Universidad Miguel Hernández de Elche

MÁSTER UNIVERSITARIO EN ROBÓTICA

“Desarrollo de algoritmos para la navegación de un robot modular con un único actuador en entornos conocidos”

Trabajo de Fin de Máster

Curso académico
2019/2020

Autor: Julio Gallego Pagán
Tutor/es: Luis Payá Castelló
Adrián Peidró Vidal

Agradecimientos

En primer lugar me gustaría agradecer a mis tutores, Luis Payá y Adrián Peidró, por su confianza y la oportunidad de trabajar en este proyecto, por su ayuda, interés y preocupación. A Adrián en concreto por su ayuda diaria, por sus consejos, ideas, soluciones y por soportar conversaciones de más de 100 correos; y a Luis por su confianza desde el minuto 1 y por darme la oportunidad de trabajar en este grupo de investigación hace ya 3 años.

A los pilares fundamentales de mi vida, mis padres y mi hermano, quienes me han ayudado y apoyado durante este año un poco ajetreado de mi vida, haciéndome ver que la vida son dos días y que por muchas dificultades que vengan, siempre seré capaz de sobrellevarlas.

Y por supuesto, a todos y cada uno de mis amigos y amigas, por tantas risas, tonterías y consejos. Sin ellos el confinamiento se habría hecho aún más eterno.

Índice general

Agradecimientos II

1. Introducción **1**
1.1. Robot MASAR: Modular and Single - Actuator Robot 1
1.2. Trabajos anteriores 4
1.3. Objetivos del trabajo 6
1.4. Estructura de la memoria 7

2. Estado del Arte **9**
2.1. Robots SAMR 9
2.1.1. SAMR de tipo 1 9
2.1.2. SAMR de tipo 2 13
2.1.3. SAMR de tipo 3 14
2.2. Robots MSRR 16
2.2.1. MSRR de tipo red (*lattice type*) 20
2.2.2. MSRR de tipo cadena (*chain type*) 21
2.2.3. MSRR de tipo MCC (*Mobile Configuration Change*) . . . 21

3. Diseño de un controlador en posición y orientación para el Robot MASAR **25**
3.1. Modelo cinemático del robot MASAR 25
3.2. Control en posición 28
3.2.1. Primer controlador propuesto 28
Modo de funcionamiento 28
Ejemplos y conclusión 29
3.2.2. Segundo controlador propuesto 31
Modo de funcionamiento 32
Ejemplos y conclusión 33
3.3. Control en posición y orientación 34
3.3.1. Entradas del sistema 36
3.3.2. Modo de movimiento de 180º 38
Cálculo del ángulo recorrido 39
Control de los movimientos de 180º 40
Empleo del biestable R-S 44
3.3.3. Modo de posición y orientación final 46
Primer giro 47
Segundo giro 49
Tercer giro 50
3.3.4. Ejemplos de simulación y error 56
Ejemplo nº 1 57

Ejemplo nº 2 57

4. Diseño e implementación de algoritmos de navegación 64
4.1. Creación del mapa 65
4.1.1. Generación de obstáculos 66
4.1.2. Mapa de ocupación, punto de inicio y punto final . . . 68
4.2. Diagrama de Voronoi 73
4.3. Grafo de Visibilidad 75
4.3.1. Modificación del mapa 78
Cálculo del polígonos circunscritos 78
Comprobación de punto dentro de polígono circunscrito 81
4.3.2. Extracción de vértices 83
Vértices de rectángulos 83
Filtrado de vértices 84
4.3.3. Generación de caminos 85
4.4. Algoritmo A* e identificación de estrecheces 95
4.4.1. Algoritmo A* 95
Modificación del Algoritmo A* 99
4.4.2. Identificación de estrecheces 103
4.5. Estimación de tiempos de recorrido 109
4.5.1. Simulación de tramos libres 110
Identificación de tramos libres 110
Ejecución de la simulación 111
4.5.2. Simulación de tramos estrechos 112
Identificación de tramos estrechos 112
Estimación del tiempo para recorrer una estrechez . . . 113
4.5.3. Ejemplos de simulación 115
Ejemplo nº 1 115
Ejemplo nº 2 119

5. Conclusión y trabajos futuros 123
5.1. Resultados y conclusiones 123
5.2. Trabajos futuros 124

Bibliografía 125

Índice de figuras

1.1. Partes que conforman el robot MASAR, vista delantera 2
1.2. Partes que conforman el robot MASAR, vista superior 2
1.3. Locomoción del robot modular propuesto 3
1.4. Ilustración de plano del robot MASAR 3
1.5. Módulos del robot MASAR combinados para subir una mesa 4
1.6. Resolución de camino poligonal con estrecheces 5
1.7. Aplicación del segundo controlador para control en posición . 6

2.1. Robot octópodo de Soyguder y Alli, 2011 10
2.2. Disposición de motor y transmisiones del robot de Soyguder y Alli, 2011 10
2.3. Robot trepador SAMR tipo 1 de Birkmeyer, Gillies y Fearing, 2011 11
2.4. Vista general del movimiento del robot de Birkmeyer, Gillies y Fearing, 2011 11
2.5. Robots trepadores de Zarrouk y Fearing, 2015, en dos tamaños distintos 12
2.6. Fases de desplazamiento del robot de Zarrouk y Fearing, 2015 12
2.7. Robot SAMR tipo 2 de Sfeir, Shammas y Asmar, 2014 13
2.8. Robot de Cheng y col., 2010, en diferentes configuraciones . . 14
2.9. Locomoción del robot de tipo serpiente 15
2.10. Robot DynaRoACH 15
2.11. Shape-memory alloys 16
2.12. Conjunto de movimientos del robot nadador de Refael y Degani, 2015 17
2.13. Micro-robot de Dharmawan y col., 2017 17
2.14. Movimiento generado por las diferentes frecuencias de resonancia 18
2.15. Prototipo del robot saltador de Zhao y col., 2013 18
2.16. Enderezamiento tras caída 19
2.17. Cambio de dirección desde el suelo 19
2.18. Módulos M-Blocks 20
2.19. Asociación de M-Blocks 21
2.20. Robot CONRO: módulo independiente y hexápodo 21
2.21. Robot Uni-Rovers 22
2.22. Robot JL-I/II 22
2.23. Robot Millibots 23
2.24. Robot Amoeba 23

3.1. Variables partícipes del modelo cinemático 26

3.2. Bloque modelado MASAR 28
3.3. Diagrama del primer controlador 29
3.4. Inicio de movimiento hacia punto objetivo 29
3.5. Final de movimiento hacia punto objetivo 30
3.6. Recorrido de una circunferencia empleando el primer controlador 30
3.7. Variación de la variable binaria empleando el primer controlador 31
3.8. Identificación de pivote más cercano al objetivo y alineación . 32
3.9. Distancia objetivo - punto medio 33
3.10. Retardo en la variable binaria 34
3.11. Diagrama del segundo controlador 34
3.12. Alineación inicial y desplazamientos de 180º 35
3.13. Cambio de control y llegada a objetivo 35
3.14. Variación de la variable binaria empleando el primer controlador 36
3.15. Ejemplo de orientación objetivo 37
3.16. Bloques para el cálculo inicial de *bin* 38
3.17. Bloque de retardo para la variable binaria 38
3.18. Cálculo del ángulo entre el robot MASAR y recta pivote-destino 39
3.19. Bloque de cálculo del ángulo recorrido 40
3.20. Distancias entre pivotes y circunferencias de destino 42
3.21. Comprobación del valor del ángulo recorrido 43
3.22. Llegada de los pivotes a su circunferencia de destino correspondiente 44
3.23. Ejemplo de biestable R-S 45
3.24. Implementación del bloque R-S en el controlador 46
3.25. Distancia entre Ao y B 47
3.26. Pivote B alcanza la circunferencia de Ao 49
3.27. Segundo giro, aplicado sobre A 49
3.28. Giro final 51
3.29. Giro corto vs Giro largo 51
3.30. Ángulos necesarios para el tercer giro 52
3.31. Ejemplo 1: Inicio del movimiento de 180º 57
3.32. Ejemplo 1: Continuación del movimiento de 180º 58
3.33. Ejemplo 1: El pivote A alcanza su circunferencia objetivo . . . 58
3.34. Ejemplo 1: Se sitúa el pivote B sobre la circunferencia de A, primer giro 59
3.35. Ejemplo 1: El pivote A alcanza su pivote objetivo, segundo giro 59
3.36. Ejemplo 1: El pivote B alcanza su pivote objetivo, tercer giro . 60
3.37. Ejemplo 2: Inicio del movimiento de 180º 61
3.38. Ejemplo 2: Continuación del movimiento de 180º 61
3.39. Ejemplo 2: El pivote A alcanza su circunferencia objetivo . . . 62
3.40. Ejemplo 2: Se sitúa el pivote B sobre la circunferencia de A, primer giro 62
3.41. Ejemplo 2: El pivote A alcanza su pivote objetivo, segundo giro 63
3.42. Ejemplo 2: El pivote B alcanza su pivote objetivo, tercer giro . 63

4.1. Proceso de navegación de un robot móvil 65
4.2. Ejemplo de mapa aleatorio en formato cartesiano 73

4.3. Ejemplo de mapa aleatorio en formato ocupación 74
4.4. Diagrama de Voronoi 74
4.5. Caminos de Voronoi con 30 obstáculos 76
4.6. Caminos de Voronoi con 40 obstáculos 76
4.7. Ejemplo de Grado de Visibilidad 77
4.8. Polígonos circunscritos 78
4.9. Cálculo de *alpha* y *p* 79
4.10. Rotaciones 2*alpha* 80
4.11. Recorrido del polígono 82
4.12. Comparación a evaluar 82
4.13. Vértices candidatos mostrados en color azul 86
4.14. Caminos desde inicio/fin a vértices visibles 86
4.15. Discretización de posibles caminos 87
4.16. Caminos en cian generados entre punto de inicio y sus vértices visibles 90
4.17. Caminos en verde generados entre punto final y sus vértices visibles 90
4.18. Caminos totales generados en magenta 91
4.19. Camino óptimo a partir de Voronoi y A* 97
4.20. Camino óptimo a partir de Grafo de Visibilidad y A* 98
4.21. Desplazamiento a lo largo de una estrechez 98
4.22. Aplicación de la máscara "*e*"sobre la posición actual 100
4.23. Identificación de obstáculos cercanos 100
4.24. Caminos propuestos con Diagrama de Voronoi y A* modificado 101
4.25. Caminos propuestos con Grafo de Visibilidad y A* modificado 102
4.26. Recta tangente entre punto anterior y punto siguiente 104
4.27. Estructura de la tabla de estrecheces 106
4.28. Estrecheces detectadas en los caminos resultantes de Voronoi y A* 107
4.29. Estrecheces detectadas en los caminos resultantes de Grafo de Visibilidad y A* 108
4.30. Recta tangente obtenida de pendiente diferente al camino . . . 108
4.31. Estrechez errónea marcada con la circunferencia amarilla . . . 109
4.32. Modelado del tiempo de pegue/despegue 111
4.33. Final del tramo 112
4.34. Parámetros de la estimación matemática 114
4.35. Mapa ejemplo nº 1 116
4.36. Caminos de Voronoi y A*, ejemplo 1 117
4.37. Caminos de Grafo de Visibilidad y A*, ejemplo 1 118
4.38. Mapa ejemplo nº 2 119
4.39. Caminos de Voronoi y A*, ejemplo 2 120
4.40. Caminos de Grafo y A*, ejemplo 2 121

Capítulo 1

Introducción

El avance actual de la tecnología está permitiendo que el ámbito de la robótica móvil se desarrolle a pasos agigantados, gracias a la tendencia ascendente en cuanto al número de aplicaciones que esta modalidad robótica puede realizar. Sumado a los incesantes avances en ámbitos de la ingeniería como sistemas de control, visión por computador, inteligencia artificial, sistemas de percepción, etc... están permitiendo que cada vez se diseñen más robots de esta categoría.

Tomamos como definición de robot móvil la siguiente:

Un robot móvil es un vehículo autónomo que se desplaza por un entorno realizando una determinada tarea de gran utilidad.

Entre estas tareas destacan las siguientes:

- Limpieza
- Logística
- Transporte
- Exploración e inspección
- Asistencia a personas

1.1. Robot MASAR: Modular and Single - Actuator Robot

Este Trabajo de Final de Máster se centra en el diseño de un nuevo robot, actualmente bajo investigación, patentado por la Universidad Miguel Hernández de Elche.

El robot MASAR se presenta como un robot móvil modular con la capacidad de realizar numerosos movimientos empleando un único actuador por módulo, controlando su posición y orientación a lo largo de un plano. La composición de un módulo del robot se muestra en la Figuras 1.1 y 2.7.

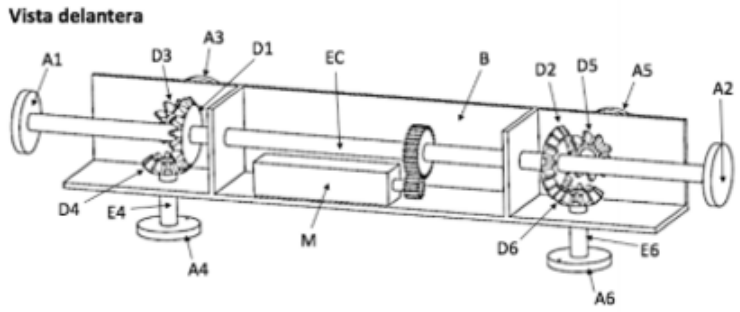

FIGURA 1.1: Partes que conforman el robot MASAR, vista delantera

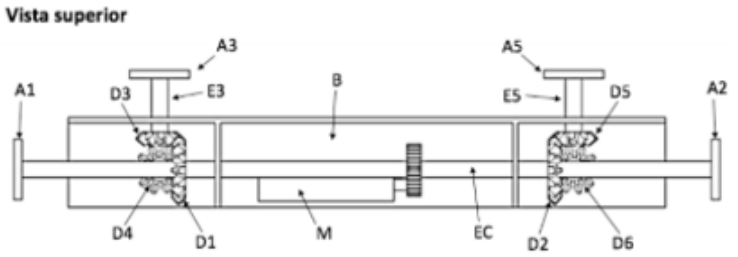

FIGURA 1.2: Partes que conforman el robot MASAR, vista superior

El robot MASAR está compuesto por un cuerpo principal B, solidario a un motor M que mueve un eje central EC. A ambos extremos de este eje central, encontramos un juego de tres ruedas dentadas cónicas, D1, D3, D4 y D2, D5, D6, cuyos ejes intersecan en un punto. Las ruedas dentadas están conectadas a pequeños ejes E3, E4, E5, E6, que rotan respecto al cuerpo principal B y en cuyos extremos se sitúan elementos de adhesión A3, A4, A5, A6. También encontramos dos elementos de adhesión A1, A2 en los extremos del eje central. Estos elementos de adhesión son las unidades que permiten al robot desplazarse a lo largo de uno o varios planos. Su modo de locomoción está basado en pegar y despegar de forma alterna los dos pivotes de giro que están en contacto con el plano de desplazamiento. El pivote que se encuentra adherido al plano se convierte en aquel alrededor del cual el robot pivota, ejecutando un giro determinado gracias a la acción de su actuador.

En la Figura 1.3 se muestra un ejemplo de movimiento del robot MASAR a lo largo de un plano.

Imaginemos que los pivotes en contacto con el plano de desplazamiento son A4 y A6, en el apartado a) de la Figura 1.3. Podemos ver como el robot fija el pivote A4 y ejecuta un giro mediante el accionamiento del actuador en sentido antihorario, desplazando el pivote opuesto. Finalizado este giro, el robot se detiene, intercambia la acción sobre los pivotes, pegando el elemento A6 y despegando el A4. Luego ejecuta una rotación de signo opuesto al anterior, es decir, en sentido horario (Figura 1.3 (b)). Finalmente la Figura 1.3

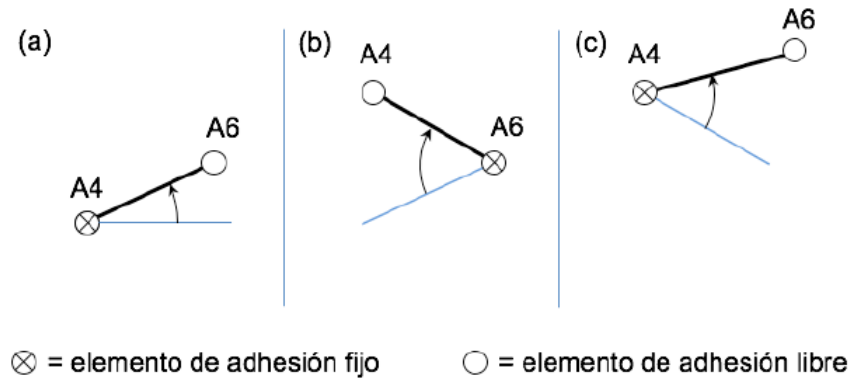

FIGURA 1.3: Locomoción del robot modular propuesto

(c) muestra un movimiento similar al de la Figura 1.3 (a).

La sucesión de estos movimientos, en los que se alterna el pivote de giro y el signo del ángulo de rotación, genera el movimiento del robot de forma libre y controlada respecto a posición y orientación.

Con este único actuador, aparte de poder desplazarse por el suelo, el robot MASAR también es capaz de realizar transiciones cóncavas entre planos perpendiculares, permitiendo al robot desplazarse por paredes y techos. Este es uno de los grandes motivos por el cual el robot dispone de dos unidades de adhesión en cada una de las caras de su cuerpo principal.

Esta posibilidad de transición entre planos se ilustra en la Figura 1.4.

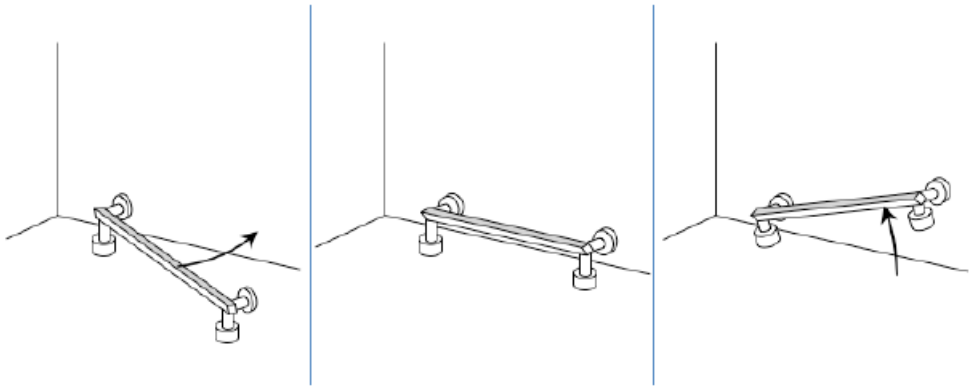

FIGURA 1.4: Ilustración de plano del robot MASAR

Una vez comentada su capacidad de locomoción empleando un único actuador, describiendo todas las posibilidades de movimiento que ofrece el robot MASAR, se profundizará en su segunda gran característica, la modularidad.

El robot MASAR es un robot modular reconfigurable (MSRR, Modular Self - Reconfigurable Robots), ya que en su propuesta de diseño se tuvo en

cuenta la posibilidad de que distintos módulos del robot puedan combinarse formando un robot más grande, a través de la conexión de sus unidades de adhesión. Afirmamos por tanto que estas unidades de adhesión o pivotes tienen dos usos fundamentales: unidades esenciales para el desplazamiento del robot y unión con otras unidades de adhesión para conformar robots mayores.

Además, esta combinación de módulos deriva en la formación de nuevos robots de arquitectura serie, paralela o híbrida, permitiendo así ampliar el abanico de resolución de tareas más complejas y moverse por entornos más exigentes.

En la Figura 1.5 se muestra un robot modular más avanzado, formado por la combinación serie de cuatro módulos del robot MASAR, trabajando de forma conjunta para para subir una mesa. Un módulo individual sería incapaz de realizar esta acción, ya que requiere una transición de tipo convexa y no cóncava.

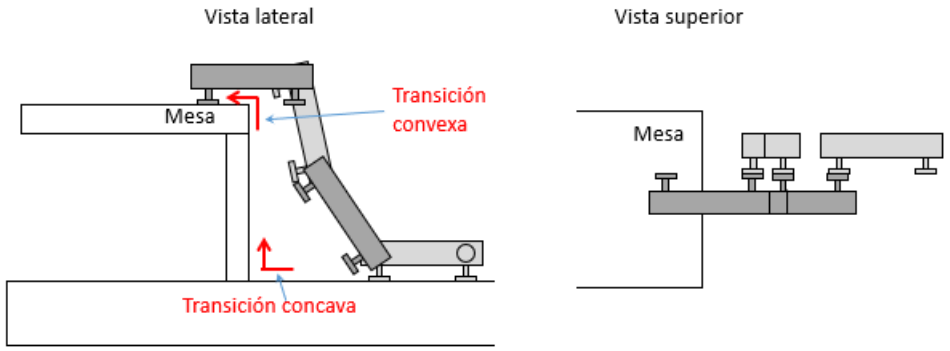

FIGURA 1.5: Módulos del robot MASAR combinados para subir una mesa

1.2. Trabajos anteriores

Este Trabajo de Final de Máster nace como continuación del Trabajo de Final de Grado "Planificación de Trayectorias para el Robot MASAR (Modular and Single Actuator Robot"presentado por el autor del presente trabajo en Julio de 2019 .

El objetivo de este Trabajo de Final de Grado consistió en abordar el inicio de la investigación acerca del robot MASAR. Para ello, se estudiaron y analizaron diferentes soluciones a la planificación de trayectorias, partiendo desde el nivel más elemental. Se trabajó primeramente en el desarrollo de la cinemática inversa, diseñando la locomoción del robot MASAR ante situaciones básicas del entorno, esto es, qué movimientos debía ejecutar el robot en cada momento. Se trabajó bajo 3 situaciones posibles:

- Desplazamiento a lo largo de un camino poligonal

- Resolución de estrecheces
- Resolución de camino combinado

En la Figura 1.6 se muestra un ejemplo de resolución de un camino poligonal resuelto mediante cinemática inversa, empleando los siguiente movimientos:

- Orientaciones para situarse sobre el camino
- Desplazamiento mediante giros de 180º
- Cambio de recta
- Resolución de estrecheces empleando una locomoción de tipo serpiente

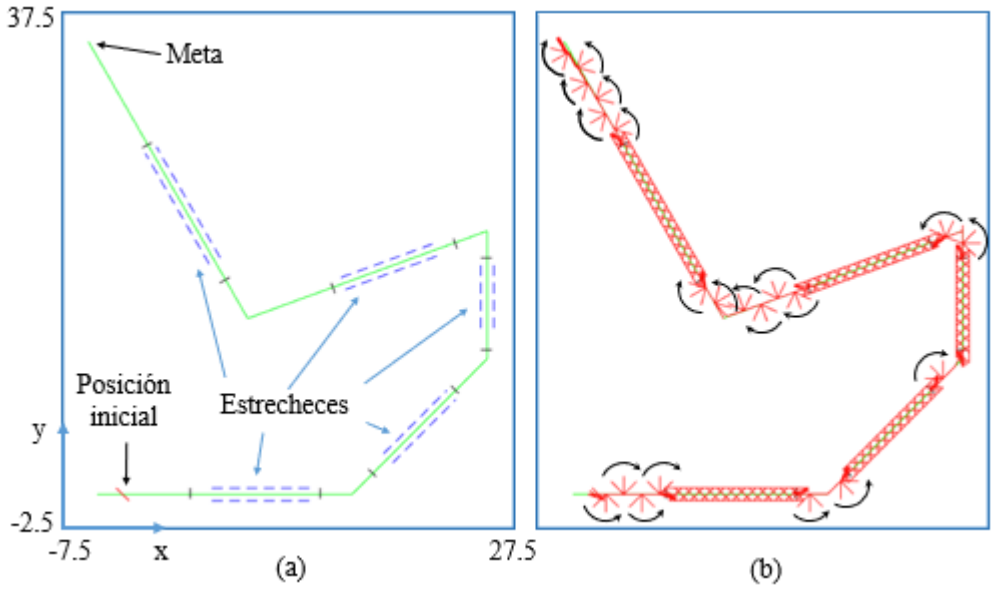

FIGURA 1.6: Resolución de camino poligonal con estrecheces

El segundo gran bloque del TFG se centró en el diseño de controladores en posición para resolver una planificación de trayectorias mediante resolución de la cinemática directa. Se estudiaron dos propuestas:

1. Control basado en pegado del pivote más cercano
2. Control basado en movimientos de 180º + controlador anterior para llegada

El primer controlador propuesto fijaba en cada momento aquel pivote que se encontrara más cerca del objetivo, variando el valor de una variable binaria. El empleo de este controlador resultó ser ineficiente debido al gran número de pegadas y despegadas de los pivotes.

El segundo controlador corrigió este suceso, pues su funcionamiento está fundamentado en giros de 180º por parte del robot, reduciendo así el número de pegadas y despegadas, y por tanto, aumentando la eficiencia en cuento al coste del movimiento. En la Figura 4.1 se muestra un ejemplo de aplicación del segundo controlador.

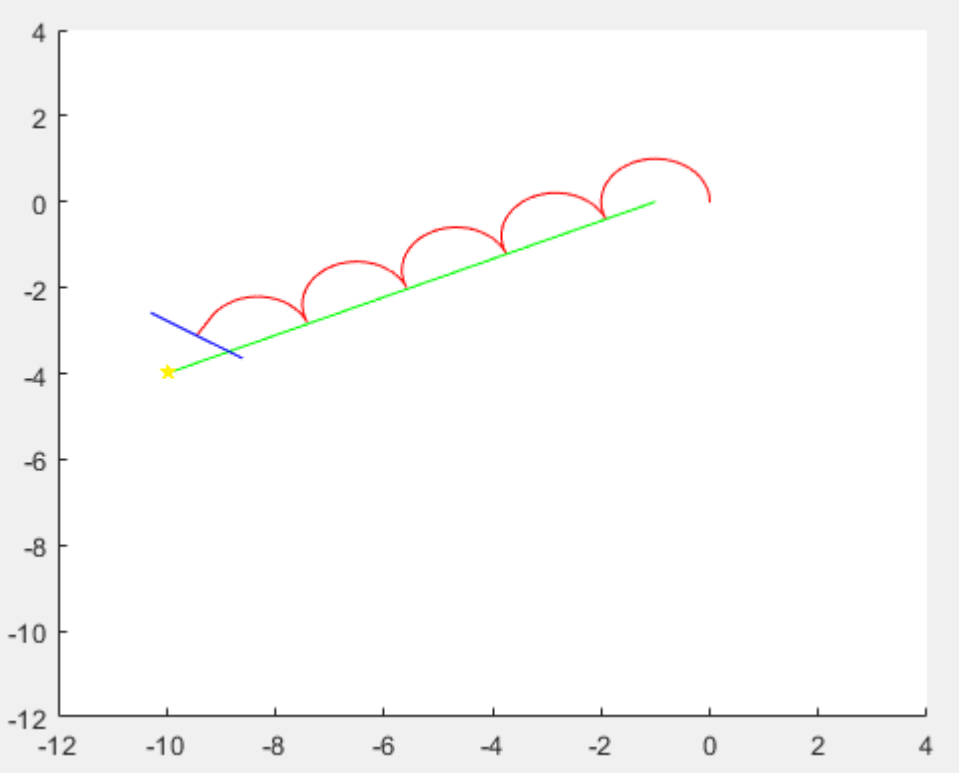

FIGURA 1.7: Aplicación del segundo controlador para control en posición

1.3. Objetivos del trabajo

El principal objetivo de este Trabajo de Final de Máster es continuar con los experimentos realizados en el Trabajo de Final de Grado.

Con este objetivo en mente, se proponen dos grandes apartados en el presente trabajo:

1. Diseño de un controlador en posición y orientación
2. Implementación y comparativa de algoritmos de navegación: Diagrama de Voronoi vs Grafo de Visibilidad

El primer punto, basado en el diseño de un controlador en posición y orientación acorde a la locomoción y cinemática del robot MASAR, pretende mejorar los controladores diseñados en el anterior TFG, conformando un

control estable y robusto con el objetivo de que el punto central del robot alcance un punto de meta, añadiendo además la posibilidad de elegir por parte del usuario una orientación determinada. Podemos considerarlo, por tanto, una mejora de control diferencial y cinemática directa comentada en el apartado anterior.

Por otro lado, atendiendo al segundo punto, este trabajo centra gran parte de sus experimentos en el estudio de dos algoritmos de navegación. Una vez diseñado el modo de locomoción del robot MASAR, así como resuelta su cinemática inversa en el TFG, el siguiente paso ha sido trabajar sobre un mapa de un entorno conocido conformado con obstáculos geométricos. Dichos mapas son generados de forma aleatoria teniendo en cuenta ciertas consignas por parte del usuario y sobre él se aplican los siguiente algoritmos de navegación para robots móviles:

- Diagama de Voronoi
- Grafo de Visibilidad

Ambos algoritmos son capaces de generar múltiples posibles caminos desde un punto inicio hasta un punto final, salvando todos los obstáculos presentes. Los criterios seguidos por cada algoritmo para generar estos caminos se explicarán en sus correspondientes apartados.

Finalmente, se hallarán los caminos más óptimos mediante la aplicación del algoritmo A* y aplicando diferentes criterios de acuerdo a la penalización de tramos estrechos, en función de una función de coste temporal para comparar estos caminos óptimos.

El material empleado para el desarrollo del Trabajo de Final de Máster ha sido el software **MATLAB** y su complemento **Simulink**.

1.4. Estructura de la memoria

Este Trabajo de Final de Máster se distribuye de la siguiente forma:

El capítulo 2 se propone un estado del arte del robot MASAR. Se sitúa al lector de esta memoria en el contexto de robots móviles similares al MASAR, hablando de las diferentes tipologías de robots accionados mediante un único actuador y los diferentes tipos de robots modulares. Se muestran y mencionan ejemplos de los mismos y cuál ha sido la evolución y desarrollo de estos robots a lo largo de los últimos años.

En el capítulo 3 reside uno de los gruesos de este trabajo. Aquí se desarrolla el diseño de un controlador en posición y orientación para el robot MASAR, mejorando así el controlador propuesto en el anterior TFG. Se explica de forma detallada como se ha ejecutado el diseño del regulador, sus

principios de funcionamiento y ejemplos de simulación del mismo.

El capítulo 4 es el segundo gran bloque del trabajo. Está enfocado en el desarrollo de algoritmos de navegación para el robot MASAR en entornos conocidos. Dichos mapas de entornos conocidos también han sido diseñados y programados en este TFM. Se comentan las bases de funcionamiento sobre las que descansan los algoritmos basados en Diagramas de Voronoi y Grafos de Visibilidad, aplicándolos de forma práctica y realizando un estudio comparativo entre ambos.

En el capítulo 5 se analizan y comentan los resultados obtenidos, derivando así en la posibilidad de establecer las conclusiones a las que ha llevado este trabajo. Partiendo de estas, se propone un conjunto de trabajos futuros para profundizar en la investigación del desarrollo del robot MASAR, así como nuevas líneas de investigación.

Capítulo 2

Estado del Arte

Para situar al lector en contexto, mediante este Estado el Arte se busca dar a conocer cuál ha sido la evolución y desarrollo de robots similares al robot MASAR hasta la fecha actual. Se procederá a exponer la evolución de las dos clases de robots que abarca el robot MASAR: los robots que únicamente emplean un actuador para realizar la locomoción (SAMR, *Single - Actuator Mobile Robots*) y los robots auto-reconfigurables, capaces de combinarse con otros módulos (MSRR, *Modular Self - Reconfigurable Robots*).

2.1. Robots SAMR

Como ya se ha citado, los robots SAMR emplean un único motor eléctrico o actuador para realizar la locomoción y moverse por el entorno en el que se encuentra. Esto deriva en una serie de ventajas que recaen principalmente en la eficiencia del robot en su desplazamiento, dado que se consigue reducir costes económicos en su desarrollo o montaje, menor consumo energético, así como un peso y tamaño inferior. Gracias a estas características, los robots SAMR especialmente útiles para tareas que requieran bajo coste o recursos, y navegación por entornos complejos. Dentro de esta tipología, existen tres grandes grupos que permiten su clasificación.

2.1.1. SAMR de tipo 1

Son robots cuya capacidad de movimiento se limita a un desplazamiento a lo largo de un espacio de trabajo unidimensional, sin elementos binarios auxiliares. Por ejemplo, Soyguder y Alli, 2011 (Figuras 2.1 y 2.2) presentaron un robot de tipo octópodo, cuyo movimiento viene coordinado por un sistema formado por un actuador y un conjunto de levas, permitiéndole moverse en avance o retroceso en línea recta.

Se presenta a continuación el proyecto de Birkmeyer, Gillies y Fearing, 2011, quienes desarrollaron un robot trepador (Figuras 2.3 y 2.4) movido también por un único actuador, con la habilidad de escalar por planos y paredes verticales.

Como ejemplo final de robots SAMR de tipo 1, se presenta el robot de Zarrouk y Fearing, 2015, un robot trepador que emplea una locomoción de tipo

FIGURA 2.1: Robot octópodo de Soyguder y Alli, 2011

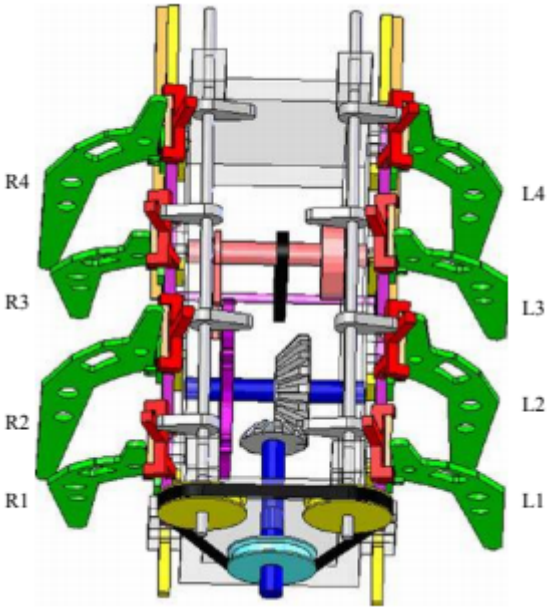

FIGURA 2.2: Disposición de motor y transmisiones del robot de Soyguder y Alli, 2011

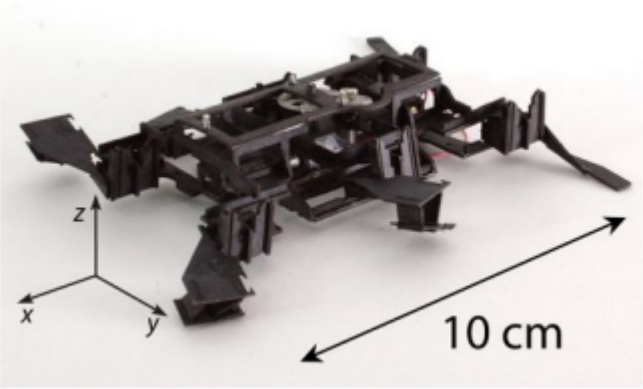

FIGURA 2.3: Robot trepador SAMR tipo 1 de Birkmeyer, Gillies y Fearing, 2011

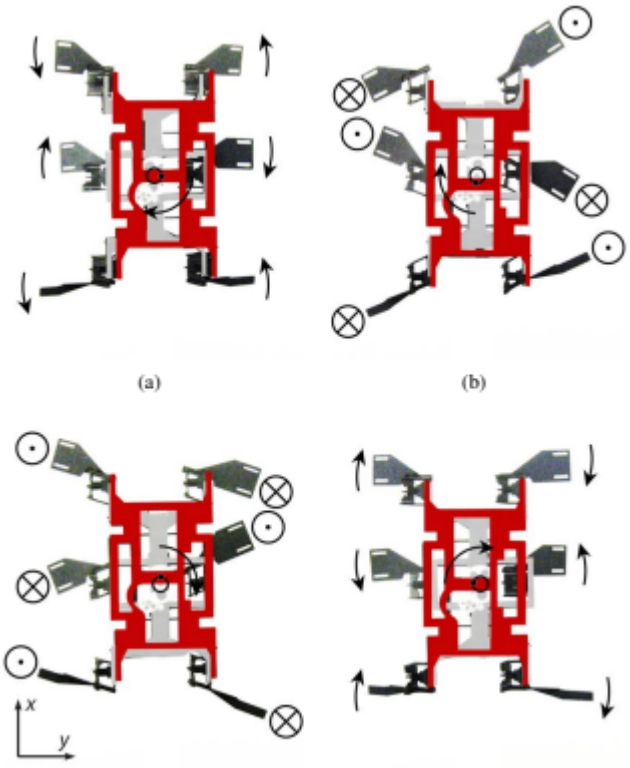

FIGURA 2.4: Vista general del movimiento del robot de Birkmeyer, Gillies y Fearing, 2011

onda, generada por un único motor (Figuras 2.5 y 2.6).

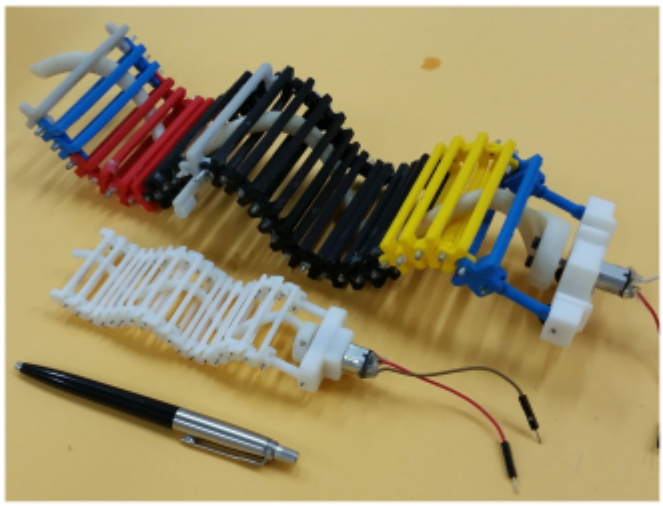

FIGURA 2.5: Robots trepadores de Zarrouk y Fearing, 2015, en dos tamaños distintos

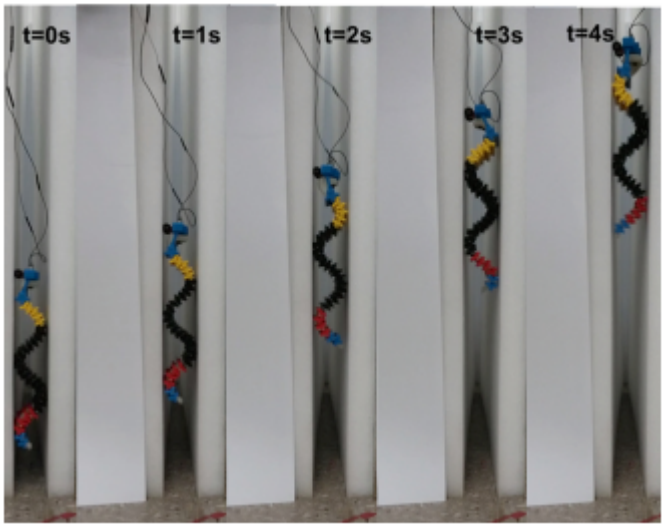

FIGURA 2.6: Fases de desplazamiento del robot de Zarrouk y Fearing, 2015

En el caso de tratar de cambiar la capacidad movimiento de un robot SAMR de tipo 1 ampliando el número de dimensiones de desplazamiento, es requisito imprescindible aumentar el número de motores de su sistema de locomoción, por tanto, dejarían de considerarse SAMR. Birkmeyer, Gillies y Fearing, 2011, y Zarrouk y Fearing, 2015, proponen una configuración de este tipo para sus respectivos robots.

2.1.2. SAMR de tipo 2

En contraposición a la tipología anterior, el movimiento de estos robots no se limita a un desplazamiento en línea recta con avance y retroceso, pues son capaces de controlar su posición y orientación sobre un plano, siendo la dimensión de su espacio de trabajo mayor a 1. Además del actuador que ostentan, los robot SAMR de tipo 2 están caracterizados por poseer actuadores binarios que permiten al robot modificar su topología, cambiando así el efecto que el actuador principal produce sobre el conjunto del robot. Tomando estos actuadores binarios auxiliares como un complemento al actuador principal, el aprovechamiento de éste crece de forma notable, otorgando al robot la capacidad de realizar movimientos tales como: desplazamientos en línea recta, cambios de dirección, cambios de plano, saltos, etc...

Es necesario remarcar que estos actuadores binarios no tienen por qué ser obligatoriamente motores o actuadores, también pueden ser ventosas de succión, electroimanes, embragues, etc...

A continuación, se muestran diversos ejemplos de robots SAMR de tipo 2.

Sfeir, Shammas y Asmar, 2014 presentaron un robot móvil compuesto por dos ruedas (Figura 2.7). Estas ruedas son controladas por un embrague, el cual al entrar en acción desplaza una plataforma central ("Carriage", Figura 2.7) de modo que el centro de gravedad del robot se mueve hacia el costado de la plataforma central, decreciendo así el radio de la rueda hacia la que se ha desplazado el centro de gravedad, y consiguiendo una rotación alrededor de esta rueda cambiando la dirección de avance del robot. Al volver a activar el embrague, se desplaza de nuevo esta plataforma central, modificando de nuevo el radio de las ruedas.

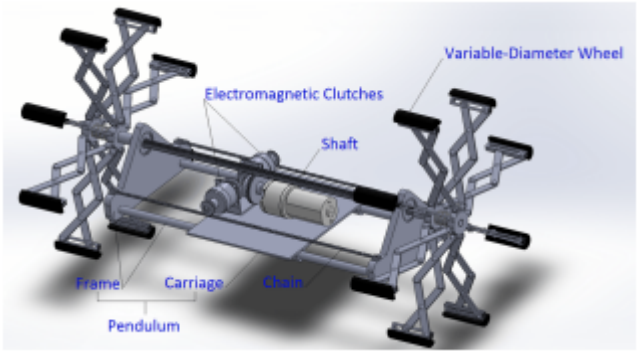

FIGURA 2.7: Robot SAMR tipo 2 de Sfeir, Shammas y Asmar, 2014

Cheng y col., 2010, investigadores del MIT, proponen un robot de tipología serpiente gobernado por un motor de tipo tendón (Figuras 2.8 y 2.9). El modo de funcionamiento de este robot resulta cuanto menos peculiar. Está formado por varios segmentos soldados, de modo que al aplicar corriente a alguna de estas uniones, éstas se funden o derriten de forma temporal

para modificar la geometría del robot. Derritiendo las uniones correctas y empleando su único actuador, este robot logra desplazarse en línea recta, cambiando de dirección.

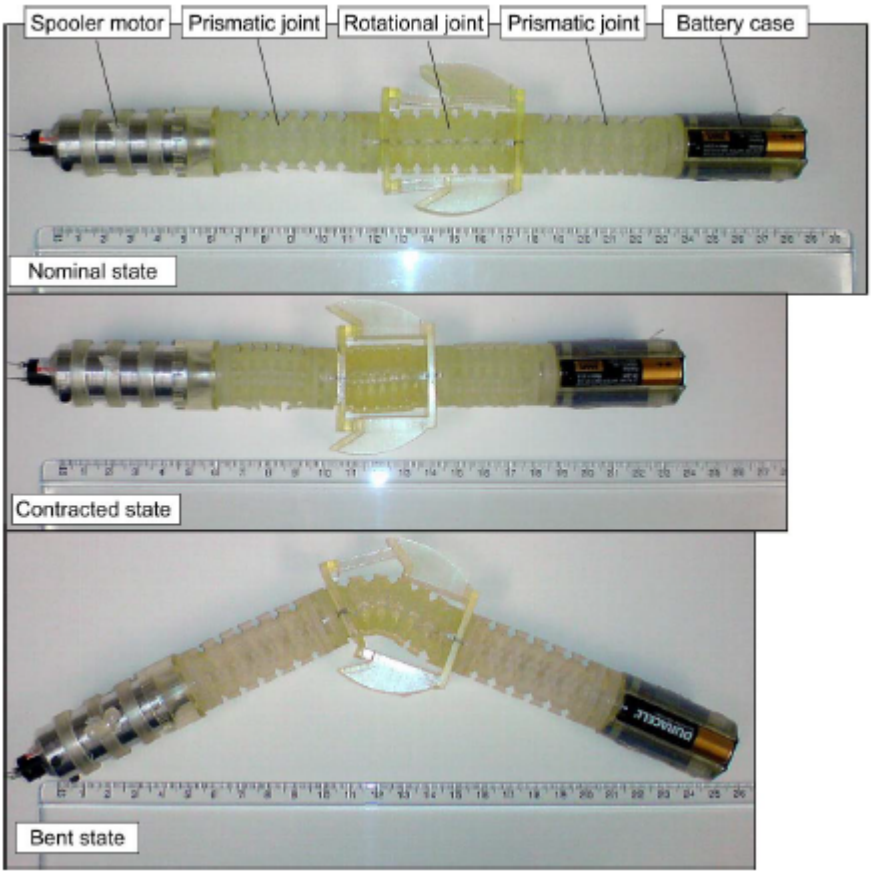

FIGURA 2.8: Robot de Cheng y col., 2010, en diferentes configuraciones

Finalmente, como último ejemplo, se comenta el robot DynaRoACH de Hoover y col., 2010 (Figura 2.10), el cual trata de copiar la apariencia y movimiento de una cucaracha. Su característica más importante es la habilidad para modificar la rigidez de sus patas gracias a los *shape-memory alloys* (Figura 2.11), modificando en consecuencia la dirección de avance.

2.1.3. SAMR de tipo 3

Este último tipo de robots SAMR se enfoca en robots móviles sub-actuados que, empleando únicamente el actuador principal, son capaces de realizar movimientos en dos dimensiones. Dada la carencia de actuadores auxiliares, la ejecución de un tipo de movimiento u otro vendrá dada en función de las características de la señal de control enviada al actuador.

Refael y Degani, 2015, lanzaron un robot nadador movido por un único actuador (Figura 2.12), el cual realiza un avance recto al recibir una señal de

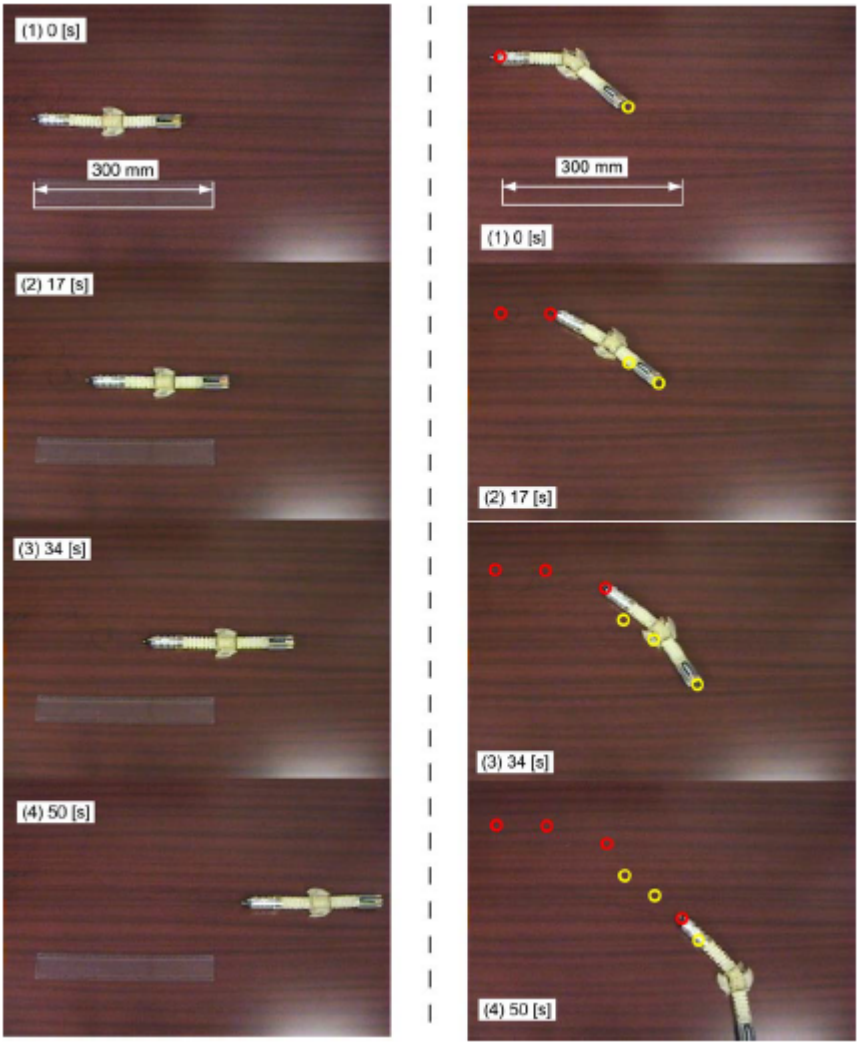

FIGURA 2.9: Locomoción del robot de tipo serpiente

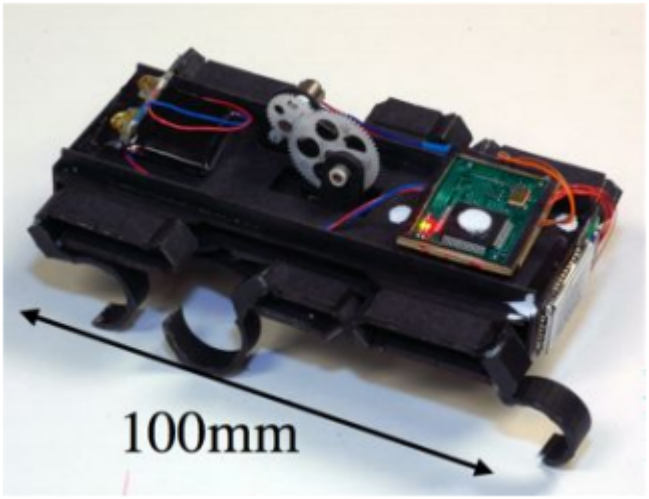

FIGURA 2.10: Robot DynaRoACH

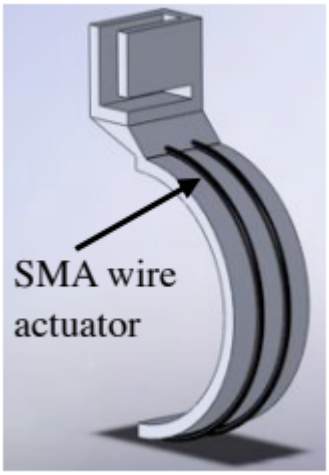

FIGURA 2.11: Shape-memory alloys

control de frecuencia constante, y cambia de dirección tras excitarlo con frecuencias diferentes.

Dharmawan y col., 2017 presentaron un micro-robot conformado por un piezoeléctrico (Figura 2.13), capaz de desplazar al robot en línea recta, con posibilidad de giro en función de la frecuencia de resonancia a la que se excite (Figura 2.14).

Para concluir con los robots SAMR, Zhao y col., 2013, presentaron un novedoso robot saltador (Figura 2.15) con la habilidad de ejecutar acciones tales como: comprimir un muelle permitiéndole saltar, enderezarse tras caer (Figura 2.16) y modificar la dirección para próximo salto desde el suelo (Figura 2.17). La decisión para realizar una acción u otra viene determinado por el sentido de giro de su único actuador.

2.2. Robots MSRR

Los robot auto-reconfigurables son considerados sistemas multi-robot, esto es, un conjunto compuesto por un elevado número de mini robots más sencillos, con la capacidad de agruparse entre ellos para lograr conformar robots más complejos. La formación de este nuevo robot persigue lograr una mejora en ciertos ámbitos o tareas respecto a los robots simples, por ejemplo, consiguiendo una mayor movilidad y capacidad de resolución de problemas. Los robots elementales o módulos simples pueden ser todos iguales (MSRR homogéneo), o existe la posibilidad de lograr diferentes formas y características (MSRR heterogéneo).

Las principales ventajas de los robots MSRR, según Yim, 2007, son tres:

- Versatilidad: pueden adaptarse a diferentes situaciones y problemas gracias a la capacidad que tienen para reconfigurar su morfología.

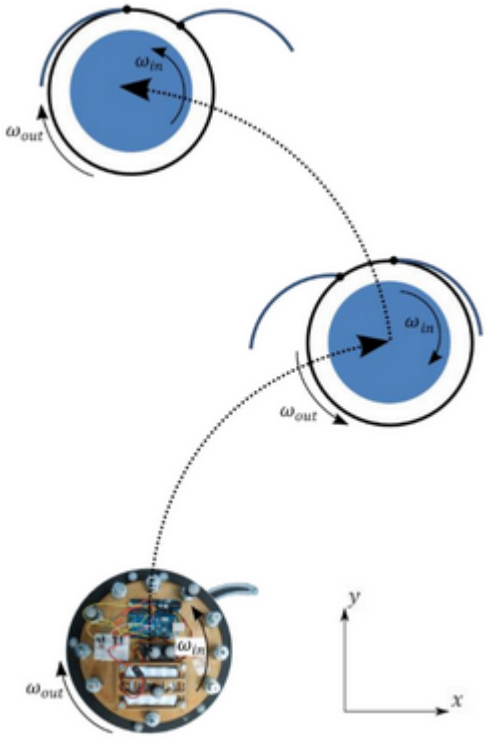

FIGURA 2.12: Conjunto de movimientos del robot nadador de Refael y Degani, 2015

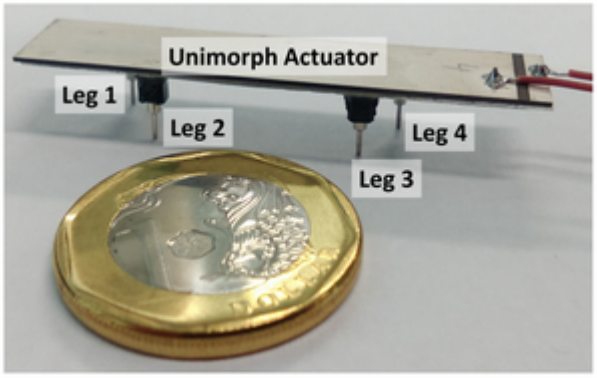

FIGURA 2.13: Micro-robot de Dharmawan y col., 2017

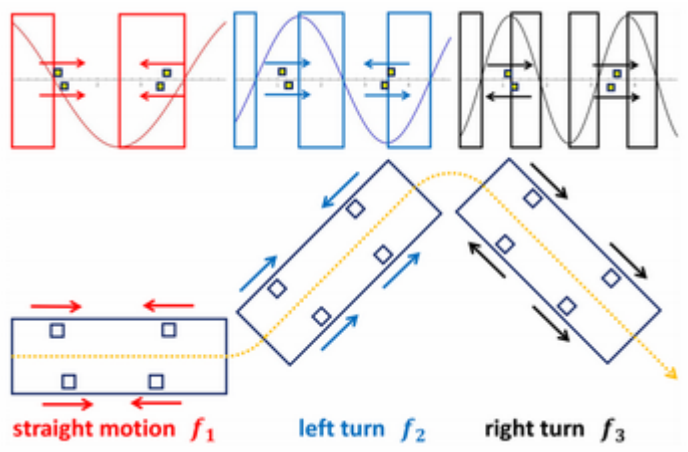

FIGURA 2.14: Movimiento generado por las diferentes frecuencias de resonancia

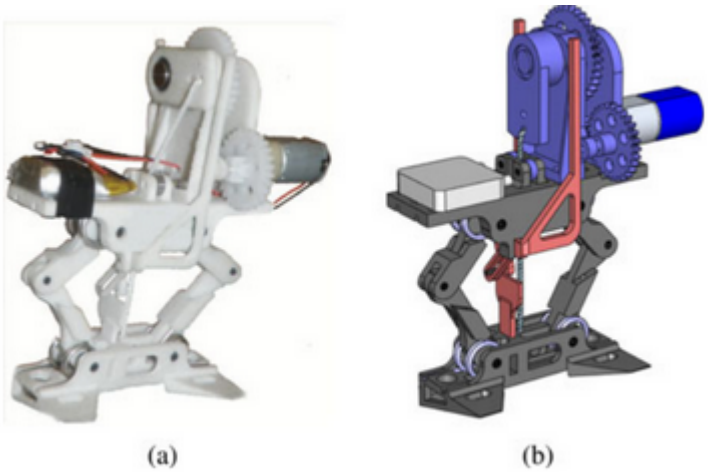

FIGURA 2.15: Prototipo del robot saltador de Zhao y col., 2013

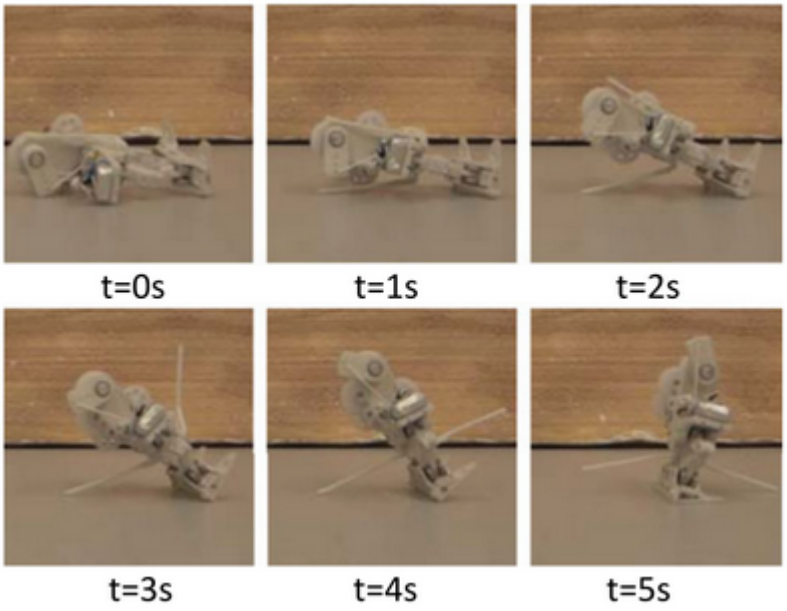

FIGURA 2.16: Enderezamiento tras caída

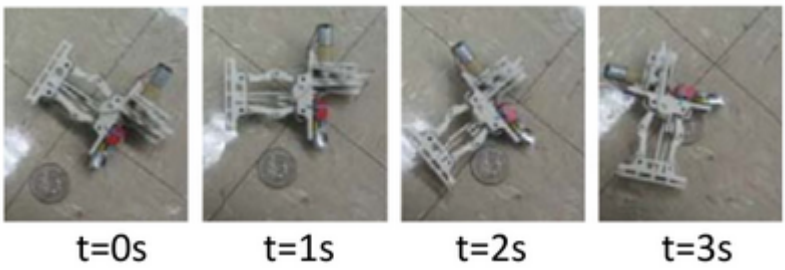

FIGURA 2.17: Cambio de dirección desde el suelo

- Robustez: los módulos individuales pueden ser reemplazados fácilmente ante problemas que generen daños en el robot
- Bajo coste: al estar formados por un elevado número de módulos idénticos, el coste de producción de estos mini robots es relativamente bajo gracias a las economías de escala (mayor volumen de producción deriva en un menor coste unitario).

Teniendo en cuenta estas ventajas, algunas de las principales funciones de esta tipología de robots pueden ser: tareas de búsqueda y rescate, exploración espacial, inspección industrial o tareas domésticas (Yim, 2007).

Existe un número elevado de robots de tipo MSRR, ampliamente analizados y clasificados. Para este Trabajo de Final de Máster nos centraremos en comentar aquellas clasificaciones y ejemplos de robots que pueden resultar de mayor valor para la investigación del robot MASAR. Existen diferentes criterios para clasificar los robots MSRR. Lo habitual es clasificarlos en función del tipo de conexión que emplean sus módulos, existiendo tres tipos: tipo red, tipo cadena y tipo MCC.

2.2.1. MSRR de tipo red (*lattice type*)

Se trata de distribuciones en la que los módulos robóticos simples pueden ocupar estrictamente posiciones y orientaciones discretas dentro de una red (redes cúbicas o hexagonales). Esta distribución de los módulos robóticos suele derivar en una maniobrabilidad más limita sobre el robot final. Sin embargo, el factor de ocupar posiciones discretas facilita enormemente el análisis de estos robots. Un ejemplo representativo de esta tipología son los M-Blocks de Romanishin, 2015 (Figura 2.18).

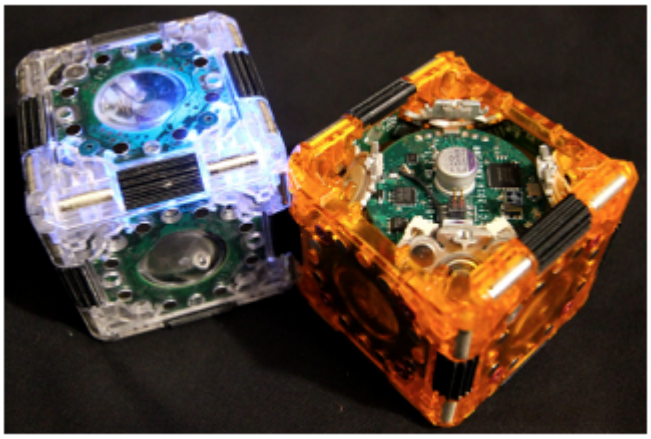

FIGURA 2.18: Módulos M-Blocks

Los M-Bloks son robots auto-reconfigurables que se enlazan magnéticamente con sus vecinos, formando así redes de categoría cúbica (Figura 2.19).

	Traverse	Horizontal Traverse	Vertical Traverse	Horizontal Convex	Vertical Convex	Horizontal Concave	Vertical Concave	Corner Climb	Stair Step
Illustration									
Attempted	41	20	20	20	20	20	20	20	20
Success	100%	70%	80%	95%	100%	55%	90%	100%	95%

FIGURA 2.19: Asociación de M-Blocks

2.2.2. MSRR de tipo cadena (*chain type*)

Se basan en una distribución en la cual los módulos interconectados pueden alcanzar cualquier pose arbitraria en el espacio de trabajo. Mientras que en un robot de tipo red los módulos sólo podían adoptar un conjunto de poses discretas con relación estricta al módulo vecino, los robots de tipo cadena pueden optar por cualquier configuración continua compatible con la unión a su vecino.

Un ejemplo representativo de esta tipología puede ser el robot CONRO (Shen, 2000). Los módulos que forman este robot (Figura 2.20 (a)) son autónomos, y están compuestos por dos baterías, un microcontrolador, dos motores y cuatro transmisores/receptores infrarrojos. En la Figura 2.20 (b), se muestra una combinación de estos formando un robot hexápodo.

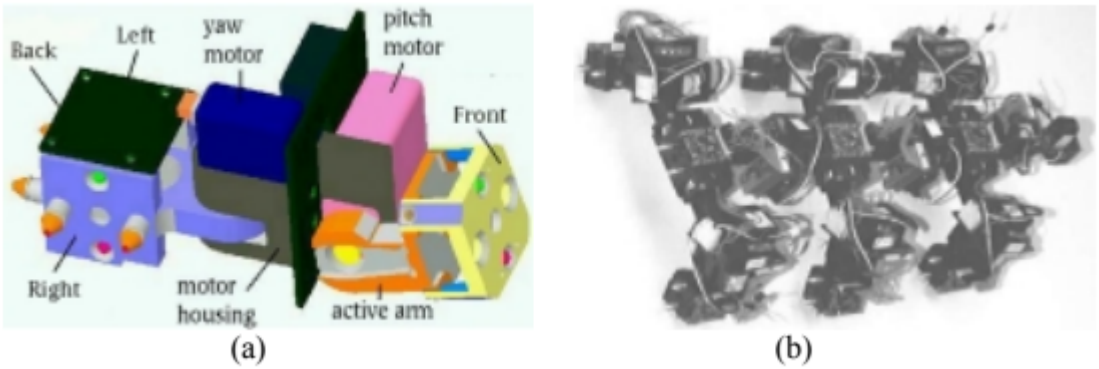

FIGURA 2.20: Robot CONRO: módulo independiente y hexápodo

El robot MASAR se encuentra dentro de esta tipología de robots tipo cadena.

2.2.3. MSRR de tipo MCC (*Mobile Configuration Change*)

El último grupo de clasificación, de especial interés para el robot MASAR, es la propuesta por Moubarak y Ben-Tzvi, 2012, quienes clasificaron los robots de tipo MSRR en función de si disponen de movilidad eficiente en el plano (robots MCC, Mobile Configuration Change) o no.

Dentro del segundo grupo (sin movilidad eficiente en el plano), lugar en el que se incluyen la mayoría de robots MSRR, los módulos sólo presentan movilidad eficiente cuando están asociados a otros módulos, formando robots más completos. Para la investigación del robot MASAR resultan más relevantes los robots MCC, ya que los movimientos eficientes mejoran el proceso de reconfiguración de los módulos, así como el desempeño de los robots MSRR en operaciones no-supervisadas, permitiendo al conjunto de robots trabajar como una colmena o una unidad compacta (Moubarak y Ben-Tzvi, 2012). Atendiendo a este grupo de robots MCC, Moubarak y Ben-Tzvi, 2012, expertos en dicha tipología, presentan varios robots de esta familia, entre los que destacan: Uni-Rovers (Figura 2.21), JL-I/II (Figura 2.22), Millibots (Figura 2.23) y Amoeba (Figura 2.24).

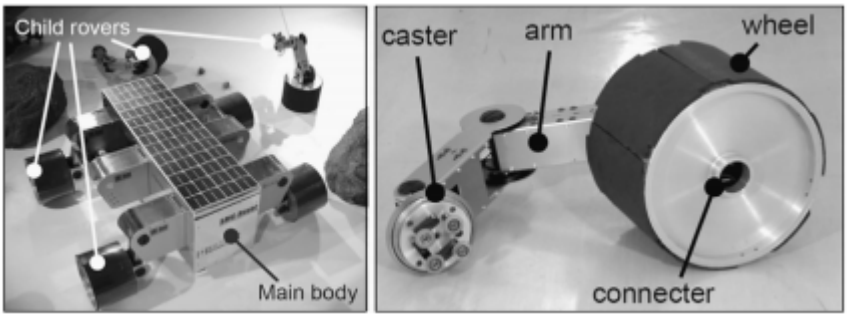

FIGURA 2.21: Robot Uni-Rovers

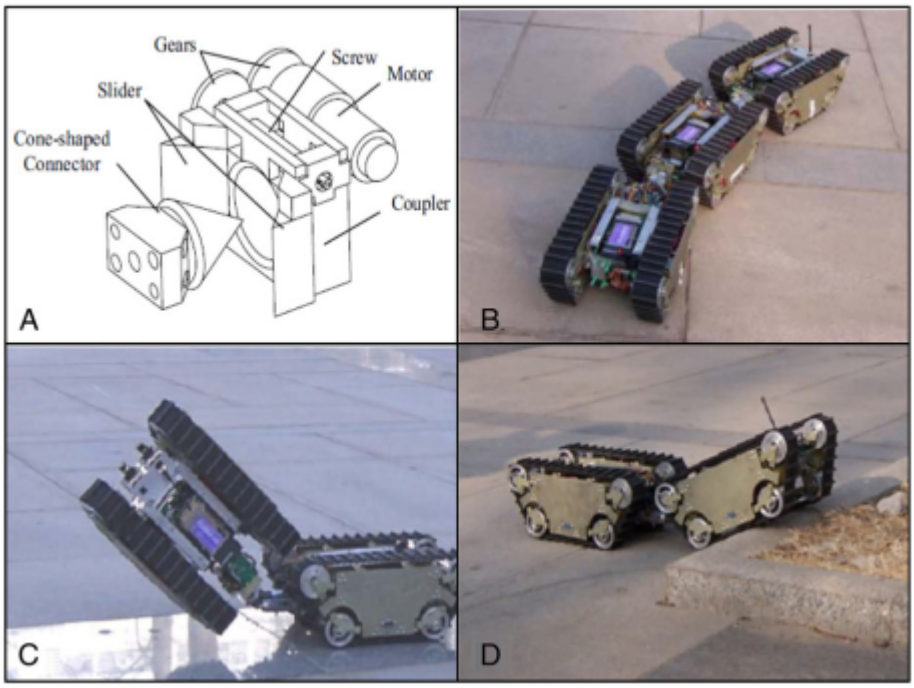

FIGURA 2.22: Robot JL-I/II

El principal inconveniente que presentan los robots MCC es que cada uno sus módulos requieren varios motores, para lograr una movilidad eficiente,

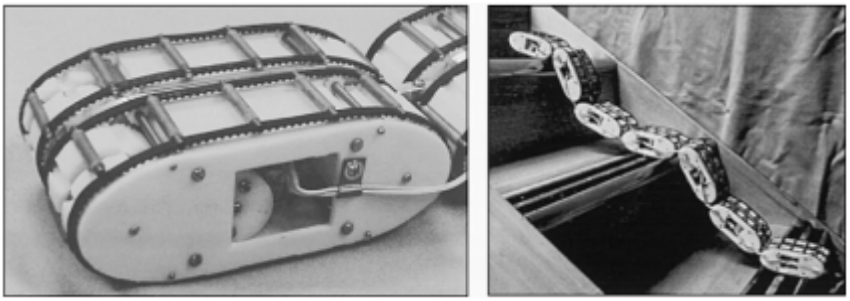

FIGURA 2.23: Robot Millibots

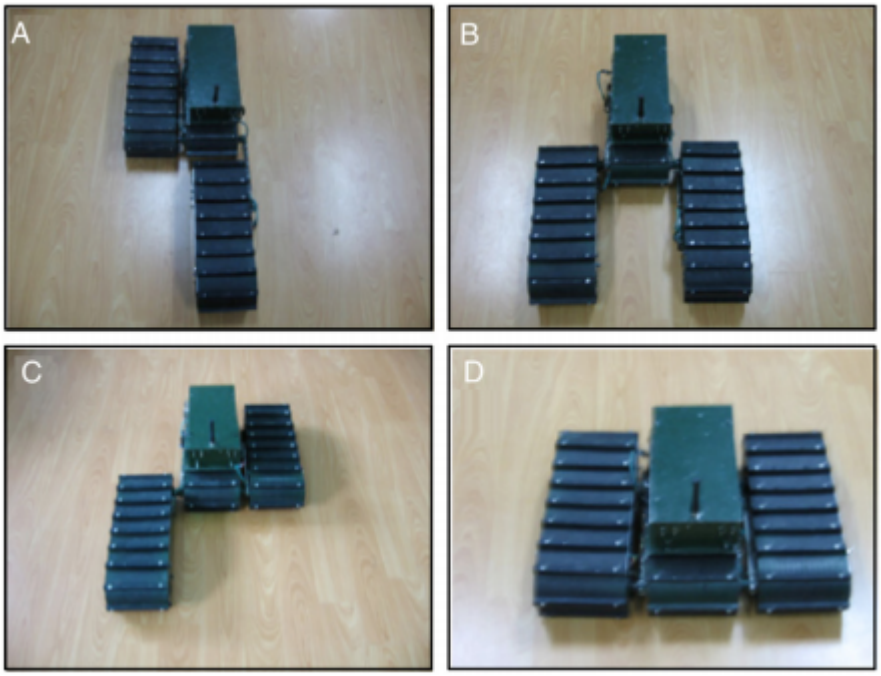

FIGURA 2.24: Robot Amoeba

o para otorgar movimientos articulados a los módulos vecinos. Por ejemplo, los módulos de los robots Uni-Rovers y JL-I/II (Figuras 2.21 y 2.22) necesitan cuatro motores, y los módulos de Millibots y Amoeba (Figuras 2.23 y 2.24) utilizan tres motores. A mayor número de motores por módulo, mayor será su peso y más compleja será su morfología, puesto que por cada motor añadido también será necesario incorporar hardware adicional como encoders, drivers, etc... (Wolfe et al., 2012).

Como conclusión, un robot modular con alta movilidad individual, como es el robot MASAR, supera a estos robot en esos aspectos, debido a que es capaz de experimentar una gran movilidad gracias a la combinación de módulos, y una gran eficiencia y bajo coste de operación debido al empleo de un único actuador.

Capítulo 3

Diseño de un controlador en posición y orientación para el Robot MASAR

El capítulo número 3 de este trabajo de final de máster se enfoca en el diseño de un controlador en posición y orientación para el robot MASAR, mejorando así los controladores diseñados en el anterior TFG y comentados brevemente en el apartado 1.2 de esta memoria.

En dicho TFG, se propuso la implementación de controladores para resolver la planificación de trayectorias del robot MASAR, estudiando la posibilidad de diseñar una solución basada en la cinemática directa, empleando el modelo cinemático del robot, mediante la ecuación en velocidad que modela la velocidad robot MASAR. De forma generalizada, lo que se realizó fue, tras obtener la ecuación que modela la velocidad de giro del robot, diseñar y estudiar diversos modos de control que permitieran al robot alcanzar un determinado objetivo o recorrer una trayectoria predefinida.

Previamente a comentar las nuevas funcionalidades del controlador implementado en el presente trabajo, es necesario recordar cómo se determinó el modelo cinemático del robot MASAR, así como describir las características y principios de funcionamiento de los dos controladores diseñados en su día, tomados como base para este trabajo.

Este capítulo estará compuesto por las siguientes secciones:

1. Modelo cinemático del robot MASAR
2. Control en posición
3. Nuevo controlador propuesto: control en posición y orientación

3.1. Modelo cinemático del robot MASAR

El primer bloque de este capítulo trata de recordar la obtención de las ecuaciones y expresiones que modelan la cinemática del robot, necesarias

para controlar y simular el movimiento de éste mediante el empleo de reguladores de velocidad. Se modela la velocidad del punto medio partiendo de la posición conocida del mismo, la cual variará en función del tipo de movimiento que realice el robot, directamente dependiente de qué pivote pega y del valor del ángulo a rotar.

En la Figura (3.1) se muestran las principales variables que se emplean en este apartado:

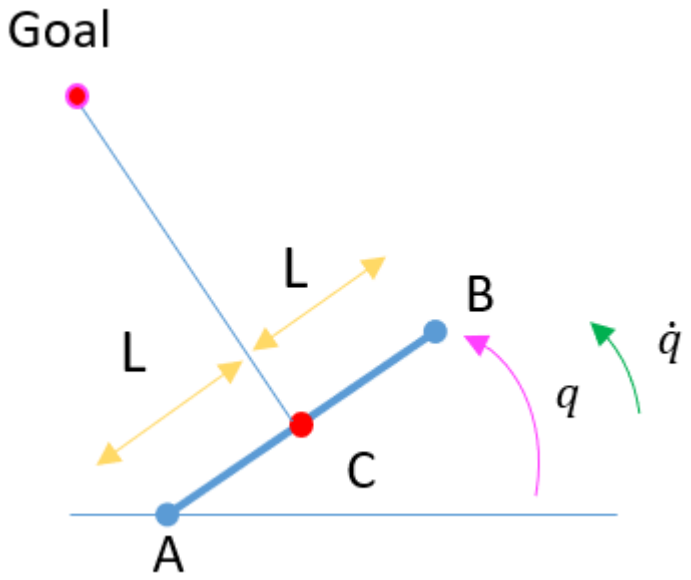

FIGURA 3.1: Variables partícipes del modelo cinemático

Nombrando al punto medio del robot como C (Figura 3.1), su posición se calcula hallando el promedio entre los pivotes A y B (Figura 3.1), como se muestra en la Ecuación (3.1), puesto que son conocidas sus posiciones en cada instante de tiempo.

$$\dot{C} = (\dot{A} + \dot{B})/2 \tag{3.1}$$

Comenzando con el desarrollo, partiendo de la posición de los pivotes A y B, se halla la velocidad de lo mismos, para así posteriormente poder sustituirlas en la ecuación 3.1.

Ambas posiciones de los pivotes dependen estrechamente entre ellas. Por ejemplo, si pretendemos definir la posición del pivote B en coordenadas del mundo, ésta se obtiene a partir de la posición global del pivote A y del ángulo que forma el robot con la horizontal (q, Figura 3.1). En la Ecuación 3.2 se muestra la expresión trigonométrica.

$$B = A + 2L * \begin{bmatrix} \cos(q) \\ \sin(q) \end{bmatrix} \tag{3.2}$$

Aplicando el operador derivada sobre esta expresión, se obtiene una ecuación dependiente de la velocidad de ambos pivotes, la posición angular q y la velocidad angular q *(punto)*:

$$\dot{B} = \dot{A} + 2L * \begin{bmatrix} -\sin(q) \\ \cos(q) \end{bmatrix} * \dot{q} \tag{3.3}$$

Pensando en el funcionamiento del robot MASAR, el robot fijará un pivote u otro en función del valor de una variable binaria, y la velocidad del pivote fijado será igual a cero durante este período. Si el pivote fijado es A (bin = 0), la velocidad de dicho pivote será cero y se obtiene la velocidad del pivote B durante el desplazamiento del robot según la Ecuación 3.4, haciendo 0 la velocidad de A.

$$\dot{B} = 2L * \begin{bmatrix} -\sin(q) \\ \cos(q) \end{bmatrix} * \dot{q} \tag{3.4}$$

En contraposición, cuando el pivote fijado es B (bin = 1), la velocidad del pivote A vedrá determinada por la Ecuación 3.5.

$$\dot{A} = -2L * \begin{bmatrix} -\sin(q) \\ \cos(q) \end{bmatrix} * \dot{q} \tag{3.5}$$

Para poder aplicar estos términos en la ecuación 3.1, es necesaria la inclusión de la variable binaria, pues esta es la base de decisión de qué pivote se fija en cada instante, y por tanto, parte esencial para el desplazamiento del robot. Ésta variable bin será la encargada de poner a cero la velocidad del pivote fijado, y mantener la expresión de velocidad sobre el pivote libre. La Ecuación 3.6 consigue fielmente este comportamiento, este es, que la velocidad del punto medio sea igual a la velocidad del pivote no fijado dividido entre dos.

$$\dot{C} = ((\dot{A} * bin) + (\dot{B} * (1 - bin)))/2 \tag{3.6}$$

Sustituyendo las Ecuaciones 3.4 y 3.5 en la Ecuación 3.6, se obtiene una expresión más extendida (Ecuación 3.7).

$$\dot{C} = 1/2 * (-2L * \begin{bmatrix} -\sin(q) \\ \cos(q) \end{bmatrix} * \dot{q} * bin + 2L * \dot{q} * \begin{bmatrix} -\sin(q) \\ \cos(q) \end{bmatrix} * (1 - bin)) \tag{3.7}$$

Simplificándola, permite obtener la siguiente expresión (Ecuación 3.8) para definir de forma tan completa como sencilla la velocidad del punto medio, en función de las variables clave y elementales del desplazamiento del robot:

variable binaria (*bin*), posición angular (q), velocidad angular (q *(punto)*) y longitud del robot entre dos (L).

$$\dot{C} = L * \dot{q} \begin{bmatrix} -\sin(q) \\ \cos(q) \end{bmatrix} * (1 - 2 * bin) \tag{3.8}$$

De este modo, en la Figura 3.2 se muestra cómo queda el bloque de Simulink que modela el robot MASAR.

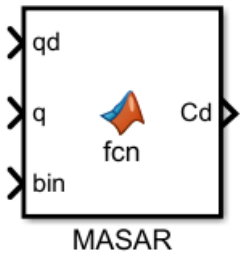

FIGURA 3.2: Bloque modelado MASAR

3.2. Control en posición

Previamente a explicar el diseño del controlador en posición y orientación, se introducirán de forma resumida los controladores implementados en el TFG, y que han servido como base a mejorar en el presente TFM. Se propusieron dos controladores.

3.2.1. Primer controlador propuesto

El primero de ellos basaba su ley de control en pegar en cada instante el pivote más cercano al punto de destino. Esta estrategia aseguraba una gran precisión de llegada al punto objetivo. Sin embargo, el coste del desplazamiento resultó ser muy elevado debido al gran número de pegadas y despegadas de los pivotes que debía realizar el robot MASAR.

Modo de funcionamiento

Únicamente se debían seguir tres leyes básicas:

1. Estimar cual era el pivote más cercano al objetivo mediante cálculo de distancias, para así dar valor a la variable binaria
2. Identificar el sentido de giro, es decir, el signo de la velocidad, para generar avance en función del pivote fijado
3. Controlar la velocidad en función de la cercanía del punto medio del robot al punto objetivo

Dicho controlador recibía como entrada el punto objetivo y estimaba, a partir de él y de la posición del robot en dicho instante, la variable binaria, la velocidad angular *q_d* y la posición angular *q* integrando la velocidad. Dichas variables son las entradas del modelo cinemático del robot MASAR. El esquema se muestra en la Figura 3.3.

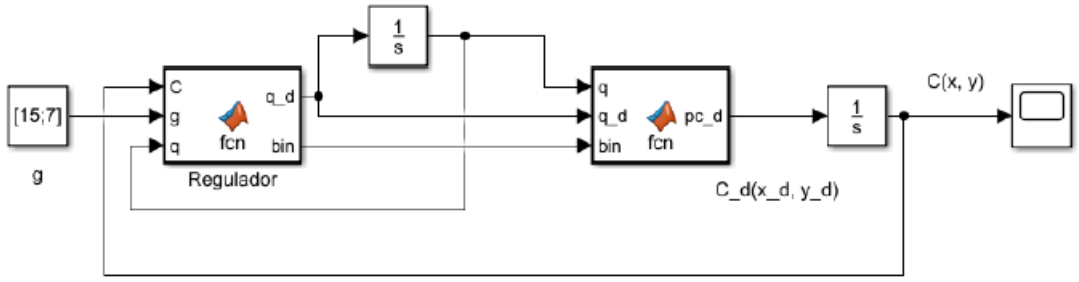

FIGURA 3.3: Diagrama del primer controlador

Ejemplos y conclusión

Para concluir, se muestran algunos ejemplo de empleo del controlador comentado. En las Figura 3.4 y 3.5 se muestra al robot MASAR dirigido por el controlador propuesto hacia un punto objetivo.

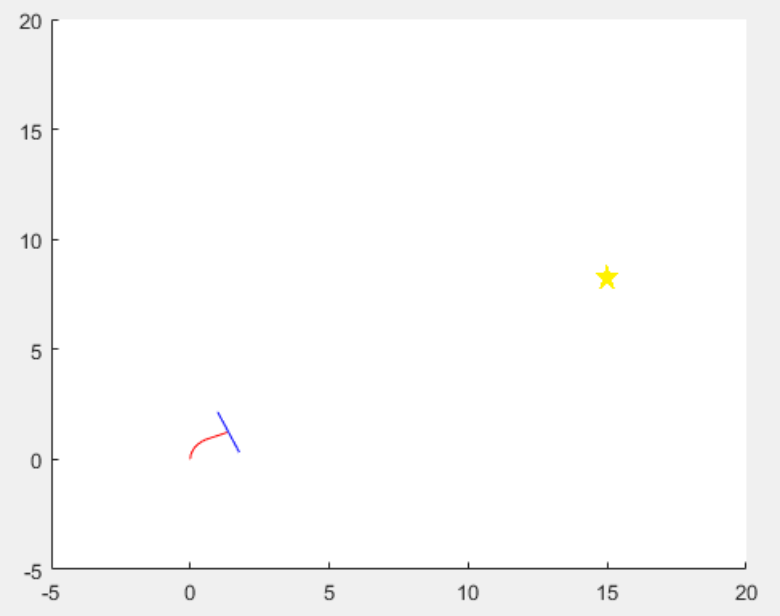

FIGURA 3.4: Inicio de movimiento hacia punto objetivo

Como último ejemplo, se muestra dicho control aplicado al recorrido de una circunferencia (Figura 3.39).

Se muestra una gráfica (Figura 3.7) para visualizar la excesiva variación en el valor de la variable binaria (entre 0 y 1), y confirmar que el número

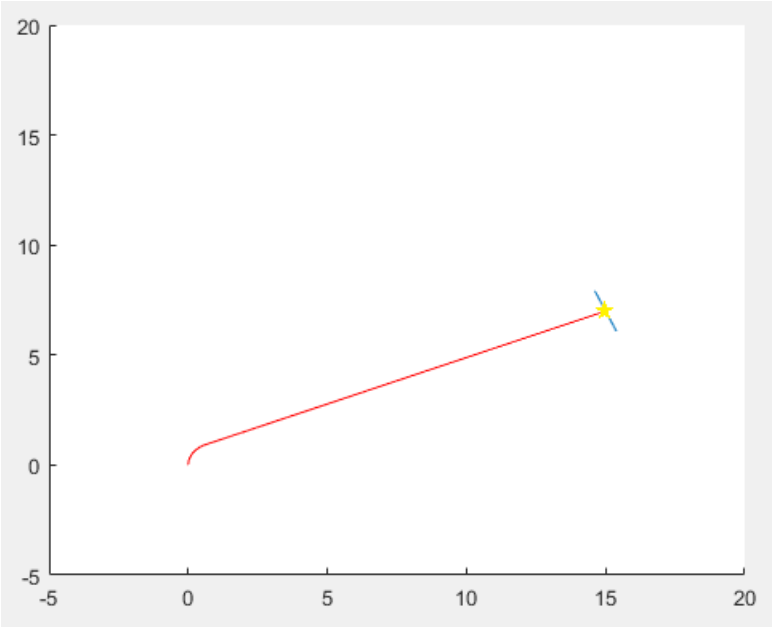

FIGURA 3.5: Final de movimiento hacia punto objetivo

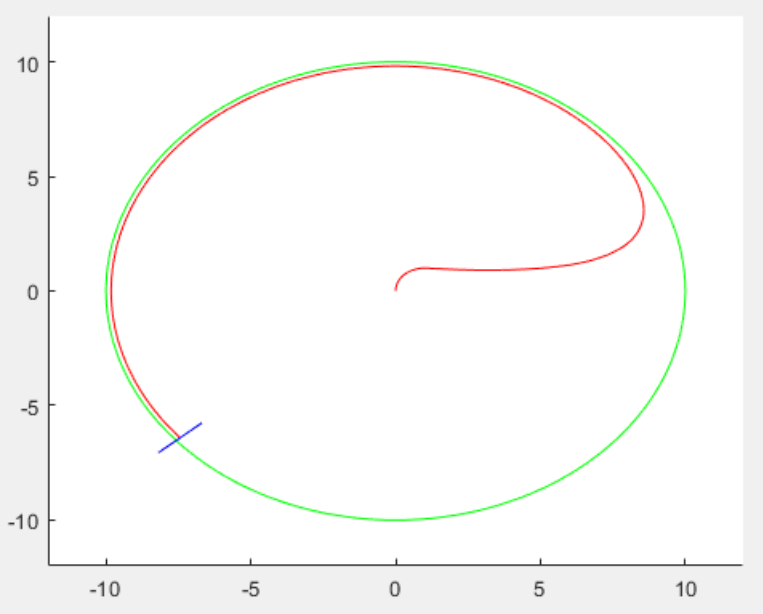

FIGURA 3.6: Recorrido de una circunferencia empleando el primer controlador

de pegues y despegues es desproporcionado. Esta gráfica se ha obtenido del primer ejemplo durante únicamente 2 segundos de simulación (Figura3.4).

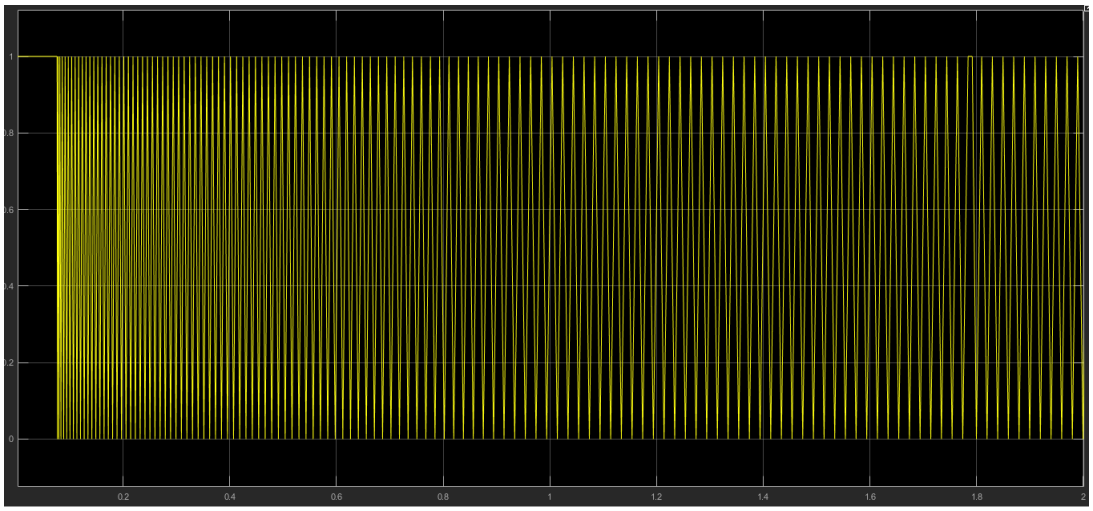

FIGURA 3.7: Variación de la variable binaria empleando el primer controlador

Como se puede apreciar en los ejemplos, el regulador que se propuso funciona de forma exitosa. Logra que el robot alcance un punto objetivo y también que recorra una trayectoria suministrada. Pero tiene un defecto muy relevante, el elevado número de pegadas y despegadas que efectúa. Al diseñar un modo de control en el que el robot pega el pivote más cercano en cada instante, cuando el robot se encara hacia el objetivo, situándose prácticamente perpendicular al mismo, el desplazamiento del robot comienza a estar basado una alternancia desproporcionada entre pegadas y despegadas de los pivotes, lo que conlleva un elevado coste de operación.

3.2.2. Segundo controlador propuesto

Posteriormente, en base a los resultados obtenidos con el controlador anterior, se propuso un nuevo regulador que tratara de mejorar el principal defecto que sufría el regulador anterior, éste era el elevado coste de operación debido al gran número de pegadas y despegadas que ejecutaban los pivotes.

Con el regulador que se describirá a continuación, se logró implementar un movimiento basado en una locomoción del robot en giros de 180º, siendo estos giros tan amplios los que producen un mayor avance con el menor número de pegadas y despegadas posibles.

El regulador que se propuso se considera de carácter híbrido, ya que dispone de dos modos de control. Durante el desplazamiento del robot hacia el objetivo, se ejecutan las acciones de control correspondientes para generar giros de 180º. Cuando el punto medio C del robot MASAR se encuentra próximo al objetivo, el control pasará a ser el sugerido en el apartado anterior, pues logra conseguir una precisión milimétrica y situará el punto medio del

robot sobre el punto objetivo, precisión que la locomoción basada en giros de 180º no puede asegurar casi nunca.

Modo de funcionamiento

El modo de funcionamiento de este segundo controlador se dividía en tres fases.

La **Fase 1** se corresponde con el inicio del movimiento.

En la primera fase de funcionamiento, el regulador busca comprobar al inicio del movimiento ($t = 0$), cuál es el pivote más cercano al objetivo (será el pivote a fijar para generar avance) y en qué sentido debe rotar el pivote libre para alinear el robot con la recta imaginaria que une el pivote fijado con el punto objetivo (Figura 3.8).

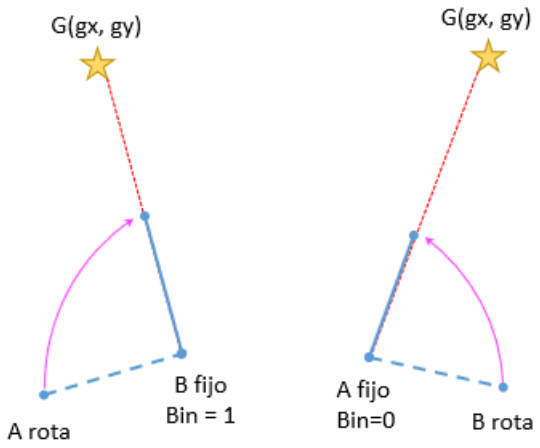

FIGURA 3.8: Identificación de pivote más cercano al objetivo y alineación

La **Fase 2** trata de estimar la cercanía del robot al objetivo para cambiar el modo de control al primer controlador propuesto, y así asegurar la precisión final.

Ejecutada la orientación propuesta de fase 1, el posterior movimiento será aquel basado en giros de 180º para desplazarse hacia la meta, pero el árbol de prioridades implementado en el regulador comprueba primero si el punto medio del robot se encuentra cercano al punto objetivo. Si esta comprobación resulta ser cierta (Figura 3.9), el controlador cambiará el modo de control a aquel implementado en el primer controlador, para así afirmar que el robot alcanza al objetivo de forma precisa, y si por contrario el robot aún se encuentra alejado, el MASAR se desplazará siguiendo las acciones de control que se detallarán en la fase tres.

La **Fase 3** es la encargada de ejecutar las acciones de control necesarias dentro del controlador para realizar los movimientos de 180º.

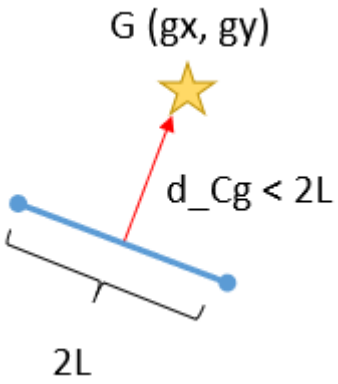

FIGURA 3.9: Distancia objetivo - punto medio

El comportamiento del regulador es el siguiente, partiendo de la posición lograda en la fase 1:

1. Comprobación de pivote actualmente fijado
2. Calculo de la distancia perpendicular entre pivote libre hasta la recta que une el pivote fijado con el objetivo
3. Si dicha distancia es menor al 1 % de la longitud total del robot MASAR, y además el pivote más cercano al objetivo es el libre, el regulador considera que se ha ejecutado un giro de 180º completo y cambiará el pivote fijado
4. Por el contrario, aún no debe producirse el cambio de pivote, pues el robot debe girar a velocidad constante hasta que se produzca la rotación completa de 180º
5. Vuelta al paso 1 hasta que el punto medio del MASAR esté cerca del objetivo y entre en fase 2, con cambio e modo de control

El controlador presentaba un problema muy a tener en consideración. La variable clave en todo este proceso es la variable binaria *bin*, y para asignar un nuevo valor a la misma, se debe tener en cuenta su valor en el instante anterior, es decir, qué pivote hay fijado actualmente. Siendo esta variable binaria una salida del regulador, y queriendo llevar su valor hasta la entrada del regulador se crea un lazo algebraico. La solución provisional propuesta fue introducir un retardo discreto de una unidad en la variable binaria que realimentamos hasta la entrada del regulador (Figura 3.10).

El diagrama completo quedó distribuido de esta forma (Figura 3.11):

Ejemplos y conclusión

Como ejemplo de aplicación para este controlador, se propuso una simulación en la que el robot MASAR, partiendo desde el origen, trate de alcanzar

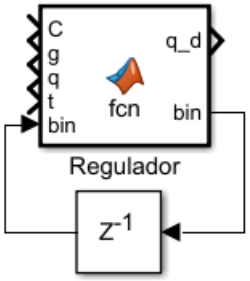

FIGURA 3.10: Retardo en la variable binaria

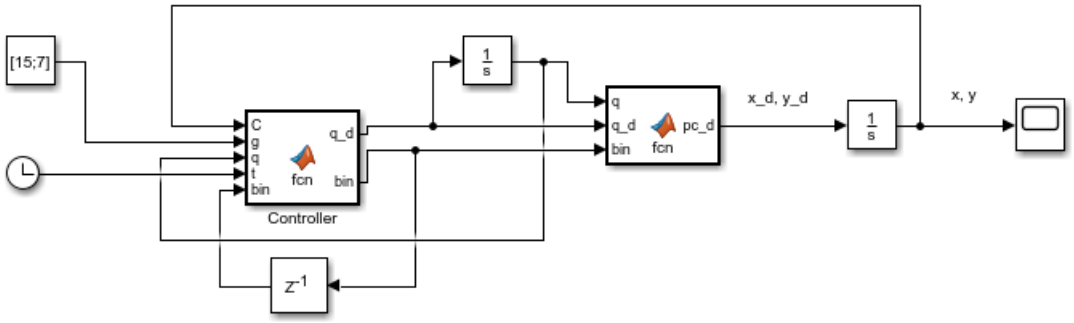

FIGURA 3.11: Diagrama del segundo controlador

un punto objetivo en [-10,-4]. Dicho desplazamiento se representa en la Figura 3.12, con alineación inicial y desplazamientos de 180º, y con el posterior cambio de modo de control para alcanzar el objetivo en la Figura 3.13.

En comparativa con el controlador anterior, se muestra como varió con el nuevo controlador el valor de la variable binaria en la Figura 3.14:

3.3. Control en posición y orientación

Una vez introducidos los antecedentes del presente controlador, se procederá a abordar el primer gran bloque de este Trabajo de Final de Máster.

El anterior controlador (Segundo controlador propuesto), pese a que su funcionamiento se podía calificar como exitoso, tenía el inconveniente de ofrecer únicamente la capacidad de llegar a un punto en concreto sin tener en cuenta la orientación final. Además, dicho regulador no se podía considerar una versión definitiva o 100 % fiable, debido a la inclusión de un retardo como solución provisional para evitar un lazo algebraico con la variable binaria en el propio regulador. Por estos motivos, nace la necesidad de profundizar en este diseño y ofrecer un controlador que sea capaz de permitir al robot MASAR alcanzar una posición final con un orientación prefijada, basando este comportamiento en un control robusto, fiable y válido para diferentes situaciones.

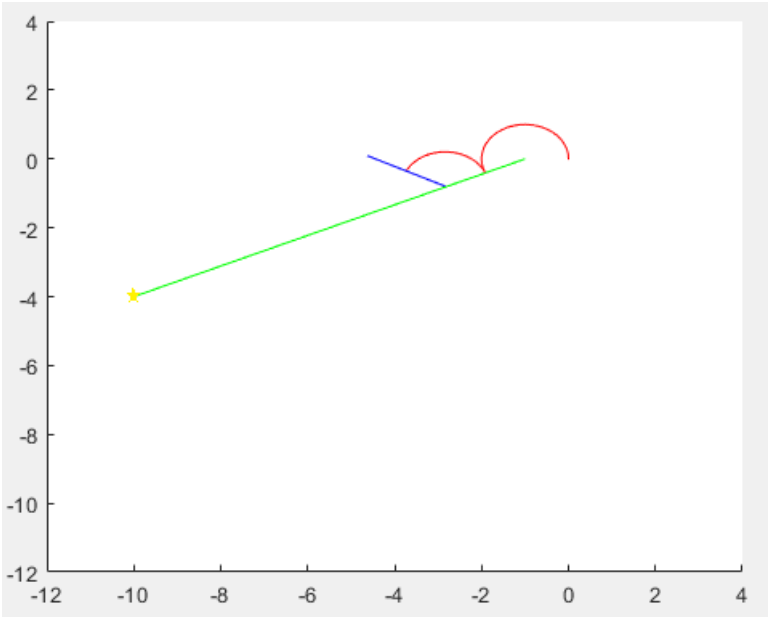

FIGURA 3.12: Alineación inicial y desplazamientos de 180º

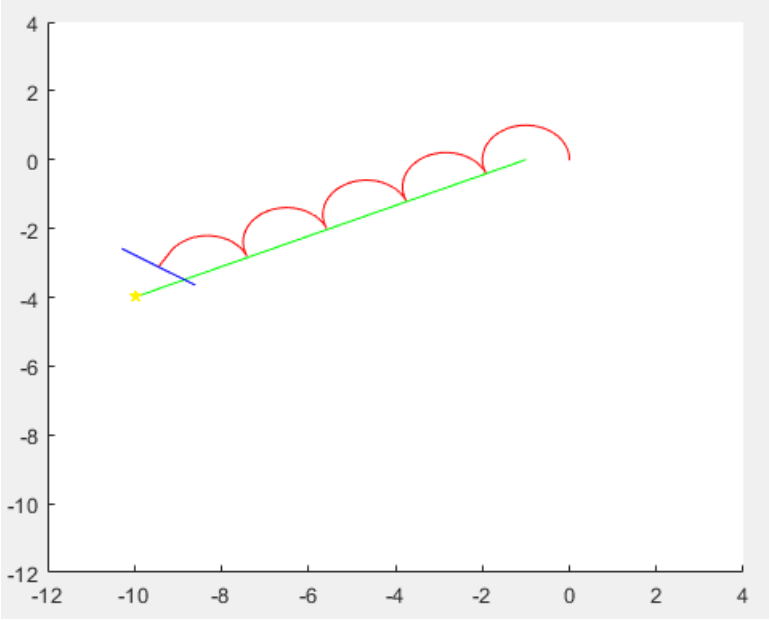

FIGURA 3.13: Cambio de control y llegada a objetivo

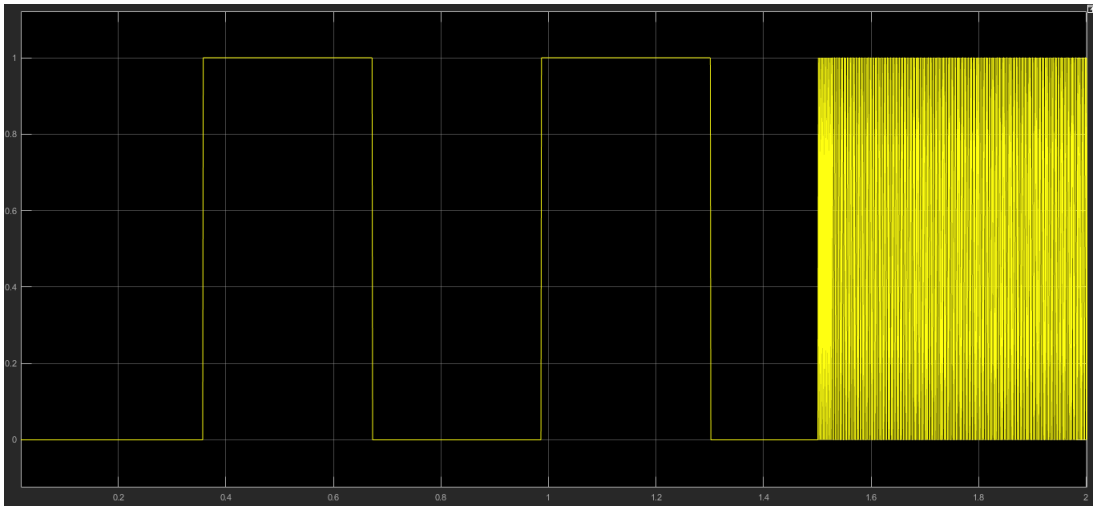

FIGURA 3.14: Variación de la variable binaria empleando el primer controlador

El controlador desarrollado en este trabajo se podría decir que actúa de forma similar al anterior, pues pretende generar movimientos basados en giros de 180º, para así lograr un desplazamiento más rápido y eficiente. Una vez cerca del objetivo, en lugar cambiar a un modo de control que pegue siempre el pivote más cercano, cambiará a un modo de control basado en ejecutar un número determinado de giros y así alcanzar, tanto el punto objetivo, como la orientación deseada. Por tanto, se deben tener en cuenta diferentes acciones a ejecutar:

- Identificar cuándo el robot ha ejecutado un giro de 180º, para cambiar el pivote fijado
- Variar la velocidad de giro en función de la distancia hasta el punto objetivo
- Calcular el instante en el que el controlador cambiará el modo de control para lograr la posición y orientación definitiva

Se explicará de forma detallada el proceso seguido, así como los principios de funcionamiento del controlador. El proceso se divide en los siguiente bloques

1. Entradas del sistema
2. Modo de movimiento de 180º
3. Modo de posición y orientación final

3.3.1. Entradas del sistema

Al inicio de la simulación, el esquema de Simulink diseñado requiere recibir 3 entradas por parte del usuario, estas son:

- Punto objetivo
- Orientación deseada en (º)
- Posición inicial de los pivotes A y B

La orientación introducida por el usuario debe estar dentro del intervalo [0,360º] y en coordenadas globales, es decir, el ángulo que forma el robot con la horizontal, considerando el pivote A como origen del vector y el B como final del vector (Figura 3.15).

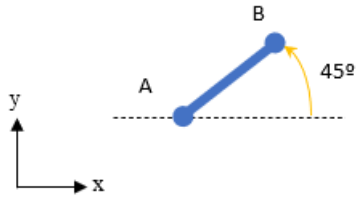

FIGURA 3.15: Ejemplo de orientación objetivo

Todas estas entradas entrar directamente como entrada al controlador. La orientación será un valor referencia, mientras que el valor de posición de los pivotes A y B variaría con el avance del robot.

Sin embargo, existe una condición inicial para el sistema que no es suministrada por el usuario, esta es el valor inicial de la variable binaria que identifica qué pivote se fija de inicio. Para ello, se deberá calcular qué pivote se encuentra más cerca del punto objetivo, dando a la variable binaria el valor indicado para fijar dicho pivote y generar por tanto avance en el movimiento inicial. Se hallará por tanto dentro un bloque función, la distancia entre cada pivote (A y B) y el punto objetivo (G), estas serán *d_AG* y *d_BG*. La salida de dicha función será el valor de la variable *bin_ini*, que indicará al regulador qué pivote debe fijar o está fijado de inicio (Figura 3.16). El código de la función se muestra en la siguiente tabla.

```
function bin_ini = fcn(A,B,G)

d_AG = norm(G-A);
d_BG = norm(G-B);
if d_AG < d_BG
    bin_ini = 0;
else
    bin_ini = 1;
end

end
```

Dicha variable de salida *bin_ini* será la condición inicial de un bloque retardo z^{-1}. Dicho bloque será el encargado de indicar al regulador cual era

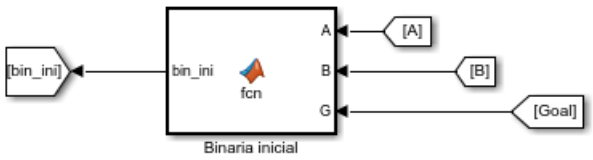

FIGURA 3.16: Bloques para el cálculo inicial de *bin*

el valor de la variable binaria en el instante al anterior al actual, pues como veremos más tarde, el valor estimado para la variable *bin* en cada instante dependerá directamente de cual era su valor en el instante anterior (Figura 3.17).

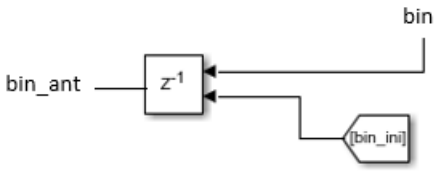

FIGURA 3.17: Bloque de retardo para la variable binaria

3.3.2. Modo de movimiento de 180º

Este modo de control, aplicado para la consecución de una locomoción basada en giros de 180º por parte del robot MASAR, es diferente al control del movimiento aplicado en el anterior controlador. Esto se debe a que el diseño implementado anteriormente requería de una solución provisional basada en el empleo de un retardo para evitar un lazo algebraico. Por ello, se ha propuesto un control más robusto para conseguir estos movimientos.

Actualmente, se ha implementado este control en base al cálculo del ángulo que forma el robot MASAR con la recta que une su pivote más adelantado (fijado) con el punto objetivo. Cuando dicho ángulo sea muy cercano a 0, se considera que el robot MASAR ha ejecutado un giro de 180º, por tanto deberá cambiar el valor de la variable binaria. Para ello, se ha definido un bloque de función propia con las siguientes entradas:

- Pivote A
- Pivote B
- Punto objetivo
- Variable binaria

Es necesario remarcar que, cuando el robot complete un giro de 180º e intercambie sus pivotes fijados, este cálculo de ángulo girado se reinicia, comenzando de nuevo en 180º y decreciendo conforme se desplaza el robot

hasta haber completado otro giro (valor del ángulo cercano a 0). Se puede considerar un proceso cíclico hasta que el robot completa todos los giros de 180º.

Cálculo del ángulo recorrido

De inicio, se comprueba el valor de la variable binaria para así identificar qué pivote se encuentra más cercano al punto objetivo. En base a esto, supongamos que tenemos una recta imaginaria que une este pivote con el punto objetivo (Figura 3.18). Una vez identificados los pivotes delantero y trasero, se calcula el ángulo que forma con la horizontal el vector, con inicio en el pivote fijado (*piv*) y final en el punto objetivo (*final*), con la función *atan2* (Ecuación 3.9 y Figura 3.18).

$$\alpha = atan2(final(2) - piv(2), final(1) - piv(1)) \tag{3.9}$$

Luego, se calcula el ángulo que forma con la horizontal el vector, con inicio en el pivote fijado (*piv*) y final en el pivote libre (*libre*), con la función *atan2* (Ecuación 3.11 y Figura 3.18). Vemos que dicho vector se corresponde con el robot MASAR.

$$\beta = atan2(libre(2) - piv(2), libre(1) - piv(1)) \tag{3.10}$$

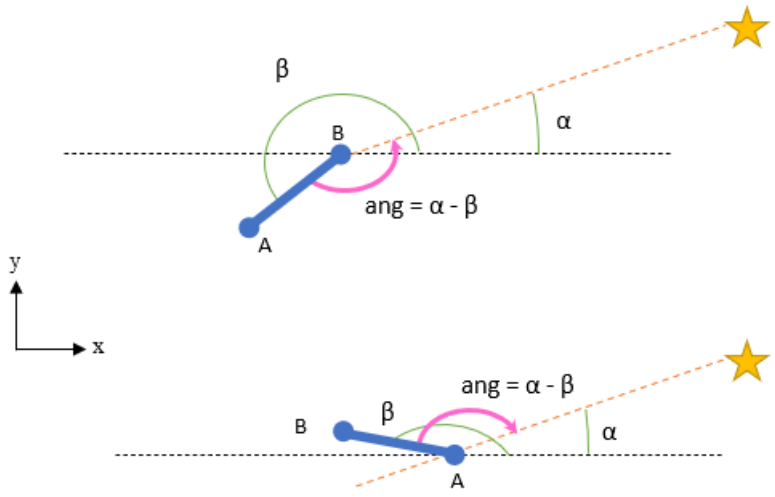

FIGURA 3.18: Cálculo del ángulo entre el robot MASAR y recta pivote-destino

Finalmente, el ángulo que requiere recorrer el robot en cada instante para haber recorrido un ángulo total de 180º será:

$$ang = \alpha - \beta \tag{3.11}$$

Al haber realizado los cálculos con la función *atan2*, la cual devuelve un ángulo entre [*-180º,180º*], el ángulo hallado es independiente del sentido de giro del robot (horario o antihorario), el cual será determinado en el bloque del controlador. Se muestra el fragmento de código que alberga este bloque función.

```
function ang = fcn(A,B,final,bin)

if bin==0
    piv=A;
    libre=B;
else
    piv=B;
    libre=A;
end

alpha=atan2(final(2)-piv(2),final(1)-piv(1));
beta=atan2(libre(2)-piv(2),libre(1)-piv(1));
ang=alpha-beta;

end
```

La salida de esta función (Figura 3.19), denominada *ang* será entrada del controlador.

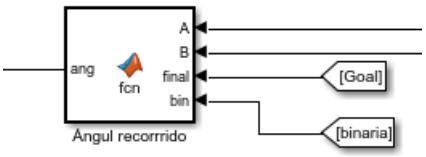

FIGURA 3.19: Bloque de cálculo del ángulo recorrido

Pasamos ahora a analizar como controla el bloque regulador este movimiento basado en giros de 180º haciendo uso del ángulo calculado *ang*.

Control de los movimientos de 180º

Como se ha comentado anteriormente, el controlador será capaz de otorgar dos comportamientos de movimiento al robot MASAR: movimiento de 180º y movimiento de posicionamiento y orientación final. La condición para que se ejecute el movimiento de 180º es que el robot se encuentre alejado del objetivo.

Es necesario remarcar que todo el código mostrado en este apartado se encuentra dentro del bloque que alberga el controlador.

Se ha estimado que esta distancia debe ser mayor a la longitud del robot, y una vez rebasada esta distancia, el controlador cambiará el modo de control. Estas distancias se calculan desde pivote actual a pivote objetivo (dA y dB) en la Figura 3.20), es decir, conociendo la posición y orientación deseada, es posible calcular la situación de los pivotes A y B de dicha pose. Estos puntos objetivos se calculan de la siguiente forma (Ecuaciones 3.12, 3.13, 3.14 y 3.15), dónde L es la semilongitud del robot MASAR:

$$PxA = final(1) - L * \cos(orientacion) \tag{3.12}$$

$$PyA = final(2) - L * \sin(orientacion) \tag{3.13}$$

$$PxB = final(1) + L * \cos(orientacion) \tag{3.14}$$

$$PyB = final(2) + L * \sin(orientacion) \tag{3.15}$$

$$Ao = [PxA; PyA] \tag{3.16}$$

$$Bo = [PxB; PyB] \tag{3.17}$$

Por tanto, las distancias entre pivotes serán (Ecuaciones 3.18 y 3.19):

$$dA = norm(A - Ao) \tag{3.18}$$

$$dB = norm(B - Bo) \tag{3.19}$$

```
%Robot objetivo
PxB=final(1)+L*cos(orientacion);
PyB=final(2)+L*sin(orientacion);
PxA=final(1)-L*cos(orientacion);
PyA=final(2)-L*sin(orientacion);

Bo=[PxB;PyB];
Ao=[PxA;PyA];

%Distancias
dA=norm(A -Ao);
dB=norm(B -Bo);
```

Estas distancias deberán ser mayores a la longitud total del robot MASAR ($2L$, Figura 3.20), de este modo más adelante será capaz de situarse en el punto objetivo con la orientación deseada. Cuando uno de los pivotes entre

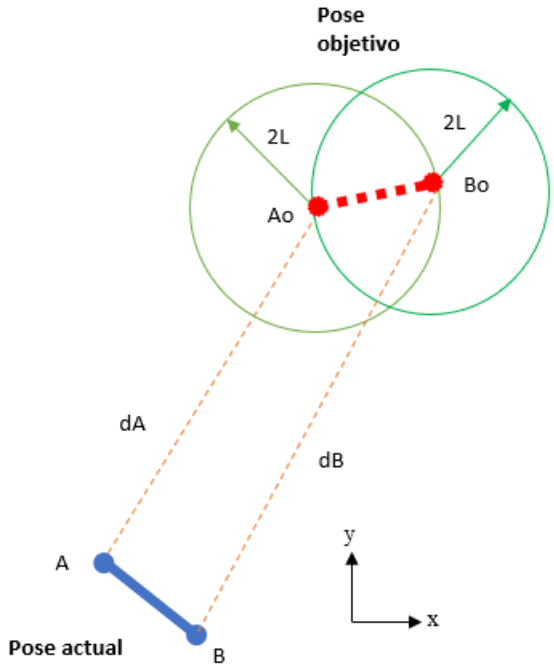

FIGURA 3.20: Distancias entre pivotes y circunferencias de destino

dentro de su circunferencia de radio *2L* correspondiente, finalizarán los movimientos de 180º. Por tanto, la condición de entrada al movimiento de 180º será (Ecuación 3.20):

$$if \quad dA > 2L \ \ and \ \ dB > 2L \tag{3.20}$$

De no cumplirse la condición, el modo de control pasaría al de posición y orientación final.

Una vez dentro de la condición, el siguiente paso es evaluar la variable *ang* calculada en el apartado anterior, para así identificar si se ha completado un giro de 180º. Se considera que el robot MASAR ha ejecutado un giro de 180º cuando el valor de *ang* se encuentre dentro del intervalo [-2º,2º], de esto modo es independiente si el sentido de giro es horario o antihorario. Se abren dos vías:

- El valor del ángulo recorrido no entra dentro de intervalo (Figura 3.21 a)): el robot debe continuar ejecutando el giro y se mantiene el valor de la variable binaria (pivote fijado) *bin = bin_ant*.
- El valor del ángulo recorrido sí entra dentro del intervalo (Figura 3.21 b)): evaluamos el valor de *bin_ant* y lo cambiamos por el binario contrario (cambiamos el pivote fijado).

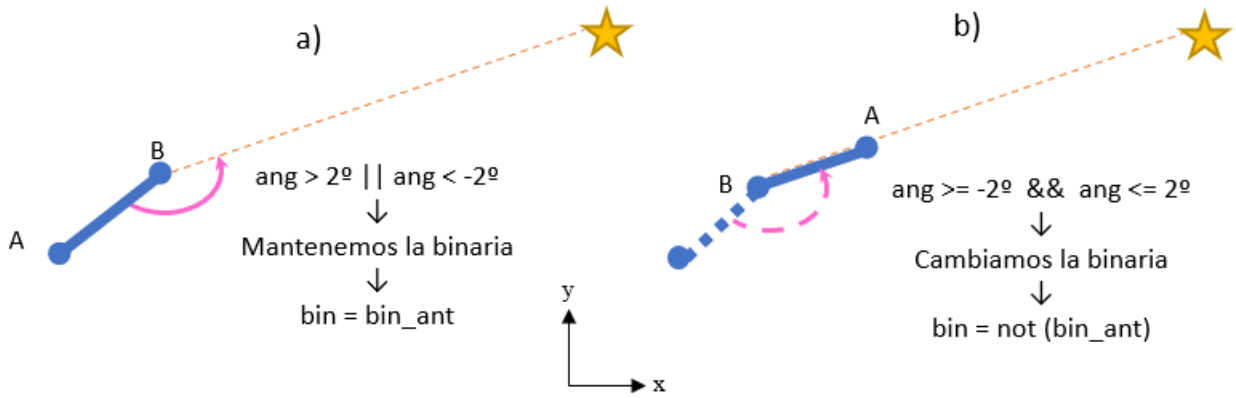

FIGURA 3.21: Comprobación del valor del ángulo recorrido

Con ello, la variable binaria *bin* quedará definida para cada instante de tiempo que se ejecute el modo de movimiento en 180º.

```
%Control movimiento 180
if dA > (2*L) && dB > (2*L)

    %Pasamos la variable ans a de radianes a grados
    ang_deg = ang*180/pi;

    if ang_deg ≥ −2 && ang_deg ≤ 2
        if bin_ant == 0
            bin = 1;
        else
            bin = 0;
        end
    else
        bin = bin_ant;
    end

    %El codigo continua en el siguiente fragmento
```

De forma paralela al calculo de la variable binaria, se calcula la velocidad de rotación del robot MASAR (*q_d*). Esta variable dependerá únicamente de la distancia a la que se encuentren los pivotes de sus puntos de destino. Cuando estos se acerquen a su circunferencia correspondiente de radio *2L* (se ha establecido un intervalo de *1.2*2L*) que marca el inicio del cambio de modo de control, más lento será el giro. Cuando el robot se encuentre lo suficientemente alejado, la velocidad de giro será constante y unitaria. Mencionar también que por convenio se ha decidido que, cuando el pivote fijado sea A (*bin=0*) la velocidad será positiva y antihoraria, y cuando el pivote fijado sea B (*bin=1*) la velocidad será negativa y horaria), tal y como se muestra en el siguiente fragmento de código:

```
    %Regulador de velocidad
    if dA > (2*L)*1.2
```

```
3           qd = (1-2*bin);
4       else
5           qd = (1-2*bin)*abs(dA);
6       end
7
8       if dB > (2*L)*1.2
9           qd = (1-2*bin);
10      else
11          qd = (1-2*bin)*abs(dB);
12      end
13
14  else %else de la primera condicion "if dA > (2*L) && dB > (2*L)"
```

El hecho de reducir la velocidad a medida que el robot se acerque al cambio de modo de control, persigue asegurar que uno de los pivotes (el más cercano depende de la situación de partida) se sitúe justo sobre su circunferencia de destino correspondiente (Figura 3.22) , pues cuanto más preciso sea este movimiento, con mayor precisión se ejecutará el segundo modo de control.

Con esto, quedará definida la velocidad q_d como salida del controlador en cada instante.

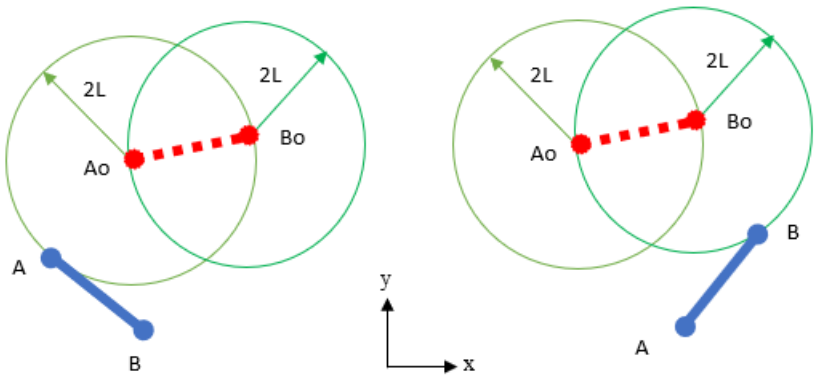

FIGURA 3.22: Llegada de los pivotes a su circunferencia de destino correspondiente

Como se ha mencionado anteriormente, el modo de control cambiará cuando uno de los pivotes se sitúe sobre el perímetro de su circunferencia objetivo correspondiente (Figura 3.22). Para enlazar con el siguiente modo de control, es necesario identificar y almacenar qué pivote ha llegado antes a su circunferencia. Se realizará mediante el empleo de un biestable R-S.

Empleo del biestable R-S

Un biestable R-S (Figura 3.23) es un dispositivo con dos entradas R y S (Reset y Set) y una variable de estado o salida Q capaz de almacenar un bit de información. Su funcionamiento es el siguiente:

- Si su entrada Set se activa su estado Q se pone en Alto
- Si su entrada Reset se activa su estado Q se pone en Bajo
- Si no se activa ni Set ni Reset su estado no cambia
- No se permite activar Set y Reset simultáneamente.

Pueden ser síncronos, que la salida se actualice cada pulsación del reloj, o asíncronos, dónde la salida se actualiza ante variación en la entrada.

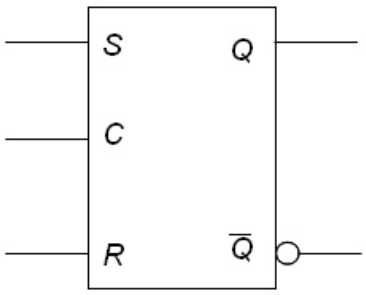

FIGURA 3.23: Ejemplo de biestable R-S

El empleo del biestable permitirá codificar qué pivote ha llegado a su circunferencia correspondiente al final del modo de control basado en movimientos de 180º, se ha empleado esta notación:

- Si A ha llegado antes, R = 1 y S = 0, por tanto Q = 0 (líneas 25 y 26 del siguiente código)
- Si B ha llegado antes, R = 0 y S = 1, por tanto Q = 1 (líneas 33 y 34 del siguiente código)

La asignación de valores a las entradas del biestable se realiza mediante salidas del controlador. Cada vez que se acceda al controlador, se pondrán las entradas R y S a 0 (líneas 17 y 18 del siguiente código), manteniendo así el estado anterior. De inicio, es indiferente el valor de la salida Q. Los valores de las entradas se verán alterados cuando el pivote A se encuentre cerca de su respectiva circunferencia, o el pivote B llegue a la suya. De este modo se mantendrá en la salida el valor de Q indicativo del pivote que ha llegado antes, siendo esto crucial para el siguiente apartado.

A continuación, se muestra una tabla de código con todos los comandos descritos en este apartado, así como dónde se encuentran las asignaciones a las entradas del biestable, detalladas en el apartado anterior.

```
1       %Control movimiento 180
2  if dA > (2*L) && dB > (2*L)
3
4       %Pasamos la variable ang de radianes a grados
```

```
5       ang_deg = ang*180/pi;
6
7       if ang_deg ≥ -2 && ang_deg ≤ 2
8           if bin_ant == 0
9               bin = 1;
10          else
11              bin = 0;
12          end
13      else
14          bin = bin_ant;
15      end
16
17      R=false;
18      S=false;
19
20      %Regulador de velocidad
21      if dA > (2*L)*1.2
22          qd = (1-2*bin);
23      else
24          qd = (1-2*bin)*abs(dA);
25          R=true;
26          S=false;
27      end
28
29      if dB > (2*L)*1.2
30          qd = (1-2*bin);
31      else
32          qd = (1-2*bin)*abs(dB);
33          R=false;
34          S=true;
35      end
36
37  else %else de la primera condicion "if dA > (2*L) && dB > (2*L)"
```

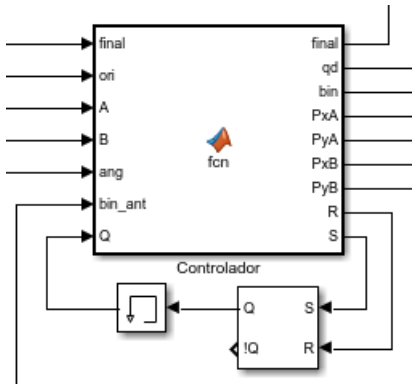

FIGURA 3.24: Implementación del bloque R-S en el controlador

3.3.3. Modo de posición y orientación final

Partiendo desde cualquiera de las 2 situaciones mostradas en la Figura 3.22, se he implementado un modo de control basado en la ejecución de únicamente 3 giros para alcanzar la posición y orientación final deseada.

Se explicará el proceso para aquella situación en la que el pivote A alcanza su circunferencia en primer lugar (salida del biestable $Q = 0$). Siendo el caso opuesto, cuando B llega antes a su circunferencia, homólogo.

Primer giro

Supuesta aquella situación en la que el pivote A llega a su circunferencia de destino antes que B, es decir, no se cumple la condición de movimiento de 180º (Ecuación 3.20) y el valor de $Q = 0$, el objetivo del **primer giro** será situar el pivote B sobre la circunferencia cuyo centro es Ao, se presentan dos casos posibles.

Antes de abordarlos, es necesario realizar algunos cálculos de distancias a los cuales daremos uso a continuación. Estas distancias son:

- Distancia entre el pivote que no ha llegado a su circunferencia (B) y pivote objetivo opuesto (Ao), Ecuación 3.21 y Figura 3.25 a).
- Distancia más corta entre el pivote que no ha llegado a su circunferencia (B) y la circunferencia opuesta (circunferencia con centro en Ao). Se calcula restando el radio a la distancia anterior, Ecuación 3.22 y Figura 3.25 b).

$$dBA = norm(B - Ao) \tag{3.21}$$

$$dist = dBA - 2L \tag{3.22}$$

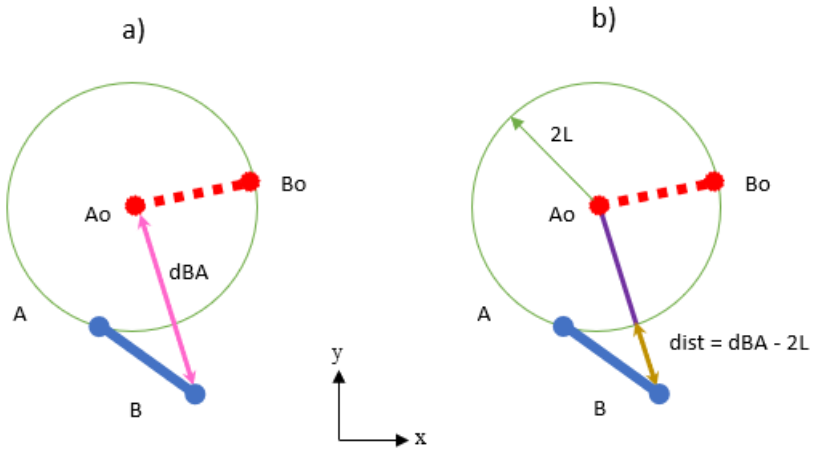

FIGURA 3.25: Distancia entre Ao y B

El cálculo de la distancia *dist* nos permite identificar los dos casos posibles que podemos encontrar una vez A llega a su circunferencia, estos son:

- B se encuentra fuera de la circunferencia de A
- B se encuentra dentro de la circunferencia de A

Siendo el objetivo de este giro situar B sobre la circunferencia de A, identificar la posición de B resulta de vital importancia para ordenar al robot una velocidad en sentido positivo o negativo. Esto se realiza poniendo el foco sobre el signo de la distancia *dist*:

- Si B se encuentra fuera de la circunferencia de A, $dBA > 2L$ y el signo de *dist* será positivo, aplicando una velocidad positiva y antihoraria
- Si B se encuentra dentro de la circunferencia de A, $dBA < 2L$ y el signo de *dist* será negativo, aplicando una velocidad negativa y horario

Se alcanza la conclusión de que, cuando el pivote A se encuentra sobre la circunferencia, el sentido de giro del pivote B (Velocidad *q_d*) será directamente el signo de *dist*. En el caso opuesto, cuando el pivote a rotar es A, el sentido de giro será opuesto al signo de *dist*.

Además, resulta de gran utilidad aportar también al cálculo de *q_d* la distancia entre B y la circunferencia de A, para así reducir la velocidad conforme se acerque a ella y conseguir una precisión mayor. Tras esto, el valor de la velocidad *q_d* cuando se desplaza el pivote B resulta ser igual a *dist* (Ecuación 3.23).

$$q\_d = dist = dBA - 2L \tag{3.23}$$

Para el caso análogo en el que se desplaza el pivote A, se debe cambiar el signo de la expresión (Ecuación 3.26).

$$q\_d = -dist = -(dBA - 2L) \tag{3.24}$$

Este giro finalizará cuando el valor de *dist* sea inferior a 0,01, dando por hecho que B ha llegado a su objetivo (Figura 3.26).

A continuación, se añade el código responsable de la ejecución del giro descrito:

```
else %else de la primera condicion "if dA > (2*L) && dB > (2*L)"

    %Biestable mantiene el estado para saber que pivote ha ...
       llegado antes
    R=false;
    S=false;
    %Reiniciamos las variables de salida
    qd=0;
    bin=0;

    if Q == false %Ha llegado A antes
```

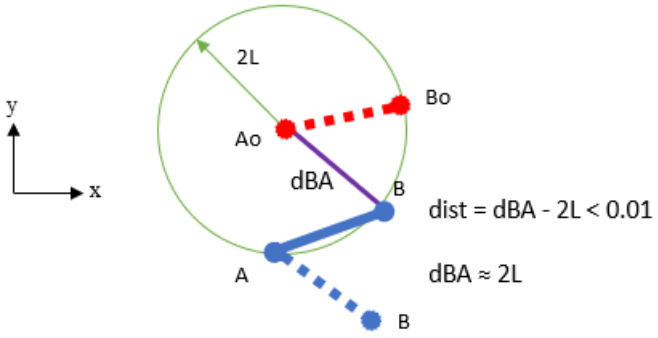

FIGURA 3.26: Pivote B alcanza la circunferencia de Ao

```
11
12          %Calculo de dist
13          dist=abs(sqrt(((B(1)-PxA)^2)+((B(2)-PyA)^2))-2*L);
14
15          %Se fija A y se rota antihorario (giro 1), hasta ...
               cumplir el siguiente if
16          bin=0;
17          qd=dist*sign(dBA - 2*L);
18
19          %Compruebo si B ya esta sobre la circunferencia de A
20          %En cumplirse el siguiente if, cambiamos de giro
21          if dist ≤ 0.01
```

Segundo giro

El siguiente giro parte desde la situación mostrada en la Figura 3.26, donde los pivotes A y B se encuentran sobre la circunferencia de Ao (o sobre la de Bo en el caso homólogo. El objetivo de este giro es situar el pivote que anteriormente estaba fijado, es decir, el que había llegado antes sobre su circunferencia (en este caso A), sobre su punto objetivo correspondiente. Queremos situar A sobre Ao, tal y como se muestra en la Figura 3.27.

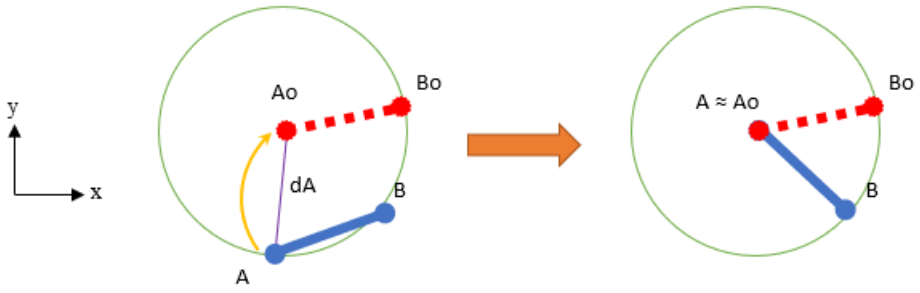

FIGURA 3.27: Segundo giro, aplicado sobre A

Para ello, cuando se cumpla la condición de la Ecuación 3.25, se cambiará el pivote fijado, asignando *bin = 1* y fijando B.

$$if \quad dist \quad <= \quad 0{,}01 \tag{3.25}$$

En cuanto a la velocidad de rotación del pivote A, esta debe ser negativa (sentido horario). El sentido vendrá dado por la variable binaria según la expresión de la Ecuación 3.26, al ser la variable *bin=1*, el resultado de la operación resulta ser un signo negativo aplicado sobre la distancia entre A y Ao (dA), para regular la velocidad en función de la cercanía al destino.

$$qd = (1 - 2 * bin) * abs(dA); \tag{3.26}$$

El segundo giro concluye cuando la distancia dA (Figura 3.27) sea inferior a 0.02 (Ecuación 3.25). En el caso homólogo, se comprobaría dB tras rotar B hasta Bo.

$$if \quad dA \quad <= \quad 0{,}02 \tag{3.27}$$

El código desde el inicio del modo de posición y orientación final hasta el segundo giro, se muestra a continuación:

```
else %else de la primera condicion "if dA > (2*L) && dB > (2*L)"

    %Biestable mantiene el estado
    R=false;
    S=false;
    qd=0;
    bin=0;

    if Q == false %Ha llegado A antes

        dist=abs(sqrt(((B(1)-PxA)^2)+((B(2)-PyA)^2))-2*L);

        %Se fija A y se rota antihorario (giro 1), hasta ...
           cumplir el siguiente if
        bin=0;
        qd=dist*sign(dBA - 2*L);

        %Compruebo si B ya esta sobre la circunferencia de A
        %En cumplirse el siguiente if, cambiamos de giro
        if dist <= 0.01
            bin=1;
            qd=(1-2*bin)*abs(dA);
        end

        %Final del giro 2, A llega a su objetivo
        if dA <= 0.02
```

Tercer giro

El último giro será el encargado de lograr situar finalmente al robot MASAR en la pose deseada. Partiendo de la situación final mostrada en la Figura

3.27.

El proceso es muy similar a los anteriores giros, en este caso se vuelve a fijar el pivote A cambiando el valor de la variable binaria a *bin=0* y se rota B hasta hacerlo llegar hasta Bo (Figura 3.28).

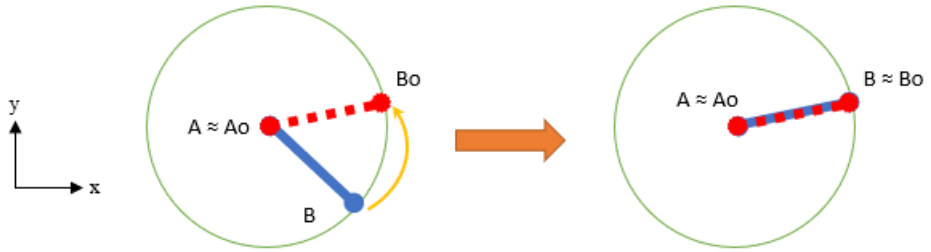

FIGURA 3.28: Giro final

Sin embargo, se ha introducido una pequeña modificación para asegurar que el pivote libre alcanza su pivote destino rotando por el lado corto (En la Figura 3.29, se muestran 2 ejemplos de cuales son los lados cortos y cuales los largos).

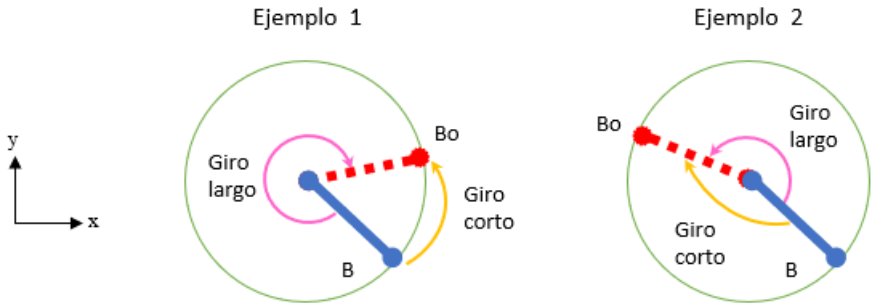

FIGURA 3.29: Giro corto vs Giro largo

Dicha modificación pretende identificar el sentido del giro del pivote libre, partiendo de la magnitud angular que todavía debe recorrer el robot MASAR para alcanzar la orientación final. Se deben calcular dos ángulos, el ángulo que forma la posición inicial del MASAR con la horizontal (Ecuación 3.28), tomando como inicio del vector el pivote A y como extremo el pivote B, (Ángulo *source*, Figura 3.30). Y el ángulo que forma el robot objetivo con la horizontal, que se corresponde con la orientación objetivo en radianes (*target*).Sin embargo, se realizará el cálculo con *atan2* para obtener el ángulo en un intervalo entre [-180º,180º] (Ecuación 3.29).

$$source = atan2(B(2) - A(2), B(1) - A(1)) \tag{3.28}$$

$$target = atan2(Bo(2) - Ao(2), Bo(1) - Ao(1)) \tag{3.29}$$

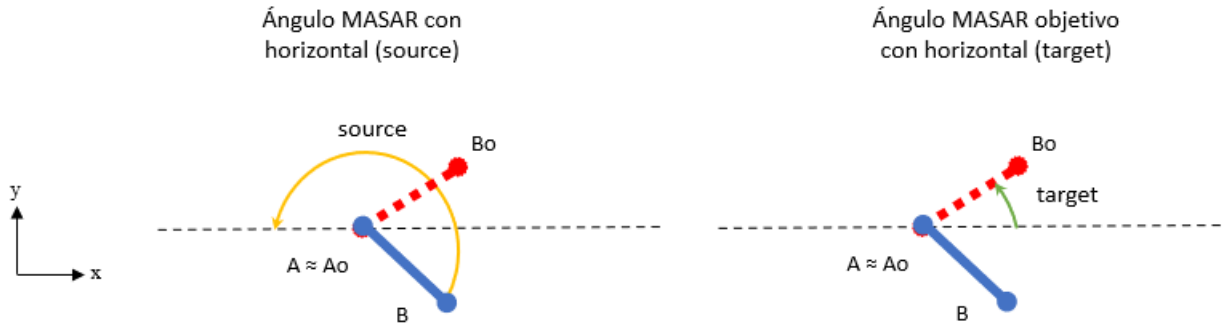

FIGURA 3.30: Ángulos necesarios para el tercer giro

Identificados estos ángulos, la resta entre el ángulo objetivo (*target*) y el ángulo actual (*source*) será el ángulo restante a recorrer, del cual se analizará su signo.

$$rest = target - source \tag{3.30}$$

Continuando con el proceso, para identificar el sentido de giro del robot, se ha seguido el siguiente análisis. Se ha considerado como giro corto aquel menor a 180º o mayor a -180º, es decir, el recorrido debe ser menor o igual media circunferencia. Se siguen los siguientes pasos:

1. Comprobación de si el ángulo restante es mayor o menor que *pi* o menor que *-pi*. De ser así, estaríamos en el camino largo y debemos cambiarlo.

2. Si el ángulo restante es mayor que *pi*, estamos en el camino largo en sentido antihorario, lo opuesto sería el camino corto y horario. Por ello, al ángulo restante mayor que *pi* se le restará *2pi*, cambiando el sentido de giro.

3. Si por el contrario, el ángulo restante es menor que *pi*, estamos en el camino largo en sentido horario, lo opuesto sería el camino a corto y antihorario. Por ello, al ángulo restante menor que *pi* se le sumará *2pi*,cambiando el sentido de giro.

4. Si no cumple ninguna de estas condiciones, el signo de *rest* ya indica el camino corto.

La velocidad de rotación será: el signo de *rest* para dar sentido de giro, y la distancia entre B y B objetivo (dB) para conseguir la precisión deseada (Ecuación 3.31).

$$qd = sign(rest) * dB \tag{3.31}$$

Se adjunta el código del proceso descrito:

```
%Final del giro 2, A llega al su objetivo
if dA≤0.02
    bin=0;
    %Calculo el angulo que me falta por rotar
    target=atan2(PyB-PyA,PxB-PxA);
    source=atan2(B(2)-A(2),B(1)-A(1));
    rest=target-source;

    %Nos aseguramos que vamos por el lado corto
    if rest>pi
        rest=rest-2*pi;
    end

    if rest<-pi
        rest=rest+2*pi;
    end

    qd=sign(rest)*dB;
```

Cuando el pivote libre protagonista de este último giro (en este caso el pivote B) se encuentre sobre su pivote destino (Bo), respetando un margen de dB = 0.02), el robot MASAR habrá alcanzado la posición y orientación deseada.

```
%Final
    if dB≤0.02
        final=1;
        bin=0;
        qd=0;
    end
```

Antes de finalizar con el apartado, es necesario volver a destacar que todo el proceso descrito se enfoca en aquel caso en el cual, durante el movimiento de 180º, el pivote A llega antes a su circunferencia de destino, dando un valor de 0 a la salida *Q* del biestable. El proceso opuesto, cuando B llega antes a su circunferencia *Q=1*, es el homólogo cambiando un pivote por otro en las acciones realizadas.

En el código final, se muestran todos los procesos que alberga el controlador descrito, con ambos modos de control:

```
function [final,qd,bin,PxA,PyA,PxB,PyB,R,S] = ...
    fcn(final,ori,A,B,ang,bin_ant,Q)

L=1;
orientacion=ori*pi/180;

%Robot objetivo
PxB=final(1)+L*cos(orientacion);
PyB=final(2)+L*sin(orientacion);
PxA=final(1)-L*cos(orientacion);
PyA=final(2)-L*sin(orientacion);

```

```
Bo=[PxB;PyB];
Ao=[PxA;PyA];

%Distancias
dA=norm(A -Ao);
dB=norm(B -Bo);


dAB=norm (A -Bo);
dBA=norm(B -Ao);

%Control movimiento 180
if dA > (2*L) && dB > (2*L)

    %Pasamos la variable ans de radianes a grados
    ang_deg = ang*180/pi;

    if ang_deg ≥ -2 && ang_deg ≤ 2
        if bin_ant == 0
            bin = 1;
        else
            bin = 0;
        end
    else
        bin = bin_ant;
    end

    R=false;
    S=false;

    %Regulador de velocidad
    if dA > (2*L)*1.2
        qd = (1-2*bin);
    else
        qd = (1-2*bin)*abs(dA);
        R=true;
        S=false;
    end

    if dB > (2*L)*1.2
        qd = (1-2*bin);
    else
        qd = (1-2*bin)*abs(dB);
        R=false;
        S=true;
    end

else %else de la primera condicion "if dA > (2*L) && dB > (2*L)"

    %Biestable mantiene el estado
    R=false;
    S=false;
    qd=0;
    bin=0;

    if Q == false %Ha llegado A antes

```

```
69          dist=abs(sqrt(((B(1)-PxA)^2)+((B(2)-PyA)^2))-2*L);
70
71          %Se fija A y se rota antihorario, hasta cumplir el ...
               siguiente if
72          bin=0;
73          qd=dist*sign(dBA - 2*L);
74
75          %Compruebo si B ya esta sobre la circunferencia de A
76          %En cumplirse el siguiente if, cambiamos de giro
77          if dist <= 0.01
78              bin=1;
79              qd=(1-2*bin)*abs(dA);
80          end
81
82          %Final del giro 2, A llega al su objetivo
83          if dA<=0.02
84              bin=0;
85              %Calculo el angulo que me falta por rotar
86              target=atan2(PyB-PyA,PxB-PxA);
87              source=atan2(B(2)-A(2),B(1)-A(1));
88              rest=target-source;
89
90              %Nos aseguramos que vamos por el lado corto
91              if rest>pi
92                  rest=rest-2*pi;
93              end
94
95              if rest<-pi
96                  rest=rest+2*pi;
97              end
98
99              qd=sign(rest)*dB;
100
101             %Final
102             if dB<=0.02
103                 final=1;
104                 bin=0;
105                 qd=0;
106             end
107         end
108
109     end
110
111     if Q == true  %Ha llegado B antes
112
113         dist=abs(sqrt(((A(1)-PxB)^2)+((A(2)-PyB)^2))-2*L);
114         bin=1;
115         qd=-dist*sign(dAB - 2*L);
116
117         if dist <= 0.01
118             bin=0;
119             qd=(1-2*bin)*abs(dB);
120         end
121
122         if dB<=0.02
123             bin=1;
124             target=atan2(PyB-PyA,PxB-PxA);
```

```
125             source=atan2(B(2)-A(2),B(1)-A(1));
126             rest=target-source;
127 
128             if rest>pi
129                 rest=rest-2*pi;
130             end
131 
132             if rest<-pi
133                 rest=rest+2*pi;
134             end
135 
136             qd=sign(rest)*dA;
137 
138             if dA<=0.02
139                 final=1;
140                 bin=0;
141                 qd=0;
142             end
143         end
144 
145     end
146 end
```

3.3.4. Ejemplos de simulación y error

Para concluir el capítulo, se mostrarán algunos ejemplos de simulación aplicando el controlador descrito, indicando también el error en posición y orientación final, es decir, la diferencia entre la pose deseada y la lograda.

Los errores se han calculado hallando la diferencia entre los pivotes logrados y deseados, y la diferencia de orientación entre [0, 360º].

```
1 error_A = [A(1)-PxA;A(2)-PyA]
2 error_B = [B(1)-PxB;B(2)-PyB]
3 orientacion = atan2(B(2)-A(2),B(1)-A(1))*180/pi;
4 if orientacion < 0
5     orientacion=orientacion+360;
6 else
7     orientacion;
8 end
9 error_orientacion = orientacion_deseada - orientacion
```

Los elementos graficados serán lo siguientes:

- Recta negra: robot MASAR
- Curvas azules: desplazamiento del punto medio del robot
- Circunferencia azul: circunferencia objetivo del pivote A
- Circunferencia negra: circunferencia objetivo del pivote B
- Recta roja: pose objetivo para el robot MASAR
- Circunferencia roja: señala el pivote A objetivo, el opuesto será B

Ejemplo nº 1

Para el primer ejemplo, se propone que el punto medio del robot MASAR se sitúe en [0, 0], y el objetivo en [0, 20] con una orientación de 45º. Las Figuras 3.31 y 3.32 se muestra el desplazamiento del robot MASAR empleando el modo de control de 180º. En la Figura 3.33, el pivote A es el primero en alcanzar su circunferencia, cambiando el modo de control. Luego, se sitúa el pivote B sobre la circunferencia de A, movimiento correspondiente al primer giro del segundo modo de control (Figura 3.34). El segundo giro consiste en rotar A hasta alcanzar su pivote objetivo, como se muestra en la Figura 3.35. Para terminar, se ejecuta el tercer giro para alcanzar la pose deseada (Figura 3.36).

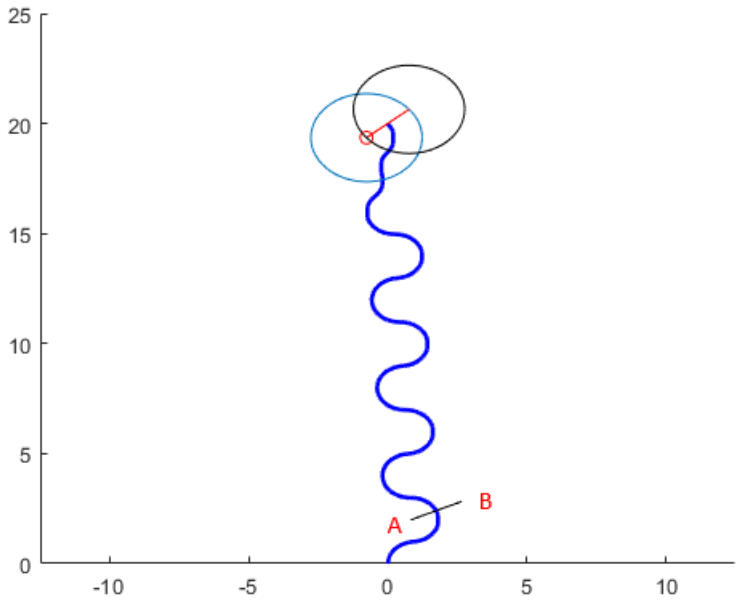

FIGURA 3.31: Ejemplo 1: Inicio del movimiento de 180º

Los errores obtenidos han sido los siguientes:

- Pivote A, desviación de la coordenada x = - 0.0040 uds de longitud
- Pivote A, desviación de la coordenada y = - 0.0196 uds de longitud
- Pivote B, desviación de la coordenada x = - 0.0040 uds de longitud
- Pivote B, desviación de la coordenada y = - 0.0196 uds de longitud
- Error de orientación = - 0.000247º

Ejemplo nº 2

Para el segundo ejemplo, se situado de inicio el punto medio del robot MASAR en [3, 5], y el objetivo en [-5, 18] con una orientación de 200º. Las

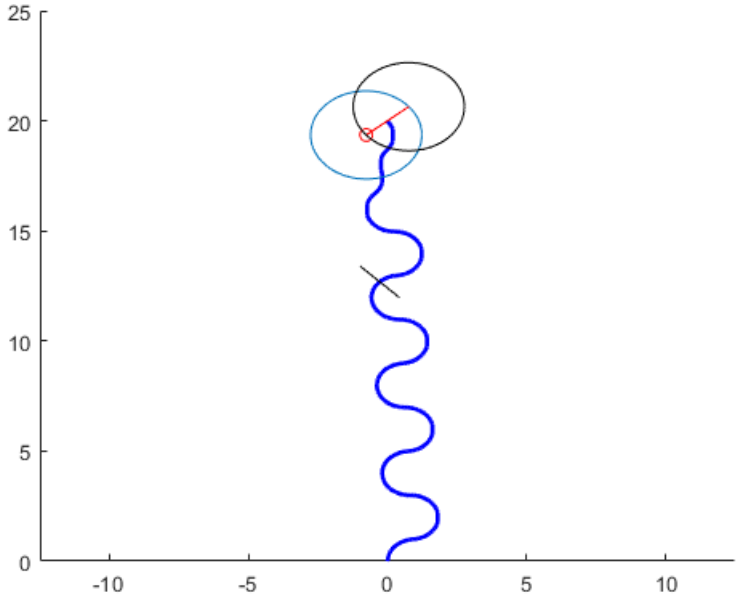

FIGURA 3.32: Ejemplo 1: Continuación del movimiento de 180º

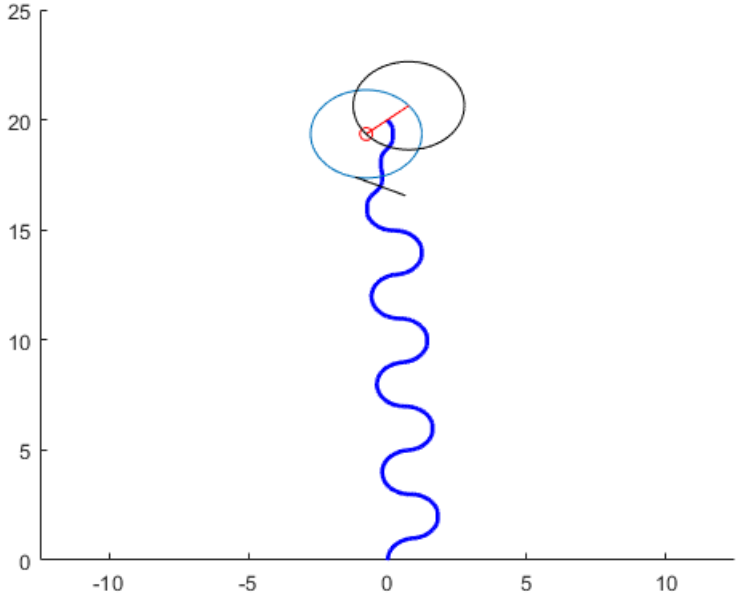

FIGURA 3.33: Ejemplo 1: El pivote A alcanza su circunferencia objetivo

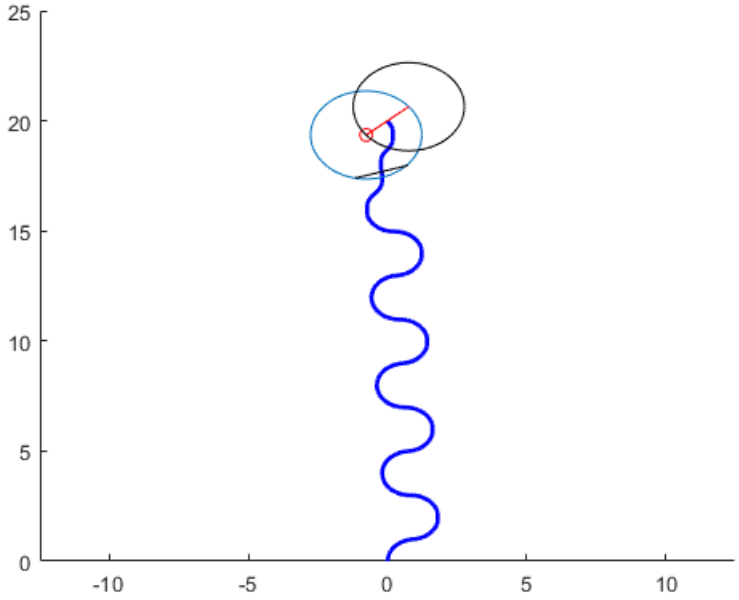

FIGURA 3.34: Ejemplo 1: Se sitúa el pivote B sobre la circunferencia de A, primer giro

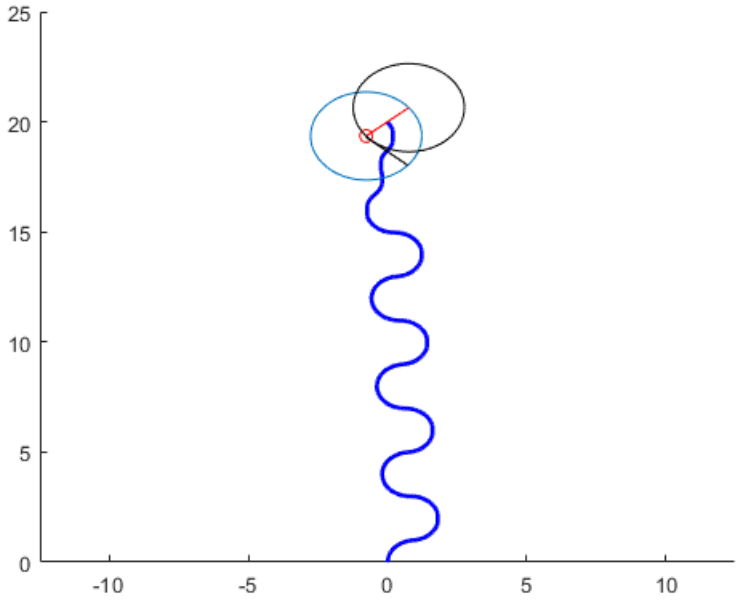

FIGURA 3.35: Ejemplo 1: El pivote A alcanza su pivote objetivo, segundo giro

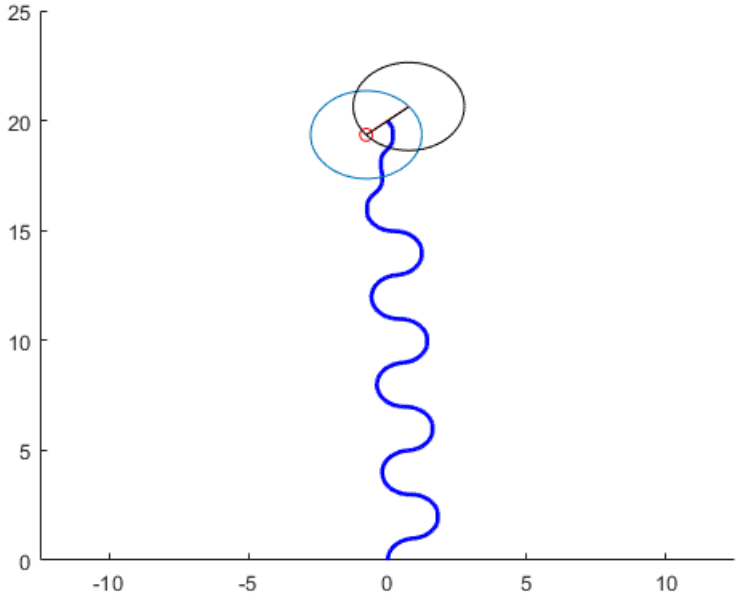

FIGURA 3.36: Ejemplo 1: El pivote B alcanza su pivote objetivo, tercer giro

Figuras 3.37 y 3.38 se muestra el desplazamiento del robot MASAR empleando el modo de control de 180º. En la Figura 3.39, el pivote A, de nuevo, es el primero en alcanzar su circunferencia, cambiando el modo de control. Se sitúa el pivote B sobre la circunferencia de A, primer giro del segundo modo de control (Figura 3.40). El segundo giro consiste en rotar A hasta alcanzar su pivote objetivo, como se muestra en la Figura 3.41. Para finalizar, se ejecuta el tercer giro para alcanzar la pose deseada (Figura 3.42).

Los errores obtenidos han sido los siguientes:

- Pivote A, desviación de la coordenada x = 0.0192 uds de longitud
- Pivote A, desviación de la coordenada y = -0.0057 uds de longitud
- Pivote B, desviación de la coordenada x = 0.0112 uds de longitud
- Pivote B, desviación de la coordenada y = - 0.0165 uds de longitud
- Error de orientación = 0.6755º

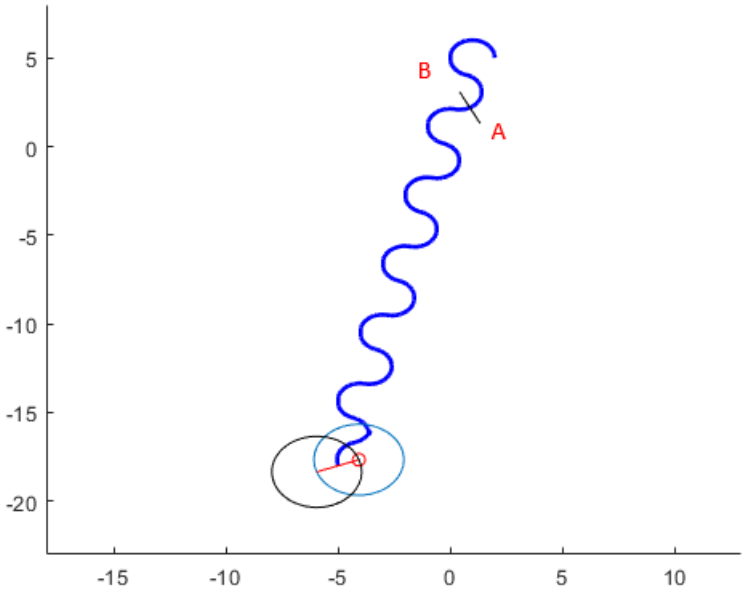

FIGURA 3.37: Ejemplo 2: Inicio del movimiento de 180º

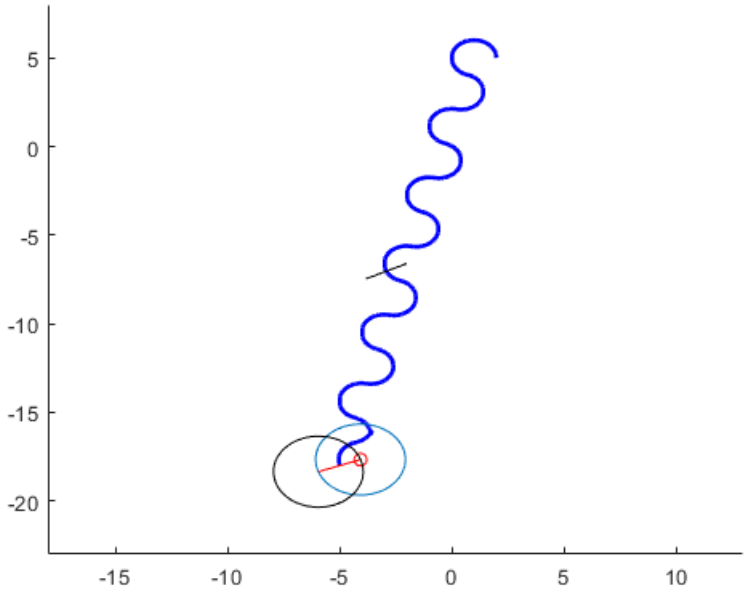

FIGURA 3.38: Ejemplo 2: Continuación del movimiento de 180º

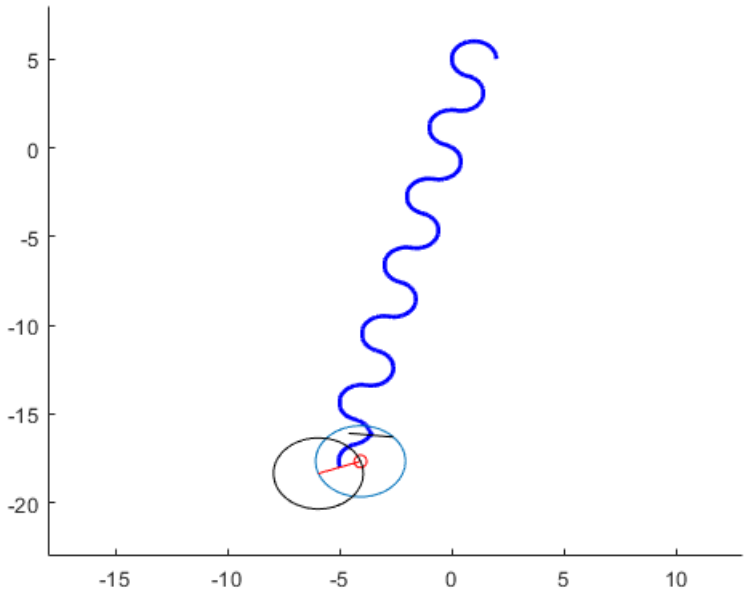

FIGURA 3.39: Ejemplo 2: El pivote A alcanza su circunferencia objetivo

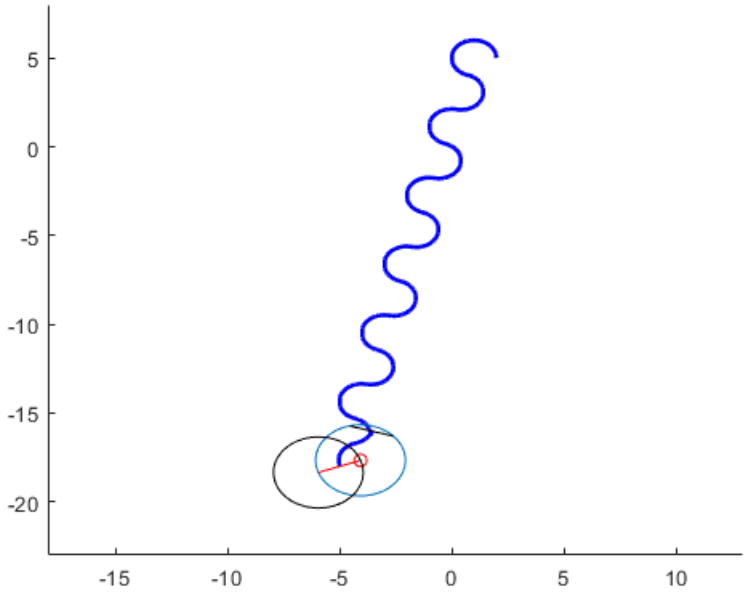

FIGURA 3.40: Ejemplo 2: Se sitúa el pivote B sobre la circunferencia de A, primer giro

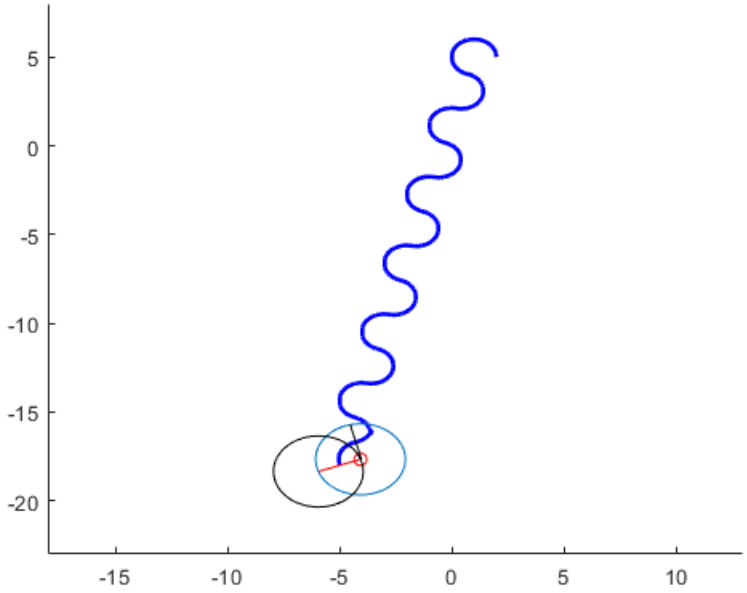

FIGURA 3.41: Ejemplo 2: El pivote A alcanza su pivote objetivo, segundo giro

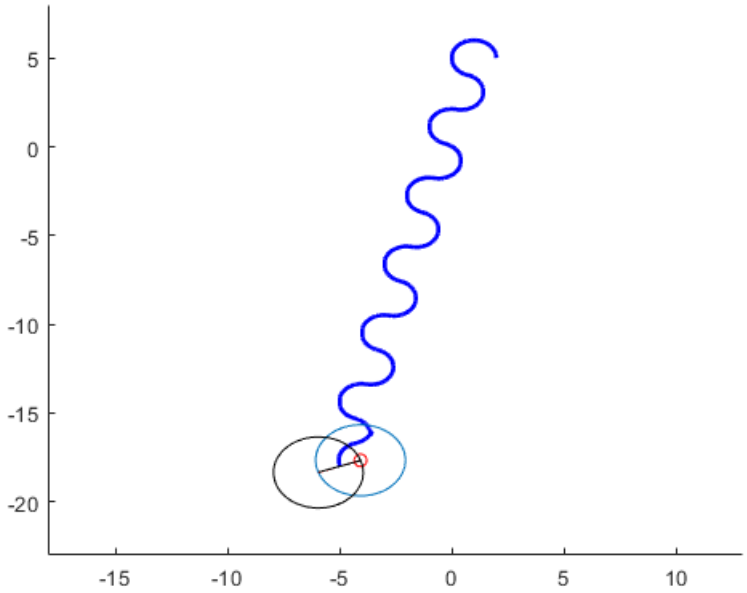

FIGURA 3.42: Ejemplo 2: El pivote B alcanza su pivote objetivo, tercer giro

Capítulo 4

Diseño e implementación de algoritmos de navegación

Se define como **Navegación** la acción que ejecuta un vehículo para desplazarse desde un punto inicio hasta un punto final, pasando por cada uno de los puntos que conforman esta ruta. El concepto de planificación se divide en diversas etapas:

1. Percepción y recopilación de información del entorno: empleo de sensores para generar un mapa del entorno
2. Planificación de la ruta: conformar una secuencia ordenada de objetivos que enlacen la posición inicial con la final, evitando obstáculos
3. Generación del camino: de todas las rutas posibles, se define una función continua para posibilitar el recorrido del robot a lo largo de la ruta seleccionada
4. Seguimiento del camino: acciones de control sobre el robot, teniendo en cuenta las características motrices del mismo, la características de sus actuadores, el camino seleccionado y las restricciones y problemas que este puede imponer. Dicho punto ha sidoa bordado en el capítulo anterior.

El presente trabajo se centra en las etapas 2 y 3 del proceso de navegación, pues el robot trabaja sobre un entorno conocido (la etapa 1 no es necesaria), y acerca de la etapa 4, se realiza una simulación aproximada para calcular el tiempo que tarda el robot en recorrer el camino.

Como entorno conocido, se propone un algoritmo que genera mapas aleatorios formados por obstáculos con forma rectangular, circular y combinación de ambos.

Generado el mapa, se implementan dos algoritmos de planificación de trayectorias para generar caminos entre posición inicial y final, estos son:

1. **Diagrama de Voronoi**, maximizando la distancia del robot a los obstáculos del entorno

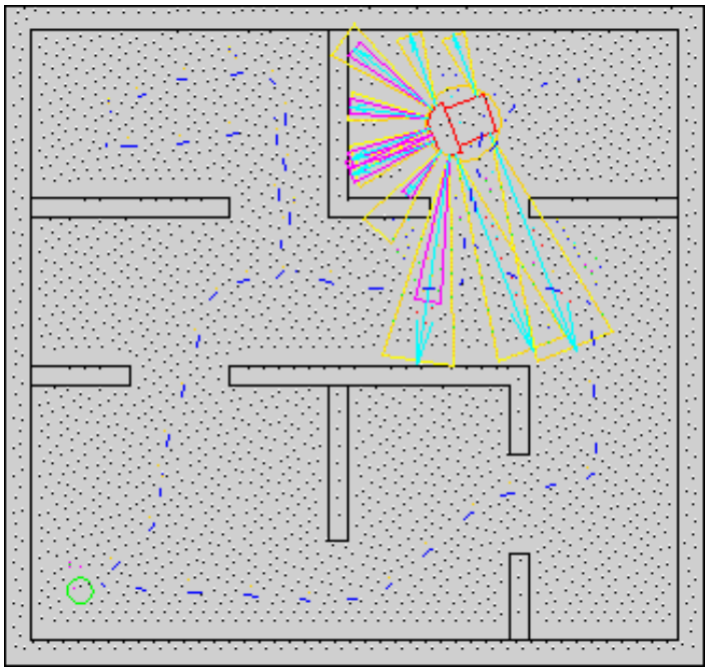

FIGURA 4.1: Proceso de navegación de un robot móvil

2. **Grafo de Visibilidad**, el cual trabaja con los vértices de los objetos, y por tanto, el robot pasará más cerca de ellos

Una vez generados todos los caminos posibles, se emplea el planificador A* para hallar los caminos más óptimos y eficientes, los cuales serán evaluados mediante una simulación aproximada del robot MASAR recorriendo dicho camino y obteniendo el tiempo transcurrido.

4.1. Creación del mapa

Este proceso comienza con la creación del mapa sobre el que se desplazará el robot MASAR. Como se ha mencionado anteriormente, para posibilitar el estudio ante diversos casos, estos mapas se generan de forma aleatoria a través de la situación de los obstáculos que lo conforman.

El proceso de generación del mapa será prácticamente idéntico independiente de emplear el planificador de Voronoi o Grafo de Visibilidad, salvo ciertos cambios menores en este último, que serán explicados en su apartado.

Dicho mapa estará disponible en formato mapa de ocupación y cartesiano, pues resultará útil para las diferentes acciones a realizar sobre el mismo. Un mapa de ocupación divide su espacio en celdas de ocupación binarias, cuyos valores puedes ser 1 si están libres o 0 si están ocupadas por obstáculos, es decir, zonas no accesibles.

El usuario debe de suministrar algunas consignas de entrada, tales como:

- Tamaño del mapa de ocupación: *N x M* celdas
- Límites cartesianos (*xmin, xmax, ymin, ymax*)
- Número de obstáculos del tipo rectángulo
- Número de obstáculos del tipo circunferencia
- Tamaño del robot MASAR

En el inicio del código mostrado a continuación se muestran los valores introducidos para el ejemplo:

```
%Entradads del usuario
Nx = 200;
Ny = 200;
xmin = 1;
ymin = -1;
Delta = 4;
xmax = xmin + Delta;
ymax = ymin + Delta;

masar = 0.1;

n_circulos = 20;
n_rectangulos = 20;
```

4.1.1. Generación de obstáculos

Para la generación de los obstáculos y la situación aleatoria de los mismos se han implementado dos funciones propias: *Circunferencia* y *Rectángulo*. Ambas funciones basan su funcionamiento en el empleo de la función *rand* de MATLAB, la cual genera un valor aleatorio entre 0 y 1, existiendo la posibilidad de generar un intervalo propio (b,a) por parte del usuario (Ecuación 4.1). El tamaño máximo de los obstáculos vendrá dado por el tamaño total del mapa y un factor reductor definible por el usuario.

$$r = a + (b - a) * rand \tag{4.1}$$

Dentro de la función responsable de generar los rectángulos, se establece de forma aleatoria la longitud de la base del mismo, la altura y la situación de su vértice inferior izquierdo [*xr,yr*]. Tanto la longitud de la base y la altura, como la posición del vértice vendrá limitada por la constante *delta*, que es el tamaño total del mapa, dividido entre 8, generando así la variable *lim*. De esto modo se asegura una composición del mapa acorde a su tamaño. Se muestra el código de dicha función, donde también se aprecia el uso de la función *rand* combinado con la variable limitadora *lim* y *Delta*:

```
function [xr,yr,altura,base]=rectangulo(Delta,xmin,ymin)
```

```
2
3  lim=Delta/8;
4   %Base aleatoria, con un minimo de 0.2 y un maximo de lim
5  base = 0.2+(lim-0.2)*rand;
6   %Altura aleatoria, con un minimo de 0.2 y un maximo de lim
7  altura = 0.2+(lim-0.2)*rand;
8   %xr aleatorio entre xmin y xmax
9  xr = xmin+(xmin+Delta-xmin)*rand;
10  %yr aleatorio entre ymin e ymax
11 yr = ymin+(ymin+Delta-ymin)*rand;
12
13 end
```

Por otro lado, en la circunferencia se establece de forma aleatoria el radio y la posición de su centro [*xc,yc*], también limitados por la constante *delta*. A continuación, se describe el código de la función *Circunferencia*:

```
1  function [xc,yc,radio] = circunferencia(Delta,xmin,ymin)
2
3  lim=Delta/8;
4   %Radio aleatorio entre 0.1 y lim
5  radio = 0.1+(lim-0.1)*rand;
6   %xc aleatorio entre xmin y xmax
7  xc = xmin+(xmin+Δ-xmin)*rand;
8   %yc aleatorio entre ymin e ymax
9  yc = ymin+(ymin+Δ-ymin)*rand;
10
11 end
```

Estos valores geométricos serán las variables de salida de la funciones, y se almacenarán en vectores de información (uno para rectángulos y otro para circunferencias), los cuales almacenarán de forma ordenada estos valores geométricos de cada uno de los obstáculos.

Cada una de las funciones es llamada desde un bucle *for* tantas veces como obstáculos se deban generar. Dentro de dicho bucle será donde también se completen los vectores de información de los obstáculos. El almacenamiento de información se realiza en el siguiente orden:

$$datos\_circulos = [xc_1,\ yc_1,\ radio_1,\ \ldots\ xc_n,\ yc_n,\ radio_n] \tag{4.2}$$

$$datos\_rectangulos = [xr_1,\ yr_1,\ altura_1,\ base_1,\ \ldots\ xr_n,\ yr_n,\ altura_n\ base_n] \tag{4.3}$$

Se emplean contadores para ir almacenando cada parámetro en su posición correcta dentro de los vectores de información. Los contadores se incrementan en cada iteración tantas posiciones como variables geométricas

se determinen para cada tipo de obstáculo, esto, 3 para el contador de las circunferencias, y 4 para el contador de los rectángulos.

La implementación de los bucles, que se ejecutan tantas veces como obstáculos haya, donde se forman los vectores de información, se ha realizado de la siguiente forma:

```
% Creacion de obstaculos
datos_circulos = [];
cont_circ=1;
datos_rectangulos = [];
cont_rect=1;

for i=1:1:n_circulos
    [xc,yc,radio] = circunferencia(Delta,xmin,ymin);
    datos_circulos(cont_circ)=xc;
    datos_circulos(cont_circ+1)=yc;
    datos_circulos(cont_circ+2)=radio;
    cont_circ=cont_circ+3;
end

for i=1:1:n_rectangulos
    [xr,yr,altura,base]=rectangulo(Delta,xmin,ymin);
    datos_rectangulos(cont_rect)=xr;
    datos_rectangulos(cont_rect+1)=yr;
    datos_rectangulos(cont_rect+2)=altura;
    datos_rectangulos(cont_rect+3)=base;
    cont_rect=cont_rect+4;
end
```

4.1.2. Mapa de ocupación, punto de inicio y punto final

Definidos los obstáculos mediante el tamaño y coordenadas de los mismos, se procede a crear el mapa de ocupación del entorno.

El primer paso será crear una matriz de unos (espacio vacío) de tamaño *Nx x Ny* que representa el entorno discretizado en celdas. El proceso está basado en recorrer celda por celda el entorno, hallar las coordenadas cartesianas de dicha celda, y tras ello, comprobar si dicho punto se encuentra dentro de algún obstáculo. De ser así, dicha celda se corresponde con un espacio prohibido en el mapa.

Para comprobar si un punto se encuentra dentro de una circunferencia, únicamente debe satisfacer la ecuación de la misma (Ecuación 4.4), siendo (*a*,*b*) el centro de la circunferencia, y (*x*,*y*) el punto cartesiano del mapa a evaluar.

$$(x-a)^2+(y-b)^2 <= r^2 \tag{4.4}$$

Para comprobar si un punto se encuentra dentro de un rectángulo, se ha implementado una función propia denominada *dentro_rectángulo*, que recibe

como parámetros los datos del rectángulo a comprobar (posición del vértice inferior izquierdo, base, altura y el punto a evaluar [*Px,Py*]).

$$salida = dentro\_rectangulo(xr,\ yr,\ altura,\ base,\ x,\ y) \tag{4.5}$$

En ella, se comprueba si la coordenada x del punto está entre xr y $xr+base$, y si la coordenada y del punto está entre yr y $yr+altura$. De ser así, la salida de la función será igual a 1.

La función se describe de la siguiente forma:

```
1  function [salida]=dentro_rectangulo(xr,yr,altura,base,Px,Py)
2
3  if Px≥xr && Px≤xr+base
4      if Py≥yr && Py≤yr+altura
5          salida=1;
6      else
7          salida=0;
8      end
9  else
10     salida=0;
11 end
12
13 end
```

Tanto los valores para comprobar si un punto se encuentra dentro de una circunferencia, como los parámetros de la función anteriormente descrita, son tomados de los vectores de información conformados en el apartado interior, y accediendo a ellos de forma ordenada empleando un contador. Se accederá a ellos tantas veces como obstáculos de ese tipo existan.

El proceso para crear el mapa de ocupación es el siguiente:

1. Comenzamos a recorrer el mapa empleando las celdas de ocupación, iniciando el proceso en el punto límite [*Nx=1 y Ny=1*], filas 3 y 5 del código siguiente.

2. A dicha celda, asociamos su coordenada cartesiana equivalente. Comenzamos con [*xmin,ymin*]

3. Comprobamos si ese punto se encuentra dentro de algún obstáculo (filas 7-11 y 13-18 del código), de ser así, dicha celda quedará ocupada, asignando un valor de 0 a la matriz creada al inicio del script (*mapa*), que representa el mapa en formato binario

4. Avanzamos en el proceso de recorrer el mapa, incrementando los valores de las coordenadas cartesianas, sumando *dx* y *dy* al punto actual (filas 20 y 22 del código). Volvemos al punto 2.

Dicho bucle se ejecutará *Nx x Ny* veces, para así estudiar todas las celdas. De este modo se ha implementado todo el proceso de comprobación de celdas ocupadas:

```
%Comprobacion de celdas ocupadas y vacias
x = xmin;
for i=1:Nx
    y = ymin;
    for j=1:Ny

        for k=1:3:cont_circ-2
            if (x-datos_circulos(k))^2 + ...
                (y-datos_circulos(k+1))^2 <= datos_circulos(k+2)^2
                mapa(i,j)=0;
            end
        end

        for l=1:4:cont_rect-3
            salida=dentro_rectangulo(datos_rectangulos(l),.,x,y);
            if salida==1
                mapa(i,j)=0;
            end
        end

        y = y + dy;
    end
    x = x + dx;
end
```

Para finalizar con el apartado, se procede a elegir de forma aleatoria un punto de inicio para el robot MASAR y un punto de destino. El proceso es muy sencillo. Para estimar el punto de inicio, se selecciona de forma aleatoria una celda del mapa, se comprueba mediante una condición dentro de un bucle *while* si esta celda se corresponde con una celda ocupada. De ser así, pasamos a seleccionar otra celda de forma aleatoria hasta encontrar una celda libre. Una vez encontrada dicha celda, se halla sus correspondientes coordenadas cartesianas.

```
%Inicio del recorrido en celda libre
%Primer punto generado
i_ini = ceil(Nx*rand);
j_ini = ceil(Ny*rand);
%Comprobacion de si la celda esta libre
while mapa(i_ini,j_ini)==0
    %Si esta ocupada, se busca otro punto
    i_ini = ceil(Nx*rand);
    j_ini = ceil(Ny*rand);
end
%Cartesiano
xini = (i_ini-1)*dx + xmin;
yini = (j_ini-1)*dy + ymin;
```

Luego, para la celda de destino el proceso es similar, pero se tiene en cuenta una condición añadida: el punto seleccionado debe de estar libre y además a una distancia mínima del punto de inicio. En este caso, para poder comprobar si se cumple la distancia de separación mínima (con valor igual a 2), se

debe conocer en todo momento las coordenadas cartesianas de la celda aleatoria seleccionada, para así hallar la distancia entre el punto de inicio y el candidato a final (Ecuación 4.6). El cálculo de las coordenadas cartesianas se realiza dentro del bucle *while*, filas 13 y 14 del siguiente código, y antes de entrar al bucle para realizar una comprobación inicial con el primer punto generado.

$$separacion = \sqrt{(xini - xfin)^2 + (yini - yfin)^2} \tag{4.6}$$

```
1  %Distancia minima entre inicio y fin
2  separacion_minima = 2;
3
4  %Final del recorrido
5  %Primer punto generado
6  i_fin = ceil(Nx*rand);
7  j_fin = ceil(Ny*rand);
8  xfin = (i_fin-1)*dx + xmin;
9  yfin = (j_fin-1)*dy + ymin;
10 separacion = sqrt( (xini-xfin)^2 + (yini-yfin)^2 );
11 %Comprobacion de celda libre y separacion minima
12 while mapa(i_fin,j_fin)==0 || separacion < separacion_minima
13 %Si la celda esta ocupada o no cumple la separacion, se ...
      genera otro punto
14     i_fin = ceil(Nx*rand);
15     j_fin = ceil(Ny*rand);
16     xfin = (i_fin-1)*dx + xmin;
17     yfin = (j_fin-1)*dy + ymin;
18     separacion = sqrt( (xini-xfin)^2 + (yini-yfin)^2 );
19 end
```

Para concluir con la explicación, se adjunta el código de todos los comandos comentados para generar el mapa.

```
1  %Entradas del usuario
2  Nx = 200;
3  Ny = 200;
4  xmin = 1;
5  ymin = -1;
6  Delta = 4;
7  xmax = xmin + Delta;
8  ymax = ymin + Delta;
9
10 masar = 0.1;
11
12 n_circulos = 20;
13 n_rectangulos = 20;
14
15 %Creacion del mapa de ocupacion
16 dx = (xmax-xmin)/(Nx-1);
17 dy = (ymax-ymin)/(Ny-1);
18
19 mapa = ones(Nx,Ny);
20
```

```
% Creacion de obstaculos
datos_circulos = [];
cont_circ=1;
datos_rectangulos = [];
cont_rect=1;

for i=1:1:n_circulos
    [xc,yc,radio] = circunferencia(Delta,xmin,ymin);
    datos_circulos(cont_circ)=xc;
    datos_circulos(cont_circ+1)=yc;
    datos_circulos(cont_circ+2)=radio;
    cont_circ=cont_circ+3;
end

for i=1:1:n_rectangulos
    [xr,yr,altura,base]=rectangulo(Delta,xmin,ymin);
    datos_rectangulos(cont_rect)=xr;
    datos_rectangulos(cont_rect+1)=yr;
    datos_rectangulos(cont_rect+2)=altura;
    datos_rectangulos(cont_rect+3)=base;
    cont_rect=cont_rect+4;
end

%Comprobacion de celdas ocupadas y vacias
x = xmin;
for i=1:Nx
    y = ymin;
    for j=1:Ny

        for k=1:3:cont_circ-2
            if (x-datos_circulos(k))^2 + ...
                (y-datos_circulos(k+1))^2 <= datos_circulos(k+2)^2
                mapa(i,j)=0;
            end
        end

        for l=1:4:cont_rect-3
            salida=dentro_rectangulo(datos_rectangulos(l),.,x,y);
            if salida==1
                mapa(i,j)=0;
            end
        end

        y = y + dy;
    end
    x = x + dx;
end

%Inicio del recorrido en celda libre
i_ini = ceil(Nx*rand);
j_ini = ceil(Ny*rand);
while mapa(i_ini,j_ini)==0
    i_ini = ceil(Nx*rand);
    j_ini = ceil(Ny*rand);
end
%Cartesiano
xini = (i_ini-1)*dx + xmin;
```

```
77 yini = (j_ini-1)*dy + ymin;
78
79 %Distancia minima entre inicio y fin
80 separacion_minima = 2;
81
82 %Final del recorrido
83 i_fin = ceil(Nx*rand);
84 j_fin = ceil(Ny*rand);
85 xfin = (i_fin-1)*dx + xmin;
86 yfin = (j_fin-1)*dy + ymin;
87 separacion = sqrt( (xini-xfin)^2 + (yini-yfin)^2 );
88 while mapa(i_fin,j_fin)==0 || separacion < separacion_minima
89     i_fin = ceil(Nx*rand);
90     j_fin = ceil(Ny*rand);
91     xfin = (i_fin-1)*dx + xmin;
92     yfin = (j_fin-1)*dy + ymin;
93     separacion = sqrt( (xini-xfin)^2 + (yini-yfin)^2 );
94 end
```

Se muestran 2 ejemplos diferentes de mapas generados de forma aleatoria con 20 rectángulos y 20 circunferencias. La Figura 4.2 muestra el mapa representado mediante coordenadas cartesianas, marcando los puntos de inicio y final del recorrido, mientras que la Figura 4.3 representa el mapa en formato ocupación mediante el comando *imshow*, que aparece rotado ya quelos ejes se intercambian.

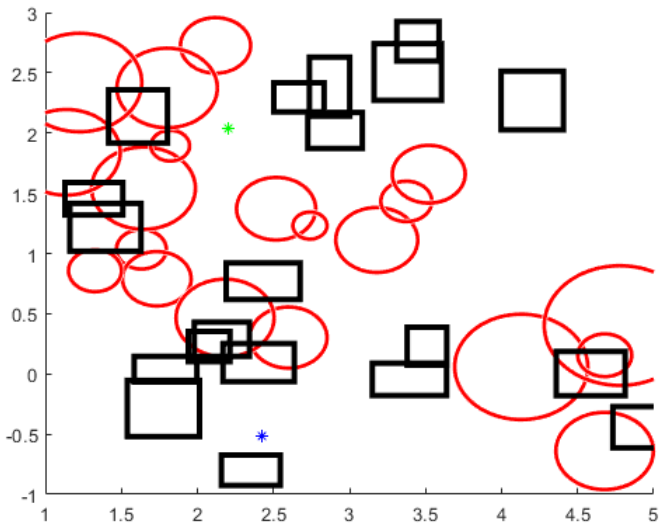

FIGURA 4.2: Ejemplo de mapa aleatorio en formato cartesiano

4.2. Diagrama de Voronoi

El Diagrama de Voronoi es una técnica de búsqueda de grafo cuya función es la construcción del grafo. Basa su funcionamiento en la búsqueda de

FIGURA 4.3: Ejemplo de mapa aleatorio en formato ocupación

maximizar la distancia entre el robot y los obstáculos presentes en el entorno (Figura 4.4).

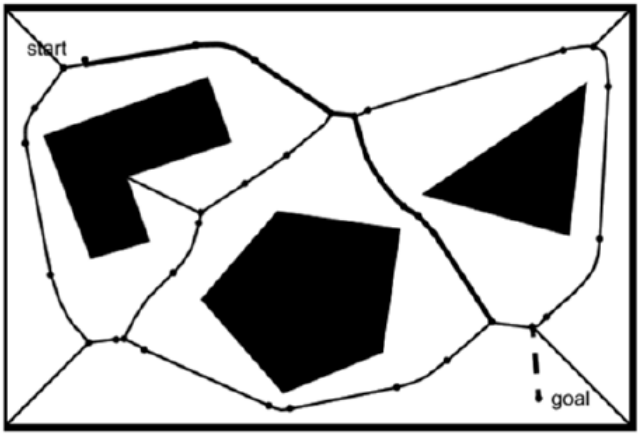

FIGURA 4.4: Diagrama de Voronoi

Permite generar los caminos dentro del mapa que mayor distancia entre obstáculos consiguen, para así evitar posibles colisiones. Se realiza uniendo entre sí los puntos de obstáculos colindantes y trazando las mediatrices de los segmentos de unión. Las intersecciones de estas mediatrices forman una serie de polígonos alrededor de los objetos, que estos son los caminos de Voronoi. Estos segmentos o caminos serán rectilíneos o parabólicos.

Como una definición más generalizada y aplicada a un problema de robótica móvil, se podría decir que el Diagrama de Voronoi es el lugar geométrico de los puntos del espacio libre que se encuentran a igual distancia entre sus dos obstáculos más próximos.

La toolbox de MATLAB de acceso libre proporcionada por Peter Corke contiene una función denominada *ithin* (Ecuación 4.7) que, recibiendo como parámetro de entrada un mapa de ocupación, devuelve como salida los caminos obtenidos aplicando el criterio de máxima distancia entre obstáculos. Esta salida de encuentra en formato matriz de mapa de ocupación, del tamaño del mapa suministrado, marcando las celdas que conforman los caminos con 1.

$$caminos = ithin(mapa) \tag{4.7}$$

Antes de suministrar el mapa a la función mencionada, se deben de realizar ciertas modificaciones para sacar mayor provecho al algoritmo, estas son:

- Marcar los bordes del mapa como obstáculos
- Marcar el inicio y el final del camino como obstáculos, para así obligar al algoritmo a generar caminos que pasen por estos puntos

```
%Inicio y final del recorrido como celdas ocupadas
mapa(i_ini,j_ini)=0;
mapa(i_fin,j_fin)=0;

%Margenes ocupados
mapa(1,:)=0;
mapa(end,:)=0;
mapa(:,1)=0;
mapa(:,end)=0;
```

Hecho esto, llamamos al comando *ithin* y devolvemos las celdas de inicio y final de recorrido al grupo de celdas vacías:

```
caminos = ithin(mapa);
caminos(i_ini,j_ini)=1;
caminos(i_fin,j_fin)=1;
```

Se muestran algunos ejemplos del Diagrama de Voronoi obtenidos para mapas geométricos aleatorios (Figuras 4.5 y 4.6).

4.3. Grafo de Visibilidad

El segundo método de planificación de caminos implementado y estudiado ha sido el método de Grafos de Visibilidad. Al igual que Voronoi, estamos ante una técnica de construcción de grafo basada en mapas geométricos.

A diferencia del anterior método de planificación, el cual trata de maximizar la distancia entre obstáculos para evitar colisiones, en este caso se persigue que los caminos pasen tan cerca de los obstáculos como sea posible. De este modo, se generan posibles caminos de mínima distancia desde inicio

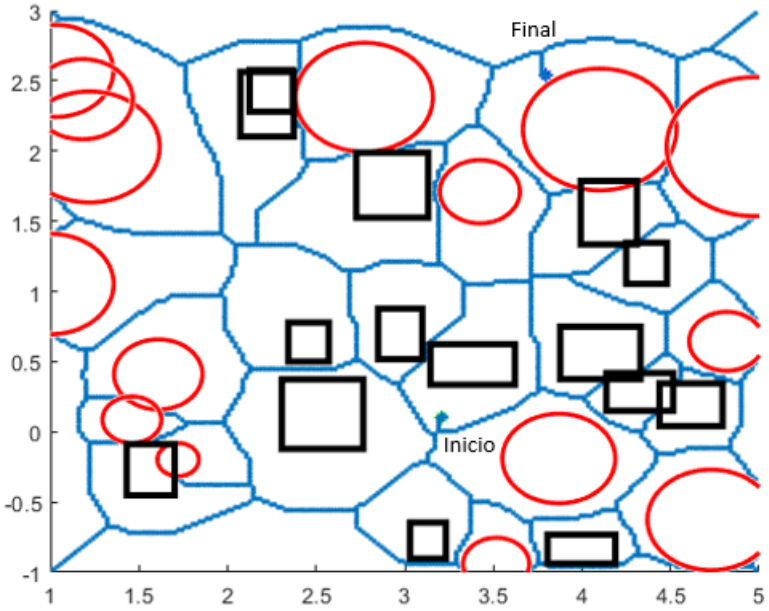

FIGURA 4.5: Caminos de Voronoi con 30 obstáculos

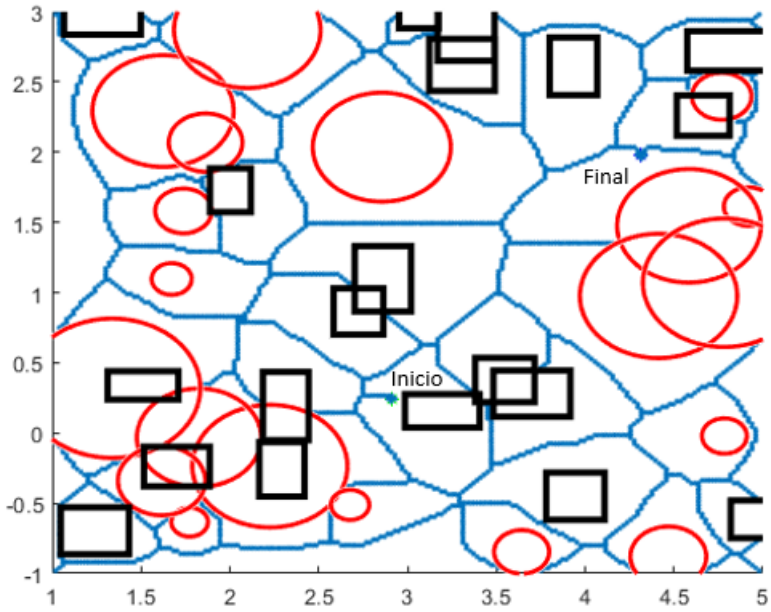

FIGURA 4.6: Caminos de Voronoi con 40 obstáculos

a objetivo, aunque supone asumir un riesgo mayor ante posible colisión del robot. Está basado en dos normas fundamentales:

- Los caminos se forman uniendo pares de vértices visibles desde cada uno de ellos
- Los obstáculos deben estar definidos como polígonos, para así disponer siempre de vértices

En la Figura se muestra un ejemplo general de Grafo de Visibilidad, uniendo un punto de inicio *qa* y un punto final *qf* con sus vértices visibles, y a partir de dichos vértices alcanzados, se generan nuevos segmentos con sus respectivos vértices visibles.

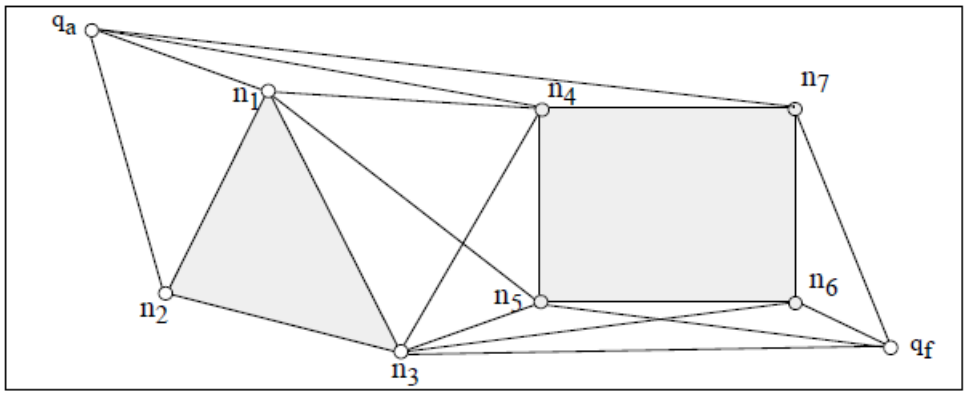

FIGURA 4.7: Ejemplo de Grado de Visibilidad

De forma más esquematizada, el proceso a seguir es el siguiente:

1. Se hallan los segmentos que unen el punto de inicio con sus vértices visibles
2. Se hallan los segmentos que unen el punto de meta con sus vértices visibles
3. Creación de una lista de vértices alcanzados
4. Desde cada uno de los vértices de esta lista, se dibujan segmentos hasta los vértices visibles desde cada uno de ellos

La desventaja de este método de planificación, junto a la ya citada carencia de seguridad respecto a los obstáculos, es el coste computacional que supone su implementación en mapas densos conformados con un gran número de obstáculos, pues a mayor número de enlaces generados.

El diseño del algoritmo se ha dividido en diversos pasos y funciones, tales como:

1. Modificación del mapa

2. Extracción de vértices
3. Generación de caminos
4. Discretización de caminos

4.3.1. Modificación del mapa

Recordando lo comentado en el apartado 4.1, el cual alberga la explicación de la formación de mapas aleatorios mediante obstáculos del tipo rectángulo y circunferencia, debemos rescatar dicha explicación para proponer una modificación.

Dicha modificación viene impuesta por la condición de que, para este método de planificación, todos los obstáculos deben tener vértices, y las circunferencias son una figura geométrica que carece de vértices. Para solventar este problema, se propone formar polígonos circunscritos sobre las circunferencias obstáculos (Figura 4.8).

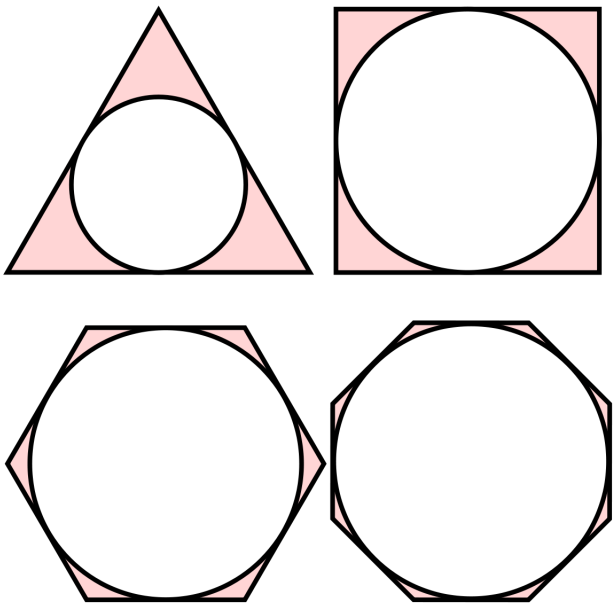

FIGURA 4.8: Polígonos circunscritos

A mayor número de lados del polígono circunscrito, mejor reproducción del perímetro de la circunferencia se logrará. Sin embargo, al elevar de este modo el número de vértices, el coste computacional se elevará muy notablemente. Para los ejemplos propuestos, se ha decidido que los polígonos circunscritos sean hexágonos.

Cálculo del polígonos circunscritos

Para generar el hexágono circunscrito sobre las circunferencias, y de tal modo extraer 6 vértices por cada una de las circunferencias, se modificado la

función *circunferencia* introducida en la generación del mapa en el apartado 4.1. Dentro de ella, aparte de generar de forma aleatoria una circunferencia dentro de unos límites establecidos, se calcularán los vértices del polígono circunscrito. Este nuevo proceso añadido será explicado a continuación.

Conociendo el número de lados del polígono circunscrito (*n_lados*), el primer paso será hallar los ángulos *alpha* que forman el segmento que une el centro de la circunferencia con los vértices (*xc - p*), y con el segmento que une el centro de la circunferencia con el punto medio del lado contiguo al vértice, de longitud igual al radio. Dichas variables se representan en la Figura 4.9.

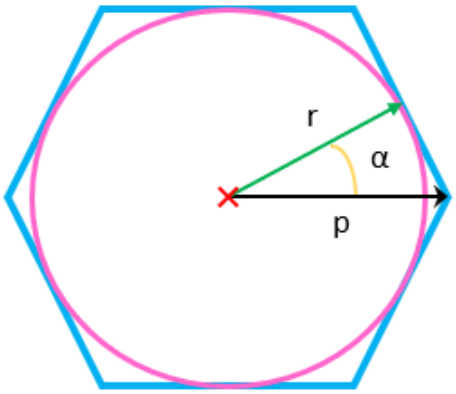

FIGURA 4.9: Cálculo de *alpha* y *p*

Para hallar el ángulo *alpha*, basta con dividir los 360º que conforman una circunferencia entre el doble del número total de lados (Ecuación 4.8):

$$\alpha = 360/2 * (n\_lados) \tag{4.8}$$

Calculado dicho ángulo, mediante trigonometría es posible obtener la longitud del segmento *p* (Figura 4.9) de la siguiente forma:

$$p = r/\cos(\alpha) \tag{4.9}$$

Teniendo en cuenta que el cálculo explicado se realiza con la circunferencia se encuentra en el origen, y que el cálculo de *p* se ha realizado con el vértice de coordenada *x* positiva y coordenada *y* nula, el primer vértice el polígono será (Ecuación 4.10):

$$vertice1 = (p, 0) \tag{4.10}$$

Finalmente, para completar el polígono con los demás vértices, se rotará el segmento *centro - p* una magnitud angular de *2alpha* tantas veces como vértices restantes se deban calcular. Luego, se sumará al vértice hallado la posición real del centro de la circunferencia *[xc, yc]*.

La expresión matemática para ejecutar las rotaciones se encuentra dentro de un bucle que se ejecuta *n_lados* veces, y es la siguiente:

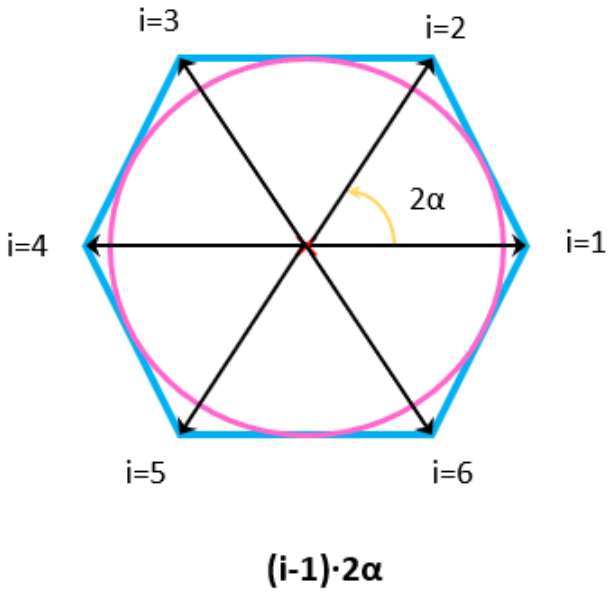

FIGURA 4.10: Rotaciones 2*alpha*

$$2\alpha = 2pi/n\_lados \tag{4.11}$$

$$vertice(i) = p * [\cos((i-1) * 2\alpha) \;\; \sin((i-1) * 2\alpha)] \tag{4.12}$$

El nuevo código de la función generadora de obstáculos aleatorios, cuya modificación incluye la extracción de vértices, es el siguiente:

```
function [xc,yc,radio,vertice] = ...
    circunferencia_poligono(Delta,xmin,ymin,n_lados)

vertices = [];
%Generamos el circulo
lim=Delta/8;
%Radio aleatorio
radio = 0.1+(lim-0.1)*rand;
%xc aleatorio
xc = xmin+(xmin+Delta-xmin)*rand;
%yc aleatorio
yc = ymin+(ymin+Delta-ymin)*rand;

%Poligono circunscrito
alpha = (2*pi)/(2*n_lados);
p = radio/cos(alpha);

for i=1:n_lados
    vertice(i,:) = p*[cos((i-1)*((2*pi)/n_lados)) ...
        sin((i-1)*((2*pi)/n_lados))];
    vertice(i,1) = vertice(i,1)+xc;
    vertice(i,2) = vertice(i,2)+yc;
end
```

```
22
23 end
```

Y en su llamada en la creación del mapa, en lugar de almacenar únicamente el radio y el centro en el vector de información, se crea un nuevo vector de vértices (fila 8 del siguiente código):

```
1  %Creamos por un lado los circulos, con los vertices del poligono
2  %circunscrito
3  for i=1:1:n_circulos
4      [xc,yc,radio,vertices_pol] = ...
           circunferencia_poligono(Delta,xmin,ymin,n_lados);
5      datos_circulos(cont_circ)=xc;
6      datos_circulos(cont_circ+1)=yc;
7      datos_circulos(cont_circ+2)=radio;
8      vertices_poligonos = [vertices_poligonos;vertices_pol];
9      cont_circ=cont_circ+3;
10 end
```

Comprobación de punto dentro de polígono circunscrito

Para la composición del mapa de ocupación, es necesario identificar qué celdas están libres y cuales están ocupadas por obstáculos. Anteriormente, bastaba con comprobar si un punto se encontraba en el interior de un rectángulo o circunferencia para marcar su celda correspondiente como ocupada.

Al sustituir las circunferencias por polígonos, dicha comprobación presenta mayores dificultades, y por tanto, un algoritmo específico para ello. Se empleará el algoritmo descrito en *https://demonstrations.wolfram.com/AnEfficientTestForAPointToBeInAConvexPolygon/*.

Queremos comprobar si un punto P pertenece al polígono.

1. Primero se resta a todos los vértices del polígono el punto P (Vértices - P). En adelante, trabajaremos con los vértices obtenidos como resultado de esta resta. Lo que hace esto es desplazar el polígono de forma que el nuevo origen de coordenadas sea P.

2. A continuación, se recorren los vértices del polígono (del polígono desplazado) en sentido antihorario, empezando por cualquier vértice. Por ejemplo, si es un pentágono, se podría recorrer como se muestra en la Figura 4.11.

3. Se realiza la siguiente comparación, para cada par de vértices sucesivos (Ecuación 4.13 y Figura 4.12).

4. Si el signo de dicha comparativa resulta ser igual para todos los pares de vértices sucesivos, el punto evaluado está dentro del polígono.

Si el polígono dispone de 5 lados, la comparativa se realizará 5 veces (5 pares de vértices).

$$X_{i+1} * Y_i - X_i * Y_{i+1} \quad > 0 \ || < 0 \tag{4.13}$$

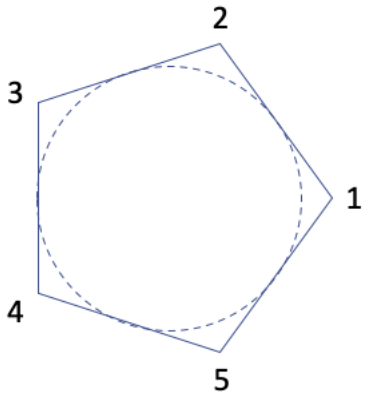

FIGURA 4.11: Recorrido del polígono

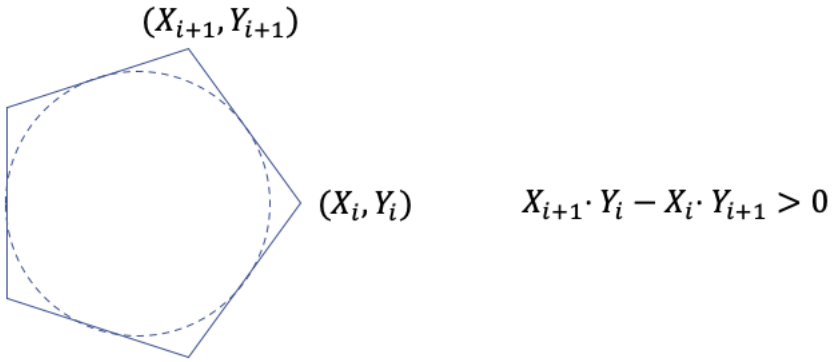

FIGURA 4.12: Comparación a evaluar

El código de la función es el siguiente:

```
1  function [salida]=dentro_poligono(n_lados,punto,vertices)
2
3  signo_ant = 0;
4  %De partida se asume que el punto esta dentro
5  salida = 1;
6
7  %Trasladamos origen del poligono al punto, y por tanto, los ...
       vertices
8  for i=1:n_lados
9      vertices_new(i,1) = vertices(i,1) - punto(1,1);
10     vertices_new(i,2) = vertices(i,2) - punto(1,2);
11 end
12
13 for j=1:n_lados
14
```

```
15      %Evaluamos la funcion comparativa
16      if j == n_lados
17          signo2 = sign(vertices_new(1,1)*vertices_new(j,2)
18          -vertices_new(j,1)*vertices_new(1,2));
19      else
20          signo2 = sign(vertices_new(j+1,1)*vertices_new(j,2)
21          -vertices_new(j,1)*vertices_new(j+1,2));
22      end
23
24      if j == 1
25          signo_ant = signo2;
26      else
27          if signo_ant ~= signo2
28              %Si el signo es diferente al anterior, el punto ...
                    esta fuera
29              salida = 0;
30              break;
31          end
32      end
33      signo_ant = signo2;
34  end
35  end
```

4.3.2. Extracción de vértices

El siguiente paso para el diseño del algoritmo de Grafo de Visibilidad es la creación de una base de datos de los vértices disponibles en el mapa.

Sabiendo que en el anterior apartado se han generado ya los vértices de los polígonos circunscritos, ahora se deben identificar los vértices que conforman los diferentes obstáculos del tipo rectángulo.

Una vez se tienen todos los vértices del mapa identificados, se ejecutará un filtrado para desechar los vértices que no serán útiles para generar caminos, estos son:

1. Vértices fuera del mapa
2. Vértices dentro de obstáculos

Vértices de rectángulos

Para identificar los vértices de los rectángulos, basta con acceder al vector de información de rectángulos *datos_rectángulos* tantas veces como rectángulos albergue el mapa, como se muestra en el bucle *for* del siguiente código. Una vez dentro, los vértices serán:

$$vertice1 = [xr, yr] \tag{4.14}$$

$$vertice2 = [xr + base, yr] \tag{4.15}$$

$$vertice3 = [xr, yr + altura] \tag{4.16}$$

$$vertice4 = [xr + base, yr + altura] \tag{4.17}$$

```
%Extraccion de vertices de rectangulos
for i=1:4:length(datos_rectangulos)-3
    vertice1 = [datos_rectangulos(i) datos_rectangulos(i+1)];
    vertice2 = [datos_rectangulos(i)+datos_rectangulos(i+3) ...
        datos_rectangulos(i+1)];
    vertice3 = [datos_rectangulos(i) ...
        datos_rectangulos(i+1)+datos_rectangulos(i+2)];
    vertice4 = [datos_rectangulos(i)+datos_rectangulos(i+3) ...
        datos_rectangulos(i+1)+datos_rectangulos(i+2)];

    vertices1 = [vertices1;vertice1;vertice2;vertice3;vertice4];
end
```

A estos vértices, se les añade los vértices anteriormente extraídos de los polígonos circunscritos (*vertices_poligonos*), formando la nueva lista de vértices *vertices2*.

Filtrado de vértices

Para realizar el filtrado de vértices que se encuentran fuera del mapa, bastará con comprobar que estos respetan los límites establecidos en la creación del mismo [*xmin,xmax,ymin,ymax*], tal y como se muestra en la condición *if* del siguiente bucle:

```
%Comprobamos que los vertices de rectangulos y poligonos ...
   estan dentro de los limites del mapa
for j=1:length(vertices2)
    if vertices2(j,1)≥xmin && vertices2(j,1)≤xmax && ...
        vertices2(j,2)≥ymin && vertices2(j,2)≤ymax
         vertices3=[vertices3;vertices2(j,:)];
    end
end
```

Se genera una nueva base de datos de vértices filtrada, *vertices3*.

El último filtrado eliminará los vértices que se encuentran dentro de cualquier obstáculo, pues estos vértices son inaccesibles para los posibles futuros caminos. Para ello, se comprobará si los vértices están dentro de un rectángulo, de ser así, se rechazará. Si no, se comprobará si está dentro de un polígono.

Se emplearán las funciones explicadas con anterioridad para dichas comprobaciones, *dentro_rectangulo* y *dentro_poligono*. Solo existe una modificación en la función *dentro_rectangulo*, en la cual ahora se consideran los bordes de

los rectángulos como exterior del rectángulo, para evitar etiquetar a un vértice como interino de su propio rectángulo.

```
1  %Comprobamos si los vertices se encuentran dentro de obstaculos
2  for k=1:length(vertices3)
3
4      for z=1:4:length(datos_rectangulos)-3
5          salida=dentro_rectangulo_vertice(datos_rectangulos(z)
6          ,datos_rectangulos(z+1),datos_rectangulos(z+2),
7          datos_rectangulos(z+3),vertices3(k,1),vertices3(k,2));
8          if salida == 1
9              break;
10         end
11     end
12
13     if salida == 0
14      for m=1:n_lados:length(vertices_poligonos)
15          salida2 = dentro_poligono(n_lados,vertices3(k,:),
16          vertices_poligonos(m:m+n_lados-1,:));
17          if salida2 == 1
18              break;
19          end
20      end
21     end
22
23     if salida == 0 && salida2 == 0
24         vertices4=[vertices4;vertices3(k,:)];
25     end
26
27 end
```

Al comprobarse que el vértice no se encuentra dentro de ningún obstáculo (línea 23 del anterior código), se añade dicho punto a la nueva y definitiva base de datos *vertices4*. En la Figura 4.13 se muestran los vértices visitables o útiles para un mapa aleatorio.

4.3.3. Generación de caminos

Una vez identificados todos los vértices disponibles, la siguiente fase es establecer la conexión entre vértices visibles. El proceso se divide en dos pasos:

1. Caminos desde inicio a vértices visibles y caminos desde meta a vértices visibles (Figura 4.14)

2. Desde estos vértices visitados, se hallan los caminos con los vértices visibles desde ellos

Para comprobar si un vértice es visible desde otro, se trazará una recta imaginaria entre ellos, discretizándola en sus diferentes puntos cartesianos, y se comprobará si alguno de estos puntos se encuentra dentro de un obstáculo. De ser así, existe un obstáculo entre los vértices, y por tanto no son

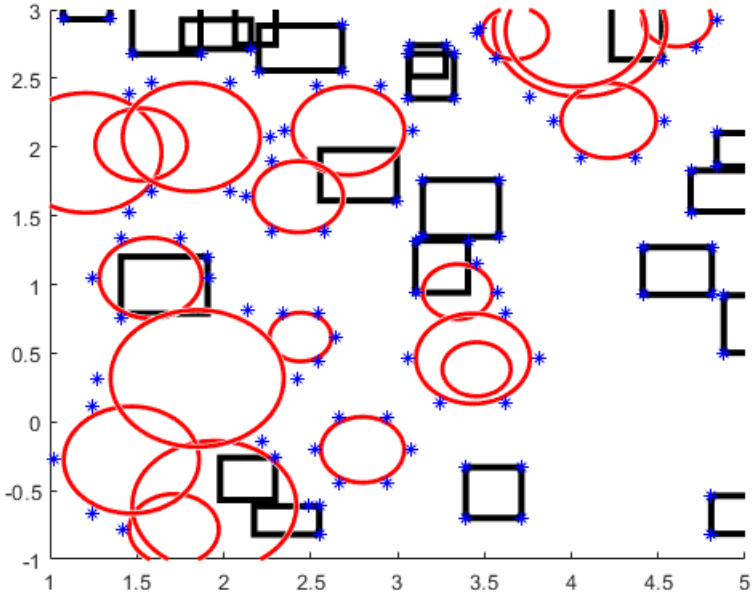

FIGURA 4.13: Vértices candidatos mostrados en color azul

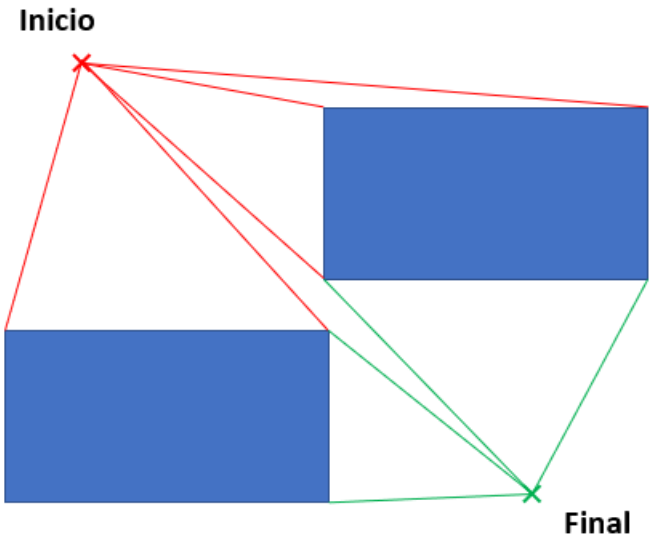

FIGURA 4.14: Caminos desde inicio/fin a vértices visibles

visibles entre ellos. Figura 4.15.

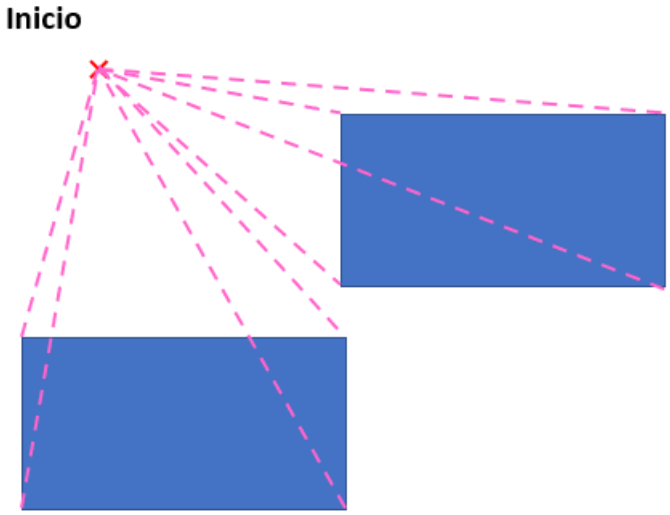

FIGURA 4.15: Discretización de posibles caminos

A continuación, se procede a explicar el proceso de forma más detallada junto con el código pertinente. La función ha sido denominada *Grafo_Visibilidad*, y recibe como parámetros de entrada: los puntos inicial y final, los vértices filtrados anteriormente, los vértices de los polígonos por separado, el número de lados del polígono circunscrito y el vector de información de los rectángulos.

$$Grafo\_Visibilidad(inicio, fin, vertices, vertices\_poligonos, n\_lados, datos\_rectangulos) \tag{4.18}$$

Primero, se buscarán todos los vértices visibles desde lo puntos inicio y final. Para ello, se recorrerá un bucle de tamaño igual al número total de vértices a examinar. Dentro el bucle, se halla la diferencia en eje x y en eje y desde los puntos inicio y final hasta dicho vértice. Estos diferencias se discretizan dividiéndolas entre 100.

```
1  %Caminos desde inicio y final hasta vertices
2  for i=1:length(vertices)
3
4      flag_ini = 0;
5      flag_fin = 0;
6
7      incremento_x_ini = (vertices(i,1)-inicio(1,1))/100;
8      incremento_y_ini = (vertices(i,2)-inicio(1,2))/100;
9
10     incremento_x_fin = (vertices(i,1)-fin(1,1))/100;
11     incremento_y_fin = (vertices(i,2)-fin(1,2))/100;
```

Luego, se recorrerá estas rectas imaginarias discretizadas punto por punto, dentro de un bucle for igual al tamaño del paso restando una unidad. Tendremos las coordenadas de los puntos a evaluar, tanto de la recta que une el inicio con el vértice, como la que une el final con el vértice:

```
    for j=1:99

        x_ini = inicio(1,1) + incremento_x_ini*j;
        y_ini = inicio(1,2) + incremento_y_ini*j;

        x_fin = fin(1,1) + incremento_x_fin*j;
        y_fin = fin(1,2) + incremento_y_fin*j;
```

Para cada uno de estos dos puntos, se comprobará si se encuentran dentro de rectángulos o de polígonos circunscritos. Si la salida de las funciones es igual a 1, el punto se encuentra dentro de un obstáculo.

```
%Comprobacion de rectangulos
    for z=1:4:length(datos_rectangulos)-3
        [salida1]=dentro_rectangulo_vertice(datos_rectangulos(z),
        datos_rectangulos(z+1),datos_rectangulos(z+2),
        datos_rectangulos(z+3),x_ini,y_ini);
           if salida1 == 1
            break;
        end
    end

    for z=1:4:length(datos_rectangulos)-3
        [salida2]=dentro_rectangulo_vertice(datos_rectangulos(z),
        datos_rectangulos(z+1),datos_rectangulos(z+2),
        datos_rectangulos(z+3),x_fin,y_fin);
        if salida2 == 1
            break;
        end
    end

    %Comprobacion de poligono
    for m=1:n_lados:length(vertices_poligonos)
        salida3 = dentro_poligono(n_lados,[x_ini ...
           y_ini],vertices_poligonos(m:m+n_lados-1,:));
        if salida3 == 1 || salida1 == 1
            %Punto dentro de obstaculo
            flag_ini = 1;
            break;
        end
    end

    for m=1:n_lados:length(vertices_poligonos)
        salida4 = dentro_poligono(n_lados,[x_fin ...
           y_fin],vertices_poligonos(m:m+n_lados-1,:));
        if salida4 == 1 || salida2 == 1
            %Punto dentro de obstaculo
            flag_fin = 1;
            break;
```

```
36          end
37      end
```

Si tanto para el punto de la recta (*inicio - vértice*), como el punto de la recta (*final - vértice*) se encuentran dentro de un obstáculo, dicho vértice se descarta y se pasa al siguiente.

```
1       if flag_ini == 1 && flag_fin == 1
2           %Dejamos de comprobar mas puntos de esas rectas
3           break;
4       end
5   end %end de final de bucle para recorrer las rectas
```

Sin embargo, si se recorre alguna de las rectas y no se halla obstáculo, dicho vértice es visitable. Es necesario identificar si es visitable desde el punto inicio o desde el punto final.

```
1       if flag_ini == 0
2           %El punto inicio ha encontrado vertice
3           vertices_inicio = [vertices_inicio;vertices(i,:)];
4       end
5
6       if flag_fin == 0
7           %El punto final ha encontrado vertice
8           vertices_final = [vertices_final;vertices(i,:)];
9       end
10  end %end de final de bucle para recorrer vertices
```

Llegados a este punto, ya se conoce qué vértices han sido visitados partiendo desde el punto de inicio y desde el punto final, formando así un conjunto de vértices etiquetados como *visitados*:

```
1       visitados = [vertices_inicio;vertices_final];
```

En las Figuras 4.16 y 4.17 se muestran los caminos generados mediante el código anterior para el mapa dado.

La siguiente parte del proceso es prácticamente igual a la anterior, pero en vez de buscar vértices visibles desde los puntos inicial y final, se buscaran los vértices visibles desde todos los vértices visitados anteriormente (*visitados*). Tendremos por tanto 2 bucles para iniciar este segundo proceso. El primero selecciona un vértice de los visitados, y el segundo se encarga de relacionar cada uno de los vértices del mapa con el seleccionado. Se debe de hacer una distinción especial para que dicho vértice seleccionado no se compruebe consigo mismo.

```
1   for i=1:length(visitados)
2       for j=1:length(vertices)
3           %Aseguramos que no se visita a  s   mismo
4           distancia = norm(visitados(i,:) -vertices(j,:));
```

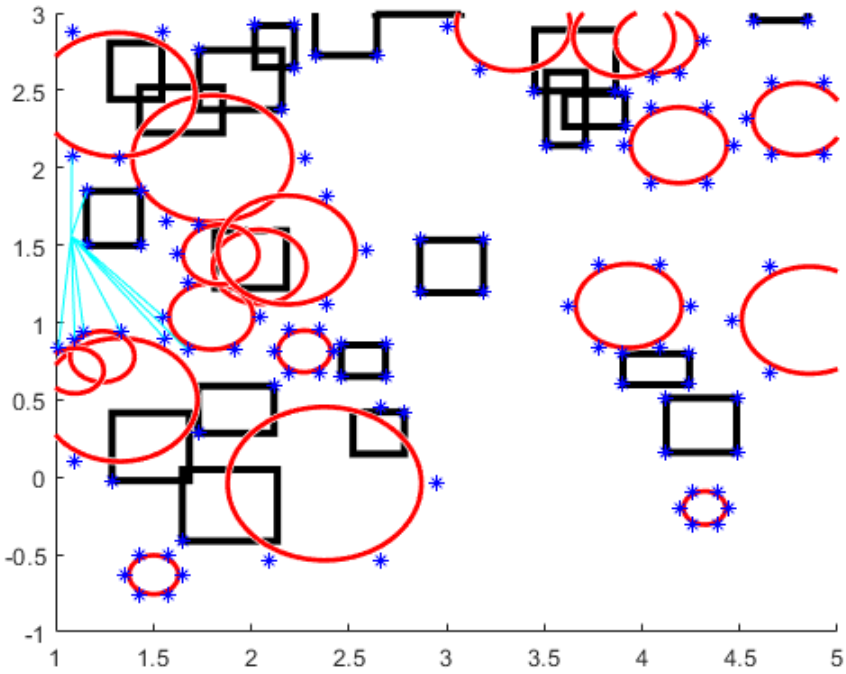

FIGURA 4.16: Caminos en cian generados entre punto de inicio y sus vértices visibles

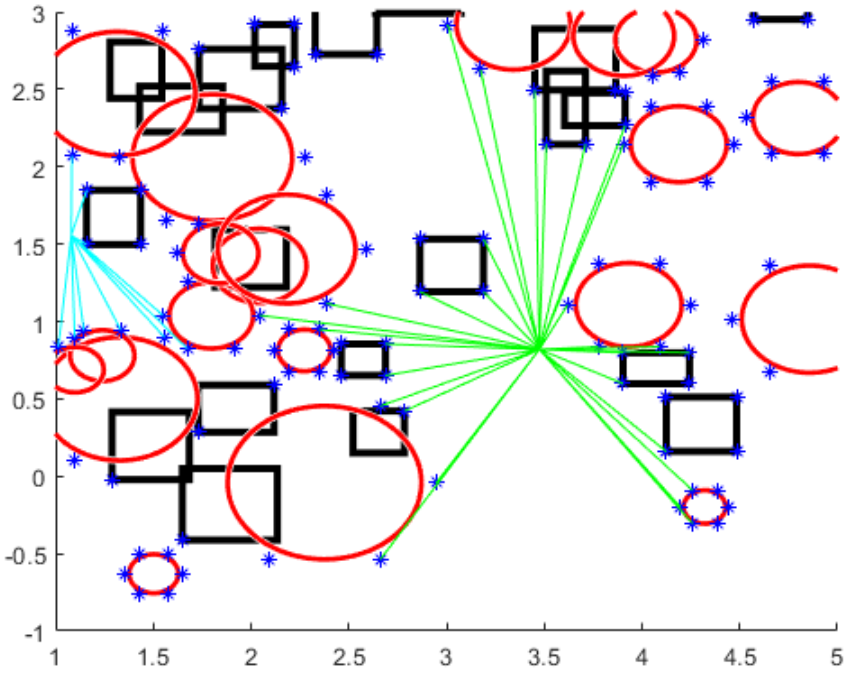

FIGURA 4.17: Caminos en verde generados entre punto final y sus vértices visibles

```
5           if distancia == 0
6               continue;
7           else %Continuamos con el proceso anteriormente descrito
```

Finalizado el algoritmo, se obtiene la gran cantidad de caminos intermedios que podemos ver en la Figura 4.18. A mayor número de obstáculos, mayor número de vértices a evaluar, y en conclusión, más posibles caminos obtenidos, generando así un mapa más denso lleno de posibilidades.

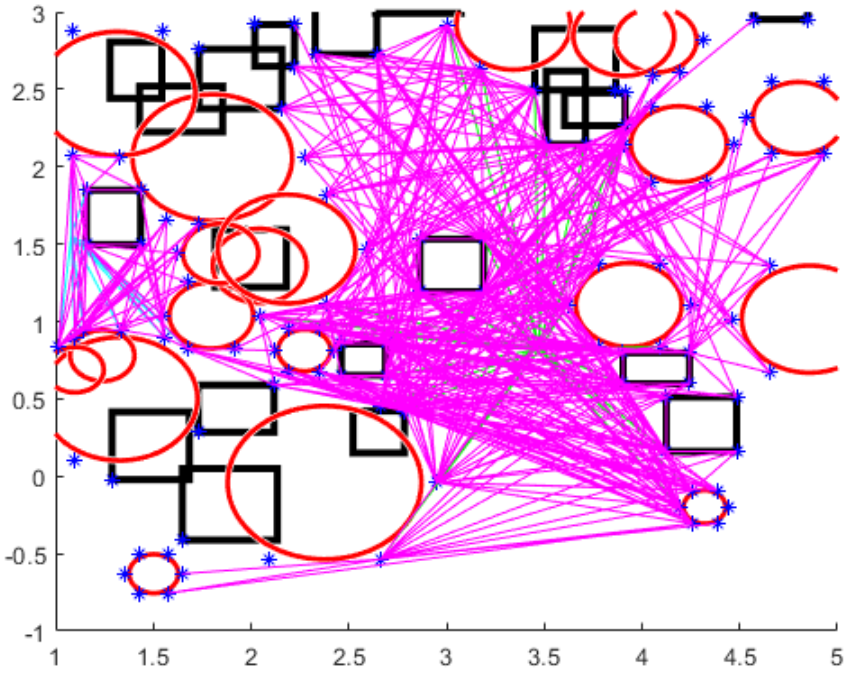

FIGURA 4.18: Caminos totales generados en magenta

Como variable de salida de este algoritmo, se define *caminos* como el conjunto de puntos que conforman todas las conexiones posibles, teniendo en cuenta las conexiones: inicio - vértices, final - vértices y las conexiones intermedias. Cada fila de dicha variable será un camino, compuesta por el punto de inicio del camino y el punto final.

Se adjunta el código completo de la función *Grafo de Visibilidad*:

```
1 function [caminos3] = ...
      Grafo_Visibilidad(inicio,fin,vertices,vertices_poligonos,
2 n_lados,datos_rectangulos)
3
4 vertices_inicio = [];
5 vertices_final = [];
6 vertices_inicio_fin = [];
7 vertices_final_fin = [];
8 visitados = [];
9 camino = [];
```

```
10 caminos = [];
11
12 %Caminos desde inicio y final hasta vertices
13 for i=1:length(vertices)
14
15     flag_ini = 0;
16     flag_fin = 0;
17
18     incremento_x_ini = (vertices(i,1)-inicio(1,1))/100;
19     incremento_y_ini = (vertices(i,2)-inicio(1,2))/100;
20
21     incremento_x_fin = (vertices(i,1)-fin(1,1))/100;
22     incremento_y_fin = (vertices(i,2)-fin(1,2))/100;
23
24     %Comprobaci n si la recta que los une pasa por alguna ...
          celda ocupada
25     %(dentro de rectagulo o poligono)
26     for j=1:99
27
28         x_ini = inicio(1,1) + incremento_x_ini*j;
29         y_ini = inicio(1,2) + incremento_y_ini*j;
30
31         x_fin = fin(1,1) + incremento_x_fin*j;
32         y_fin = fin(1,2) + incremento_y_fin*j;
33
34         %Comprobacion de rectangulos
35         for z=1:4:length(datos_rectangulos)-3
36             [salida1]=dentro_rectangulo_vertice(datos_
37             rectangulos(z),
38             datos_rectangulos(z+1),datos_rectangulos(z+2),
39             datos_rectangulos(z+3)
40             ,x_ini,y_ini);
41             if salida1 == 1
42                 break;
43             end
44         end
45
46         for z=1:4:length(datos_rectangulos)-3
47             [salida2]=dentro_rectangulo_vertice(datos_
48             rectangulos(z),datos_rectangulos(z+1),
49             datos_rectangulos(z+2),
50             datos_rectangulos(z+3),
51             x_fin,y_fin);
52             if salida2 == 1
53                 break;
54             end
55         end
56
57         %Comprobacion de poligono
58         for m=1:n_lados:length(vertices_poligonos)
59             salida3 = dentro_poligono(n_lados,[x_ini ...
                  y_ini],vertices_poligonos(m:m+n_lados-1,:));
60             if salida3 == 1 || salida1 == 1
61                 %Punto dentro de obstaculo
62                 flag_ini = 1;
63                 break;
64             end
```

```
65          end
66
67          for m=1:n_lados:length(vertices_poligonos)
68              salida4 = dentro_poligono(n_lados,[x_fin ...
                    y_fin],vertices_poligonos(m:m+n_lados-1,:));
69              if salida4 == 1 || salida2 == 1
70                  %Punto dentro de obstaculo
71                  flag_fin = 1;
72                  break;
73              end
74          end
75
76
77          if flag_ini == 1 && flag_fin == 1
78              %Dejamos de comprobar mas puntos de esas rectas
79              break;
80          end
81
82          %Si no se cumple, alguna recta todavia es candidata ...
                a vertice
83          %visible
84
85      end
86
87      %Se recorren las rectas
88      if flag_ini == 0
89          %El punto inicio ha encontrado vertice
90          vertices_inicio = [vertices_inicio;vertices(i,:)];
91      end
92
93      if flag_fin == 0
94          %El punto final ha encontrado vertice
95          vertices_final = [vertices_final;vertices(i,:)];
96      end
97  end
98
99
100 for i=1:length(vertices_inicio)
101     hold on
102     plot([inicio(1,1),vertices_inicio(i,1)],
103     [inicio(1,2),vertices_inicio(i,2)],'c',
104     'MarkerSize',10);
105     vertices_inicio_fin = [vertices_inicio_fin; inicio ...
            vertices_inicio(i,:)];
106
107 end
108
109
110 for j=1:length(vertices_final)
111     hold on
112     plot([fin(1,1),vertices_final(j,1)],
113     [fin(1,2),vertices_final(j,2)],'g',
114     'MarkerSize',10');
115     vertices_final_fin = [vertices_final_fin; fin ...
            vertices_final(j,:)];
116 end
117
```

```
118
119 visitados = [vertices_inicio;vertices_final];
120
121
122 %A partir de los alcanzados, buscamos sus visibles
123 for i=1:length(visitados)
124     for j=1:length(vertices)
125
126
127         flag = 0;
128         flag2 = 0;
129         flag3 = 0;
130         %Aseguramos que no se visita a  s   mismo
131         distancia = norm(visitados(i,:) -vertices(j,:));
132         if distancia == 0
133             continue;
134         else
135
136             %Ahora comprobamos si pasa por alguna celda ocupada
137             incremento_x = (vertices(j,1)-visitados(i,1))/100;
138             incremento_y = (vertices(j,2)-visitados(i,2))/100;
139
140             for z=1:99
141
142                 x = visitados(i,1) + incremento_x*z;
143                 y = visitados(i,2) + incremento_y*z;
144
145                 for m=1:4:length(datos_rectangulos)-3
146                     [salida1]=dentro_rectangulo_vertice(
147                     datos_rectangulos(m),datos_rectangulos(m+1),
148                     datos_rectangulos(m+2),
149                     datos_rectangulos(m+3),
150                     x,y);
151                     if salida1 == 1
152                         flag = 1;
153                         break;
154                     end
155                 end
156
157                 for m=1:n_lados:length(vertices_poligonos)
158                     salida2 = dentro_poligono(n_lados,[x ...
                            y],vertices_poligonos(m:m+n_lados-1,:));
159                     if salida2 == 1
160                         flag2 = 1;
161                         break;
162                     end
163                 end
164
165                 if flag == 1 || flag2 == 1
166                     flag3 = 1;
167                     break;
168                 end
169             end
170
171             if flag3 == 0
172                 camino = [visitados(i,:) vertices(j,:)];
173                 caminos = [caminos;camino];
```

```
174                 end
175             end
176         end
177 end
178
179 for t=1:1:length(caminos)
180     hold on
181     plot([caminos(t,1),caminos(t,3)],[caminos(t,2),caminos(t,4)],
182     'm');
183 end
184
185 caminos2 = [vertices_inicio_fin;caminos;vertices_final_fin];
186 caminos3 = unique(caminos2,'rows');
187
188 end
```

4.4. Algoritmo A* e identificación de estrecheces

En el apartado anterior, se han diseñado e implementado dos algoritmos para la construcción del grafo en mapas geométricos aleatorios. Hemos visto por un lado el empleo del Diagrama de Voronoi, que extrae caminos que pasan lo mas alejados posible de los obstáculos, y por otro lado, el Grafo de Visibilidad, cuya condición de mínima distancia respecto a los obstáculos genera una amplia variedad de posibilidades.

Ahora bien, una vez generadas todas las opciones para conectar el punto de inicio con el punto final, ¿cuál de todas estas rutas resulta ser la más eficiente, teniendo en cuenta longitud y viabilidad del camino? Para tomar dicha decisión, se propone aplicar un algoritmo de búsqueda de grafos como el *Algoritmo A**.

4.4.1. Algoritmo A*

El *Algoritmo A** basa su funcionamiento en el empleo de una función de evaluación heurística, etiquetando los nodos de la red determinando la probabilidad de dichos nodos de pertenecer al camino óptimo.

La función de evaluación del camino está compuesta a su vez por dos funciones:

- *g(n)*: representa la distancia entre el nodo origen del camino al nodo n
- *h(n)*: representa la distancia entre el nodo n y el nodo destino

La función *h(n)* es la función heurística, y como idea general, representa como de lejos está todavía el punto objetivo conforme se va recorriendo un posible camino. Luego, la función de evaluación *f(n)* es aquella que etiqueta cada nodo en función del coste que conlleva incluir el nodo en el camino óptimo (mérito). El proceso se basa en dos actividades:

- El algoritmo A* recorre y explora todas las celdas que conforman los caminos, así como sus celdas sucesoras. Este recorrido se realiza en función del mérito de las celdas.
- Creación de una lista ordenada de forma creciente con los méritos de los nodos con posibilidad de ser explorados, seleccionando el de menor valor.

A grandes rasgos, el proceso que se sigue es el siguiente:

1. Comenzando por el nodo origen, se calcula su mérito y se identifican sus nodos sucesores
2. Se calcula el mérito de estos nodos creando una lista ordenada, y se selecciona aquel con menor mérito
3. Realizado este sencillo proceso de forma repetida, seleccionando como nodo actual aquel de menor mérito de los sucesores anteriores, se alcanzará el nodo meta trazando el camino más corto

A continuación, se muestra el Algoritmo A* en formado pseudocódigo descrito en *http://idelab.uva.es/algoritmo*.

Algoritmo 1: Algoritmo A*

1.- Establecer el nodo s como origen. Hacer f(s) = 0, y f(i) = *inf* para todos los nodos i diferentes del nodo s. Iniciar el conjunto Q vacío.

2.- Calcular el valor de f(s) y mover el nodo s al conjunto Q .

3.- Seleccionar el nodo i del conjunto Q que presente menor valor de la función f(i) y eliminarlo del conjunto Q.

4.- Analizar los nodos vecinos j de i. Para cada enlace (i, j) con coste cij hacer:

4.1.-Calcular: f(j)' = g(i) + cij + h(j)

4.2.-Si f(j)' <f(j),

4.2.1.-Actualizar la etiqueta de j y su nuevo valor será: f(j) = g(i) + cij + h(j)

4.2.2.-Insertar el nodo j en Q

4.3.-Si f(j)' >= f(j)

4.3.1.-Dejar la etiqueta de j como está, con su valor f(j)

5.- Si Q está vacío el algoritmo se termina. Si no está vacío, volver al paso 3.

Para la implementación de el Algoritmo A* en este trabajo, se ha tomado una función de MATLAB implementada en un Trabajo de Final de Grado realizado por un antiguo colaborador del grupo de investigación ARVC del Departamento de Ingeniería de Sistemas y Automática (Zambrana, 2016. Dicha función empleada recibe como parámetros una matriz binaria con los caminos marcados como 1, obtenida del Diagrama de Voronoi o del Grafo de Visibilidad, el punto de inicio, el punto final, y devuelve el camino seleccionado que une inicio y final.

A continuación, se muestra un ejemplo de aplicación del algoritmo sobre un mapa aleatorio donde se ha extraído el Diagrama de Voronoi (Figura 4.19, camino verde), y un ejemplo aplicando el método de Grafos de Visibilidad (Figura 4.20, camino azul).

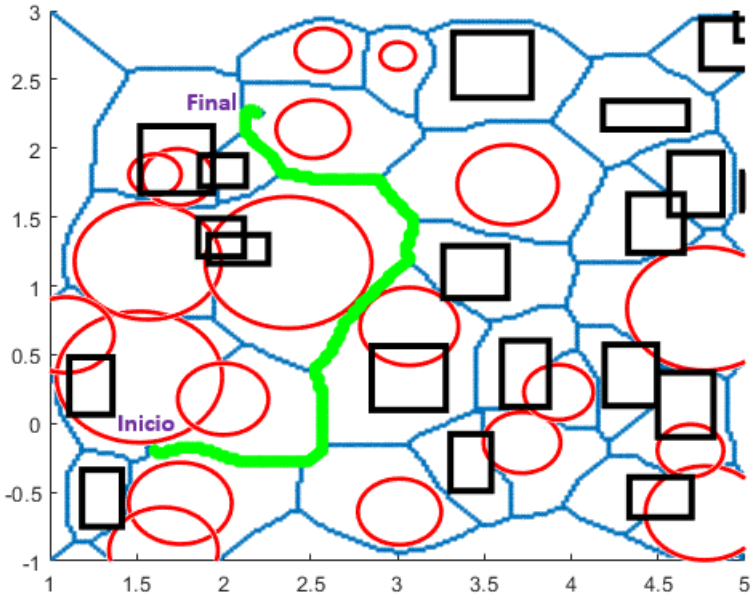

FIGURA 4.19: Camino óptimo a partir de Voronoi y A*

El Algoritmo A* devuelve, como se ha explicado y comprobado, el camino más corto entre un punto inicio y fin. La lógica nos lleva a pensar que, al ser el camino más corto, será aquel que menor coste de desplazamiento conlleve para el robot (tiempo de desplazamiento, distancia recorrida, tiempo de acción de los actuadores, etc...).

Sin embargo, el curioso modo de locomoción del robot MASAR deriva en cierta limitación de movimiento a lo largo de ciertas trayectorias. Estas trayectorias , estudiadas anteriormente en el Trabajo de Final de Grado del autor de este TFM, son aquellas que se encuentran entre dos obstáculos muy cercanos, denominadas estrecheces. Aquí, el robot no tiene la posibilidad de desplazarse mediante amplios giros de 180º debido a las posibles colisiones

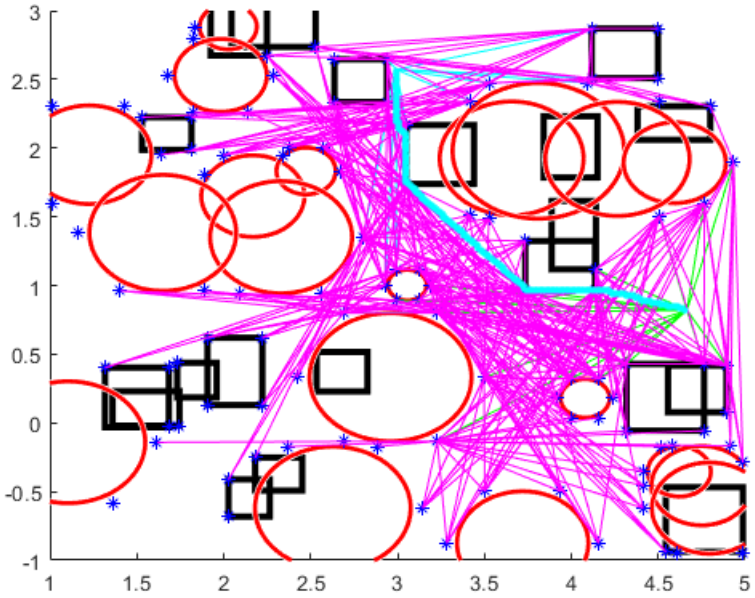

FIGURA 4.20: Camino óptimo a partir de Grafo de Visibilidad y A*

con el entorno. Como solución, se propuso un modo de locomoción de tipo serpiente, posando su cuerpo sobre las paredes de la estrechez de forma alterna (Figura 4.21).

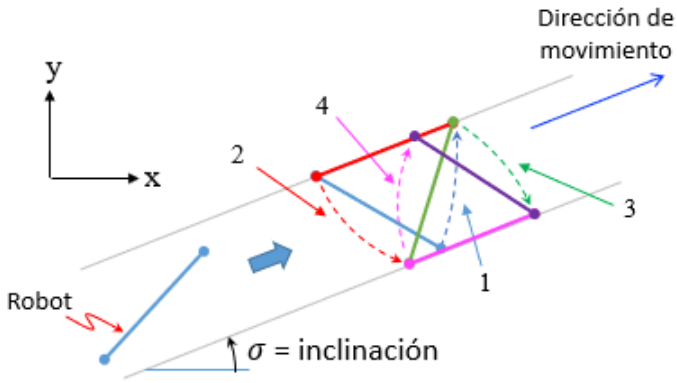

FIGURA 4.21: Desplazamiento a lo largo de una estrechez

Este modo de locomoción se caracteriza por cambiar en numerosas ocasiones el pivote fijado. El hecho de parar para despegar un pivote y fijar el otro, deriva en un aumento del coste de operación, relacionado con el tiempo que tarda el robot en recorrer un camino estrecho. A causa del tiempo que tarda el robot en pegar y despegar un pivote, un camino estrecho tardará más en ser recorrido que un camino libre, además de la limitación de amplitud de las rotaciones. Esto nos lleva a la conclusión de que, si el camino más corto proporcionado por el Algoritmo A* está plagado de estrecheces, seguramente no sea el camino óptimo (el que se recorra en menor tiempo posible) para

el robot MASAR.

Como solución a ello, se propone añadir una modificación al Algoritmo A* que penalice el mérito de aquellas celdas cercanas a obstáculos. De este modo, se generarán diversos caminos solución que penalicen en mayor o menor medida la cercanía del camino a obstáculos, evitando así tanto estrecheces, como posibles colisiones. De este modo, las estrecheces se verán muy penalizadas, aunque en menos medida también se penalizará la cercanía a obstáculos del método Grafo de Visibilidad.

Modificación del Algoritmo A*

Para esta modificación del Algoritmo A* aparecen 3 grupos de variables altamente importantes en este proceso, estas son:

- Variable de control: indica al algoritmo si debe penalizar cercanía a obstáculos o no
- Máscara: radio de circunferencia, la cual situará su centro en celda a evaluar
- Mapa en formato ocupación

El proceso a seguir para generar caminos que penalicen la cercanía a obstáculos es el siguiente:

1. Si la variable de control es verdadera, se accede a la parte del algoritmo que penaliza las celdas cercanas a obstáculos.
2. Sobre cada una de las celdas se sitúa una máscara de un radio introducido por el usuario (Figura 4.22), radio e)
3. En función del radio de la máscara, esta circunferencia abarcará más vecinos de la celda actual o menos. Se identifican cuantas celdas entran dentro de la máscara y se recorrerá cada una de ellas.
4. Dichas celdas vecinas se comparan con el mapa de ocupación, el cual identifica las celdas ocupadas por obstáculos (Figura 4.23, área magenta).
5. Una celda ocupada conlleva una penalización, es decir, un aumento del coste que supone al robot pasar por ese camino (Ecuación 4.19, penalización del mérito de la celda). Por tanto, se aumentan las posibilidades de que esta celda no sea candidata a formar el camino.
6. Cuantos más obstáculos tenga alrededor la celda actual, mayor penalización tendría dicha celda. En este caso, estaremos ante una estrechez.

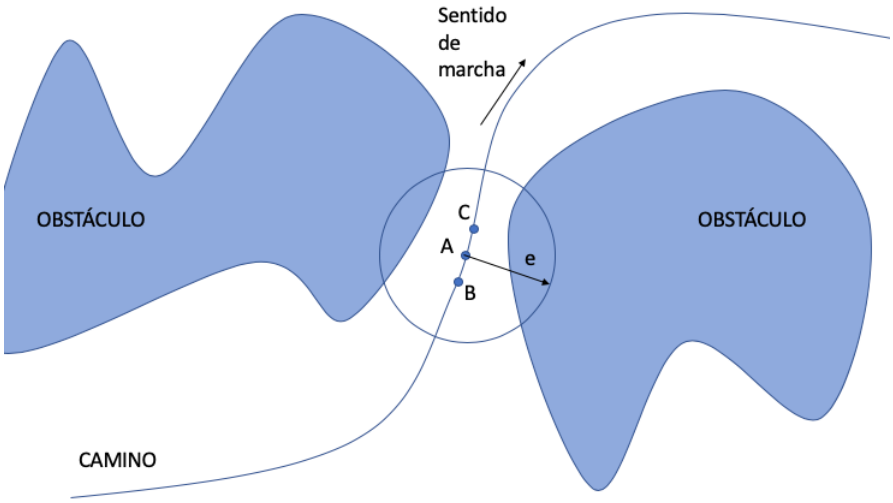

FIGURA 4.22: Aplicación de la máscara "*e*"sobre la posición actual

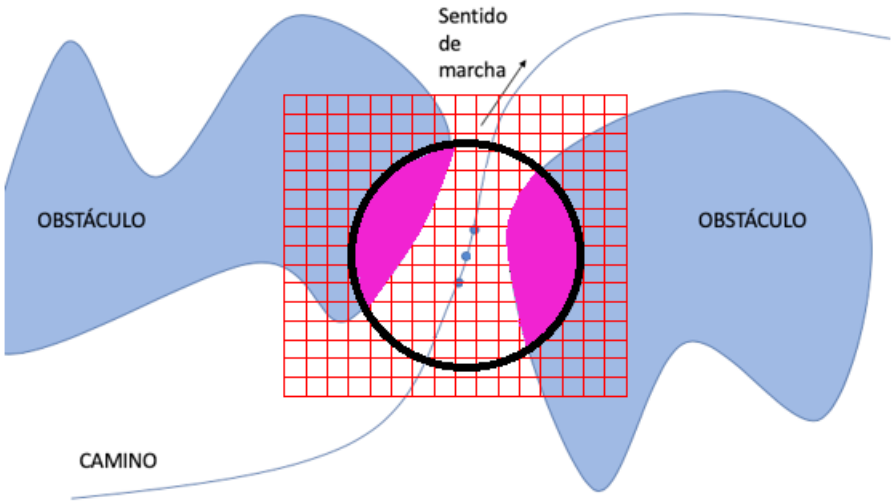

FIGURA 4.23: Identificación de obstáculos cercanos

$$coste\_celda = coste\_celda\_previo + n\_celdas\_obstaculo * penalizacion \quad (4.19)$$

En la Figura 4.24 se muestran 3 propuestas de camino proporcionadas por el algoritmo A* modificado, aplicado a un mapa aleatorio donde se ha ejecutado Voronoi. En color verde encontramos el camino sin penalización, el cual busca el camino más corto sin importar los obstáculos cercanos. En color magenta, es un camino propuesto con una penalización de radio de máscara igual a la longitud del robot MASAR, es decir, penaliza la cantidad de obstáculos que se encuentran dentro de una circunferencia con centro en el punto medio del robot, y radio igual a la longitud del mismo. Por ello, evita la estrechez que atraviesa el camino verde. Finalmente, el camino azul ha penalizado los obstáculos empleando un radio de máscara de 2 veces la longitud del robot, proponiendo un camino más largo y alejado de celdas ocupadas.

Aplicando este algoritmo a los caminos de Grafo de Visibilidad, se proponen dos caminos solución (Figura 4.25). En color azul podemos ver el camino propuesto sin restricciones, atravesando una zona estrecha. Mientras que en color verde, un camino alternativo que respeta mayores márgenes y no atraviesa la estrechez, se ha obtenido con un radio de máscara de 2 veces la longitud el robot.

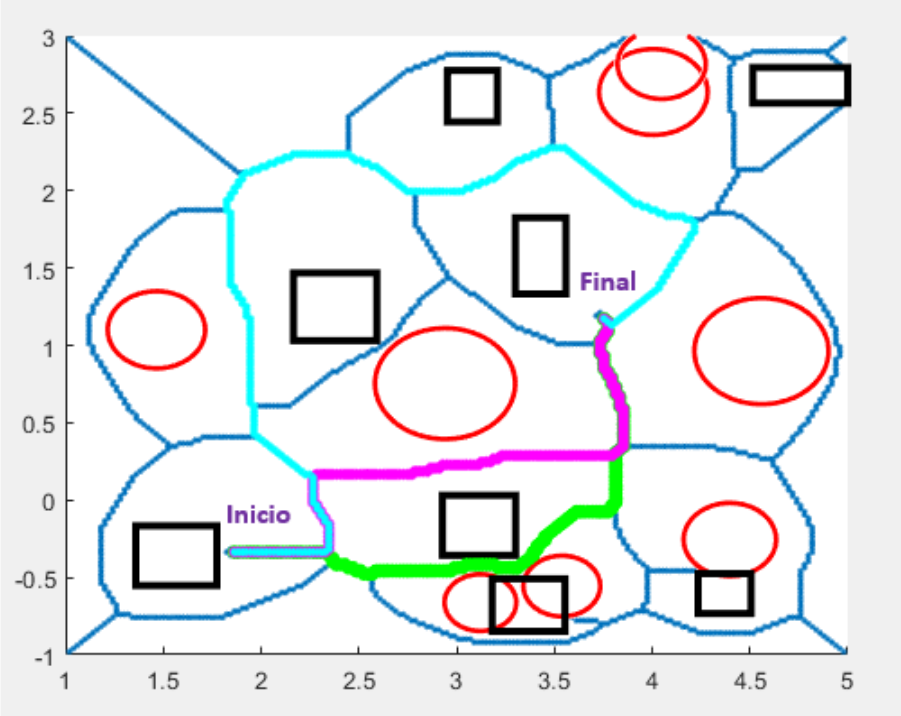

FIGURA 4.24: Caminos propuestos con Diagrama de Voronoi y A* modificado

A continuación, se muestra la modificación implementada para penalizar la cercanía a obstáculos:

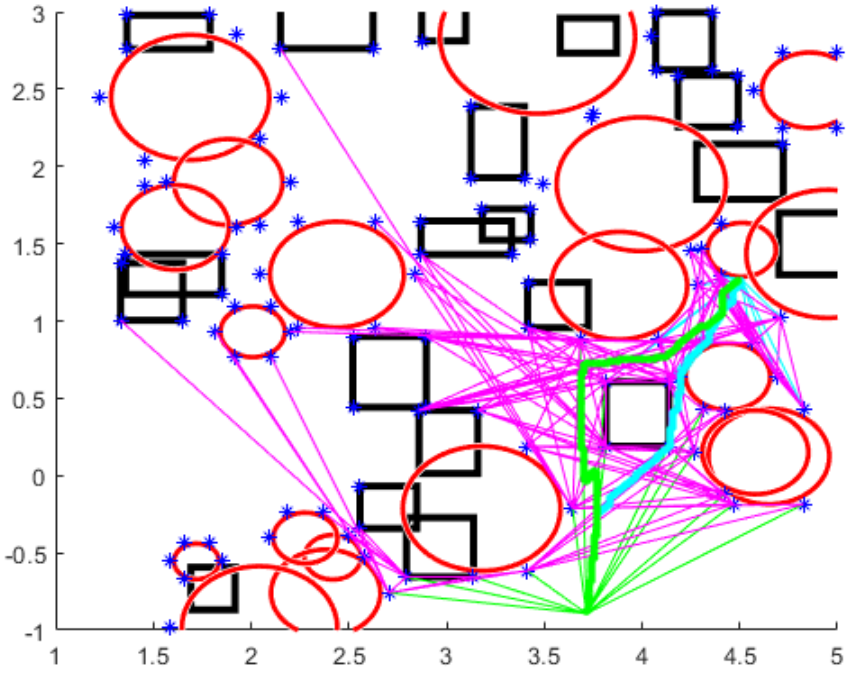

FIGURA 4.25: Caminos propuestos con Grafo de Visibilidad y A* modificado

```
if penalizar_estrecheces %Variable de activacion
    obstaculos_cercanos = 0;
    % Recorremos las celdas vecinas
    for da=-radio_mascara:1:radio_mascara
        a_vecino = a + da;
        for db=-radio_mascara:1:radio_mascara
            b_vecino = b + db;
            %Se comprueba si el vecino esta dentro del mapa
            if a_vecino>=1 && a_vecino<g && b_vecino>=1 && ...
               b_vecino<h
                %Se comprueba si el vecino esta ocupado
                if obstaculos_originales(a_vecino,b_vecino)==0
                    obstaculos_cercanos = ...
                       obstaculos_cercanos + 1;
                end
            else
                obstaculos_cercanos = obstaculos_cercanos + 1;
            end
        end
    end
    %Aumentamos el coste de pasar por dicha celda
    f_score(a,b) = f_score(a,b) + ...
       obstaculos_cercanos*penalizacion;
end
```

4.4.2. Identificación de estrecheces

Obtenidos los diferentes caminos solución para el mapa del entorno, resulta de alto interés identificar qué tramos de dichos caminos atraviesan zonas estrechas. Por ello, se ha diseñado una función propia, *busqueda_estrecheces*, para tal fin. Esta función recibe únicamente 3 parámetros de entrada:

- Camino a estudiar, obtenido de A*
- Puntos que conforman los obstáculos en formato cartesiano, lista con las coordenadas (x,y) de cada obstáculo
- Tamaño del robot MASAR, pues una estrechez será todo aquel camino de anchura inferior a dicho tamaño

Se explicará el proceso seguido para identificar puntos estrechos en combinación con el código de la función.

El primer paso del algoritmo es recorrer cada punto del camino. Para cada uno de los puntos, se aproxima la recta tangente en el punto actual, como la recta que pasa por el punto siguiente y el anterior al actual (Figura 4.26). Se deben distinguir los casos especiales cuando el punto actual es el inicial del camino, o el punto final.

```
for i=1:length(camino)
    punto=[camino(i,1) camino(i,2)];

    if i==1
        inicio = [camino(i,1) camino(i,2)];
        final = [camino(i+1,1) camino(i+1,2)];
    else if i==length(camino)
            inicio = [camino(i-1,1) camino(i-1,2)];
            final = [camino(i,1) camino(i,2)];
        else
            inicio = [camino(i-1,1) camino(i-1,2)];
            final = [camino(i+1,1) camino(i+1,2)];
        end
    end
```

Una vez identificados los puntos, se forma la ecuación general de la recta $Ax+By+C=0$.

```
A=final(2)-inicio(2);
B=-(final(1)-inicio(1));
C=-inicio(1)*(final(2)-inicio(2))+inicio(2)*
((final(1)-inicio(1)));
```

El siguiente paso es recorrer todos los puntos que conforman los obstáculos, e identificar qué puntos se encuentran a una distancia, respecto al punto actual del camino, igual o menor a la longitud del robot MASAR entre dos, pues estamos comprobando una distancia igual a la mitad de la estrechez

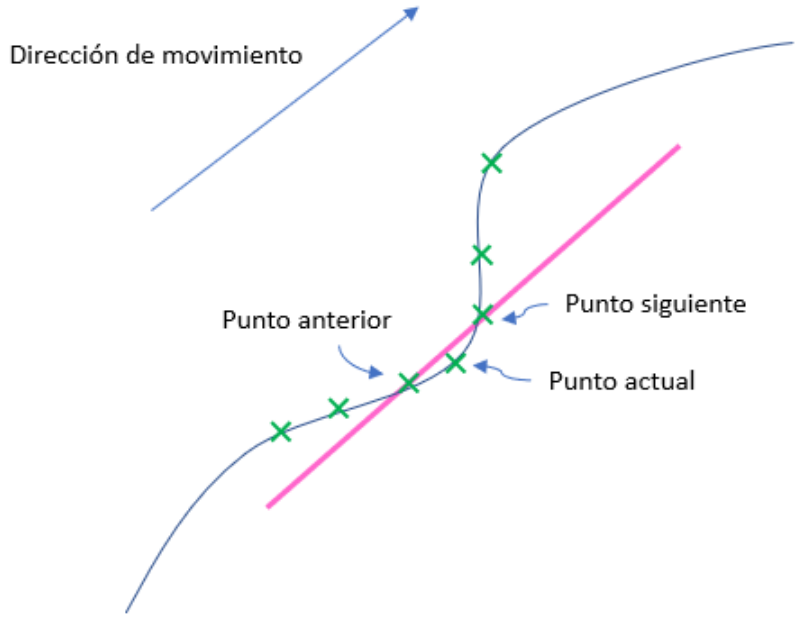

FIGURA 4.26: Recta tangente entre punto anterior y punto siguiente

(desde el camino hasta un punto de obstáculo) (*anchura<=limite/2*). De cumplirse esta condición, se comprueba si el punto se encuentra a un lado o a otro de la recta, sustituyendo dicho punto en la ecuación. En cada caso, se pone como "true"la variable correspondiente (*lado_positivo*) o (*lado_negativo*).

```
for j=1:length(obstaculos) %Para cada punto del camino, ...
    calculamos distancia a objetos

        obstaculo=[obstaculos(j,1) obstaculos(j,2)];
        anchura = norm(punto -obstaculo);

        %Se comprueba la distancia desde el punto actual al ...
           punto objeto seleccionado
        if anchura <= limite/2
            %Identificamos si se encuentra un lado de la ...
               recta u a otro
            if (A*obstaculo(1)+B*obstaculo(2)+C) > 0
                lado_positivo = true;
                %Ante la posibilidad de encontrar varios ...
                   puntos objeto que
                %cumplan la condicion, nos quedamos con el ...
                   mas cercano al
                %punto actual del camino
                if anchura<d_min_positivo
                    d_min_positivo = anchura;
                end
            elseif (A*obstaculo(1)+B*obstaculo(2)+C) < 0
                lado_negativo = true;
```

```
19                  if anchura<d_min_negativo
20                     d_min_negativo = anchura;
21                  end
22              end
23          end
24      end
```

Además, ante la posibilidad de que existan varios puntos que cumplan la condición de distancia, nos quedaremos con aquel cuya distancia sea menor, asegurando la estrechez menos ancha posible (*d_min_positivo*) y (*d_min_negativo*). Ambas inicializadas anteriormente a un valor alto.

El objetivo es encontrar dos puntos que se encuentren a una distancia menor a la longitud del robot, uno a un lado de la recta, y otro al otro lado. Esta es la condición esencial para etiquetar dicho punto como estrechez.

La información acerca de las estrecheces encontradas se almacena de la siguiente forma:

- *cont_estrechez*: vector de tamaño igual al vector de caminos (vector columna) donde se marca con un 1 el punto del camino (celda) que contiene estrechez
- Matriz *Estrechez*: matriz que almacena las puntos cartesianos del camino que conforman la estrechez, así como la anchura de la estrechez de cada punto (Figura 4.27). Se formarán grupos de 3 columnas por tramo estrecho (tanto grupos como tramos se identifiquen), para así distinguir los diferentes tramos estrechos que posee el camino. Para comprobar el final de un tramo estrecho, y por tanto crear 3 nuevas columnas en la matriz para almacenar el siguiente tramo, se detectará un cambio de 1 a 0 en el vector *cont_estrechez*.

```
1  if lado_positivo && lado_negativo
2      estrechez(cont,col)=punto(1);
3      estrechez(cont,col+1)=punto(2);
4      estrechez(cont,col+3)=2*min(d_min_positivo,d_min_negativo);
5      cont_estrechez(i,1)=1;
6      cont=cont+1;
7      hold on
8      plot(punto(1),punto(2),'*');
9  end
10
11
12
13 if i>1 && cont_estrechez(i-1,1) == 1 && cont_estrechez(i,1) ...
       == 0 %Separamos los tramos estrechos en columnas
14     col = col+3;
15     cont = 1;
16 end
```

Formadas todas las variables de información requeridas, se procede a realizar un filtrado de los puntos estrechos obtenidos con el fin de eliminar puntos estrechos sueltos, pues el objetivo es encontrar caminos estrechos, lo cual

	Tramo 1			Tramo 2			...
	X	Y	Anchura	X	Y	Anchura	...
1							...
2							...
3							...
4							...
5							...
6							...
.							...
.							...
.							...

FIGURA 4.27: Estructura de la tabla de estrecheces

implica un mínimo de 2 puntos estrechos consecutivos. Por un lado, se revisará si en el vecto *cont_estrechez* encontramos algún punto estrecho independiente (cadena 0, 1, 0). Y por otro lado, distinguir si alguno de los grupos de tramos de la matriz de información *estrechez* tiene únicamente una fila.

```
1  %Filtrado de estrecheces, eliminar puntos sueltos del vector ...
       de informacion
2  matriz_tam=size(estrechez);
3  n_tramos=matriz_tam(1,2)/3;
4
5  if matriz_tam(1,1)>1
6      k=1;
7      while k≤n_tramos
8          if estrechez(2,k*3) == 0
9              estrechez(:,(k*3)-2:k*3)=[];
10             n_tramos = n_tramos-1;
11             k=k-1;
12         end
13         k=k+1;
14     end
15 else
16     estrechez=[];
17 end
18
19 %Eliminar tramos de la matriz que solo tiene una fila
20 if cont_estrechez(1) == 1 && cont_estrechez(2) == 0
21     cont_estrechez(1) = 0
22 end
23 if cont_estrechez(end) == 1 && cont_estrechez(end-1) == 0
24     cont_estrechez(end) = 0
25 end
26 for f=2:length(cont_estrechez)-1
27     if cont_estrechez(f-1) == 0 && cont_estrechez(f) == 1 && ...
           cont_estrechez(f+1) == 0
28         cont_estrechez(f)=0;
29     end
30 end
31
```

```
32  %Ploteo de puntos filtrados
33  for t=1:length(camino)
34      if cont_estrechez(t)==1
35          hold on
36          plot(camino(t,1),camino(t,2),'*');
37      end
38  end
```

Se muestra un ejemplo de las estrecheces identificadas en un mapa de Voronoi (Figura 4.28).

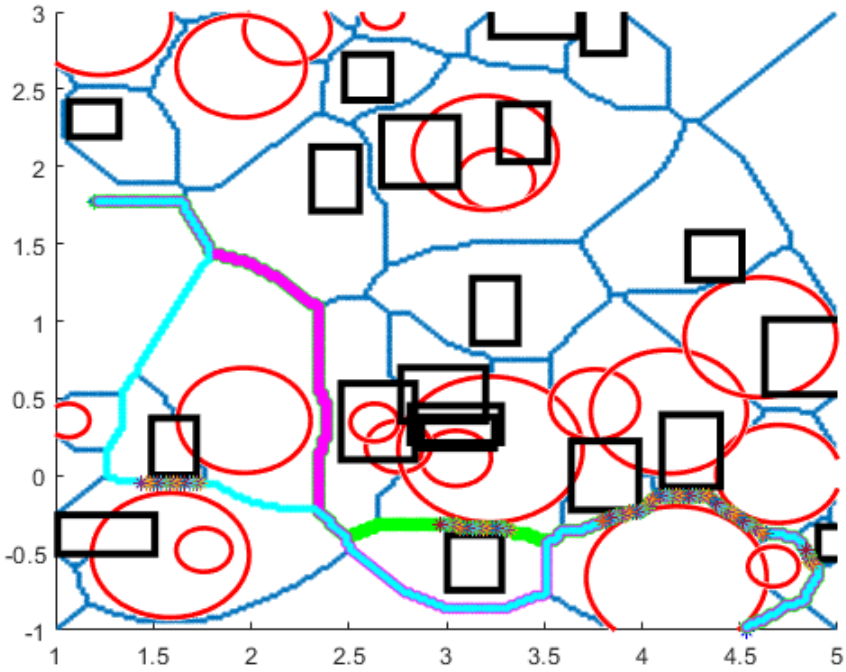

FIGURA 4.28: Estrecheces detectadas en los caminos resultantes de Voronoi y A*

Y finalmente, un ejemplo de las estrecheces identificadas en un mapa de Grafo de Visibilidad (Figura 4.29).

Es necesario remarcar que, la naturaleza poligonal de los caminos generados por el Grafo de Visibilidad con cambios bruscos de dirección, puede llevar a error en la detección de las estrecheces. Esto se debe al empleo de una recta tangente a la trayectoria para identificar los puntos estrechos, pues se puede dar el caso de que esta recta no represente fielmente al tramo de camino sometido a estudio, debido a un cambio brusco de pendiente (Figura 4.30). Estos casos son más probables cuando un camino pasa por un vértice de obstáculo, identificando estrecheces donde no las hay (Figura 4.30).

Esto se podría corregir suavizando la trayectoria del Grafo de Visibilidad, aunque esto se plantea como futuro trabajo a desarrollar.

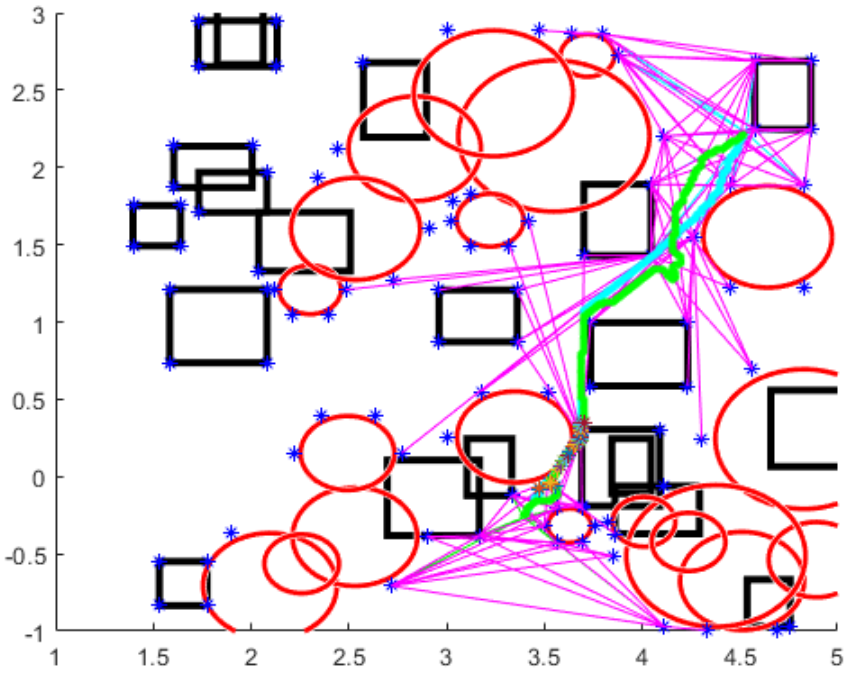

FIGURA 4.29: Estrecheces detectadas en los caminos resultantes de Grafo de Visibilidad y A*

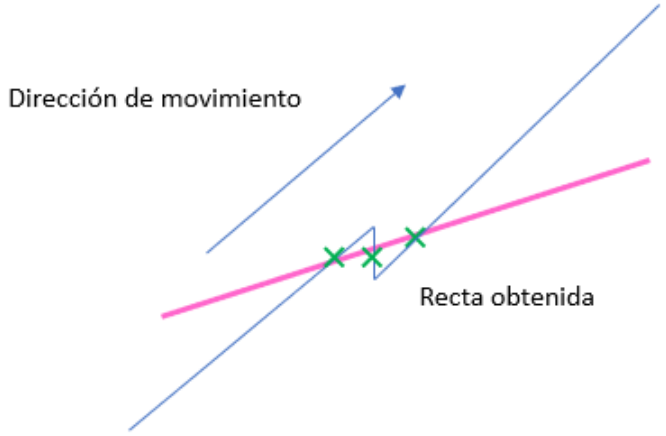

FIGURA 4.30: Recta tangente obtenida de pendiente diferente al camino

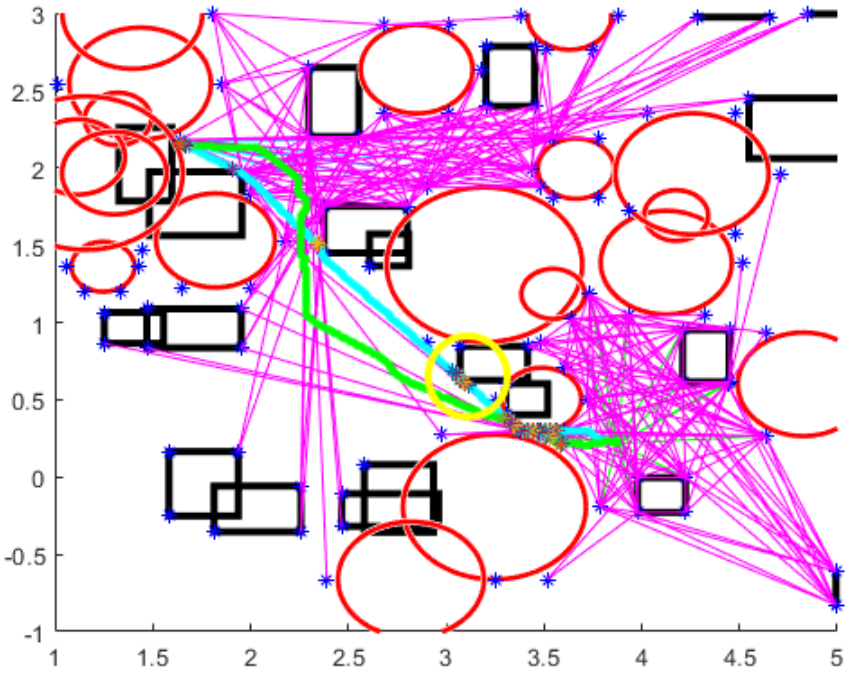

FIGURA 4.31: Estrechez errónea marcada con la circunferencia amarilla

4.5. Estimación de tiempos de recorrido

El último apartado de este capítulo busca finalizar el estudio de los métodos de planificación de trayectorias empleados, estimando un tiempo aproximado de recorrido del robot para cada camino.

Partimos desde dos elementos fundamentales: el camino a recorrer y la información acerca de los tramos estrechos del mismo. En función de ello, se propone una simulación del camino, distinguiendo los tramos libres y los tramos estrechos, que se simularán de forma separada de la siguiente forma:

- Tramos libres: simulación con control en posición con movimientos de 180º
- Tramos estrechos: estimación matemática del tiempo

El procedimiento general es recorrer el camino y haciendo uso del vector *cont_estrechez*, obtenido del apartado anterior (que marca con 1 las celdas del camino correspondientes con estrechez), se identificarán los cambios de 0 a 1 y de 1 a 0, para detectar un cambio de tramo y lanzar la simulación correspondiente. La función *coste_camino* será la encargada de este proceso.

El cálculo de tiempo requerido para recorrer un camino se estima a partir de unas variables de movimiento determinadas para el robot, estas son:

- Velocidad de giro

- Tiempo de pegue/despegue de 0.5 segundos

4.5.1. Simulación de tramos libres

El propósito de esta simulación es identificar aquellos tramos del camino a estudiar que no ostentan puntos estrechos, para posteriormente introducirlos como camino de referencia en el esquema Simulink de *Control en posición* introducido a modo de resumen en el apartado 3.2.2. De este, se enviará como variable de salida el tiempo total que ha tardado el robot en recorrer dicho tramo.

Identificación de tramos libres

Recorriendo todos los puntos del camino sometido a estudio mediante un bucle for, y tomando el punto inicial del tramo como el primer punto del camino, se tratará de identificar el paso de un tramo libre a un tramo estrecho. La variable inicio es aquella que indica el punto inicial del tramo a simular. Si dicho punto de inicio se comprueba en el vector de información de estrecheces *cont_estrechez* y se corresponde con un 0, estamos ante el inicio del un camino libre.

```
1 for i=2:length(camino)
2     if cont_estrecheces(inicio) == 0
```

Luego, se continúa recorriendo el camino hasta detectar un punto estrecho (*cont_estrechez = 1*) o bien, el final del camino. Identificado este final del camino libre, se tomarán dichos puntos del camino que lo conforman y se extraen en un tramo independiente (*tramo_sin*).

```
1  %Buscamos el siguiente 1 o final del camino
2  if cont_estrecheces(i) == 1 || i == length(camino)
3
4      if cont_estrecheces(i) == 1
5          %Calculo coste de camino sin restricciones
6          tramo_sin = camino(inicio:(i-1),:);
7          time = (0:60/(length(tramo_sin)-1):60);
8          tramo_sin_x = [time',tramo_sin(:,1)];
9          tramo_sin_y = [time',tramo_sin(:,2)];
10     end
11
12     if i == length(camino)
13         tramo_sin = camino(inicio:(i),:);
14         time = (0:60/(length(tramo_sin)-1):60);
15         tramo_sin_x = [time',tramo_sin(:,1)];
16         tramo_sin_y = [time',tramo_sin(:,2)];
17     end
18
19     final = tramo_sin(end,:);
```

A partir de este tramo extraído, se forman los vectores de simulación (*tramo_sin_x* y *tramo_sin_y*), empleando un tiempo de recorrido del camino de

60 segundos. Estos vectores de simulación junto con el punto final del tramo serán las variables necesarias a suministrar al esquema de Simulink.

Ejecución de la simulación

Recordando el modo de control comentado en el apartado 3.2.2, tenemos un controlador en posición que se diseñó para ejecutar una locomoción basada de giros de 180º para el robot MASAR. A dicho esquema Simulink se le podía proporcionar, tanto un punto objetivo, como una trayectoria a recorrer. Se reciclará dicho esquema de control salvo ciertas modificaciones necesarias para realizar la estimación de tiempos. Estas modificaciones son 2.

La primera es el modelado del tiempo de pegue y despegue. Se propone introducir una pausa de tiempo igual al tiempo de pegue/despegue cada vez que se deba cambiar el valor de la variable binaria. Esto se consigue empleando un switch que controle el valor de la velocidad el robot. Cuando el valor de la variable binaria cambie, el interruptor pondrá a 0 la velocidad del robot durante un tiempo igual a tiempo de pegue/despegue. Pasado este tiempo, el interruptor devolverá al robot MASAR el valor de velocidad proporcionado por el controlador (Figura 4.32).

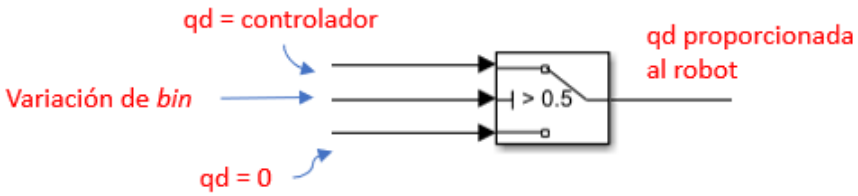

FIGURA 4.32: Modelado del tiempo de pegue/despegue

La segunda debe detener la simulación cuando el robot alcance el punto final del camino (Figura 4.33). Para ello, se evalúa la distancia entre el punto medio del robot y el fin del tramo, y si este valor de distancia es menor a un determinado valor, se detiene la simulación. En ese momento, se devolverá el valor de tiempo que ha necesitado el robot para llegar desde el inicio del tramo hasta el punto objetivo.

```
1  function stop  = fcn(final,C)
2
3  dist = norm(final -C);
4
5  if dist≤0.05
6      stop=1;
7  else
8      stop=0;
9  end
10 end
```

FIGURA 4.33: Final del tramo

Aparte del tiempo de recorrido, también se obtendrá la velocidad angular media del robot durante el recorrido, pues esta será de utilidad para la evaluación de tramos estrechos.

```
coste_sin = tiempo.signals.values(end);
velocidad = mean(qd_sin.signals.values);
```

4.5.2. Simulación de tramos estrechos

La segunda parte de este apartado trata los tramos estrechos del camino a estudiar. Inicialmente se identificarán los tramos estrechos, de forma similar a como se identificaron los tramos libres, y luego se realizará una estimación matemática del tiempo que requiere el robot para atravesar la estrechez, teniendo en cuenta la longitud de la misma y su anchura.

Identificación de tramos estrechos

Si anteriormente se debía identificar un paso de 0 a 1 en el vector de información de estrecheces *cont_estrecheces*, ahora se debe identificar un cambio de 1 a 0. Es necesario recordar que, cuando se identifica un tramo ya sea libre o estrecho, el punto siguiente será el punto *inicio* del siguiente tramo. Si este punto es estrecho, será cuando, para catalogar el tramo como estrecho, se deberá detectar un cambio de 1 a 0.

```
if cont_estrecheces(inicio) == 1
    if cont_estrecheces(i) == 0 || i == length(camino) ...
        %Inicio camino estrecho
```

Una vez reconocido el tramo estrecho, se accede a la matriz de información de estrecheces (*estrechez*) proporcionada en el apartado 4.4.2, donde se separó en grupos de columnas la información de cada tramo estrecho (Figura 4.27). Será necesario tener en cuenta si este tramo que se va a someter a estudio es el primero que encontramos en el camino o se encuentra en otro puesto, para acceder al grupo de columnas correspondiente. Como solución, se emplea un contador de tramos estrechos visitados (*cont_est*).

Dentro del tramo a analizar, se recorren sus filas hasta identificar un 0 en la columna que almacena la anchura, pues en esa fila termina la estrechez y la

matriz se completa con 0. De filas anteriores, se extrae la anchura mínima del tramo, que será al anchura que ostentará la estrechez a simular, y la longitud de la misma, que será la distancia entre el punto inicial y final.

```
anchura = estrechez(1,cont_est+2);
inicio_estrechez = ...
    [estrechez(1,cont_est),estrechez(1,cont_est+1)];
for j=2:size(estrechez,1)
    %Si hemos llegado a la zona nula de la matriz, paramos
    if estrechez(j,cont_est+2)==0
        break;
    end
    %Anchura
    if (estrechez(j,cont_est+2)<anchura) && ...
       (estrechez(j,cont_est+2)≠0)
        anchura = estrechez(j,cont_est+2);
    end
    %Longitud
    if (estrechez(j,cont_est))≠0 && (estrechez(j,cont_est+1))≠0
        final_estrechez = ...
            [estrechez(j,cont_est),estrechez(j,cont_est+1)];
    end
end

longitud = norm(final_estrechez −inicio_estrechez);
```

La anchura del camino y la longitud es todo lo que se necesita conocer para realizar la siguiente estimación matemática.

Estimación del tiempo para recorrer una estrechez

La estimación matemática consiste en, conocidos un grupo de parámetros físicos, estimar cuanto tiempo tardaría el robot en atravesar una estrechez. Estos parámetros físicos son los siguientes:

- Longitud de la estrechez
- Anchura de la estrechez
- Tamaño del robot MASAR
- Velocidad angular media del robot MASAR
- Número de pegues y despegues
- Magnitud angular total recorrida

El proceso a seguir es el siguiente, partiendo el robot MASAR desde una posición en la que tiene sus pivotes pegados a las paredes de la estrechez (Figura 4.34).

Conociendo la longitud del robot y la anchura de la estrechez, por el Teorema de Pitágoras se calcula la base del triángulo. Luego, a partir de la base

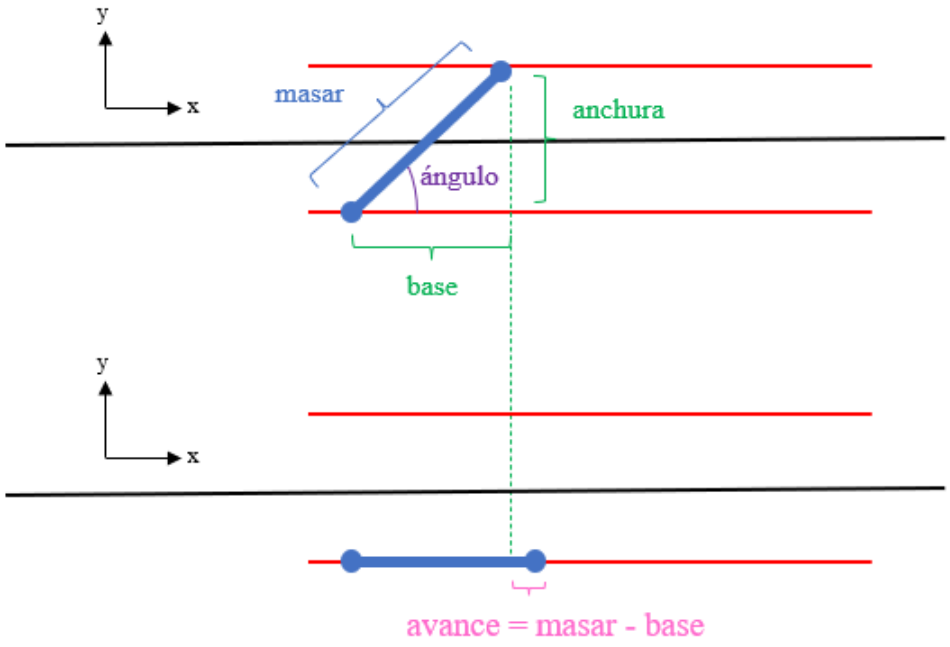

FIGURA 4.34: Parámetros de la estimación matemática

y la anchura, empleando el comando *atan2* se calcula el ángulo que forma el robot con la estrechez. El avance del robot cada vez que ejecuta un giro de la magnitud calculada, fijando su pivote trasero para posar su cuerpo de forma alternada sobre las paredes, será igual a la resta entre la longitud del robot y la distancia base. A partir de este avance calculado y de la longitud del robot, se puede calcular el número de avances a realizar para recorrer la longitud completa de la estrechez (Ecuación 4.20).

$$mun\_avances = round(longitud/avance) \tag{4.20}$$

Partiendo desde la posición en la que el robot está situado sobre una pared, un movimiento completo de avance tiene 2 fases:

1. Fijar el pivote delantero para llevar el pivote trasero a la pared opuesta, rotando el ángulo anteriormente calculado
2. Fijar el pivote trasero en la nueva pared para llevar el pivote delantero a dicha pared, rotando el ángulo anteriormente calculado

Por tanto, para ejecutar un movimiento de avance se requiere rotar 2 veces el ángulo que forma el robot con la estrechez y realizar 2 pegues/despegues.

```
1 pegadas = num_avances*2;
2 angulos = num_avances*angulo*2;
```

Estos valores se sustituyen en la función de coste para calcular el tiempo (4.21). La velocidad empleada en dicha función de coste es la velocidad media del anterior tramo libre, o una velocidad de 1 radián por segundo en caso de que el primer tramo sea estrecho.

$$coste\_con = pegadas * t\_pegada + angulos/velocidad \tag{4.21}$$

Calculado el coste del tramo estrecho, se adecúa el algoritmo dejándolo listo para continuar, avanzando 3 columnas en la matriz estrechez y nombrando el siguiente punto del camino como *inicio*.

4.5.3. Ejemplos de simulación

Calculados los costes de recorrer tramos libres y tramos estrechos, el coste total será la suma de los anteriores.

```
1 coste = coste_sin+coste_con;
```

En este apartado se muestran algunos ejemplos de costes obtenidos al recorrer diversos caminos. Por supuesto, estos caminos vendrán dados en función del empleo del Diagrama de Voronoi o Grafo de Visibilidad y A* con restricciones.

Todos los ejemplos se han obtenido con un mapa de 300x300 celdas, con un eje x que va desde 1 a 5, y un eje y que va desde -1 a 4. Además, se proponen dos caminos mediante ambos métodos:

- Diagrama de Voronoi + A*: camino verde sin máscara de penalización de obstáculos cercanos, y camino cian con una penalización de aquellos que se encuentren cerca del robot
- Grafo de Visibilidad + A*: al contrario, los caminos cian pasan lo más cerca posible de obstáculos, mientras que los verdes se alejan de ellos empleando la máscara penalizadora en A*

Ejemplo nº 1

El primer ejemplo trabaja sobre el siguiente mapa generado de forma aleatoria, con 20 obstáculos de tipo rectángulo, 20 obstáculos de tipo circunferencia y un tamaño del MASAR de 0.1 unidades de longitud (Figura 4.35).

Se obtienen, por un lado los caminos de Voronoi + A* (Figura 4.36), obteniendo un camino verde que atraviesa una estrechez y un camino cian más alejado de los obstáculos; y por otro los caminos de Grafo de Visibilidad + A* (Figura 4.37), con un camino cian que pasa lo más cerca posible de los obstáculos y otro verde que se aleja de ellos. Se puede observar que en el camino cian se obtienen algunos puntos erróneos de estrechez.

La principal diferencia en este ejemplo es que, la estrechez detectada en el Diagrama de Voronoi, no es detectada en el Grafo de Visibilidad puesto que el camino generado, justo en el tramo donde en Voronoi existe una estrechez, en Grafo de Visibilidad el camino pasa muy pegado a la circunferencia

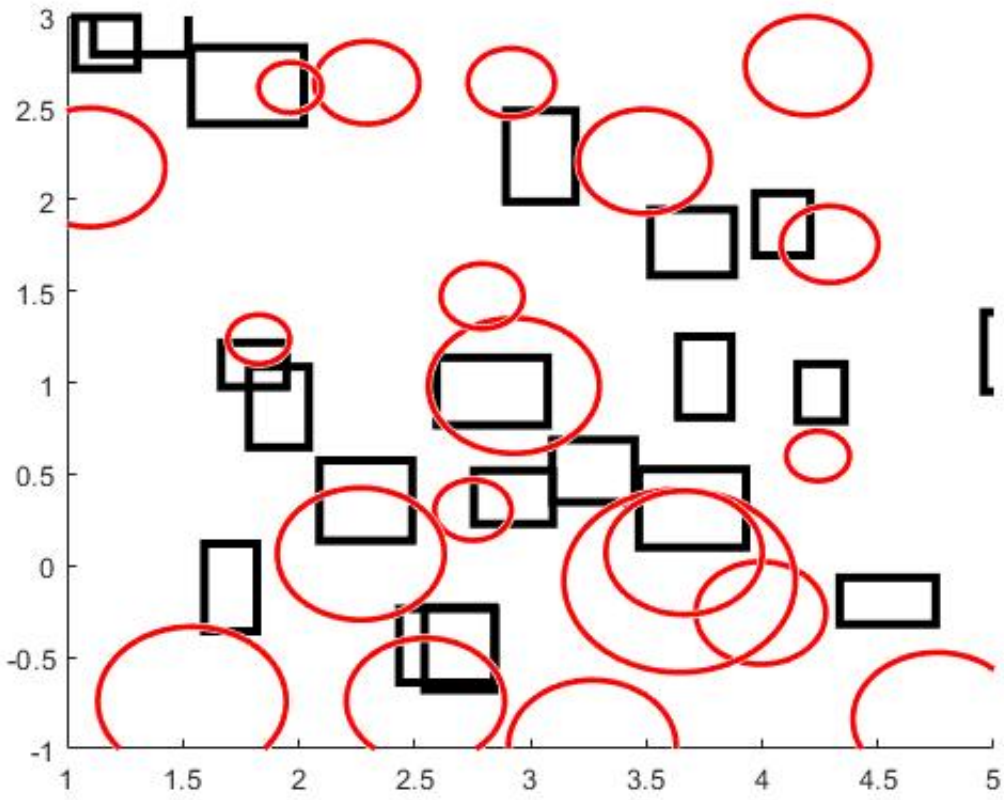

FIGURA 4.35: Mapa ejemplo nº 1

inferior (Figura 4.37). Comienza a quedar reflejada la ventaja del empleo de Grafo de Visibilidad a la hora de librarse de a travesar estrecheces
.

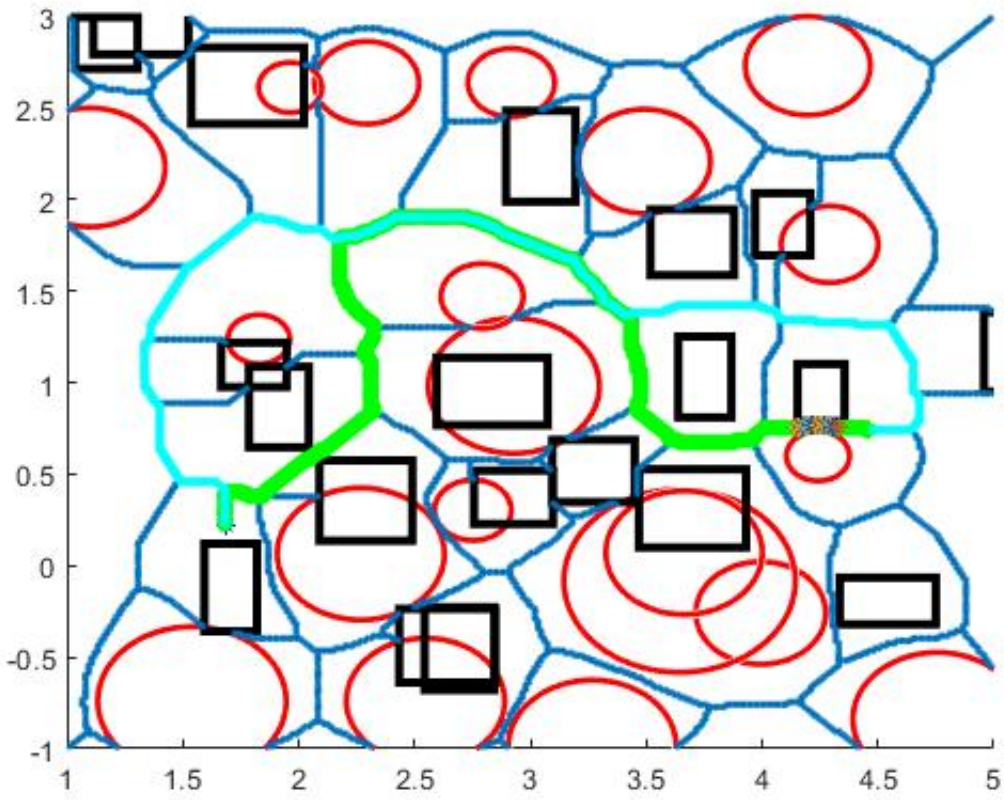

FIGURA 4.36: Caminos de Voronoi y A*, ejemplo 1

Los tiempos de recorrido para cada camino han sido los siguientes:

- Voronoi + A*, camino verde: tramos libres 78.5s + tramo estrecho 22.05s = 100,55s
- Voronoi + A*, camino cian: tramos libres = 79s
- Grafo + A*, camino cian: tramos libres 78,1s + tramos estrechos erróneos 30,5082s = 108.6082s
- Grafo + A*, camino verde: tramos libres = 81.09s

Para el ejemplo propuesto, las mejores soluciones son aquellas que se alejan de los obstáculos evitando así estrecheces, pues se puede apreciar el aumento del tiempo de recorrido en el camino verde de Voronoi, atravesando la estrechez.

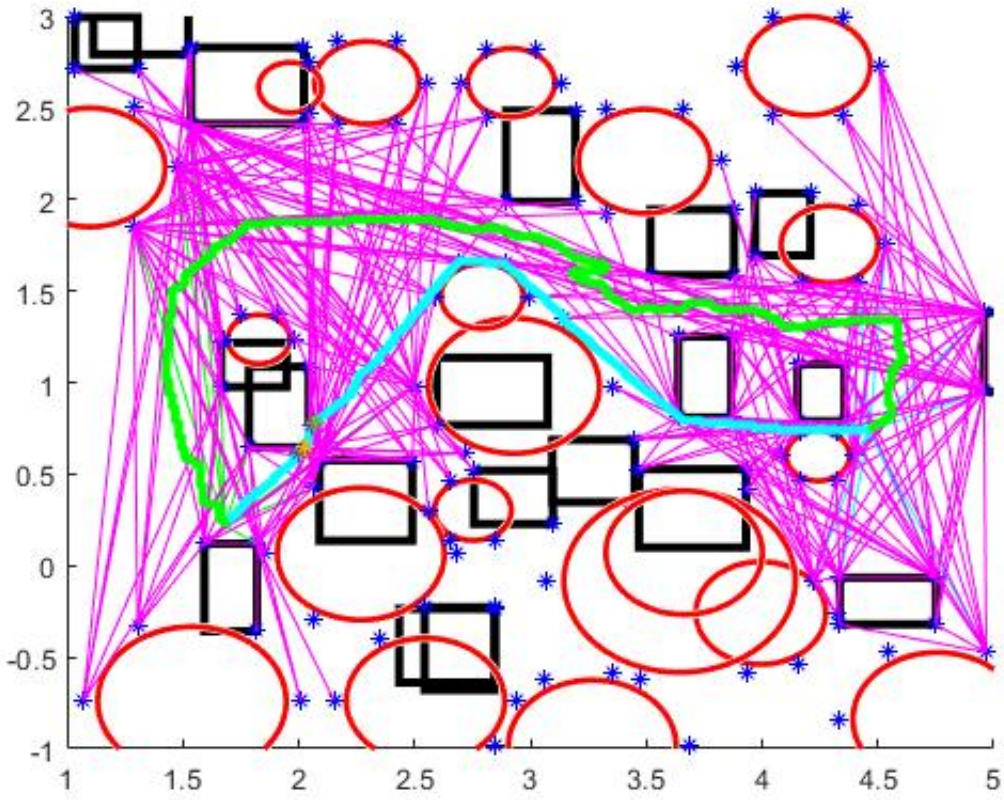

FIGURA 4.37: Caminos de Grafo de Visibilidad y A*, ejemplo 1

Ejemplo nº 2

En el ejemplo 2 se somete a análisis el siguiente mapa (Figura 4.38), con 30 obstáculos de cada tipo con un tamaño para el robot MASAR de 0.1 unidades de longitud.

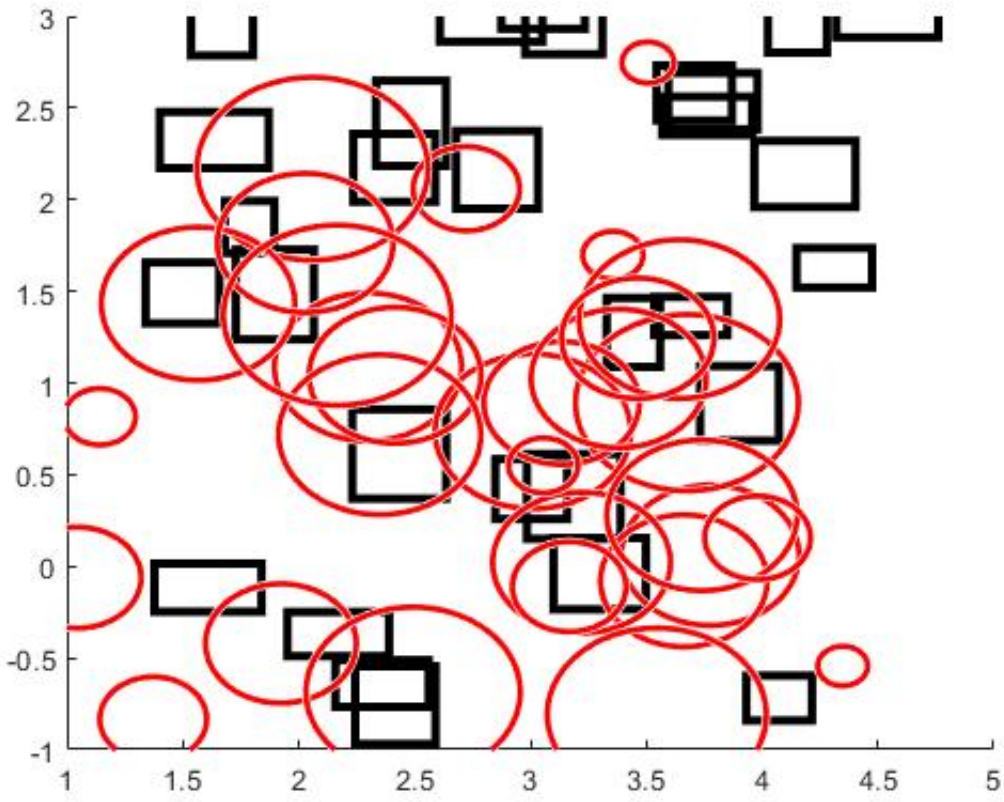

FIGURA 4.38: Mapa ejemplo nº 2

Se obtienen los siguiente caminos propuestos por Voronoi y A* (Figura 4.39) y los caminos de Grado de Visibilidad y A* (Figura 4.40).

Estamos ante una comparativa de métodos en caminos sin estrecheces, obteniendo los siguientes costes:

- Voronoi + A*, camino verde: tramos libres = 78.8s
- Voronoi + A*, camino cian: tramos libres = 79.81s
- Grafo + A*, camino cian: tramos libres = 77.27s
- Grafo + A*, camino verde: tramos libres = 81.5s

Ante la inexistencia de estrecheces, ambos métodos resultan ser muy similares en cuento a eficiencia de los caminos propuestos. Vemos que, debido

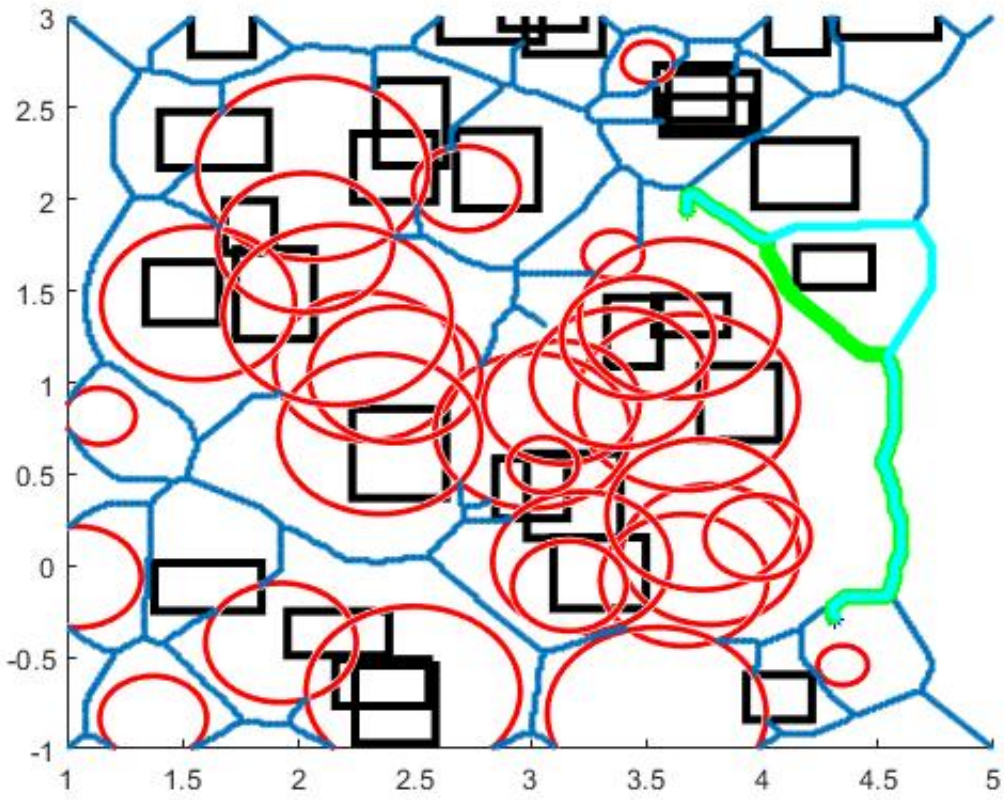

FIGURA 4.39: Caminos de Voronoi y A*, ejemplo 2

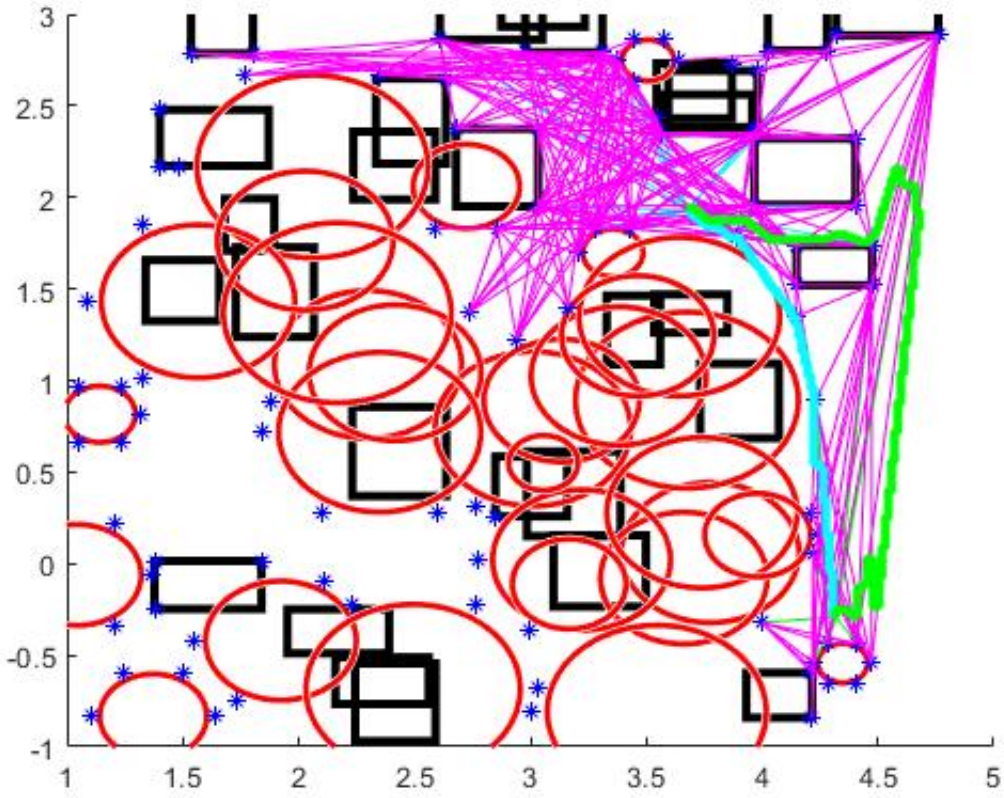

FIGURA 4.40: Caminos de Grafo y A*, ejemplo 2

a la naturaleza del Grafo de Visibilidad que pasa lo más cercano a los obstáculos, los tiempos de dicho método son ligeramente inferiores.

En base a las simulaciones propuestas, y en vistas a continuar experimentando con estos algoritmos aplicados al robot MASAR, podemos prever que los caminos sin estrecheces llegarán a ser más eficientes en la mayoría de los casos, en comparativa con aquellos que si presentan tramos estrechos. Encontramos el caso contrario cuando el camino propuesto para evitar estrecheces sea exageradamente extenso en comparación con el estrecho.

Capítulo 5

Conclusión y trabajos futuros

Este capítulo constituye la conclusión y posibles trabajos futuros derivados del desarrollo de este TFM. Como principal objetivo de este Trabajo de Final de Máster, se han centrado los esfuerzos en continuar y mejorar los desarrollos realizados en el TFG que precede a este trabajo y mejorar el desempeño del robot MASAR. A grandes rasgos, en primer lugar, se han desarrollado nuevos controladores para el robot MASAR, para conseguir un comportamiento más robusto y un control más completo del estado final. En segundo lugar, se han implementado dos algoritmos de planificación de trayectorias considerando las particularidades de dicho robot, en entornos previamente conocidos.

5.1. Resultados y conclusiones

Separando las conclusiones en los dos capítulos de desarrollo, se realizarán algunos comentarios sobre el capítulo 3, y luego sobre el capítulo 4.

En el capítulo 3 se ha propuesto un controlador en posición y orientación para el robot MASAR. Dicho control, gracias a la robustez que ofrece y a la posibilidad de conseguir una orientación determinada, mejora con creces sus controladores predecesores. Mediante los experimentos realizados, se ha demostrado una gran precisión a la hora de situar el robot en su pose final.

En el capítulo 4, a partir de la generación de un mapa geométrico con obstáculos situados en posiciones aleatorias, se implementan diversos algoritmos de generación de caminos, estos son el Diagrama de Voronoi y el Grafo de Visibilidad. Ambos métodos presentan soluciones muy diferentes, pues Voronoi maximiza la distancia a obstáculos y el Grafo Visibilidad pretende pasar lo más cerca posible de ellos, y a partir del empleo de un Algoritmo A* y del cálculo de coste de tiempo de recorrido, se estudia la viabilidad de cada método para el mapa dado.

Tal y como se ha descrito previamente, el apartado 4 ha dado lugar a un trabajo futuro, consistente en mejorar la detección de estrecheces en los caminos generados a partir del grafo de visibilidad. Esto posibilitará una comparación homogénea entre ambos métodos de planificación de trayectorias.

En cualquier caso, la implementación de los métodos y los diseños propios para adaptar las necesidades del robot MASAR a la planificación han resultado ser satisfactorios.

5.2. Trabajos futuros

Para finalizar con la memoria, se proponen algunas líneas de progreso tomando como punto de partida el presente trabajo.

Como trabajo a corto plazo, se debe profundizar y mejorar la planificación basada en un entorno conocido, probando algunos métodos extra de planificación de trayectorias, como por ejemplo el *"Probabilistic Roadmap Method"*, para así disponer de mayor variedad de estudio. Por supuesto, mejorar los ya implementados, reduciendo posibles errores y tiempos de cómputo. En concreto, solucionar el problema de detectar estrecheces de forma robusta y fiable en el Grafo Visibilidad, intentando suavizar la trayectoria para que la línea tangente en el punto actual (que separa obstáculos a un lado y a otro del camino) esté mejor definida.

Dichos experimentos se proponen para comenzar a finalizar el estudio de un módulo independiente del robot MASAR.

Luego, con el fin de continuar con la investigación, se comenzará a tratar la posibilidad de cambio de plano que ofrece el robot MASAR, así como la combinación de módulos del robot MASAR, analizando la cinemática de estos módulos y comenzando a plantear posibles métodos de control y planificación de trayectorias para este nuevo robot a partir de lo ya estudiado. Finalmente, se comenzará a estudiar la aplicación del robot MASAR a tareas reales. Por ejemplo, el hecho de combinar los módulos del robot MASAR puede permitirle el desplazamiento por trayectorias más complejas en 3D, como por ejemplo estructuras metálicas o celosías, con el fin de realizar determinadas labores.

Bibliografía

Birkmeyer, Paul, Andrew G Gillies y Ronald S Fearing (2011). «CLASH: Climbing vertical loose cloth». En: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, págs. 5087-5093.

Cheng, Nadia y col. (2010). «Design and analysis of a soft mobile robot composed of multiple thermally activated joints driven by a single actuator». En: *2010 IEEE International Conference on Robotics and Automation*. IEEE, págs. 5207-5212.

Dharmawan, Audelia G y col. (2017). «Steerable miniature legged robot driven by a single piezoelectric bending unimorph actuator». En: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, págs. 6008 -6013.

Hoover, Aaron M y col. (2010). «Bio-inspired design and dynamic maneuverability of a minimally actuated six-legged robot». En: *2010 3rd IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics*. IEEE, págs. 869-876.

IDELab. *http://idelab.uva.es/algoritmo*.

Moubarak, Paul y Pinhas Ben-Tzvi (2012). «Modular and reconfigurable mobile robotics». En: *Robotics and Autonomous Systems* 60.12, págs. 1648 -1663.

Project, Wolfram Demostration. *https://demonstrations.wolfram.com/AnEfficient TestForAPointToBeInAConvexPolygon/*.

Refael, Gilad y Amir Degani (2015). «Momentum-driven single-actuated swimming robot». En: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, págs. 2285-2290.

Romanishin J. W., Gilpin K. Claici S. Rus D. (2015). «3D M-Blocks: Self-reconfiguring robots capable of locomotion via pivoting in three dimensions». En: *Robotics and Automation (ICRA)*, págs. 925-1932.

Sfeir, Joy, Elie Shammas y Daniel Asmar (2014). «Design and modeling of a novel single-actuator differentially driven robot». En: *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. IEEE, págs. 1079 -1084.

Shen W. M., Salemi B. Will P. (2000). «Hormones for self-reconfigurable robots». En: *International Conference on Intelligent Autonomous Systems*.

Soyguder, Servet y Hasan Alli (2011). «Motion mechanism concept and morphology of a single actuator tetrapod walking spider robot: the ROBOTURK.» En: *Industrial Robot: An International Journal* 38.4, págs. 361 -371.

Yim Shen, Salemi Rus Moll Lipson (2007). «Modular self-reconfigurable robot systems (grand challenges of robotics». En: *Robotics Automation Magazine*, págs. 43-52.

Zambrana, David (2016). «Planificación de trayectorias y control visual de robots paralelos planos». Tesis doct. Elche (Spain): Trabajo Fin de Grado, Universidad Miguel Hernandez.

Zarrouk, David y Ronald S Fearing (2015). «Controlled in-plane locomotion of a hexapod using a single actuator». En: *IEEE Transactions on Robotics* 31.1, págs. 157-167.

Zhao, Jianguo y col. (2013). «MSU jumper: A single-motor-actuated miniature steerable jumping robot». En: *IEEE Transactions on Robotics* 29.3, págs. 602 -614.