
125 -         while 1
```

```
126 -          if norm(P_ref_n-P0) < 10
127 -            break
128 -          end
129 -          if f_aux_2 == 1
130 -            global_counter = 0;
131 -            ciclo_completo(P0,phi0,T,1,N)
132 -            f_aux_2 = 0;
133 -            distancias_n = zeros(size(nube,1),1);
134 -              for i=1:size(nube,1)
135 -                distancias_n(i) = norm(P_ref_n-nube(i,1:2));
136 -              end
137 -          end
138 -          encontrar_punto(distancias_n)
139 -          for d = 1:size(P_obs,1)
140 -            if norm(P_mas_cercano-P_obs(d,:)) > radio_robot
141 -              flag_conf_2 = 1;
142 -            else
143 -              flag_conf_2 = 0;
144 -              break;
145 -            end
146 -          end
147
148 -          if flag_conf_2 == 1
149 -            registro_seq=[registro_seq; secuencia_encontrada];
150 -            puntos_encontrados=[puntos_encontrados; P_mas_cercano];
151 -            ori_encontradas=[ori_encontradas; ori_mas_cercano];
152 -            P0 = P_mas_cercano;
153 -            phi0 = ori_mas_cercano;
154 -            f_aux_2 = 1;
155 -            nube_grande = [nube_grande;nube];
156 -          else
157 -            distancias(indice_min) = 1000000000;
158 -          end
159 -        end
```

Esta parte del código consiste en un bucle while que se rompe cuando el robot alcanza la posición que se le ha impuesto (líneas 126-128). La variable *f_aux_2* se inicializa fuera del bucle, y su valor es 1 cuando es la primera vez que se ejecuta el algoritmo para la posición actual del robot, y 0 en caso de que no sea la primera vez; cuando esta es 1, se inicializa *global_counter* (línea 130), se ejecuta **ciclo_completo.m** (línea 131), se actualiza su propio valor (línea 132), y se calcula el vector *distancias_n* (líneas 133-136), que contiene todas las distancias entre el punto actual y los puntos de *nube*. Una vez ejecutadas estas líneas en caso de que fuera necesario, se ejecuta la función **encontrar_punto.m** con el vector *distancias_n* como entrada (línea 138), que va a ser la

primera línea que se ejecute dentro del bucle en caso de que *f_aux_2* sea 0. A continuación, se comprueba que el punto encontrado mediante la función encontrar_punto.m es válido en cuanto a su distancia con respecto a los obstáculos, y esto se comprueba mediante el bucle situado en las líneas 139-146, dando a la variable *flag_conf_2* los valores de 1 o 0 en caso de que el punto sea válido o no válido respectivamente. En caso de que el punto sea válido, se almacenan la secuencia, la posición y la orientación en sus respectivas variables (líneas 149-151) y se actualizan los valores de P0, phi0 y f_aux_2 (líneas 152-154). La línea 155 se encarga de almacenar la nube de puntos actual en una matriz llamada *nube_grande*, que se encarga de almacenar todas las nubes del recorrido para poder dibujarlas al terminar los cálculos. Esta línea también se encuentra en el bucle *while* general, pero más adelante, por lo que se mostrará en una sección de código posterior. Finalmente, en caso de que el punto encontrado no sea válido (*else* de la línea 158) se penaliza la distancia del punto actual y se procede a ejecutar el algoritmo de nuevo para buscar el siguiente punto más cercano.

```
160 -         end
161 -       end
162 -       nube_grande = [nube_grande;nube];
163 -     else
164 -       distancias(indice_min) = 1000000000;
165 -     end
166
167 -   end
```

Finalmente, las líneas que se acaban de mostrar cierran el código principal de este algoritmo. El *end* de la línea 160 cierra el *if* de la línea 93, que era el que comprobaba si las desviaciones típicas tenían valores reducidos, y el *end* de la línea 161 cierra el *if* de la línea 88, que se encargaba de comprobar si el robot había pasado por 5 puntos para poder comparar sus desviaciones típicas. La línea 162 se encarga del almacenamiento de la nube de puntos para su posterior dibujo, como se indicaba en el párrafo anterior. Finalmente, las líneas 163-165 se tratan del caso en que *flag_conf* es 0 el cual ha sido explicado anteriormente y se ha vuelto a mostrar simplemente para que sea más fácil situarlo dentro del código. Como última línea a comentar, el *end* de la línea 167 se encarga de cerrar el bucle *while* de la línea 54, que es el que engloba todo este código.

Por último, las líneas restantes del código que se muestran en el Anexo 6 se encargan, en pocas palabras, de comprobar si se debe ejecutar o no el script **ori_final.m**.

Estas líneas ya han sido explicadas con detenimiento en el capítulo 5, por lo que no se volverá a incidir.

A continuación, se van a mostrar dos ejemplos del resultado de este algoritmo. Ambos tendrán como referencia la misma que se tomó en el capítulo 5, es decir, la posición de referencia se situará en (400, 500) y la orientación deseada será π/4, lo que cambiará en cada ejemplo será la posición de los obstáculos, representados en ambos casos como círculos negros.

En el primer caso, existirán tres puntos obstáculo: (200, 200), (300, 300) y (200, 400). Este es un caso en el que el robot no se atasca, por ello, en este caso la ejecución del algoritmo resulta ser muy eficiente y rápida. Al igual que en el capítulo 5, la posición y orientación de referencia se encuentran representados con un círculo y una flecha rojas respectivamente, la posición y orientación del robot en el momento antes de orientarse se encuentran representadas mediante un asterisco y una flecha magenta respectivamente, y la posición y orientación finales se encuentran representadas mediante un cuadrado y una flecha verdes respectivamente.

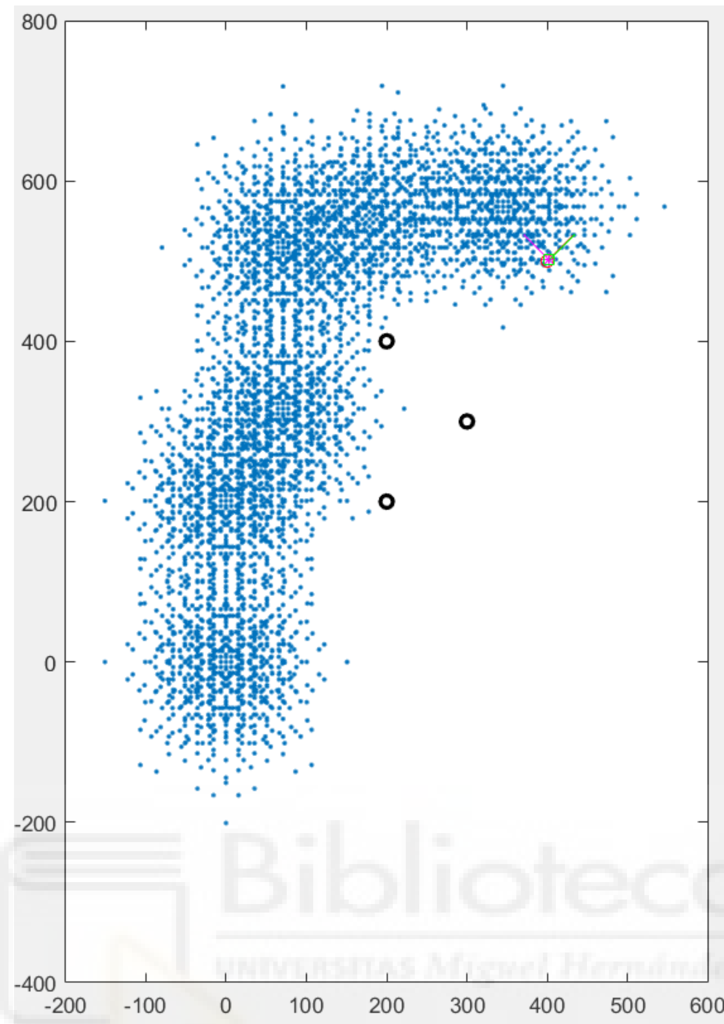


Figura 29: Solución con obstáculos sin mínimo local.

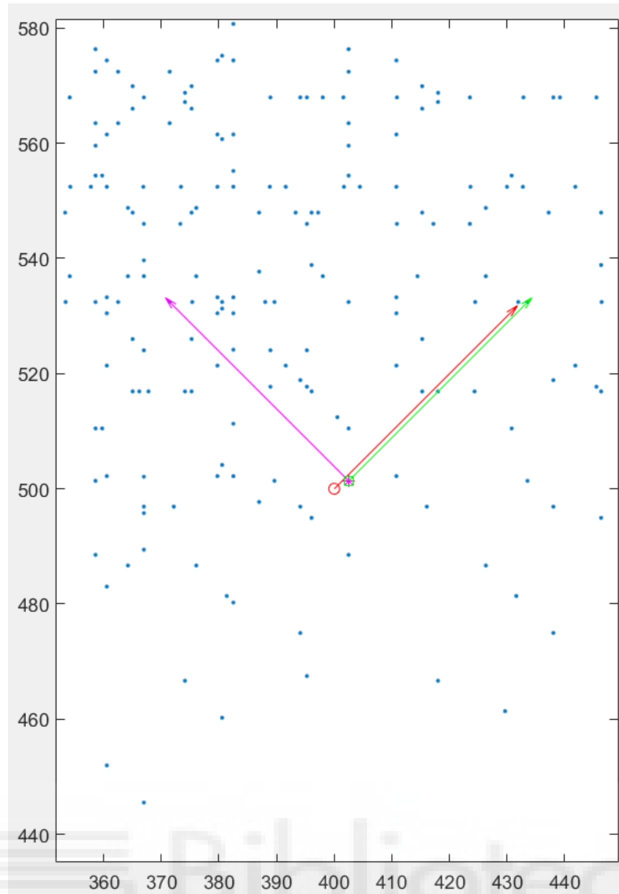


Figura 30: Ampliación de la Figura 29.

En este caso, el robot ha realizado 12 ciclos de movimiento hasta llegar al objetivo y otros 3 para orientarse, haciendo un total de 15 ciclos.

El segundo ejemplo se trata de una situación en la que existe un mínimo local, por lo que en este caso sí se ejecuta el algoritmo entero. Para ello, los obstáculos se deben situar en las posiciones (300, 300) y (200, 375). El código de colores para este caso es idéntico al del caso anterior.

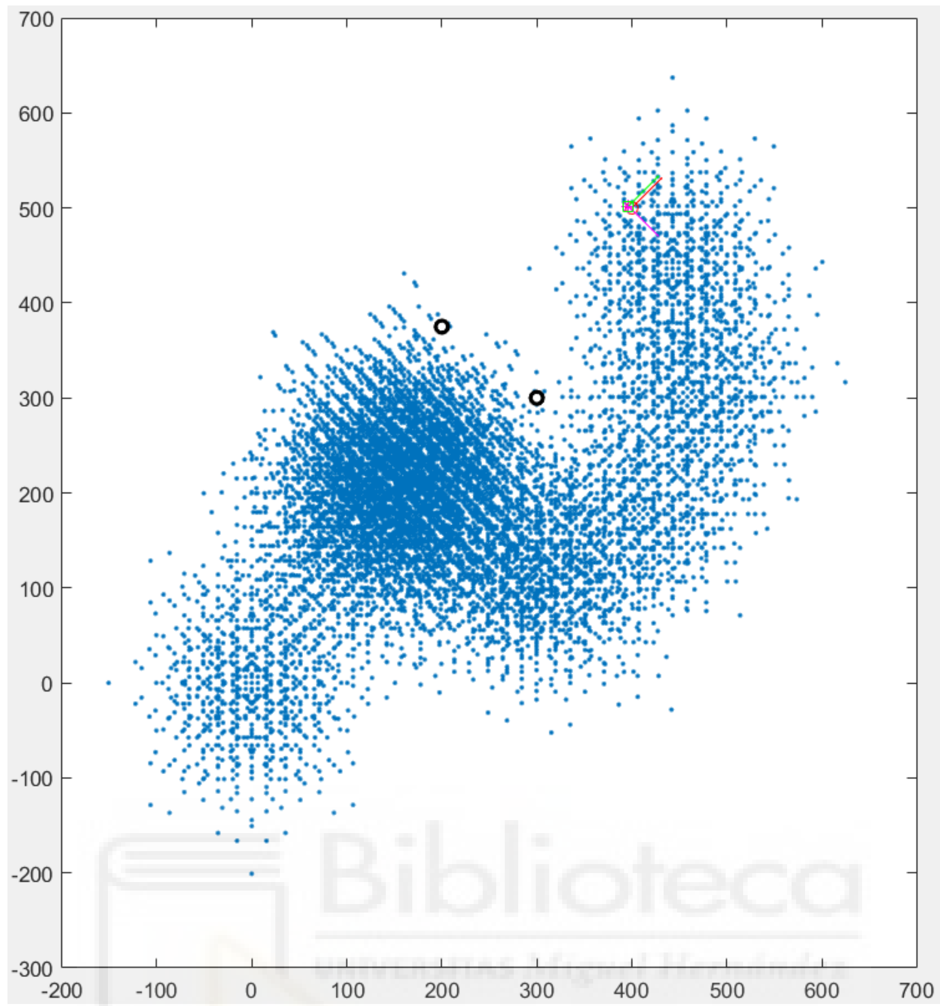


Figura 31: Solución con obstáculos con mínimo local.

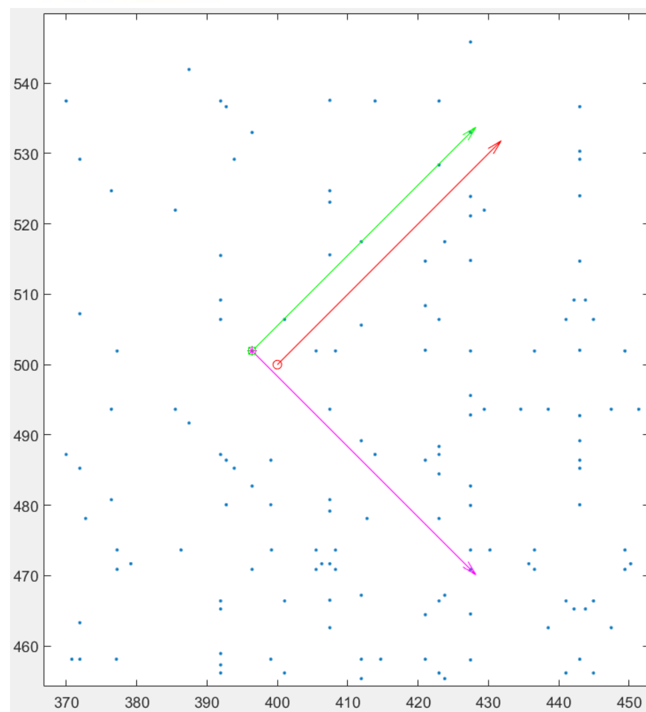


Figura 32: Ampliación de la Figura 31.

Como puede apreciarse claramente en la Figura 31, la nube de puntos se hace muy densa en la zona situada alrededor del punto (150, 200), lo que significa que el robot ha ido maniobrando por la zona hasta encontrar un camino que le permitiera continuar su trayectoria. En este caso, el robot ha realizado 26 ciclos de movimiento antes de la orientación y otros 3 para orientarse, en total 29. Se ha ejecutado el algoritmo varias veces y se han llegado a obtener valores de coste de entre 23 y 45 ciclos totales, y este número depende de los puntos aleatorios que genere el algoritmo, por lo que es difícil de optimizar.

En estos casos, hablar sobre a precisión del algoritmo en cuanto a posición y orientación resulta algo redundante, puesto que se sigue el mismo principio de precisión que en el capítulo 5, por lo que los valores de los errores van a ser semejantes en ambos casos, y en este algoritmo lo que se ha tratado de priorizar es la capacidad del robot para evitar obstáculos puntuales.

7.2. Obstáculos no puntuales.

Para concluir con la consideración de obstáculos en la planificación de trayectorias de este robot, se ha intentado implementar la opción de que el robot sea capaz de evitar obstáculos no puntuales. Esta implementación se ha decidido hacer mediante la discretización en nubes de puntos de los obstáculos que se quieren imponer.

Los *scripts* que se van a emplear para el desarrollo de este algoritmo se encuentran en el Anexo 7, y estos son: **ws_binario_mp.m**, el cual va a ser el único que reciba modificaciones con respecto a la sección 7.2 de este capítulo, y **ciclo_completo.m**, **encontrar_punto.m**, **calculo_phi_dif.m**, **ori_final.m** y **dibujo.m**, que se mantienen idénticos a sus versiones anteriores del Anexo 6.

La primera parte del código que se va a explicar es la generación de los obstáculos. Cabe destacar que mientras todos ellos se almacenen dentro de la matriz *P_obs*, no es necesario que solamente haya un elemento obstáculo, sino que puede haber varios, pero en este caso solamente se va a generar un elemento, que será una elipse, por razones de simplicidad y rendimiento.

```
19  -  %Obstáculos
20  -  P_obs = [];
21  -  resol_obs = 20;
22  -  for xobs = [400-300:resol_obs:400+300]
```

```
23  -     for yobs = [300-50:resol_obs:300+50]
24  -       if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
25  -         P_obs = [P_obs;[xobs,yobs]];
26  -       end
27  -     end
28  - end
```

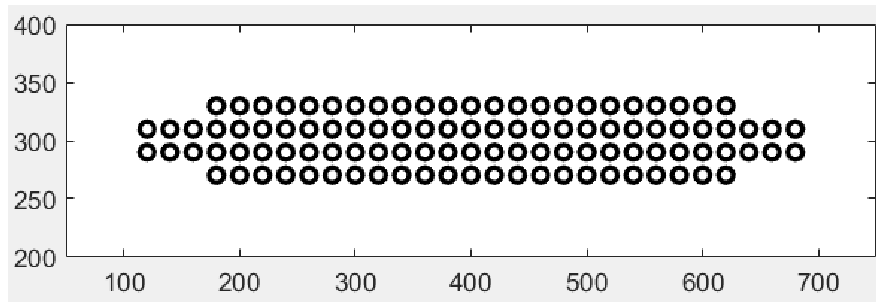


Figura 33: Obstáculo elíptico discretizado.

El obstáculo que se ha elegido para ejemplificar esta sección, como se ha comentado, es una elipse centrada en el punto (400,300), que se genera mediante los bucles anidados de las líneas 22-28. La línea 21 corresponde con la resolución de la discretización, es decir, con la cantidad de puntos que se van a representar. En este caso se va a tomar una resolución no demasiado buena, puesto que, como se ve en la Figura 33, la elipse no está del todo bien representada, pero es necesario para que el algoritmo trabaje de una forma mínimamente eficiente.

A continuación, se va a proceder a explicar los cambios realizados en **ws_binario_mp.m** para el desarrollo del algoritmo.

```
115 -           for h = 1:7
116 -             while 1
117 -               R = Rmax*rand;
118 -               angulo_hacia_destino=atan2(P_ref(2)-P0(2),P_ref(1)-P0(1));
119 -               if rand > 0.5 % Virar hacia la derecha
120 -                 angulo = angulo_hacia_destino - 3*pi/4 + pi/2*rand;
121 -               else % virar hacia la izquierda
122 -                 angulo = angulo_hacia_destino + 3*pi/4 + pi/2*rand;
123 -               end
124 -               x_p = P_mas_cercano(1) + R*cos(angulo);
125 -               y_p = P_mas_cercano(2) + R*sin(angulo);
126 -               P_ref_n = [x_p,y_p];
127
128 -               for d = 1:size(P_obs,1)
129 -                 if norm(P_ref_n-P_obs(d,:)) > radio_robot
130 -                   flag_conf_3 = 1;
```

```
131 -             else
132 -               flag_conf_3 = 0;
133 -               break;
134 -             end
135 -           end
136 -           if flag_conf_3 == 1
137 -             puntos_destino(h,:) = P_ref_n;
138 -             break;
139 -           end
140 -         end
141 -       end
```

Para este caso, es muy complicado para el robot salir del mínimo local buscando un punto aleatorio, por lo que en este caso se van a generar 7 puntos aleatorios y se va a buscar el que se encuentre lo más lejos posible del mínimo local y lo más cerca posible de la referencia. Para ello, las únicas diferencias que se han hecho en esta parte del código son que el bucle *while* que genera el punto aleatorio (líneas 116-140) se ejecute 7 veces (línea 115) y que se almacenen los puntos en una matriz (línea 137). El resto de este código es igual que en la versión de obstáculos puntuales.

```
143 -       l = length(puntos_destino);
144 -       dist_repulsor = zeros(l,1);
145 -       dist_ref = zeros(l,1);
146 -       dist_aux = zeros(l,1);
147 -       for k = 1:l
148 -         dist_repulsor(k) = norm(repulsor-puntos_destino(k,:));
149 -         dist_ref(k) = norm(P_ref-puntos_destino(k,:));
150 -         dist_aux(k) = dist_ref(k) - dist_repulsor(k);
151 -       end
152 -         [dist_def,ind_def] = min(dist_aux);
153 -         P_ref_n = puntos_destino(ind_def,:);
```

Lo siguiente es escoger el punto de referencia más correcto. Para ello, se escoge el punto que minimiza la diferencia entre la distancia desde el punto a la referencia y la distancia al mínimo local o repulsor. La línea que calcula la diferencia es la 150, y la que minimiza esa ecuación es la 152, y en la línea 153 se asigna el punto obtenido a la variable correspondiente.

A partir de este punto, el algoritmo continúa exactamente igual a su versión de la sección 7.1. El resultado de este no resulta satisfactorio, puesto que, a pesar de haber realizado este cambio, el robot no es capaz de llegar a su destino en todos los casos, y esto se debe a la aleatoriedad de la generación de los puntos. El algoritmo ha sido

Biblioteca UNIVERSITAS Miguel Hernández

ejecutado un número considerable de veces, y solamente una de ellas ha logrado un resultado correcto. Las siguientes figuras muestran de manera más visual el resultado obtenido.

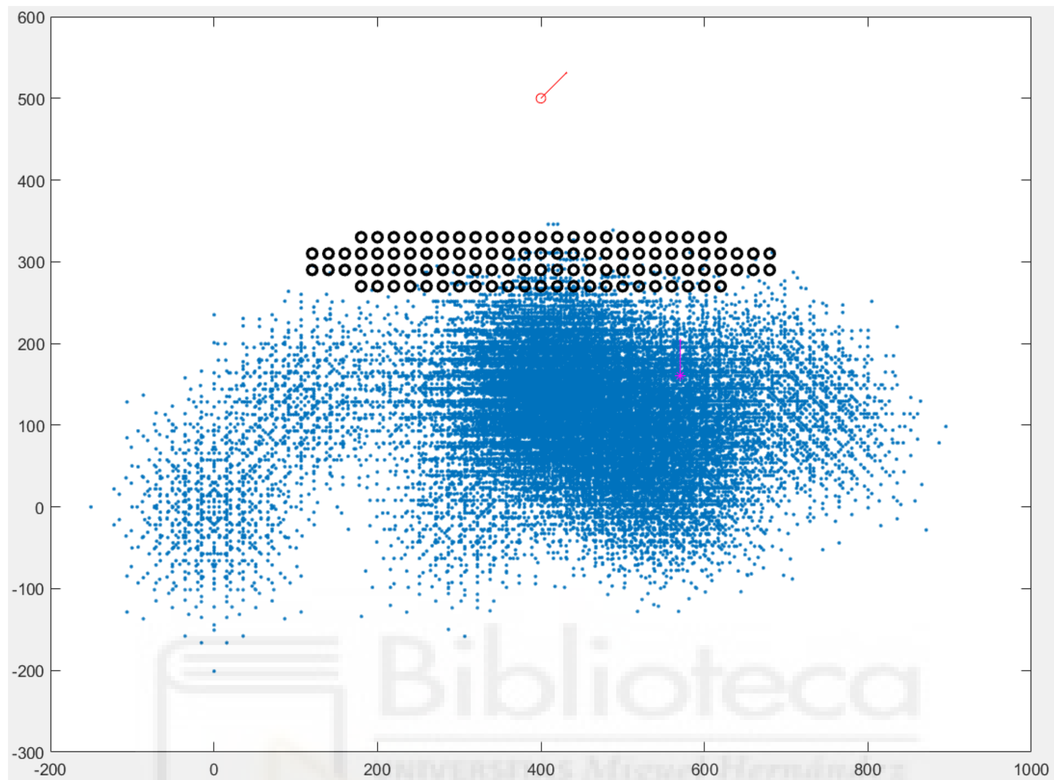


Figura 34: Resultado habitual del algoritmo de obstáculos no puntuales.

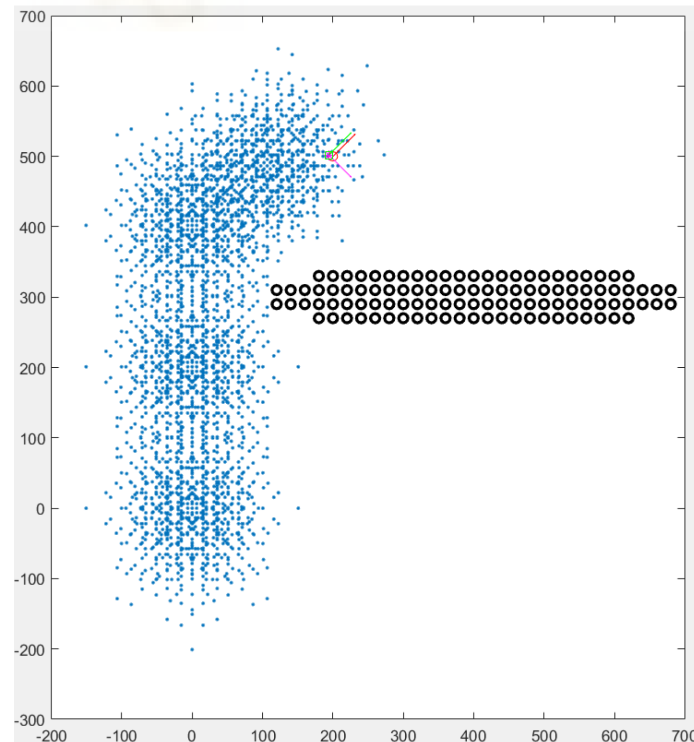


Figura 35: Resultado correcto sin atravesar el mínimo local.

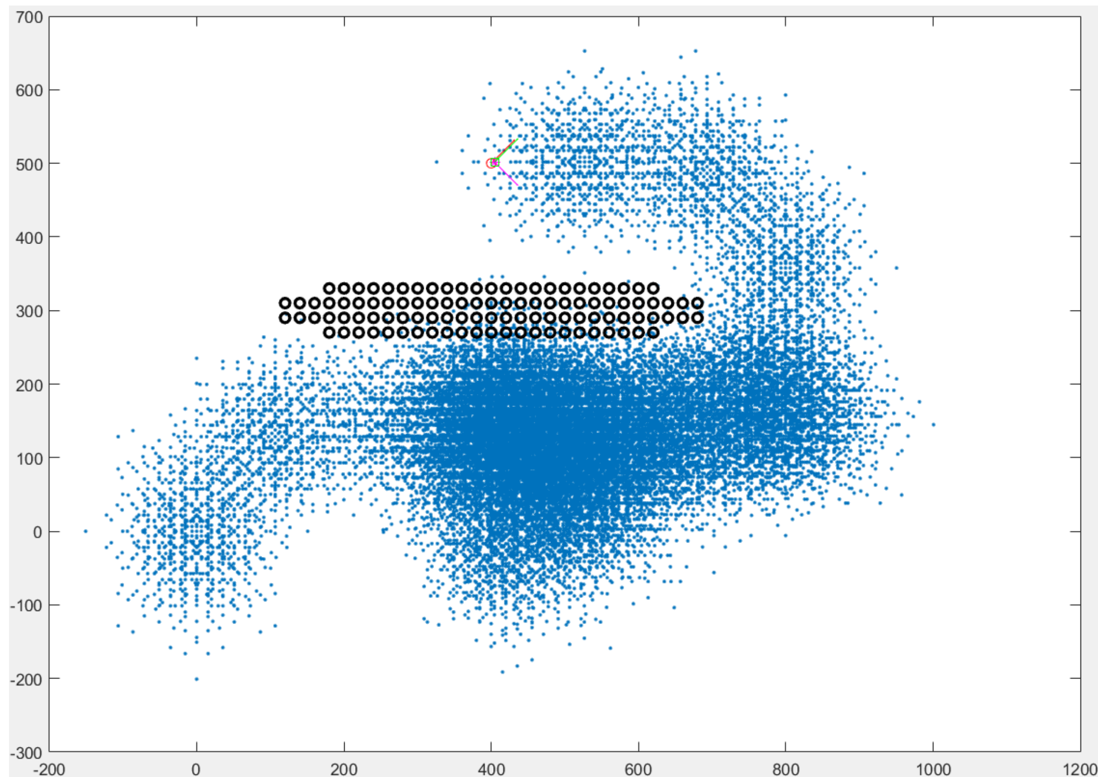


Figura 36: Único resultado correcto obtenido del algoritmo de obstáculos no puntuales atravesando el mínimo local.

Biblioteca UNIVERSITAS Miguel Hernández

El mínimo local del algoritmo siempre se encontrará, en este caso, en la vertical del punto de referencia por debajo del obstáculo. En la Figura 35 se muestra la solución del algoritmo cuando el robot no necesita atravesar el mínimo local. En este caso, el algoritmo funciona perfectamente y se ejecuta con una velocidad muy elevada, pero este es el caso más sencillo y favorable posible, por lo que a continuación trataremos el caso desfavorable.

En la Figura 34 se puede apreciar cuál es la situación habitual del algoritmo cuando el robot debe atravesar el mínimo local. Se ve a partir de la nube de puntos que el robot se mueve por debajo del obstáculo sin encontrar la salida del mínimo local. La razón de esto es, como ya se ha comentado, que la aleatoriedad de la generación de puntos no beneficia al algoritmo, puesto que lo único que debería de pasar es que, por ejemplo, se genere un punto por la zona de (900,350) y así el robot pueda alcanzar el objetivo a partir de ahí.

Tras realizar muchas pruebas de ejecución del algoritmo para comprobar si funcionaba o no, se ha conseguido un resultado correcto, que es el que se muestra en la Figura 36, el cual hace que, tras 121 ciclos de movimiento, el robot sea capaz de alcanzar

el objetivo. No es un resultado muy satisfactorio que, de unas 20 ejecuciones, solo una haya resultado ser correcta, pero al menos sirve para demostrar que el código está correctamente planteado, pero será necesario mejorarlo en trabajos futuros.

Biblioteca
UNIVERSITAS Miguel Hernández

Capítulo 8. Conclusiones y trabajos futuros.

Este Trabajo Fin de Grado se ha desarrollado desde lo más simple a lo más complejo. En primer lugar, se ha resuelto el problema de alcanzar la posición más cercana posible a una deseada dentro del alcance del espacio de trabajo del robot. Esta sección parece simple, pero resulta ser un punto de inflexión importante, puesto que la mayor parte de algoritmos desarrollados en este Trabajo Fin de Grado para la planificación de trayectorias de este robot se basa en buscar los puntos del espacio de trabajo que cumplen las condiciones correspondientes en cada caso.

Posteriormente, se ha resuelto el mismo problema planteado anteriormente, pero teniendo en cuenta no solo la posición sino también la orientación, para lo que se han ideado dos métodos de resolución. En este apartado, el método de selección de punto óptimo (desarrollado en la sección 3.1) ha resultado ser especialmente bueno en cuanto a su fácil parametrización, ya que en este se permite definir de manera sencilla los errores máximos permitidos en posición y orientación.

A continuación, se ha resuelto la planificación de trayectorias para cualquier posición del espacio 2D libre de obstáculos, y para ello se ha empleado un método de solapamiento de espacios de trabajo, a lo que después se le incluyó que el robot fuera capaz de orientarse una vez alcanzado el objetivo. Estos apartados resultaron tener una precisión, además de parametrizable, muy elevada; es más, en caso de que N (número de ciclos del espacio de trabajo que se solapa) sea igual a 3 en lugar de 2 como en el caso mostrado, la precisión podría llegar a considerarse casi perfecta, teniendo un error máximo de aproximadamente 0.2 mm tal y como se muestra en la Figura 16.

Con el fin de validar los anteriores cálculos y verificar que el robot realiza las trayectorias de forma correcta, se ha elaborado un código que genera una animación que representa el movimiento del robot a lo largo del espacio. Este código ha resultado ser realmente útil para la comprensión del movimiento del robot y la comprobación de posibles errores.

Finalmente, se ha incluido un capítulo que se dedica al estudio de posibles obstáculos en la trayectoria del robot. En una primera instancia, se pretendía simplemente que el robot rodeara los obstáculos que se le plantearan para llegar a su objetivo, pero se han encontrado situaciones en las que existían mínimos locales en los cuales se atascaba

el robot y de los cuales este no podía salir. Se ha optado por que el robot escape de esos mínimos locales mediante la generación de puntos aleatorios a los que enviar al robot antes de volver a dirigirse a su destino, pero en el caso en el que los obstáculos no son puntuales, este método no ha resultado ser satisfactorio, puesto que la probabilidad de que se genere un punto aleatorio que permita al robot escapar del mínimo local de forma efectiva es baja, según han mostrado los experimentos. Este resultado correcto demuestra que el algoritmo es factible, pero también ofrece margen de mejora, para facilitar el escape eficiente de mínimos locales.

En lo referente a trabajos futuros, el primero que se propone es el mencionado en el capítulo 4, el cual consistía en hacer un algoritmo que sea capaz de comparar los métodos expuestos en el mismo de forma precisa y con un criterio correcto. Los métodos mencionados son el solapamiento de espacios de trabajo y el incremento del valor de N indefinidamente hasta que el robot fuera capaz de alcanzar el objetivo impuesto. Más concretamente, la función de este algoritmo debería encargarse de encontrar una forma de saber cuántos ciclos de movimiento haría el robot para alcanzar el objetivo mediante el segundo método, puesto que el primer método ya da un resultado de la cantidad de ciclos realizados para comparar con el resultado del algoritmo mencionado.

El segundo trabajo futuro que se propone consiste en la elaboración de un algoritmo que se encargue de optimizar el código de la sección 7.1 del capítulo 7, que se encarga de realizar la planificación de trayectorias del robot para espacios con obstáculos puntuales. El objetivo de esta optimización sería el de poder realizar los cálculos de trayectoria sin tener que mayorar el radio de la circunferencia circunscrita al robot como se hace en el algoritmo planteado en este Trabajo Fin de Grado. Se plantea como posible alternativa la de comprobar que el robot no está colisionando en cada instante del movimiento, pero este método resulta ser demasiado exigente para el ordenador, por ello no se ha implementado en este proyecto.

Como último trabajo futuro, se propone adaptar el código presentado sobre obstáculos no puntuales en la sección 7.2 del capítulo 7. Dicha adaptación consistiría en, en primer lugar, mejorar el método de escapar de los mínimos locales para lograr una mayor tasa de éxito que con el método actual. Además, también se debería añadir a esta adaptación el hecho de poder comprobar la colisión del robot con el obstáculo utilizando

un método más eficiente que el de comprobar cada uno de los puntos de la nube en cada iteración, para lo que se propone usar KD-trees.

Biblioteca
UNIVERSITAS Miguel Hernández

ANEXOS

Anexo 0

Este anexo recopila el código del que se parte para la realización de este Trabajo Fin de Grado, explicado en la sección 1.4.

ws_binario_mp.m

```
1  - close all;
2  - clear all;
3
4  - global nube
5  - global global_counter
6
7  - N = 2; % 1 a 3 (no más)
8
9  - nube = zeros(64^N,3);
10 - global_counter = 0;
11
12   % Depende de: b, p, rho0, Delta_rho
13 - phi = [pi,3*pi/4,pi/2,3*pi/4,pi,pi+pi/4,pi+pi/2,pi+pi/4];
14 - y = [5.452580023,2.472051996,0,-2.472051996,-5.452580023,2.472051996,
15 -      0,2.472051996];
16
17 - for i=1:8
18 -     T{i}=[cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
19 - end
20
21 - ciclo_completo([0,0],0,T,1,N)
22
23 - nube = nube(1:global_counter,:);
24 - plot(nube(:,1),nube(:,2),'.');
25 - set(gca,'DataAspectRatio',[1,1,1]);
26 - hold on
27 - quiver(nube(:,1),nube(:,2),-sin(nube(:,3)),cos(nube(:,3)));
```

ciclo_completo.m

```
1  - function ciclo_completo(pos_0,ori_0,T,n,N)
2
3  - global nube
4  - global global_counter
5
6  - if n<=N
7
8  -     TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
9  -             sin(ori_0), cos(ori_0),pos_0(2);
10 -                  0     ,     0      ,    1    ];
11
12       % Fijo A, muevo B (medio ciclo)
13 -     for i=1:8
14 -         TB = TA_0*T{i};
15           % Fijo B, muevo A (el otro medio ciclo)
16 -         for j=1:8
17                   % Obtenemos posición y orientación de la matriz de
18                   % transformación en el nuevo punto, y los almacenamos
19                   % en la variable nube
20 -             if i~=j
21 -                 TA = TB*inv(T{j});
22 -                 pos = TA(1:2,3)';
23 -                 ori = atan2(TA(2,1),TA(1,1));
24 -                 global_counter = global_counter + 1;
25 -                 nube(global_counter,:) = [pos,ori];
26 -                 ciclo_completo(pos,ori,T,n+1,N);
27 -             end
28 -         end
29 -     end
30
31 - end
32
33 - end
```

Anexo 1

Este anexo recopila el código explicado en el capítulo 2 en el que se pretende resolver el problema de encontrar la posición más cercana a la deseada dentro del espacio alcanzable por el espacio de trabajo del robot.

ws_binario_mp.m

```
1  - clear all;
2
3  - global nube
4  - global global_counter
5  - global secuencia
6
7  - N = 2; % 1 a 3 (no más)
8  - P_ref = [50 120];
9
10 - nube = zeros(64^N,3);
11 - global_counter = 0;
12 - secuencia = (-1)*ones(64^N,2*N);
13 - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
14
15   % Depende de: b, p, rho0, Delta_rho -> Se ha cambiado a mm
16 - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
17 - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,
18 -      0,21.95478428];
19
20 - for i=1:8
21 -     T{i} = [cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
22 - end
23
24   % Script que calcula el espacio de trabajo
25 - ciclo_completo([0,0],0,T,1,N)
26
27   % Script que encuentra el punto deseado
28 - encontrar_punto
```

ciclo_completo.m

```
1   - function ciclo_completo(pos_0,ori_0,T,n,N)
2
3   - global nube
4   - global global_counter
5   - global secuencia
6
7   - v_i=-1*ones(1,N);
8   - v_j=-1*ones(1,N);
9
10  - if n<=N
11
12        % Cálculo de matriz de transformación
13  -     TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
14  -             sin(ori_0), cos(ori_0),pos_0(2);
15  -                  0 ,     0      ,     1    ];
16
17        % Fijo A, muevo B (medio ciclo)
18  -     for i=1:8
19  -         TB = TA_0*T{i};
20            % Fijo B, muevo A (el otro medio ciclo)
21  -         for j=1:8
22                % Obtenemos posición y orientación y los almacenamos en nube
23  -             TA = TB*inv(T{j});
24  -             pos = TA(1:2,3)';
25  -             ori = atan2(TA(2,1),TA(1,1));
26  -             global_counter = global_counter + 1;
27  -             nube(global_counter,:) = [pos,ori];
28                % Almacenamos los valores de i y j en la variable secuencia
29  -             if n==1
30  -                 v_i(1)=i;
31  -                 v_j(1)=j;
32  -                 secuencia(global_counter,:) = [v_i v_j];
33  -             else
34  -                 Auxiliar = secuencia(global_counter-1,:);
35  -                 auxiliar(n)=i;
36  -                 auxiliar(N+n)=j;
37  -                 secuencia(global_counter,:) = auxiliar;
38  -             end
39                % Se trata de una función recursiva
40  -             ciclo_completo(pos,ori,T,n+1,N);
41  -         end
42  -     end
43  -     nube = nube(1:global_counter,:);
44  -     secuencia = secuencia(1:global_counter,:);
45  - end
46  - end
```

encontrar_punto.m

```
1   -   close all
2
3   -   plot(nube(:,1),nube(:,2),'.');
4   -   set(gca,'DataAspectRatio',[1,1,1]);
5   -   hold on
6   -   plot(P_ref(1),P_ref(2),'or');
7       % quiver(nube(:,1),nube(:,2),cos(nube(:,3)),sin(nube(:,3)));
8
9   -   distancia_minima = 1000000;
10  -   indice_min = -1;
11  -   P_mas_cercano = [0,0];
12  -   T_encontrado = eye(3);
13
14      % Buscamos el punto más cercano al de referencia
15  -   for i=1:size(nube,1)
16  -       distancia = norm(P_ref-nube(i,1:2));
17  -       if distancia < distancia_minima
18  -           indice_min = i;
19  -           distancia_minima = distancia;
20  -           P_mas_cercano = nube(i,1:2);
21  -       end
22  -   end
23
24      % Sacamos la secuencia de i y j del punto encontrado
25  -   secuencia_encontrada = secuencia(indice_min,:);
26
27      % Calculamos el punto elegido mediante su secuencia y lo dibujamos
28  -   for k=1:N
29  -       i = secuencia_encontrada(k);
30  -       j = secuencia_encontrada(k+N);
31  -       if i==-1 || j==-1
32  -           continue
33  -       end
34  -       T_encontrado = T_encontrado*T{i}*inv(T{j});
35  -   end
36
37  -   plot(T_encontrado(1,3),T_encontrado(2,3),'sg');
38  -   plot(P_mas_cercano(1),P_mas_cercano(2),'*m');
```

Anexo 2

Este anexo se encarga de recopilar el código empleado en el capítulo 3 para resolver el problema de la búsqueda de posición y orientación más óptimas con respecto a las referencias de posición y orientación predefinidas mediante dos métodos diferentes.

ws_binario_mp.m

```
1   - clear all;
2
3   - global nube
4   - global global_counter
5   - global secuencia
6
7   - N = 2; % 1 a 3 (no más)
8
9     % Posición y orientación deseadas
10  - P_ref = [0 0];
11  - phi_ref = pi/2;
12
13  - nube = zeros(64^N,3);
14  - global_counter = 0;
15  - secuencia = (-1)*ones(64^N,2*N);
16  - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
17
18    % Depende de: b, p, rho0, Delta_rho
19  - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
20  - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,
21  -      0,21.95478428];
22
23  - for i=1:8
         T{i}=[cos(phi(i)),-
24  - sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
25  - end
26
27    % Script que calcula el espacio de trabajo
28  - ciclo_completo([0,0],0,T,1,N)
29
30    % Script que encuentra el punto deseado
31  - encontrar_punto
```

ciclo_completo.m

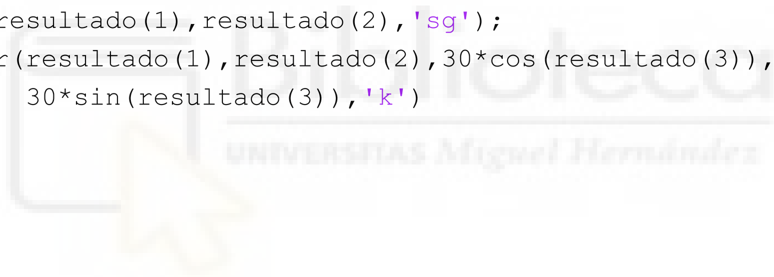
```
1  - function ciclo_completo(pos_0,ori_0,T,n,N)
2
3  - global nube
4  - global global_counter
5  - global secuencia
6
7  - v_i=-1*ones(1,N);
8  - v_j=-1*ones(1,N);
9
10 - if n<=N
11
12       % Cálculo de matriz de transformación
13 -     TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
14 -             sin(ori_0), cos(ori_0),pos_0(2);
15 -                  0 ,     0      ,     1    ];
16
17       % Fijo A, muevo B (medio ciclo)
18 -     for i=1:8
19 -         TB = TA_0*T{i};
20           % Fijo B, muevo A (el otro medio ciclo)
21 -         for j=1:8
22               % Obtenemos posición y orientación y los almacenamos en nube
23 -             TA = TB*inv(T{j});
24 -             pos = TA(1:2,3)';
25 -             ori = atan2(TA(2,1),TA(1,1));
26 -             global_counter = global_counter + 1;
27 -             nube(global_counter,:) = [pos,ori];
28               % Almacenamos los valores de i y j en la variable secuencia
29 -             if n==1
30 -                 v_i(1)=i;
31 -                 v_j(1)=j;
32 -                 secuencia(global_counter,:) = [v_i v_j];
33 -             else
34 -                 auxiliar=secuencia(global_counter-1,:);
35 -                 auxiliar(n)=i;
36 -                 auxiliar(N+n)=j;
37 -                 secuencia(global_counter,:) = auxiliar;
38 -             end
39               % Se trata de una función recursiva
40 -             ciclo_completo(pos,ori,T,n+1,N);
41 -         end
42 -     end
43 -     nube = nube(1:global_counter,:);
44 -     secuencia = secuencia(1:global_counter,:);
45 - end
46 - end
```

encontrar_punto.m

```
1   - close all
2
3   - v_distancia = zeros(size(nube,1),1);
4   - res_aux = [];
5   - dist_aux = [];
6   - ind_aux = [];
7   - cont = 0;
8   - radio = 20; % Radio de la circunferencia de puntos admisibles
9   - T_encontrado = eye(3);
10
11  - plot(nube(:,1),nube(:,2),'.');
12  - set(gca,'DataAspectRatio',[1,1,1]);
13  - hold on
14  - plot(P_ref(1),P_ref(2),'or');
15  - quiver(P_ref(1),P_ref(2),30*cos(phi_ref),30*sin(phi_ref),'r');
16
17    % Cálculo del vector de distancias
18  - for u = 1:size(nube,1)
19  -     v_distancia(u) = norm(P_ref-nube(u,1:2));
20  - end
21
22    % Cálculo del punto considerando la orientación
23  - while 1
24  -     cont=cont+1;
25  -     [minimo,indice_min] = min(v_distancia);
26  -     resultado = nube(indice_min,:);
27  -     phi_dif = calculo_phi_dif(phi_ref,resultado);
28  -     if v_distancia(indice_min) < radio abs(phi_dif) <= pi/4
29  -         if abs(phi_dif) <= pi/4
30  -             break
31  -         else
32  -             res_aux = [res_aux;resultado];
33  -             dist_aux = [dist_aux;minimo];
34  -             ind_aux = [ind_aux;indice_min];
35  -             v_distancia(indice_min) = 9999;
36          end
37  -     else
38  -         if cont==1
39  -             break
40  -         else
41  -             [tmp,indice_tmp] = min(dist_aux);
42  -             resultado = res_aux(indice_tmp);
43  -             indice_min = ind_aux(indice_tmp);
44  -             break
45  -         end
46  -     end
```

Biblioteca UNIVERSITAS Miguel Hernández

```
47 - end
48
49 - secuencia_encontrada = secuencia(indice_min,:);
50
51    % Dibujo de la circunferencia de puntos admisibles
52 - th = 0:pi/50:2*pi;
53 - xunit = radio*cos(th)+P_ref(1);
54 - yunit = radio*sin(th)+P_ref(2);
55 - h = plot(xunit,yunit);
56
57    % Utilizamos la secuencia obtenida para hallar el punto correcto
58 - for k=1:N
59 -     i = secuencia_encontrada(k);
60 -     j = secuencia_encontrada(k+N);
61 -     if i==-1 || j==-1
62 -         continue
63 -     end
64 -     T_encontrado = T_encontrado*T{i}*inv(T{j});
65 - end
66
67 - plot(T_encontrado(1,3),T_encontrado(2,3),'*m');
68 - plot(resultado(1),resultado(2),'sg');
69 - quiver(resultado(1),resultado(2),30*cos(resultado(3)),
70 -        30*sin(resultado(3)),'k')
```



calculo_phi_dif.m

```
1  - function phi_dif = calculo_phi_dif(phi_ref,resultado)
2
3  - phi_dif_tmp = phi_ref-resultado(3);
4
5  - if phi_dif_tmp > pi
6  -     phi_dif_tmp = phi_dif_tmp-2*pi;
7  - end
8  - if phi_dif_tmp < -pi
9  -     phi_dif_tmp = phi_dif_tmp+2*pi;
10 - end
11
12 - phi_dif = abs(phi_dif_tmp);
13
14 - end
```

computación_distancia_w.m

```
1   - w=10;
2   - P_ref = [90 70];
3   - phi_ref = pi/2;
4
5   - registro_eq = [];
6
7   - for f=1:size(nube,1)
8   -     Pos_actual = nube(f,1:2);
9   -     resultado = nube(f,:);
10  -     phi_dif = calculo_phi_dif(phi_ref,resultado);
11  -     eq = norm(P_ref-Pos_actual) + w*abs(phi_dif);
12  -     registro_eq = [registro_eq; eq];
13  - end
14
15    % Se busca el mínimo
16  - [min_eq,ind_eq] = min(registro_eq);
17
18  - plot(nube(:,1),nube(:,2),'.');
19  - hold on
20  - set(gca,'DataAspectRatio',[1,1,1]);
21  - plot(P_ref(1),P_ref(2),'or');
22  - plot(nube(ind_eq,1),nube(ind_eq,2),'*m');
```

Biblioteca UNIVERSITAS Miguel Hernández

Anexo 3

Este anexo recopila el código correspondiente al algoritmo desarrollado en el capítulo 4, el cual se encarga de la búsqueda del punto más cercano al de referencia, pero en este caso con la posibilidad de que este se encuentre en cualquier punto del espacio 2D libre de obstáculos.

ws_binario_mp.m

```
1   - clear all;
2
3   - global nube
4   - global global_counter
5   - global secuencia
6
7   - N = 2; % 1 a 3 (no más)
8   - P_ref = [500 800];
9
10  - nube = zeros(64^N,3);
11  - global_counter = 0;
12  - secuencia = (-1)*ones(64^N,2*N);
13  - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
14  - registro_seq = [];
15
16    % Depende de: b, p, rho0, Delta_rho
17  - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
18  - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,
19  -         0,21.95478428];
20
21  - for i=1:8
22  -     T{i}=[cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
23  - end
24
25  - P0 = [0,0];
26  - phi0 = 0;
27
28  - while 1
29  -     if norm(P_ref-P0) < 10
30  -         break
31  -     end
32        % Script que calcula el espacio de trabajo
33  -     ciclo_completo(P0,phi0,T,1,N)
34        % Script que encuentra el punto deseado
35  -     encontrar_punto
```

```
36        % Actualización de condiciones iniciales
37 -      P0 = P_mas_cercano;
38 -      phi0 = nube(indice_min,3);
39 - end
40
41 - plot(P_mas_cercano(1),P_mas_cercano(2),'*m');
42   % quiver(nube(:,1),nube(:,2),cos(nube(:,3)),sin(nube(:,3)));
```

ciclo_completo.m

```
1  - function ciclo_completo(pos_0,ori_0,T,n,N)
2
3  - global nube
4  - global global_counter
5  - global secuencia
6
7  - v_i=-1*ones(1,N);
8  - v_j=-1*ones(1,N);
9
10 - if n<=N
11 -
12       % Cálculo de matriz de transformación
13 -     TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
14 -             sin(ori_0), cos(ori_0),pos_0(2);
15 -                  0 ,     0      ,     1    ];
16
17       % Fijo A, muevo B (medio ciclo)
18 -     for i=1:8
19 -         TB = TA_0*T{i};
20           % Fijo B, muevo A (el otro medio ciclo)
21 -         for j=1:8
22               % Obtenemos posición y orientación y los almacenamos en nube
23 -             TA = TB*inv(T{j});
24 -             pos = TA(1:2,3)';
25 -             ori = atan2(TA(2,1),TA(1,1));
26 -             global_counter = global_counter + 1;
27 -             nube(global_counter,:) = [pos,ori]; % [ x y phi ];
28               % Almacenamos los valores de i y j en la variable secuencia
29 -             if n==1
30 -                 v_i(1)=i;
31 -                 v_j(1)=j;
32 -                 secuencia(global_counter,:) = [v_i v_j];
33 -             else
34 -                 auxiliar=secuencia(global_counter-1,:);
35 -                 auxiliar(n)=i;
36 -                 auxiliar(N+n)=j;
37 -                 secuencia(global_counter,:) = auxiliar;
38 -             end
39 -             ciclo_completo(pos,ori,T,n+1,N);
40 -         end
41 -     end
42
43 -     nube = nube(1:global_counter,:);
44 -     secuencia = secuencia(1:global_counter,:);
45 - end
46 - end
```

encontrar_punto.m

```
1   - close all
2
3   - plot(nube(:,1),nube(:,2),'.');
4   - set(gca,'DataAspectRatio',[1,1,1]);
5   - hold on
6   - plot(P_ref(1),P_ref(2),'or');
7     % quiver(nube(:,1),nube(:,2),-sin(nube(:,3)),cos(nube(:,3)));
8
9   - distancia_minima = 1000000;
10  - indice_min = -1;
11  - P_mas_cercano = [0,0];
12  - T_encontrado = eye(3);
13
14    % Buscamos el punto más cercano al de referencia
15  - for i=1:size(nube,1)
16  -     distancia = norm(P_ref-nube(i,1:2));
17  -     if distancia < distancia_minima
18  -         indice_min = i;
19  -         distancia_minima = distancia;
20  -         P_mas_cercano = nube(i,1:2);
21  -     end
22  - end
23
24    % Sacamos la secuencia de i y j del punto encontrado
25  - secuencia_encontrada = secuencia(indice_min,:);
26  - registro_seq = [registro_seq; secuencia_encontrada];
```

Biblioteca UNIVERSITAS Miguel Hernández

Anexo 4

Este anexo recopila el código explicado en el capítulo 5 que se encarga de buscar el punto más cercano al de referencia a lo largo del espacio 2D libre de obstáculos, y, una vez alcanzado este punto, orientar al robot según se le imponga.

ws_binario_mp.m

```
1  -  clear all;
2
3  -  global nube
4  -  global global_counter
5  -  global secuencia
6
7  -  N = 2; % 1 a 3 (no más)
8  -  P_ref = [400 500];
9  -  phi_ref = pi/4;
10
11 -  nube = zeros(64^N,3);
12 -  global_counter = 0;
13 -  secuencia = (-1)*ones(64^N,2*N);
14 -  T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
15 -  registro_seq = [];
16
17    % Depende de: b, p, rho0, Delta_rho
18 -  phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
19 -  y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,
20 -       0,21.95478428];
21
22 -  for i=1:8
23 -      T{i}=[cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
24 -  end
25
26 -  P0 = [0,0];
27 -  phi0 = 0;
28
29 -  while 1
30 -      if norm(P_ref-P0) < 10
31 -          break
32 -      end
33        % Script que calcula el espacio de trabajo
34 -      ciclo_completo(P0,phi0,T,1,N)
35        % Script que encuentra el punto deseado
36 -      encontrar_punto
37        % Actualización de condiciones iniciales
```

```
38 -       P0 = P_mas_cercano;
39 -       phi0 = nube(indice_min,3);
40 -   end
41
42 -   phi_pi=0;
43
44 -   if phi_ref == pi || phi_ref == -pi
45 -       if phi0+phi_ref < 0.001
46 -           phi_pi = 1;
47 -       else
48 -           phi_pi = 0;
49 -       end
50 -   end
51
52 -   if phi0 ~= phi_ref && phi_pi == 0
53 -       flag = 1;
54 -       ori_final
55 -       dibujo
56 -   elseif phi0 == phi_ref || phi_pi == 1
57 -       flag = 0;
58 -       dibujo
59 -   end
```

ciclo_completo.m

```
 1 - function ciclo_completo(pos_0,ori_0,T,n,N)
 2
 3 - global nube
 4 - global global_counter
 5 - global secuencia
 6
 7 - v_i=-1*ones(1,N);
 8 - v_j=-1*ones(1,N);
 9
10 - if n<=N
11
12         % Cálculo de matriz de transformación
13 -       TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
14 -               sin(ori_0), cos(ori_0),pos_0(2);
15 -                    0 ,     0      ,     1    ];
16
17         % Fijo A, muevo B (medio ciclo)
18 -       for i=1:8
19 -           TB = TA_0*T{i};
20             % Fijo B, muevo A (el otro medio ciclo)
21 -           for j=1:8
22                 % Obtenemos posición y orientación y los almacenamos en nube
23 -               TA = TB*inv(T{j});
24 -               pos = TA(1:2,3)';
25 -               ori = atan2(TA(2,1),TA(1,1));
26 -               global_counter = global_counter + 1;
27 -               nube(global_counter,:) = [pos,ori]; % [ x y phi ];
28                 % Almacenamos los valores de i y j en la variable secuencia
29 -               if n==1
30 -                   v_i(1)=i;
31 -                   v_j(1)=j;
32 -                   secuencia(global_counter,:) = [v_i v_j];
33 -               else
34 -                   auxiliar=secuencia(global_counter-1,:);
35 -                   auxiliar(n)=i;
36 -                   auxiliar(N+n)=j;
37 -                   secuencia(global_counter,:) = auxiliar;
38 -               end
39 -               ciclo_completo(pos,ori,T,n+1,N);
40 -           end
41 -       end
42
43 -       nube = nube(1:global_counter,:);
44 -       secuencia = secuencia(1:global_counter,:);
45 - end
46 - end
```

encontrar_punto.m

```
1   - close all
2
3   - distancia_minima = 1000000;
4   - indice_min = -1;
5   - P_mas_cercano = [0,0];
6
7     % Buscamos el punto más cercano al de referencia
8   - for i=1:size(nube,1)
9   -     distancia = norm(P_ref-nube(i,1:2));
10  -     if distancia < distancia_minima
11  -         indice_min = i;
12  -         distancia_minima = distancia;
13  -         P_mas_cercano = nube(i,1:2);
14  -     end
15  - end
16
17    % Sacamos la secuencia de i y j del punto encontrado
18  - secuencia_encontrada = secuencia(indice_min,:);
19  - registro_seq = [registro_seq; secuencia_encontrada];
```

calculo_phi_dif.m

```
1  - function phi_dif = calculo_phi_dif(phi_ref,resultado)
2
3  -     phi_dif_tmp = phi_ref-resultado(3);
4
5  -     if phi_dif_tmp > pi
6  -         phi_dif_tmp = phi_dif_tmp-2*pi;
7  -     end
8  -     if phi_dif_tmp < -pi
9  -         phi_dif_tmp = phi_dif_tmp+2*pi;
10 -     end
11
12 -     phi_dif = abs(phi_dif_tmp);
13
14 - end
```

Biblioteca
UNIVERSITAS Miguel Hernández

ori_final.m

```
 1 - v_indices = [];
 2 - v_dist = [];
 3 - T_encontrado = eye(3);
 4
 5 - % Puntos más favorables en posición
 6 - puntos_seq = {  [0 0], 0, [1 -1 -1 1 -1 -1];
 7 -                 [0 0], pi/2, [1 7 8 1 2 7];
 8 -                 [0 0], pi, [1 1 3 1 1 7];
 9 -                 [0 0], -pi/2, [1 3 4 1 6 3];
10 -                 [0.151 0.151], pi/4, [5 1 5 3 2 7];
11 -                 [0.151 0.151], -3*pi/4, [5 1 5 3 2 3];
12 -                 [0.151 -0.151], 3*pi/4, [1 1 5 3 8 3];
13 -                 [0.151 -0.151], -pi/4, [1 1 5 3 8 7];
14 -                 [-0.151 0.151], 3*pi/4, [5 5 1 3 6 3];
15 -                 [-0.151 0.151], -pi/4, [5 5 1 3 6 7];
16 -                 [-0.151 -0.151], pi/4, [1 5 1 3 4 7];
17 -                 [-0.151 -0.151], -3*pi/4, [1 5 1 3 4 3];
18 -                 [0.2135 0], pi/4, [3 8 7 5 1 5];
19 -                 [0.2135 0] 3*pi/4 [3 2 3 5 1 5];
20 -                 [0.2135 0] -pi/4 [3 2 7 5 1 1];
21 -                 [0.2135 0] -3*pi/4 [3 8 3 5 1 1];
22 -                 [-0.2135 0] pi/4 [3 6 7 1 5 1];
23 -                 [-0.2135 0] 3*pi/4 [3 4 3 1 5 1];
24 -                 [-0.2135 0] -pi/4 [3 4 7 1 5 5];
25 -                 [-0.2135 0] -3*pi/4 [3 6 3 1 5 5]};
26
27   % Cálculo de la diferencia de ángulos
28 - phi_dif = phi_ref-nube(indice_min,3);
29 - if phi_dif > pi
30 -     phi_dif = phi_dif-2*pi;
31 - elseif phi_dif < -pi
32 -     phi_dif = phi_dif+2*pi;
33 - end
34
35   % Puntos que coinciden en orientación con la de referencia
36 - for x = 1:20
37 -     if abs(puntos_seq{x,2}-phi_dif) < 0.001
38 -         v_indices = [v_indices;x];
39 -     end
40 - end
41
42   % Cálculo de la distancia entre los seleccionados antes y la referencia
43 - for y = 1:size(v_indices)
44 -     v_aux = puntos_seq{v_indices(y),1};
45 -     theta = atan2(v_aux(2),v_aux(1));
46 -     phi_dif_aux = theta+nube(indice_min,3);
```

```
47 -     mod = norm(v_aux);
48
49 -     P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
50                P_mas_cercano(2)+mod*sin(phi_dif_aux)];
51 -     dist = norm(P_aux-P_ref);
52 -     v_dist = [v_dist;dist];
53 - end
54
55 - [min_tmp,ind_tmp] = min(v_dist);
56 - secuencia_final = puntos_seq{v_indices(ind_tmp),3};
57
58   % Obtención del punto con la secuencia correspondiente
59 - for k=1:3
60 -     i = secuencia_final(k);
61 -     j = secuencia_final(k+3);
62 -     if i==-1 || j==-1
63 -         continue
64 -     end
65 -     T_encontrado = T_encontrado*T{i}*inv(T{j});
66 - end
67
68 - ang = nube(indice_min,3)
69 - x = nube(indice_min,1)
70 - y = nube(indice_min,2);
71 - T0 = [ cos(ang) -sin(ang)  x;
72 -        sin(ang)  cos(ang)  y;
73 -           0         0      1];
74
75 - T1 = T0*T_encontrado;
76 - P_final = T1(1:2,3)';
77 - phi_final = atan2(T1(2,1),T1(1,1));
```

Biblioteca UNIVERSITAS Miguel Hernández

dibujo.m

```
1      % Dibujo
2   -  plot(nube(:,1),nube(:,2),'.');
3   -  set(gca,'DataAspectRatio',[1,1,1]);
4   -  hold on
5   -  plot(P_ref(1),P_ref(2),'or');
6   -  quiver(P_ref(1),P_ref(2),cos(phi_ref),sin(phi_ref),'r');
7   -  plot(P_mas_cercano(1),P_mas_cercano(2),'*m');
8   -  quiver(P_mas_cercano(1),P_mas_cercano(2),cos(nube(indice_min,3)),
9   -         sin(nube(indice_min,3)),'m');
10  -  if flag == 1
11  -      plot(P_final(1),P_final(2),'sg');
12  -      quiver(P_final(1),P_final(2),cos(phi_final),sin(phi_final),'g');
13  -  end
```

Anexo 5

Este anexo se encarga de recopilar todo lo referente a la animación del robot con el fin de verificar el correcto cálculo de la trayectoria del mismo.

animacion_completa.m

```
1  - pos_0 = [0,0];
2  - ori_0 = 0;
3  -
4  - % Cada fila de registro_seq será: INICIAL i1 i2 ... iN, j1 j2 ... jN
5  - ultimo_indice = 1;
6  - for h = 1:size(registro_seq(:,1),1)
7  -     seq_actual = registro_seq(h,:);
8  -     seq_actual = [ultimo_indice,seq_actual];
9  -     for i = 2:(N+1)
10 -         if seq_actual(i)==-1 || seq_actual(i+N)==-1
11 -             break
12 -         end
13 -         indice_inicio = ultimo_indice;
14 -         indice_medio = seq_actual(i);
15 -         indice_fin = seq_actual(i+N);
16 -         [pos_F,ori_F] = anim_ciclo_completo(indice_inicio,
17 -                         indice_medio,indice_fin,pos_0,ori_0);
18 -         ultimo_indice = indice_fin;
19 -         pos_0 = pos_F;
20 -         ori_0 = ori_F;
21 -     end
22 - end
23 -
24 - if flag == 1
25 -     seq_actual = secuencia_final;
26 -     seq_actual = [ultimo_indice,seq_actual];
27 -     for z = 2:4
28 -         if seq_actual(z)==-1 || seq_actual(z+3)==-1
29 -             break
30 -         end
31 -         indice_inicio = ultimo_indice;
32 -         indice_medio = seq_actual(z);
33 -         indice_fin = seq_actual(z+3);
34 -         [pos_F,ori_F] = anim_ciclo_completo(indice_inicio,
35 -                         indice_medio,indice_fin,pos_0,ori_0);
36 -         ultimo_indice = indice_fin;
37 -         pos_0 = pos_F;
38 -         ori_0 = ori_F;
39 -     end
40 - end
```

anim_ciclo_completo.m

```
1   - function [pos_F,ori_F] = anim_ciclo_completo(indice_inicio,
2   -                               indice_medio,indice_fin,pos_0,ori_0)
3
4   - movB = 1;
5   - [pos_F1,ori_F1] = con_adyacentes(indice_inicio,indice_medio,
6   -                     movB,pos_0,ori_0);
7   - movB = 0;
8   - [pos_F,ori_F] = con_adyacentes(indice_medio,indice_fin,movB,
9   -                   pos_F1,ori_F1);
10
11  - end
```

con_adyacentes.m

```
1    - function [pos_F,ori_F] = con_adyacentes(indice_inicio,indice_fin,movB,
2    -                              pos_0,ori_0)
3
4    - secuencia_animacion = calculo_secuencia(indice_inicio,indice_fin);
5
6    - if length(secuencia_animacion) == 1
7    -     [pos_F,ori_F] = anim_adyacentes(movB,secuencia_animacion,
8    -                         secuencia_animacion,pos_0,ori_0);
9    - else
10   -     for c = 1:length(secuencia_animacion)-1
11   -         [pos_F,ori_F] = anim_adyacentes(movB,secuencia_animacion(c),
12   -                             secuencia_animacion(c+1),pos_0,ori_0);
13   -     end
14   - end
15
16   - end
```

calculo_secuencia.m

```
1  - function secuencia_animacion=calculo_secuencia(indice_inicio,indice_fin)
2
3  -     ind_horario = zeros(1,8);
4  -     ind_antihorario = zeros(1,8);
5  -     v = [1,2,3,4,5,6,7,8];
6  -     v_triplicado = [v v v];
7
8  -     for t = 9:16
9  -         if v_triplicado(t) == indice_inicio
10 -             break
11 -         end
12 -     end
13
14 -     cont_asc = t;
15 -     cont_desc = t;
16 -     cont = 1;
17
18 -     while 1
19 -         ind_antihorario(cont) = v_triplicado(cont_asc);
20 -         ind_horario(cont) = v_triplicado(cont_desc);
21 -         if v_triplicado(cont_asc) == indice_fin
22 -             ascendente = 1;
23 -             break
24 -         end
25 -         if v_triplicado(cont_desc) == indice_fin
26 -             ascendente = 0;
27 -             break
28 -         end
29 -         cont_asc = cont_asc+1;
30 -         cont_desc = cont_desc-1;
31 -         cont = cont+1;
32 -     end
33
34 -     if ascendente == 1
35 -         secuencia_animacion = ind_antihorario(1:cont);
36 -     else
37 -         secuencia_animacion = ind_horario(1:cont);
38 -     end
39
40 - end
```

anim_adyacentes.m

```
1   - function [pos_F,ori_F] = anim_adyacentes(movB,indice_inicio,indice_fin,
2   -                              pos_0,ori_0)
3
4   - b = 18.59;
5   - p = 101.31;
6   - m = 20;
7   - esc = 50;
8
9   - phi_v = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
10  - y_v = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,
11  -        0,21.95478428];
12
13  - y_inicial = y_v(indice_inicio);
14  - phi_inicial = phi_v(indice_inicio);
15  - y_final = y_v(indice_fin);
16  - phi_final = phi_v(indice_fin);
17
18  - [li,ri] = inversa(phi_inicial,y_inicial);
19  - [lf,rf] = inversa(phi_final,y_final);
20  - dl = (lf-li)/(m-1);
21  - dr = (rf-ri)/(m-1);
22
23  - l = li;
24  - r = ri;
25  - phi = phi_inicial;
26  - y = y_inicial;
27
28    % pasador de B
29  - N_pasador = 30;
30  - radio_pasador = b/2;
31  - d_ang = 2*pi/(N_pasador-1);
32  - ang = 0;
33  - pasador = zeros(N_pasador,2);
34  - for i=1:N_pasador
35  -     pasador(i,1) = radio_pasador*cos(ang);
36  -     pasador(i,2) = radio_pasador*sin(ang);
37  -     ang = ang + d_ang;
38  - end
39
40  - for i = 1:m
41
42  -    [phi,y] = directa(l,r,phi,y);
43  -    l = l+dl;
44  -    r = r+dr;
45  -    T_BA = [ cos(phi)  -sin(phi)    0;
46  -             sin(phi)   cos(phi)    y;
```

```
47  -                 0          0        1];
48
49  -     if movB == 1
50  -         T_A = [ cos(ori_0) -sin(ori_0)  pos_0(1);
51  -                 sin(ori_0) cos(ori_0)   pos_0(2);
52  -                     0          0           1   ];
53  -         T_B = T_A*T_BA;
54  -     else
55  -         T_B = [ cos(ori_0) -sin(ori_0)  pos_0(1);
56  -                 sin(ori_0) cos(ori_0)   pos_0(2);
57  -                     0          0           1   ];
58  -         T_A = T_B*inv(T_BA);
59  -     end
60
61  -     pos_A = [T_A(1,3) T_A(2,3)];
62  -     ori_A = atan2(T_A(2,1),T_A(2,2));
63  -     pos_B = [T_B(1,3) T_B(2,3)];
64  -     ori_B = atan2(T_B(2,1),T_B(2,2));
65
66        % Puntos de los actuadores
67  -     B1 = T_B*[p;0;1];
68  -     B2 = T_B*[-p;0;1];
69  -     A1 = T_A*[-b;0;1];
70  -     A2 = T_A*[b;0;1];
71
72        % Puntos de la ranura
73  -     altura_pasador = 70;
74  -     a11 = T_A*[-radio_pasador;altura_pasador;1];
75  -     a12 = T_A*[ radio_pasador;altura_pasador;1];
76  -     a21 = T_A*[-radio_pasador;-altura_pasador;1];
77  -     a22 = T_A*[ radio_pasador;-altura_pasador;1];
78
79  -     plot(nube(:,1),nube(:,2),'.y');
80  -     hold on
81
82        % Cuerpo A
83  -     plot(pos_A(1),pos_A(2),'xb')
84  -     axis([-600,600,-600,600])
85  -     set(gca,'DataAspectRatio',[1,1,1])
86  -     quiver(pos_A(1),pos_A(2),esc*cos(ori_A),esc*sin(ori_A),'b');
87  -     plot([a11(1),a12(1)],[a11(2),a12(2)],'g','LineWidth',2);
88  -     plot([a11(1),a21(1)],[a11(2),a21(2)],'g','LineWidth',2);
89  -     plot([a22(1),a12(1)],[a22(2),a12(2)],'g','LineWidth',2);
90  -     plot([a22(1),a21(1)],[a22(2),a21(2)],'g','LineWidth',2);
91  -     plot([ (a11(1)+a21(1))/2 , A1(1) ],[ (a11(2)+a21(2))/2 , A1(2) ],
92  -            'g','LineWidth',2);
93  -     plot([ (a22(1)+a12(1))/2 , A2(1) ],[ (a22(2)+a12(2))/2 , A2(2) ],
94  -            'g','LineWidth',2);
```

```
95  -       plot([pos_B(1),pos_A(1)],[pos_B(2),pos_A(2)],'k');
96
97          % Cuerpo B
98  -       plot([B1(1),B2(1)],[B1(2),B2(2)],'c','LineWidth',3);
99  -       plot(pos_B(1)+pasador(:,1),pos_B(2)+pasador(:,2),'c',
100              'LineWidth',2);
101 -       plot(pos_B(1),pos_B(2),'xr');
102 -       quiver(pos_B(1),pos_B(2),esc*cos(ori_B),esc*sin(ori_B),'r');
103
104         % Actuador r
105 -       plot([B2(1),A2(1)],[B2(2),A2(2)],'color',[0.5,0.5,0.5],
106              'LineWidth',2);
107
108         % Actuador l
109 -       plot([B1(1),A1(1)],[B1(2),A1(2)],'color',[0.5,0.5,0.5],
110              'LineWidth',2);
111
112         % Articulaciones
113 -       plot(B1(1),B1(2),'ok'); plot(B2(1),B2(2),'ok');
114 -       plot(A1(1),A1(2),'ok'); plot(A2(1),A2(2),'ok');
115
116 -       pause(0.05)
117 -       if i ~= m
118 -           clf
119 -       end
120 -
121 -       if movB == 1
122 -           pos_F = pos_B;
123 -           ori_F = ori_B;
124 -       else
125 -           pos_F = pos_A;
126 -           ori_F = ori_A;
127 -       end
128
129 -   end
130
131 -   end
```

inversa.m

```
1 - function [l,r] = inversa(phi,y)
2
3 - b = 18.59;
4 - p = 101.31;
5
6 - l = sqrt((p*cos(phi)+b)^2+(y+p*sin(phi))^2);
7 - r = sqrt((-p*cos(phi)-b)^2+(y-p*sin(phi))^2);
8
9 - end
```

directa.m

```
1  - function [phi,y] = directa(l,r,phi_p,y_p)
2
3  - b = 18.59;
4  - p = 101.31;
5
6  - F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
7  -      (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
8
9  - J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
10 -      -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
11
12 - aux = [phi_p;y_p];
13
14 - for iter = 1:10
15 -     aux = aux - inv(J)*F;
16 -     phi_p = aux(1);
17 -     y_p = aux(2);
18 -     J = [-2*p*(b*sin(phi_p)-y_p*cos(phi_p)) , 2*(y_p+p*sin(phi_p));
19 -          -2*p*(b*sin(phi_p)+y_p*cos(phi_p)) , 2*(y_p-p*sin(phi_p))];
20 -     F = [(p*cos(phi_p)+b)^2+(y_p+p*sin(phi_p))^2-l^2 ;
21 -          (-p*cos(phi_p)-b)^2+(y_p-p*sin(phi_p))^2-r^2];
22 - end
23
24 - phi = aux(1);
25 - y = aux(2);
26
27 - end
```

Anexo 6

Este anexo se encarga de recopilar el algoritmo desarrollado en la sección 7.1 que se encarga de la consideración de obstáculos puntuales en la trayectoria del robot y de la forma de evitar mínimos locales.

ws_binario_mp.m

```
1    - clear all;
2
3    - global nube
4    - global global_counter
5    - global secuencia
6
7      % Globales específicas de encontrar_punto
8    - global indice_min
9    - global secuencia_encontrada
10   - global P_mas_cercano
11   - global ori_mas_cercano
12
13   - N = 2;
14   - P_ref = [400 500];
15   - phi_ref = pi/4;
16
17   - radio_robot = 150;
18
19     %Obstáculos
20   - P_obs = [300 300;200 375];
21   - Rmax = 50;
22   - x_p = 0;
23   - y_p = 0;
24
25   - tamanyo = 0;
26   - for i=1:N
27   -     tamanyo = tamanyo + 64^i;
28   - end
29
30   - nube = zeros(tamanyo,3);
31   - global_counter = 0;
32   - secuencia = (-1)*ones(tamanyo,2*N);
33   - T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
34
35   - registro_seq = [];
36   - puntos_encontrados = [];
37   - ori_encontradas = [];
```

```
38  - nube_grande = [];
39
40     % Depende de: b, p, rho0, Delta_rho
41  - phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
42  - y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,
43  -      -21.95478428,0,21.95478428];
44  - for i=1:8
45  -   T{i}=[cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
46  - end
47
48  - P0 = [0,0];
49  - phi0 = 0;
50
51  - f_aux = 1;
52  - f_aux_2 = 1;
53
54  - while 1
55  -   if norm(P_ref-P0) < 10
56  -     break
57  -   end
58
59  -   if f_aux == 1
60  -     global_counter = 0;
61  -     ciclo_completo(P0,phi0,T,1,N)
62  -     f_aux = 0;
63  -     distancias = zeros(size(nube,1),1);
64  -     for i=1:size(nube,1)
65  -       distancias(i) = norm(P_ref-nube(i,1:2));
66  -     end
67  -   end
68
69  -   encontrar_punto(distancias)
70
71  -   for d = 1:size(P_obs,1)
72  -     if norm(P_mas_cercano-P_obs(d,:)) > radio_robot
73  -       flag_conf = 1;
74  -     else
75  -       flag_conf = 0;
76  -       break;
77  -     end
78  -   end
79
80  -   if flag_conf == 1
81  -     registro_seq = [registro_seq; secuencia_encontrada];
82  -     puntos_encontrados = [puntos_encontrados; P_mas_cercano];
83  -     ori_encontradas = [ori_encontradas; ori_mas_cercano];
84  -     P0 = P_mas_cercano;
85  -     phi0 = nube(indice_min,3);
```

Biblioteca UNIVERSITAS Miguel Hernández

```
86  -     f_aux = 1;
87  -     tam_p = length(puntos_encontrados(:,1));
88  -     if tam_p > 4
89  -       X_v = puntos_encontrados(tam_p-4:tam_p,1);
90  -       Y_v = puntos_encontrados(tam_p-4:tam_p,2);
91  -       DX = std(X_v);
92  -       DY = std(Y_v);
93  -       if DX < 5 && DY < 5
94  -         Rmax = Rmax + 25;
95  -         tam = length(registro_seq(:,1));
96  -         registro_seq = registro_seq(1:tam-4,:);
97  -         puntos_encontrados = puntos_encontrados(1:tam-4,:);
98  -         ori_encontradas = ori_encontradas(1:tam-4,:);
99  -         P_mas_cercano = puntos_encontrados(tam-4,:);
100 -         P0 = P_mas_cercano;
101 -         phi0 = ori_encontradas(tam-4,:);
102 -         while 1
103 -           R = Rmax*rand;
104 -           angulo_hacia_destino = atan2(P_ref(2)-P0(2),P_ref(1)-P0(1));
105 -           if rand > 0.5 % Virar hacia la derecha
106 -             angulo = angulo_hacia_destino - pi/2 + pi/2*rand;
107 -           else % Virar hacia la izquierda
108 -             angulo = angulo_hacia_destino + pi/2 + pi/2*rand;
109 -           end
110 -           x_p = P_mas_cercano(1) + R*cos(angulo);
111 -           y_p = P_mas_cercano(2) + R*sin(angulo);
112 -           P_ref_n = [x_p,y_p];
113 -           for d = 1:size(P_obs,1)
114 -             if norm(P_ref_n-P_obs(d,:)) > radio_robot
115 -               flag_conf_3 = 1;
116 -             else
117 -               flag_conf_3 = 0;
118 -               break;
119 -             end
120 -           end
121 -           if flag_conf_3 == 1
122 -             break;
123 -           end
124 -         end
125 -         while 1
126 -           if norm(P_ref_n-P0) < 10
127 -             break
128 -           end
129 -           if f_aux_2 == 1
130 -             global_counter = 0;
131 -             ciclo_completo(P0,phi0,T,1,N)
132 -             f_aux_2 = 0;
133 -             distancias_n = zeros(size(nube,1),1);
```

```
134 -               for i=1:size(nube,1)
135 -                 distancias_n(i) = norm(P_ref_n-nube(i,1:2));
136 -               end
137 -           end
138 -           encontrar_punto(distancias_n)
139 -           for d = 1:size(P_obs,1)
140 -             if norm(P_mas_cercano-P_obs(d,:)) > radio_robot
141 -               flag_conf_2 = 1;
142 -             else
143 -               flag_conf_2 = 0;
144 -               break;
145 -             end
146 -           end
147
148 -           if flag_conf_2 == 1
149 -             registro_seq = [registro_seq; secuencia_encontrada];
150 -             puntos_encontrados = [puntos_encontrados; P_mas_cercano];
151 -             ori_encontradas = [ori_encontradas; ori_mas_cercano];
152 -             P0 = P_mas_cercano;
153 -             phi0 = ori_mas_cercano;
154 -             f_aux_2 = 1;
155 -             nube_grande = [nube_grande;nube];
156 -           else
157 -             distancias(indice_min) = 1000000000;
158 -           end
159 -         end
160 -       end
161 -     end
162 -     nube_grande = [nube_grande;nube];
163 -   else
164 -     distancias(indice_min) = 1000000000;
165 -   end
166
167 - end
168
169 - phi_pi=0;
170
171 - if phi_ref == pi || phi_ref == -pi
172 -   if abs(phi0+phi_ref) < 0.001
173 -     phi_pi = 1;
174 -   else
175 -     phi_pi = 0;
176 -   end
177 - end
178
179 - if phi0 ~= phi_ref && phi_pi == 0
180 -   flag = 1;
181 -   ori_final
```

Biblioteca UNIVERSITAS Miguel Hernández

```
182 -   dibujo
183 - elseif phi0 == phi_ref || phi_pi == 1
184 -   flag = 0;
185 -   dibujo
186 - end
```

ciclo_completo.m

```
1 - function ciclo_completo(pos_0,ori_0,T,n,N)
2
3 - global nube
4 - global global_counter
5 - global secuencia
6
7 - v_i=-1*ones(1,N);
8 - v_j=-1*ones(1,N);
9
10 - if n<=N
11
12         % Cálculo de matriz de transformación
13 -       TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
14 -               sin(ori_0), cos(ori_0),pos_0(2);
15 -                    0 ,     0      ,     1    ];
16
17         % Fijo A, muevo B (medio ciclo)
18 -       for i=1:8
19 -           TB = TA_0*T{i};
20             % Fijo B, muevo A (el otro medio ciclo)
21 -           for j=1:8
22                 % Obtenemos posición y orientación y los almacenamos en nube
23 -               TA = TB*inv(T{j});
24 -               pos = TA(1:2,3)';
25 -               ori = atan2(TA(2,1),TA(1,1));
26 -               global_counter = global_counter + 1;
27 -               nube(global_counter,:) = [pos,ori]; % [ x y phi ];
28                 % Almacenamos los valores de i y j en la variable secuencia
29 -               if n==1
30 -                   v_i(1)=i;
31 -                   v_j(1)=j;
32 -                   secuencia(global_counter,:) = [v_i v_j];
33 -               else
34 -                   auxiliar=secuencia(global_counter-1,:);
35 -                   auxiliar(n)=i;
36 -                   auxiliar(N+n)=j;
37 -                   secuencia(global_counter,:) = auxiliar;
38 -               end
39 -               ciclo_completo(pos,ori,T,n+1,N);
40 -           end
41 -       end
42
43 - end
44 - end
```

encontrar_punto.m

```
1   - function encontrar_punto(distancias)
2
3   - close all
4
5   - global nube
6   - global indice_min
7   - global secuencia_encontrada
8   - global P_mas_cercano
9   - global ori_mas_cercano
10  - global secuencia
11
12    % Búsqueda del punto más cercano al de referencia
13  - [distancia_minima,indice_min] = min(distancias);
14  - P_mas_cercano = nube(indice_min,1:2);
15  - ori_mas_cercano = nube(indice_min,3);
16
17    % Obtención de la secuencia de i y j del punto encontrado
18  - secuencia_encontrada = secuencia(indice_min,:);
19
20  - end
```

calculo_phi_dif.m

```
1   - function phi_dif = calculo_phi_dif(phi_ref,resultado)
2
3   -     phi_dif_tmp = phi_ref-resultado(3);
4
5   -     if phi_dif_tmp > pi
6   -         phi_dif_tmp = phi_dif_tmp-2*pi;
7   -     end
8   -     if phi_dif_tmp < -pi
9   -         phi_dif_tmp = phi_dif_tmp+2*pi;
10  -     end
11
12  -     phi_dif = abs(phi_dif_tmp);
13
14  - end
```

Biblioteca
UNIVERSITAS Miguel Hernández

ori_final.m

```
 1 - v_indices = [];
 2 - v_dist = [];
 3 - T_encontrado = eye(3);
 4
 5 - % Puntos más favorables en posición
 6 - puntos_seq = {  [0 0], 0, [1 -1 -1 1 -1 -1];
 7 -                 [0 0], pi/2, [1 7 8 1 2 7];
 8 -                 [0 0], pi, [1 1 3 1 1 7];
 9 -                 [0 0], -pi/2, [1 3 4 1 6 3];
10 -                 [0.151 0.151], pi/4, [5 1 5 3 2 7];
11 -                 [0.151 0.151], -3*pi/4, [5 1 5 3 2 3];
12 -                 [0.151 -0.151], 3*pi/4, [1 1 5 3 8 3];
13 -                 [0.151 -0.151], -pi/4, [1 1 5 3 8 7];
14 -                 [-0.151 0.151], 3*pi/4, [5 5 1 3 6 3];
15 -                 [-0.151 0.151], -pi/4, [5 5 1 3 6 7];
16 -                 [-0.151 -0.151], pi/4, [1 5 1 3 4 7];
17 -                 [-0.151 -0.151], -3*pi/4, [1 5 1 3 4 3];
18 -                 [0.2135 0], pi/4, [3 8 7 5 1 5];
19 -                 [0.2135 0] 3*pi/4 [3 2 3 5 1 5];
20 -                 [0.2135 0] -pi/4 [3 2 7 5 1 1];
21 -                 [0.2135 0] -3*pi/4 [3 8 3 5 1 1];
22 -                 [-0.2135 0] pi/4 [3 6 7 1 5 1];
23 -                 [-0.2135 0] 3*pi/4 [3 4 3 1 5 1];
24 -                 [-0.2135 0] -pi/4 [3 4 7 1 5 5];
25 -                 [-0.2135 0] -3*pi/4 [3 6 3 1 5 5]};
26
27   % Cálculo de la diferencia de ángulos
28 - phi_dif = phi_ref-nube(indice_min,3);
29 - if phi_dif > pi
30 -     phi_dif = phi_dif-2*pi;
31 - elseif phi_dif < -pi
32 -     phi_dif = phi_dif+2*pi;
33 - end
34
35   % Puntos que coinciden en orientación con la de referencia
36 - for x = 1:20
37 -     if abs(puntos_seq{x,2}-phi_dif) < 0.001
38 -         v_indices = [v_indices;x];
39 -     end
40 - end
41
42   % Cálculo de la distancia entre los seleccionados antes y la referencia
43 - for y = 1:size(v_indices)
44 -     v_aux = puntos_seq{v_indices(y),1};
45 -     theta = atan2(v_aux(2),v_aux(1));
46 -     phi_dif_aux = theta+nube(indice_min,3);
```

```
47 -     mod = norm(v_aux);
48
49 -     P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
50                P_mas_cercano(2)+mod*sin(phi_dif_aux)];
51 -     dist = norm(P_aux-P_ref);
52 -     v_dist = [v_dist;dist];
53 - end
54
55 - [min_tmp,ind_tmp] = min(v_dist);
56 - secuencia_final = puntos_seq{v_indices(ind_tmp),3};
57
58   % Obtención del punto con la secuencia correspondiente
59 - for k=1:3
60 -     i = secuencia_final(k);
61 -     j = secuencia_final(k+3);
62 -     if i==-1 || j==-1
63 -         continue
64 -     end
65 -     T_encontrado = T_encontrado*T{i}*inv(T{j});
66 - end
67
68 - ang = nube(indice_min,3)
69 - x = nube(indice_min,1)
70 - y = nube(indice_min,2);
71 - T0 = [ cos(ang) -sin(ang)   x;
72 -        sin(ang)  cos(ang)   y;
73 -           0         0       1];
74
75 - T1 = T0*T_encontrado;
76 - P_final = T1(1:2,3)';
77 - phi_final = atan2(T1(2,1),T1(1,1));
```

dibujo.m

```
1        % Dibujo
2     -  plot(nube_grande(:,1),nube_grande(:,2),'.');
3     -  set(gca,'DataAspectRatio',[1,1,1]);
4     -  hold on
5     -  plot(P_ref(1),P_ref(2),'or');
6
7     -  for d = 1:size(P_obs,1)
8     -      P = P_obs(d,:);
9     -      plot(P(1),P(2),'ok','Linewidth',2);
10    -  end
11
12    -  quiver(P_ref(1),P_ref(2),50*cos(phi_ref),50*sin(phi_ref),'r');
13    -  plot(P_mas_cercano(1),P_mas_cercano(2),'*m');
14    -  quiver(P_mas_cercano(1),P_mas_cercano(2),50*cos(nube(indice_min,3)),
15    -          50*sin(nube(indice_min,3)),'m');
16    -  if flag == 1
17    -      plot(P_final(1),P_final(2),'sg');
18    -      quiver(P_final(1),P_final(2),50*cos(phi_final),
19    -              50*sin(phi_final),'g');
20    -  end
```

Anexo 7

Este anexo se encarga de recopilar el algoritmo desarrollado en la sección 7.2 que se encarga de la consideración de obstáculos no puntuales en la trayectoria del robot y de la forma de evitar mínimos locales. Este algoritmo tiene un elevado margen de mejora.

ws_binario_mp.m

```
1    - clear all;
2
3    - global nube
4    - global global_counter
5    - global secuencia
6
7      % Globales para encontrar_punto
8    - global indice_min
9    - global secuencia_encontrada
10   - global P_mas_cercano
11   - global ori_mas_cercano
12
13   - N = 2;
14   - P_ref = [400 500];
15   - phi_ref = pi/4;
16
17   - radio_robot = 125;
18
19   - %Obstáculos
20   - P_obs = [];
21   - resol_obs = 20;
22   - for xobs = [400-300:resol_obs:400+300]
23   -   for yobs = [300-50:resol_obs:300+50]
24   -     if (xobs-400)^2/300^2 + (yobs-300)^2/50^2 < 1
25   -       P_obs = [P_obs;[xobs,yobs]];
26   -     end
27   -   end
28   - end
29
30   - Rmax_ini = 50;
31   - Rmax = Rmax_ini;
32   - x_p=0;
33   - y_p=0;
34
35   - tamanyo = 0;
36   - for i=1:N
```

```
37  -     tamanyo = tamanyo + 64^i;
38  -   end
39
40  -   nube = zeros(tamanyo,3);
41  -   global_counter = 0;
42  -   secuencia = (-1)*ones(tamanyo,2*N);
43  -   T = {eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3),eye(3)};
44
45  -   registro_seq = [];
46  -   puntos_encontrados = [];
47  -   ori_encontradas = [];
48  -   nube_grande = [];
49
50      % Depende de: b, p, rho0, Delta_rho
51  -   phi = [0,-pi/4,-pi/2,-pi/4,0,pi/4,pi/2,pi/4];
52  -   y = [50.24201358,21.95478428,0,-21.95478428,-50.24201358,-21.95478428,
53  -        0,21.95478428];
54  -   for i=1:8
55  -     T{i}=[cos(phi(i)),-sin(phi(i)),0;sin(phi(i)),cos(phi(i)),y(i);0,0,1];
56  -   end
57
58  -   P0 = [0,0];
59  -   phi0 = 0;
60
61  -   f_aux = 1;
62  -   f_aux_2 = 1;
63
64  -   while 1
65  -     if norm(P_ref-P0) < 10
66  -       break
67  -     end
68
69  -     if f_aux == 1
70  -       global_counter = 0;
71  -       ciclo_completo(P0,phi0,T,1,N)
72  -       f_aux = 0;
73  -       distancias = zeros(size(nube,1),1);
74  -       for i=1:size(nube,1)
75  -         distancias(i) = norm(P_ref-nube(i,1:2));
76  -       end
77  -     end
78
79  -     encontrar_punto(distancias)
80
81  -     for d = 1:size(P_obs,1)
82  -       if norm(P_mas_cercano-P_obs(d,:)) > radio_robot
83  -         flag_conf = 1;
84  -       else
```

```
85  -         flag_conf = 0;
86  -         break;
87  -       end
88  -     end
89
90  -     if flag_conf == 1
91  -       registro_seq = [registro_seq; secuencia_encontrada];
92  -       puntos_encontrados = [puntos_encontrados; P_mas_cercano];
93  -       ori_encontradas = [ori_encontradas; ori_mas_cercano];
94  -       P0 = P_mas_cercano;
95  -       phi0 = nube(indice_min,3);
96  -       f_aux = 1;
97  -       tam_p = length(puntos_encontrados(:,1));
98  -       if tam_p > 4
99  -         X_v = puntos_encontrados(tam_p-4:tam_p,1);
100 -         Y_v = puntos_encontrados(tam_p-4:tam_p,2);
101 -         DX = std(X_v);
102 -         DY = std(Y_v);
103 -         if DX < 5 && DY < 5
104 -           Rmax = Rmax + 25;
105 -           tam = length(registro_seq(:,1));
106 -           registro_seq = registro_seq(1:tam-4,:);
107 -           puntos_encontrados = puntos_encontrados(1:tam-4,:);
108 -           ori_mas_cercano = ori_encontradas(1:tam-4,:);
109 -           P_mas_cercano = puntos_encontrados(tam-4,:);
110 -           P0 = P_mas_cercano;
111 -           phi0 = ori_encontradas(tam-4,:);
112 -           repulsor = puntos_encontrados(tam-4,:);
113 -           puntos_destino = zeros(5,2);
114
115 -           for h = 1:7
116 -             while 1
117 -               R = Rmax*rand;
118 -               angulo_hacia_destino=atan2(P_ref(2)-P0(2),P_ref(1)-P0(1));
119 -               if rand > 0.5 % Virar hacia la derecha
120 -                 angulo = angulo_hacia_destino - 3*pi/4 + pi/2*rand;
121 -               else % virar hacia la izquierda
122 -                 angulo = angulo_hacia_destino + 3*pi/4 + pi/2*rand;
123 -               end
124 -               x_p = P_mas_cercano(1) + R*cos(angulo);
125 -               y_p = P_mas_cercano(2) + R*sin(angulo);
126 -               P_ref_n = [x_p,y_p];
127
128 -               for d = 1:size(P_obs,1)
129 -                 if norm(P_ref_n-P_obs(d,:)) > radio_robot
130 -                   flag_conf_3 = 1;
131 -                 else
132 -                   flag_conf_3 = 0;
```

```
133 -                 break;
134 -               end
135 -             end
136 -             if flag_conf_3 == 1
137 -               puntos_destino(h,:) = P_ref_n;
138 -               break;
139 -             end
140 -           end
141 -         end
142
143 -         l = length(puntos_destino);
144 -         dist_repulsor = zeros(l,1);
145 -         dist_ref = zeros(l,1);
146 -         dist_aux = zeros(l,1);
147 -         for k = 1:l
148 -           dist_repulsor(k) = norm(repulsor-puntos_destino(k,:));
149 -           dist_ref(k) = norm(P_ref-puntos_destino(k,:));
150 -           dist_aux(k) = dist_ref(k) - dist_repulsor(k);
151 -         end
152 -           [dist_def,ind_def] = min(dist_aux);
153 -           P_ref_n = puntos_destino(ind_def,:);
154
155 -         while 1
156 -           if norm(P_ref_n-P0) < 10
157 -             break
158 -           end
159 -           if f_aux_2 == 1
160 -             global_counter = 0;
161 -             ciclo_completo(P0,phi0,T,1,N)
162 -             f_aux_2 = 0;
163 -             distancias_n = zeros(size(nube,1),1);
164 -             for i=1:size(nube,1)
165 -               distancias_n(i) = norm(P_ref_n-nube(i,1:2));
166 -             end
167 -           end
168 -           encontrar_punto(distancias_n)
169 -           for d = 1:size(P_obs,1)
170 -             if norm(P_mas_cercano-P_obs(d,:)) > radio_robot
171 -               flag_conf_2 = 1;
172 -             else
173 -               flag_conf_2 = 0;
174 -               break;
175 -             end
176 -           end
177
178 -           if flag_conf_2 == 1
179 -             registro_seq = [registro_seq; secuencia_encontrada];
180 -             puntos_encontrados=[puntos_encontrados; P_mas_cercano];
```

```
181 -                 ori_encontradas = [ori_encontradas; ori_mas_cercano];
182 -                 P0 = P_mas_cercano;
183 -                 phi0 = ori_mas_cercano;
184 -                 f_aux_2 = 1;
185 -                 nube_grande = [nube_grande;nube];
186 -               else
187 -                 distancias(indice_min) = 1000000000;
188 -                 tam = length(registro_seq(:,1));
189 -                 registro_seq = registro_seq(1:tam,:);
190 -                 puntos_encontrados = puntos_encontrados(1:tam,:);
191 -                 ori_encontradas = ori_encontradas(1:tam,:);
192 -               end
193 -             end
194 -           end
195 -         end
196 -         nube_grande = [nube_grande;nube];
197 -       else
198 -           distancias(indice_min) = 1000000000;
199 -           tam = length(registro_seq(:,1));
200 -           registro_seq = registro_seq(1:tam,:);
201 -           puntos_encontrados = puntos_encontrados(1:tam,:);
202 -           ori_encontradas = ori_encontradas(1:tam,:);
203 -       end
204
205 -     end
206
207 -     phi_pi=0;
208
209 -     if phi_ref == pi || phi_ref == -pi
210 -       if abs(phi0+phi_ref) < 0.001
211 -           phi_pi = 1;
212 -       else
213 -           phi_pi = 0;
214 -       end
215 -     end
216
217 -     if phi0 ~= phi_ref && phi_pi == 0
218 -       flag = 1;
219 -       ori_final
220 -       dibujo
221 -     elseif phi0 == phi_ref || phi_pi == 1
222 -       flag = 0;
223 -       dibujo
224 -     end
```

ciclo_completo.m

```
 1 - function ciclo_completo(pos_0,ori_0,T,n,N)
 2
 3 - global nube
 4 - global global_counter
 5 - global secuencia
 6
 7 - v_i=-1*ones(1,N);
 8 - v_j=-1*ones(1,N);
 9
10 - if n<=N
11
12         % Cálculo de matriz de transformación
13 -       TA_0 = [cos(ori_0),-sin(ori_0),pos_0(1);
14 -               sin(ori_0), cos(ori_0),pos_0(2);
15 -                    0 ,      0      ,    1    ];
16
17         % Fijo A, muevo B (medio ciclo)
18 -       for i=1:8
19 -           TB = TA_0*T{i};
20             % Fijo B, muevo A (el otro medio ciclo)
21 -           for j=1:8
22                 % Obtenemos posición y orientación y los almacenamos en nube
23 -               TA = TB*inv(T{j});
24 -               pos = TA(1:2,3)';
25 -               ori = atan2(TA(2,1),TA(1,1));
26 -               global_counter = global_counter + 1;
27 -               nube(global_counter,:) = [pos,ori]; % [ x y phi ];
28                 % Almacenamos los valores de i y j en la variable secuencia
29 -               if n==1
30 -                   v_i(1)=i;
31 -                   v_j(1)=j;
32 -                   secuencia(global_counter,:) = [v_i v_j];
33 -               else
34 -                   auxiliar=secuencia(global_counter-1,:);
35 -                   auxiliar(n)=i;
36 -                   auxiliar(N+n)=j;
37 -                   secuencia(global_counter,:) = auxiliar;
38 -               end
39 -               ciclo_completo(pos,ori,T,n+1,N);
40 -           end
41 -       end
42
43 - end
44 - end
```

encontrar_punto.m

```
1   - function encontrar_punto(distancias)
2
3   - close all
4
5   - global nube
6   - global indice_min
7   - global secuencia_encontrada
8   - global P_mas_cercano
9   - global ori_mas_cercano
10  - global secuencia
11
12    % Búsqueda del punto más cercano al de referencia
13  - [distancia_minima,indice_min] = min(distancias);
14  - P_mas_cercano = nube(indice_min,1:2);
15  - ori_mas_cercano = nube(indice_min,3);
16
17    % Obtención de la secuencia de i y j del punto encontrado
18  - secuencia_encontrada = secuencia(indice_min,:);
19
20  - end
```

calculo_phi_dif.m

```
1  - function phi_dif = calculo_phi_dif(phi_ref,resultado)
2
3  -     phi_dif_tmp = phi_ref-resultado(3);
4
5  -     if phi_dif_tmp > pi
6  -         phi_dif_tmp = phi_dif_tmp-2*pi;
7  -     end
8  -     if phi_dif_tmp < -pi
9  -         phi_dif_tmp = phi_dif_tmp+2*pi;
10 -     end
11
12 -     phi_dif = abs(phi_dif_tmp);
13
14 - end
```

Biblioteca
UNIVERSITAS Miguel Hernández

ori_final.m

```
 1 - v_indices = [];
 2 - v_dist = [];
 3 - T_encontrado = eye(3);
 4
 5 - % Puntos más favorables en posición
 6 - puntos_seq = {  [0 0], 0, [1 -1 -1 1 -1 -1];
 7 -                 [0 0], pi/2, [1 7 8 1 2 7];
 8 -                 [0 0], pi, [1 1 3 1 1 7];
 9 -                 [0 0], -pi/2, [1 3 4 1 6 3];
10 -                 [0.151 0.151], pi/4, [5 1 5 3 2 7];
11 -                 [0.151 0.151], -3*pi/4, [5 1 5 3 2 3];
12 -                 [0.151 -0.151], 3*pi/4, [1 1 5 3 8 3];
13 -                 [0.151 -0.151], -pi/4, [1 1 5 3 8 7];
14 -                 [-0.151 0.151], 3*pi/4, [5 5 1 3 6 3];
15 -                 [-0.151 0.151], -pi/4, [5 5 1 3 6 7];
16 -                 [-0.151 -0.151], pi/4, [1 5 1 3 4 7];
17 -                 [-0.151 -0.151], -3*pi/4, [1 5 1 3 4 3];
18 -                 [0.2135 0], pi/4, [3 8 7 5 1 5];
19 -                 [0.2135 0] 3*pi/4 [3 2 3 5 1 5];
20 -                 [0.2135 0] -pi/4 [3 2 7 5 1 1];
21 -                 [0.2135 0] -3*pi/4 [3 8 3 5 1 1];
22 -                 [-0.2135 0] pi/4 [3 6 7 1 5 1];
23 -                 [-0.2135 0] 3*pi/4 [3 4 3 1 5 1];
24 -                 [-0.2135 0] -pi/4 [3 4 7 1 5 5];
25 -                 [-0.2135 0] -3*pi/4 [3 6 3 1 5 5]};
26
27   % Cálculo de la diferencia de ángulos
28 - phi_dif = phi_ref-nube(indice_min,3);
29 - if phi_dif > pi
30 -     phi_dif = phi_dif-2*pi;
31 - elseif phi_dif < -pi
32 -     phi_dif = phi_dif+2*pi;
33 - end
34
35   % Puntos que coinciden en orientación con la de referencia
36 - for x = 1:20
37 -     if abs(puntos_seq{x,2}-phi_dif) < 0.001
38 -         v_indices = [v_indices;x];
39 -     end
40 - end
41
42   % Cálculo de la distancia entre los seleccionados antes y la referencia
43 - for y = 1:size(v_indices)
44 -     v_aux = puntos_seq{v_indices(y),1};
45 -     theta = atan2(v_aux(2),v_aux(1));
46 -     phi_dif_aux = theta+nube(indice_min,3);
```

```
47 -       mod = norm(v_aux);
48
49 -       P_aux = [P_mas_cercano(1)+mod*cos(phi_dif_aux),
50                  P_mas_cercano(2)+mod*sin(phi_dif_aux)];
51 -       dist = norm(P_aux-P_ref);
52 -       v_dist = [v_dist;dist];
53 -   end
54
55 -   [min_tmp,ind_tmp] = min(v_dist);
56 -   secuencia_final = puntos_seq{v_indices(ind_tmp),3};
57
58     % Obtención del punto con la secuencia correspondiente
59 -   for k=1:3
60 -       i = secuencia_final(k);
61 -       j = secuencia_final(k+3);
62 -       if i==-1 || j==-1
63 -           continue
64 -       end
65 -       T_encontrado = T_encontrado*T{i}*inv(T{j});
66 -   end
67
68 -   ang = nube(indice_min,3)
69 -   x = nube(indice_min,1)
70 -   y = nube(indice_min,2);
71 -   T0 = [ cos(ang) -sin(ang)   x;
72 -          sin(ang)  cos(ang)   y;
73 -             0          0      1];
74
75 -   T1 = T0*T_encontrado;
76 -   P_final = T1(1:2,3)';
77 -   phi_final = atan2(T1(2,1),T1(1,1));
```

dibujo.m

```
1       % Dibujo
2     - plot(nube_grande(:,1),nube_grande(:,2),'.');
3     - set(gca,'DataAspectRatio',[1,1,1]);
4     - hold on
5     - plot(P_ref(1),P_ref(2),'or');
6
7     - for d = 1:size(P_obs,1)
8     -     P = P_obs(d,:);
9     -     plot(P(1),P(2),'ok','Linewidth',2);
10    - end
11
12    - quiver(P_ref(1),P_ref(2),50*cos(phi_ref),50*sin(phi_ref),'r');
13    - plot(P_mas_cercano(1),P_mas_cercano(2),'*m');
14    - quiver(P_mas_cercano(1),P_mas_cercano(2),50*cos(nube(indice_min,3)),
15    -        50*sin(nube(indice_min,3)),'m');
16    - if flag == 1
17    -     plot(P_final(1),P_final(2),'sg');
18    -     quiver(P_final(1),P_final(2),50*cos(phi_final),
19    -            50*sin(phi_final),'g');
20    - end
```

Referencias bibliográficas

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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

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