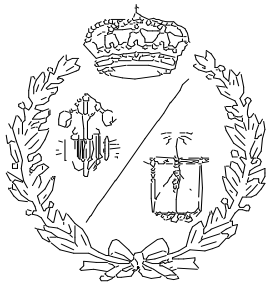
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Trabajo Fin de Grado

PLANIFICACIÓN DE RUTAS ÓPTIMAS DE ROBOTS MÓVILES

(Mobile Robots Optimal Path Planning)

Para acceder al Título de

GRADUADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

Autor: Rafael Ortiz Imedio

Julio – 2016

AGRADECIMIENTOS

En primer lugar, a Pedro Corcuera, por su valiosa ayuda, abundante información y cercanía proporcionada durante este Trabajo Fin de Grado.

A los compañeros de clase por su ánimo y apoyo a lo largo de estos años y alegrarme el día a día con su presencia.

A los profesores que he tenido durante la realización de este grado por formarme y compartir su experiencia conmigo dentro y fuera de las clases.

A mis amigos, que tantas veces me han hecho salir de la monotonía y haber podido vivir grandes momentos con ellos.

A mi Comunidad, por su apoyo y oraciones que en numerosas ocasiones he necesitado a lo largo de este tiempo.

A mi familia, por su inestimable ayuda y comprensión día a día, animándome cada vez que se me hacía cuesta arriba el grado.

A Dios, por todo lo que he recibido hasta ahora, incluyendo la finalización del grado, y por todo lo que esté por venir.

ÍNDICE

LISTA DE FIGURAS V

1 INTRODUCCIÓN 1

1.1 ROBOTS MÓVILES 1

1.2 CONTROL Y SENSORES 2

1.3 PLANIFICACIÓN DE CAMINOS 3

1.4 OBJETIVOS 4

2 ROBOTS MÓVILES 1

2.1 VEHÍCULOS CON RUEDAS 1

2.2 LOCOMOCIÓN MEDIANTE PATAS 4

2.3 CONFIGURACIONES ARTICULADAS 5

2.4 ROBOTS AÉREOS Y SUBMARINOS 6

2.5 APLICACIONES 8

3 REPRESENTACIÓN DE LA POSICIÓN Y ORIENTACIÓN 12

3.1 SISTEMAS DE COORDENADAS EN EL PLANO 12

3.2 SISTEMAS DE COORDENADAS EN EL ESPACIO 14

3.3 POSICIÓN Y ORIENTACIÓN 14

3.3.1 Posición y orientación en el plano 14

3.3.2 Posición y orientación en el espacio 17

4 MODELOS CINEMÁTICOS Y DINÁMICOS DE ROBOTS MÓVILES 20

4.1 CINEMÁTICA 20

4.1.1 Modelo cinemático directo 21

4.1.2 Restricciones cinemáticas de las ruedas 24

4.1.3 Vehículo de Ackerman 26

4.2 DINÁMICA 27

5 CONTROL DE ROBOTS MÓVILES 31

5.1 SEGUIMIENTO DE TRAYECTORIA 33

5.1.1 Control basado en linealización aproximada 36
6 SENSORES 41
6.1 POSICIÓN Y ORIENTACIÓN 42
6.1.1 Odometría 43
6.1.2 Efecto Doppler 44
6.1.3 Navegación inercial 45
6.1.4 Estaciones de transmisión 52
6.1.5 Percepción del entorno 55
7 PLANIFICACIÓN DE RUTAS Y MOVIMIENTO 58
7.1 ESPACIO DE CONFIGURACIONES 58
7.2 ALGORITMOS DE PLANIFICACIÓN 60
7.2.1 Basados en geometría 60
7.2.2 Basados en grafos 63
7.2.3 Basados en cuadrículas 70
7.2.4 Métodos probabilísticos 81
7.2.5 Funciones potenciales 89
8 LOCALIZACIÓN Y GENERACIÓN DE MAPAS 95
8.1 FILTRO DE KALMAN 95
8.1.1 Estimación probabilística 95
8.1.2 Filtro de Kalman lineal 96
8.2 LOCALIZACIÓN CON MARKOV 97
8.3 MÉTODO DE MONTE CARLO 98
8.4 GENERACIÓN DE MAPAS 100
9 SIMULACIÓN 104
9.1 SEGUIMIENTO DE TRAYECTORIA 104
9.2 VORONOI 105
9.3 DIJKSTRA EN CUADRÍCULA 107
9.4 A* 108

9.5 PRM....112
9.6 RRT....115
9.7 FUNCIONES POTENCIALES....116
10 PRESUPUESTO....120
11 CONCLUSIONES....121
12 ANEXOS....123
12.1 VORONOI....123
12.2 DIJKSTRA Y A* EN CUADRÍCULA....125
12.3 A* [69]....131
12.4 PRM [69]....134
12.5 RRT [69]....135
12.6 CAMPOS POTENCIALES [69]....137
13 REFERENCIAS....141

LISTA DE FIGURAS

Figura 1: Ballbot 2

Figura 2: Ruedas síncronas 3

Figura 3: Robots con ruedas 4

Figura 4: Robots con patas 5

Figura 5: Configuraciones articuladas 6

Figura 6: Vehículos aéreos 7

Figura 7: Vehículos submarinos 8

Figura 8: Robots limpieza 9

Figura 9: Robots de transporte 9

Figura 10: Aplicaciones de sanidad y exploración 10

Figura 11: Robots militares 10

Figura 12: Sistemas de coordenadas cartesianas [12, p. 15] 12

Figura 13: Punto P descrito con varios sistemas de coordenadas 13

Figura 14: Múltiples sistemas de coordenadas 3D y poses relativas [12, p. 17] 14

Figura 15: Relación entre sistemas de coordenadas en 2D [12, p. 20] 15

Figura 16: Sistema de coordenadas rotado en 2D [12, p. 21] 15

Figura 17: Relación entre sistemas de coordenadas en 3D [12, p. 25] 17

Figura 18: a) Sistemas de referencia global y local b) Alineado con un eje global .. 22

Figura 19: Robot móvil diferencial en sist. de referencia global [2, p. 51] 22

Figura 20: Rueda de castor; parámetros relativos a referencia móvil [2, p. 57] 25

Figura 21: Modelo cinemático 27

Figura 22: Robot móvil no holonómico [24, p. 531] 28

Figura 23: Esquema general del sistema de control de un robot móvil [1, p. 13] 32

Figura 24: Control de velocidad y desplazamiento lateral. 32

Figura 25: Coordenadas generalizadas para un monociclo [8, p. 478] 34

Figura 26: Error de posición ep en seguimiento de trayectoria [8, p. 503] 35

Figura 27: Seguimiento de trayectoria circular 38

Figura 28: Esquema Simulink: seguimiento de trayectorias con control lineal y "escalado de velocidad" [6, p. 280] 39

Figura 29: Velocidades lineal y angular ("escalado de velocidad") [6, p. 281] 39

Figura 30: Resultados posiciones y orientaciones 40

Figura 31: Encoders..... 44

Figura 32: Efecto Doppler [28, p. 47] 45

Figura 33: Acelerómetro ADXL335 47

Figura 34: Giróscopos..... 48

Figura 35: Giróscopos MEMS 49

Figura 36: MEMS magnetómetro 51

Figura 37: Triangulación LORAN [1, p. 36] 53

Figura 38: Sistema GPS tiene tres partes: Espacio, Control y Usuario [1, p. 39]..... 55

Figura 39: Espacio de configuraciones 59

Figura 40: Diagrama de Voronoi 61

Figura 41: Voronoi 62

Figura 42: Grafos..... 63

Figura 43: Búsqueda en grafo..... 66

Figura 44: Grafo de visibilidad 67

Figura 45: Descomposición en celdas y canal [8, p. 537]..... 68

Figura 46: Descomposición exacta 69

Figura 47: Descomposición aproximada [8, p. 540]..... 69

Figura 48: Grassfire [50] 70

Figura 49: Dijkstra [23, p. 532] 72

Figura 50: A*..... 75

Figura 51: D* [57, p. 58] 77
Figura 52: D* Lite [57, p. 131] 79
Figura 53: Comparación A* y ARA* 80
Figura 54: Comparación ARA* y AD* 81
Figura 55: PRM 83
Figura 56: EST 85
Figura 57: RRT 86
Figura 58: SRT [23, p. 239] 88
Figura 59: Trampas en métodos probabilísticos [45, p. 219] 89
Figura 60: Funciones potenciales. Suma de atractiva más repulsiva [38, p. 46] 91
Figura 61: Brushfire [23, p. 88] 92
Figura 62: Problema mínimo local 92
Figura 63: Planif. onda frontal (inicio en esq. sup. dcha y meta en 2) [23, p. 92] 94
Figura 64: Filtro de Partículas (10000 muestras) 100
Figura 65: Secuencia mapas con filtro partículas Rao-Blackwellized [23, p. 344] . 101
Figura 66: Mapa inconsistente con SLAM en ciclo cerrado [31, p. 21] 102
Figura 67: Mapa de cuadrícula obtenido con SLAM [31, p. 14] 103
Figura 68: Modelo Simulink persecución pura [12, p. 74] 104
Figura 69: Resultados persecución pura 105
Figura 70: Simulación 1 Voronoi 106
Figura 71: Simulación 2 Voronoi 107
Figura 72: Simulación Dijkstra en cuadrícula 108
Figura 73: Simulación A* en cuadrícula 109
Figura 74: Simulación 1 A* con [69] 111
Figura 75: Simulación 2 A* con [69] 111
Figura 76: Simulación 3 A* con [69] 112

Figura 77: Simulación 1 PRM 113

Figura 78: Simulación 2 PRM 114

Figura 79: Simulación 3 PRM 114

Figura 80: Simulación 1 RRT 116

Figura 81: Simulación 2 RRT 116

Figura 82: Simulación 1 Funciones Potenciales 117

Figura 83: Simulación 2 Funciones Potenciales 117

Figura 84: Simulación 3 Funciones Potenciales 118

Figura 85: Simulación 4 Funciones Potenciales 118

Figura 86: Simulación 5 Funciones Potenciales 119

1 INTRODUCCIÓN

1.1 ROBOTS MÓVILES

La robótica es uno de los campos multidisciplinares presentes en la ingeniería que más ha crecido en los últimos años. Aúna conceptos de mecánica, electrónica, automática, computación, inteligencia artificial… Esto se debe a la inmensa variedad de aplicaciones que puede tener, desde manipulación y manufactura hasta navegación. Además, es una fuente de entretenimiento para el ocio ya que el abaratamiento del hardware y el fácil acceso a librerías de "código abierto" permite construir robots simples de una manera sencilla por cualquier persona. Sin embargo, pueden alcanzar una complejidad muy elevada según la funcionalidad y características deseadas, proporcionando un campo de investigación muy creativo e instructivo para los interesados en este mundo de la ingeniería.

Dentro de la robótica en general, un campo destacado es la robótica móvil, que es aquella que permite el movimiento de su base, a diferencia de los robots manipuladores que tienen la base fija, permitiendo una mayor autonomía y una menor intervención humana. Sin embargo, los robots móviles trabajan en ambientes no estructurados con grandes incertidumbres en la posición e identificación de los objetos, mientras que los manipuladores operan en ambientes estáticos, estructurados y en gran parte conocidos. Esto provoca una menor precisión en las trayectorias con respecto a los manipuladores.

Diremos que un vehículo es holonómico cuando el número de grados de libertad de su movimiento es igual al número de grados de libertad globales. Por ejemplo: un tren es holonómico ya que solo puede moverse en un grado de libertad: hacia delante y hacia detrás. Un coche es no-holonómico ya que posee tres grados de libertad globales: "*x", "y"* y "θ" la orientación, pero sin embargo no puede desplazarse lateralmente, debe avanzar para poder girar. Sin embargo, un robot con ruedas "Mecanum" o "suecas" es holonómico ya que puede rotar sobre sí mismo y desplazarse lateralmente, como veremos en el capítulo 2 (Figura 3a).

Dentro del campo de la robótica móvil podemos encontrar varios tipos de configuraciones que en el siguiente capítulo trataremos con más detalle:

- Vehículos con ruedas, los cuales utilizan una serie de motores para mover dos o más ruedas según su morfología.
- Locomoción mediante patas, pudiendo adoptar formas de insectos, mamíferos, humanoides…
- Configuraciones articuladas, que según su configuración y algoritmos tendrán un tipo de movimiento determinado.
- Robots submarinos y aéreos, los cuales poseen seis grados de libertad.

También veremos la gran cantidad de aplicaciones posibles que hay para los robots móviles, desde limpieza hasta militar. A medida que se van creando y estudiando nuevos robots, más aplicaciones podremos encontrar para facilitar y hacer más segura la vida de las personas.

1.2 CONTROL Y SENSORES

En el capítulo 5 describiremos el esquema general de control de un robot móvil con sus tres jerarquías y la función de cada una. Además, podemos encontrarnos distintas estrategias de control según si es capaz de planificar las acciones a ejecutar para llevar a cabo la tarea asignada creando un modelo del entorno, o si es capaz de reaccionar ante los cambios inesperados en el entorno de forma rápida.

Nosotros nos centraremos en el control necesario para el seguimiento de trayectorias en el plano. Para ello utilizaremos el error existente entre la posición actual y la deseada en la trayectoria, alcanzando una ley de la dinámica del error de seguimiento. Podremos hacer una linealización del control resultante introduciendo una ley de control lineal propuesta en la bibliografía, obteniendo las ganancias que tendremos que colocar en el controlador para reducir el error y así seguir dicha trayectoria.

Una parte imprescindible en los robots móviles, además del control para crear el movimiento deseado, son los sensores. Éstos permiten obtener información del estado del robot y del entorno para poder actuar en consecuencia. Podemos realizar una clasificación fundamental según si son propioceptivos, obteniendo información interna del robot como temperatura, batería, número de vueltas de los motores, etc. o exteroceptivos, que recogen información del entorno evitando que el robot móvil se choque con obstáculos, condiciones atmosféricas, sonido, etc.

En la robótica móvil la función principal de los sensores será obtener la posición y orientación (pose) del robot en el entorno respecto a un sistema de referencia absoluto para ser capaz de generar trayectorias y evitar obstáculos. Dentro de esta función podemos encontrar dos tipos de medición. La primera es la medición de posición relativa o navegación por estima, es decir, mediante sensores montados sobre el propio robot calculando su pose, velocidad y aceleración. Para ello recurriremos a odometría, efecto Doppler y navegación inercial. La segunda es la medición de posición absoluta mediante estaciones de transmisión (balizas externas) fijas o móviles de posiciones conocidas. A partir de ellas podremos calcular la localización absoluta en entornos muy diversos y pudiendo recorrer grandes distancias.

1.3 PLANIFICACIÓN DE CAMINOS

Una de las aplicaciones más importantes de los robots móviles es la capacidad de ir de un punto a otro de forma segura y factible sin colisionar con obstáculos. Además, se suele desear realizarlo de forma óptima en un tiempo, distancia o coste mínimos. El mapa puede ser creado externamente e introducido en el robot móvil o generado por el propio robot a medida que se desplaza por el entorno y se localiza en él utilizando sus sensores, llamándose a este proceso SLAM. Mientras que solo describiremos un poco el segundo método en el capítulo 8, trataremos con más detalle el primer caso, es decir, planificando el camino a partir de mapas introducidos en el robot móvil, en el capítulo 7.

Para este objetivo se han desarrollado multitud de algoritmos diversos de planificación de caminos. En este trabajo los clasificaremos y explicaremos según la representación o método utilizado en el mapa: geometría, grafos, cuadrículas, métodos probabilísticos y funciones potenciales. Una herramienta importante para la representación del entorno será el espacio de configuraciones. Con ella transformaremos los obstáculos y los límites del espacio presentes en el entorno del robot para poder representarlo como un punto y a la vez se tenga en cuenta su geometría. Por tanto, el problema se reducirá a encontrar el camino desde la posición inicial a la final para un punto de referencia del móvil sin preocuparnos del riesgo de colisiones a causa de la geometría del robot.

1.4 OBJETIVOS

Este trabajo podemos dividirlo en dos partes principales.

La primera parte tratará de dar una visión general de los robots móviles exponiendo los tipos básicos que hay en el mercado o en investigación. Después trataremos el modelo cinemático y dinámico de algún robot móvil con ruedas, ya que son los más usuales y sencillos. Además, estudiaremos el control lineal de un robot monociclo para que ejecute un seguimiento de una trayectoria de referencia. Por último, describiremos los principales sensores utilizados por un robot para desplazarse por su entorno.

La segunda parte tiene como meta realizar un estado del arte de muchos de los algoritmos de planificación de rutas que han sido desarrollados hasta ahora. Mediante ellos, seremos capaces de ir de un punto a otro por el camino óptimo y evitando los obstáculos que haya en un mapa previamente creado y proporcionado al robot móvil.

También presentaremos métodos para estimar la localización del robot cuando se desconoce, como el filtro de Kalman o el de partículas. Asimismo, hablaremos un poco sobre un método para construir un mapa del entorno y localizarse en él simultáneamente cuando no lo conocemos de primera mano.

Para finalizar, haremos unas simulaciones de algunos algoritmos mediante el software Matlab.

2 ROBOTS MÓVILES

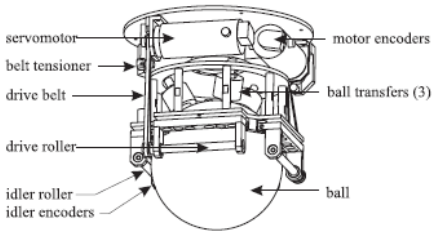
Como hemos introducido en el primer capítulo, los robots móviles deben trabajar generalmente en ambientes desconocidos y con grandes incertidumbres en la posición e identificación de los objetos. Los robots industriales son efectivos con una información sensorial mínima y un "raciocinio" básico debido a que operan en ambientes estáticos, estructurados y relativamente conocidos. Por el contrario, los robots móviles generan trayectorias y guían su movimiento basándose en la información procedente de los sensores incorporados. Además, requieren un alto nivel de "razonamiento" para decidir en cada momento las acciones requeridas para alcanzar el objetivo según el estado del robot y del entorno. Esto se traduce en planificar trayectorias globales seguras y ser capaz de modificarlas en presencia de obstáculos inesperados mediante un control local de trayectoria [1, pp. 11-12].

Ahora veremos los distintos tipos de robots móviles que podemos encontrar basándonos en su configuración y medio de locomoción.

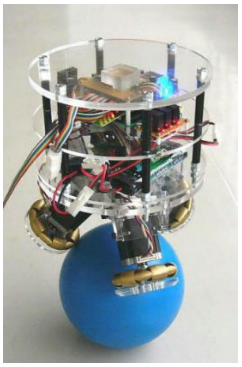
2.1 VEHÍCULOS CON RUEDAS

Son los robots que proporcionan una solución más simple y eficiente para moverse en terrenos suficientemente duros, alcanzando velocidades relativamente altas, aunque también hay mayor riesgo de deslizamiento. Casi siempre son diseñados para que tengan las ruedas en contacto con el suelo. Esto hace que cuando tienen más de tres ruedas, incorporen un sistema de suspensión para garantizarlo [2, p. 32]. En esta categoría podemos hacer una distinción según el número de ruedas que posea el robot.

Cuando solo tiene una bola esférica, se llama "mono bola" o "ballbot" [3, p. 7]. El robot se equilibra dinámicamente, ya que no está en equilibrio estático, haciendo rodar la bola y provocando que la base se desplace en la dirección deseada (sin restricciones al ser esférica). Dos proyectos de este tipo son: el desarrollado por la universidad Carnegie Mellon [4] (Figura 1a), el cual posee cuatro rodillos giratorios en el perímetro de la bola (como en los ratones de ordenador antiguos); y el creado por la universidad de Tohoku Gakuin (Figura 1b), que utiliza tres ruedas omnidireccionales sobre la bola para mantener el equilibrio. Utilizan giroscopios y pueden reaccionar ante perturbaciones externas.



a)



b)

Figura 1: Ballbot
a) Carniege Mellon b) Tohoku Gakuin
a) [4] b) (http://data.mecheng.adelaide.edu.au/robotics/projects/2009/Ballbot/TGU2.bmp)

En el caso de dos ruedas, éstas serán coaxiales y una a cada lado del cuerpo, lo que provoca la necesidad de un equilibrado dinámico debido a la falta de un tercer punto de apoyo (a no ser que el centro de masas esté debajo del eje de la ruedas, como en el robot Cye [2, p. 33]). Esto complica su controlabilidad ya que debe balancearse hacia delante y hacia atrás, pero una vez controlada su estabilidad, puede moverse por terrenos inclinados y con distinta rugosidad actuando autónomamente para compensarlo [3, p. 4]. Se consigue una mayor maniobrabilidad cuando las ruedas son diferenciales ya que la diferencia de velocidades entre ambas ruedas provoca un cambio de dirección sin precisar desplazamientos [5]. Sin embargo, con estas ruedas el control es más complejo debido a que las ruedas deben girar a la misma velocidad para moverse en línea recta [2, p. 38].

Cuando está formado por tres ruedas, posee estabilidad estática debido a que tiene tres puntos de apoyo. La rueda individual suele servir para direccionamiento mientras que las coaxiales para tracción. Es más frecuente que el caso de dos ruedas debido a su mayor simplicidad.

Si posee cuatro ruedas, como en el caso de los vehículos convencionales, es denominado de Ackerman [6, p. 28]. Las ruedas delanteras proporcionan la dirección, y la tracción en las ruedas que deseemos según la configuración elegida. Tiene una menor maniobrabilidad que en los otros dos casos, pero es el más utilizado y permite mayores velocidades debido a una mayor estabilidad con respecto a los anteriores.

Los vehículos con pistas de deslizamientos son muy útiles en navegación por terrenos irregulares con una mayor resistencia al desgaste [6, p. 32]. Al igual que los tanques,

consiguen el direccionamiento girando las pistas a velocidades distintas o en sentidos opuestos, y la impulsión sincronizando ambas. Un ejemplo es el robot Nanokhod.

En el caso de ruedas síncronas, todas las ruedas actúan simultáneamente mediante un motor que define la velocidad del vehículo a través de una transmisión con coronas de engranajes o correas concéntricas. Otro motor establece la orientación de todas las ruedas (Figura 2) [7]. Como el chasis no cambia de orientación a pesar de que sí lo hagan las ruedas, a menudo se incorpora un tercer motor para girar el chasis en la dirección deseada y así utilizar los sensores incorporados [8, p. 12].

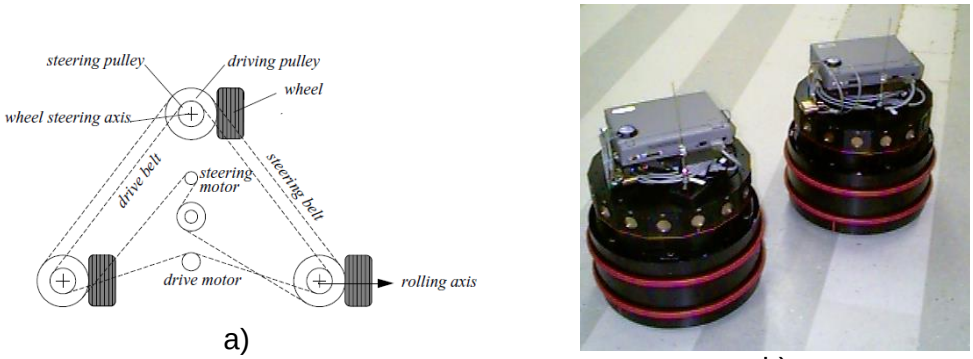


Figura 2: Ruedas síncronas
a) Synchro b) Nomadics 150
a) [2] b) (http://www.cc.gatech.edu/ai/robot-lab/images/nomads.jpg)

Otro tipo de ruedas son las Mecanum (también llamadas “suecas” o “ilon”) que permiten una mayor libertad de movimiento reduciendo las restricciones cinemáticas al poseer unos rodillos en su perímetro a un cierto ángulo, como por ejemplo a 45º (Figura 3a). Pueden desplazarse hacia delante y detrás como los de robots Ackerman. Sin embargo, girando las ruedas de un lado en el sentido opuesto a las del otro lado, puede rotar sobre sí mismo, y girando las ruedas que están en una diagonal en sentido opuesto a las ruedas de la otra diagonal permite desplazarse lateralmente [9]. La eficiencia es menor debido a que se produce deslizamiento, pero su omnidireccionalidad es total. Precisan una construcción y un control más complicados.

Por último, queda citar los robots con ruedas en extremos de patas. Permiten la mayor maniobrabilidad en terrenos difíciles con obstáculos ya que combinan las ventajas de las patas y las ruedas. El robot Shrimp (Figura 3b) tiene seis ruedas motorizadas, siendo capaz de escalar obstáculos dos veces más grandes que el tamaño de sus

ruedas. La dirección la consiguen sincronizando el giro de la rueda delantera y trasera y la velocidad de las cuatro intermedias. Gracias a esto pueden realizar maniobras muy precisas [2, p. 43].

a) b)

Figura 3: Robots con ruedas
a) Mecanum MIT b) Shrimp
a) (https://upload.wikimedia.org/wikipedia/commons/0/00/UranusOmniDirectionalRobotPodnar.png)
b) (http://www.unusuallocomotion.com/medias/images/Shrimp-robot.jpg?fx=r_250_250)

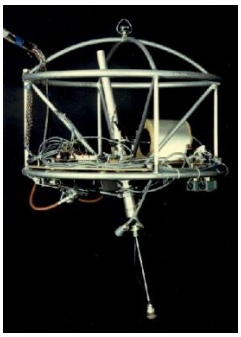
2.2 LOCOMOCIÓN MEDIANTE PATAS

Los robots con esta configuración pueden separar su cuerpo del terreno mediante una serie de puntos de soporte, las patas. Según su disposición y morfología, poseerán mayor o menor estabilidad estática y dinámica. En contraste a los robots con ruedas, tienen un deslizamiento menor y pueden llegar a moverse en todas las direcciones sin restricciones. Asimismo, poseen la capacidad de superar obstáculos en terrenos irregulares, incluso salvar agujeros y grietas siempre que no superen su paso [2, p. 17]. Sin embargo, la complejidad para la planificación y control se eleva debido a un mayor número de mecanismos y grados de libertad. Esto también conlleva un mayor consumo de energía en su locomoción [6, p. 35]. Además, sus patas deben soportar el peso del cuerpo y normalmente, elevar y descender el cuerpo del robot.

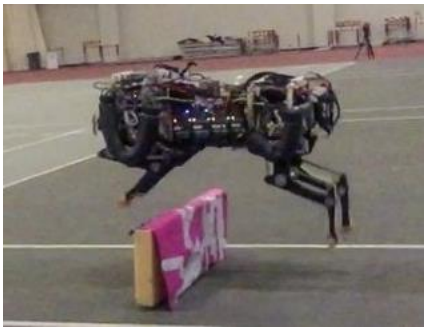
Como ya se ha dicho, podemos configurar los robots con distinto número de patas, asemejándose a seres vivos de iguales características, siendo las más comunes los bípedos, los cuadrúpedos, los de seis patas y los de ocho. No obstante, también existen robots con una sola pata (Figura 4a), aunque poco extendidos, que poseen ciertas ventajas: no hace falta coordinar su movimiento con otras patas, pueden saltar grietas más grandes con previa carrerilla… Sin embargo, al igual que los robots de dos patas y de dos ruedas, mantienen el equilibrio de forma dinámica, y no de forma estática [2, p. 21].

Aquellos con dos y cuatro patas (Figura 4b) han evolucionado en los últimos años, desarrollando modelos capaces de correr, saltar, subir y bajar escaleras, e incluso dar saltos mortales. Estando quietos pueden permanecer estables estáticamente dentro de unos límites, sin embargo, al moverse necesitan equilibrado dinámico ya que el centro de gravedad debe ser desplazado activamente durante el paso [2, p. 24].

Por el contrario, las configuraciones con seis patas o más, al igual que los insectos, son estables estáticamente durante su movimiento ya que siempre tienen tres puntos apoyados en el suelo (Figura 4c), reduciendo la complejidad del control [2, p. 27].



a)



b)

c)

Figura 4: Robots con patas

a) Hopper MIT b) Robot Cheetah MIT c) Genghis MIT

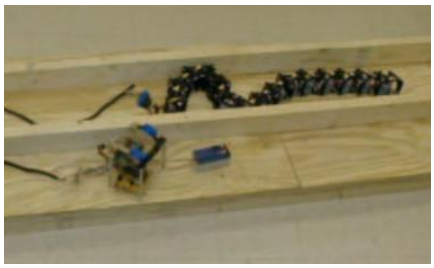
a) (http://www.ai.mit.edu/projects/leglab/robots/3D_hopper/3D_hopper.jpeg)
b) (http://www.cadena365.com/wp-content/uploads/2015/06/Cheetah-robot-530x350-460x307.jpg)
c) (http://cdn1.bostonmagazine.com/wp-content/uploads/2014/10/brooks1-1015x895.jpg)

2.3 CONFIGURACIONES ARTICULADAS

Este tipo de robots estudiados y clasificados en [10] reciben los nombres de “serpiente” o “ápodos” debido a que no tienen patas ni miembros para desplazarse. Son muy útiles para caminos estrechos debido a su reducida sección y uniformidad, y para terrenos difíciles ya que pueden moverse por cualquier superficie, independientemente de lo escarpada que sea. Constan de una serie de módulos conectados entre sí, y que por tanto, pueden intercambiarse por otros cuando hace

falta. Según su morfología y los algoritmos implantados, se mueven de distintas formas, siendo las más usuales:

- Sinusoidal, que al igual que las serpientes, propagan un movimiento ondulatorio desde la cabeza hasta la cola.
- Concertina, para la que siempre debe apoyarse en dos puntos, bien en suelo y techo dentro de agujeros, bien en dos paredes paralelas y cercanas (Figura 5a).
- Mediante ruedas en cada uno de los segmentos (Figura 5b).
- Rodante, en la que la cabeza se une a la cola formando un bucle, y rueda al cambiar los servos de ángulo consecutivamente.
- Onda de contracción y dilatación que se propaga desde la cabeza hasta la cola (Copernicus), asemejándose a los gusanos de tierra.



a)

b)

c)

Figura 5: Configuraciones articuladas a) Mov. concertina b) Makro c) Recarga eléctrica de Tesla

a) [10] b) [10] c) (http://www.motor.es/fotos-noticias/2015/08/tesla-serpiente-robotica-201522610_1.jpg)

2.4 ROBOTS AÉREOS Y SUBMARINOS

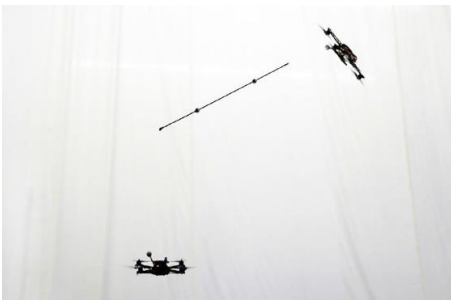
Como se puede observar en [2, p. 78], los robots aéreos o UAV (“Unmanned Aerial Vehicles”) han aumentado en diversidad durante los últimos años debido a la gran amplitud de aplicaciones: operaciones militares, vigilancia, investigaciones meteorológicas, investigación robótica, ocio, etc. A diferencia de los robots terrestres, poseen seis grados de libertad para moverse en el espacio tridimensional.

Básicamente se pueden clasificar en dos grupos: robots aéreos de ala fija y los de rotores o giroaviones (“rotorcraft”).

Los primeros son similares a los aviones de transporte de pasajeros con alas para proporcionar la sustentación, con propulsores para dar el empuje y superficies de control para maniobrar (Figura 6a).

Los segundos tienen la ventaja de despegar verticalmente y pueden tener distintas configuraciones, entre ellas los helicópteros, con rotor principal y de cola, pero también los cuadrópteros (Figura 6b). Estos últimos son cada vez más usados por la facilidad de maniobrarlos, incluso en entornos cerrados, y por la cantidad de código abierto disponible en Internet para programar y construir ya que a diferencia de los helicópteros, no tienen un mecanismo de plato oscilante. Modificando la velocidad de giro de los rotores se puede conseguir el movimiento deseado en los seis grados de libertad. En el ETH de Zúrich, Raffaello D’Andrea, profesor de sistemas dinámicos y control, ha creado un programa de investigación innovando nuevas plataformas y aplicaciones para varios tipos de robots. En el campo de los cuadrópteros en el FMA (Flying Machine Arena) ha realizado grandes avances permitiendo una interacción y colaboración entre dichos aparatos: construcción, jugar con pelotas y con barras, aprender trayectorias y transmitir la información aprendida a otros, etc. Utiliza un sistema de captura de movimiento de alta precisión, redes de comunicación inalámbrica y algoritmos sofisticados de software para estimaciones y control [11].

a) [12]



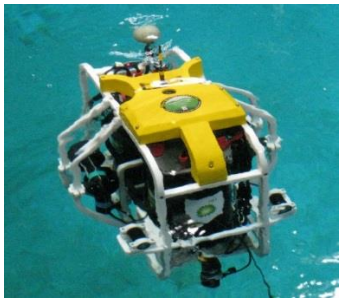
b) [13]

Figura 6: Vehículos aéreos
a) UAV Global Hawks de ala fija b) Cuadrópteros haciendo acrobacias con pértiga

Los robots submarinos o AUV (“Autonomous Underwater Vehicles”) son muy similares a los aéreos ya que se puede considerar que “vuelan” por el agua. A diferencia de los UAV, están sometidos a una fuerza de flotación hacia arriba, fuerzas de arrastre mayores que el aire y una masa añadida. Además, poseen la desventaja de la

navegación ya que no pueden utilizar satélites a partir de una determinada profundidad, implicando la necesidad de utilizar otros métodos como sensores de presión, Doppler Velocity Log (DVL), Acoustic Doppler Current Profiler (ADCP)… [14] Este tipo de robots permiten obtener información y datos de entornos poco accesibles para el ser humano: hábitats, peces y organismos, volcanes submarinos, etc. además de fines militares como por ejemplo buscar minas. Hay robots equivalentes a los de ala fija y a los de rotores. En la Figura 7a podemos observar el Sparus II desarrollado por la Universidad de Gerona y en la Figura 7b el Nessie IV creado por el Heriot-Watt University Ocean Systems Laboratory.

a)



b)

Figura 7: Vehículos submarinos
a) Sparus II (Universidad Gerona) b) Nessie IV
a) (http://cirs.udg.edu/wp-content/uploads/2014/09/cirs3-780x500.jpg)
b) (http://auvac.org/uploads/configuration/nessie.jpg)

2.5 APLICACIONES

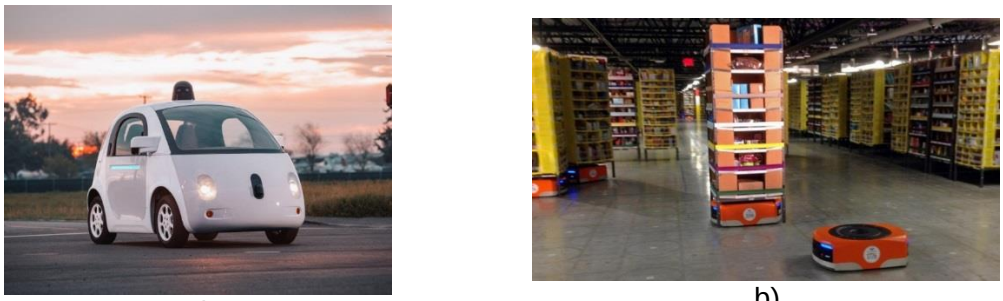
Los robots móviles tienen una gran variedad de aplicaciones y cada vez son más a medida que se desarrollan nuevas plataformas, sensores, mecanismos, etc. que permiten una mayor autonomía, seguridad y controlabilidad. Gracias a ellos se puede explorar, estudiar o limpiar sitios poco accesibles o inseguros para los seres humanos, así como realizar tareas repetitivas o pesadas.

Algunas de estas aplicaciones con algún ejemplo de robots son:

- Limpieza, ya sea doméstica (iRobot Roomba, Cye) o de grandes superficies (SwingoBot 1650, AeroBot 1850 creados por Intellibot Robotics y mostrados en la Figura 8), ya en prueba en hospitales, hoteles y en estaciones de París. También limpieza de lugares peligrosos e inaccesibles para el ser humano, como la central nuclear de Fukushima, para la que se siguen desarrollando modelos de robots que puedan resistir la alta radiación presente y desmantelar los reactores.

Figura 8: Robots limpieza
Captura de http://www.intellibotrobotics.com/products/

- Transporte, tanto de pasajeros mediante vehículos autónomos (Google Car [7, pp. 60-63] como el de la Figura 9a u otras marcas que lo están desarrollando) o de objetos y cargas (Amazon Prime Air [15] que enviará paquetes a casas mediante drones, robot DRU desarrollado por Domino's que enviará pizzas a casas [16], transporte de maletas…). Actualmente se utilizan los robots en logística de almacenamiento en los almacenes de Amazon con los robots Kiva (Figura 9b) [17]. También para transporte de determinados equipos en centros sanitarios y hospitales (AVGS de Robotnik [18]).



a) b)

Figura 9: Robots de transporte
a) Google Car b) Kiva para logística de almacén
a) (http://www.wired.com/wp-content/uploads/2015/10/selfdriving-car-gallery5-1024x768.jpg)
b) (http://mediad.publicbroadcasting.net/p/shared/npr/styles/x_large/nprshared/201412/367707362.jpg)

- Sanidad, como el robot enfermero Hospi de la Figura 10a, desarrollado por la empresa Matsushita o RoboCourier entre otros que permite transporte de comida, análisis y medicamentos.

- Agricultura para aplicaciones de fumigación, estado de la cosecha, riego, entre otros. En este campo podemos hablar por ejemplo de robots como Aurora desarrollado por la Universidad de Málaga [1, p. 5] o Summit XL creado por Robotnik [18]. También hay robots para cortar árboles, ramas [19], limpieza de bosques…

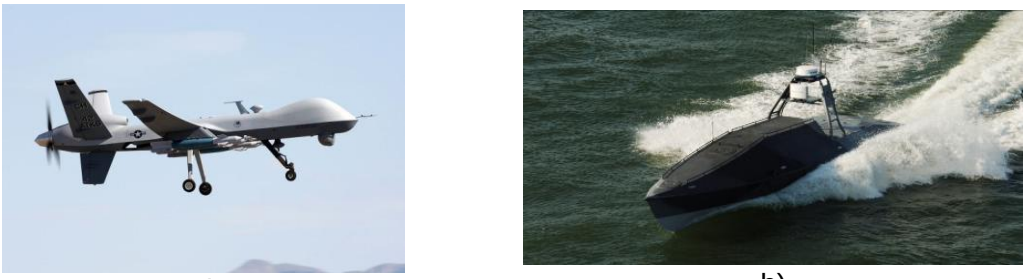
- Exploración en otros planetas como Marte (Mars Rover [20]), en la Antártida (Nomad [7, pp. 64-66]), en océanos (Figura 7) y minera (Groundhog [7, pp. 58-60]).

a) b)

Figura 10: Aplicaciones de sanidad y exploración
a) Hospi b) Mars Rovers

a) [1, p. 8] b) (http://mars.nasa.gov/msl/multimedia/images/?ImageID=3793)

- Inspección tanto de alcantarillas y tuberías [10] (Figura 5b).
- Comprobación de puentes para revisar soldaduras y realizar operaciones de limpieza y pintura con las configuraciones articuladas. Un robot destacado del tipo con patas es el trepador que se sostiene mediante garras, succión o dispositivos magnéticos permitiendo moverse por paredes para realizar su misión [6, p. 35].
- Militar, controlados remotamente. Hay drones con el fin de tomar instantáneas de terreno enemigo, explorar (General Atomics Guardian), para atacar (MQ-9 Reaper Figura 11a), para vigilancia y rescate (buques de superficie autónomos -ASV- Figura 11b), detectar minas, etc.



a) b)

Figura 11: Robots militares
a) MQ-9 Reaper b) ASV

a) (http://www.military.com/equipment/mq-9-reaper)
b) (http://www.mapcorp.com/wp-content/uploads/image11.jpeg)

- En situaciones peligrosas: radiación, humo, calidad del aire, edificios, situaciones de catástrofes (terremotos), explosivos, etc. [21, pp. 7-8]. Los de tipo serpiente pueden utilizarse para introducirse entre los escombros y con una mini-cámara en la cabeza puede ser manipulado por un operador o de forma autónoma.
- Investigación mediante todos los tipos de robots móviles vistos, por ejemplo, cuadrópteros (Figura 6b), de distinto número de patas (Figura 4) o distinto número de ruedas (Figura 3), vehículos autónomos, con Lego Mindstorms, etc.
- Ocio, para el entretenimiento de personas de cualquier edad. Con forma y comportamiento de animal (AIBO [8, p. 31]), helicópteros, cuadrópteros…

3 REPRESENTACIÓN DE LA POSICIÓN Y ORIENTACIÓN

En robótica es un requisito fundamental representar la posición y orientación de objetos en el entorno, ya sean robots, obstáculos, cámaras, caminos o piezas de trabajo. Primero estudiaremos la posición y orientación en el plano y posteriormente en el espacio. Esto nos permitirá entender y desarrollar los modelos cinemáticos y dinámicos de los robots móviles.

Existen diversas notaciones para representar la posición y la orientación. En [6] se utiliza la notación de J.J.Craig ("*Introduction to Robotics: Mechanics and Contro*l"). Sin embargo, utilizaré la notación de [12] que procede de R.P. Paul (*"Robot manipulators: mathematics, programming and control"*).

3.1 SISTEMAS DE COORDENADAS EN EL PLANO

Un punto en el plano puede ser descrito por medio de un vector en un sistema de coordenadas. Utilizaremos el sistema de coordenadas cartesianas, formado por ejes ortogonales, dos para el plano, que se cortan en un punto, el origen (Figura 12a).

Normalmente consideraremos el objeto como un conjunto de puntos. Asumiremos que el objeto es rígido y por tanto, dichos puntos mantienen una posición relativa constante entre ellos. Si asociamos un sistema de referencia al objeto, en lugar de describir cada uno de los puntos individualmente con vectores, podremos definir la posición y orientación de todo el objeto, llamada "pose", mediante dicha referencia ya que no hay movimiento relativo entre ellos. Llamando a este sistema de coordenadas {B}, sus ejes serán definidos como x_B e y_B, indicando con el subíndice su marco de referencia (Figura 12b).

La pose relativa de un sistema con respecto a uno de referencia lo denotaremos mediante el símbolo ξ. Por tanto, la pose relativa del sistema {B} respecto al {A} será: ξ_B^A. El superíndice es el sistema de referencia y el subíndice el sistema que estamos describiendo.

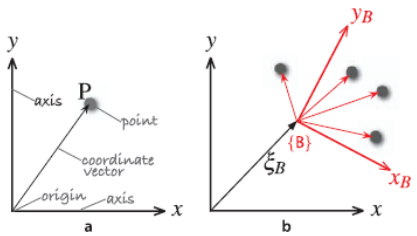


Figura 12: Sistemas de coordenadas cartesianas [12, p. 15]

También podemos escribir ξ_B^A para describir un movimiento, como un desplazamiento y una rotación, transformando de {A} a {B}. Si no hay superíndice, asumiremos que el cambio en la pose es relativa al sistema de coordenadas global "O".

Un punto P puede ser descrito respecto a cualquier sistema de referencia (Figura 13a). La relación entre estas descripciones puede venir dada por:

$$p_A = \xi_B^A \cdot p_B \qquad (3.1)$$

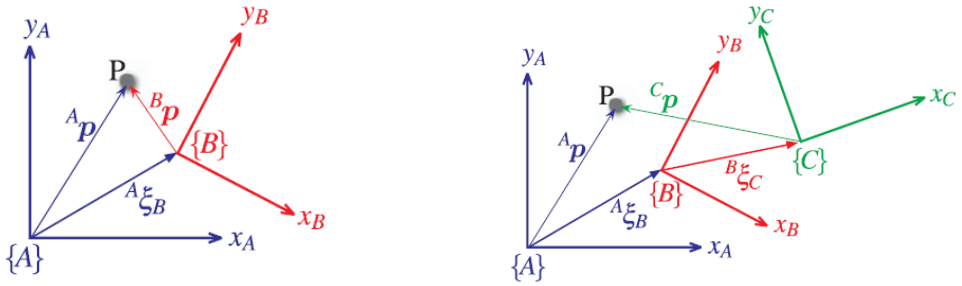
en la que el lado derecho de la ecuación expresa el movimiento de {A} a {B} y luego a P.

Además, también podemos hacer una composición de poses relativas, describiendo un sistema en función de otro intermedio. Es decir, si por ejemplo tenemos el sistema de la Figura 13b, el sistema {C} respecto a {A} se puede obtener componiendo las poses relativas de {A} a {B} y de {B} a {C}. Matemáticamente lo escribiríamos así:

$$\xi_C^A = \xi_B^A \oplus \xi_C^B \qquad (3.2)$$

Podemos comprobar que está bien escrito tachando en el lado derecho de la ecuación el subíndice y el superíndice intermedios ya que se "cancelan", quedando ambos índices iguales a izquierda y derecha de la ecuación.

(En las figuras el superíndice está escrito a la izquierda y no a la derecha).



a) b)

Figura 13: Punto P descrito con varios sistemas de coordenadas

a) [12, p. 15] b) [12, p. 16]

En la misma figura, podemos expresar el punto P según el sistema {A} o según el {C} teniendo en cuenta la composición de poses relativas que acabamos de hacer. Quedaría:

$$p_A = (\xi_B^A \oplus \xi_C^B) \cdot p_C \qquad (3.3)$$

3.2 SISTEMAS DE COORDENADAS EN EL ESPACIO

Si ahora pasamos a tres dimensiones, utilizaremos sistemas de coordenadas con tres ejes ortogonales entre sí. Al igual que en 2D, podemos describir unos sistemas en función de otros componiéndolos. Por ejemplo, como observamos en la Figura 14, una cámara fija llamada {F} observaría un objeto {B} desde un punto de vista fijo y estima la pose relativa de él con respecto a sí misma. Otra cámara no fija {C}, sino unida a un robot {R} (con lo que la pose relativa entre estos no varía), estima la pose relativa del mismo objeto respecto a ella misma. Por tanto, podemos describir el objeto de distintas formas según los sistemas utilizados.

$$\xi_R \oplus \xi_C^R \oplus \xi_B^C = \xi_F \oplus \xi_B^F \qquad (3.4)$$

Asimismo, el robot puede ser descrito a través de la cámara fija:

$$\xi_R = \xi_F \oplus \xi_R^F \qquad (3.5)$$

Si queremos despejar la pose relativa del robot respecto a la cámara fija, podemos restar ξ_F a ambos lados de la ecuación añadiendo el inverso de ξ_F mediante el símbolo $\ominus \xi_F$ (esto significa: $\ominus \xi_Y^X = \xi_X^Y$). Por tanto:

$$\ominus \xi_F \oplus \xi_F \oplus \xi_R^F = \ominus \xi_F \oplus \xi_R$$
$$\xi_R^F = \ominus \xi_F \oplus \xi_R \qquad (3.6)$$

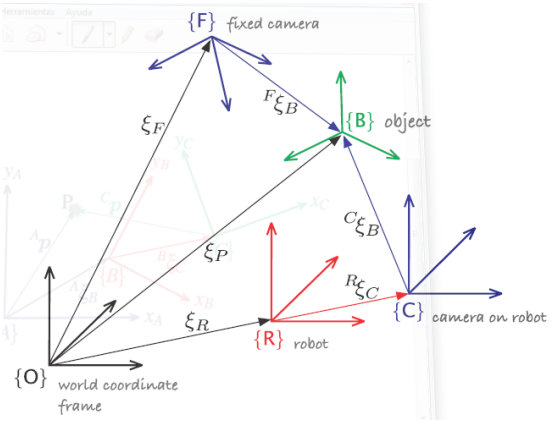


Figura 14: Múltiples sistemas de coordenadas 3D y poses relativas [12, p. 17]

3.3 POSICIÓN Y ORIENTACIÓN

3.3.1 Posición y orientación en el plano

En un sistema de coordenadas cartesianas en el plano, típicamente designaremos al eje horizontal “x” y al eje vertical “y”. La intersección de ambos es el origen. Los

vectores unitarios paralelos a estos ejes los designaremos con: $\hat{x}$ e $\hat{y}$. Por tanto, un punto P será representado en dicho sistema con sus coordenadas (x,y) o como un vector:

$$p = x \cdot \hat{x} + y \cdot \hat{y} \qquad (3.7)$$

Si queremos describir un sistema {B} como el de la Figura 15 respecto al {A}, observamos que el origen está desplazado un vector t=(x,y) y rotado un ángulo θ en sentido anti horario. Por tanto, la pose se puede representar así: $\xi_B^A \sim (x, y, \theta)$, sabiendo que $\backsim$ significa equivalencia entre dos representaciones. Sin embargo, está representación no se utiliza para composición porque es una función trigonométrica compleja de ambas poses.

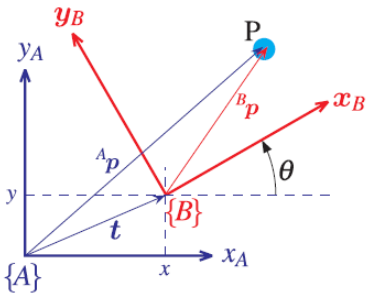


Figura 15: Relación entre sistemas de coordenadas en 2D [12, p. 20]

En cambio, dividiremos el problema en dos partes: rotación y traslación, observando la relación entre p_A y p_B de un punto P cualquiera.

Para la rotación, si en la figura anterior colocamos un nuevo marco {V} con el mismo origen que {B} y los ejes paralelos a los de {A}, el punto P se puede definir en función de los vectores unitarios de {B} y de {V} (Figura 16).

$$p_B = x_B \cdot \hat{x}_B + y_B \cdot \hat{y}_B = (\hat{x}_B \quad \hat{y}_B) \begin{pmatrix} x_B \\ y_B \end{pmatrix} \qquad (3.8)$$

$$p_V = x_V \cdot \hat{x}_V + y_V \cdot \hat{y}_V = (\hat{x}_V \quad \hat{y}_V) \begin{pmatrix} x_V \\ y_V \end{pmatrix} \qquad (3.9)$$

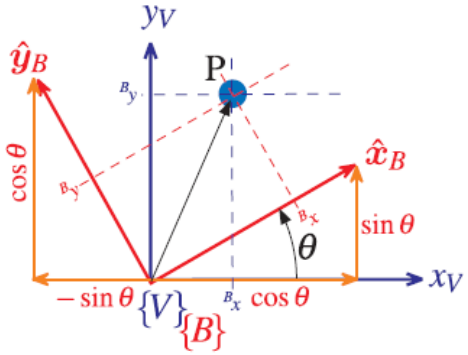


Figura 16: Sistema de coordenadas rotado en 2D [12, p. 21]

Los vectores unitarios del sistema {B} pueden describirse en función de los de {V}.

$$\hat{x}_B = cos\theta \cdot \hat{x}_V + sin\theta \cdot \hat{y}_V \quad (3.10)$$

$$\hat{y}_B = -\sin\theta \cdot \hat{x}_V + \cos\theta \cdot \hat{y}_V \quad (3.11)$$

Expresado matricialmente:

$$(\hat{x}_B \quad \hat{y}_B) = (\hat{x}_V \quad \hat{y}_V)\begin{pmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{pmatrix} \quad (3.12)$$

Si sustituimos estos vectores unitarios en la descripción de P en función de {B} queda:

$$p_B = (\hat{x}_V \quad \hat{y}_V)\begin{pmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}x_B \\ y_B\end{pmatrix} \quad (3.13)$$

Igualando los segundos miembros de las ecuaciones (3.9) y (3.13), obtenemos la transformación de los puntos del marco {B} al {V} cuando el sistema es rotado.

$$\begin{pmatrix}x_V \\ y_V\end{pmatrix} = \begin{pmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}x_B \\ y_B\end{pmatrix} \quad (3.14)$$

A esta matriz la llamamos matriz de rotación y se denotará por R_B^V.

$$\begin{pmatrix}x_V \\ y_V\end{pmatrix} = R_B^V \cdot \begin{pmatrix}x_B \\ y_B\end{pmatrix} \quad (3.15)$$

La matriz de rotación es ortonormal ya que cada una de sus columnas es un vector unitario y son ortogonales entre sí. Al ser ortogonal, la inversa es igual a la traspuesta: $R^{-1} = R^T$, lo cual permite obtener fácilmente la matriz de transformación de {V} a {B} trasponiendo la original: $(R_B^V)^{-1} = (R_B^V)^T = R_V^B$. Esto nos lleva a la identidad: $R(-\theta) = R(\theta)^T$. Además, su determinante es +1, por lo que la longitud de un vector no varía tras una rotación.

La segunda parte del problema era la traslación entre los orígenes de los sistemas {A} y {B}. Utilizaremos el marco {V} ya que comparte el mismo origen que {B} pero sus ejes son paralelos a los de {A}. Cualquier punto P del plano puede ser descrito según {A} mediante {V} más las coordenadas del origen (x,y) del sistema trasladado.

$$\begin{pmatrix}x_A \\ y_A\end{pmatrix} = \begin{pmatrix}x_V \\ y_V\end{pmatrix} + \begin{pmatrix}x \\ y\end{pmatrix} = \begin{pmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}x_B \\ y_B\end{pmatrix} + \begin{pmatrix}x \\ y\end{pmatrix} \quad (3.16)$$

De forma más compacta:

$$\begin{pmatrix}x_A \\ y_A\end{pmatrix} = \begin{pmatrix}\cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y\end{pmatrix}\begin{pmatrix}x_B \\ y_B \\ 1\end{pmatrix} \quad (3.17)$$

Podemos representar en una sola matriz T la rotación y traslación de un sistema a otro, quedando:

$$\begin{pmatrix} x_A \\ y_A \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} = \begin{pmatrix} R_B^A & t \\ 0_{1x2} & 1 \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} = T \cdot \begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} \qquad (3.18)$$

Para esta representación más compacta hemos utilizado coordenadas homogéneas, las cuales utilizan N+1 componentes cuando solo hacen falta N. En 2D solo hacen falta 2 parámetros, pero añadimos un tercero que es un factor de escala. No existe una representación única de un punto en el sistema de coordenadas homogéneas, ya que para diferentes valores del factor de escala, se obtienen diferentes representaciones del mismo vector físico. En robótica utilizaremos normalmente un factor de escala unidad, siendo las coordenadas homogéneas iguales a las físicas [22, p. 21].

3.3.2 Posición y orientación en el espacio

Este caso es una extensión del visto en el plano añadiendo un eje z ortogonal a los otros dos. Su sentido será aquel que queden positivos sus vectores unitarios al realizar los siguientes productos vectoriales:

$$\begin{aligned} \hat{z} &= \hat{x} \times \hat{y} \\ \hat{x} &= \hat{y} \times \hat{z} \\ \hat{y} &= \hat{z} \times \hat{x} \end{aligned} \qquad (3.19)$$

Un punto P lo representaríamos con las coordenadas (x,y,z) o el vector:

$$p = x \cdot \hat{x} + y \cdot \hat{y} + z \cdot \hat{z} \qquad (3.20)$$

Como puede verse en la Figura 17, si queremos representar el sistema {B} respecto a la referencia {A}, vemos que su origen está desplazado un vector t=(x,y,z) y rotado. Por tanto, describiremos un punto P en ambos sistemas y estudiaremos la relación entre ellos. Para ello, al igual que en plano, dividiremos el problema en rotación y traslación.

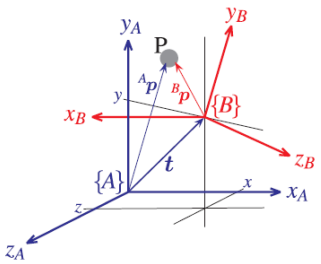


Figura 17: Relación entre sistemas de coordenadas en 3D [12, p. 25]

Para la rotación, el teorema de rotación de Euler dice que se puede transformar un sistema de coordenadas en otro con una secuencia de máximo tres rotaciones alrededor de sus ejes, no siendo dos rotaciones sucesivas sobre el mismo eje [12, p. 25]. Hay que destacar la importancia del orden en que se aplican las rotaciones ya que no son conmutativas, al igual que el operador composición $\oplus$. Por tanto, se han desarrollado varios métodos para representar rotaciones: matriz de rotación ortonormal, ángulos de Euler y Cardan, rotación de eje y ángulos y los cuaterniones unitarios. Solo veremos el primer tipo, explicándose los demás en [12, pp. 27-37].

La matriz de rotación ortonormal se obtiene expresando los vectores unitarios del sistema deseado en función del marco de referencia. Cada vector tiene tres elementos y forman las columnas de una matriz ortonormal 3x3 R_B^A. El siguiente sistema rota un vector definido en un sistema {B} a otro descrito respecto al {A}.

$$\begin{pmatrix} x_A \\ y_A \\ z_A \end{pmatrix} = R_B^A \cdot \begin{pmatrix} x_B \\ y_B \\ z_B \end{pmatrix} \qquad (3.21)$$

Al igual que en el caso del plano, se cumplen las propiedades de las matrices ortonormales: $R^{-1} = R^T$ y det(R)=1. Las matrices de rotación ortonormales para un ángulo θ alrededor de los ejes coordenados son:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \qquad (3.22)$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \qquad (3.23)$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (3.24)$$

Para expresar la pose relativa en el espacio nos falta combinar junto con la rotación la traslación. Las dos representaciones más prácticas son el par vector cuaternión y la matriz 4x4 de transformación homogénea. No veremos la primera, estando explicada en [12]. La segunda es similar a la obtenida en el plano, pero extendida para la dimensión z.

$$\begin{pmatrix} x_A \\ y_A \\ z_A \\ 1 \end{pmatrix} = \begin{pmatrix} R_B^A & t \\ 0_{1x3} & 1 \end{pmatrix} \begin{pmatrix} x_B \\ y_B \\ z_B \\ 1 \end{pmatrix} = T \begin{pmatrix} x_B \\ y_B \\ z_B \\ 1 \end{pmatrix} \qquad (3.25)$$

Esta matriz T está compuesta por la submatriz 3x3 R_B^A, el vector de traslación cartesiano entre los orígenes t, y una última fila compuesta por tres 0 y un 1.

Al igual que en el caso del plano, utilizaremos coordenadas homogéneas, pero aquí añadiremos un cuarto parámetro ya que necesitamos 3 por ser 3D. Volveremos a utilizar un factor de escala unidad para que sean iguales las coordenadas físicas y las homogéneas.

4 MODELOS CINEMÁTICOS Y DINÁMICOS DE ROBOTS MÓVILES

4.1 CINEMÁTICA

La cinemática es el estudio más básico del comportamiento de los sistemas mecánicos [2, pp. 47-88], [6, pp. 97-113], [23, pp. 450-452], [12, pp. 67-69], [8, pp. 469-785], [24, pp. 89-93]. Debemos entender dicho comportamiento para diseñar robots móviles apropiados para realizar las tareas deseadas y con el software de control correspondiente a ese hardware.

A diferencia de robots o brazos manipuladores, un robot móvil se puede mover íntegramente con respecto a su entorno. El espacio de trabajo de un robot móvil define el rango de las posibles poses (posición y orientación) que puede alcanzar en su entorno. La controlabilidad define los posibles caminos y trayectorias en ese espacio.

Estudiaremos el caso de los robots móviles con ruedas, adoptando las siguientes hipótesis simplificadoras [6, p. 97]:

- El robot se mueve sobre una superficie plana con rodadura pura, es decir, sin deslizamiento en el periodo de control.
- El robot no tiene partes flexibles, sino que se comporta como un sólido rígido, por lo que si existen partes móviles (ruedas de dirección), se situarán en la posición adecuada mediante el sistema de control.
- Durante un periodo de tiempo pequeño en el que se mantiene constante la dirección, el vehículo se moverá de un punto al siguiente siguiendo un arco de circunferencia.
- Los ejes de guiado son perpendiculares al suelo.

Cada una de sus ruedas tiene un papel para permitir el movimiento del robot e introduce una restricción, como impedir el derrape lateral. Como las ruedas están “atadas” entre sí debido a la geometría del robot, sus restricciones se combinan para crear unas restricciones conjuntas para todo el movimiento del chasis del robot.

4.1.1 Modelo cinemático directo

Representaremos el robot como un cuerpo rígido con ruedas moviéndose en un plano horizontal. Tendremos por tanto, tres grados de libertad: dos para la posición en el plano y una para la orientación en el eje vertical, perpendicular al plano. Hay más grados de libertad y de flexibilidad internos debido a los ejes de las ruedas, juntas… pero nosotros consideraremos solo el cuerpo rígido del robot para el chasis.

Siguiendo la teoría y notación de [2, pp. 47-88], especificaremos la posición del robot en el plano utilizando una referencia global o fija del plano $\{X_I, Y_I\}$ con origen en *O* y una local para el robot $\{X_R, Y_R\}$ con origen en *P*, que es su punto de referencia. La posición de *P* en el marco de referencia global vendrá dada por las coordenadas "*x*" e "*y*", y la orientación θ como la diferencia angular entre ambas referencias. Por tanto, la pose del robot en el sistema de referencia global según la Figura 18a vendrá dada por:

$$\xi_I = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \qquad (4.1)$$

Para describir el movimiento del robot en función de sus componentes, tendremos que observar la relación del movimiento entre ambas referencias, lo cual es función de la pose actual del robot. Esto lo realizaremos con la matriz de rotación ortogonal vista anteriormente para el caso de rotación alrededor del eje *z* (perpendicular al plano) en el espacio (3.24), pero denominada en este caso:

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (4.2)$$

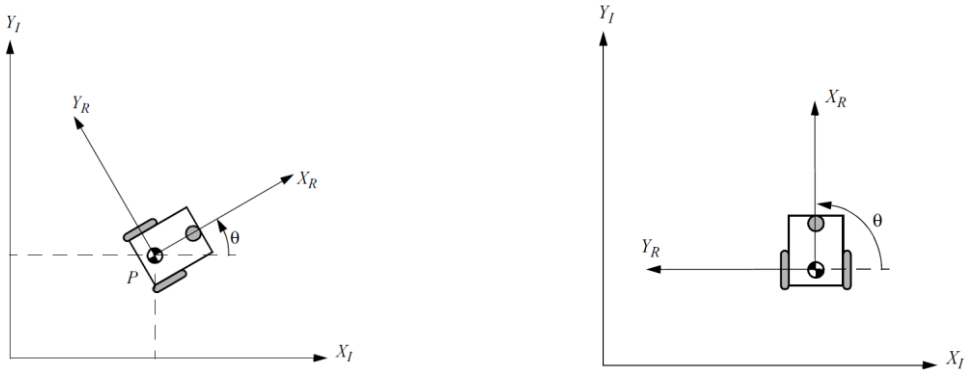
La matriz (4.2) la utilizaremos para pasar del sistema local (móvil) al global, pero si queremos pasar del global al móvil, utilizaremos la inversa, que es:

$$R(\theta)^{-1} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad (4.3)$$

Si tenemos una velocidad $(\dot{x}, \dot{y}, \dot{\theta})$ en el sistema de referencia global, y para el ejemplo de la Figura 18b, con un $\theta = \frac{\pi}{2}$, calculamos las componentes del movimiento en los ejes locales del robot con la siguiente ecuación:

$$\dot{\xi}_R = R\left(\frac{\pi}{2}\right)^{-1} \cdot \dot{\xi}_I = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \dot{y} \\ -\dot{x} \\ \dot{\theta} \end{pmatrix} \qquad (4.4)$$

Se puede observar en la Figura 18b que el movimiento a lo largo del eje X_R es paralelo al eje Y_I mientras que el eje Y_R tiene la dirección del eje X_I pero en sentido negativo.



a) [2, p. 49] b) [2, p. 50]

Figura 18: a) Sistemas de referencia global y local b) Alineado con un eje global

Podemos obtener el movimiento de un robot, por ejemplo, el de accionamiento diferencial de la Figura 19 dada su geometría y la velocidad de sus ruedas. El ejemplo considerado tiene dos ruedas traseras, cada una de radio r, con el punto de referencia *P* en la mitad del segmento que une ambas a una distancia l de cada una. A partir de r, l , θ y la velocidad de giro de cada una de las ruedas: $\dot{\varphi}_1$ y $\dot{\varphi}_2$, seremos capaces de deducir la velocidad general del robot en el sistema de referencia global:

$$\dot{\xi}_I = R(\theta) \cdot \dot{\xi}_R = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = f(l, r, \theta, \dot{\varphi}_1, \dot{\varphi}_2) \qquad (4.5)$$

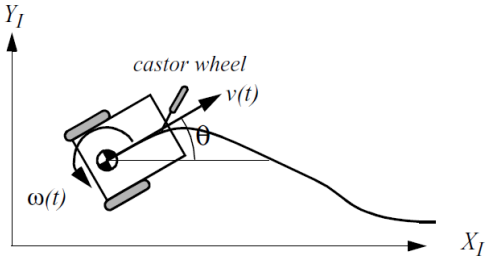


Figura 19: Robot móvil diferencial en sist. de referencia global [2, p. 51]

Para este ejemplo calcularemos primero la contribución de cada rueda al marco de referencia local dibujado en la Figura 18a, moviéndose el robot y el punto *P* a lo largo de $+X_R$. Si una rueda gira mientras la otra no contribuye y está estacionaria, como *P* está a mitad de camino entre ambas, tendrá instantáneamente la mitad de velocidad:

$$\dot{x}_{r1} = \frac{1}{2} \cdot r \cdot \dot{\varphi}_1 \qquad (4.6)$$

$$\dot{x}_{r2} = \frac{1}{2} \cdot r \cdot \dot{\varphi}_2 \qquad (4.7)$$

En un robot diferencial se pueden sumar ambas contribuciones para calcular la componente $\dot{x}_R$ de $\dot{\xi}_R$. Si cada rueda girara con la misma velocidad pero en sentidos opuestos, el resultado de $\dot{x}_R$ es cero quedándose estacionario pero rotando alrededor de *P*. El valor de $\dot{y}_R$ es siempre nulo ya que ninguna rueda contribuye al movimiento en la dirección de Y_R, es decir, no se puede mover lateralmente. Finalmente, el valor de $\dot{\theta}_R$ se puede hallar sumando las contribuciones independientes de cada rueda. Si consideramos que solo la rueda de la derecha (rueda 1) gira hacia delante, el móvil y el punto *P* rotarán en sentido contrario a las agujas del reloj pivotando sobre la rueda de la izquierda (rueda 2). La velocidad de rotación ω_1 en *P* se puede calcular ya que la rueda 1 se mueve instantáneamente a lo largo de un arco circular de radio $2l$:

$$\omega_1 = \frac{r \cdot \dot{\varphi}_1}{2l} \qquad (4.8)$$

Se puede aplicar el mismo cálculo a la rueda de la izquierda, teniendo en cuenta que si gira hacia delante, el robot rota en sentido de las agujas del reloj. Esto resulta en un signo negativo ya que es el sentido negativo del eje perpendicular al plano definido anteriormente.

$$\omega_2 = -\frac{r \cdot \dot{\varphi}_2}{2l} \qquad (4.9)$$

Podemos por tanto, obtener un modelo cinemático para el ejemplo del robot diferencial con (4.2) y las ecuaciones que acabamos de hallar.

$$\dot{\xi}_I = R(\theta) \cdot \dot{\xi}_R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{r\dot{\varphi}_1}{2} + \frac{r\dot{\varphi}_2}{2} \\ 0 \\ \frac{r\dot{\varphi}_1}{2l} + \frac{-r\dot{\varphi}_2}{2l} \end{pmatrix} \qquad (4.10)$$

Suponiendo que nos dan los siguientes datos: $\theta = \frac{\pi}{2}$, $r = 1$, $l = 1$ y las ruedas giran desigualmente con $\dot{\varphi}_1 = 4$ y $\dot{\varphi}_2 = 2$, podemos calcular la velocidad en el sistema de referencia global:

$$\dot{\xi}_I = R(\theta) \cdot \dot{\xi}_R = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2+1 \\ 0 \\ 2-1 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \\ 1 \end{pmatrix} \qquad (4.11)$$

Lo que significa que el robot tendría en ese instante una velocidad de 3 (las unidades que sean) a lo largo del eje Y_I del sistema fijo mientras rota con una velocidad de 1 en sentido contrario a las agujas del reloj. Por tanto, hemos visto un procedimiento sencillo para calcular de forma directa el movimiento del robot, dadas las velocidades de sus ruedas.

4.1.2 Restricciones cinemáticas de las ruedas

Podemos diferenciar cinco tipos básicos de ruedas: rueda estándar fija, rueda estándar dirigible, rueda de castor, rueda sueca y rueda esférica. Se distinguen en las restricciones cinemáticas que cada una impone. Luego, el movimiento de todas las ruedas se puede combinar para calcular el movimiento de todo el robot. Describiremos cada una brevemente, pudiéndolo encontrar explicado y desarrollado con ecuaciones en [2, pp. 53-61].

Rueda estándar fija

La rueda estándar fija no tiene un eje vertical de orientación, por lo que su ángulo es fijo respecto al chasis. El movimiento está limitado hacia delante y hacia atrás a lo largo del plano de la rueda y rotando en torno al del punto de contacto con el plano del suelo.

Impone las restricciones de rodadura pura en el punto de contacto cuando se mueve en la dirección del movimiento y no hay deslizamiento en la dirección ortogonal del plano de movimiento de la rueda.

Rueda estándar dirigible

La rueda estándar dirigible se diferencia de la fija en que hay un grado de libertad más debido a que puede rotar alrededor de un eje vertical que pasa por el centro de la rueda y por el punto de contacto con el suelo.

Impone las mismas restricciones que la rueda fija.

Rueda de castor

Las ruedas de castor (“castor wheel”) son también capaces de orientarse alrededor de un eje vertical. Sin embargo, a diferencia de la rueda estándar dirigible, dicho eje no pasa por el centro de la rueda ni por el punto de contacto con el suelo.

La restricción de rodadura no varía debido a que el offset en el eje vertical no afecta durante el movimiento a lo largo del plano de la rueda, debido a que dicho eje también pertenece a ese plano. Esto implica que el ángulo de giro sobre este eje vertical coincide con el que adquiere la rueda para moverse.

Por otro lado, la restricción de deslizamiento se ve afectada por la geometría. Esto es debido a que las fuerzas laterales sobre las ruedas actúan en la unión del eje vertical

con el chasis del robot, y no el punto de contacto con el suelo. Debido a este offset, la restricción no consiste en que el movimiento lateral sea cero, sino que cualquier movimiento ortogonal al plano de la rueda se equilibre con una cantidad igual y opuesta del movimiento de la dirección del castor. Es decir, si el chasis experimenta una fuerza hacia la izquierda, el offset provoca que la rueda gire en torno al eje vertical para compensarlo hasta que la fuerza y el plano del movimiento de la rueda estén alineados. Podemos observarlo por ejemplo, en las sillas de oficinas y carros de la compra que poseen este tipo de rueda (Figura 20).

Por tanto, las ruedas de castor proporcionan un movimiento omnidireccional al robot.

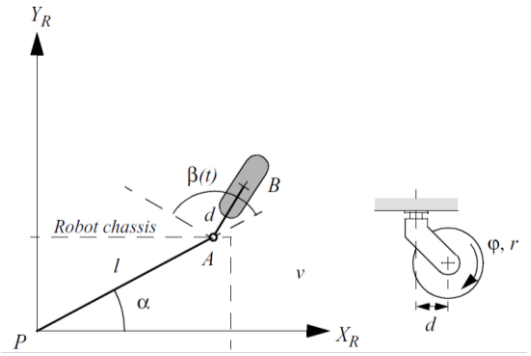


Figura 20: Rueda de castor; parámetros relativos a referencia móvil [2, p. 57]

Rueda sueca

Como ya describimos en 2.1, las ruedas suecas, Mecanum o ilon, no tienen eje vertical de rotación y poseen unos rodillos en su perímetro a un cierto ángulo γ, normalmente de 45º (Figura 3a). Esto les permite ser omnidireccionales como las ruedas de castor ya que además de desplazarse hacia delante y detrás en la dirección del plano de la rueda, pueden utilizar los rodillos para deslizarse lateralmente, es decir, en la dirección ortogonal al plano de la rueda.

Para ello, se debe sincronizar el movimiento giratorio de las ruedas. Por ejemplo, el robot puede rotar sobre sí mismo si las ruedas de un lado del chasis giran en sentido opuesto a las del otro lado. Si en cambio, giramos las ruedas que están en una diagonal en sentido opuesto a las ruedas de la otra diagonal, se puede desplazar lateralmente [9].

Requieren una construcción y un control más complicados. Sin embargo, la eficiencia es menor debido a que se produce deslizamiento y no rodadura pura al moverse en el plano ortogonal al plano de la rueda.

Rueda esférica

La rueda esférica no tiene un eje principal de rotación pudiendo moverse arbitrariamente en cualquier dirección, por lo que no hay restricciones apropiadas de rodadura o deslizamiento. Al igual que las dos anteriores es omnidireccional y no impone restricciones en la cinemática del chasis del robot.

Dado un robot móvil con *M* ruedas, se puede calcular las restricciones cinemáticas del chasis del robot teniendo en cuenta las restricciones de las ruedas individuales y su disposición en él. Las ruedas de castor, suecas y esféricas no imponen restricciones cinemáticas en el chasis porque $\dot{\xi}_I$ puede variar libremente debido a los grados de libertad internos de las ruedas. Sin embargo, las ruedas estándar fija y estándar dirigible sí tienen un impacto en la cinemática del chasis del robot, por lo que habrá que tenerlo en cuenta para calcular las restricciones cinemáticas del robot.

4.1.3 Vehículo de Ackerman

Dentro de todos los tipos de robots móviles con ruedas, el vehículo tipo coche o también llamado de Ackerman es el más simple [12, pp. 67-70], [2, pp. 67-69], [21, pp. 166-168]. Está constituido por dos ruedas traseras estándar fijas, y dos ruedas delanteras estándar dirigibles. Una bicicleta representa el mismo modelo pero con sólo una rueda de cada tipo.

Como hemos dicho antes, ambos tipos de ruedas introducen la restricción cinemática de no movimiento lateral, es decir, no puede moverse a lo largo de la dirección perpendicular al plano de la rueda. En la Figura 21 ésta dirección de “movimiento cero” sería la punteada para cada rueda. La intersección de estas líneas punteadas da lugar a un punto característico denominado Centro Instantáneo de Rotación (*CIR* o *ICR* en inglés). Según el ángulo de dirección que impongamos a las ruedas delanteras, obtendremos un *CIR* en cada instante. Si mantenemos constante dicho ángulo, el robot describirá una trayectoria circular de radio *R* con centro en el *CIR*. Si suponemos un *R* infinito, es que el robot se mueve en línea recta, con un ángulo de dirección nulo.

La construcción geométrica del *CIR* es función del número de restricciones del movimiento del robot y no del número de ruedas. Esto es debido a que obtenemos la misma solución si consideramos una bicicleta con sólo una rueda trasera y una

delantera. Cada rueda contribuye a una restricción o línea de “movimiento cero”, intersectando en un punto (*CIR*). Ambas restricciones son independientes y por tanto constituyen las restricciones del movimiento total del robot.

El vehículo de Ackerman en cambio, a pesar de tener más ruedas no introduce más restricciones independientes. Las ruedas traseras al ser fijas y al estar alineadas, introducen una única restricción. Junto con la restricción de una de las ruedas delanteras, obtenemos una solución del *CIR*. La otra rueda delantera estará posicionada de forma que su línea de “movimiento cero” coincida con el *CIR* ya existente (Figura 21a), por lo que no introduce una restricción independiente al movimiento. Esto tiene como consecuencia que las ruedas delanteras tengan una ligera variación en el ángulo de dirección entre sí, conseguido mediante el mecanismo de dirección de Ackerman. Con este mecanismo conseguimos además un menor desgaste menor en los neumáticos [12, p. 69].

Por otro lado, incorporaremos engranajes diferenciales debido a que las ruedas deben girar a velocidades distintas ya que los radios entre las ruedas de ambos lados del vehículo y el *CIR* son distintos.

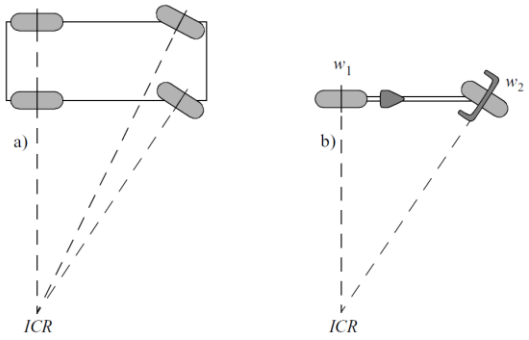


Figura 21: Modelo cinemático
a) Vehículo tipo Ackerman b) Modelo de bicicleta [2, p. 68]

En [2, pp. 69-81] podemos encontrar información y explicación de otros modelos de vehículos con ruedas, así como diversas propiedades cinemáticas, encontrándose fuera del alcance del presente trabajo.

4.2 DINÁMICA

Una trayectoria es un camino más la especificación del tiempo en que se alcanza cada configuración. Llamamos configuración a la pose (posición y orientación) que puede alcanzar el robot móvil en su movimiento. La planificación de trayectorias no es por tanto, sólo un problema cinemático, sino también dinámico. Encontrar

trayectorias factibles respetando la dinámica del sistema requiere conocer restricciones adicionales como sus masas e inercias, los límites de los actuadores, y fuerzas como la gravitacional y la de fricción. En consecuencia, en este apartado describiremos un poco la dinámica de los robots móviles [23, pp. 349-350].

Podemos obtener las ecuaciones del movimiento de un sistema mecánico de varias formas, siendo todas equivalentes. Nosotros nos centraremos en la formulación de Euler – Lagrange.

Denominaremos $q = [q_1 \dots q_n]^T$ a un vector de coordenadas generalizadas que representa la configuración del sistema en el espacio de configuraciones de *n* dimensiones. En el caso de un robot tipo coche moviéndose en un plano, suele definirse como $q = [q_1, q_2, q_3, q_4]^T$, donde (q_1, q_2) especifican la posición del centro de masas del cuerpo en el plano, q_3 la orientación del robot y q_4 el ángulo de la ruedas de dirección con respecto al eje X_R del sistema de referencia móvil [21, pp. 166-168]. Llamaremos $u = [u_1 \dots u_n]^T$ al vector de fuerzas generalizadas que actúan en las coordenadas generalizadas, pudiendo ser fuerzas o pares.

El Lagrangiano ($\mathcal{L}$) viene definido en función de la energía cinética ($\mathcal{K}$) y de la energía potencial ($\mathcal{P}$) del sistema mediante la siguiente ecuación:

$$\mathcal{L}(q, \dot{q}) = \mathcal{K}(q, \dot{q}) - \mathcal{P}(q) \qquad (4.12)$$

La energía cinética es función de la configuración y de la velocidad, mientras que la energía potencial sólo depende de la configuración. Las ecuaciones de Euler – Lagrange del movimiento son:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) - \frac{\partial \mathcal{L}}{\partial q_i} = u_i \qquad (4.13)$$

Siendo i un índice que va desde 1 hasta *n*.

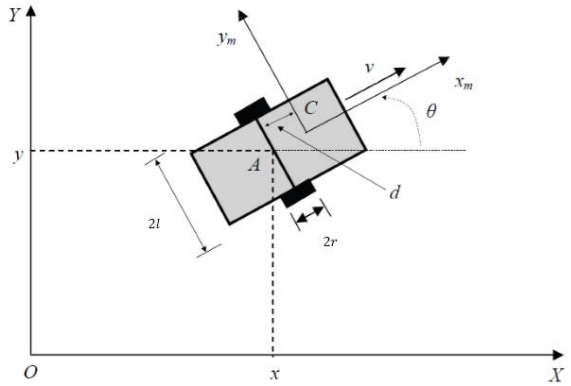


Figura 22: Robot móvil no holonómico [24, p. 531]

Seguiremos el ejemplo de un robot no holonómico (Figura 22) moviéndose en un plano descrito en [24, pp. 531-533]. El robot del ejemplo es un tipo Hilare [21, pp. 163-165] y tiene dos ruedas motrices independientes montadas en el mismo eje y una rueda delantera libre. Las dos ruedas motrices son impulsadas por dos actuadores (motores DC) para alcanzar la pose deseada. Podemos definir la posición del robot en el sistema de referencia global con el centro de masas *C* o con el punto A, punto medio del eje entre ambas ruedas motrices, al igual que en la Figura 19. θ será la orientación del robot, definida como el ángulo entre el sistema móvil y el global.

La energía cinética de toda la estructura de un está compuesta por la energía cinética de traslación ($\mathcal{K}_t$), de rotación en el plano ($\mathcal{K}_r$) y de rotación de las ruedas y rotores de los motores de corriente continua ($\mathcal{K}_{wr}$).

$$\mathcal{K} = \mathcal{K}_t + \mathcal{K}_r + \mathcal{K}_{wr} \qquad (4.14)$$

Cada una vendrá definida por las siguientes ecuaciones:

$$\mathcal{K}_t = \frac{1}{2} M {v_c}^2 = \frac{1}{2} M \left({\dot{x}_c}^2 + {\dot{y}_c}^2 \right) \qquad (4.15)$$

$$\mathcal{K}_r = \frac{1}{2} I_A \dot{\theta}^2 \qquad (4.16)$$

$$\mathcal{K}_{wr} = \frac{1}{2} I_0 {\dot{\theta}_R}^2 + \frac{1}{2} I_0 {\dot{\theta}_L}^2 \qquad (4.17)$$

Donde M es la masa del todo el vehículo, v_c es la velocidad lineal del centro de masas *C*, I_A es el momento de inercia de todo el vehículo respecto *A*, I_0 es el momento de inercia del conjunto del rotor más la rueda, y $\dot{\theta}_R$ y $\dot{\theta}_L$ son las velocidades angulares de las ruedas derecha e izquierda, respectivamente. Los valores de $\dot{x}_c = \dot{x}_A - \dot{\theta} d sin\theta$ e $\dot{y}_c = \dot{y}_A + \dot{\theta} d cos\theta$ los podemos obtener a partir de $\dot{x}_A$ e $\dot{y}_A$, calculados de la misma forma que en 4.1.1:

$$\dot{x}_A = \frac{r}{2} \left(\dot{\theta}_R + \dot{\theta}_L \right) \cdot \cos\theta \qquad (4.18)$$

$$\dot{y}_A = \frac{r}{2} \left(\dot{\theta}_R + \dot{\theta}_L \right) \cdot \sin\theta \qquad (4.19)$$

$$\dot{\theta} = \frac{r\left(\dot{\theta}_R - \dot{\theta}_L \right)}{2l} \qquad (4.20)$$

Quedando por tanto:

$$\dot{x}_c = \frac{r}{2} \left(\dot{\theta}_R + \dot{\theta}_L \right) \cdot \cos\theta - \dot{\theta} \cdot d \cdot \sin\theta \qquad (4.21)$$

$$\dot{y}_c = \frac{r}{2} \left(\dot{\theta}_R + \dot{\theta}_L \right) \cdot \sin\theta + \dot{\theta} \cdot d \cdot \cos\theta \qquad (4.22)$$

La energía potencial no viene definida en este ejemplo, pero tomaremos la utilizada en [3, p. 28]. La define como el trabajo necesario para levantar el centro de masas de cada uno de los elementos que compone la estructura desde un plano de referencia hasta su posición bajo la influencia de la gravedad.

$$\mathcal{P} = \sum_{j=1}^{k} \mathcal{P}_j = \sum_{j=1}^{k} (m_j \cdot c_j \cdot g) \qquad (4.23)$$

Siendo c_j una función de las variables del elemento tratado (robot en un plano).

Obtendremos el modelo dinámico del robot sustituyendo las ecuaciones que acabamos de describir en el Lagrangiano (4.12) y en la ecuación de Euler – Lagrange (4.13), creando una ecuación por cada coordenada generalizada. En el ejemplo utiliza como coordenadas generalizadas las velocidades de rotación de las ruedas, ya que es un robot tipo Hilare.

Existen otros métodos de modelado dinámico, como Newton – Euler, sin embargo, el de Euler – Lagrange suele ser más entendible físicamente. En resumen, este método consiste en:

1. Calcular la energía cinética del sistema.
2. Calcular la energía potencial.
3. Obtener el Lagrangiano ($\mathcal{L}$).
4. Desarrollar las ecuaciones de Euler – Lagrange.

La ecuación en forma estándar de la dinámica de un sistema que se obtiene al seguir los pasos anteriores es:

$$u = M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) \qquad (4.24)$$

Donde $C(q,\dot{q})$ es una matriz lineal en $\dot{q}$, $g(q)$ es un vector de fuerzas gravitacionales y $M(q)$ es la matriz de inercia, simétrica y definida positiva (determinante y traza son positivos).

En [23, pp. 353-371] podemos encontrar más desarrollado este método y además explica la obtención de la energía cinética de rotación para un sólido rígido en el plano y en el espacio por medio de integrales, matrices de inercia, etc. En [8, pp. 485-489] también se explica Euler – Lagrange con otra notación, y en [12, pp. 191-203] se trata la ecuación diferencial matricial que resulta, explicando cada una de sus matrices y con ejemplos numéricos en Matlab para un brazo manipulador.

5 CONTROL DE ROBOTS MÓVILES

En este capítulo trataremos unos de los temas más importantes en la robótica: el control. Gracias a él conseguiremos que el robot actúe de la forma que queramos para ejecutar tareas, ya que si actúa impredeciblemente, la tarea puede hacerse incorrectamente. Un ejemplo son los brazos manipuladores de la industria, ya que si no controlamos su movimiento de forma precisa, harán movimientos no deseados y posiblemente no obtengamos el resultado esperado. No podemos pensar en utilizar el robot Da Vinci para cirugía si no hay detrás un control preciso de su movimiento ya que lo contrario podría ser catastrófico para el paciente.

En el campo de la robótica móvil también es importante el control para que a partir de la información procedente de los sensores, sea capaz de ejecutar de forma autónoma un plan predefinido o realizar tareas reaccionando si hay cambios en el entorno.

En la Figura 23 podemos ver el esquema básico general de la estructura de control de un robot móvil según [1, pp. 12-15].

- Generador Global de Trayectorias (GGT): Es el nivel jerárquico superior con el que se decide, según la tarea que le asignemos, las coordenadas del punto de destino y de los puntos intermedios de la trayectoria. Si dicho camino está obstruido, redefinirá la trayectoria elegida. La información empleada puede ser generada “off-line”, es decir, con el conocimiento previo del ambiente de trabajo, u “on-line”, utilizando la información de los sensores y elaborando mapas del entorno (SLAM).
- Generador Local de Trayectorias (GLT): Es el nivel jerárquico intermedio que simula a un operador del robot evitando los obstáculos del camino, corrigiendo la trayectoria y adecuando la velocidad a las maniobras. Permite un control dinámico del robot móvil e informa al GGT de los resultados del objetivo. Está comunicado con el sistema sensorial, permitiéndole tomar decisiones “on-line” y generando los valores de referencia para el nivel de control inferior. Se han utilizado algoritmos de todos los tipos, desde los más básicos, hasta la Inteligencia Artificial.
- Control Local de Sistema de Tracción y Dirección (CL): Es el nivel jerárquico inferior que interpreta las referencias enviadas por el GLT y genera las

acciones de control sobre los motores de tracción y dirección. Esto permite que trabajen de forma coordinada y se alcance el punto de destino con trayectorias suaves y libres de oscilaciones y maniobras violentas. Los controladores de este nivel son generalmente los de control clásico.

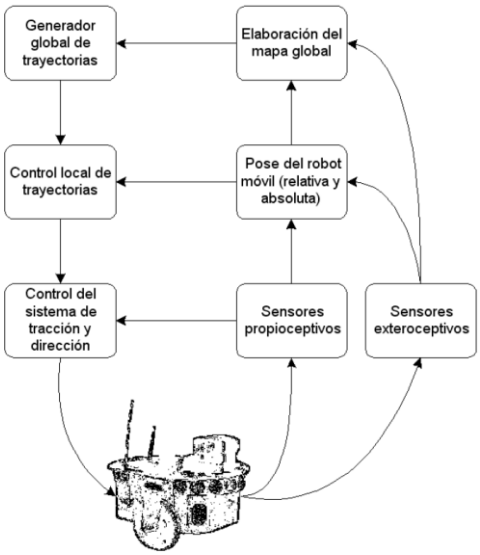


Figura 23: Esquema general del sistema de control de un robot móvil [1, p. 13]

En la Figura 24 podemos ver un esquema de control para vehículos autónomos según [6], que englobaría parte de la GLT y la CL. Está formado por dos bucles. El de la izquierda es el control de la dirección, generando las órdenes para que los actuadores sigan en cada instante la dirección apropiada. El bucle de la derecha es el control de la velocidad, generando las consignas a los actuadores encargados de la propulsión del vehículo. Según el robot estarán más o menos acoplados. En el caso de vehículos convencionales tipo triciclo o Ackerman, los bucles están fuertemente acoplados.

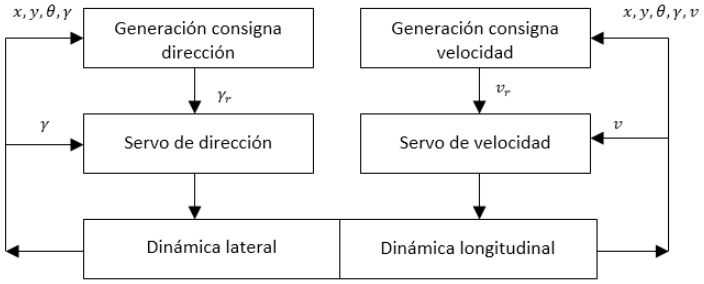


Figura 24: Control de velocidad y desplazamiento lateral. Posición (x,y), orientación (θ), curvatura (γ) y velocidad (v) [6, p. 259]

En consecuencia, la planificación de caminos en entornos desconocidos se realiza por GLT considerando el entorno próximo del robot para determinar la dirección. Esto conlleva que no sean óptimos. Si el ambiente es conocido, se realiza la planificación mediante GGT, que tiene en cuenta todos los caminos posibles y elige el óptimo.

Podemos encontrar diferentes estrategias de control, desde deliberativas hasta reactivas. El control deliberativo se asemeja a la inteligencia artificial: razona la percepción del robot (datos de los sensores) y construye un modelo del entorno (memoria) planificando las acciones del robot. Se requiere una gran capacidad computacional y toma de decisión, implicando una respuesta lenta. Además, la semejanza entre el entorno y el modelo en el robot debe ser precisa para que el comportamiento del robot sea el deseado.

El control reactivo elimina el conocimiento: no existe planificación, razonamiento ni modelo del entorno. Los estímulos del entorno (datos de los sensores) provocan unas acciones "reflejas", resultando una respuesta rápida y de bajo costo computacional. Esto permite reaccionar ante cambios en el entorno.

En la mitad entre ambos tipos de control encontramos el basado en comportamientos, aunando las ventajas de ambos: sea capaz de planificar las acciones del robot construyendo un modelo y a la vez reaccione ante obstáculos y cambios en el entorno inesperados. Así, se quiere conseguir un algoritmo confiable (como el deliberativo) con una velocidad de respuesta acorde a la velocidad del robot móvil (como el reactivo).

5.1 SEGUIMIENTO DE TRAYECTORIA

Podemos encontrarnos diversos tipos de control: clásico, moderno, fuzzy… y aplicado a distintos objetivos. Nosotros nos centraremos en el seguimiento de trayectorias en el plano, que es un tema de interés en nuestro trabajo [6, pp. 258-293], [8, pp. 502-510], [2, pp. 81-88], [24, pp. 529-538], [25, pp. 213-243]. En 9.1 veremos un poco un caso parecido: persecución pura.

Dentro del problema podemos distinguir el caso de seguimiento de la trayectoria en el tiempo de una pose (posición y orientación) de referencia $\rho_{ref} = (x, y, \theta)_{ref}$, o el caso de seguimiento en el tiempo de una posición de referencia $\rho_{ref} = (x, y)_{ref}$. En ambos casos se pretende que el error $\rho_{ref}(t) - \rho(t)$ tienda a cero manteniendo acotadas las señales de control. Este problema del seguimiento de $\rho_{ref}(t)$ involucra tanto el control de la dirección como el de la velocidad del robot autónomo. Un ejemplo es el caso de seguir a otro móvil obteniendo la referencia por los sensores. Si en

cambio, pretendemos seguir un camino predefinido sin considerar la velocidad, solo estará involucrado el control de la dirección.

Hay que tener en cuenta las características del robot móvil. Las técnicas para controlar un robot omnidireccional pequeño son diferentes que para un vehículo automóvil convencional adaptado para su funcionamiento autónomo.

Para que el problema de seguimiento tenga solución, la trayectoria cartesiana $(x_d(t), y_d(t))$ deseada debe ser admisible para el modelo cinemático que estemos tratando. En este caso utilizaremos un modelo de vehículo monociclo (también llamado uniciclo) con una rueda dirigible con la configuración: $q = [x\ y\ \theta]^T$ (Figura 25).

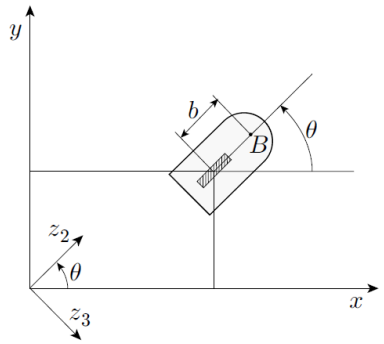


Figura 25: Coordenadas generalizadas para un monociclo [8, p. 478]

Dicho vehículo tiene el siguiente modelo cinemático:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta\\ \sin\theta\\ 0\end{bmatrix} v + \begin{bmatrix}0\\ 0\\ 1\end{bmatrix} \omega \qquad (5.1)$$

Donde v es la velocidad de conducción que introducimos y ω es la velocidad de dirección, es decir, la velocidad angular alrededor del eje vertical (perpendicular al plano). Por tanto, la trayectoria deseada debe satisfacer las siguientes ecuaciones eligiendo las entradas de referencia v_d y ω_d:

$$\begin{aligned}\dot{x}_d &= v_d \cos\theta_d\\ \dot{y}_d &= v_d \sin\theta_d\\ \dot{\theta}_d &= \omega_d\end{aligned} \qquad (5.2)$$

Según [8, p. 191], un sistema dinámico no lineal es plano (“flat”) diferencialmente si existen un conjunto de salidas $\boldsymbol{y}$, llamadas salidas planas, tales que el estado $\boldsymbol{x}$ y la entrada de control $\boldsymbol{u}$ se pueden expresar algebraicamente en función de $\boldsymbol{y}$ y sus derivadas en el tiempo hasta un cierto orden *r*.

$$\boldsymbol{x} = \boldsymbol{x}\left(\boldsymbol{y}, \dot{\boldsymbol{y}}, \ddot{\boldsymbol{y}}, \dots, \boldsymbol{y}^{(r)}\right)$$
$$\boldsymbol{u} = \boldsymbol{u}\left(\boldsymbol{y}, \dot{\boldsymbol{y}}, \ddot{\boldsymbol{y}}, \dots, \boldsymbol{y}^{(r)}\right) \quad (5.3)$$

Como consecuencia, una vez que la trayectoria de salida se asigna a $\boldsymbol{y}$, la trayectoria asociada del estado $\boldsymbol{x}$ y la historia de entradas de control $\boldsymbol{u}$ quedan singularmente determinados. En el caso del monociclo y de la bicicleta, las coordenadas cartesianas x e y son salidas planas. Esto lleva a que la orientación a lo largo de la trayectoria deseada se puede calcular de la forma:

$$\theta_d(t) = \operatorname{atan}\frac{\dot{y}_d(t)}{\dot{x}_d(t)} + k\pi \qquad k = 0{,}1. \quad (5.4)$$

Para un mismo camino cartesiano, si $k = 0$, se moverá hacia delante, y si $k = 1$, hacia atrás. Si se ha asignado una orientación inicial al robot, sólo una de las dos elecciones será la correcta. Las entradas de referencia serán:

$$v_d(t) = \pm\sqrt{\dot{x}_d{}^2(t) + \dot{y}_d{}^2(t)} \quad (5.5)$$

$$\omega_d(t) = \frac{\ddot{y}_d(t)\dot{x}_d(t) - \ddot{x}_d(t)\dot{y}_d(t)}{\dot{x}_d{}^2(t) + \dot{y}_d{}^2(t)} \quad (5.6)$$

Comparando el estado deseado $q_d(t) = [x_d(t)\ y_d(t)\ \theta_d(t)]^T$ con el actual medido $q(t) = [x(t)\ y(t)\ \theta(t)]^T$ se puede calcular un vector del error para introducir en el controlador. Pero en lugar de utilizar directamente la diferencia entre ambos, definiremos el error de seguimiento de la siguiente forma:

$$e = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d - x \\ y_d - y \\ \theta_d - \theta \end{bmatrix} \quad (5.7)$$

De acuerdo con la Figura 26, el error de posición de e es el error cartesiano $e_p = [x_d - x\ \ y_d - y]^T$ expresado en un sistema de referencia alineado con la orientación actual θ del robot.

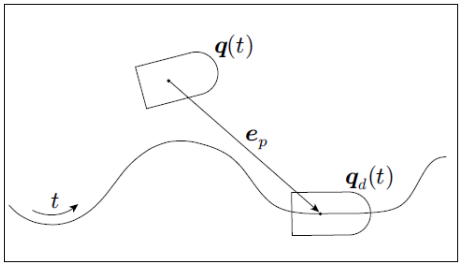


Figura 26: Error de posición e_p en seguimiento de trayectoria [8, p. 503]

Si diferenciamos e con respecto al tiempo, podemos hallar a partir de (5.1) y (5.2):

$$\begin{aligned}\dot{e}_1 &= -\omega \sin\theta\,(x_d - x) + \cos\theta\,(v_d \cos\theta_d - v\cos\theta) + \omega\cos\theta\,(y_d - y) \\ &\quad + \sin\theta\,(v_d \sin\theta_d - v\sin\theta) \\ \dot{e}_2 &= -\omega \cos\theta\,(x_d - x) - \sin\theta\,(v_d \cos\theta_d - v\cos\theta) - \omega\sin\theta\,(y_d - y) \\ &\quad + \cos\theta\,(v_d \sin\theta_d - v\sin\theta) \\ \dot{e}_3 &= \omega_d - \omega\end{aligned} \qquad (5.8)$$

Reagrupando:

$$\begin{aligned}\dot{e}_1 &= v_d(\cos\theta_d\cos\theta + \sin\theta_d\sin\theta) - v(\cos^2\theta + \sin^2\theta) \\ &\quad + \omega\big(-\sin\theta\,(x_d - x) + \cos\theta\,(y_d - y)\big) \\ \dot{e}_2 &= v_d(\sin\theta_d\cos\theta - \cos\theta_d\sin\theta) + v(\sin\theta\cos\theta - \sin\theta\cos\theta) \\ &\quad - \omega\big(\cos\theta\,(x_d - x) + \sin\theta\,(y_d - y)\big) \\ \dot{e}_3 &= \omega_d - \omega\end{aligned} \qquad (5.9)$$

Conociendo las siguientes razones trigonométricas:

$$\begin{aligned}\cos(\theta_d - \theta) &= \cos\theta_d\cos\theta + \sin\theta_d\sin\theta \\ \sin(\theta_d - \theta) &= \sin\theta_d\cos\theta - \cos\theta_d\sin\theta\end{aligned} \qquad (5.10)$$

Obtenemos las siguientes ecuaciones:

$$\begin{aligned}\dot{e}_1 &= v_d\cos e_3 - v + e_2\omega \\ \dot{e}_2 &= v_d\sin e_3 - e_1\omega \\ \dot{e}_3 &= \omega_d - \omega\end{aligned} \qquad (5.11)$$

Utilizando la siguiente transformación de entrada:

$$\begin{aligned}u_1 &= v_d\cos e_3 - v \\ u_2 &= \omega_d - \omega\end{aligned} \qquad (5.12)$$

obtenemos la siguiente expresión para la dinámica del error de seguimiento:

$$\dot{e} = \begin{bmatrix}0 & \omega & 0\\ -\omega & 0 & 0\\ 0 & 0 & 0\end{bmatrix} e + \begin{bmatrix}0\\ \sin e_3\\ 0\end{bmatrix} v_d + \begin{bmatrix}1 & 0\\ 0 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}u_1\\ u_2\end{bmatrix} \qquad (5.13)$$

5.1.1 Control basado en linealización aproximada

La aproximación más simple para diseñar un controlador de seguimiento consiste en utilizar una linealización aproximada de la dinámica del error alrededor de la trayectoria de referencia, en la que el error es cero ($e = 0$). Esta aproximación se hace

cada vez más precisa cuanto menor sea el error de seguimiento que tengamos. Para ello estableceremos $\sin e_3 = e_3$ por ser un valor pequeño. Resulta el siguiente sistema lineal variante en el tiempo:

$$\dot{e} = \begin{bmatrix} 0 & \omega_d(t) & 0 \\ -\omega_d(t) & 0 & v_d(t) \\ 0 & 0 & 0 \end{bmatrix} e + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \qquad (5.14)$$

Si $v_d(t)$ y $\omega_d(t)$ son constantes, se tiene un sistema lineal e invariante en el tiempo con la siguiente matriz de controlabilidad:

$$C = [B \;\; AB \;\; A^2B] = \begin{bmatrix} 1 & 0 & 0 & 0 & -\omega^2{}_d & v_d\omega_d \\ 0 & 0 & -\omega_d & v_d & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad (5.15)$$

Si v_d y ω_d son diferentes de cero, el sistema es controlable ya que es de rango total. Un sistema es completamente controlable cuando el estado inicial puede ser forzado a cualquier estado mediante la aplicación de una señal de entrada apropiada en el intervalo que va desde el instante inicial hasta el deseado [26]. Si $v_d = \omega_d = 0$, es decir, con el sistema de referencia en reposo, se pierde la controlabilidad ya que el rango de la matriz deja de ser total.

Consideraremos la siguiente ley de control lineal:

$$\begin{aligned} u_1 &= -k_1 e_1 \\ u_2 &= -k_2 e_2 - k_3 e_3 \end{aligned} \qquad (5.16)$$

Obtenemos la siguiente dinámica lineal en lazo cerrado:

$$\dot{e} = A(t)e = \begin{bmatrix} -k_1 & \omega_d(t) & 0 \\ -\omega_d(t) & 0 & v_d(t) \\ 0 & -k_2 & -k_3 \end{bmatrix} e \qquad (5.17)$$

El polinomio característico de la matriz A es:

$$|sI - A| = \begin{vmatrix} s + k_1 & -\omega_d & 0 \\ \omega_d & s & -v_d \\ 0 & k_2 & s + k_3 \end{vmatrix} \qquad (5.18)$$

$$s(s + k_1)(s + k_3) + v_d k_2 (s + k_1) + \omega_d{}^2 (s + k_3) = 0 \qquad (5.19)$$

Hacemos $k_1 = k_3$:

$$(s + k_1)[s(s + k_1) + v_d k_2 + \omega_d{}^2] = 0 \qquad (5.20)$$

Identificando con una ecuación característica:

$$(s + 2\delta b)[s^2 + 2\delta b s + b^2] = 0 \qquad (5.21)$$

Obtenemos los valores de las ganancias de realimentación:

$$k_1 = 2\delta b$$
$$k_2 = \frac{b^2 - {\omega_d}^2}{|v_d|} \qquad (5.22)$$
$$k_3 = 2\delta b$$

Escogeríamos los valores del coeficiente de amortiguamiento δ y frecuencia natural no amortiguada b para que los polos estén situados donde deseemos. Hay que destacar que la ganancia k_2 aumenta cuando $|v_d|$ disminuye, es decir, cuando la trayectoria de referencia cartesiana tiende a pararse. En [6, p. 278] propone para solucionarlo una variación de los polos deseados con la velocidad, llamándose "escalado de velocidad". Si por ejemplo se hace:

$$b = \sqrt{{\omega_d}^2 + \beta {v_d}^2} \qquad (5.23)$$

con $\beta > 0$, las ganancias de control serán:

$$k_1 = 2\delta\sqrt{{\omega_d}^2 + \beta {v_d}^2}$$
$$k_2 = \beta|v_d| \qquad (5.24)$$
$$k_3 = 2\delta\sqrt{{\omega_d}^2 + \beta {v_d}^2}$$

Cuando $v_d = \omega_d = 0$, no se realiza ninguna acción de control, lo cual concuerda con la pérdida de controlabilidad que decíamos antes.

La Figura 27 muestra el seguimiento de una trayectoria circular de referencia (empieza en el punto negro) con un controlador basado en linealización aproximada. A la izquierda se representa el movimiento cartesiano del monociclo y a la derecha la evolución en el tiempo de la norma del error cartesiano e_p.

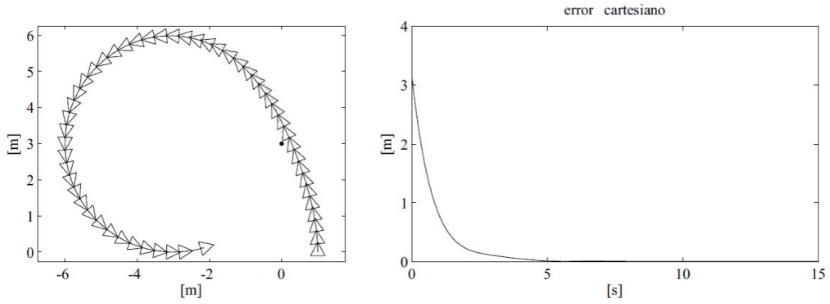


Figura 27: Seguimiento de trayectoria circular
Izq: movimiento cartesiano, dcha: evolución en el tiempo de norma e_p [8, p. 508]

En [6, pp. 278-282] podemos encontrar un ejemplo con el controlador hallado mostrando el esquema Simulink, la evolución de las señales de velocidad lineal y angular y el movimiento del robot móvil en el plano para distintas posiciones iniciales.

Además, repite el mismo ejemplo haciendo el escalado de velocidad, y se observa la mejora en los resultados obtenidos. Éste consiste en seguir una trayectoria definida con condiciones iniciales de coordenadas (0,0) y orientación 0 rad (en este caso significa alineado con el eje Y). La velocidad lineal de referencia tiene un escalón de 0.2 m/s en $t = 0$ s, y en $t = 20$ s pasa a 0.1 m/s. La velocidad angular de referencia tiene un escalón de 0.3 rad/s en $t = 0$ s y pasa a 0.2 rad/s en $t = 10$ s. El robot parte de las coordenadas (1, 0.5) con orientación de $\pi/6$ rad.

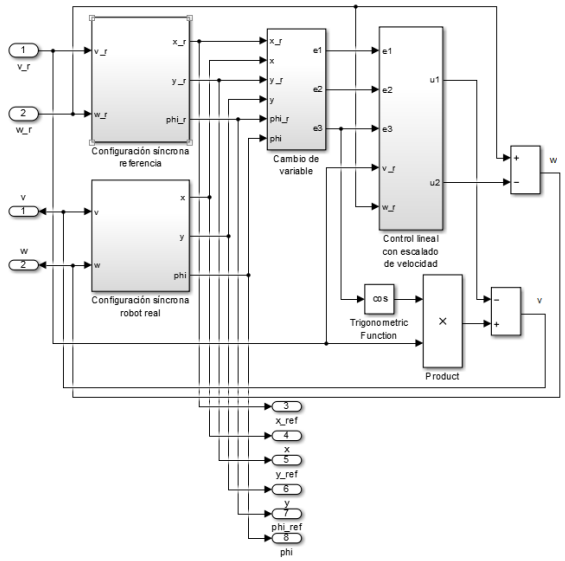


Figura 28: Esquema Simulink: seguimiento de trayectorias con control lineal y "escalado de velocidad" [6, p. 280]

Utilizaremos el "escalado de velocidad" con $\beta = 13.75$ y $\delta = 0.9$. En la Figura 28 vemos el diagrama en Simulink que resulta. En la Figura 29 se muestran los resultados de las velocidades lineal y angular obtenidas en línea continua, y las velocidades lineal y angular de referencia en línea discontinua. Vemos que conforme pasa el tiempo, los resultados alcanzan las referencias deseadas.

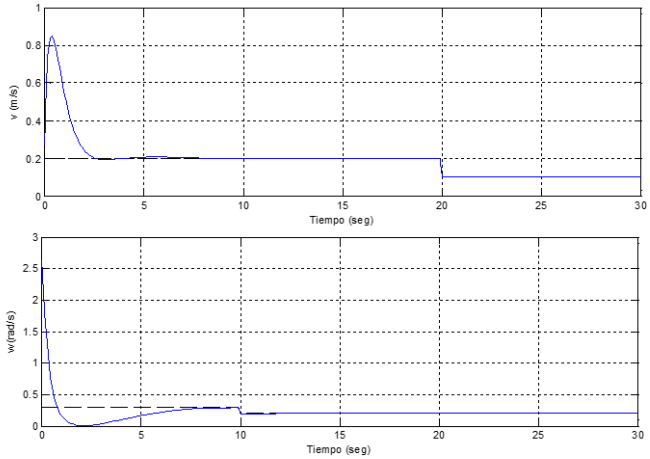
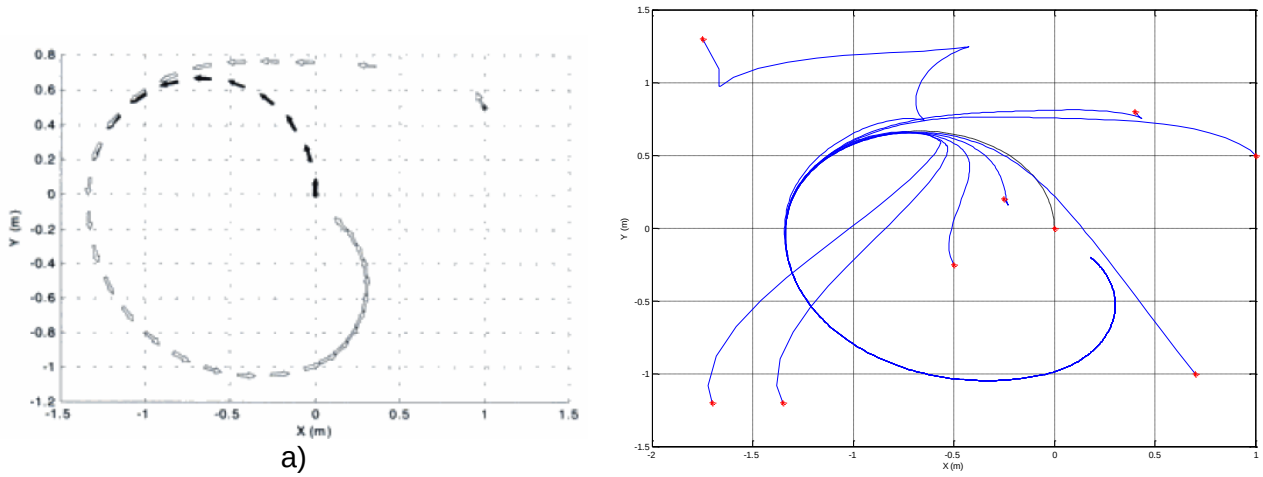


Figura 29: Velocidades lineal y angular ("escalado de velocidad") [6, p. 281]

En la Figura 30a se representa la trayectoria seguida por el robot (flechas blancas) y la referencia empezando en las condiciones iniciales (flechas negras). Se representa una flecha por cada segundo de simulación, por eso, cuando la velocidad lineal desciende a 0.1 m/s en los últimos 10 segundos, las flechas aparecen más juntas.



a) b)

Figura 30: Resultados posiciones y orientaciones
a) Posiciones y orientaciones con "escalado de velocidad" b) Diversas posiciones con orientación inicial $\pi/6$ rad

Por último, en la Figura 30b se muestra la trayectoria seguida por el robot para distintas posiciones iniciales y orientación inicial $\pi/6$ rad, pero la misma trayectoria de referencia.

En las siguientes páginas de la citada bibliografía se explica cómo sería la ley de control para el caso no lineal. Además, en [6, pp. 284-293] se desarrolla el control para el seguimiento de caminos, en lugar de trayectorias, controlando la dirección y no la velocidad. Sin embargo, debido a que se escapa al alcance de este trabajo, no lo explicaremos.

También en la bibliografía citada al principio de este capítulo podemos encontrar la descripción del control predictivo, reactivo, con lógica fuzzy, etc. para aquellos interesados en profundizar en conceptos de control para robots móviles. Una aplicación interesante que también podemos encontrar es el aparcamiento en paralelo de vehículos autónomos tipo coche y tráileres, explicándolo en [27] con fuzzy.

6 SENSORES

Una de las habilidades más importantes de un sistema autónomo es obtener información de su entorno. El conocimiento del entorno puede ser un mapa creado exteriormente y proporcionado al robot, o construido localmente por el propio robot mediante sensores y/o cámaras a medida que se desplaza por el ambiente.

Esto le permite no solo realizar tareas repetitivas sino también interactuar con él y adaptarse a nuevas situaciones mediante control. Para ello, toma medidas mediante diversos sensores y actúa en consecuencia [28, p. xi]. Podemos encontrar abundante bibliografía sobre los sensores; algunos ejemplos son: [28], [29] [2, pp. 89-180], [6, pp. 166-197] [8, pp. 209-231], [1, pp. 25-45], [30], [7, pp. 26-40], [31, pp. 31-39]

Los robots móviles pueden utilizar una gran cantidad de sensores, desde medir la temperatura interna de la electrónica del robot o la velocidad de rotación de sus motores, hasta obtener información de su entorno como distancias o su posición global [2, p. 89].

Principalmente se pueden clasificar en propioceptivos, aquellos que miden valores internos del robot: velocidad de motores, voltaje de batería… y en exteroceptivos, que adquieren información del entorno: medidas de distancia, intensidad de luz, sonido… [2, p. 89].

Dentro de los exteroceptivos podemos clasificarlos en pasivos, si miden energía del ambiente, o activos, si emiten energía al entorno y observan la reacción que se produce. Dentro del primer grupo podemos encontrar sondas de temperatura, micrófonos y cámaras. El segundo grupo tiene mayor rendimiento ya que permite interacciones con el entorno más controladas, pero hay mayor riesgo de que la energía emitida afecte a la medida real o que sufra interferencias. A este tipo pertenecen los sensores ultrasónicos y los telémetros láser [2, p. 90].

En la bibliografía ([2, p. 91], [7, p. 28]) podemos encontrar tablas con distintos tipos de sensores y a qué grupo pertenece cada uno de ellos según ambas clasificaciones. Las principales clases que aparecen son:

- Sensores táctiles: final de carrera, barrera óptica…
- Sensores de ruedas y motores: potenciómetros, resolvers, encoders (codificadores) de escobilla, ópticos, magnéticos, inductivos…

- Sensores de orientación: brújulas, giróscopos, inclinómetros…
- Balizas: GPS; balizas reflectivas, de ultrasonidos…
- Rango activo: sensores reflectivos, ultrasonidos, telémetros láser, triangulación óptica…
- Sensores de movimiento y velocidad: radar y sonido Doppler.
- Sensores de visión: cámaras CCD y CMOS.

6.1 POSICIÓN Y ORIENTACIÓN

Para que un robot móvil pueda realizar tareas tales como generar trayectorias o evitar obstáculos, necesita conocer su localización o pose (posición y orientación) con respecto a un sistema de referencia absoluto. Por tanto, para el caso bidimensional con 3 grados de libertad, requiere hallar las componentes *x* e *y* de traslación y la rotación θ del sistema de coordenadas solidario al robot $\{\xi_R\}$ respecto al absoluto $\{\xi_I\}$. Para seis grados de libertad serán tres componentes de traslación (x,y,z) y tres de orientación (θ_x, θ_y, θ_z) [1, pp. 29-45].

Generalmente los robots conocen su localización por sensores montados sobre ellos. Sin embargo, es una estimación que no es suficientemente precisa en muchas aplicaciones. Esto no se debe a la magnitud de los errores sino a su acumulación, originando una incertidumbre en posición y orientación. Esto lleva a recalibrar la estimación de la posición mediante otros sistemas de navegación, realimentación de los sensores o por referencias externas. Por tanto, podemos distinguir dos tipos de sistemas, los de medición de la posición relativa y absoluta [32, p. 1].

La medición de posición relativa, también conocida como navegación por estima o “dead reckoning” [32, p. 1], puede ser: por odometría, utilizando encoders para medir la rotación de las ruedas; por efecto Doppler para medir velocidad; o por navegación inercial, que utilizan giroscopios, acelerómetros y brújulas magnéticas para medir las rotaciones y aceleraciones.

La medición de la posición absoluta la obtendremos mediante estaciones de transmisión, también llamadas balizas de radiofrecuencia (RF), como veremos más adelante.

6.1.1 Odometría

La odometría estima la pose del vehículo a partir del número de vueltas dadas por las ruedas. Debido a que integra el movimiento en el tiempo, se acumulan los errores. Es bastante simple, barato y con alta cantidad de muestreo. Las desventajas son las imprecisiones debido a deslizamiento de las ruedas, irregularidades del terreno, desgaste de ruedas, entre otras. En [1, pp. 31-32] se explica cómo calcular la trayectoria y cambio de orientación cuando se produce un desplazamiento.

En [28, pp. 36-45] y [6, pp. 168-177] se describen los principales sensores que se utilizan para odometría. Uno de ellos son los potenciómetros, basados en la diferencia de tensión al variar una resistencia lineal o rotatoria, según el desplazamiento que interese, mediante una pista móvil.

Otro son las máquinas síncronas, que transmiten información angular eléctricamente al girar un rotor con corriente alterna dentro de uno más estatores. El acoplamiento magnético entre ambos devanados varía en función del giro del rotor, obteniendo señales en alterna cuyas magnitudes definen el ángulo del rotor en un instante determinado. Una de las configuraciones más utilizadas son los “resolver”, que proporcionan tensiones proporcionales al seno y coseno del ángulo del rotor.

El tercer sensor y más utilizado son los codificadores (“encoders”) ópticos. Convierten un desplazamiento rotacional en una señal digital sin necesitar un convertidor analógico-digital, como sí ocurre en los otros dos sensores. Se basan en la interrupción de luz entre emisor y receptor debido a un disco intermedio con un patrón de zonas opacas y transparentes. Según si el receptor recibe luz del emisor o no podremos obtener información del giro de dicho disco. Podemos distinguir entre codificadores absolutos o incrementales.

En los absolutos el disco está codificado de tal manera que a cada posición angular del eje corresponde un único código. Según el número de pares emisor-receptor, la resolución será mayor o menor, pudiendo detectar mayores o menores desplazamientos del eje. La salida es una “palabra” digital que indica la posición actual. Pueden utilizarse varios códigos, siendo el más habitual el de Gray (Figura 31izq) ya que solo cambia un bit cada vez.

En los incrementales se obtiene el desplazamiento según el número de interrupciones del haz de luz. Estas se transforman en pulsos de ondas cuadradas. Podremos

incrementar la resolución por vuelta aumentando el número de pulsos, es decir, con un disco con más transiciones opaco-transparentes. Debido a que este método no indica el sentido de rotación, se puede poner otro par emisor-receptor desfasado 90°. Cuando el primer par se adelanta al otro, girará en un sentido y viceversa. Los incrementales son más baratos y utilizados en aplicaciones de mayor velocidad, pero requiere más circuitería [6, p. 173 y 175]. En ocasiones se añade una tercera pista con sólo una zona opaca que permite definir una posición angular de referencia inicial (Figura 31dcha).

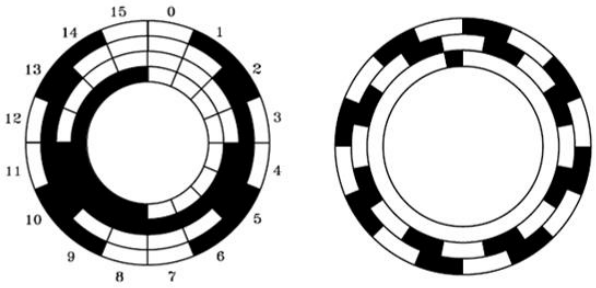


Figura 31: Encoders
Izq: absoluto, dcha: incremental [8, pp. 211-212]

6.1.2 Efecto Doppler

El principio de Doppler se basa en obtener la velocidad de un movimiento al emitir una onda sobre una superficie en movimiento relativo con el emisor y observar el desplazamiento de frecuencia de la onda reflejada [28, pp. 45-47], [6, pp. 185-186], [1, pp. 32-33]. Estos sensores son muy utilizados en vehículos aéreos y marítimos. Los primeros utilizan radiofrecuencia reflejando las microondas en la superficie de la Tierra. Los segundos emplean energía acústica que se refleja en el fondo del mar. En estos dos casos se colocan cuatro sensores a 90º entre sí y con el mismo ángulo de inclinación hacia abajo respecto al plano horizontal. Sin embargo, implica un alto coste.

En vehículos terrestres y robots se suele utilizar un sensor orientado en la dirección del movimiento hacia abajo con un cierto ángulo respecto al terreno, generalmente 45º (Figura 32). La velocidad relativa del suelo V_A se deriva de la velocidad medida V_D siguiendo la siguiente ecuación:

$$V_A = \frac{V_D}{\cos \alpha} = \frac{c \cdot F_D}{2 \cdot F_o \cdot \cos \alpha}$$

siendo:

- V_A : velocidad actual del terreno a lo largo del camino
- V_D : velocidad Doppler medida
- c : velocidad de la luz
- F_D : desplazamiento en frecuencia observado (desplazamiento Doppler)
- F_o : frecuencia de transmisión
- α : ángulo de incidencia

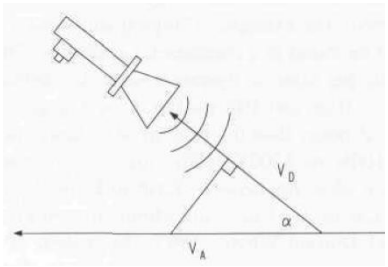


Figura 32: Efecto Doppler [28, p. 47]

Debido a que existirán ondas reflejadas de distintas frecuencias, deberemos hacer un filtrado previo para seleccionar las frecuencias más representativas y medir F_D. Se producen errores en la velocidad real del terreno debido a: interferencias de los lóbulos laterales de la respuesta en frecuencia ya que no toda la energía emitida lleva la misma dirección, componentes verticales de velocidad creadas por movimientos verticales del robot debido a irregularidades del terreno e incertidumbre en el ángulo real de incidencia.

6.1.3 Navegación inercial

Los sistemas de navegación inercial estiman la posición y orientación del robot mediante medidas de aceleraciones y ángulos de orientación [1, pp. 33-35]. Podemos destacar los siguientes sensores: los acelerómetros, los giróscopos y las brújulas. Los dos primeros tienen la ventaja de ser auto-suficientes ya que no necesitan referencias externas.

Acelerómetros

Los acelerómetros son dispositivos ampliamente utilizados en la actualidad en multitud de aplicaciones: en máquinas para detectar vibraciones o terremotos, en móviles para conocer orientación e inclinación, en cámaras de fotos, en coches para

hacer saltar los airbags en caso de accidente, en portátiles para proteger el disco duro en caso de caídas, en videoconsolas, etc. [33]. Todo esto es posible gracias al avance de la manufactura de los sensores MEMS (Micro-Electro-Mechanical-System) que hablaremos posteriormente en los giróscopos.

En los acelerómetros la primera integración de las aceleraciones proporciona la velocidad y la segunda integración la posición. Sin embargo, si se cometen pequeños errores en la estimación de la aceleración, al integrar dos veces, dichos errores aumentan notablemente ocasionando una posición estimada poco precisa. Además, si la relación señal/ruido es pequeña debido a aceleraciones pequeñas, se complica dicha estimación. Por tanto, no se recomiendan para posicionamiento preciso en periodos de tiempo largos [30, pp. 5-6].

El principio de funcionamiento del dispositivo explicado en [1, p. 34] se basa en una masa *m* vinculada al robot de masa *M* mediante un resorte de constante elástica *k*. Una condición para el correcto funcionamiento se basa en que la masa del robot sea mucho menor que la del dispositivo: $M << m$. Al moverse el robot, por la ley de Hooke:

$$F = m \cdot a = k \cdot x$$

Siendo *a* la aceleración del robot móvil y *x* la deformación del resorte debido a la fuerza *F.* Obtenemos la aceleración *a*, velocidad *u* y longitud de la trayectoria curvilínea *s* como:

$$a = \frac{k}{m} \cdot x \quad \rightarrow \quad u = \int a \cdot dt \quad \rightarrow \quad s = \int u \cdot dt$$

Existen diversas formas de construir acelerómetros: mecánicos, piezoeléctricos, de efecto Hall, de condensador, MEMS [34, pp. 11-13]… pudiendo ser de 2 ejes o de 3 ejes. Podemos utilizar dos acelerómetros de 2 ejes colocados perpendiculares entre sí para obtener las aceleraciones en 3 ejes. En [33] se describe el efecto piezoeléctrico, basado en estructuras cristalinas microscópicas que se ven "estresadas" ante fuerzas de aceleración. Estos cristales crean un voltaje y el acelerómetro interpreta dicho voltaje para determinar velocidad y orientación. En [6, p. 191] se describe más detalladamente otra forma de construir un acelerómetro pero que aquí no explicaremos.

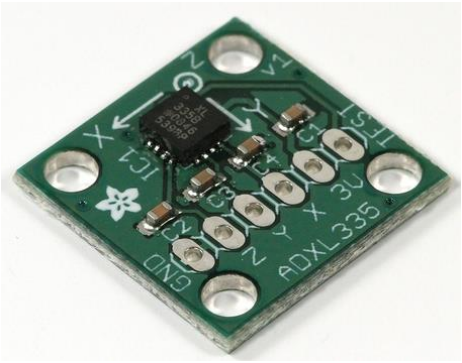


Figura 33: Acelerómetro ADXL335
(http://www.evilmadscientist.com/2009/basics-updated-using-an-accelerometer-with-an-avr-microcontroller/)

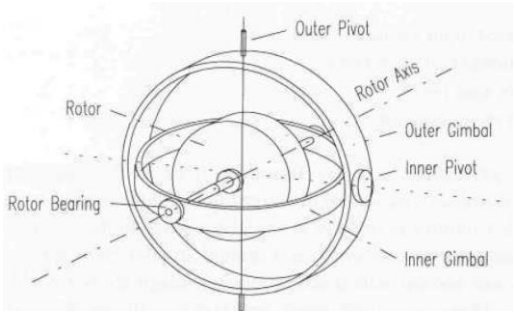
Giróscopos

Estos sensores permiten obtener la orientación del robot móvil sin necesidad de una referencia externa. No necesitan campos geomagnéticos y no son afectados por campos magnéticos locales. Principalmente podemos clasificarlos en tres tipos: mecánicos, ópticos y MEMS (Micro-Electro-Mechanical-System).

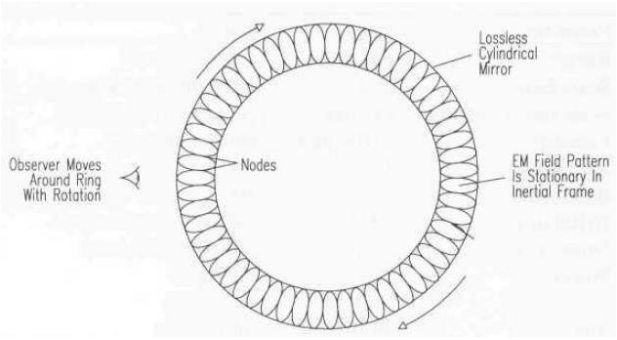
Los mecánicos están constituidos por un volante o rotor que gira suficientemente rápido alrededor de un eje con la masa distribuida en la periferia para tener un momento de inercia del eje de rotación alto. El rotor es accionado por un motor eléctrico suspendiéndose mediante un par de cojinetes de bajo rozamiento en cada extremo del eje para evitar fricciones. Estos cojinetes están soportados por un anillo circular, anillo gimbal interno, que a su vez pivota sobre un segundo juego de cojinetes unidos rígidamente a otro anillo gimbal externo. Por tanto, habrá tres ejes: el eje de rotación del volante, un eje perpendicular a éste y un tercer eje perpendicular a los dos anteriores, definiendo el pivote externo (Figura 34a) [6, p. 188].
Cuando el eje de rotación del rotor se suspende en una estructura que le permite variar libremente su orientación, aunque dicha estructura gire en el espacio, la dirección del eje del rotor se mantiene constante por el equilibrio de momentos angulares. Además de rozamientos despreciables y la masa concentrada en la periferia, la velocidad de giro debe ser alta para no perder su orientación inicial [6, p. 189].

Si un observador en la Tierra ve el eje de rotación, observará un movimiento aparente debido a la rotación de la propia Tierra. Sin embargo, si dicho eje se coloca en la dirección del eje de giro de la Tierra, no se produce el movimiento aparente. Esta propiedad es la utilizada en el girocompás, una configuración especial que busca el

norte. Cuando el eje de giro apunta al norte, no debe producirse ninguna inclinación ya que es insensible a la rotación de la Tierra. Si se produce, la medida proporciona el ángulo de orientación con el norte [6, p. 189].



a) [28, p. 364]



b) [28, p. 372]

Figura 34: Giróscopos
a) De volante con dos ejes b) Óptico ideal

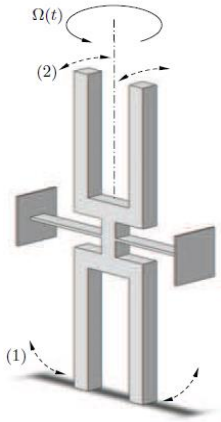
El giróscopo óptico explicado en [28, pp. 371-373] [6, pp. 190-191] consiste en dos haces de láser en direcciones opuestas y que forman un circuito cerrado. Los patrones de interferencia constructivos y destructivos formados dividiendo y mezclando porciones de los haces se pueden usar para determinar la velocidad y dirección de rotación del dispositivo. El anillo láser idealmente se podría pensar como un espejo toroidal sin pérdidas en la reflexión de la luz. Los haces de luz que se propagan en sentidos opuestos se refuerzan unos y otros creando una onda estacionaria con picos y valores nulos de intensidad (Figura 34b).

Se crean así franjas de luz y oscuridad equivalentes a los patrones de los discos de los codificadores ópticos. Por tanto, se podrían contar por un detector que gira mientras el espejo se mantiene fijo. En cada vuelta el detector vería un número de picos igual a dos veces la longitud del camino del haz dividida por la longitud de onda de la luz. Como esto es algo teórico y no factible, en [28, pp. 373-390] se explican 5 configuraciones que sí lo son. Un ejemplo de un giro láser disponible comercialmente es el “Autogyro Navigator” de Andrew Corp. de fibra óptica [30, pp. 6-7].

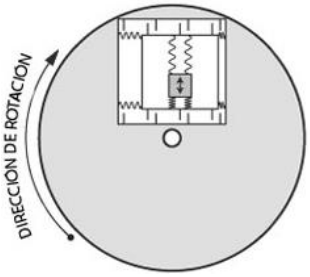
Por último, basándome en [35, pp. 255-256,259-262], los MEMS o “Micro-Electro-Mechanical-System” ([34], [36]), han hecho posible miniaturizar, disminuir el coste y la potencia de los sensores tradicionales. Esto ha permitido que junto a los acelerómetros de este tipo, puedan ser utilizados en gran variedad de aplicaciones diarias e industriales, citadas con los acelerómetros, que requieran una solución integrada para procesar movimientos e inercias.

Debido a las limitaciones de las técnicas de micro-fabricación, en lugar de miniaturizar los sensores tradicionales, que además perderían precisión, se ha optado por buscar principios de funcionamiento diferentes que se adapten mejor a la micro-escala. Para ello, se detecta los movimientos de un cuerpo vibratorio producidos por una fuerza aparente, fuerza de Coriolis, que aparece en el movimiento relativo del cuerpo vibratorio y el marco o estructura del sensor. Un giróscopo que utiliza este principio se llama “Giróscopo Vibrante de Coriolis”.

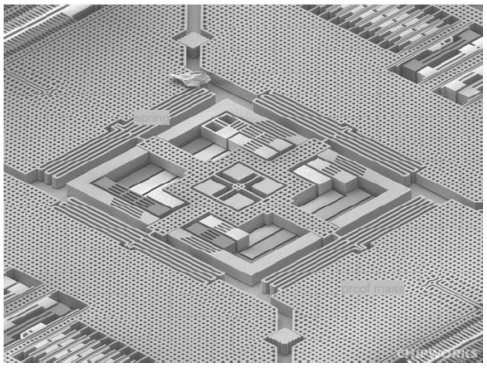
Dichos dispositivos están constituidos por una estructura mecánica con al menos dos modos de vibración. El funcionamiento normal consiste en excitar un modo de vibración a una amplitud predeterminada, y detectar las vibraciones inducidas por la fuerza de Coriolis en los demás modos debido al movimiento relativo. Hay una gran variedad de diseños, siendo algunos: vigas (voladizos), horquillas, placas, masas con resortes…



a) [35, p. 260]



b)



c) [34, p. 16]

Figura 35: Giróscopos MEMS

a) Horquilla doble; (1) primer modo vibración, (2) respuesta vibratoria por Coriolis b) Masa con resortes c) De 2 ejes creado por Chipworks para iPhone 4

b) (*http://www.sapiensman.com/tecnoficio/electricidad/electricidad_del_automotor19.php*)

Las horquillas, por ejemplo, tienen unas masas que oscilan con la misma amplitud pero en sentidos opuestos. En funcionamiento normal, los brazos vibran en anti-fase en el plano de la horquilla. Cuando el sensor rota, los brazos empiezan a oscilar en la

dirección perpendicular al plano, generando pares que excitan el modo torsional del vástago, pudiendo medirse la velocidad angular. Podemos encontrar simples, dobles (Figura 35a) u horquillas múltiples.

La masa con resorte consiste en una masa micro-mecanizada conectada a un marco interior mediante resortes, que a su vez está conectada a la placa del circuito por unos segundos resortes ortogonales. La masa se mueve continuamente sobre el primer conjunto de resortes. Cualquier rotación del sistema hará que el marco interior se desplace lateralmente debido a la fuerza de Coriolis. Este desplazamiento se mide por unos "dedos" sensores capacitivos de Coriolis que al aproximarse o alejarse varía la capacidad. Esto se puede convertir en una detección de la magnitud y dirección de la velocidad angular del sistema (Figura 35b).

Como conclusión de la navegación inercial, podemos encontrar unidades de medida inercial, IMU (Inertial Measuremente Unit) [35, p. 254], que incluyen acelerómetros y giroscopios, permitiendo medir aceleraciones, velocidades angulares, cambio de orientación… en los 6 grados de libertad.

Brújulas magnéticas

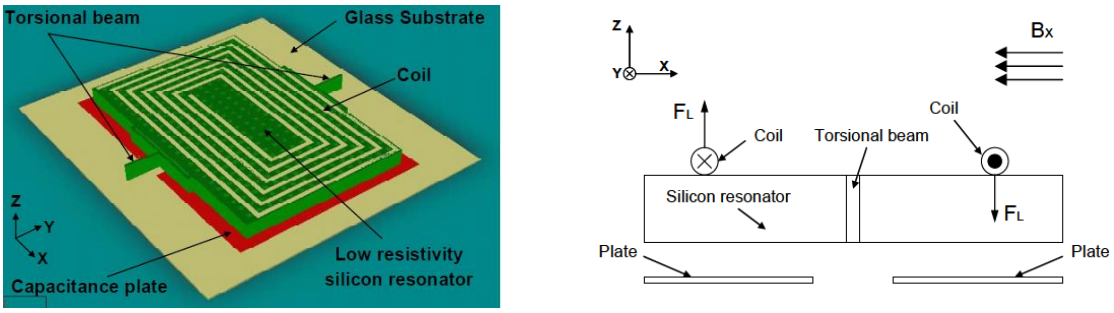
Además de los giróscopos, las brújulas magnéticas permiten obtener la orientación y velocidad angular relativa del robot. Se desea buena precisión para que los errores acumulados por la orientación en la navegación por estima sean los más pequeños posibles ya que tienen una gran influencia.

A la intensidad del campo magnético la llamaremos "densidad de flujo magnético" B, midiéndose principalmente en Tesla (T) o en Gauss (G), siendo 1 Tesla = 10^4 Gauss. La intensidad media del campo magnético de la Tierra es de 0.5 Gauss, representándose como un dipolo que varía con el tiempo y el espacio. Está inclinado unos 11º respecto al eje de rotación de la Tierra, llamándose "declinación" [28, p. 327]. Los instrumentos que miden campos magnéticos se llaman "magnetómetros", interesándonos para la navegación de robots los que midan el campo magnético terrestre. En [28, pp. 327-357] se clasifican en: mecánicos, de saturación (fluxgate), magneto-inductivos, de efecto Hall, magneto-resistivos y magneto-elásticos. En dicho libro se explican cada uno de ellos. Además, se han creado sensores de campo magnético en MEMS siguiendo el principio de funcionamiento del efecto Hall, magneto-resistencia y de saturación.

Este tipo de dispositivos poseen una serie de desventajas: perturbación del campo magnético terrestre debido a otros objetos magnéticos o estructuras hechas por el ser humano, limitaciones en el ancho de banda de las brújulas electrónicas y su susceptibilidad a vibraciones. Esto conlleva a utilizarse con poca frecuencia para navegación de robots dentro de instalaciones.

Sin embargo, se ha desarrollado otro nuevo tipo que no requiere materiales magnéticos y más fáciles de fabricar basados en la fuerza de Lorentz, que procederé a explicar un poco basándome en [37] y mostrado en la Figura 36.

El dispositivo consiste en una estructura de silicio de baja resistividad suspendida sobre un substrato de vidrio mediante dos vigas torsionales ancladas a dicho substrato. Encima de la estructura de silicio colocamos una bobina de excitación con muchas vueltas. Además, colocamos unas placas de capacidad de oro directamente en el substrato de vidrio formando un condensador con el silicio resonador (Figura 36a).



a) [37, p. 6952] b) [37, p. 6953]

Figura 36: MEMS magnetómetro
a) Esquema b) Principio de funcionamiento

El principio de funcionamiento puede observarse en la Figura 36b. Introducimos una corriente continua (DC) I en la bobina excitadora y si hay una densidad de flujo magnético B_x en la dirección *x* de la figura, para una vuelta de la bobina excitadora de longitud L_c en la dirección *y*, aparecerá una fuerza de Lorentz perpendicular al plano del silicio:

$$F_L = I \cdot L_c \cdot B_x$$

Como la fuerza es producida en ambos lados del resonador y con direcciones opuestas, se producirá un par constante que torsiona las vigas de sujeción.

Si ahora introducimos una corriente alterna en presencia del mismo campo magnético, el resonador de silicio vibrará en torno a las vigas torsionales debido a las direcciones alternantes de las fuerzas de Lorentz. Seguirán las siguientes ecuaciones:

$$i = I_0 \cdot \sin(2\pi f)$$

$$F_L = i \cdot L_c \cdot B_x = I_0 \cdot L_c \cdot B_x \cdot \sin(2\pi f)$$

Cuando la frecuencia *f* de la corriente es igual a la frecuencia de resonancia de la estructura de silicio, la amplitud de la vibración aumentará en gran medida debido a la resonancia y el factor de calidad elevado de la estructura. Este último relaciona en un elemento resonante la energía perdida debido a amortiguaciones. Si tiene un valor alto, no estará casi amortiguado, manteniendo las oscilaciones durante un periodo de tiempo largo. Por tanto, las amplitudes de las vibraciones producirán cambios de capacidad con las placas colocadas debajo. Esto se puede medir y transformar para reflejar el valor de la densidad de flujo magnético presente.

6.1.4 Estaciones de transmisión

Este tipo de sistemas, también conocido como balizas (“beacon”) de radiofrecuencia, RF, proporcionan la localización absoluta del vehículo en un área exterior suficientemente grande sin requerir estructuración del entorno. Por tanto, son adecuados para aplicaciones en las que el robot ha de moverse en entornos muy diversos y debe recorrer grandes distancias. Se basa en un receptor o transceptor, colocado en el vehículo y un conjunto de estaciones transmisoras de RF posicionadas en lugares conocidos y distantes [1, pp. 35-40]. Podemos diferenciar según si son estaciones fijas o móviles.

Estaciones fijas

Los sistemas de posicionamiento con estaciones fijas pueden ser de dos tipos: triangulación y trilateración.

La triangulación también se conoce como método de navegación hiperbólica o pasivo. El primer modelo fue desarrollado durante la Segunda Guerra Mundial por el MIT, y se llamó LORAN (Long RAnge Navigation), es decir, navegación de largo alcance. Se basa en la comparación de tiempos de llegada de dos señales idénticas transmitidas a la vez por transmisores de alta potencia localizados en posiciones de coordenadas conocidas [1, p. 36].

El transmisor “maestro” envía una señal de identificación al robot móvil y a otras estaciones trasmisoras que actúan de “esclavos”. Cuando éstas reciben la señal, la retransmiten añadiendo su propia señal de identificación, siendo también recibida por el robot un instante de tiempo después. La diferencia de tiempo entre la recepción de ambas señales en el robot se corresponde con las distancias entre las estaciones y el vehículo. Estas distancias se asocian a una constante *h* relacionada a una curva hiperbólica en la cual está el robot y cuyos focos son las estaciones de transmisión (Figura 37). La constante *h* viene dada por:

$$h = (\overline{AB} + \overline{BC}) - (\overline{AC})$$

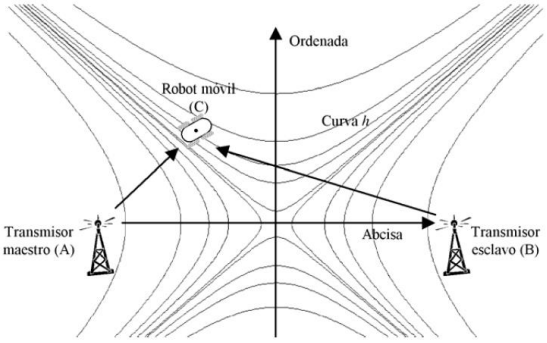


Figura 37: Triangulación LORAN [1, p. 36]

Para evitar ambigüedades en el posicionamiento del robot, se necesitan tres o más estaciones de transmisión, siendo una sola la maestra, para encontrar el punto de intersección de las líneas hiperbólicas [1, p. 37].

La trilateración, en cambio, se conoce como sistema activo, ya que miden el retardo de propagación para un número de transceptores localizados en lugares de coordenadas conocidas. El transceptor del robot móvil emite una señal de identificación que es captada por varios transceptores fijos, devolviéndola al añadir su propio código de identificación. El retardo entre el instante que el móvil emite la señal y en el que recibe la respuesta de las estaciones fijas determina la distancia que separa al robot de las estaciones fijas. Se requieren al menos tres estaciones fijas para evitar ambigüedades en la localización del robot móvil [1, pp. 37-38]. Según [2, p. 101], las tecnologías actuales permiten localizar a un robot en el exterior con una exactitud de unos 5 cm para áreas de kilómetros.

Además, se puede utilizar el mismo sistema pero con ultrasonidos u ópticamente. Los primeros proporcionan una exactitud aceptable y de bajo coste. Sin embargo, debido al alcance corto de los ultrasonidos, son adecuados para pequeñas áreas de trabajo

y si no hay obstáculos intermedios que interfieran en la propagación de la señal. Los ópticos requieren en cambio, mantener constantemente la visibilidad entre el robot y la baliza [1, p. 38].

Estaciones móviles

Estos sistemas de posicionamiento son conocidos como Sistemas de Navegación Global por Satélite (GNSS) ya que utilizan los satélites como estaciones móviles. Es una tecnología utilizada mayormente para navegación en exterior y no disponible en ambientes muy cerrados. Al principio utilizaban el efecto Doppler para observar el cambio de frecuencia experimentado por las señales de radio transmitidas por dichos satélites. Sin embargo, en la actualidad se utiliza el sistema Navstar-GPS (Global Positioning System), el cual emplea una constelación de 24 satélites, más tres de reserva, orbitando la Tierra cada 12 horas a una altura aproximada de 20200 Km [1, pp. 38-39].

Cada uno de estos satélites transmite dos señales de radio de alta frecuencia de espectro ensanchado ("spread spectrum") en las que se codifican la información sobre el instante en que la señal fue transmitida, información orbital… El receptor puede calcular mediante trilateración y con la distancia exacta a al menos tres satélites, la altitud, latitud y longitud del robot de forma casi instantánea y continua, ya que tarda entre 30 y 60 ns. Este sistema tiene que solventar cuatro desafíos tecnológicos:

- La sincronización de los relojes entre cada uno de los satélites y los receptores GPS.
- Localización exacta en tiempo real de los satélites.
- Medición exacta del tiempo de propagación de la señal.
- Buena relación señal-ruido para una operación eficaz en presencia de posibles interferencias.

En [28, p. 405], además de explicar los anteriores, se cita otro desafío relacionado con la teoría de la relatividad general de Einstein, según la cual, el tiempo transcurre más lentamente cerca de un cuerpo con gran masa como la Tierra que a elevadas alturas. Si no se tiene en cuenta, este fenómeno provoca errores de posicionamiento grandes.

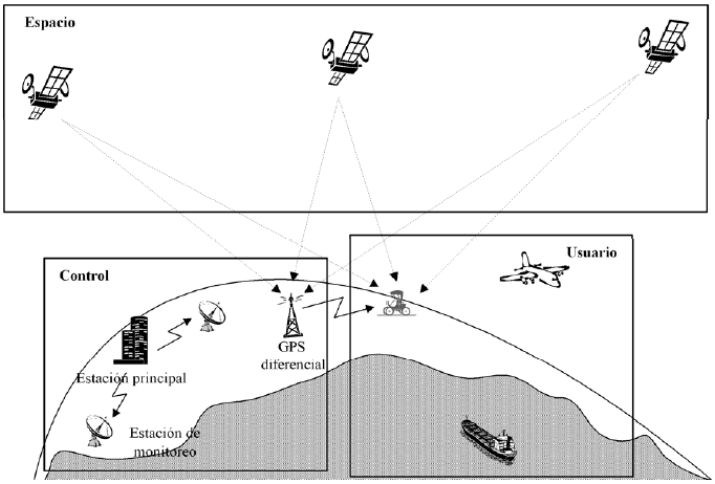


Figura 38: Sistema GPS tiene tres partes: Espacio, Control y Usuario [1, p. 39]

Se puede mejorar la precisión mediante el empleo de GPS diferencial (DGPS), basado en un segundo receptor GPS, normalmente una estación fija con coordenadas conocidas, relativamente próximo al robot (Figura 38). Como ambos están cercanos entre sí en relación a la distancia a los satélites, ambos recibirán los mismos errores de posicionamiento debido a interferencias, fenómenos meteorológicos, etc. Esto permite una corrección diferencial que se pasa al robot eliminando los efectos no deseados [28, pp. 412-413].

6.1.5 Percepción del entorno

Los estimadores de posición basados en la percepción del entorno utilizan sensores exteroceptivos, es decir, que obtienen información de él para compararla con otros datos o un modelo conocido del entorno. Como vimos en la introducción de este capítulo, se pueden clasificar en dos tipos según si son pasivos o activos.

Los sensores pasivos sólo captan energía que ya existe en el medio. Los sistemas más empleados son los sensores infrarrojos y las cámaras CCD y CMOS. Podemos encontrar abundante información de éstos últimos en [2, pp. 117-145], explicando la diferencia entre ambas tecnologías, problemas que deben tenerse en cuenta como la calibración, profundidad, enfoque, nitidez, flujo óptico, etc. [28, pp. 106-134]. Éste es un campo muy amplio y con abundante bibliografía llamado Visión Artificial, que permite procesar imágenes, extraer información y reconocer objetos, entre otros objetivos. Mediante la visión se pueden utilizar marcas o balizas especiales, reconocibles por el robot, dispuestas para ayudar a la navegación de éste mediante triangulación u otros sistemas si puede detectar tres o más marcas [1, pp. 40-42].

Pueden ser naturales, ya existentes en el ambiente, o artificiales, como bordillos, aristas entre paredes y suelo, entre paredes y techo… El robot debe conocerlas y saber su posición en el área de trabajo para poder utilizarse.

Los sensores activos, o "Time of Fligth" (TOF), miden el tiempo que tarda un pulso de una energía emitida por el robot, como luz o ultrasonidos, en viajar desde el emisor hasta un objeto reflectante y volver al receptor montando en el mismo robot. Esto permite obtener directamente medidas de las distancias al entorno: $d = v \cdot t$, siendo *d* la distancia, *v* la velocidad de propagación y *t* el tiempo transcurrido en ida y vuelta [28, p. 139].

Estos sensores aprovechan la naturaleza de este tipo de energía que viaja en línea recta. Esto implica que la señal devuelta por el objeto sigue el mismo camino que la emitida, es decir, coaxialmente, pudiendo emitirse y recibirse por un mismo dispositivo. Sin embargo, según [28, pp. 140-141] pueden producirse ciertos errores debido a:

- Variación en la velocidad de propagación en el caso de energía acústica, debido a que la velocidad del sonido está influencia por el cambio de temperatura y la humedad. Para la energía electromagnética se puede despreciar este fenómeno.
- Incertidumbres en la obtención del tiempo exacto del pulso reflejado, debido a que la señal reflejada puede tener intensidad muy distinta de la emitida, influyendo en el tiempo de subida del pulso detectado, y en caso de un umbral fijo, los objetos menos reflectantes aparecen más lejanos.
- Inexactitudes en la circuitería que mide el tiempo de viaje de la energía, ya que en el caso de energía electromagnética que viaja a la velocidad de la luz, requiere circuitería con una sincronización de tiempo por debajo de los nanosegundos. Esto aumenta su coste para aplicaciones que requieren elevada exactitud.
- Interacción de la onda con la superficie del objeto, ya que dependiendo de las características de dicha superficie o el ángulo de incidencia, sólo una pequeña parte de la energía emitida es devuelta en la misma dirección. El resto traspasa dicha superficie o es reflejada en otras direcciones.

Entre los sistemas utilizados con más frecuencia encontramos los sensores de ultrasonidos y los sensores láser. Los primeros son la técnica más común para robots móviles en interiores debido a su bajo coste y la facilidad de uso. En [28, pp. 141-150] se explican diversos sensores de ultrasonidos fabricados, siendo sólo unos pocos de todos los disponibles en el mercado y que han ido mejorándose en estos años. Uno de ellos son los transductores cerámicos que son similares a los cristales de cuarzo que resuenan a una frecuencia de resonancia y a otra de anti-resonancia. La transmisión es más eficiente a la frecuencia de resonancia mientras que la recepción óptima es a la de anti-resonancia.

Los sensores láser, también conocidos como radares láser o “lidar”, emiten energía láser en una rápida secuencia de ráfagas dirigidas directamente al objeto deseado. Siguiendo la ecuación antes citada, mide el tiempo requerido para un determinado pulso para ir y volver basándose en la velocidad de la luz. En [28, pp. 150-164] vienen algunos ejemplos de estos sensores, siendo unos pocos de todos los comercialmente disponibles y que se han ido mejorando con el tiempo.

Además de todos estos sistemas ya vistos, se ha utilizado otro tipo de sistema de navegación: la trayectoria de guía o “guidepath”. Consiste en una línea o marca continua tal que el robot la siga con un sensor muy próximo a ella. Se puede implementar mediante principios electromagnéticos, ópticos, térmicos o químicos [1, p. 42]. Por tanto, este sistema de navegación solo sirve para trayectorias predefinidas.

Por último, cabe decir que existen otros muchos sensores que no describiremos con aplicaciones básicas y en ocasiones, imprescindibles. Algunos ejemplos son los sensores de tacto, medidores de fuerza y par, térmicos, etc.

7 PLANIFICACIÓN DE RUTAS Y MOVIMIENTO

La planificación es una habilidad humana de gran complejidad destinada a organizar el tiempo y el espacio. Esto nos permite aprovechar mejor el tiempo y los recursos de los que disponemos. Dentro la gran variedad de aplicaciones que podemos encontrar, una de ellas es la planificación de caminos y rutas para movernos de un punto a otro de forma factible. Además, alguno de los factores más deseados es realizarlo en un tiempo, distancia o coste mínimo. La mayoría de los sistemas de navegación calculan varias alternativas de ruta entre los puntos proporcionados dando la posibilidad de elegir aquélla que más nos interese según la función que queramos minimizar.

En este capítulo introduciremos primero el espacio de configuraciones, el cual permite tener en cuenta las características geométricas del robot para poder evitar los obstáculos en su movimiento por el entorno. Posteriormente, explicaremos una serie de algoritmos para la planificación de rutas entre dos puntos.

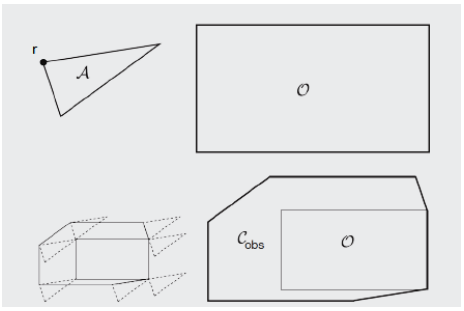
7.1 ESPACIO DE CONFIGURACIONES

Nos basaremos en [6, pp. 379-382,389-392] y [38, pp. 6-16] para explicar la representación del espacio de configuraciones (también [8, pp. 525-532]). Este modelo permite transformar los obstáculos y los límites del espacio presentes en el entorno del robot para poder representar éste como un punto y que a la vez se tenga en cuenta la geometría de dicho robot. Así, el problema se reduce a encontrar el camino desde la posición inicial a la final para un punto de referencia del móvil sin preocuparnos de riesgo de colisiones. Esto es debido a que los obstáculos crecen tal que sus nuevos límites definen el lugar de las regiones prohibidas.

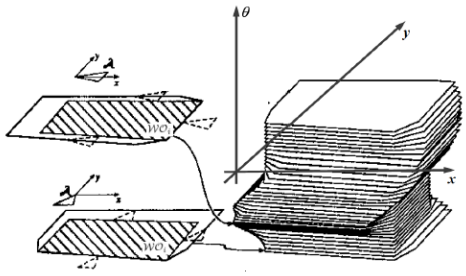
Consideraremos el caso de objetos poligonales en un plano, incluido el móvil representado con la letra A, como puede observarse en la Figura 39a. En dicho móvil definimos un punto de referencia *r*, cuyo movimiento será el considerado en lugar de todo el objeto A. La configuración “q” del robot es una especificación de las posiciones de todos los puntos del robot relativas a un sistema de coordenadas fijo. El espacio de configuraciones será de dimensión 2 si el móvil se traslada, y de dimensión 3 si además puede rotar, expresado normalmente como un vector de posiciones y

orientaciones: $q = (x, y, \theta)$. En 3D sería con los seis parámetros siguientes: $q = (x, y, z, \alpha, \beta, \gamma)$.

Para el caso en el que A no pueda rotar, los límites del nuevo obstáculo los obtendremos deslizando el objeto A alrededor de $\mathcal{O}$ manteniendo fija su orientación como se puede ver en la Figura 39a. Asimismo, se realizaría idéntico procedimiento en torno a los límites del espacio de trabajo. Los movimientos de *r* que no produzcan colisión en el espacio creado tampoco lo harán trasladando el objeto A por el espacio de trabajo inicial.



a) [38, p. 11]



b) [38, p. 13]

Figura 39: Espacio de configuraciones
a) Orientación fija b) Orientación variable

Si el móvil puede rotar, entonces el espacio de configuraciones no estará contenido en el plano sino que será multidimensional. Se obtendría deslizando el robot alrededor de las regiones de los obstáculos, con una orientación determinada en cada uno de los planos superpuestos, como puede verse en la Figura 39b (ver también [23, pp. 508-509]). En ambos casos conviene aumentar los límites de los obstáculos un poco más para evitar colisiones producidas por incertidumbres en la estimación de la posición del móvil o en la ejecución del camino.

Generalmente se trabajan polígonos convexos debido a que normalmente los algoritmos que veremos más adelante trabajan con dicha hipótesis. Sin embargo, cuando los objetos son polígonos no convexos, se pueden descomponer en polígonos que sí lo son y considerando como referencia un punto en la línea de unión de ellos. El espacio de configuraciones será la suma de los obstáculos generados por cada uno de los polígonos del objeto (véase [6, pp. 380-381] para más información).

7.2 ALGORITMOS DE PLANIFICACIÓN

En este apartado estudiaremos diversos algoritmos de planificación que se han ido desarrollando y mejorando durante las últimas décadas. En todos ellos se buscará llegar desde un punto inicial hasta uno final evitando los obstáculos presentes en el mapa mediante el camino más corto. En lugar de punto inicial y final podríamos hablar de configuración inicial y final con robots móviles si tienen una posición y orientación determinadas en dichos sitios. Podemos distinguir varios tipos de algoritmos según se basen en geometría, teoría de grafos, mallas, probabilidades o funciones potenciales.

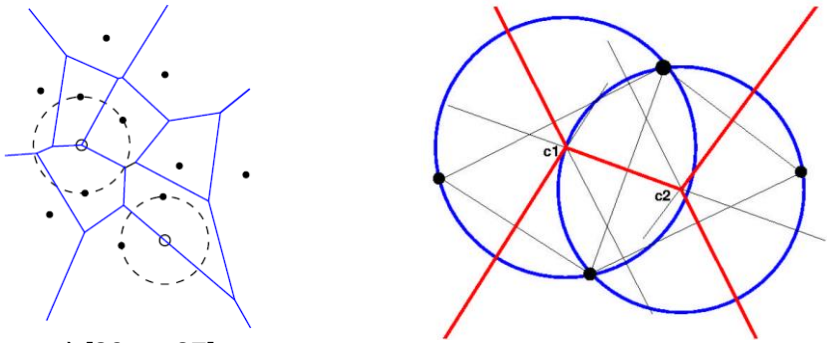
7.2.1 Basados en geometría

Dentro de esta categoría destacamos el método de Voronoi, cuyo concepto lleva utilizándose desde hace muchos años y con abundante bibliografía [12, pp. 97-99], [39], [40, pp. 27-37], [41], [32], [42], [38, pp. 20-23], [43, pp. 155-192], [23, pp. 117-130]. Comúnmente se hace una analogía con el fuego. Si tenemos un bosque y empezamos a incendiarlo simultáneamente en dos puntos del perímetro, el fuego irá avanzando hacia el centro de igual forma por ambos frentes. Llegará un momento en que se encuentren en el punto medio de dichos puntos de inicio.

Aplicado a puntos o nodos arbitrarios colocados en un plano, consiste en dividirlo en regiones poligonales, cada una asociado a uno de ellos. Estas regiones cumplen la propiedad de ser el lugar geométrico de todos los puntos más cercanos al nodo asociado a esa región que al nodo asociado a cualquier otra. Las fronteras entre dichas zonas forman el diagrama de Voronoi, siendo equidistantes a dos o más nodos [40, p. 28]. Por tanto, cada uno de los puntos de estas líneas son los centros de círculos vacíos en cuyo perímetro hay dos nodos. Cuando tres líneas del diagrama concurren entre sí, se llama vértice, y es el centro de un círculo vacío en cuyo perímetro hay tres o más nodos (Figura 40a) [39, p. 27].

En la Figura 40b podemos estudiar cómo se construye un diagrama de Voronoi basándonos en [40, pp. 27-32]. Una vez colocados los puntos negros, que son los nodos, se trazan las líneas finas que los unen y la mediatriz entre ellos. Estas mediatrices son perpendiculares a las líneas de unión entre nodos y cumplen la propiedad de ser el lugar geométrico de los puntos equidistantes a éstos. Las intersecciones de estas mediatrices determinan los vértices del diagrama de Voronoi:

c1 y c2. Si hacemos los círculos en azul con centros en dichos vértices, se comprueba que en su perímetro hay mínimo tres nodos y dentro del círculo no hay. Por último, dibujamos las líneas rojas gruesas a partir de los vértices y continuando sobre las mediatrices finalizando el diagrama.



a) [39, p. 27] b) [40, p. 28]

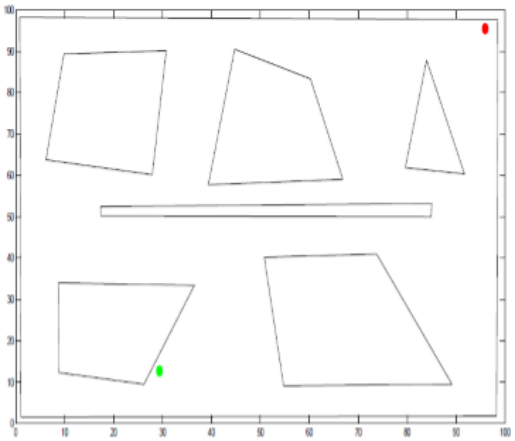
Figura 40: Diagrama de Voronoi
a) Círculos vacíos b) Construcción

Si en lugar de nodos aislados tratamos con obstáculos constituidos por segmentos formando polígonos, habrá que adaptar el método visto. En este caso conviene crear un marco externo a todo el recinto para obtener buenos resultados. Este marco puede tener cualquier forma siempre que incluya todos los obstáculos y los puntos inicial y final que deseemos.

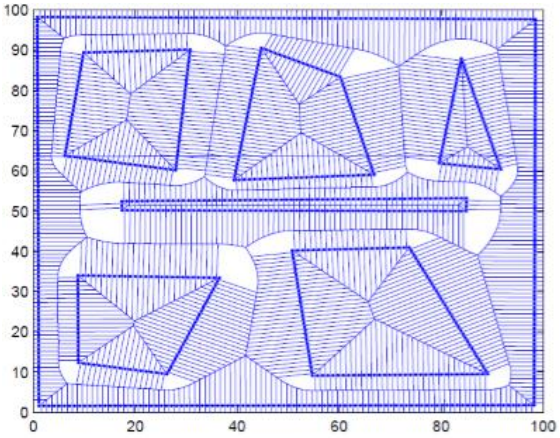
Los bordes de los obstáculos se pueden considerar como un conjunto de nodos alineados. Por tanto, un objeto poligonal se puede dividir en segmentos, cada uno de ellos formado por nodos. De acuerdo a [41], el comando “Voronoi” de Matlab sigue la siguiente metodología.

Una vez que tenemos la configuración de los obstáculos poligonales en el mapa (Figura 41a), los dividiremos en nodos y calcularemos el diagrama de Voronoi de todos ellos. La distancia entre nodos consecutivos viene dada por un parámetro Epsilon (Figura 41b). Cuanto menor sea este valor, más nodos habrá, creando un camino más suave, pero requiriendo mayor gasto computacional.

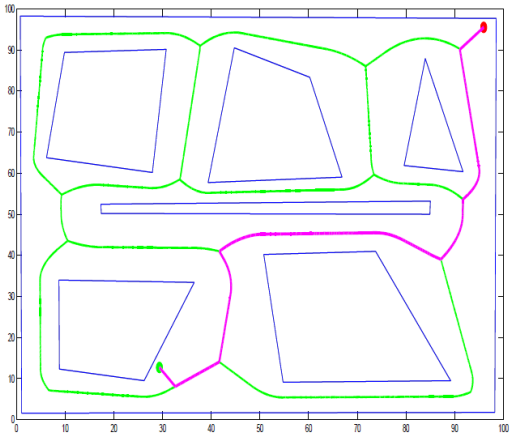
Podemos dividir las aristas del diagrama de Voronoi en dos categorías según sean generadas por nodos del mismo objeto o por distintos. Una vez que tenemos todas las aristas del diagrama, cada una creada por un par de nodos, se eliminan las aristas del primer tipo, es decir, las generadas por nodos del mismo objeto. Esto permite quitar las que cortan a los obstáculos o al marco exterior, quedando únicamente las aristas entre ellos.



a)



b)



c)

Figura 41: Voronoi

a) Obstáculos y puntos inicial y final b) Diagrama con Epsilon=1 c) Diagrama simplificado y ruta óptima [41]

Una vez realizado este proceso, es muy probable que los puntos inicial y final no estén incluidos en las aristas del diagrama. Esto implica que debemos encontrar la arista más cercana y unirlas con dichos puntos (Figura 41c). Si hay algún obstáculo en el camino que acabamos de tomar, deberemos buscar la siguiente arista más cercana [40, p. 32]. En [23, pp. 118-120] la accesibilidad, es decir, el acceso del punto de inicio al diagrama, se realiza alejándonos del obstáculo más cercano. Esto se realiza mediante un gradiente ascendente, lo que implica aumentar la distancia al obstáculo, hasta llegar a un punto del diagrama de Voronoi. La capacidad de salida desde un punto cercano a la meta será a la inversa, ir hacia dicho punto en la dirección que la distancia se haga menor.

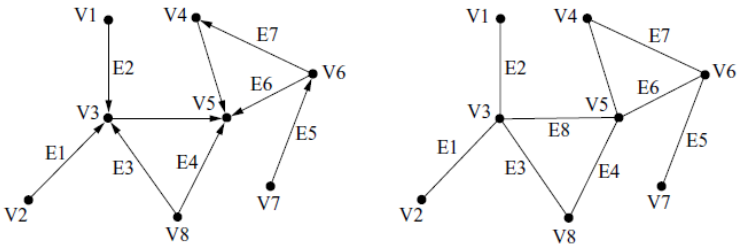
Según el algoritmo implementado, será mayor o menor el gasto computacional y el tiempo empleado. De acuerdo con la implementación de [40, pp. 28-37], la construcción del diagrama tarda O(nlogn), siendo n el número de puntos. Uno de los algoritmos más conocidos en este campo es el de Fortune o “Plane sweep” no explicado aquí, pero sí en [39, pp. 29-77], con un coste de O(nlogn) [41, p. 1].

Se han realizado multitud de estudios para la planificación basándose en el diagrama de Voronoi pero acudiendo a otros procedimientos. En [42] se construye dicho diagrama utilizando métodos de procesamiento de imagen, como la Transformación del Eje Medio (MAT) para aplicar un Método de Marcha Rápida (FMM).

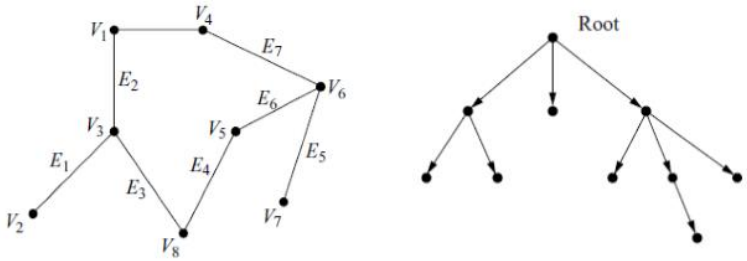
Al igual que con el resto de algoritmos veremos en este capítulo, convendría suavizar las aristas del diagrama en los bordes para que se obtenga una ruta más suave y no tan poligonal. Esto se podría alcanzar por medio de splines o curvas de Bezier [44].

7.2.2 Basados en grafos

Basándonos en [23, pp. 521-527], los grafos son conjuntos de vértices o nodos y aristas que representan, en planificación del movimiento, localizaciones destacadas y los caminos entre ellas respectivamente. Los denominaremos G=(V,E), siendo G para grafo, V para vértices y E ("edge") para las aristas. Si estas últimas poseen un valor numérico no negativo representa el coste de viajar entre los nodos adyacentes, y lo llamaremos peso. Cuando dicho trayecto se puede realizar sólo en un sentido, es decir, si se puede viajar del nodo V_1 al V_2 pero no al revés, la arista E_{12} es dirigida dibujándola con una flecha y pertenece a un grafo dirigido. Si se puede viajar en ambos sentidos serán dos aristas dirigidas E_{12} y E_{21}, lo dibujaremos como una línea simple y es un grafo no dirigido (Figura 42a).



a) [23, p. 522]



b) [23, p. 522]

Figura 42: Grafos
a) Izq: dirigido, dcha: no dirigido. b) Izq: ciclo, dcha: árbol

Un camino en un grafo es una secuencia de nodos adyacentes unidos entre sí por aristas. Un grafo es conexo si para dos nodos arbitrarios hay al menos un camino que los una entre sí. Un ciclo es un camino formado por un número de nodos, *n*, tal que el primero y el último son los mismos $V_1 = V_n$ (Figura 42b (izq)). En un grafo dirigido sólo se puede recorrer el ciclo en un sentido, mientras que uno no dirigido es indistinto.

Un árbol es un grafo dirigido sin ciclos con un nodo inicial, raíz, el cual no posee ninguna arista entrante. Se puede asemejar a un árbol genealógico, formado por nodos padre cuando tienen algún nodo hijo por debajo (Figura 42b (dcha)). El nodo raíz no tiene ningún nodo padre y un nodo sin hijos es un nodo terminal u hoja. Si quitamos algún nodo intermedio, es decir, que tenga hijos, se rompe la conectividad del árbol.

Por último, definiremos la longitud de enlaces como el número de aristas de un camino en un grafo. A diferencia de longitud del camino, no sabemos el coste o longitud de cada arista, solo su número.

Una vez definidos los grafos, estudiaremos algunos algoritmos de planificación de rutas a través de ellos, siendo aplicables con otros fines a diversas áreas de conocimiento, como la teoría de juegos [45, pp. 536-551].

Búsqueda en Amplitud

En primer lugar, veremos la Búsqueda en Amplitud o BFS (Breadth First Search) [23, p. 523], [8, p. 606], [38, p. 53], [45, p. 35] [46]. Al igual que en los demás casos, empezaremos la búsqueda en el nodo raíz hasta llegar al nodo meta u objetivo. En este primer algoritmo se visitan todos los nodos hijos de la raíz, luego todos los nodos hijos de éstos y así sucesivamente.

La implementación más común, se realiza utilizando una lista OPEN semejante a una cola, del tipo FIFO (First In First Out). Esto implica que se fuerza a que todos los nodos de misma longitud de enlaces respecto al inicio se visiten primero. Al principio solo incluye el nodo raíz marcado como visitado. Los demás nodos del grafo, G, se marcan como no visitados. En cada iteración del algoritmo se quita el primer nodo de OPEN y todos los hijos de ese se marcan como visitados e introducidos en dicha lista. La búsqueda termina cuando el nodo meta se introduce en OPEN o cuando la lista está vacía y ha fallado el proceso. En este algoritmo expandimos primero todos los nodos que son hijos de un mismo padre antes de expandir los hijos de algunos de ellos. Para

saber cuál es el camino final, utilizaremos un gradiente descendente de longitud de enlaces desde la meta hasta el inicio. Esto requiere almacenar en un array la longitud de enlaces de cada nodo hasta el inicio en el momento que visitamos cada uno de los nodos que visitamos.

Búsqueda en Profundidad

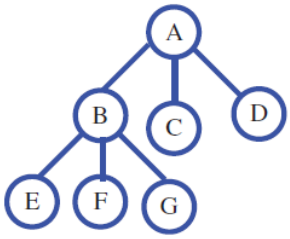
Otro algoritmo de este estilo es la Búsqueda en Profundidad o DFS (Depth First Search), tratado en [23, p. 523], [8, p. 606], [45, p. 36], [38, p. 53], [47]. Al igual que con BFS se empieza en el nodo raíz. Se elige entonces un nodo hijo y éste se expansiona eligiendo un hijo suyo y así sucesivamente hasta llegar a la meta o a un nodo hoja, es decir, sin hijos. En este último caso, se retrocede un nivel y se busca en profundidad otro nodo hijo no visitado hasta encontrar el nodo deseado o uno terminal. Se repite dicho proceso hasta que se encuentra el nodo meta o se han visitado todos los nodos.

La implementación en este caso se realiza con una lista OPEN del tipo LIFO (Last In First Out). En este caso se fuerza a los nodos de mayor longitud de enlaces respecto al inicio a visitarse primero. Inicialmente contiene el nodo inicial marcado como visitado. Cuando se inserta un nodo “j” en OPEN, el nodo “i” que ha permitido su inserción se memoriza. En cada iteración se extrae el primer nodo de OPEN; si es no visitado, se marca como visitado y se guarda en el árbol DFS la arista que conecta los nodos “i” y “j”. Posteriormente todos los nodos no visitados adyacentes a “j” se insertan en OPEN. La búsqueda termina cuando se introduce el nodo meta en OPEN o dicha lista queda vacía, fracasando el algoritmo. El árbol DFS contiene el camino solución del nodo inicial al objetivo, si existe, de igual manera que se hizo con la Búsqueda en Amplitud.

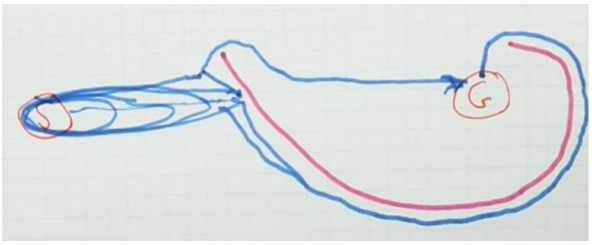
En [47] se describe detalladamente dicho proceso mediante el grafo de la Figura 43a y dos listas: OPEN que guarda lo que debes hacer, y CLOSE que graba lo que ya has hecho. El objetivo en este caso es encontrar un camino desde A hasta E. Al principio solo tenemos el nodo inicial A, y como no hemos hecho aún nada, lo añadimos a OPEN, quedando CLOSE vacía. Exploramos los nodos hijos de A, siendo B, C y D. Como ya hemos acabado con el A, lo quitamos de OPEN y lo añadimos a CLOSE. Ahora estudiamos el primer nodo de open, B, y como no es la meta, exploramos sus nodos hijos. Como ya hemos expandido B lo extraemos de OPEN y lo añadimos a

CLOSE. Además añadimos los nuevos nodos descubiertos: E, F y G al principio de OPEN. Expandimos el nodo E por ser el primero de la lista, y como ya es la meta, paramos. Agregamos E a CLOSE y ya tenemos la ruta: A → B → E.

Se puede observar que ambos algoritmos vistos son estrategias de recorrido, es decir, no utilizan ninguna información sobre el nodo meta, sino que van recorriendo el grafo hasta que dicho nodo se marca como visitado. Ambos algoritmos se dicen que son completos porque encuentran un camino si existe y si no, informan de fallo. El primero es óptimo ya que encuentra el camino de menor longitud porque siempre estudia los nodos desde los más cercanos al inicial hasta los más lejanos. Sin embargo, el segundo no lo es porque si hubiera dos metas, una en E y otra en D, encuentra la E que es de mayor longitud que la D [48].



a) [47]



b) Captura de [48]

Figura 43: Búsqueda en grafo
a) En profundidad b) Codiciosa

Búsqueda Codiciosa

A diferencia de los anteriores existen otros algoritmos que realizan búsquedas en función de información proporcionada por la posición de la meta. Este es el caso de la Búsqueda Codiciosa o GBFS (Greedy Best-First Search) [48], [23, p. 526]. Se basa en expandir aquellos nodos más próximos al nodo meta, y no de forma uniforme en todo el espacio como en los anteriores casos.

La estructura de los datos será una cola prioritaria en la que los nodos son colocados en la lista, ordenados según su distancia estimada a la meta, la cual es heurística. Esto conlleva a requerir menos cantidad de tiempo y espacio computacional debido a una búsqueda más focalizada. Sin embargo, hay casos en la que no descubre el camino más corto debido a que en ocasiones conviene alejarse un poco de la meta para superar un obstáculo y así obtener un camino más corto que rodeando todo el obstáculo aunque sus puntos sean más cercanos al objetivo. En la Figura 43b puede observarse que al llegar a la barrera, convendría retroceder un poco y subir que dar

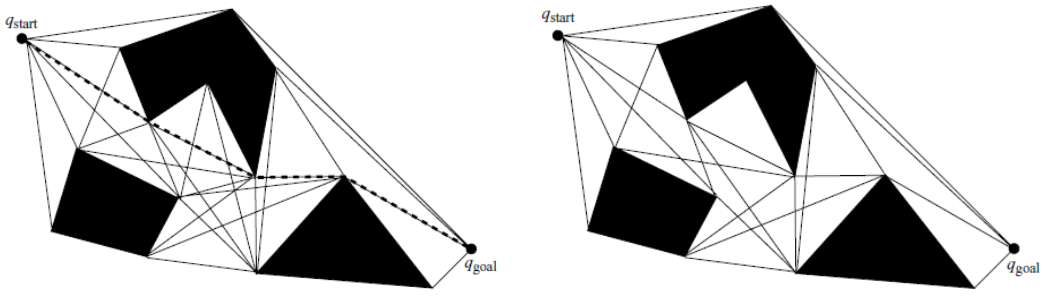
toda la vuelta por la parte inferior de dicho obstáculo. Sin embargo, este algoritmo no es capaz de ello ya que siempre va hacia los nodos de menor distancia a la meta.

Grafo de visibilidad

Otro tipo de algoritmo basado en grafos y el espacio de configuraciones es el método de Grafo de Visibilidad [23, pp. 110-117], [45, pp. 261-264], [38, p. 19], [49, pp. 5-9], [6, pp. 386-387]. Los nodos corresponden a los vértices de obstáculos poligonales, además de las configuraciones iniciales y finales. Las características definitorias de un mapa de visibilidad son que sus nodos comparten una arista si son visibles entre sí y que todos los puntos del espacio libre son visibles por al menos un nodo del mapa. Por tanto, cumple las propiedades de accesibilidad y capacidad de salida vistas en Voronoi para cualquier punto de inicio y final deseados.

Para construir el grafo primero uniremos todos los nodos visibles entre sí mediante líneas rectas, es decir, que se dibujen sobre el espacio libre (Figura 44a). Esto incluye cada par de vértices del mismo espacio de configuración de un obstáculo. Sin embargo, hay muchas líneas innecesarias, por lo que eliminaremos aquellas que no causen ninguna de estas dos situaciones:

- La línea sea tangente a dos obstáculos tal que ambos queden al mismo lado de la línea.
- La línea sea tangente a dos obstáculos pero que cada uno esté en un lado de dicha línea.



a) [23, p. 111] b) [23, p. 112]

Figura 44: Grafo de visibilidad
a) Completo b) Reducida

Por tanto, nos quedamos con un grafo de visibilidad reducida (Figura 44b) con nodos convexos y aristas o líneas tangentes. Este grafo permite encontrar una ruta más eficientemente si existe, por lo que es completo, y además ofrece el camino más corto posible. Por otro lado, tiene la desventaja de pasar muy cerca de los obstáculos, lo

cual puede ser problemático en un caso práctico cuando hay posibilidad de incertidumbres en las medidas, trayectorias, etc.

En [23, pp. 113-117] se trata la construcción del grafo mediante algoritmos de barrido con plano para obtener los nodos que sean visibles y detectar aquellas líneas que intersectan con otros obstáculos.

Descomposición en celdas

Por último, veremos un tipo diferente de representación del espacio libre en grafo, la descomposición en celdas [23, pp. 161-168], [8, pp. 536-541], [38, pp. 24-27], [2, pp. 264-267], [45, pp. 253-260]. Dividiendo dicho espacio libre en celdas, podemos construir un grafo de conectividad, cuyos nodos representan las celdas y los segmentos entre nodos indican que dichas celdas son adyacentes. Buscaremos en el grafo un camino desde la celda que contenga el punto de inicio hasta la celda que contiene la meta mediante el algoritmo de búsqueda que interese. Consecuentemente, obtendremos un "canal", que es la secuencia de las celdas contiguas de dicho camino (Figura 45).

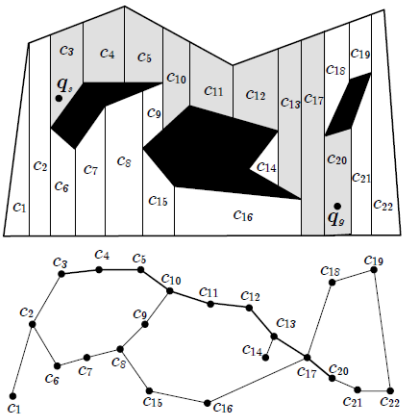


Figura 45: Descomposición en celdas y canal [8, p. 537]

La descomposición en celdas puede ser principalmente de dos tipos: la descomposición exacta y la aproximada. Para el primer caso utilizaremos el algoritmo de barrido de línea, basado en desplazar una línea vertical por el espacio de trabajo. Cada vez que encuentre un vértice de obstáculo, dibujamos una línea vertical en ambos sentidos respecto al vértice hasta que intersecta a otro obstáculo o alcanza la frontera de trabajo. Una vez obtenida la descomposición, colocamos nodos en los puntos medios de las líneas y trapezoides generados. Conectamos dichos nodos, y los puntos inicial y final del camino a los nodos más cercanos. Por último, utilizamos un algoritmo de búsqueda para encontrar el camino más corto (Figura 46). Al igual

que con otros algoritmos, se puede suavizar el camino mediante curvas de Bezier o técnicas de ajuste de curvas.

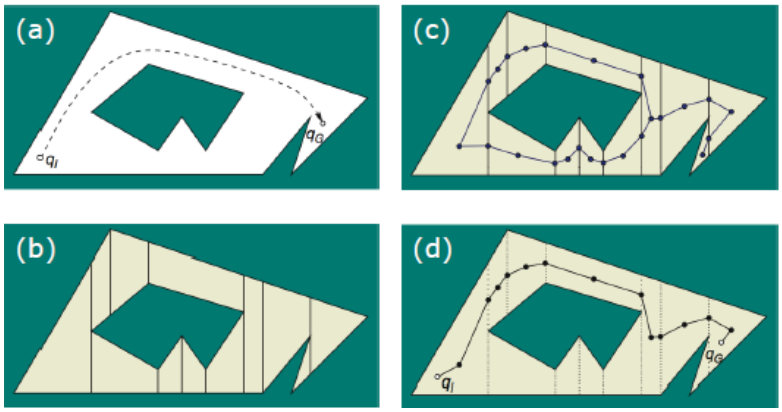


Figura 46: Descomposición exacta
a) Objetivo b) Celdas c) Nodos d) Ruta óptima [38, p. 25]

El caso de la descomposición aproximada se utiliza para crear celdas cuadradas eficientemente mediante un algoritmo recursivo [8, pp. 539-541]. Consideraremos que el límite del espacio de trabajo es cuadrado. Primero lo dividiremos en cuatro celdas iguales, pudiendo pertenecer a uno de los siguientes tipos:

- Celdas libres, en cuyo interior no hay obstáculos
- Celdas ocupadas, contenidas en región de obstáculos
- Celdas mixtas, que no están ni libres ni ocupadas

Construimos el grafo de conectividad, formado por nodos de las celdas libres y mixtas y los segmentos entre ellos. Buscamos el camino entre las celdas de los puntos inicial y final. Si existe, puede ser una secuencia de celdas libres, por lo que ya hemos encontrado la solución, o puede tener celdas mixtas. En este segundo caso volvemos a dividir cada celda mixta en cuatro, se clasifican de la misma forma que antes y buscamos el camino. Iteraremos dicho proceso hasta encontrar el canal libre (Figura 47) quedando una cuadrícula adaptada a la geometría del espacio libre.

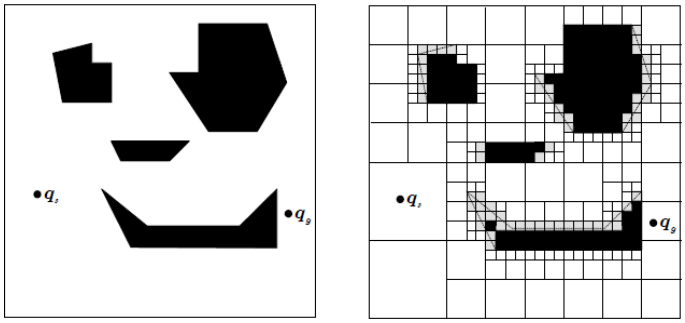


Figura 47: Descomposición aproximada [8, p. 540]

7.2.3 Basados en cuadrículas

Una cuadrícula es una representación del mapa formada por una matriz de dos dimensiones de elementos cuadrados llamadas casillas o píxeles [23, pp. 86-89]. En esta representación dejaremos en blanco las casillas libres, es decir, que no tienen ningún obstáculo, y sombreadas o negras aquellas ocupadas por un obstáculo. Podemos elegir si las casillas están relacionadas con sus vecinos por cuatro puntos: norte, sur, este y oeste, o por ocho puntos si también incluimos los vecinos de las diagonales.

Grassfire

El algoritmo más sencillo en esta categoría es el llamado "Grassfire" o "Bushfire" porque se asemeja al efecto que provoca el fuego al quemar hierba, extendiéndose en todas las direcciones. Consideraremos una conectividad de cuatro puntos. A la casilla que contiene la meta le daremos el valor 0. Después buscamos todas aquellas celdas vacías alejadas un paso del destino y les asignamos el valor 1. Todas las casillas alejadas 2 pasos de la meta les damos el valor 2 y así sucesivamente hasta rellenar toda la cuadrícula.

Por tanto, cada celda contendrá el valor mínimo de pasos que tiene que darse desde ese punto para llegar a la meta. Se puede observar que los números son radiados alejándose del destino como si fuera un fuego. Cuando el nodo inicial ya ha sido marcado con un valor, podemos construir el camino más corto a la meta moviéndonos a la celda con el menor valor de distancia (Figura 48). También podemos aplicar el algoritmo al revés, empezando por el inicio, y obteniendo la misma longitud final. Si dos vecinos tienen el mismo valor, podremos escoger aleatoriamente. Esto ocurre cuando el camino más corto no es único.

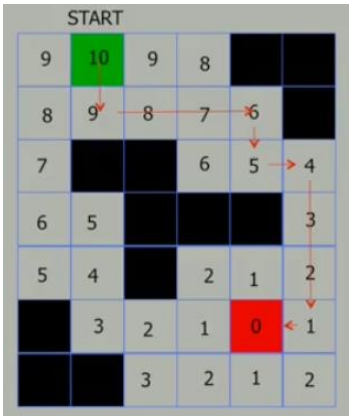


Figura 48: Grassfire [50]

La implementación será parecida a la de la Búsqueda en Amplitud. Primero daremos a todos las casillas libres un valor de distancia infinito. Creamos una lista vacía e introducimos la casilla de la meta con un valor de 0. Mientras la lista no esté vacía, ejecutamos los siguientes pasos. La primera casilla de la lista será la actual y la quitamos. Para cada celda adyacente a la actual vemos si su valor de distancia es infinito, y si es cierto, le damos un valor igual al de la actual más 1. Después añadimos dicha celda al final de la lista y continuamos iterando. Si no fuera infinito la celda ya ha sido visitada antes y no la modificamos. Finalizaría cuando la celda inicial posee un valor finito.

Sin embargo, si después del proceso la celda inicial no es marcada con un valor, significa que no hay camino posible para llegar desde el inicio hasta la meta. Por tanto, es un algoritmo completo encontrando el camino más corto posible, y si no, informa de fallo.

Los algoritmos de Búsqueda en Amplitud y Búsqueda en Profundidad ya vistos para grafos también se pueden implementar en cuadrículas. El grafo que resulta de representar dicha cuadrícula ya no será un árbol sino que expandirá en todas las direcciones, pero el concepto básico es el mismo. En lugar de usar un árbol con el coste de longitud de enlaces, será una matriz del mismo tamaño que el mapa de la cuadrícula, como puede verse en [23, pp. 525-526], [46, p. 1] y [51]. De aquí en adelante, hablaremos de estados, nodos o celdas indistintamente ya que las cuadrículas pueden transformarse en grafos.

Dijkstra

El algoritmo de Dijkstra comparte la misma base que la Búsqueda en Amplitud pero tiene en cuenta los costes de viajar de una celda a otra, o de un nodo a otro en caso de grafos [46, p. 2], [23, pp. 532-533], [52], [45, pp. 36-37,55-56], [50]. Por tanto, habrá una lista OPEN, un array con las celdas PARENTS y una matriz con los costes de distancia de cada celda al inicio, inicialmente infinito, excepto el inicio con valor 0.

Al empezar, la lista OPEN solo contiene el punto de inicio. Quitamos dicha celda de la lista, la ponemos en PARENTS, añadimos todos sus vecinos a OPEN y calculamos la distancia de cada uno. Después miramos cuál es la distancia menor, lo llamamos “actual” y lo quitamos de la lista OPEN.

Ahora estudiamos sus vecinos. Para cada uno de ellos calculamos su distancia comparando si su valor es mayor que la distancia del "actual" más el coste entre ambos. Si esto es así, significa que tenía valor infinito o previamente ya hemos calculado su distancia por otro camino. Por tanto, ahora podemos actualizar su coste a uno menor, siendo éste la longitud del "actual" más el coste entre ambos. Añadimos a PARENTS la celda "actual" que es el padre del vecino que hemos estado analizando. Metemos este vecino en OPEN y pasamos al siguiente vecino. Cuando ya hemos estudiado todos los de "actual", escogemos de OPEN la celda con menor distancia y repetimos el proceso.

El algoritmo finaliza cuando la lista OPEN está vacía, y por tanto no hay camino posible porque se han analizado todas las celdas, o cuando se va a quitar la meta de OPEN para ser estudiado. Esto es así porque pueden encontrarse caminos más cortos a la meta mientras todavía esté incluido en OPEN. Sin embargo, una vez que se quita de dicha lista, no hay camino más corto hasta él desde la meta. Dicho camino lo obtendremos a través del array PARENTS.

En la Figura 49 podemos observar un ejemplo con grafo, siendo "S" el nodo inicial y "G" la meta. Las listas durante las sucesivas iteraciones del algoritmo son las siguientes:

1. O = { S }
2. O = { 2, 4, 1, 5 }; P = { S }
3. O = { 4, 1, 5 }; P = { S, 2 }
4. O = { 1, 5, 3 }; P = { S, 2, 4 }
5. O = { 5, 3 }; P = { S, 2, 4, 1 }
6. O = { 3, G }; P = { S, 2, 4 ,1 }

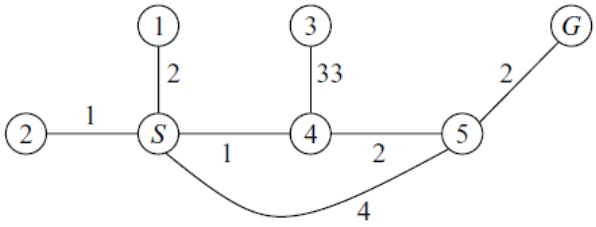


Figura 49: Dijkstra [23, p. 532]

Podemos encontrar otro ejemplo bien explicado y desarrollado en [52].

A*

El algoritmo A* es uno de los algoritmos más utilizados para encontrar el camino óptimo, pudiendo encontrar abundante bibliografía sobre él: [23, pp. 527-532], [8, pp. 607-608], [45, pp. 37-38], [38, pp. 49-64], [53, pp. 1-2], [54], [55, pp. 1-15], [46, pp. 2-3], [56, p. 184], [57, pp. 11-26]. Dicho algoritmo es una extensión del Dijkstra que intenta reducir el número de estados explorados incorporando una estimación heurística del coste de llegar desde el nodo o celda actual a la meta. La heurística será optimista y por tanto, admisible, debido a que dará un valor menor o igual a la distancia real desde la celda actual hasta la meta. Según el movimiento permitido para el móvil en la cuadrícula, podemos usar unas heurísticas u otras [54, pp. 10-11].

- Para cuadrículas cuadradas con conectividad de 4 puntos, es decir, moviéndose hacia el norte, sur, este y oeste, podemos usar la llamada Manhattan, que utiliza la siguiente función: $h(n) = |x_n - x_{meta}| + |y_n - y_{meta}|$, siendo *n* la celda actual.
- Si además puede moverse diagonalmente, es decir, conectividad de 8 puntos, usaríamos la Diagonal. Primero calcula la Manhattan; luego el número de posibles pasos diagonales con: $h_{diag}(n) = \min(|x_n - x_{meta}|, |y_n - y_{meta}|)$; por último calcula la distancia diagonal más corta añadiendo la distancia hecha sin diagonales y restando los pasos ahorrados usando diagonales. Quedaría: $h(n) = \sqrt{2} \cdot h_{diag}(n) + (h_{manh}(n) - 2 \cdot h_{diag}(n))$.
- Si puede moverse en cualquier ángulo utilizaremos la Euclídea: $h(n) = \sqrt{(x_n - x_{meta})^2 + (y_n - y_{meta})^2}$.

El algoritmo será eficiente, es decir, el menor número de celdas o estados que visita en su búsqueda, si la heurística escogida es buena, y si no, encontrará el camino pero tardará más tiempo del necesario y quizás no es óptimo. Esto implicaría que no es el camino más corto. En el caso de escoger una heurística que sobreestime la distancia a la meta, el algoritmo tiende a expandir los nodos que están en la dirección directa a la meta sin intentar otros. Esto puede llevar a búsquedas más lentas si el camino final contiene direcciones que se alejan del destino deseado [56, p. 184] (ejemplo en [23,

pp. 535-536]). Si *h(n)* es igual a 0, estaríamos utilizando el algoritmo Dijkstra, siendo por lo tanto, una búsqueda siempre sistemática [45, p. 38].

Al igual que en anteriores algoritmos, tendremos una lista OPEN, cuyos nodos o celdas se ordenarán según el menor coste de la función $f(n) = g(n) + h(n)$, siendo *n* el estado o celda actual, *g(n)* una matriz con el coste de llegar hasta dicha celda desde el inicio, y *h(n)* la heurística para la celda. También habrá otra lista CLOSE con las celdas ya estudiadas o que no hace falta explorar, como los obstáculos. Además, utilizaremos punteros en cada celda para saber cuál es su padre [8, pp. 607-608], [23, pp. 530-532].

Inicialmente todas las celdas vacías tendrán un valor de coste *g(n)* infinito y serán no visitadas. A la celda inicial le damos un valor de coste 0 y la añadimos a OPEN. Después, al expandirla, la quitamos de dicha lista y la añadimos a CLOSE. Actualizamos el coste *g(n)*, y por tanto*, f(n)* de cada uno de sus vecinos y los añadimos a OPEN. Dentro de esta lista los ordenamos por valor creciente de *f(n)*.

Cogemos el primer nodo, lo quitamos de la lista, lo expandimos y lo añadimos a CLOSE. Para cada uno de sus vecinos comprobamos si ya ha sido visitado. Si no ha sido así, lo añadimos a OPEN con su coste actualizado, lo marcamos como visitado y con el puntero dirigido a la celda que hemos expandido. Si ya había sido visitado antes pero el nuevo coste *g(n)* es menor del que tenía antes, hemos encontrado un camino más corto a esa celda y actualizamos su valor. Además, redirigimos el puntero hacia la celda que hemos expandido. Una vez hecho esto con todos los vecinos, cogemos el nodo de OPEN con el menor valor de *f(n)* y repetimos el proceso.

El algoritmo finaliza cuando vamos a expandir la celda final o cuando la lista OPEN queda vacía, lo que implica que todas las celdas ya han sido exploradas y no hay camino posible.

No se garantiza que el primer camino encontrado, es decir, al introducir la meta en OPEN, sea el más corto. Se deben expandir primero todos las celdas con valor de *f(n)* menor que el de la meta antes de dar por terminado el algoritmo porque es posible que haya otro camino más corto. Esto hace que el algoritmo A* sea óptimo y encuentre el camino más corto posible. Hay que tener en cuenta que si no se conoce la localización de la meta, el algoritmo A* no puede utilizarse.

Podemos encontrar un ejemplo del algoritmo en [23, pp. 533-535] y otros dos de grafos paso por paso en [57, pp. 11-23]. Vamos a exponer el resultado de uno de estos últimos (Figura 50). La cifra escrita en el interior de los nodos es el valor de la heurística y la de las aristas es el coste de viajar por ella. Al expandir el nodo inicial, se incorporan a OPEN los nodos A, B y C. B tiene mayor prioridad, lo expandimos y obtenemos G, H e I. Como H no se puede expandir más, y el siguiente prioritario es A, estudiamos sus vecinos: D, E y F. Podemos obtener un camino a la meta de distancia 5. Como todavía quedan más nodos por expandir, intentamos con C. Se puede observar que los nodos que tienen una prioridad igual o menor a la de la meta actual se descartan: D, I, F y G, debido a que a través de ellos, el camino no será menor. Al expandir C, obtenemos J, K y L. A través de K obtenemos un camino a la meta de distancia 4 y los demás quedan rechazados por tener menor prioridad. Por tanto, hemos encontrado finalmente el camino óptimo, conociendo cuál es la ruta mediante los punteros.

Si la heurística no fuera optimista, podríamos descartar nodos cuyos caminos luego son más cortos de los esperados. Sin embargo, al ser optimista, podremos explorar dichos nodos y luego calcular su distancia real.

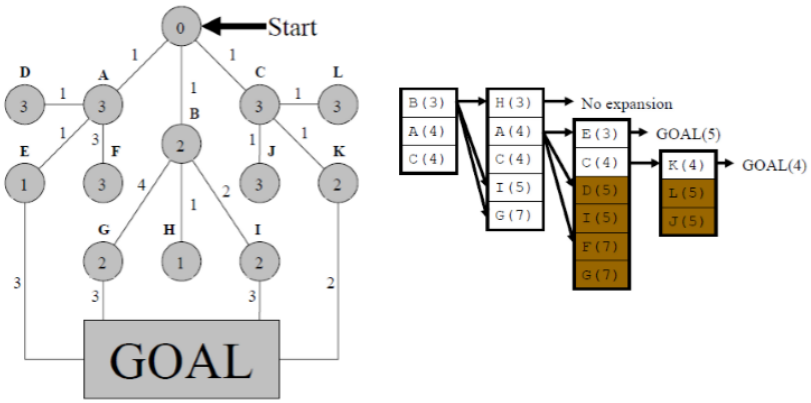


*Figura 50: A**
Izq: grafo con nodos, dcha: pasos en lista OPEN [57, p. 16]

En [54, pp. 37-44] podemos encontrar código de A* escrito en MATLAB®.

D*

Hasta ahora hemos trabajado en entornos y mapas estáticos, es decir, sin variaciones en las configuraciones de los obstáculos moviéndose solo el propio robot. Sin embargo, en la mayoría de las aplicaciones el entorno es dinámico y/o desconocido, tales como en exteriores, otros planetas, etc. También puede deberse a que el mapa inicial sea impreciso y conforme el robot lo explora con sus sensores, el mapa se

actualiza cambiando la configuración real y las distancias entre los estados o celdas. En el algoritmo A* habría que repetir el proceso entero para recalcular el camino, lo cual es muy ineficiente y con gran gasto computacional [53, p. 2].

Esto ha llevado a desarrollar un algoritmo que solo repare aquella parte modificada a partir de la solución previa. Es el algoritmo de Stentz ya que fue creado por Anthony Stentz en 1994, o más comúnmente, D*, debido a que es un A* Dinámico. Podemos encontrar abundante bibliografía sobre ello: [23, pp. 536-545], [45, pp. 662-667], [58], [53], [57, pp. 27-67], [59], [60, pp. 21-26,45-49], [38, pp. 74-75], [61]. Inicialmente ejecuta el algoritmo de Dijkstra desde la meta hacia el inicio guardando el coste de cada celda a la meta en dos funciones y los punteros de dichas celdas. Cuando el robot ya ha obtenido el camino óptimo, éste puede empezar a desplazarse por él. Sin embargo, si durante el trayecto cambia alguno de los obstáculos afectando a dicho camino, se ejecuta un algoritmo complejo que repara dicha modificación localmente buscando otro camino óptimo y evitando el obstáculo.

Gracias a esto, no hace falta recalcular todo el camino sino solo esa zona local. Para ello, actualiza el coste heurístico *h* que es una estimación de la longitud del camino de la celda a la meta, pero no necesariamente el camino más corto como en A*.

Además, utiliza los valores heurísticos mínimos *k,* que son una estimación del camino más corto a la meta. Ambos valores son iguales al inicio pero después se van modificando a medida que el algoritmo funciona. En él se comparan con los anteriores guardados y entre sí debido a las modificaciones que se producen, se reinsertan en OPEN, se reordenan mediante dicha *k* y se modifican los punteros.

Es un proceso bastante complejo que no explicaremos en el presente trabajo. Podemos encontrar el algoritmo detallado en [23, pp. 542-545], y con un ejemplo paso por paso en [57, pp. 38-59] (Figura 51). En [60, pp. 21-26;45-49] podemos encontrar otro ejemplo y la implementación básica de D*.

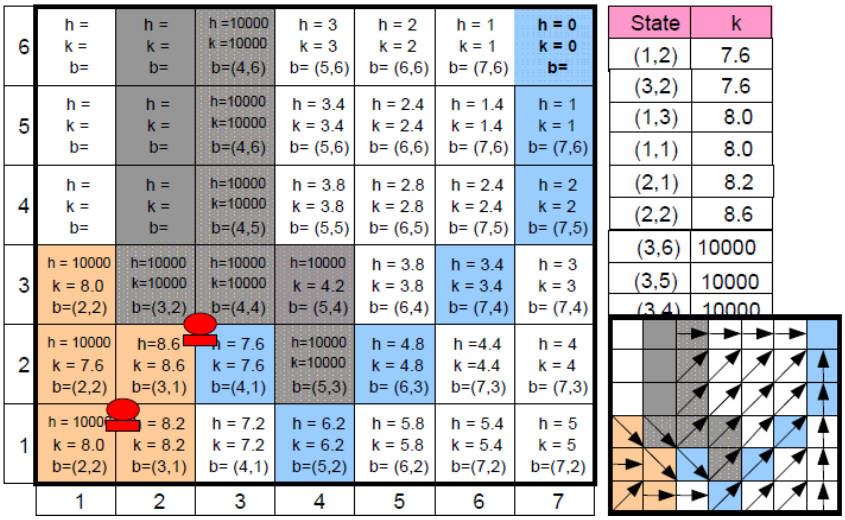


Figura 51: D [57, p. 58]*

D* Lite

Sin embargo, Koenig y Likhachev crearon en 2002 el algoritmo D* Lite [53, p. 3], [55, pp. 2-3], [57, pp. 70-132] y [38, pp. 74-78], que muy similar a D* pero más simple y más eficiente para algunas tareas de navegación [53, p. 2]. Inicialmente construye el camino óptimo como en A* pero en sentido inverso, es decir, desde la meta hacia el inicio. Cuando se produce algún cambio en el grafo, los estados cuyos caminos hacia la meta están afectados, se actualizan sus costes y se colocan en OPEN para propagar los efectos al resto de celdas. Así, solo se procesa la zona afectada, como en D*. Además, D* Lite utiliza una heurística que limita los estados procesados a aquellos cuyos cambios están relacionados al coste del camino del estado inicial.

El algoritmo mantiene el camino de menor longitud entre el inicio y la meta mediante dos funciones. *La* primera, *g(s)*, almacena una estimación del coste de cada estado a la meta, siendo *s* un estado, celda o nodo. La segunda, *rhs(s),* guarda el coste de un paso hacia delante, es decir, de la meta hasta uno de los sucesores o hijos. Viene definida de la siguiente manera siendo c(s,s') el coste de viajar entre ambos estados.

$$rhs(s) = \begin{cases} 0 & s = s_{meta} \\ min_{s' \in Suc(s)} \left(c(s,s') + g(s') \right) & demás\ casos \end{cases}$$

Un estado será consistente si el valor de *g(s)* es igual al de *rhs(s),* sobre-consistente si *g(s)>rhs(s)* y sub-consistente si *g(s)<rhs(s)*.

Al igual que A*, utiliza una heurística *h(s,s')* que será admisible si es optimista, es decir, no sobreestima la distancia entre ambos estados. Una lista OPEN siempre

contendrá estados inconsistentes para ser actualizados o hacerse consistentes. En dicha lista se ordenarán por un criterio prioritario *key* formado por dos valores:

$$key(s) = [k_1(s), k_2(s)] = \left[\min\big(g(s), rhs(s)\big) + h(s_{inicio}, s), \min\big(g(s), rhs(s)\big)\right]$$

D* Lite expande los estados de OPEN extrayendo primero aquellos con menor valor de *key*. Ahora procederemos a explicar brevemente este algoritmo.

Primero inicializamos con OPEN vacío y definiendo un valor infinito de *g(s)* y *rhs(s)* para todos los estados, incluidos los obstáculos. A la meta le damos un *rhs(s)*=0 y lo insertamos en OPEN ya que es inconsistente ($g(s_{meta})=\infty$) calculando su valor *key*. Mientras el valor de *key(s)* sea menor que el del inicio y *rhs(s_{inic})* sea distinto de *g(s_{inic})* ejecutamos los siguientes pasos.

Expandimos el estado de OPEN con menor *key(s)* sacándolo de la lista e igualamos *g(s)* a *rhs(s)* si es sobre-consistente. En el caso de que sea sub-consistente le damos a *g(s)* un valor ∞. Para cada uno de sus predecesores ***s'*** actualizamos su valor *rhs.* Si ya estaba en OPEN lo quitamos y lo metemos otra vez si se hacen inconsistentes. Volvemos a repetir el proceso cogiendo de OPEN el estado con menor valor de *key(s)* mientras se cumpla las condiciones mencionadas en el párrafo anterior.

Una vez que tenemos el camino más corto viajamos por él siguiendo el gradiente descendente de *g(s)*. Si durante el trayecto se producen cambios en los costes directos de arista (entre estados), actualizamos sus costes de arista. Si estamos en grafos no dirigidos habrá que considerar las dos aristas de cada conexión entre estados. Para todos aquellos que cambien, también actualizamos el estado recalculando su *rhs* y procedemos a seguir con el algoritmo como hemos visto antes.

Podemos encontrar el algoritmo básico en la bibliografía antes citada, y en [57, pp. 74-132] un ejemplo detallado (Figura 52), sin utilizar *h* y con el algoritmo básico, sin optimizarlo.

Existen otros algoritmos similares a D* pero optimizados, como Focused D* [57, pp. 63,67] o Field D*. El último de estos se utilizó en los rovers de Marte Spirit y Opportunity [38, p. 78], demostrando ser muy eficiente. Éste viene bien detallado y con su algoritmo en [59], pero sin embargo, no lo explicaremos en este trabajo.

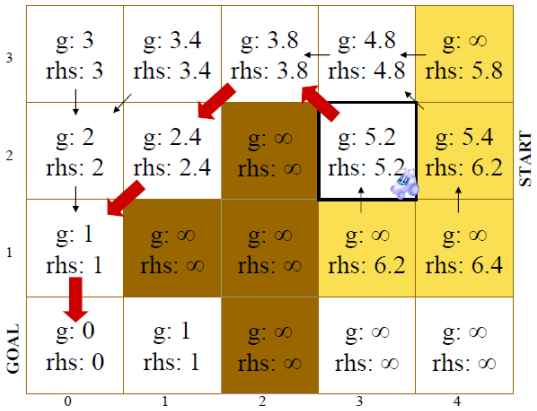


Figura 52: D Lite [57, p. 131]*

ARA*

Hay situaciones en las que el problema de planear la ruta óptima es complejo y el móvil tiene tiempo limitado. Esto puede llevar a que el algoritmo A* sea inviable debido al gran número de estados que tienen que ser procesados. En estos casos, debemos obtener la mejor solución posible rápidamente e ir mejorando el camino subóptimo mientras el tiempo lo permita. Estos algoritmos reciben el nombre de "Anytime" (en cualquier momento). Uno de estos es el ARA* (Anytime Repairing A*) creado por Likhachev, Gordon y Thrun en 2003 [58, pp. 3-4], [55, pp. 7-8;16-23], [53, pp. 5-6]. Se basa en realizar el algoritmo A* inverso sucesivamente pero utilizando un factor de inflación $\varepsilon > 1$ sucesivamente menor. Además, la heurística debe ser consistente, lo que significa:

$$h(s, s_{meta}) \leq c(s, s') + h(s', s_{meta}) \,; \quad h(s_{meta}, s_{meta}) = 0$$

para todos los estados *s*, siendo ***s'*** los sucesores de *s* y ***c(s,s')*** el coste entre ambos estados. Con el factor de inflación $f(s) = g(s) + \varepsilon \cdot h(s_{inic}, s)$ se garantiza que el coste de la solución generada esté dentro del rango de ε veces el coste de la solución óptima. En cada búsqueda se satisface el límite sub-óptimo que gradualmente va disminuyendo con ε.

Este algoritmo limita el procesamiento requerido en cada búsqueda considerando solo aquellos estados cuyos costes calculados previamente no sean válidos con el nuevo ε. Inicialmente, con todas las listas vacías, se ejecuta el A* con un factor de inflación ε_0 pero sólo se expande cada estado como mucho una vez. Una vez que un estado ha sido expandido, si se hace inconsistente debido a un cambio de coste debido a un estado vecino, no se reinserta en OPEN, sino en otra lista INCONS. Ésta contiene todos los estados inconsistentes que ya han sido expandidos. Entonces, cuando la

búsqueda actual termina, se reduce ε, los estados de INCONS se insertan en OPEN y se reordenan según la nueva $f(s)$.

Se eliminan los estados de CLOSE, y se expanden los nodos que haya ahora en OPEN, repitiendo el proceso. Se dejan de expandir los estados de OPEN cuando tengan un valor de $f(s)$ mayor que el del inicio $f(s_{inic})$. Una vez que llegamos a un factor de inflación igual a 1, ya sería el camino óptimo, deteniendo el algoritmo.

Esta forma de expandir los estados solo una vez permite encontrar una solución mucho más rápido y al considerar en una nueva búsqueda los estados de INCONS, se aprovecha el esfuerzo anteriormente utilizado. Así, al reducir ε, solo se requiere una pequeña cantidad de computación para generar una nueva solución.

En [59, pp. 5-6] se enuncia alguna aplicación de ARA*. Una es la planificación de movimientos en espacios de estados de grandes dimensiones, como brazos robóticos con 20 articulaciones. Otro ejemplo es planear trayectorias suaves para robots móviles, como el ATRV, en ambientes exteriores conocidos utilizando 4 dimensiones: posición (x,y), orientación y velocidad del robot.

En la fila de arriba de la Figura 53 se ha ejecutado el A* con distintos valores de ε, mientras que abajo se ha utilizado el ARA* con los mismos valores. Puede observarse que la primera búsqueda en ambos algoritmos es igual, pero en la segunda búsqueda con ARA* sólo se ha expandido un estado (los asteriscos indican estados inconsistentes) mientras que con A* 15 celdas. Por último, en la tercera con ARA* solo se han expandido 9 estados y con A* 20 celdas ya que ha empezado desde cero. Finalmente se ha llegado a la misma solución óptima necesitando expandir con ARA* solo 23 estados y en menor tiempo.

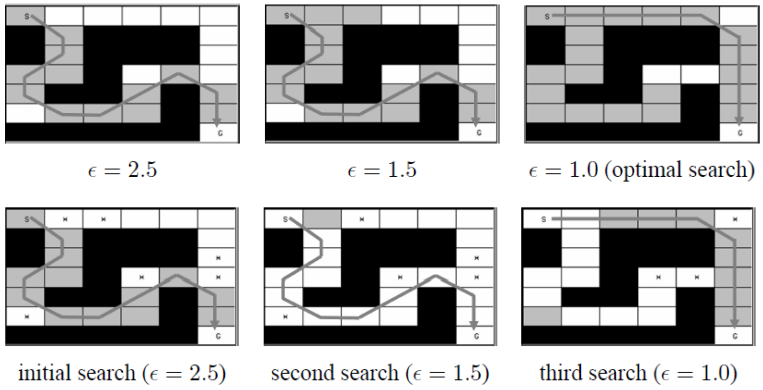


Figura 53: Comparación A y ARA*
Arriba: A*, abajo: ARA* [55, p. 18]*

AD*

Como hemos visto, existen algoritmos eficientes que funcionan en problemas de planificación complejos (ARA*) y en entornos dinámicos (D* y sus variantes). Sin embargo, puede darse el caso de que ambas condiciones se produzcan simultáneamente. Para ello, los mismos autores de los anteriores algoritmos han creado el AD* (Anytime Dynamic A*, o Anytime D*), que aúna ambos algoritmos.

Básicamente, como el ARA* ejecuta una serie de búsquedas sucesivas con un factor de inflación decreciente. Cuando hay cambios en el entorno que afectan a los costes de aristas entre nodos, dichos estados afectados se incorporan a OPEN con prioridades iguales al mínimo de su valor *key* anterior y el nuevo valor, como en D* Lite. Los estados de la lista son procesados hasta que la solución actual garantice el límite subóptimo marcado por ε.

No explicaremos el algoritmo en este trabajo, encontrándose en [58, p. 4], [59, pp. 6-7] y [55, pp. 29-31]. Podemos encontrar un ejemplo en la Figura 54 sacado de la bibliografía. A la izquierda tenemos el cálculo con ARA* y a la derecha con AD*. En los dos primeros pasos se comportan de la misma forma, pero al cambiar el ambiente, el ARA* debe recalcular desde el inicio ya que no incorpora los cambios en los costes de aristas. Sin embargo, AD* es capaz de reparar su solución previa dada la nueva información y disminuyendo su factor de inflación a la vez. Esto conlleva que solo tenga que expandir las 5 celdas que están directamente afectadas por los cambios y que se encuentran entre el móvil y la meta. En total, el segundo requiere expandir 20 celdas, el primero 24 y D* Lite 27 (también representado en la bibliografía) para encontrar el camino óptimo.

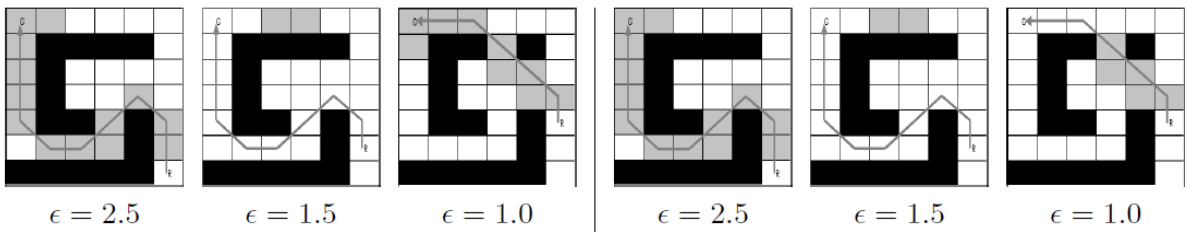


Figura 54: Comparación ARA y AD*
Izquierda: ARA*, derecha: AD* [58, p. 6]*

7.2.4 Métodos probabilísticos

Siguiendo [23, pp. 197-202], los métodos de planificación normalmente requieren una representación explícita de la geometría del espacio de configuraciones. Esto lleva a

ser inviable para problemas con mayor número de dimensiones de dicho espacio. Los métodos probabilísticos o basados en muestras ("Sampling-Based Algorithms") pueden resolver esos problemas generando muestras en el espacio libre de obstáculos y conectándolas para crear caminos factibles. Además de ser útil para robots con muchos grados de libertad, puede utilizarse en casos con restricciones, como cinemáticas, dinámicas, de estabilidad, energía…

Según si están diseñados para una consulta o varias en un mismo mapa, podemos encontrar diversos algoritmos. El PRM fue concebido para múltiples búsquedas mientras que EST y RRT para sólo una consulta. Para algunos problemas de planificación de caminos difíciles, los de una sola consulta requieren construir árboles muy grandes, siendo mejor un método más potente intermedio entre los anteriores, el SRT. Explicaremos brevemente cada uno de ellos, estando bien explicados y con pseudo-algoritmos en [23, pp. 197-253].

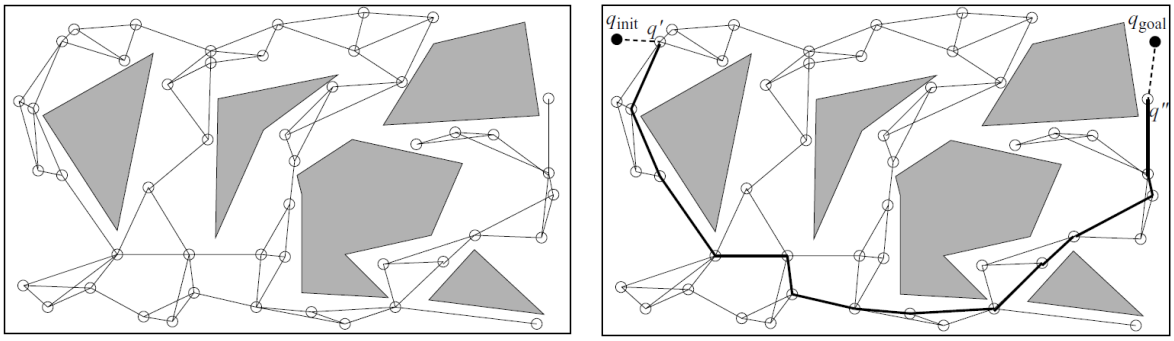
PRM

El PRM, o "Probabilistic Road Map", está hecho para representar exhaustivamente la conectividad de un determinado espacio de configuraciones y obtener un mapa para múltiples búsquedas entre puntos arbitrarios [23, pp. 202-227], [8, pp. 541-543], [12, pp. 99-102], [62, pp. 4-6], [38, pp. 30-33]. Se puede separar en dos fases: fase de aprendizaje, en la que se construye el "mapa de carreteras", y la fase de consulta, en cual se conectan los puntos o configuraciones inicial y final deseadas mediante dicho mapa.

El algoritmo básico construye el mapa de carreteras de una manera probabilística, representándose mediante un grafo no dirigido $G = (V, E)$. V es el conjunto de configuraciones del robot muestreadas aleatoriamente mediante una distribución uniforme en el espacio libre (Q_{libre}). E corresponde al conjunto de aristas o caminos sin obstáculos entre dos configuraciones (q_1, q_2). Además, utiliza dos funciones: una de planificación (*Δ*) que devuelve un camino libre de obstáculos entre dos configuraciones o NIL si no existe, y otra de distancia (*dist*) para hallar los nodos más cercanos a uno dado.

Inicialmente el grafo está vacío. Tomamos una muestra aleatoria de Q con una distribución uniforme, agregándola a V si no está en ningún obstáculo, es decir, en Q_{libre}. Repetimos el proceso hasta un número deseado de nodos en V que indique el

usuario. Una vez hecho esto, para cada uno de estos nodos q de V elegimos un conjunto N_q de los *k* vecinos más cercanos a ese nodo de acuerdo a *dist*. Dicho número de vecinos *k* también es dado por el usuario que invoca el algoritmo. El planificador local conectará cada configuración q a cada nodo q' incluido en N_q. Si el *Δ* resulta bien (no devuelve NIL), entonces hay un camino factible entre ambos nodos y la arista (q, q') se agrega al grafo en E. En la figura se puede observar el mapa construido para un robot puntual en un espacio de dos dimensiones con una *k* igual a 3, es decir, tres vecinos por cada nodo (Figura 55a).



a) [23, p. 205] b) [23, p. 207]

Figura 55: PRM
a) Mapa de carreteras b) Camino óptimo

En la fase de búsqueda, el mapa es utilizado para resolver problemas de planificación de caminos individuales. Asumiendo que Q_{libre} está conectado, la principal cuestión es conectar unas configuraciones deseadas (q_{inicio} y q_{meta}) al mapa. Una estrategia consiste en ordenar los *k* nodos más próximos a q_{inicio} en orden ascendente de acuerdo a la función *dist*. Se intenta conectar q_{inicio} a cada uno de dichos nodos hasta que el planificador local *Δ* triunfa. Realizamos el mismo procedimiento con q_{meta}. Una vez realizado esto, podemos conectar ambas configuraciones mediante *dist* y algún algoritmo de búsqueda, como Dijkstra o el A*, calculando el camino más corto por el grafo construido (Figura 55b). Conviene emplear un tiempo considerable para construir un mapa completo y así ser útil para múltiples búsquedas.

En [23, pp. 204,206] podemos encontrar el pseudo-código de ambas fases. Además, en las siguientes páginas trata sobre diferentes formas de muestrear, de obtener la función *dist*, de distintos planificadores locales… Podemos destacar dos estrategias muy comunes para comprobar si se produce colisión con obstáculos entre dos configuraciones q' y q'': la incremental y la subdivisión. Para ello, dividimos el segmento que las une mediante un número finito de configuraciones o pasos

intermedios. Éstos estarán separados con una distancia menor que un valor de paso positivo y predefinido, siendo normalmente pequeño para garantizar todas las colisiones posibles.

En el caso de la estrategia incremental, el robot parte de q' y se mueve paso a paso a lo largo del segmento hacia q''. Se realiza la comprobación de colisión al final de cada paso. El algoritmo termina cuando detecta una colisión o cuando llega a q''. La estrategia de la subdivisión, en cambio, chequea el punto medio q_m del segmento. Entonces, estudia los puntos intermedios entre el nuevo q_m y un extremo. Vuelve a seccionar los nuevos segmentos por la mitad y estudia los puntos medios. Así sucesivamente hasta que detecta una colisión o la longitud entre los pasos es menor que el valor de paso predefinido.

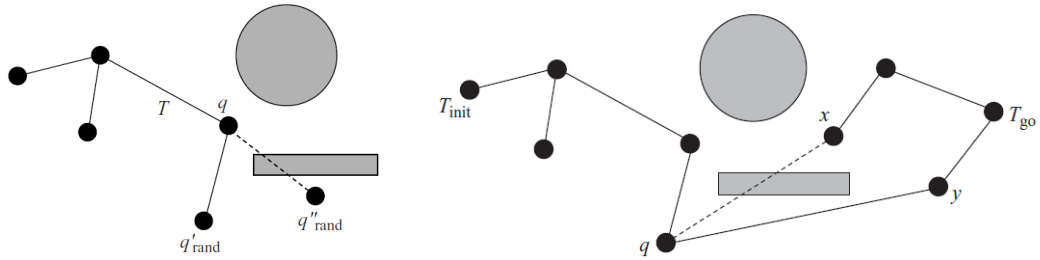
Por otro lado, en dicho libro enuncia uno de los principales problemas del muestreo aleatorio uniforme, el cual es encontrar una solución cuando tiene que pasar por un pasillo estrecho entre obstáculos ya que es poco probable que las muestras aleatorias coincidan dentro de él. Para ello, propone diversas soluciones: muestreando cerca de los obstáculos, basado en visibilidad, etc. [23, pp. 216-225].

EST

El EST [23, pp. 230-233], que procede de "Expansive-Spaces Trees", es un algoritmo de planificación de una sola búsqueda, por lo que intenta encontrar para la consulta realizada, una solución de forma rápida. Se basa en la construcción simultánea de dos "árboles" T_{inicio} y T_{meta} con inicio en q_{inicio} y q_{meta}, respectivamente. Si llamamos *T* a uno de ellos, el planificador primero selecciona una configuración q en *T* basándonos en una probabilidad $\pi_T(q)$ desde la cual hacer crecer *T* y tomamos una muestra aleatoria q_{aleat} de una distribución uniforme en la vecindad de *q*. El planificador local *Δ* intenta una conexión entre q y q_{aleat} de la misma manera que en PRM. Si ha sido exitoso, q_{aleat} se agrega a los vértices de *T* (V) y la arista a E de *T*. El proceso se repite hasta que obtengamos un número *n* de configuraciones en *T* especificado por el usuario.

En el PRM se agregaban todas las configuraciones a V que estuvieran en el espacio libre ($\mathcal{Q}_{libre}$) y luego se conectaban los factibles con el planificador *Δ* agregándose a E. En el EST, en cambio, antes de agregar las nuevas muestras a V, se verifica que

la planificación local con *Δ* es factible. Si es el caso, se agrega la nueva configuración y la arista, y si no, se rechaza, como puede verse en la Figura 56a con q''_{rand}.



a) [23, p. 231] b) [23, p. 233]

Figura 56: EST
a) Nueva configuración b) Conexión árboles

La eficacia del EST reside en la habilidad de evitar tomar demasiadas muestras en alguna región de $\mathcal{Q}_{libre}$, especialmente, en las zonas cercanas a las configuraciones inicial y final. Por ello, se han desarrollado diversas funciones de densidad (π_T) basadas en probabilidad para elegir la q dando prioridad a aquellas con poca densidad de vecinos. En [23, p. 232] se enuncian diversas estrategias. Una de ellas es asociar a cada configuración q de *T* un peso $w_T(q)$ que cuenta el número de vecinos en una vecindad predeterminada de q. Si definimos $\pi_T(q)$ inversamente proporcional a $w_T(q)$, las configuraciones con pocos vecinos serán más probables de ser elegidas.

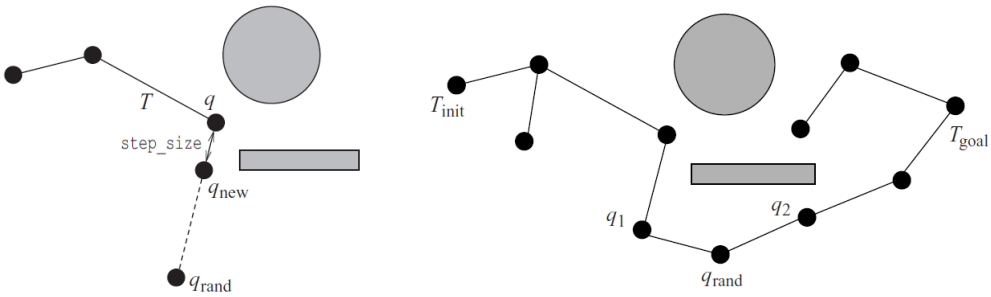
La unión de los dos árboles es realizada "empujando" la búsqueda en el espacio de uno hacia el otro. Cuando se genera una nueva configuración q en T_{inicio}, el planificador local intenta conectarlo a las configuraciones más cercanas de T_{meta} . Si la conexión es exitosa, los dos árboles ya se han unido (Figura 56b). Si no es así, los árboles son intercambiados, es decir, se genera una nueva configuración en T_{meta} y se intenta conectar a T_{inicio}. El proceso se repite hasta que se conectan o un número de veces especificado. El pseudo-código viene explicado en la bibliografía citada al principio de este algoritmo.

RRT

El algoritmo RRT, o "Rapidly-Exploring Random Trees", también fue creado para una sola búsqueda, explorando sólo aquella zona del espacio relevante para resolver el problema deseado [23, pp. 233-238], [8, pp. 543-545], [12, pp. 102-104], [62, pp. 6-8], [38, pp. 34-42]. Al igual que el EST, utiliza dos árboles T_{inicio} y T_{meta} con inicio en q_{inicio} y q_{meta}, respectivamente. En cada iteración se muestrea uniformemente una

configuración aleatoria q_{aleat} de $\mathcal{Q}_{libre}$. Buscamos la configuración más cercana q_{cerca} a q_{aleat} del árbol *T* que estamos actualmente expandiendo. Normalmente se traza una línea recta entre ambas configuraciones y nos movemos desde q_{cerca} una distancia igual a un valor de paso. Este valor de paso se puede elegir dinámicamente según la distancia dada por *dist* entre ellas. Se suele coger un valor grande si las configuraciones están lejos de chocarse con obstáculos y pequeño en otro caso. Se genera así una nueva configuración q_{nueva} (Figura 57a). Si está libre de obstáculos, se añade a los vértices V de *T* y la arista (q_{cerca}, q_{nueva}) a E, y en caso contrario devuelve NIL.

Hay que destacar el balance entre el número de muestras añadidas al árbol y la exploración del espacio de configuraciones $\mathcal{Q}$, especialmente para problemas de muchas dimensiones. Si el valor de paso es pequeño, entonces los pasos en la exploración serán pequeños y los nodos del árbol estarán más juntos.



a) [23, p. 235] b) [23, p. 237]

Figura 57: RRT
a) Nueva configuración con valor de paso b) Unión árboles

El muestreo de q_{aleat} puede ser de distintas maneras. Una es cogiendo nodos uniformemente al azar, como se ha dicho antes, pero también se puede elegir q_{aleat} igual a q_{meta} si estamos en el árbol T_{inicio} , y viceversa. Sin embargo, según [23, p. 236], acaba comportándose como un planificador de potencial aleatorizado, pudiendo atascarse en un mínimo local. Por tanto, una buena elección es una función de muestreo que alterne de acuerdo a una distribución de probabilidad, entre muestras uniformes y muestras en regiones que contienen a las configuraciones q_{inicio} o q_{meta} según el caso.

La unión de los dos árboles T_{inicio} y T_{meta} es similar al EST haciendo crecer los árboles uno hacia el otro. Se genera una nueva configuración q_{nueva} en T_{inicio} en la dirección de una muestra q_{aleat} . Entonces, se intentan conectar los nodos más cercanos de

T_{meta} a q_{nueva}. Si es exitoso, la planificación ha terminado, y si no, los árboles son intercambiados, obteniendo un nuevo nodo en T_{meta} e intentado conectarla a los nodos de T_{inicio}. Se repite el proceso hasta que se unan o un número determinado de veces. El pseudo-código viene explicado en la bibliografía citada al principio de este algoritmo. En el ejemplo de la Figura 57b, q_1 es de T_{inicio} extendiéndose a q_{aleat}. q_2 es de T_{meta}, conectándose a q_{aleat} exitosamente

La implementación del RRT es más fácil que el EST, pero sin embargo, no computa el número de configuraciones que caen dentro del vecindario predefinido de un nodo, ni mantiene una distribución de probabilidad para sus configuraciones.

Los tres algoritmos vistos son probabilísticamente completos, es decir, la probabilidad de encontrar la solución tiende a uno a medida que el tiempo de ejecución tiende a infinito. Si no existiera solución, el algoritmo se ejecutaría continua e indefinidamente, a menos que se le imponga un número máximo de muestras, como hemos hecho en las explicaciones anteriores.

SRT

Por último, podemos combinar los dos tipos de algoritmos probabilísticos vistos para crear el planificador SRT, o "Sampling-Based Roadmap of Trees" [23, pp. 238-242]. Combina el método del PRM para múltiples búsquedas junto con el del EST o RRT para una sola. Por tanto, el SRT es capaz de resolver muchas consultas ya que crea un "mapa de carreteras", pero también puede ser visto como de una sola consulta ya que para ciertos problemas difíciles, el coste de resolverlo por SRT es menor que con el EST o el RRT.

Al igual que el PRM, el SRT construye dicho mapa representando la conectividad de $\mathcal{Q}_{libre}$, pero en lugar de estar formado por configuraciones simples o puntuales, son árboles. La conexión entre dichos árboles se realiza mediante un algoritmo de árbol bidireccional como el EST o el RRT. Por tanto, el grafo no dirigido del mapa global $G = (V, E)$, está formado por subgrafos $G_T = (V_T, E_T)$ que corresponden a cada uno de los árboles.

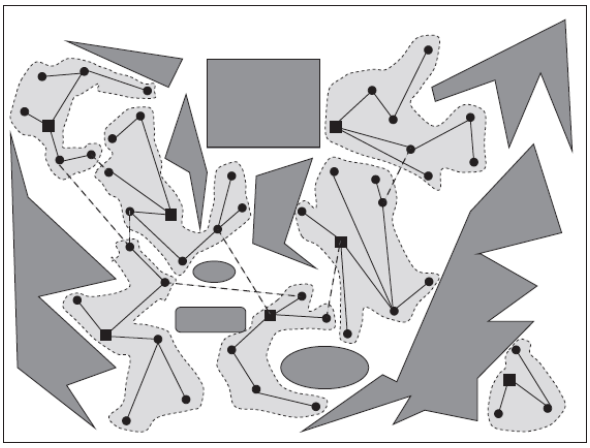


Figura 58: SRT [23, p. 239]

En SRT los árboles del mapa se empiezan a crear tomando muestras uniformemente aleatorias en Q_{libre}, y se hacen crecer mediante un planificador local como los vistos hasta ahora. Después se intentan unir mediante aristas considerando para cada árbol T_i el conjunto N_{T_i} formado por los árboles más cercanos. La distancia entre dos árboles T_i y T_j la denotaremos como $dist(q_{T_i}, q_{T_j})$. Durante la conexión, es probable que se añadan configuraciones adicionales. Si la conexión es exitosa, la arista es añadida a E_T y las componentes de ambos árboles se unen en uno solo. En la Figura 58 se ve que los árboles empiezan en los cuadrados negros. Las líneas negras son conexiones dentro del mismo árbol y las discontinuas entre diferentes árboles.

El algoritmo intercala los intentos de conectar los árboles T_{inicio} y T_{meta} a sus respectivos árboles vecinos. Habremos encontrado un camino entre q_{inicio} y q_{meta} cuando ambos estén en la misma componente conectada del mapa. Para saber la secuencia de configuraciones que determina dicho camino, es necesario conocer la secuencia de los árboles que definen el camino entre T_{inicio} y T_{meta}. Sabido esto, concatenaremos los caminos locales entre dos árboles consecutivos. En la bibliografía citada al inicio del algoritmo podemos encontrar el pseudo-código correspondiente a este algoritmo.

En [45, pp. 217-220,228-244] se explican algunos de estos métodos probabilísticos y además, explica la diferencia de usar sólo un árbol de búsqueda, dos o más. Hay casos en los que los mapas poseen configuraciones parecidas a “trampas” (Figura 59). Si tenemos el inicio dentro y la meta fuera de dicha trampa, puede que con sólo un árbol no se pueda solucionar y hay que recurrir a dos, pudiéndose así solucionar.

Si tuviéramos dos trampas iguales, posiblemente harían falta más de dos árboles y hay casos difíciles de resolver incluso con múltiples árboles.

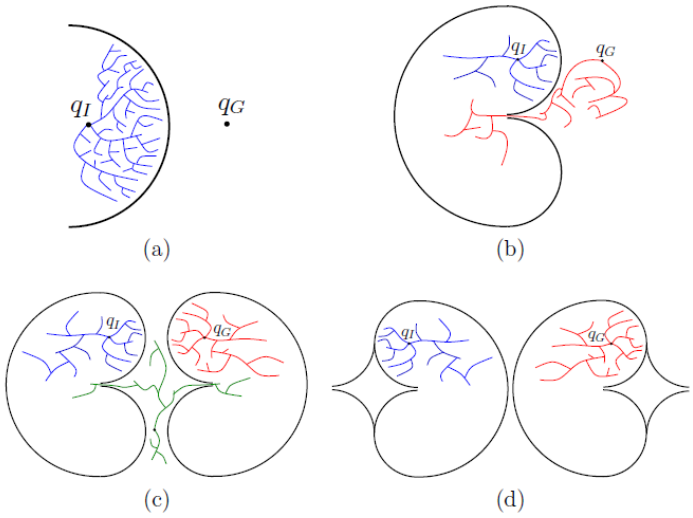


Figura 59: Trampas en métodos probabilísticos [45, p. 219]

7.2.5 Funciones potenciales

Podemos encontrar otros métodos para cumplir nuestro objetivo de navegación: llegar de un punto inicial a otro final y además evitando obstáculos. Uno de ellos son las funciones potenciales [23, pp. 77-92], [8, pp. 546-554], [2, pp. 267-272], [38, pp. 45-47], [63, pp. 15-18]. Este es un método de planificación reactiva, es decir, reacciona consecuentemente a medida que se acerca a la meta y a los obstáculos, sin preocuparse mucho por el futuro, solo intenta acercarse más a la meta alejándose de los obstáculos.

En casos de navegación, si vemos las funciones potenciales (diferenciables) como energía, su gradiente serán fuerzas.

Siguiendo [23, pp. 77-92], el gradiente es un vector $\nabla U(q) = \left[\frac{\partial U(q)}{\partial q_1} \quad \frac{\partial U(q)}{\partial q_2} \quad \dots \quad \frac{\partial U(q)}{\partial q_m}\right]^T$ que apunta en la dirección en la que dicha función potencial *U* varía más rápidamente en un punto o configuración *q*. Utilizaremos el gradiente para generar un campo vectorial asignando un vector a cada punto. Cuando la función potencial *U* es energía, el campo vectorial del gradiente tiene la propiedad que el trabajo realizado a través de cualquier camino cerrado es cero. Para nuestro caso, la fuerza en un punto será el gradiente de la función potencial en dicho punto pero cambiado de signo, para que sea descendente.

En dicho libro, se hace un símil entre navegación con un robot por el campo vectorial gradiente creado con las fuerzas que actúan entre cargas positivas y negativas. Si el robot es una carga positiva y la meta una carga negativa, el robot se sentirá atraído por una fuerza hacia la meta ya que cargas de signo opuesto se atraen. Si además los obstáculos los definimos como cargados positivamente, el robot sentirá fuerzas de repulsión cuando se acerque a ellos ya que cargas del mismo signo se repelen.

En [2, p. 267] hace otra comparación: el robot es una pelota que rueda "colina abajo" hacia la meta que es el punto más bajo. Los obstáculos actuarán como "montañas" o "picos", por lo que el robot se aleja de ellos porque tienen posiciones más elevadas (Figura 60).

La superposición de las fuerzas de atracción y repulsión ejercidas sobre el robot crea el campo vectorial que lo hace moverse hacia la meta de manera suave. Si aparecen nuevos obstáculos durante el movimiento del robot, se puede actualizar el campo de potencial integrando la nueva información.

El robot termina su movimiento cuando alcanza un punto *q** en el que el gradiente es cero, $\nabla U(q^*) = 0$, llamándose punto crítico de *U*. En [23, pp. 78-79] distingue si *q** es un máximo, un mínimo o un punto de silla (como en una silla de montar a caballo). Para ello, observa la segunda derivada con la matriz Hessiana según si es no singular y definida positiva o negativa. Sin embargo, esto no es necesario en métodos descendentes ya que el robot termina su movimiento en mínimos locales. No puede terminar en máximos locales ya que el gradiente es descendente, y si empezase en un máximo, cualquier perturbación en la posición lo liberaría de dicho máximo yendo al mínimo. Tampoco acabará en puntos de silla porque también son inestables.

Podemos encontrar muchos tipos de funciones potenciales. Sin embargo, todos sufren el mismo problema de poder acabar en mínimos locales que no son la meta debido a la configuración de los obstáculos, como veremos más adelante.

Potencial atractivo/repulsivo

La función potencial más simple en Q_{libre} es la atractiva/repulsiva. La meta atrae al robot mientras que los obstáculos lo repelen. La suma de estos efectos lleva al robot a la meta de manera suave evitando los obstáculos (Figura 60). La función potencial será la suma de ambas:

$$U(q) = U_{atr}(q) + U_{rep}(q)$$

En [23, pp. 81-84] explica matemática cómo deben ser ambas funciones potenciales, combinando funciones cónicas y cuadráticas. Sin embargo, nosotros sólo nos quedaremos con que el gradiente de la función potencial de atracción en cada punto *q* será un vector con el sentido tal que se aleje de la meta y proporcional a la distancia de dicho punto a la meta. Como la fuerza de atracción es el gradiente cambiado de signo, se acercará a la meta. Cuanto más lejos esté el punto *q* de la meta, mayor magnitud tendrá el vector y la fuerza será mayor. A medida que se acerque, la fuerza será menor acercándose cada vez más despacio. Así evitamos oscilaciones al llegar a la meta adecuando la velocidad.

En la función potencial de repulsión ocurre lo mismo pero a la inversa. El gradiente será un vector en el sentido hacia el obstáculo y con una mayor magnitud a medida que se acerca a dicho obstáculo. La fuerza, al ser menos el gradiente, hará que el robot se aleje de los obstáculos. Cuanto más cerca esté de ellos, mayor fuerza experimentará, alejándose más rápido.

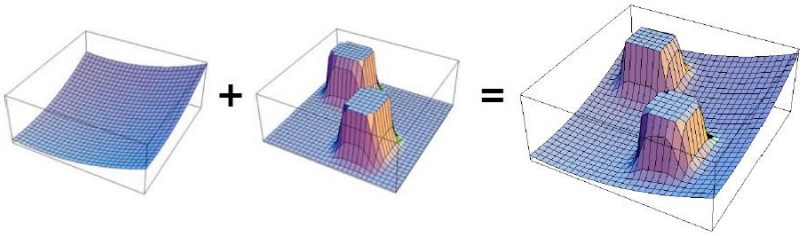


Figura 60: Funciones potenciales. Suma de atractiva más repulsiva [38, p. 46]

Implementación

En la realidad existe una cierta dificultad de medir las distancias a los obstáculos, y por tanto, obtener las funciones potenciales repulsivas. Por tanto, en [23, pp. 85-89] describe el procedimiento a realizar con los sensores que hay en el robot y en el caso de un mapa representado con una cuadrícula.

Para este segundo caso utiliza el algoritmo “Bushfire” o “Grassfire” visto en 7.2.3, introduciendo el mapa de píxeles con un valor uno para los obstáculos y cero para los libres. Podemos utilizar una conectividad de cuatro u ocho vecinos según deseemos. En el primer paso todas las casillas de valor 0 vecinas a aquellas que tengan un valor 1, les damos un valor 2. Luego, todas las casillas de valor 0 que sean vecinas a las de valor 2, les ponemos un 3 y así sucesivamente hasta completar el mapa. En la Figura 61 las líneas continuas son el diagrama de Voronoi indicando dónde chocan los frentes de los obstáculos. Se genera así un mapa con las distancias a los

obstáculos más cercanos. El gradiente descendente será un vector que apunte al vecino con el valor más cercano. Si hay varios con el mismo valor, escogemos uno.

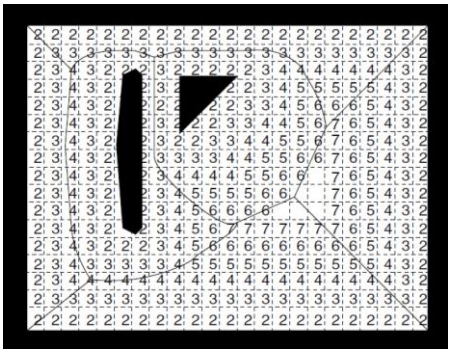
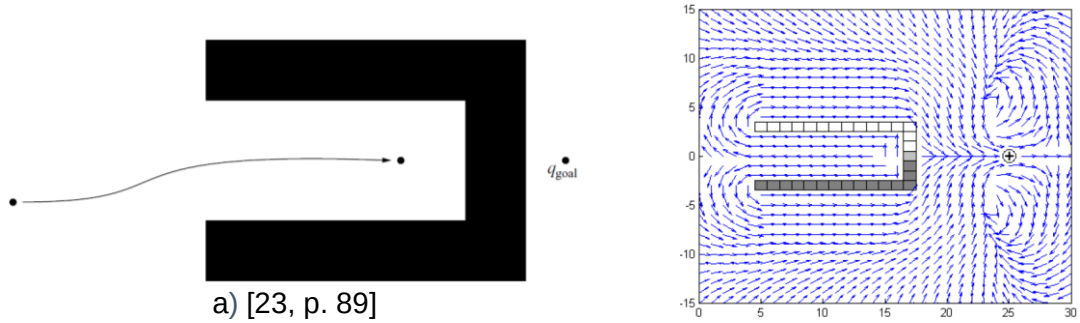


Figura 61: Brushfire [23, p. 88]

Añadiendo a esta función potencial repulsiva la de atracción, podemos ejecutar un algoritmo de planificación para obtener el camino más corto. Este método se podría aplicar a 3D utilizando cubos en lugar de casillas. La conectividad de 4 vecinos pasaría a 6 vecinos, y la de 8 a 26 vecinos. Los valores los asignaríamos de la misma forma que en 2D. Para mayores dimensiones no es factible computacionalmente.

Problema de mínimo local

Como ya hemos dicho anteriormente, los algoritmos de gradientes descendentes sufren el problema de la existencia de mínimo local en el campo potencial. Éste aparece cuando el gradiente converge a un mínimo pero no es el mínimo global sino uno local debido a la disposición de los obstáculos, como puede verse en la Figura 62a. El robot es atraído a la meta y se “mete entre los brazos” del obstáculo cóncavo. La meta sigue atrayendo al robot, pero el brazo inferior lo repele hacia arriba y viceversa. Se queda en una posición de equilibrio en la que el efecto del obstáculo contrarresta la atracción de la meta. Por tanto, el robot ha llegado a un punto *q** con gradiente nulo sin ser la meta. Esto no sólo se produce con obstáculos cóncavos, como puede verse en [23, p. 90].



a) [23, p. 89] b) [64, p. 5]

Figura 62: Problema mínimo local
a) Mínimo local b) Suma de campo rotacional

Este problema puede afrontarse de diversas maneras. Una de ellas es agregar a la función potencial básica un campo potencial rotatorio alrededor de los obstáculos [64], [65, p. 13]. Esto rompe simetrías y guía al robot para que rodee el obstáculo, permitiendo que salga de esos mínimos locales, como puede verse en la Figura 62b.

Otra forma de evitar atascarse en mínimos locales es mediante campos potenciales aleatorizados ("randomized potential fields") [23, p. 90], [45, pp. 225-226], [65, p. 13]. Éste método se basa en utilizar "paseos" aleatorios en esos casos durante un número de iteraciones. En cada iteración, cada coordenada q_i es aumentada o disminuida por una longitud de paso Δq_i basado en una probabilidad. Después de cada iteración, se comprueba que la nueva configuración no produce un choque o se sale fuera de los límites del mapa. Si sucede esto, se descarta y se realiza otro intento a partir de la previa configuración. Los fallos se repiten indefinidamente hasta que una nueva configuración en $\mathcal{Q}_{libre}$ es hallada. En [45, pp. 225-226] se explica más detalladamente este proceso.

En [23, pp. 90-92] se explica un último método para afrontar este problema: el planificador de onda frontal ("wave-front planner"), aunque sólo puede realizarse sobre cuadrículas ("grids"). Para nuestro caso consideraremos en el espacio de dos dimensiones. Podemos elegir entre una conectividad de 4 u 8 vecinos.

Inicialmente el mapa es binario, con 0 en las casillas libres y 1 en los obstáculos. Además, también sabemos la posición de la casilla de inicio y final, teniendo esta última un valor de 2. En el primer paso, todas las casillas de valor 0 vecinas a la meta se etiquetan con un 3. Luego, todas las casillas de valor 0 vecinas a las de valor 3 se les pone un 4 y así sucesivamente hasta llegar a la casilla de inicio (Figura 63). El algoritmo de planificación determina el camino mediante un gradiente descendente empezando en la casilla inicial. Si hay múltiples opciones en torno a una casilla, cogemos una cualquiera.

Se asemeja a una onda u ola que crece desde la meta en cada iteración. Todos los píxeles en el frente de la onda tienen la misma longitud de camino hasta la meta. Por tanto, esencialmente es una función potencial con un mínimo local. Sin embargo, posee dos desventajas: puede acercarse mucho a los obstáculos ya que no hay una función de repulsión, y que el método se expande por todo el espacio libre para buscar la solución.

Al igual que en el Brushfire, se puede generalizar a mayores dimensiones, como en 3D, teniendo voxels (píxeles de 3 dimensiones) con 6 caras. La implementación en mayores dimensiones no es factible computacionalmente.

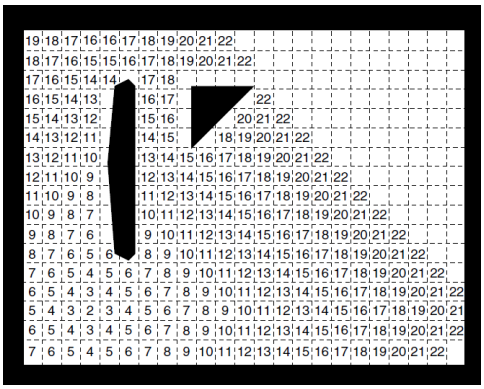


Figura 63: Planif. onda frontal (inicio en esq. sup. dcha y meta en 2) [23, p. 92]

8 LOCALIZACIÓN Y GENERACIÓN DE MAPAS

8.1 FILTRO DE KALMAN

Hasta ahora hemos supuesto un conocimiento exacto del mapa o sensores que proporcionan una información perfecta del entorno. Sin embargo, en la realidad no ocurre así ya que dichas medidas son imperfectas introduciendo "ruido". El Filtro de Kalman proporciona un método recursivo para estimar el estado del sistema dinámico en presencia de ruido. Para ello, da una estimación del vector de estado ($\hat{x}$) y una matriz de covarianza (*P*) con el error de estimación. Esto significa que devuelve una función de densidad de probabilidad (PDF) Gaussiana de media $\hat{x}$ y covarianza *P*. Para el caso de la localización del robot, da una distribución de las posiciones probables en las que se puede encontrar el robot en lugar de una estimación de una posición única [23, p. 269], [12, p. 113].

En [2, pp. 235-238] se da una explicación sencilla de los pasos que realiza este método en un robot móvil. Primero hace una predicción de la posición con el modelo de movimiento de error Gaussiano a la medida dada por el *encoder*. Entonces, en la fase de observación recoge datos con sus sensores de características del ambiente, como líneas, puertas, columnas… Al mismo tiempo, basado en la predicción de la posición en el mapa, genera una predicción de medida que identifica las características que el robot que debería encontrar y sus posiciones. En la etapa de comparación identifica los mejores emparejamientos entre las características extraídas en la observación y las predichas. Finalmente, el Filtro de Kalman une la información de esas comparaciones para actualizar su estimación de la posición.

8.1.1 Estimación probabilística

En este sub-apartado explicaremos un poco el concepto de estimación probabilística para el caso de estimar la localización de un robot móvil, estando bien desarrollado en [23, pp. 270-272]. En lugar de dar una hipótesis de dónde puede estar el robot, da una distribución de probabilidad de la zona en la que puede estar, por eso es un algoritmo probabilístico. Así, podemos tener en cuenta las incertidumbres de los modelos del movimiento del robot y de las lecturas del ruido de los sensores.

Una forma simple de localización es integrando los comandos de velocidad del robot desde una posición inicial conocida. Cuando se ejecutan perfectamente y la posición es conocida con exactitud, el método da una estimación perfecta. Sin embargo, esto no es real, ya que hay errores que se van acumulando, perdiendo información de su posición real. Otro caso similar es integrar las medidas de velocidad del robot dadas por sensores de odometría vistos en 6.1.1.

Debido a que la posición inicial tampoco es exacta, se utiliza una PDF (función de densidad de probabilidad) con las posiciones posibles. Si tenemos una buena estimación, la PDF tendrá un pico elevado en esa localización y con gran pendiente, y al revés en caso contrario. Ahora, conforme el robot se mueve hay que propagar dicha función. Para ello se integran los comandos de velocidad, pero como no son exactos, la estimación posee cada vez más incertidumbre y el pico se hace más pequeño y más esparcido hasta que llega a ser despreciable.

Para solucionarlo, se debe proporcionar nueva información a través de los sensores exteroceptivos. Un ejemplo sería obtener información a través de una marca o baliza de posición conocida. Esta nueva información se utiliza para actualizar y corregir la información que ya teníamos, produciendo una estimación más precisa. Este proceso se debe repetir con gran frecuencia para corregir los errores acumulados. De ahí la importancia de tener información exacta del entorno mediante marcas, balizas, otros sensores…

8.1.2 Filtro de Kalman lineal

El Filtro de Kalman es muy utilizado por su facilidad de implementación en sistemas lineales. Las ecuaciones utilizadas para modelar la dinámica y los sensores del robot en sistemas de tiempo discreto con un periodo de muestreo *T* son:

$$x(k+1) = F(k)x(k) + G(k)u(k) + v(k)$$

$$y(k) = H(k)x(k) + w(k)$$

El vector *x(k)* denota el estado del robot conforme pasa el tiempo. El vector *u(k)* representa las entradas aplicadas al sistema como comandos de velocidad, pares, fuerzas… El vector *y(k)* es la salida del sistema que contiene los valores proporcionados por los sensores. Las matrices *F(k), G(k)* y *H(k)* describen la dinámica del sistema, la relación entre las entradas y la dinámica, y la relación entre los estados

y las salidas, respectivamente. Por último, *v(k)* y *w(k)* son los ruidos que afectan al sistema y los de las medidas, respectivamente. Ambos se asumen ruido blanco (valor en instante *k* es independiente de *k-1*) Gaussiano, de media cero y matrices de covarianza *V(k)* y W*(k)*.

El objetivo del Filtro de Kalman es determinar la mejor estimación del estado *x* en instante *k* dada una estimación previa junto con una entrada *u* conocida y la salida *y*. Para ello, hay que solucionar dos problemas: la presencia de ruido desconocido y no medible, por lo que hay que filtrarlos; y que el estado en general no se puede observar o predecir directamente de las salidas porque *H(k)* puede no ser invertible. Por tanto, hay que estimarlo utilizando un observador, que utiliza la historia de las señales de salida y entrada junto con las matrices de las ecuaciones anteriores.

El Filtro de Kalman, además de dar una estimación del vector de estado $\hat{x}(k|k)$, también proporciona una estimación del error de covarianza $P(k|k)$, asociado a dicha estimación. Esto implica tener que trabajar con un observador que utilice distribuciones de probabilidad.

Debido a la complejidad creciente del desarrollo y fuera del alcance de este trabajo, no seguiremos explicando más el Filtro de Kalman. En la siguiente bibliografía podemos encontrar una explicación más profunda y matemáticamente desarrollada [23, pp. 272-289], [12, pp. 114,529-532], [45, pp. 615-617], [63, pp. 35-45,52-58]. Además, también se trata el Filtro de Kalman extendido (EFK) para sistemas no lineales, lo cuales encontramos abundantemente en la práctica.

8.2 LOCALIZACIÓN CON MARKOV

La estimación vista en Kalman asume que el error en los sensores y la posición del robot sigue una función de densidad de probabilidad Gaussiana. Sin embargo, la localización de Markov considera cada una de las posibles poses del robot en el espacio de configuraciones y utiliza una distribución de probabilidad explícitamente especificada para cada una de ellas. Las acciones y procesos de actualización de la información con los sensores vistas en Kalman se deben realizar para actualizar la probabilidad de cada una de las poses posibles [2, pp. 212,213].

Con Kalman debíamos partir de una posición conocida. Sin embargo, el método de Markov permite la localización empezando de una posición desconocida o

recuperarse de situaciones ambiguas ya que puede seguir la pista a muchas posiciones posibles distintas. Para actualizar la probabilidad de todas esas posiciones, debemos discretizar el espacio de configuraciones en un número finito de posibles poses. Cuanto mayor sea el número de posibles poses, mayor será el gasto computacional y la memoria requerida aunque los avances tecnológicos permiten que sea un problema cada vez menor. La discretización puede ser en forma de mapa topológico o de cuadrícula.

El proceso realizado para la localización es similar a Kalman. La actualización de la acción consiste en estimar una nueva posición basada en un estado previo y las medidas de un sensor propioceptivo. La actualización de la percepción reside en refinar la estimación de la posición actual comparando la estimación que acabamos de hacer con los datos recibidos de sensores exteroceptivos. La diferencia con Kalman reside en la actualización de la percepción ya que se basa en la fórmula de Bayes (probabilidad) para actualizar cada una de las posibles posiciones que puede tener el robot creando una distribución multimodal, es decir, un valor máximo para cada posible posición. En Kalman hacemos la actualización para crear una distribución Gaussiana unimodal, es decir, con sólo un valor máximo.

En resumen, con Markov la función de densidad de probabilidad asociada a la configuración de robot es multimodal, teniendo tantos picos como hipótesis de posición que explican los datos recogidos de los sensores. En Kalman la función de densidad de probabilidad es Gaussiana unimodal, con el máximo en la posición con mayor probabilidad de encontrar el robot.

8.3 MÉTODO DE MONTE CARLO

El método de Monte Carlo o también llamado Filtro de Partículas se puede considerar dentro del tipo de localización de Markov, explicándolo más detalladamente ya que hay más información de este método en la bibliografía. Puede manejar las múltiples hipótesis de los estados del sistema y no hace suposiciones sobre la distribución de los errores [12, pp. 125-128], [63, pp. 83-85], [23, pp. 316-321], [31, pp. 18-19], [2, pp. 225-227]. Básicamente consiste en mantener muchas versiones diferentes (que varíen ligeramente) del vector de estados del robot. Cuando obtenemos una nueva medida, puntuamos cómo cada versión explica los datos nuevos. Hacemos copia de los vectores que mejor se ajusten y los perturbamos aleatoriamente para generar

nuevos estados candidatos. Estos miles de estados posibles y sus puntuaciones definen conjuntamente la PDF que estamos buscando estimar.

Un gran problema de este método es que el número de estados de muestra (versiones) que tenemos generar aumentan geométricamente con el número de estados a estimar. Por ejemplo, para un vector 1D podemos usar 100 muestras, pero para 2D unas 100^2 y 100^3 para 3D, siendo computacionalmente costoso. Si dibujamos estos vectores de estados como puntos, obtenemos "nubes" de partículas, y por eso a este tipo de estimador también se le llama Filtro de Partículas.

Consideraremos el problema de localización utilizando tres estados: (x, y, θ). Los pasos a seguir en este algoritmo son:

1. Inicializar creando *n* partículas en un mapa distribuidas aleatoriamente. Cada una será un vector de estado 3x1 del vehículo y con el mismo peso o probabilidad inicial $w_i = 1/N$.
2. Actualizar los estados de cada partícula según una medida tomada. También añadimos un vector aleatorio que representa incertidumbre en la posición del vehículo. Se puede utilizar cualquier distribución, aunque normalmente se utiliza una variable aleatoria Gaussiana.
3. Para cada partícula predecimos una observación de una característica del entorno. Para cada una comparamos el error entre la predicción y la medida tomada. Utilizamos una función de probabilidad que proporciona un valor escalar w de lo bien que cada partícula explica la observación. Dicho escalar lo llamaremos peso o importancia y cuanto mayor sea, mejor.
4. Escogemos las partículas que mejor explican la observación. Una forma es ordenarlas según w y seleccionar las mejores, concentrando las futuras muestras en los picos de las funciones de densidad de probabilidad, pero esto reduciría el número de partículas. Otra solución es copiar los *n* mejores, pero sin embargo, artificialmente reduciríamos la dispersión de las partículas. En lugar de eso, muestreamos aleatoriamente de las muestras ordenadas según w. Esto hace que haya una oportunidad finita de que las partículas que no explican bien la observación sean reelegidas. Puede ser una buena estrategia ya que funciona mejor de cara a futuras observaciones por si el robot recibe medidas de los sensores improbables, evitando fallar en la localización.

5. Volver al paso 2.

Este método pertenece a la clase de algoritmos computacionales que confían en tomar muestras aleatorias repetidamente para calcular el resultado. Se suele utilizar cuando es inviable calcular un resultado exacto con un algoritmo determinístico. Podemos encontrar los algoritmos más detallados en [23, pp. 317-318].

En la Figura 64 podemos ver un ejemplo de este método para la localización global de un robot. En la imagen de la izquierda, se inicializa el filtro creando 10000 muestras distribuidas uniformemente. En la imagen central tenemos la distribución de partículas después de 10 medidas con un sensor de ultrasonidos. Después de 65 medidas, el Filtro de Partículas ha convergido a las configuraciones cercanas a la posición real del robot, como puede verse en la imagen de la derecha.

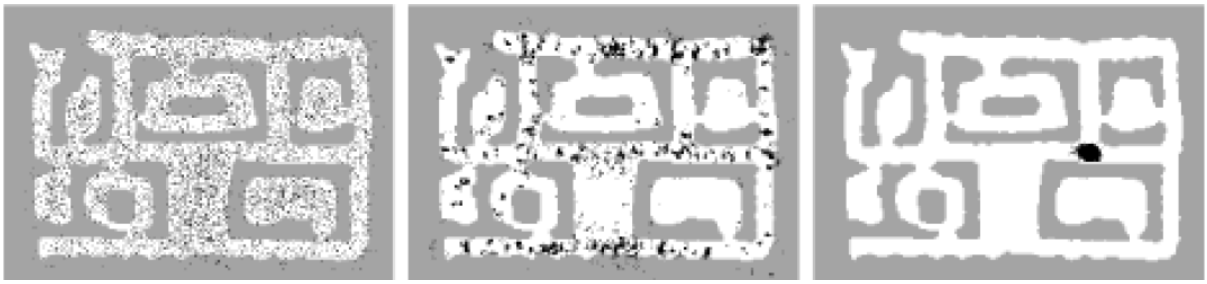


Figura 64: Filtro de Partículas (10000 muestras)
Izq: distribución inicial, centro: después de 10 medidas (ultrasonidos), dcha: después de 65 medidas [23, p. 320]

8.4 GENERACIÓN DE MAPAS

Hasta ahora, hemos realizado la localización en mapas conocidos, es decir, con información de la disposición de los obstáculos. Sin embargo, una de las habilidades importantes de un robot móvil autónomo es ser capaz de empezar en un punto cualquiera y explorar el entorno con sus sensores, aprender de él, interpretar la escena, construir un mapa y ser capaz de localizarse en él. Debido a que los sensores tienen un rango limitado, para construir el mapa y actualizarlo debe hacerlo mientras se mueve y se localiza en él. En la comunidad robótica este problema se llama Localización y Mapeo Simultáneo o SLAM (“Simultaneous Localization and Mapping”) con abundante bibliografía sobre ello, por ejemplo: [2, pp. 250-256], [23, pp. 294-297,337-345], [63, pp. 78-82], [12, pp. 123-124], [24, pp. 387-392], [31, pp. 5-6,11-30], [66, pp. 3-5], [67, pp. 9-11], [68].

Se suele caracterizar como el problema de la “gallina y el huevo”, debido a la relación entre ambas partes del problema: se necesita un mapa exacto para localizar el agente, pero el agente necesita saber su localización exacta para construir un mapa exacto. Para solucionar esto, se han desarrollado muchas técnicas, pudiendo destacar el emparejamiento de mapas (“map matching”) o aquellas que utilizan métodos probabilísticos, como el Filtro de Kalman o de Partículas ya vistos, además de otras variantes como el Filtro de Kalman Extendido (EFK) o el Filtro de Partículas Rao-Blackwellized (Figura 65). Debido a que los desarrollos son muy largos y fuera del alcance de este trabajo, no los seguiremos explicando, pudiendo encontrarlos en la bibliografía citada. Sin embargo, sí hablaremos de los problemas, representaciones y aplicaciones del SLAM.

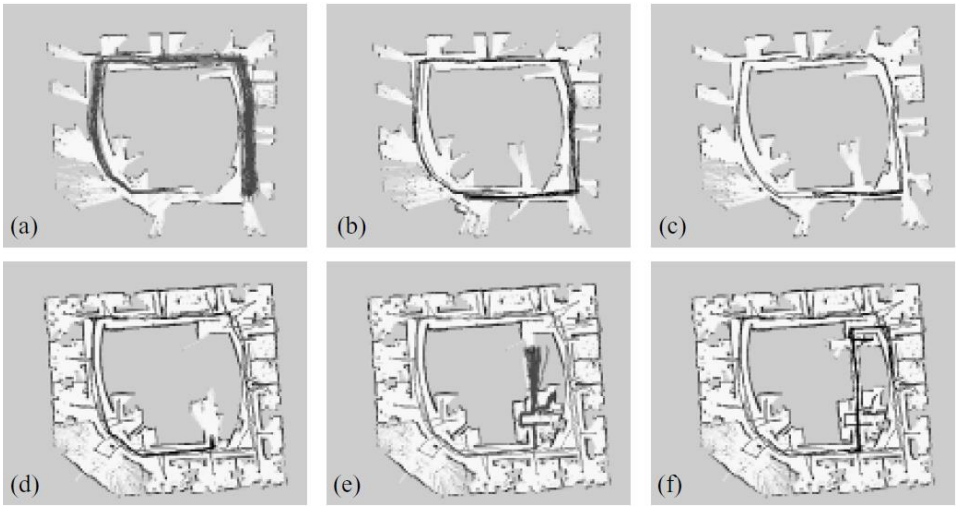


Figura 65: Secuencia mapas con filtro partículas Rao-Blackwellized [23, p. 344]

Un problema importante del SLAM es el lazo cerrado, es decir, dado un mapa por el que se puede llegar al mismo sitio de varias formas. Como los mapas creados no son perfectos, es difícil crear un mapa global que sea consistente para todo el entorno y que el robot sea capaz de reconocer que ya ha pasado por el mismo sitio. Las imperfecciones locales se van acumulando, conduciendo a un error global importante entre el mapa y el mundo real, lo cual afecta a los ciclos o lazos cerrados, como puede verse en la Figura 66 en la que después de una vuelta, no coinciden los datos observados ahora con los previos, creando una doble pared y por tanto, un mapa inconsistente.

Una solución es basarse en representaciones topológicas más simples, ignorando muchos detalles irrelevantes. Cuando el robot llega a un nodo topológico que puede ser el mismo que uno ya visitado debido a unas características distinguibles, deduce

que ha vuelto al mismo nodo. Para comprobarlo, se mueve a un nodo vecino para ver si las lecturas de los sensores son consistentes con la hipótesis del lazo [2, pp. 254-255].

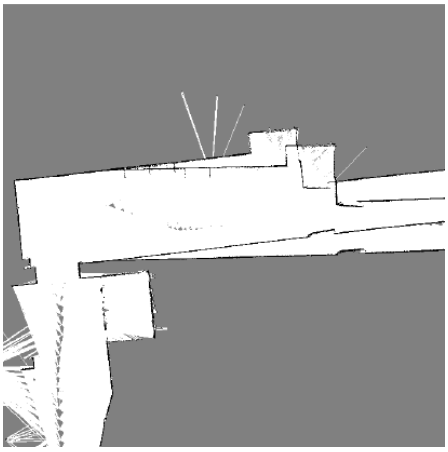


Figura 66: Mapa inconsistente con SLAM en ciclo cerrado [31, p. 21]

Otro problema importante del SLAM son los ambientes dinámicos para los métodos probabilísticos ya que suponen que los cambios son nulos o muy pequeños. Sin embargo, en la mayoría de los casos no es así, debido a que se abren y cierran puertas, las cosas se cambian de sitio en almacenes, pasan personas, etc. En estos ambientes se debería detectar los cambios y poder actualizarlos en el mapa. Para ello, puede hacerse el mapeo continuamente pero solo si es en tiempo real e incremental, ya que si requiere una optimización global offline, no se podrán satisfacer los ajustes incrementales. Una posibilidad es tener en cuenta qué características mapear ya que posiblemente las personas no es necesario debido a que son cambios muy rápidos que no afectan a un mapa más “estáticos” [2, pp. 255-256].

Se han utilizado diversos tipos de mapas con SLAM, destacando dos categorías: mapas basados en referencias y mapas de ocupación de cuadrículas [31, pp. 11-14]. Los primeros se basan en referencias o marcas que son características del entorno. Pueden ser tanto artificiales (esquinas, bordillos, líneas, puntos…) como naturales (árboles) distribuidas en el espacio. Se asumen sin ambigüedades, dispersas y cada una asociada a una posición en el espacio. Una dificultad de este tipo de mapa es asociar la referencia observada con la correcta posición en el mapa.

Los mapas de ocupación de cuadrículas discretizan el entorno en una cuadrícula de dos dimensiones generalmente. Cada celda puede representarse como un bit binario simple o parte de una estructura de un árbol. Asimismo, cada una tiene una posición y tamaño predefinido. El conjunto de todas las celdas forma el mapa, similar a los

píxeles de una imagen digital. Se basa en determinar si cada celda está ocupada o libre. Para ello, utiliza sensores que lo comprueban sin necesidad de hacer referencia al resto del mapa. En [24, pp. 381-383] se explica más detalladamente el mapeo en este tipo de mapas.

Una ventaja importante de este tipo de mapas es la facilidad de visualizarlo ya que no es abstracto como el mapa de referencias, y se asemeja a los planos en planta de un edificio (Figura 67). Por tanto, será más fácil de interpretar por las personas y es más simple la planificación de los caminos. Otra ventaja es el nivel de detalle, dependiente de la resolución deseada, pero normalmente, elevado. Sin embargo, hay que tener en cuenta la gran cantidad de memoria que requiere, por lo tanto, a mayor resolución y nivel de detalle, más memoria necesitaremos y menos mapas podremos almacenar.

Podemos encontrar un gran número de aplicaciones para robots capaces de navegar sin mapas previos ni infraestructuras externas como GPS, como exploración en sitios cerrados, otros planetas (Marte), búsquedas en zonas submarinas o subterráneas, zonas de desastres para rescate [63, p. 78], [31, p. 6].

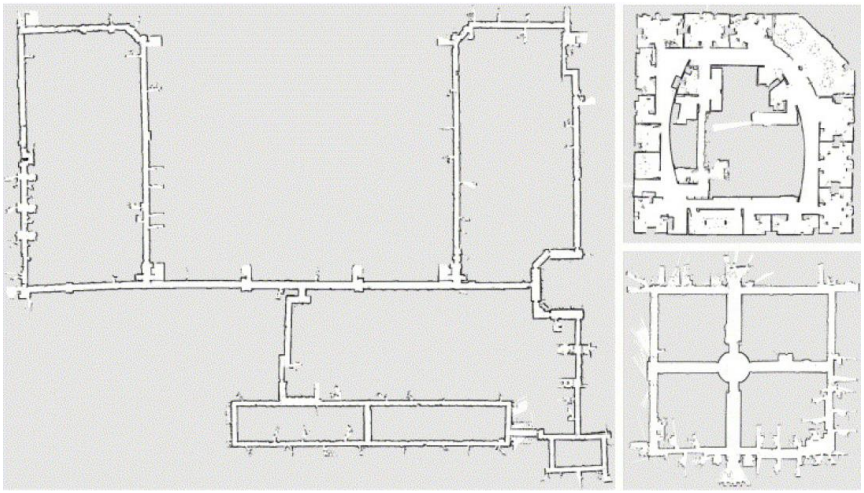


Figura 67: Mapa de cuadrícula obtenido con SLAM [31, p. 14]

9 SIMULACIÓN

En primer lugar, realizaremos una simulación con Simulink de seguimiento de trayectoria pura y después simulaciones de los algoritmos de planificación mediante los códigos proporcionados por algunos autores libremente.

9.1 SEGUIMIENTO DE TRAYECTORIA

En 4.2 vimos el procedimiento de seguir una trayectoria de referencia e incluimos un ejemplo en Simulink. Sin embargo, también existe la opción de seguir un punto objetivo en el plano que va describiendo un camino a una velocidad constante. A este problema lo llamaremos persecución pura, pudiéndose encontrar explicado en [12, pp. 74-75] y [6, pp. 261-267].

El modelo de Simulink de la Figura 68 es proporcionado libremente por [12] en su toolbox con el nombre "*sl_pursuit*". Ha sido creado para modelar una bicicleta, que es una aproximación de un vehículo tipo Ackerman, es decir, de cuatro ruedas a semejanza de un coche. El vehículo seguirá un punto que se mueve alrededor de un círculo de radio unidad con una frecuencia de 0.1 Hz.

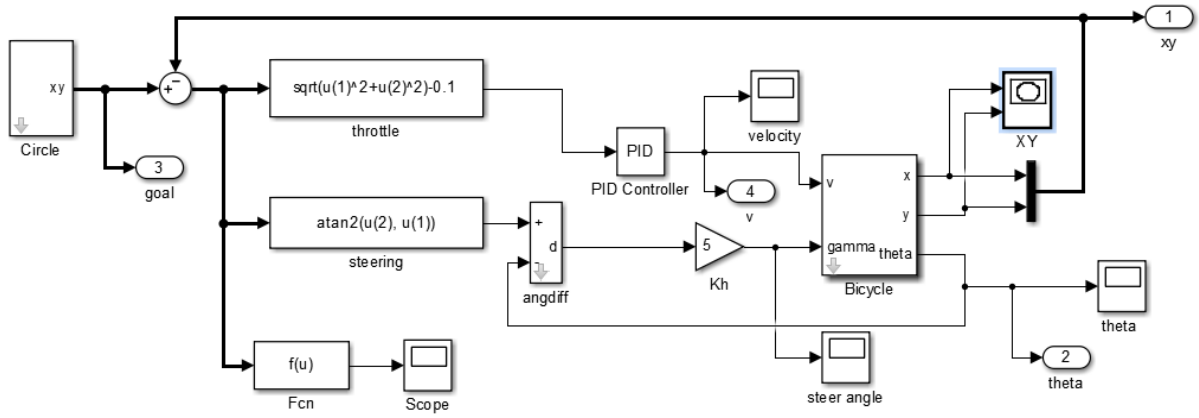


Figura 68: Modelo Simulink persecución pura [12, p. 74]

El robot mantendrá una distancia d^* detrás del punto de persecución y tendremos el siguiente error:

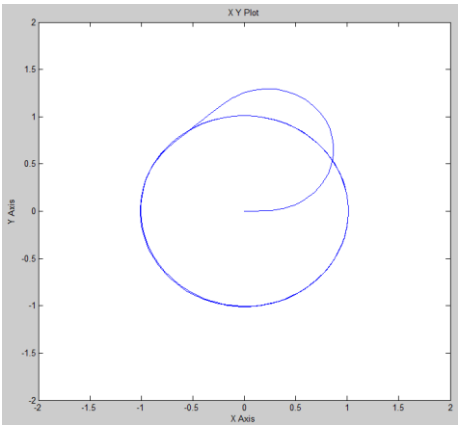
$$e = \sqrt{(x^* - x)^2 + (y^* - y)^2} - d^* \qquad (9.1)$$

Controlaremos la velocidad del robot mediante un controlador PI:

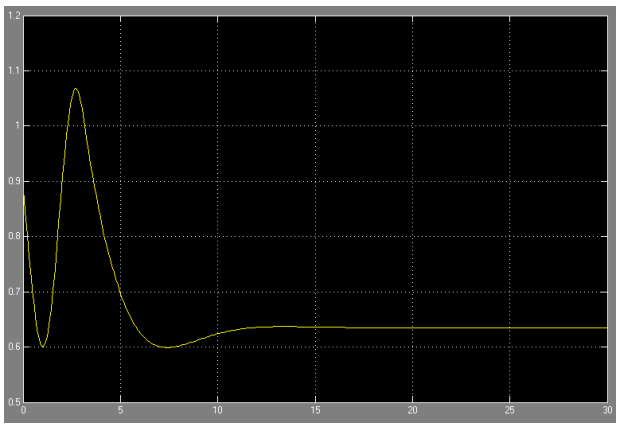
$$v^* = K_v e + K_i \int e\, dt \qquad (9.2)$$

El término integral es necesario para dar un valor finito a la velocidad cuando el error se ha anulado. El segundo controlador gira el robot hacia el objetivo que está a un ángulo relativo θ^* y emplea un controlador proporcional sencillo.

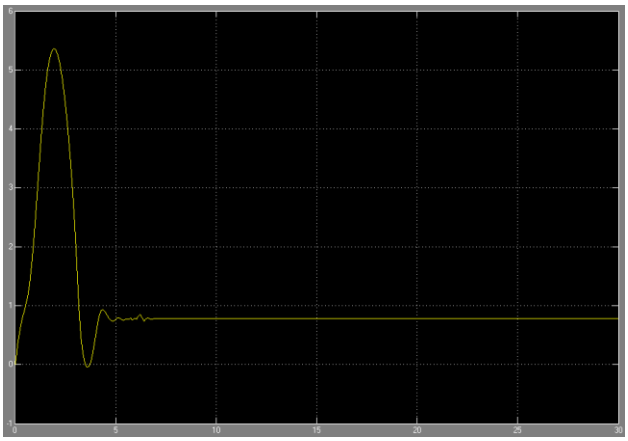
$$\theta^* = \text{atan}\frac{y^* - y}{x^* - x} \qquad (9.3)$$



a)



b)



c)

Figura 69: Resultados persecución pura

a) Camino en plano x-y b) Velocidad c) Ángulo de dirección

En la Figura 69a observamos que el robot empieza en el origen pero alcanza y sigue al objetivo móvil. En las Figura 69b y c se puede ver la evolución de la velocidad y de la dirección a lo largo del tiempo. Inicialmente sufren grandes variaciones para luego alcanzar sendos valores de régimen permanente.

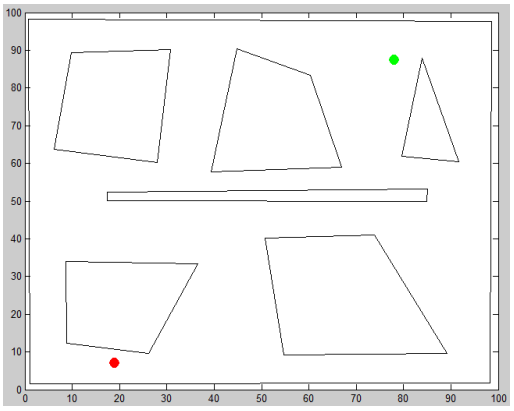
9.2 VORONOI

Como ya vimos en 7.2.1, el algoritmo de Voronoi es un método de planificación basado en la geometría. El código de [41] utiliza Voronoi como pre-procesamiento para obtener el grafo, y luego recurre a Dijkstra para hallar el camino óptimo. A partir de unos mapas ya dibujados, aunque nosotros también podríamos generarlos, el primer paso es seleccionar un punto de inicio (en verde) y un punto de meta (en rojo).

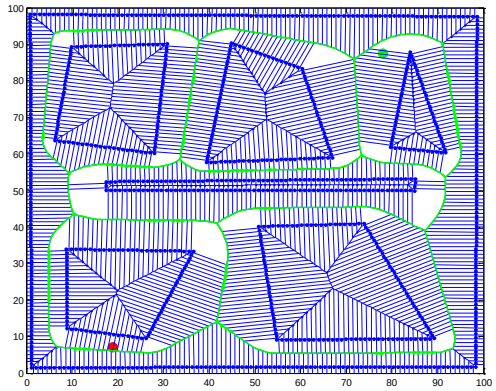
Luego, el código ejecuta dicho algoritmo con un épsilon determinado creando el diagrama de Voronoi en verde. Éste parámetro indica la distancia entre dos nodos consecutivos, siendo dichos nodos la subdivisión de los obstáculos poligonales. Cuanto menor sea, más nodos habrá y el camino será más suave, pero conllevará un mayor gasto computacional. En los ejemplos simulados, el Épsilon vale 1.

Una vez tenemos el diagrama de Voronoi generalizado dibujado, ejecutaremos el código correspondiente para obtener el camino final más corto en color magenta. Si el estado inicial no pertenece al diagrama, irá en dirección perpendicular al punto más cercano de dicho diagrama. Con el estado final realizará el mismo proceso yendo desde el punto más cercano del diagrama a la meta.

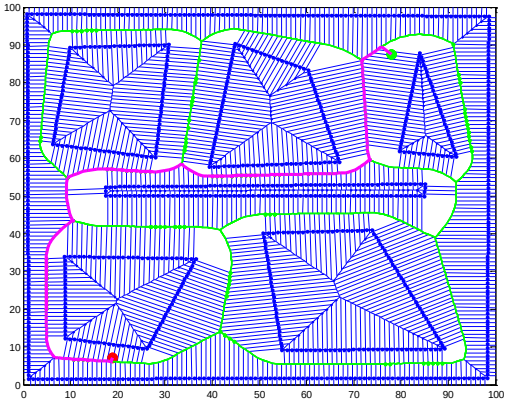
Simulación 1



a)



b)

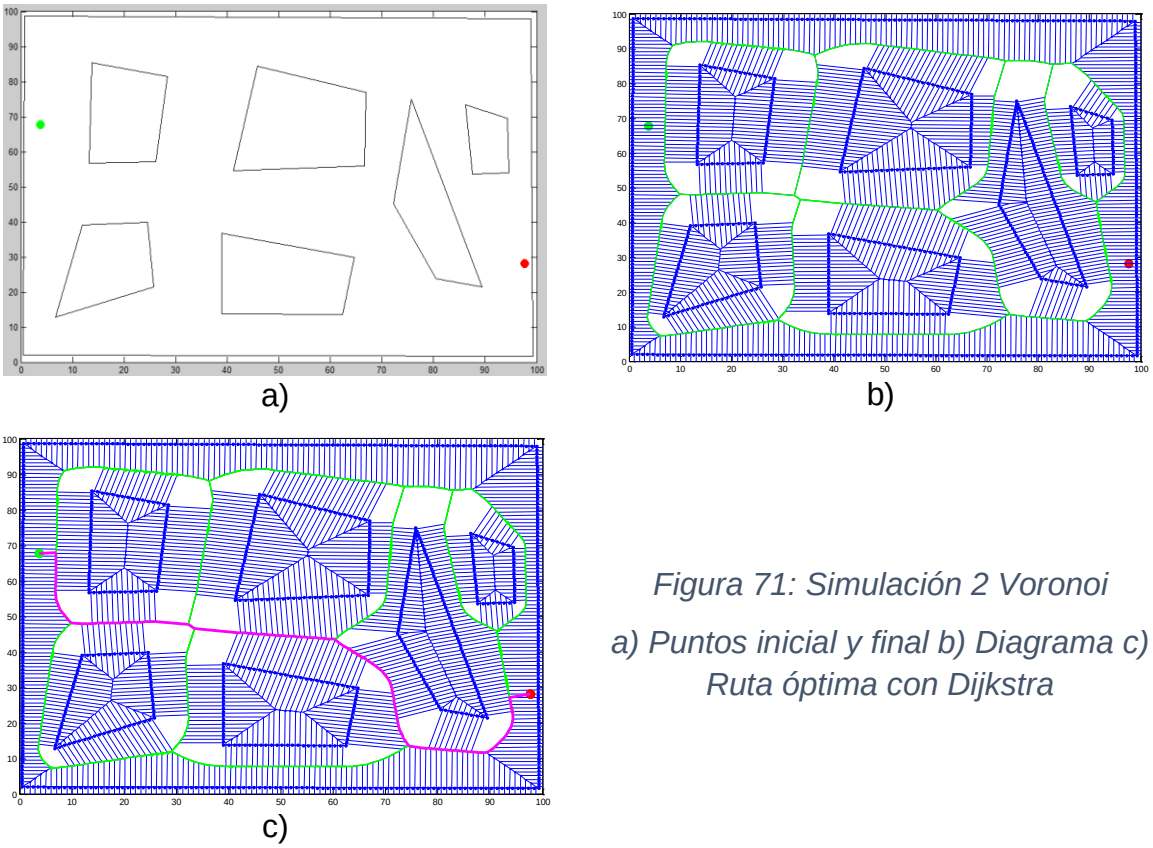


c)

Figura 70: Simulación 1 Voronoi

a) Puntos inicial y final b) Diagrama c) Ruta óptima con Dijkstra

Simulación 2



a) b) c)

Figura 71: Simulación 2 Voronoi

a) Puntos inicial y final b) Diagrama c) Ruta óptima con Dijkstra

9.3 DIJKSTRA EN CUADRÍCULA

El algoritmo de Dijkstra se basaba en ir cogiendo las celdas con menor coste respecto al inicio para estudiar sus vecinos e ir guardando los de menor valor. Para cada uno de estos vecinos comparamos el valor que tenían con el nuevo valor calculado hallado mediante el coste hasta la celda "padre" y la distancia entre ambos. Si el nuevo valor calculado es menor que el previo que tenían, lo actualizamos y lo guardamos, para estudiar posteriormente sus vecinos. Así sucesivamente hasta que hemos explorado todas las celdas y no tenemos más casillas para estudiar, o hemos llegado a la meta.

El código utilizado para la siguiente simulación es proporcionado por [50]. Nos permite crear un mapa cuadrado de las dimensiones que deseemos. Las casillas con valor 0 serán blancas indicando espacio libre, y las casillas con valor 1 serán negras representando obstáculos. Para la simulación he creado unos obstáculos rectangulares arbitrarios con las coordenadas en formato (Y, X) con el origen en la esquina superior izquierda. También definiremos la casilla inicial (en verde) y la casilla final (en amarillo).

El código ejecuta todo el proceso sin pausas, por lo que tomaré unas capturas intermedias. Las casillas rojas son aquellas que hemos visitado y guardado y las azules los vecinos que estudiaremos posteriormente. Se utilizará una conectividad de cuatro vecinos (norte, sur, este y oeste). El resultado final es el camino más corto calculado por el algoritmo Dijkstra en color gris, proporcionando el número de casillas que contiene. También obtendremos el número total de casillas que hemos visitado (rojas).

Dimensiones 50x50, Inicio = [6,2], Meta = [49,30]

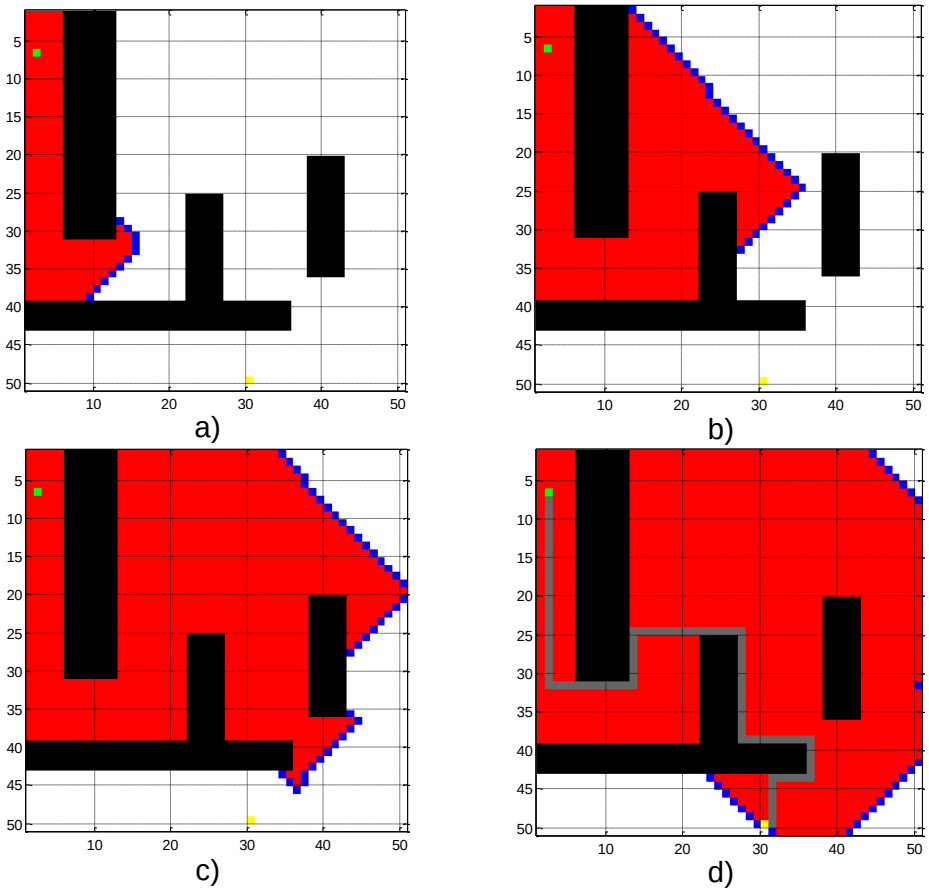


Figura 72: Simulación Dijkstra en cuadrícula

Ha expandido 1702 celdas y el camino es de 98 celdas.

9.4 A*

El algoritmo que emplearemos ahora es el A*. Haremos simulaciones con dos códigos distintos, uno más simple de [50] y otro más complejo de [69]. En ambos se explorará todas las regiones factibles entre el inicio y la meta para encontrar el camino más corto a través de dichas zonas exploradas. La exploración es dirigida hacia la meta

mediante una función heurística, tal y como vimos en la teoría de dicho algoritmo en 7.2.3.

Para el primer código, muy similar al del Dijkstra en su estructura, utilizaremos el mismo mapa con los mismos obstáculos y mismas posiciones de inicio y meta para comparar ambos algoritmos. Las casillas tendrán la misma representación en color que antes. Además, utilizaremos la misma conectividad de cuatro vecinos.

Dimensiones 50x50, Inicio = [6,2], Meta = [49,30]

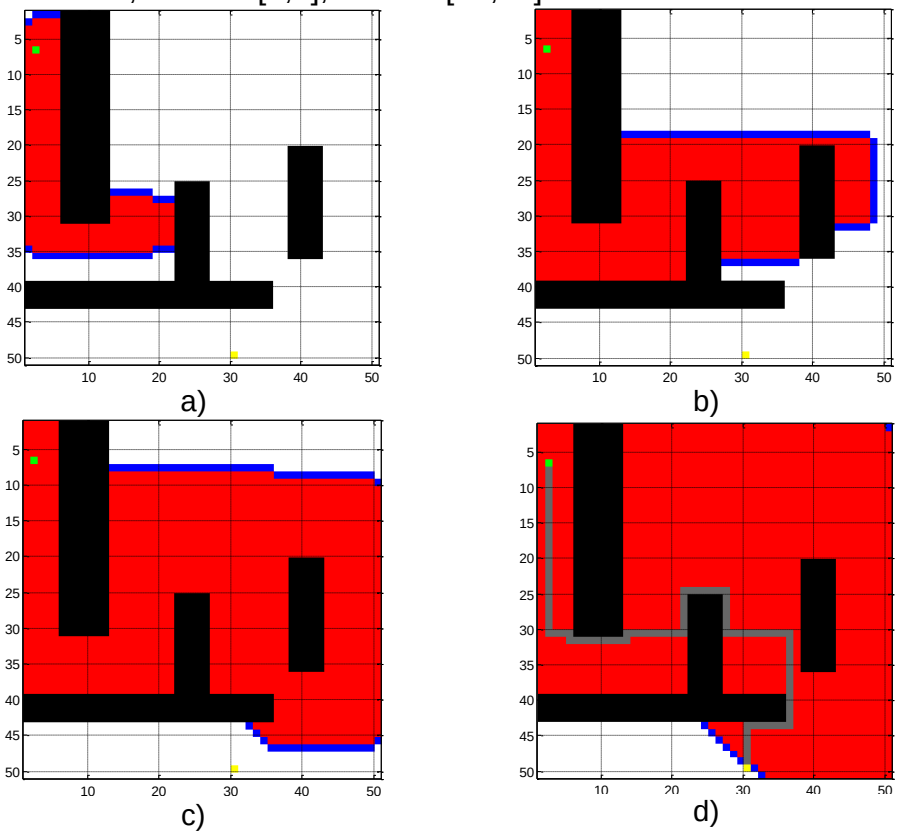


Figura 73: Simulación A en cuadrícula*

Ha expandido 1777 celdas y el camino es de 98 celdas. Por tanto, podemos ver que el A* ha expandido más celdas, lo cual no es lo usual debido a que es una búsqueda más dirigida. Es posible que se deba a la elección de obstáculos elegida. Sin embargo, el camino óptimo calculado por ambos algoritmos es de la misma longitud aunque transcurre por diferentes casillas.

El segundo código es creado por Rahul Kala, del Indian Institute of Information Technology (IIIT) Allahabad [69]. También utilizaremos los códigos del mismo autor para los siguientes algoritmos que simularemos más adelante: PRM, RRT y campos potenciales.

En general, estos códigos permiten utilizar mapas creados por el autor o crear nosotros uno cualquiera mediante una herramienta de dibujo y guardarlos en formato 'bmp' en la carpeta de trabajo. Además, podremos cambiar la posición de los puntos inicial y final, con el origen de coordenadas en la esquina superior izquierda de la imagen, y escritas en formato (Y, X). Los códigos verifican que dichos puntos no se encuentran en obstáculos, mostrando un mensaje de error en caso contrario.

En el caso del algoritmo A*, transformará el mapa discreto de píxeles en un grafo. Debido a que el mapa tiene una muy alta resolución, sería muy costoso computacionalmente considerar cada píxel como un nodo. Por tanto, reduciremos dicha resolución mediante un parámetro fácilmente accesible en el código. Cuanto mayor sea la resolución, mejores serán los resultados, discretizando el mapa en cuadrículas más finas.

Podemos aplicar distintas conectividades para conectar un nodo con los vecinos. En este código, dicha conectividad la representaremos con una matriz, en la que el 2 es la posición del robot, los 1 son las posiciones que puede alcanzar en un movimiento, y los 0 las casillas a las que no puede desplazarse. Nosotros podremos crear el movimiento que queramos para simular el diseño de nuestro robot deseado. Cuanto mayor sea la matriz y más posibilidades de movimiento tenga, más flexible será y mejor el camino resultante, pero también conlleva mayor gasto computacional.

Los costes incluirán los pesos de las aristas, tomadas como distancias Euclídeas entre los puntos a conectar. La función heurística denota la cercanía del punto a la meta también mediante distancia Euclídea.

Veremos ahora algunas simulaciones hechas con Matlab R2014a sobre los mapas proporcionados por el autor, con dimensiones 500x500. Cuando lo ejecutamos, en el mapa se van cambiando a color gris aquellas celdas que el algoritmo explora. Cuando ha finalizado y llega a la meta, veremos toda aquella región explorada. Si hacemos clic en la imagen, el algoritmo mostrará el camino óptimo. En este caso, la diferencia del tiempo procesado mostrando y no mostrando el proceso es muy grande, y además tiene en cuenta el tiempo que se tarda en dar clic sobre la imagen. Por tanto, el tiempo procesado será de otra simulación con la opción "display" off para que no muestre nada. El camino será el mismo en todas las simulaciones si es en el mismo mapa y con los mismos parámetros.

Inicio = [10,10], Meta = [490,490], Resolución X = 80, Resolución Y = 80.

Simulación 1

Conectividad = [1 1 1;
1 2 1;
1 1 1];

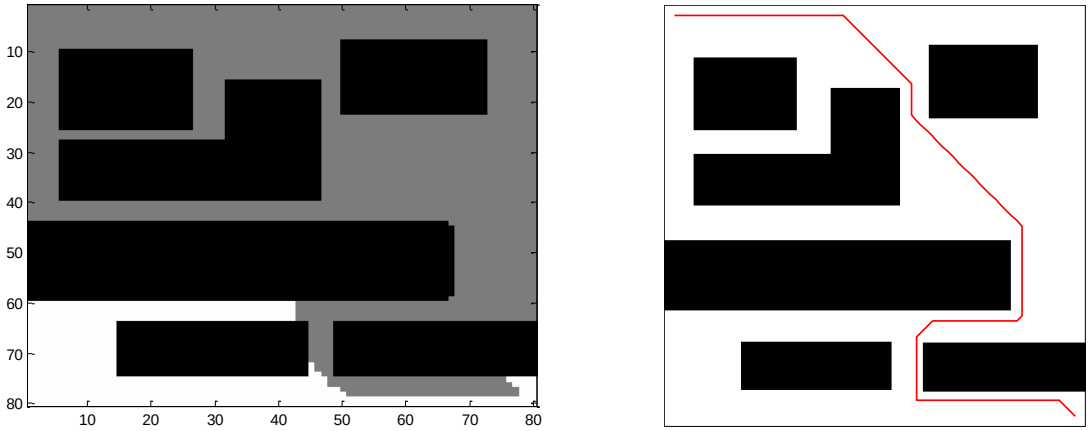


Figura 74: Simulación 1 A con [69]*

Tiempo de procesado	1.405 segundos
Longitud de camino	1049.892 píxeles

En esta simulación observamos que debe explorar casi todo el mapa cuando encuentra el obstáculo inferior debido a las funciones de coste y heurística.

Simulación 2

Conectividad = [1 1 1 1 1;
1 1 1 1 1;
1 1 2 1 1;
1 1 1 1 1;
1 1 1 1 1];

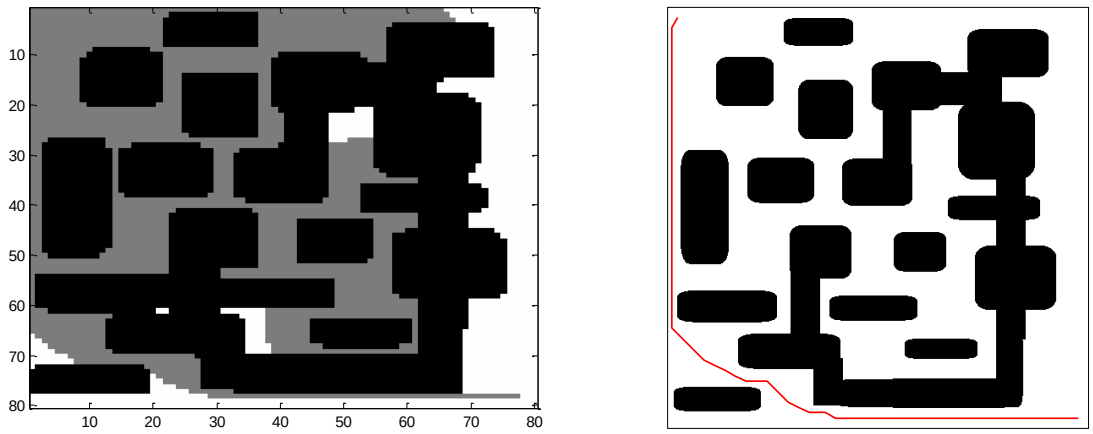


Figura 75: Simulación 2 A con [69]*

Tiempo de procesado	2.995 segundos
Longitud de camino	887.69 píxeles

El algoritmo intenta ir en diagonal, el camino recto, pero debido a los obstáculos que encuentra, se tendrá que expandir lateralmente buscando un pasillo libre.

Simulación 3

```
Conectividad = [1 1 1 1 1;
                1 1 1 1 1;
                1 1 2 1 1;
                1 1 1 1 1;
                1 1 1 1 1];
```

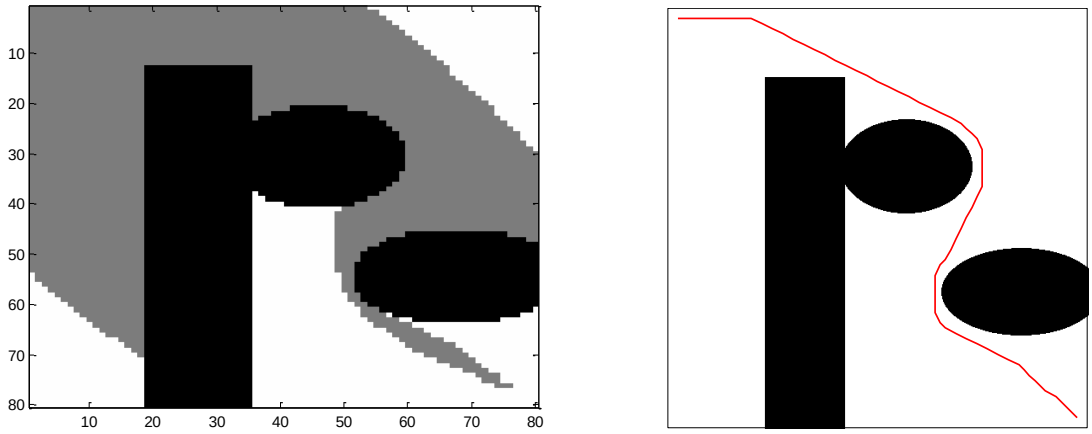


Figura 76: Simulación 3 A con [69]*

Tiempo de procesado	5.566 segundos
Longitud de camino	829.81 píxeles

Podemos observar en las dos últimas simulaciones que cuanto mejor sea la conectividad, es decir, mayor flexibilidad de movimiento del robot, mejor será la búsqueda focalizándola hacia la meta, como podemos ver en la esquina inferior derecha de esta simulación.

9.5 PRM

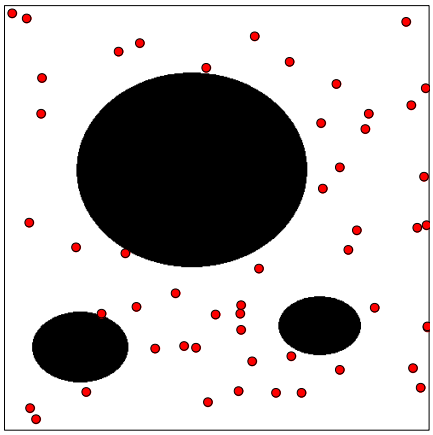
Como ya explicamos en el algoritmo, se crea un número *k* (proporcionado por el usuario) de muestras aleatorias (estados) en el mapa verificando si se encuentran en el espacio libre o en un obstáculo. El algoritmo intentará conectar todos los pares de muestras mediante líneas rectas, incorporándolas al mapa en caso afirmativo. Después buscará el camino más corto mediante un algoritmo de búsqueda en grafos, en este caso el A*. Al finalizar, se mostrará en la ventana de comandos el tiempo que ha tardado en el procesamiento y la longitud del camino final.

Veremos ahora algunas simulaciones hechas con Matlab R2014a sobre los mapas proporcionados por el mismo autor del código anterior, con dimensiones 500x500.

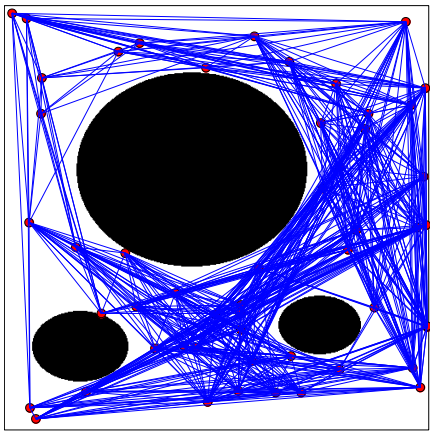
Cuando lo ejecutamos, veremos el mapa con el número de muestras introducido. Tendremos que hacer clic en la imagen para que siga transcurriendo el algoritmo y muestre ahora las líneas de unión entre las muestras. Cuando lo ha realizado, volveremos a hacer clic en la imagen para que muestre el camino óptimo. Debido a que tarda un cierto tiempo en mostrar los resultados y tiene en cuenta los tiempos que se tarda hasta hacer clic en la imagen para ejecutar los siguientes pasos, los resultados finales de tiempo y longitud de camino lo haremos para otras muestras aleatorias y con "display" off, eliminando esos tiempos intermedios.

Simulación 1

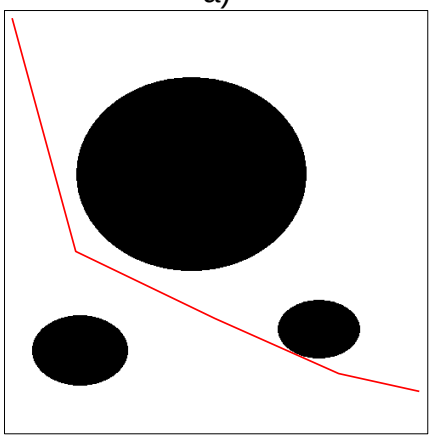
k=50, Inicio = [10,10], Meta = [450,490].



a)



b)



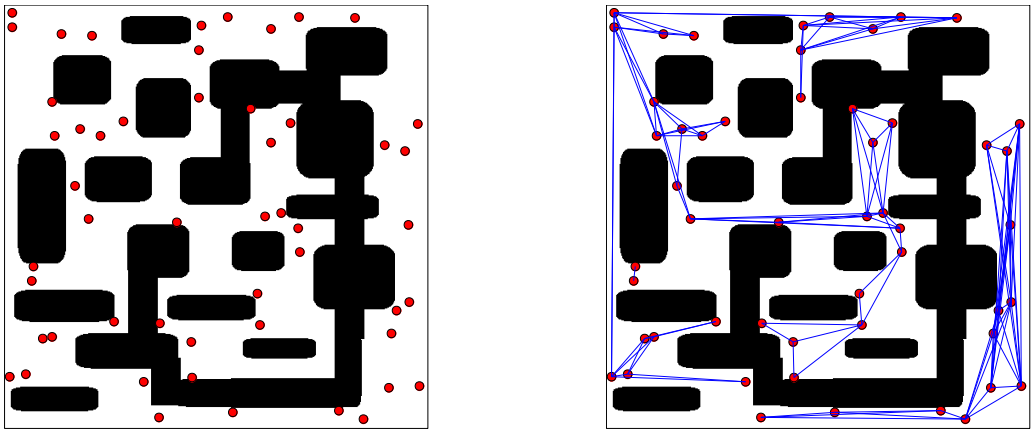
c)

Figura 77: Simulación 1 PRM

Tiempo de procesado	3.399 segundos
Longitud de camino	715.228 píxeles

Simulación 2

k=50, Inicio = [10,10], Meta = [450,490].



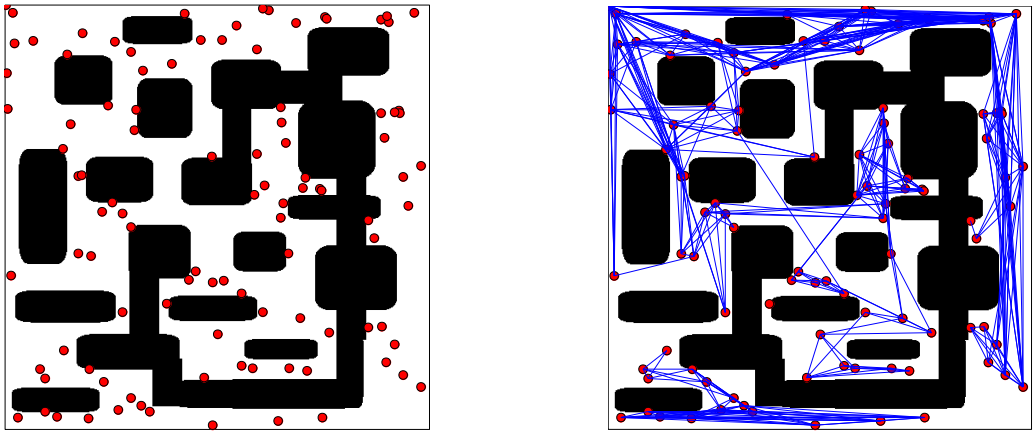
a) b)

Figura 78: Simulación 2 PRM

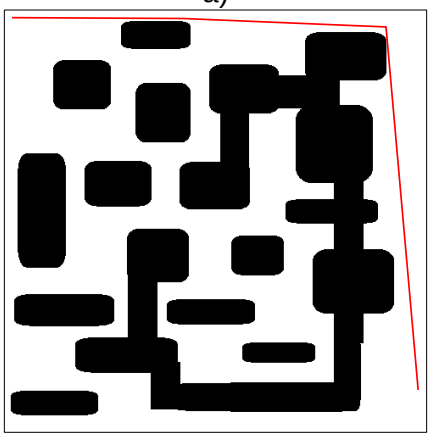
No encuentra un camino debido a muestras insuficientes.

Simulación 3

k=100, Inicio = [10,10], Meta = [450,490].



a) b)



c)

Figura 79: Simulación 3 PRM

Tiempo de procesado	4.558 segundos
Longitud de camino	888.82 píxeles

9.6 RRT

El código creado por [69] para el RRT solo crea un árbol con el nodo inicial en el punto de inicio. En cada iteración el árbol se expande seleccionando un estado aleatorio y extendiendo el nodo del árbol más cercano hacia dicho estado aleatorio con un pequeño paso. Si es factible dicho paso, lo incorpora al árbol, y si no, es rechazado y se vuelve a intentar. Cada nodo del árbol es un punto (estado) en el espacio de trabajo, que en este caso, son posiciones en el plano.

El algoritmo sigue funcionando hasta que una expansión permite lleva al árbol lo suficientemente cerca de la meta o se han realizado 10000 intentos fallidos (valor por defecto en el código que nosotros podemos modificar). El tamaño del paso es otro parámetro del algoritmo modificable, por defecto está en 20. Además, el árbol puede ser dirigido a la meta seleccionándola como el estado aleatorio con una cierta probabilidad.

Una vez tenemos el árbol construido, haremos clic en la imagen para que se ejecute un algoritmo de búsqueda en grafos. En este código utiliza el A*. Al finalizar, se mostrará en la ventana de comandos el tiempo que ha tardado en el procesamiento y la longitud del camino final.

Veremos ahora algunas simulaciones hechas con Matlab R2014a sobre los mapas proporcionados por el autor del código, con dimensiones 500x500. Al igual que en los otros casos podemos construir nosotros nuestros propios mapas. Debido a que tarda un cierto tiempo en mostrar los resultados y tiene en cuenta el tiempo transcurrido hasta hacer clic en la imagen para ejecutar el siguiente paso, los resultados finales de tiempo y longitud de camino lo haremos para otras muestras aleatorias y con “display” off, eliminando esos tiempos intermedios.

Como veremos en las simulaciones, obtendremos el camino óptimo dentro del árbol, pero podríamos obtener uno mejor si suavizamos las curvas para que no viaje tan “poligonalmente”.

<u>Simulación 1</u>
Longitud de paso = 20, Inicio = [10,10], Meta = [480,490].

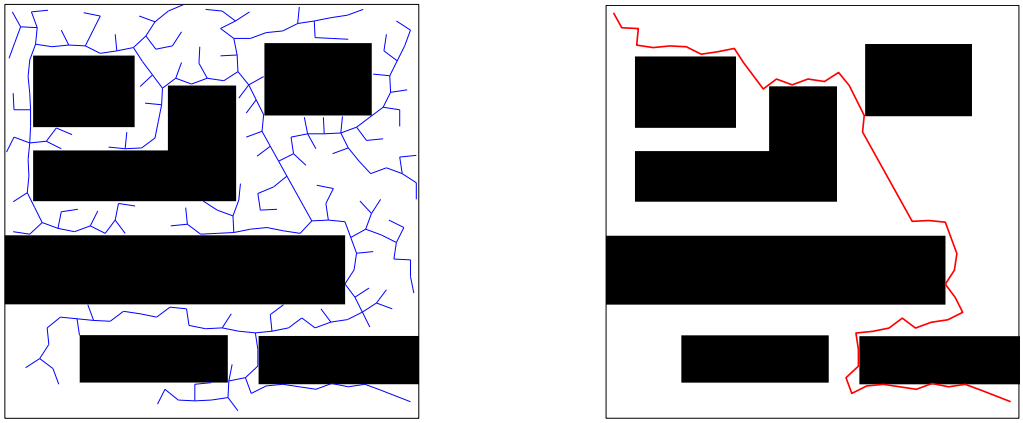


Figura 80: Simulación 1 RRT

Tiempo de procesado	0.443 segundos
Longitud de camino	1147.91 píxeles

Simulación 2

Longitud de paso = 20, Inicio = [10,10], Meta = [480,490].

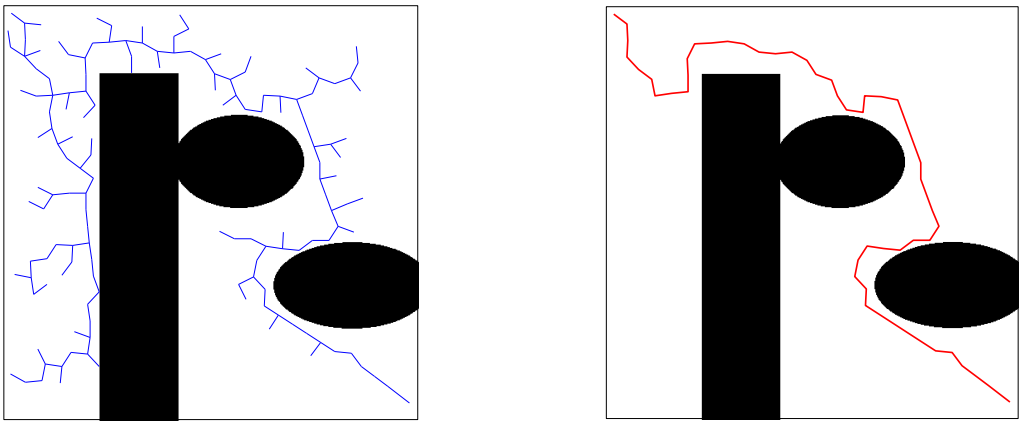


Figura 81: Simulación 2 RRT

Tiempo de procesado	0.464 segundos
Longitud de camino	1160.26 píxeles

9.7 FUNCIONES POTENCIALES

Como ya vimos en 7.2.5, el algoritmo de las funciones potenciales se basa en una planificación reactiva, es decir, reacciona el robot a medida que se acerca a los obstáculos y a la meta. Podemos decir que considera las distancias inmediatas a los obstáculos para calcular el movimiento inmediato, sin preocuparse mucho por el futuro, solo intentar acercarse cada vez más a la meta. Dichos obstáculos repelen al robot con una magnitud inversamente proporcional a la distancia. La meta atrae al robot con un potencial atrayente. El potencial resultante será la suma de ambos, y determinará el movimiento del robot.

En el código de [69], se tendrán en cuenta las distancias del robot a los obstáculos en 5 ángulos determinados para calcular el potencial repulsivo. Éstos son respecto al robot: hacia delante, a la izquierda, a la derecha, diagonal izquierda hacia delante y diagonal derecha hacia delante.

Podremos utilizar los mapas ya creados por el autor o generar nosotros unos nuevos. Podemos definir las posiciones de inicio y final y el ángulo inicial de orientación en radianes. También podemos modificar parámetros cinemáticos y dinámicos del robot, dibujado como un cuadrado, tales como tamaño, velocidad, distancia de seguridad, aceleración máxima, giro máximo… Además, se puede cambiar el factor de escala de los potenciales atractivo y repulsivo. Yo dejaré los parámetros tal y como están.

Veremos ahora algunas simulaciones hechas con Matlab R2014a sobre los mapas proporcionados por el autor del código, con dimensiones 500x500.

Simulación 1

Inicio = [50,50], Meta= [450,450], Dirección inicial del robot = pi/8.

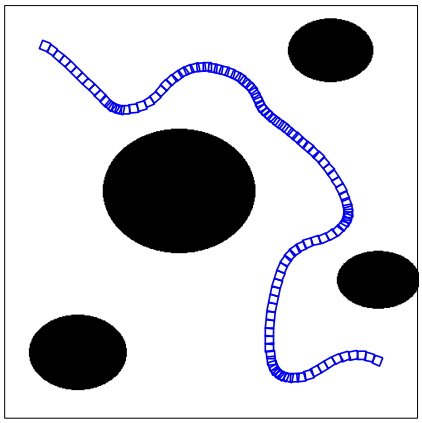


Figura 82: Simulación 1 Funciones Potenciales

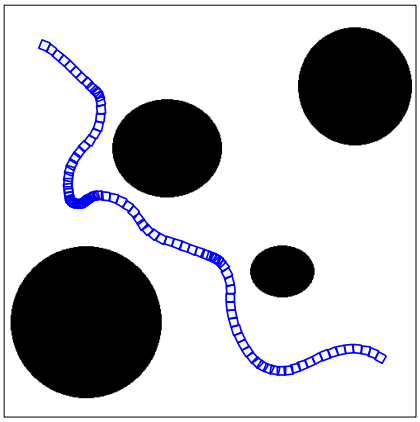
Tiempo de procesado	5.168 segundos
Longitud de camino	839.39 píxeles

Simulación 2

Inicio = [50,50], Meta= [450,450], Dirección inicial del robot = pi/8.

Figura 83: Simulación 2 Funciones Potenciales

Tiempo de procesado	4.797 segundos
Longitud de camino	784.79 píxeles



Simulación 3

Inicio = [50,50], Meta= [400,450], Dirección inicial del robot = pi/8.

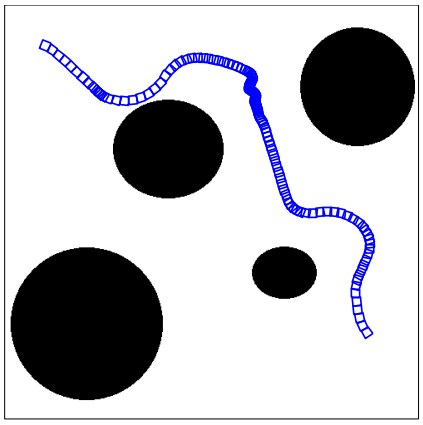


Figura 84: Simulación 3 Funciones Potenciales

Tiempo de procesado	8.059 segundos
Longitud de camino	742.71 píxeles

En esta simulación observamos que al pasar entre los dos obstáculos, tarda bastante tiempo debido a que los respectivos potenciales repulsivos hacen que tenga que moverse por la mediatriz, y para ello, tiene que adquirir la posición y orientación adecuada.

En las siguientes simulaciones podemos observar un caso más complicado. Tendremos que re-escalar el tamaño del robot para que pueda pasar entre los obstáculos más fácilmente. Además, reduciremos las velocidades y aceleraciones para que disponga de más tiempo de procesamiento conforme se mueve. También aumentaremos la maniobrabilidad del móvil permitiendo mayor facilidad de giro y aumentaremos el factor de repulsión del potencial para que se aleje más de los obstáculos. En el primer caso (Simulación 4) el robot caerá en un problema de mínimo local, moviéndose indefinidamente en los bucles de la parte inferior después de intentar ir por arriba.

Simulación 4

Inicio = [20,20], Meta= [490,490], Dirección inicial del robot = pi/8.

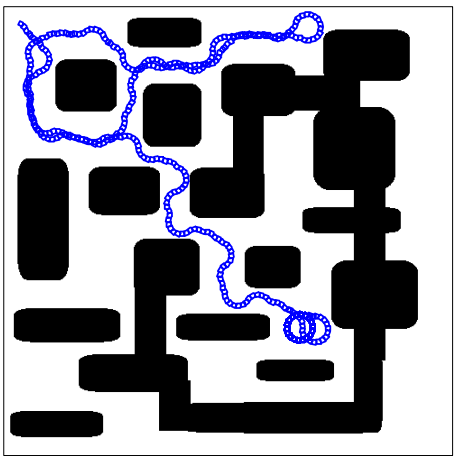


Figura 85: Simulación 4 Funciones Potenciales

En el segundo caso (Simulación 5) hemos conseguido obtener un camino aumentando el factor de repulsión de los obstáculos. Podemos destacar que no es una solución única, sino que cada vez que hagamos la simulación con los mismos parámetros, obtendremos soluciones distintas, o ninguna solución en algunas ocasiones. Por tanto, se puede ver la importancia de las restricciones cinemáticas y dinámicas del robot a la hora de diseñarlo.

Simulación 5

Inicio = [20,20], Meta= [490,490], Dirección inicial del robot = pi/8.

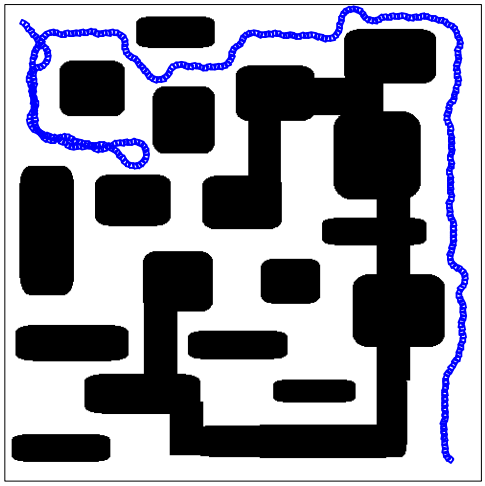


Figura 86: Simulación 5 Funciones Potenciales

Tiempo de procesado	21.082 segundos
Longitud de camino	1147.15 píxeles

10 PRESUPUESTO

Las simulaciones han requerido la utilización del siguiente software:

Producto	Cantidad	Precio unitario (€/unidad)	Precio total (€)
Matlab R2014a (Student Version)	1	69	69

El desarrollo del proyecto, desde la lectura de la bibliografía hasta finalizar de escribirlo, ha llevado en torno a unas 330 horas. Si establecemos un coste horario de unos 25€, incluyendo conceptos como la seguridad social, vacaciones e impuestos, el coste de la mano de obra asciende a:

330 horas por 25 €/hora…… …… 8250 €

El presupuesto total lo obtendremos agregando al anterior los siguientes conceptos:

- 6% de beneficio industrial
- 21% de impuesto sobre el valor añadido (IVA)

Descripción	Presupuesto (€)
Coste de los medios	69
Coste del diseño y montaje	8250
Coste de fabricación	8319
Beneficio Industrial (6%)	499.14
Total parcial	8818.14
IVA (21%)	1851.81
Coste total	10669.95

El coste total del proyecto asciende a:

DIEZ MIL SEISCIENTOS SESENTA Y NUEVE CON NOVENTA Y CINCO EUROS

11 CONCLUSIONES

La primera parte de este trabajo ha permitido dar una visión general de la robótica móvil, de creciente importancia para la sociedad actual. Gracias a esta visión, el lector podrá formarse una idea inicial de este campo, que podrá desarrollar y complementar con abundante bibliografía existente, además de la ya citada.

La segunda parte y el tema central ha sido, sin embargo, la planificación de rutas entre dos puntos. Dentro de todos los algoritmos de planificación de rutas vistos en este trabajo, podemos destacar algunos. El Voronoi resalta por su sencillez y la seguridad de los caminos generados, debido a que atraviesa por la mitad entre los obstáculos. El A* es muy eficiente, siendo uno de los más utilizados en mapas estáticos. Sin embargo, si el entorno es dinámico, recurriremos al D* o alguna de sus variantes debido a que han sido desarrollados para esos casos. También podrían utilizarse funciones potenciales ya que a medida que se mueve por el entorno, el robot se verá repelido por los obstáculos que se vaya encontrando. Si en cambio, queremos explorar rápidamente el mapa y no importa que el camino sea lo más óptimo posible, acudiremos a métodos probabilísticos como el PRM o el RRT.

En cada uno podemos encontrar puntos débiles y puntos fuertes para los intereses que tengamos y la aplicación deseada. Asimismo, las simulaciones realizadas con Matlab muestran la cantidad de código y formas diversas que existen para implementar un algoritmo cualquiera. Según cuál sea nuestro objetivo, tendremos que crear o hacer uso de códigos que sean los adecuados para cumplirlo.

Debido a que puede darse el caso de no conocer el mapa o la localización del robot con exactitud, tendremos que recurrir a métodos como el filtro de Kalman, Markov o el filtro de partículas. Éstos, basados en probabilidades, sensores propioceptivos y referencias externas, nos permitirán estimar dicha localización, imprescindible para llevar a cabo nuestros propósitos. Si en cambio, se da la situación de no conocer de primera mano el mapa, podremos crearlo y localizarnos en él simultáneamente mediante SLAM, tal y como se describe ligeramente en el apartado 8.4.

Personalmente, este trabajo ha sido bastante satisfactorio a pesar de que ha alcanzado una extensión superior a la esperada. Gracias a él, he podido introducirme un poco en el campo de la robótica móvil, el cual es muy interesante y puede ser muy

útil de cara a mi futuro profesional. Además, me ha permitido mejorar en cierto grado en otros aspectos como aprender muchos términos en inglés sobre estos campos, seleccionar información de una bibliografía abundante, redactar un documento extenso y utilizar Word de manera más óptima, entre otras.

Este trabajo consiste sólo en un punto de partida para multitud de proyectos y trabajos que pueden realizarse. El primer paso sería conceder mayor autonomía al robot siendo capaz de realizar SLAM para generar su propio mapa del entorno y localizarse a medida que se mueve por él. También se podría incorporar una cámara para utilizar visión artificial y sea capaz de reconocer objetos o personas. Gracias a este tipo de avances, podemos encontrar numerosas aplicaciones de interés industrial, tales como logística de almacenamiento, limpieza exhaustiva en fábricas, inspección, agricultura….

Por otro lado, se puede realizar la implementación en robots móviles, para dedicarlos a exploración. Un ejemplo son los Mars Rovers, enviados a explorar Marte. A pesar de que nos hemos centrado en robots con ruedas moviéndose en un plano, se podría extrapolar a tres dimensiones para utilizar algoritmos de planificación en cuadrópteros, vehículos submarinos, robots con patas, etc. Estos tipos de robots podrían ser más eficaces a la hora de explorar territorios desconocidos o incluso en océanos.

Además de lo anterior, podemos encontrarnos aplicaciones que se necesitan con cierta frecuencia, como por ejemplo limpieza de centrales nucleares, salvamento o búsqueda en desastres naturales, que podrían ser peligrosas para los seres humanos pero factibles para robots móviles bien diseñados con los algoritmos adecuados. Corresponde ahora al lector desarrollar su creatividad y seguir formándose para impulsar estos campos de gran importancia futura.

12 ANEXOS

12.1 VORONOI

Get_Voronoi.m

```
%function [ Temp_Edge ] = Voronoi_Graph( Voro_Vertex, Voro_Cell,
All_cells_Number,Cell_start )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here
clear all;
close all;
clc;

%specify file name here to load from
LOAD_FILE_NAME = 'Obstacle_config004';

load(strcat(strcat('./Obstacle_Files/',LOAD_FILE_NAME)));

%Code for drawing obstale configuration
for i=1:Num_Object
    for r=1:length(X1{i})
       a=r;
       if(r==length(X1{i}))
           b=1;
       else
           b=r+1;
       end
       x=[X1{i}(a,1) X1{i}(b,1)];
       y=[X1{i}(a,2) X1{i}(b,2)];
       plot(x,y,'Color', 'Black');
       hold on;
    end
end

%Code for taking Start and End point as input
Start = ginput(1);
plot(Start(1),Start(2),'--go','MarkerSize',10,'MarkerFaceColor','g');
drawnow;
Goal  = ginput(1);
plot(Goal(1),Goal(2),'--ro','MarkerSize',10,'MarkerFaceColor','r');
drawnow;

%Uncomment following to Draw voronoi diagram of point obstacles
voronoi(X_Total_points,Y_Total_points);
%Getting Parameters of Voronoi Diagram
[Voro_Vertex,Voro_Cell] = voronoin([X_Total_points' Y_Total_points']);

k=1;
temp=0;
for i=1:length(All_cells_Number)
    L=length(Voro_Cell{i});
  for j=1:L
      a=Voro_Cell{i}(j);
      if(j==L)
          b=Voro_Cell{i}(1);
      else
          b=Voro_Cell{i}(j+1);
      end
      for l=1:Num_Object
          if(temp==1)
              temp=0;
              break;
```

```
            end
            if (l==All_cells_Number(i));
                continue;
            end
            for m=Cell_start(l):Cell_start(l+1)-2

if((~isempty(find(Voro_Cell{m}==a)))&(~isempty(find(Voro_Cell{m}==b))))
                    Temp_Edge(k,:)=[a b];
                    k=k+1;
                    temp=1;
                    break;
                end
            end
        end
    end
end

Temp_Edge=unique(Temp_Edge,'rows');

%figure;
axis([0 100 0 100]);
hold on;

for i=1:length(Temp_Edge)
    Edge_X1(i)=Voro_Vertex(Temp_Edge(i,1),1);
    Edge_X2(i)=Voro_Vertex(Temp_Edge(i,2),1);
    Edge_Y1(i)=Voro_Vertex(Temp_Edge(i,1),2);
    Edge_Y2(i)=Voro_Vertex(Temp_Edge(i,2),2);
    plot([Edge_X1(i) Edge_X2(i)],[Edge_Y1(i)
Edge_Y2(i)],'color','g','LineWidth',2);
end
```

Final_Path.m

```
%Minimum Distance

Vertex = unique(Temp_Edge);
N = length(Vertex);
M = length(Temp_Edge);

for i=1:N
    Vertex_Cord(i,:)=Voro_Vertex(Vertex(i),:);
    Start_distance(i)=norm(Start-Vertex_Cord(i,:));
    Goal_distance(i)=norm(Goal-Vertex_Cord(i,:));
end


Voro_Graph = inf*ones(N);

%figure;
axis([0 100 0 100]);
hold on;

for i = 1:M
    a= find(Vertex==Temp_Edge(i,1));
    b= find(Vertex==Temp_Edge(i,2));
    Distance = norm(Vertex_Cord(a,:)-Vertex_Cord(b,:));
    Voro_Graph(a,b)=Distance;
    Voro_Graph(b,a)=Distance;
%     Voro_Graph(a,b)=1;
%     Voro_Graph(b,a)=1;

    x=[Vertex_Cord(a,1) Vertex_Cord(b,1)];
    y=[Vertex_Cord(a,2) Vertex_Cord(b,2)];
```

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

for i=1:N
    Start_distance(i)=norm(Start-Vertex_Cord(i,:));
    Goal_distance(i)=norm(Goal-Vertex_Cord(i,:));
end

[Dummy Index_Start]=min(Start_distance);
[Dummy Index_Goal]=min(Goal_distance);
path = dijkstra(Voro_Graph,Index_Start,Index_Goal);

for i=1:Num_Object
    for r=1:length(X1{i})
       a=r;
       if(r==length(X1{i}))
           b=1;
       else
           b=r+1;
       end
       x=[X1{i}(a,1) X1{i}(b,1)];
       y=[X1{i}(a,2) X1{i}(b,2)];
       plot(x,y);
       hold on;
    end
end
drawnow;

plot(Start(1),Start(2),'--go','MarkerSize',10,'MarkerFaceColor','g');
plot(Goal(1),Goal(2),'--ro','MarkerSize',10,'MarkerFaceColor','r');

 figure(1);
 axis([0 100 0 100]);
 hold on;

 for i=1:length(Temp_Edge)
    Edge_X1(i)=Voro_Vertex(Temp_Edge(i,1),1);
    Edge_X2(i)=Voro_Vertex(Temp_Edge(i,2),1);
    Edge_Y1(i)=Voro_Vertex(Temp_Edge(i,1),2);
    Edge_Y2(i)=Voro_Vertex(Temp_Edge(i,2),2);
    plot([Edge_X1(i) Edge_X2(i)],[Edge_Y1(i)
Edge_Y2(i)],'color','g','LineWidth',2);
end

 x=[Start(1) Vertex_Cord(path(1),1)];
 y=[Start(2) Vertex_Cord(path(1),2)];
 plot(x,y,'-','color','m','LineWidth',3);
 drawnow;

 for i=1:length(path)-1
 x=[Vertex_Cord(path(i),1) Vertex_Cord(path(i+1),1)];
 y=[Vertex_Cord(path(i),2) Vertex_Cord(path(i+1),2)];
 plot(x,y,'-','color','m','LineWidth',3);
 drawnow;
 hold on;
 end

 x=[Vertex_Cord(path(i),1) Goal(1)];
 y=[Vertex_Cord(path(i),2) Goal(2)];
 plot(x,y,'-','color','m','LineWidth',3);
 drawnow;
```

12.2 DIJKSTRA Y A* EN CUADRÍCULA

TestScript1.m

```
% TestScript for Assignment 1
%
%% Define a small map
map = false(50);

% Add an obstacle
map (1:30, 6:12) = true;
map (39:42, 1:35) = true;
map (25:38, 22:26) = true;
map (20:35, 38:42) = true;

start_coords = [6, 2];
dest_coords  = [49, 30];

%%
close all;
[route, numExpanded] = DijkstraGrid (map, start_coords, dest_coords);
% Uncomment following line to run Astar
%[route, numExpanded] = AStarGrid (map, start_coords, dest_coords);
```

DijkstraGrid.m

```
function [route,numExpanded] = DijkstraGrid (input_map, start_coords, dest_coords)
% Run Dijkstra's algorithm on a grid.
% Inputs :
%   input_map : a logical array where the freespace cells are false or 0 and
%   the obstacles are true or 1
%   start_coords and dest_coords : Coordinates of the start and end cell
%   respectively, the first entry is the row and the second the column.
% Output :
%    route : An array containing the linear indices of the cells along the
%    shortest route from start to dest or an empty array if there is no
%    route. This is a single dimensional vector
%    numExpanded: Remember to also return the total number of nodes
%    expanded during your search


% set up color map for display
% 1 - white - clear cell
% 2 - black - obstacle
% 3 - red = visited
% 4 - blue  - on list
% 5 - green - start
% 6 - yellow - destination

cmap = [1 1 1; ...
    0 0 0; ...
    1 0 0; ...
    0 0 1; ...
    0 1 0; ...
    1 1 0;
    0.4 0.4 0.4];

colormap(cmap);

% variable to control if the map is being visualized on every
% iteration
drawMapEveryTime = true;

[nrows, ncols] = size(input_map);

% map - a table that keeps track of the state of each grid cell
map = zeros(nrows,ncols);

map(~input_map) = 1;   % Mark free cells
```

```
map(input_map)  = 2;   % Mark obstacle cells

% Generate linear indices of start and dest nodes
start_node = sub2ind(size(map), start_coords(1), start_coords(2));
dest_node  = sub2ind(size(map), dest_coords(1),  dest_coords(2));

map(start_node) = 5;
map(dest_node)  = 6;

% Initialize distance array
distanceFromStart = Inf(nrows,ncols);

% For each grid cell this array holds the index of its parent
parent = zeros(nrows,ncols);

distanceFromStart(start_node) = 0;

% keep track of number of nodes expanded
numExpanded = 0;

% Main Loop
while true

    % Draw current map
    map(start_node) = 5;
    map(dest_node) = 6;

    % make drawMapEveryTime = true if you want to see how the
    % nodes are expanded on the grid.
    if (drawMapEveryTime)
        image(1.5, 1.5, map);
        grid on;
        axis image;
        drawnow;
    end

    % Find the node with the minimum distance
    [min_dist, current] = min(distanceFromStart(:));

    if ((current == dest_node) || isinf(min_dist))
        break;
    end;

    % Update map
    map(current) = 3;         % mark current node as visited
    distanceFromStart(current) = Inf; % remove this node from further consideration

    % Compute row, column coordinates of current node
    [i, j] = ind2sub(size(distanceFromStart), current);


    %Build neighbor node list
    neighborList=[];
    if (i+1>0 && i+1<=nrows && j>0 && j<=ncols)
        neighborList = [neighborList sub2ind(size(map),i+1,j)];
    end
    if (i-1>0 && i-1<=nrows && j>0 && j<=ncols)
        neighborList = [neighborList sub2ind(size(map),i-1,j)];
    end
    if (i>0 && i<=nrows && j+1>0 && j+1<=ncols)
        neighborList = [neighborList sub2ind(size(map),i,j+1)];
    end
    if (i>0 && i<=nrows && j-1>0 && j-1<=ncols)
        neighborList = [neighborList sub2ind(size(map),i,j-1)];
    end

    if (isempty(neighborList))
        break;
```

```
    end

    numExpanded=numExpanded+1;

    for ineigh=1:length(neighborList)
        if (map(neighborList(ineigh))==1)
            if (input_map(neighborList(ineigh)))
                map(neighborList(ineigh))=2;
            else
                map(neighborList(ineigh))=4;
                distanceFromStart(neighborList(ineigh))=min_dist+1;
                parent(neighborList(ineigh))=current;
            end
        elseif (map(neighborList(ineigh))==6)
            distanceFromStart(neighborList(ineigh))=min_dist+1;
            parent(neighborList(ineigh))=current;
        end
    end

end

%% Construct route from start to dest by following the parent links
if (isinf(distanceFromStart(dest_node)))
    route = [];
else
    route = [dest_node];

    while (parent(route(1)) ~= 0)
        route = [parent(route(1)), route];
    end

    % Snippet of code used to visualize the map and the path
    for k = 2:length(route) - 1
        map(route(k)) = 7;
        pause(0.1);
        image(1.5, 1.5, map);
        grid on;
        axis image;
    end
end
end
```

AStarGrid.m

```
function [route,numExpanded] = AStarGrid (input_map, start_coords, dest_coords)
% Run A* algorithm on a grid.
% Inputs :
%   input_map : a logical array where the freespace cells are false or 0 and
%   the obstacles are true or 1
%   start_coords and dest_coords : Coordinates of the start and end cell
%   respectively, the first entry is the row and the second the column.
% Output :
%    route : An array containing the linear indices of the cells along the
%    shortest route from start to dest or an empty array if there is no
%    route. This is a single dimensional vector
%    numExpanded: Remember to also return the total number of nodes
%    expanded during your search

% set up color map for display
% 1 - white - clear cell
% 2 - black - obstacle
% 3 - red = visited
% 4 - blue  - on list
% 5 - green - start
% 6 - yellow - destination
```

```
cmap = [1 1 1; ...
    0 0 0; ...
    1 0 0; ...
    0 0 1; ...
    0 1 0; ...
    1 1 0; ...
    0.4 0.4 0.4];

colormap(cmap);

% variable to control if the map is being visualized on every
% iteration
drawMapEveryTime = true;

[nrows, ncols] = size(input_map);

% map - a table that keeps track of the state of each grid cell
map = zeros(nrows,ncols);

map(~input_map) = 1;   % Mark free cells
map(input_map)  = 2;   % Mark obstacle cells

% Generate linear indices of start and dest nodes
start_node = sub2ind(size(map), start_coords(1), start_coords(2));
dest_node  = sub2ind(size(map), dest_coords(1),  dest_coords(2));

map(start_node) = 5;
map(dest_node)  = 6;

% meshgrid will `replicate grid vectors' nrows and ncols to produce
% a full grid
% type `help meshgrid' in the Matlab command prompt for more information
parent = zeros(nrows,ncols);

%
[X, Y] = meshgrid (1:ncols, 1:nrows);

xd = dest_coords(1);
yd = dest_coords(2);

% Evaluate Heuristic function, H, for each grid cell
% Manhattan distance
H = abs(X - xd) + abs(Y - yd);

% Initialize cost arrays
f = Inf(nrows,ncols);
g = Inf(nrows,ncols);

g(start_node) = 0;
f(start_node) = H(start_node);

% keep track of the number of nodes that are expanded
numExpanded = 0;

% Main Loop

while true

    % Draw current map
    map(start_node) = 5;
    map(dest_node) = 6;

    % make drawMapEveryTime = true if you want to see how the
    % nodes are expanded on the grid.
    if (drawMapEveryTime)
        image(1.5, 1.5, map);
        grid on;
```

```
        axis image;
        drawnow;
    end

    % Find the node with the minimum f value
    [min_f, current] = min(f(:));

    if ((current == dest_node) || isinf(min_f))
        break;
    end;

    % Update input_map
    map(current) = 3;
    f(current) = Inf; % remove this node from further consideration

    % Compute row, column coordinates of current node
    [i, j] = ind2sub(size(f), current);


    %Build neighbor node list
    neighborList=[];
    if (i+1>0 && i+1<=nrows && j>0 && j<=ncols)
        neighborList = [neighborList sub2ind(size(map),i+1,j)];
    end
    if (i-1>0 && i-1<=nrows && j>0 && j<=ncols)
        neighborList = [neighborList sub2ind(size(map),i-1,j)];
    end
    if (i>0 && i<=nrows && j+1>0 && j+1<=ncols)
        neighborList = [neighborList sub2ind(size(map),i,j+1)];
    end
    if (i>0 && i<=nrows && j-1>0 && j-1<=ncols)
        neighborList = [neighborList sub2ind(size(map),i,j-1)];
    end

    if (isempty(neighborList))
        break;
    end

    numExpanded=numExpanded+1;

    for ineigh=1:length(neighborList)
        if (map(neighborList(ineigh))==1)
            if (input_map(neighborList(ineigh)))
                map(neighborList(ineigh))=2;
            else
                if (g(neighborList(ineigh))>g(current)+1)
                    map(neighborList(ineigh))=4;
                    g(neighborList(ineigh))=g(current)+1;

f(neighborList(ineigh))=g(neighborList(ineigh))+H(neighborList(ineigh));
                    parent(neighborList(ineigh))=current;
                end
            end
        elseif (map(neighborList(ineigh))==6)
            g(neighborList(ineigh))=0;
            f(neighborList(ineigh))=0;
            parent(neighborList(ineigh))=current;
        end
    end

end

%% Construct route from start to dest by following the parent links
if (isinf(f(dest_node)))
    route = [];
else
    route = [dest_node];
```

```
    while (parent(route(1)) ~= 0)
        route = [parent(route(1)), route];
    end

    % Snippet of code used to visualize the map and the path
    for k = 2:length(route) - 1
        map(route(k)) = 7;
        pause(0.1);
        image(1.5, 1.5, map);
        grid on;
        axis image;
    end
end
end
```

12.3 A* [69]

astart.m

```
%  Rahul Kala, IIIT Allahabad, Creative Commons Attribution-ShareAlike 4.0
International License.
% The use of this code, its parts and all the materials in the text; creation of
derivatives and their publication; and sharing the code publically is permitted
without permission.
% Please cite the work in all materials as: R. Kala (2014) Code for Robot Path
Planning using A* algorithm, Indian Institute of Information Technology Allahabad,
Available at: http://rkala.in/codes.html

mapOriginal=im2bw(imread('map5.bmp')); % input map read from a bmp file. for new
maps write the file name here
resolutionX=80;
resolutionY=80;
source=[10 10]; % source position in Y, X format
goal=[490 490]; % goal position in Y, X format

% conn=[1 1 1; % robot (marked as 2) can move up, left, right, down and diagonally
(all 1s)
%       1 2 1;
%       1 1 1];

  conn=[1 1 1 1 1; % another option of conn
        1 1 1 1 1;
        1 1 2 1 1;
        1 1 1 1 1;
        1 1 1 1 1];

%   conn=[0 1 0; % another option of conn. % robot (marked as 2) can move up, left,
right and down (all 1s), but not diagonally (all 0). you can increase/decrease the
size of the matrix
%         1 2 1;
%         0 1 0];

display=false; % display processing of nodes

%%%%% parameters end here %%%%%

mapResized=imresize(mapOriginal,[resolutionX resolutionY]);
map=mapResized; % grow boundary by a unit pixel
for i=1:size(mapResized,1)
    for j=1:size(mapResized,2)
        if mapResized(i,j)==0
            if i-1>=1, map(i-1,j)=0; end
            if j-1>=1, map(i,j-1)=0; end
            if i+1<=size(map,1), map(i+1,j)=0; end
```

```
            if j+1<=size(map,2), map(i,j+1)=0; end
            if i-1>=1 && j-1>=1, map(i-1,j-1)=0; end
            if i-1>=1 && j+1<=size(map,2), map(i-1,j+1)=0; end
            if i+1<=size(map,1) && j-1>=1, map(i+1,j-1)=0; end
            if i+1<=size(map,1) && j+1<=size(map,2), map(i+1,j+1)=0; end
        end
    end
end
source=double(int32((source.*[resolutionX resolutionY])./size(mapOriginal)));
goal=double(int32((goal.*[resolutionX resolutionY])./size(mapOriginal)));

if ~feasiblePoint(source,map), error('source lies on an obstacle or outside map');
end
if ~feasiblePoint(goal,map), error('goal lies on an obstacle or outside map'); end
if length(find(conn==2))~=1, error('no robot specified in connection matrix'); end

%structure of a node is taken as positionY, positionX, historic cost, heuristic
cost, total cost, parent index in closed list (-1 for source)
Q=[source 0 heuristic(source,goal) 0+heuristic(source,goal) -1]; % the processing
queue of A* algorihtm, open list
closed=ones(size(map)); % the closed list taken as a hash map. 1=not visited,
0=visited
closedList=[]; % the closed list taken as a list
pathFound=false;
tic;
counter=0;
colormap(gray(256));
while size(Q,1)>0
     [A, I]=min(Q,[],1);
     n=Q(I(5),:); % smallest cost element to process
     Q=[Q(1:I(5)-1,:);Q(I(5)+1:end,:)]; % delete element under processing
     if n(1)==goal(1) && n(2)==goal(2) % goal test
         pathFound=true;break;
     end
     [rx,ry,rv]=find(conn==2); % robot position at the connection matrix
     [mx,my,mv]=find(conn==1); % array of possible moves
     for mxi=1:size(mx,1) %iterate through all moves
         newPos=[n(1)+mx(mxi)-rx n(2)+my(mxi)-ry]; % possible new node
         if checkPath(n(1:2),newPos,map) %if path from n to newPos is collission-
free
             if closed(newPos(1),newPos(2))~=0 % not already in closed
                 historicCost=n(3)+historic(n(1:2),newPos);
                 heuristicCost=heuristic(newPos,goal);
                 totalCost=historicCost+heuristicCost;
                 add=true; % not already in queue with better cost
                 if length(find((Q(:,1)==newPos(1)) .* (Q(:,2)==newPos(2))))>=1
                     I=find((Q(:,1)==newPos(1)) .* (Q(:,2)==newPos(2)));
                     if Q(I,5)<totalCost, add=false;
                     else Q=[Q(1:I-1,:);Q(I+1:end,:);];add=true;
                     end
                 end
                 if add
                     Q=[Q;newPos historicCost heuristicCost totalCost
size(closedList,1)+1]; % add new nodes in queue
                 end
             end
         end
     end
     closed(n(1),n(2))=0;closedList=[closedList;n]; % update closed lists
     if display
        image((map==0).*0 + ((closed==0).*(map==1)).*125 +
((closed==1).*(map==1)).*255);
        counter=counter+1;
        M(counter)=getframe;
     end
end
if ~pathFound
    error('no path found')
```

```
end
if display
    disp('click/press any key');
    waitforbuttonpress;
end

path=[n(1:2)]; %retrieve path from parent information
prev=n(6);
while prev>0
    path=[closedList(prev,1:2);path];
    prev=closedList(prev,6);
end
path=[(path(:,1)*size(mapOriginal,1))/resolutionX
(path(:,2)*size(mapOriginal,2))/resolutionY];
pathLength=0;
for i=1:length(path)-1, pathLength=pathLength+historic(path(i,:),path(i+1,:)); end
fprintf('processing time=%d \nPath Length=%d \n\n', toc,pathLength);
imshow(mapOriginal);
rectangle('position',[1 1 size(mapOriginal)-1],'edgecolor','k')
line(path(:,2),path(:,1),'color','red','linewidth',1.5);
```

checkPath.m

```
function feasible=checkPath(n,newPos,map)
feasible=true;
dir=atan2(newPos(1)-n(1),newPos(2)-n(2));
for r=0:0.5:sqrt(sum((n-newPos).^2))
    posCheck=n+r.*[sin(dir) cos(dir)];
    if ~(feasiblePoint(ceil(posCheck),map) && feasiblePoint(floor(posCheck),map) &&
            feasiblePoint([ceil(posCheck(1)) floor(posCheck(2))],map) &&
feasiblePoint([floor(posCheck(1)) ceil(posCheck(2))],map))
        feasible=false;break;
    end
    if ~feasiblePoint(newPos,map), feasible=false; end
end
```

feasiblePoint.m

```
function feasible=feasiblePoint(point,map)
feasible=true;
% check if collission-free spot and inside maps
if ~(point(1)>=1 && point(1)<=size(map,1) && point(2)>=1 && point(2)<=size(map,2)
&& map(point(1),point(2))==1)
    feasible=false;
end
```

heuristic.m

```
function h=heuristic(X,goal)
h = sqrt(sum((X-goal).^2));
```

historic.m

```
function h=historic(a,b)
h = sqrt(sum((a-b).^2));
```

12.4 PRM [69]

astart.m (tiene el mismo nombre que en A*, pero es distinta)

```
% Rahul Kala, IIIT Allahabad, Creative Commons Attribution-ShareAlike 4.0
International License.
% The use of this code, its parts and all the materials in the text; creation of
derivatives and their publication; and sharing the code publically is permitted
without permission.
% Please cite the work in all materials as: R. Kala (2014) Code for Robot Path
Planning using Probabilistic Roadmap, Indian Institute of Information Technology
Allahabad, Available at: http://rkala.in/codes.html

map=im2bw(imread('map3.bmp')); % input map read from a bmp file. for new maps write
the file name here
source=[10 10]; % source position in Y, X format
goal=[450 490]; % goal position in Y, X format
k=50; % number of points in the PRM
display=true; % display processing of nodes

%%%%% parameters end here %%%%%

if ~feasiblePoint(source,map), error('source lies on an obstacle or outside map');
end
if ~feasiblePoint(goal,map), error('goal lies on an obstacle or outside map'); end

imshow(map);
rectangle('position',[1 1 size(map)-1],'edgecolor','k')
vertex=[source;goal]; % source and goal taken as additional vertices in the path
planning to ease planning of the robot
if display, rectangle('Position',[vertex(1,2)-5,vertex(1,1)-
5,10,10],'Curvature',[1,1],'FaceColor','r'); end
if display, rectangle('Position',[vertex(2,2)-5,vertex(2,1)-
5,10,10],'Curvature',[1,1],'FaceColor','r'); end
tic;
while length(vertex)<k+2 % iteratively add vertices
    x=double(int32(rand(1,2) .* size(map)));
    if feasiblePoint(x,map),
        vertex=[vertex;x];
        if display, rectangle('Position',[x(2)-5,x(1)-
5,10,10],'Curvature',[1,1],'FaceColor','r'); end
    end
end
if display
    disp('click/press any key');
    waitforbuttonpress;
end
edges=cell(k+2,1); % edges to be stored as an adjacency list
for i=1:k+2
    for j=i+1:k+2
        if checkPath(vertex(i,:),vertex(j,:),map);
            edges{i}=[edges{i};j];edges{j}=[edges{j};i];
            if display, line([vertex(i,2);vertex(j,2)],[vertex(i,1);vertex(j,1)]);
end
        end
    end
end
if display
    disp('click/press any key');
    waitforbuttonpress;
end

%structure of a node is taken as index of node in vertex, historic cost, heuristic
cost, total cost, parent index in closed list (-1 for source)
Q=[1 0 heuristic(vertex(1,:),goal) 0+heuristic(vertex(1,:),goal) -1]; % the
processing queue of A* algorihtm, open list
```

```
closed=[]; % the closed list taken as a list
pathFound=false;
while size(Q,1)>0
     [A, I]=min(Q,[],1);
     n=Q(I(4),:); % smallest cost element to process
     Q=[Q(1:I(4)-1,:);Q(I(4)+1:end,:)]; % delete element under processing
     if n(1)==2 % goal test
         pathFound=true;break;
     end
     for mv=1:length(edges{n(1),1}) %iterate through all edges from the node
         newVertex=edges{n(1),1}(mv);
         if length(closed)==0 || length(find(closed(:,1)==newVertex))==0 % not
already in closed
             historicCost=n(2)+historic(vertex(n(1),:),vertex(newVertex,:));
             heuristicCost=heuristic(vertex(newVertex,:),goal);
             totalCost=historicCost+heuristicCost;
             add=true; % not already in queue with better cost
             if length(find(Q(:,1)==newVertex))>=1
                 I=find(Q(:,1)==newVertex);
                 if Q(I,4)<totalCost, add=false;
                 else Q=[Q(1:I-1,:);Q(I+1:end,:);];add=true;
                 end
             end
             if add
                 Q=[Q;newVertex historicCost heuristicCost totalCost
size(closed,1)+1]; % add new nodes in queue
             end
         end
     end
     closed=[closed;n]; % update closed lists
end
if ~pathFound
    error('no path found')
end

fprintf('processing time=%d \nPath Length=%d \n\n', toc,n(4));
path=[vertex(n(1),:)]; %retrieve path from parent information
prev=n(5);
while prev>0
    path=[vertex(closed(prev,1),:);path];
    prev=closed(prev,5);
end

imshow(map);
rectangle('position',[1 1 size(map)-1],'edgecolor','k')
line(path(:,2),path(:,1),'color','r','linewidth',1.5);
```

Las funciones checkPath.m, feasiblePoint.m y heuristic.m son las mismas que con A*.

12.5 RRT [69]

astart.m (tiene el mismo nombre que en A*, pero es distinta)

```
%  Rahul Kala, IIIT Allahabad, Creative Commons Attribution-ShareAlike 4.0
International License.
% The use of this code, its parts and all the materials in the text; creation of
derivatives and their publication; and sharing the code publically is permitted
without permission.
```

```
% Please cite the work in all materials as: R. Kala (2014) Code for Robot Path
Planning using Rapidly-exploring Random Trees, Indian Institute of Information
Technology Allahabad, Available at: http://rkala.in/codes.html

map=im2bw(imread('map3.bmp')); % input map read from a bmp file. for new maps write
the file name here
source=[10 10]; % source position in Y, X format
goal=[480 490]; % goal position in Y, X format
stepsize=20; % size of each step of the RRT
disTh=20; % nodes closer than this threshold are taken as almost the same
maxFailedAttempts = 10000;
display=true; % display of RRT

%%%%% parameters end here %%%%%

tic;
if ~feasiblePoint(source,map), error('source lies on an obstacle or outside map');
end
if ~feasiblePoint(goal,map), error('goal lies on an obstacle or outside map'); end
if display, imshow(map);rectangle('position',[1 1 size(map)-1],'edgecolor','k');
end
RRTree=double([source -1]); % RRT rooted at the source, representation node and
parent index
failedAttempts=0;
counter=0;
pathFound=false;
while failedAttempts<=maxFailedAttempts  % loop to grow RRTs
    if rand < 0.5,
        sample=rand(1,2) .* size(map); % random sample
    else
        sample=goal; % sample taken as goal to bias tree generation to goal
    end
    [A, I]=min( distanceCost(RRTree(:,1:2),sample) ,[],1); % find closest as per
the function in the metric node to the sample
    closestNode = RRTree(I(1),1:2);
    theta=atan2(sample(1)-closestNode(1),sample(2)-closestNode(2));  % direction to
extend sample to produce new node
    newPoint = double(int32(closestNode(1:2) + stepsize * [sin(theta)
cos(theta)]));
    if ~checkPath(closestNode(1:2), newPoint, map) % if extension of closest node
in tree to the new point is feasible
        failedAttempts=failedAttempts+1;
        continue;
    end
    if distanceCost(newPoint,goal)<disTh, pathFound=true;break; end % goal reached
    [A, I2]=min( distanceCost(RRTree(:,1:2),newPoint) ,[],1); % check if new node
is not already pre-existing in the tree
    if distanceCost(newPoint,RRTree(I2(1),1:2))<disTh,
failedAttempts=failedAttempts+1;continue; end
    RRTree=[RRTree;newPoint I(1)]; % add node
    failedAttempts=0;
    if display,
        line([closestNode(2);newPoint(2)],[closestNode(1);newPoint(1)]);
        counter=counter+1;M(counter)=getframe;
    end
end
if display && pathFound
    line([closestNode(2);goal(2)],[closestNode(1);goal(1)]);
    counter=counter+1;M(counter)=getframe;
end
if display
    disp('click/press any key');
    waitforbuttonpress;
end
if ~pathFound, error('no path found. maximum attempts reached'); end
path=[goal];
prev=I(1);
while prev>0
```

```
    path=[RRTree(prev,1:2);path];
    prev=RRTree(prev,3);
end
pathLength=0;
for i=1:length(path)-1,
pathLength=pathLength+distanceCost(path(i,1:2),path(i+1,1:2)); end
fprintf('processing time=%d \nPath Length=%d \n\n', toc,pathLength);
imshow(map);rectangle('position',[1 1 size(map)-1],'edgecolor','k');
line(path(:,2),path(:,1),'color','r','linewidth',1.5);
```

distanceCost.m

```
function h=distanceCost(a,b)
h = sqrt((a(:,1)-b(:,1)).^2 + (a(:,2)-b(:,2)).^2 );
```

Las funciones checkPath.m y feasiblePoint.m son las mismas que con A*.

12.6 CAMPOS POTENCIALES [69]

astart.m (tiene el mismo nombre que en A*, pero es distinta)

```
%  Rahul Kala, IIIT Allahabad, Creative Commons Attribution-ShareAlike 4.0
International License.
% The use of this code, its parts and all the materials in the text; creation of
derivatives and their publication; and sharing the code publically is permitted
without permission.
% Please cite the work in all materials as: R. Kala (2014) Code for Robot Path
Planning using Artificial Potential Fields, Indian Institute of Information
Technology Allahabad, Available at: http://rkala.in/codes.html

map=int16(im2bw(imread('map7.bmp'))); % input map read from a bmp file. for new
maps write the file name here
source=[20 20]; % source position in Y, X format
goal=[490 490]; % goal position in Y, X format
robotDirection=pi/8; % initial heading direction
robotSize=[5 5]; %length and breadth
robotSpeed=4; % arbitrary units
maxRobotSpeed=4; % arbitrary units
S=15; % safety distance
distanceThreshold=30; % a threshold distace. points within this threshold can be
taken as same.
maxAcceleration=5; % maximum speed change per unit time
maxTurn=18*pi/180; % potential outputs to turn are restriect to -60 and 60 degrees.
k=3; % degree of calculating potential
attractivePotentialScaling=300000; % scaling factor for attractive potential
repulsivePotentialScaling=300000; % scaling factor for repulsive potential
minAttractivePotential=0.5; % minimum attractive potential at any point

%%%%% parameters end here %%%%%

currentPosition=source; % position of the centre of the robot
currentDirection=robotDirection; % direction of orientation of the robot
robotHalfDiagonalDistance=((robotSize(1)/2)^2+(robotSize(2)/2)^2)^0.5; % used for
distance calculations
pathFound=false; % has goal been reached
pathCost=0;
t=1;
```

```
imshow(map==1);
rectangle('position',[1 1 size(map)-1],'edgecolor','k')
pathLength=0;
if ~plotRobot(currentPosition,currentDirection,map,robotHalfDiagonalDistance)
     error('source lies on an obstacle or outside map');
end
M(t)=getframe;
t=t+1;

if ~feasiblePoint(goal,map), error('goal lies on an obstacle or outside map'); end

tic;
while ~pathFound

    % calculate distance from obstacle at front
    i=robotSize(1)/2+1;
    while true
        x=int16(currentPosition+i*[sin(currentDirection) cos(currentDirection)]);
        if ~feasiblePoint(x,map), break; end
        i=i+1;
    end
    distanceFront=i-robotSize(1)/2; % robotSize(1)/2 distance included in i was
inside the robot body

    % calculate distance from obstacle at left
    i=robotSize(2)/2+1;
    while true
        x=int16(currentPosition+i*[sin(currentDirection-pi/2) cos(currentDirection-
pi/2)]);
        if ~feasiblePoint(x,map), break; end
        i=i+1;
    end
    distanceLeft=i-robotSize(2)/2;

    % calculate distance from obstacle at right
    i=robotSize(2)/2+1;
    while true
        x=int16(currentPosition+i*[sin(currentDirection+pi/2)
cos(currentDirection+pi/2)]);
        if ~feasiblePoint(x,map), break; end
        i=i+1;
    end
    distanceRight=i-robotSize(2)/2;

    % calculate distance from obstacle at front-left diagonal
    i=robotHalfDiagonalDistance+1;
    while true
        x=int16(currentPosition+i*[sin(currentDirection-pi/4) cos(currentDirection-
pi/4)]);
        if ~feasiblePoint(x,map), break; end
        i=i+1;
    end
    distanceFrontLeftDiagonal=i-robotHalfDiagonalDistance;

    % calculate distance from obstacle at front-right diagonal
    i=robotHalfDiagonalDistance+1;
    while true
        x=int16(currentPosition+i*[sin(currentDirection+pi/4)
cos(currentDirection+pi/4)]);
        if ~feasiblePoint(x,map), break; end
        i=i+1;
    end
    distanceFrontRightDiagonal=i-robotHalfDiagonalDistance;

    % calculate angle from goal
     angleGoal=atan2(goal(1)-currentPosition(1),goal(2)-currentPosition(2));

     % calculate diatnce from goal
```

```
     distanceGoal=( sqrt(sum((currentPosition-goal).^2)) );
     if distanceGoal<distanceThreshold, pathFound=true; end

     % compute potentials
     repulsivePotential=(1.0/distanceFront)^k*[sin(currentDirection)
cos(currentDirection)] + ...
     (1.0/distanceLeft)^k*[sin(currentDirection-pi/2) cos(currentDirection-pi/2)] +
...
     (1.0/distanceRight)^k*[sin(currentDirection+pi/2) cos(currentDirection+pi/2)]
+ ...
     (1.0/distanceFrontLeftDiagonal)^k*[sin(currentDirection-pi/4)
cos(currentDirection-pi/4)] + ...
     (1.0/distanceFrontRightDiagonal)^k*[sin(currentDirection+pi/4)
cos(currentDirection+pi/4)];

     attractivePotential=max([(1.0/distanceGoal)^k*attractivePotentialScaling
minAttractivePotential])*[sin(angleGoal) cos(angleGoal)];
     totalPotential=attractivePotential-
repulsivePotentialScaling*repulsivePotential;

     % perform steer

preferredSteer=atan2(robotSpeed*sin(currentDirection)+totalPotential(1),robotSpeed*
cos(currentDirection)+totalPotential(2))-currentDirection;
     while preferredSteer>pi, preferredSteer=preferredSteer-2*pi; end % check to
get the angle between -pi and pi
     while preferredSteer<-pi, preferredSteer=preferredSteer+2*pi; end % check to
get the angle between -pi and pi
     preferredSteer=min([maxTurn preferredSteer]);
     preferredSteer=max([-maxTurn preferredSteer]);
     currentDirection=currentDirection+preferredSteer;

     % setting the speed based on vehicle acceleration and speed limits. the
vehicle cannot move backwards.
     preferredSpeed=sqrt(sum((robotSpeed*[sin(currentDirection)
cos(currentDirection)] + totalPotential).^2));
     preferredSpeed=min([robotSpeed+maxAcceleration preferredSpeed]);
     robotSpeed=max([robotSpeed-maxAcceleration preferredSpeed]);
     robotSpeed=min([robotSpeed maxRobotSpeed]);
     robotSpeed=max([robotSpeed 0]);

     if robotSpeed==0, error('robot had to stop to avoid collission'); end

     % calculating new position based on steer and speed
     newPosition=currentPosition+robotSpeed*[sin(currentDirection)
cos(currentDirection)];
     pathCost=pathCost+distanceCost(newPosition,currentPosition);
     currentPosition=newPosition;
     if ~feasiblePoint(int16(currentPosition),map), error('collission recorded');
end

     % plotting robot
     if ~plotRobot(currentPosition,currentDirection,map,robotHalfDiagonalDistance)
        error('collission recorded');
     end
     M(t)=getframe;t=t+1;
end
fprintf('processing time=%d \nPath Length=%d \n\n', toc,pathCost);
```

distanceCost.m (distinta que en RRT)

```
function h=distanceCost(a,b)
h = sqrt(sum(a-b).^2);
```

plotRobot.m

```
function feasible=plotRobot(position,direction,map,robotHalfDiagonalDistance)
% plot robot with specified configuration and check its feasibility
 corner1=position+robotHalfDiagonalDistance*[sin(direction-pi/4) cos(direction-
pi/4)];
 corner2=position+robotHalfDiagonalDistance*[sin(direction+pi/4)
cos(direction+pi/4)];
 corner3=position+robotHalfDiagonalDistance*[sin(direction-pi/4+pi) cos(direction-
pi/4+pi)];
 corner4=position+robotHalfDiagonalDistance*[sin(direction+pi/4+pi)
cos(direction+pi/4+pi)];

line([corner1(2);corner2(2);corner3(2);corner4(2);corner1(2)],[corner1(1);corner2(1
);corner3(1);corner4(1);corner1(1)],'color','blue','LineWidth',2);
 if ~feasiblePoint(int16(corner1),map) || ~feasiblePoint(int16(corner2),map) ||
~feasiblePoint(int16(corner3),map) || ~feasiblePoint(int16(corner4),map)
     feasible=false;
 else
     feasible=true;
 end
```

La función feasiblePoint.m es la misma que con A*.

13 REFERENCIAS

[1] «Aadeca (Asocicación Argentina de Control Automático),» [En línea]. Available: http://www.aadeca.org/pdf/cp_monografias/monografia_robot_movil.pdf. [Último acceso: 22 Marzo 2016].

[2] R. Siegwart, I. R. Noubakhsh y D. Scaramuzza, Introduction to Autonomous Mobile Robots, Mit Press Ltd, 2011.

[3] A. S. Terán, «Control de Robots Móviles Autoequilibrados,» TFG, Universidad de Cantabria, 2014.

[4] T. Lauwers, G. Kantor y R. L. Hollis, «Microdynamic Systems Laboratory,» [En línea]. Available: http://www.msl.ri.cmu.edu/publications/pdfs/ballbot_ICRA06_web.pdf. [Último acceso: 12 Marzo 2016].

[5] C. V. Venegas, «Universidad de Sevilla,» [En línea]. Available: http://www.esi2.us.es/~vivas/ayr2iaei/LOC_MOV.pdf. [Último acceso: Marzo 2016].

[6] A. O. Baturone, Robótica: Manipuladores y Robots Móviles, Marcombo, 2005.

[7] R. G. Sánchez, «Universidad de Málaga,» [En línea]. Available: http://www.ual.es/personal/rgonzalez/papers/Robotica_movil.pdf. [Último acceso: 11 Marzo 2016].

[8] B. Siciliano, L. Sciavicco, L. Villani y G. Oriolo, Robotics. Modelling, Planning and Control, Springer, 2009.

[9] Technology Robotix Society, «Robotix,» 2015. [En línea]. Available: https://www.robotix.in/tutorial/mechanical/wheelstut/. [Último acceso: 12 Marzo 2016].

[10] J. G. Gómez, «Principales Líneas de Investigación en Robots Reptores tipo Ápodos,» Septiembre 2002. [En línea]. Available: http://www.iearobotics.com/personal/juan/doctorado/Robots_apodos/estudio_apodos.pdf. [Último acceso: 12 Marzo 2016].

[11] R. D'Andrea. [En línea]. Available: http://raffaello.name/projects/flying-machine-arena/. [Último acceso: 21 Marzo 2016].

[12] P. Corke, Robotics, Vision and Control, Springer, 2011.

[13] D. Brecianini, M. Hehn y R. D'Andrea, «Flying Machine Arena,» [En línea]. Available: http://flyingmachinearena.org/wp-content/publications/2013/breIEEE13.pdf. [Último acceso: 21 Marzo 2016].

[14] O. Yildiz, A. E. Yilmaz y B. Gokalp, «State-of-the-Art System Solutions for for Unmanned Underwater Vehicles,» *Radioengineering Journal,* vol. 18, nº 4, pp. 590-600, Diciembre 2009.

[15] Amazon, «Amazon,» [En línea]. Available: http://www.amazon.com/b?node=8037720011. [Último acceso: 22 Marzo 2016].

[16] Domino's Pizza, «Domino's Pizza Enterprises Ltd 2015,» 2015. [En línea]. Available: https://www.dominos.com.au/inside-dominos/technology/dru. [Último acceso: 22 Marzo 2016].

[17] Amazon Robotics, «Amazon Robotics LLC,» 2015. [En línea]. Available: https://www.amazonrobotics.com/#/vision. [Último acceso: 22 Marzo 2016].

[18] Robotnik, [En línea]. Available: http://www.robotnik.eu/services-robotic/mobile-robotics-applications/. [Último acceso: 22 Marzo 2016].

[19] «IEEE Spectrum,» [En línea]. Available: http://spectrum.ieee.org/automaton/robotics/industrial-robots/no-tree-is-safe-from-this-chainsaw-wielding-robot. [Último acceso: 23 Marzo 2016].

[20] Nasa , «Mars Exploration Rovers,» [En línea]. Available: http://mars.nasa.gov/mer/technology/is_autonomous_mobility.html. [Último acceso: 22 Marzo 2016].

[21] F. Fahimi, Autonomous Robots. Modeling, Path Planning, and Control, Springer, 2009.

[22] J. R. Llata, *G-731. Robótica Industrial y Visión Artificial. Cinemática Directa,* Universidad de Cantabria, 2016.

[23] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki y S. Thrun, Principles of Robot Motion. Theory, Algorithms and Implementation, Cambridge (Massachusetts): MIT Press, 2005.

[24] X.-J. Jing, Mobile Robots Motion Planning. New Challenges, Vienna: In-Teh, 2008.

[25] L. I. G. Calandín y D. J. T. i. Montserrat, «Modelado Cinemático y Control de Robots Móviles con Ruedas,» Tesis Doctoral, Universidad Politécnica de Valencia.

[26] J. M. P. Oria y L. A. Rentería, *G-732 Control Avanzado,* Universidad de Cantabria, 2016.

[27] F. Cuesta y A. Ollero, Intelligent Mobile Robot Navigation, Springer, 2005.

[28] H. R. Everett, Sensors for Mobile Robots. Theory and Application, Wellesley, Massachusetts: A K Peters, 1995.

[29] J. Borenstein, H. R. Everett, L. Feng, S. W. Lee y R. H. Byrne, Where am I? Sensors and Methods for Mobile Robot Positioning, Michigan: The University of Michigan, 1996.

[30] J. Borenstein, H. R. Everett, L. Feng y D. Wehe, «Mobile Robot Positioning - Sensors and Techniques,» *Journal of Robotic Systems,* vol. 14, nº 4, pp. 231-249, 1997.

[31] M. Berg, Navigation with Simultaneous Localization and Mapping. For Indoor Mobile Robot., NTNU-Trondheim. Norwegian University of Science and Technology: TFM, Junio 2013.

[32] E. Masehian y A. Naseri, «Mobile Robot Online Motion Planning Using Generalized Voronoi Graphs,» *Journal of Industrial Engineering,* vol. 5, pp. 1-15, 2010.

[33] R. Goodrich, «Live Science,» 1 Octubre 2013. [En línea]. Available: http://www.livescience.com/40102-accelerometers.html. [Último acceso: 17 Mayo 2016].

[34] A. M. Fitzgerald, «AMFITZGERALD,» 14 Noviembre 2013. [En línea]. Available: http://www.amfitzgerald.com/papers/131114_AMFitzgerald_MEMS_Inertial_Sensors.pdf. [Último acceso: 18 Mayo 2016].

[35] R. Antonello y R. Oboe, «InTech Open,» [En línea]. Available: http://cdn.intechweb.org/pdfs/14952.pdf. [Último acceso: 18 Mayo 2016].

[36] D. Xia, C. Yu y L. Kong, «The Development of Micromachined Gyroscope Structure and Circuitry Technology,» *Sensors (ISSN 1424-8220). Open Access Journal,* nº 14, pp. 1394-1473, 2014.

[37] D. Ren, L. Wu, M. Yan, M. Cui, Z. You y M. Hu, «Design and Analyses of a MEMS Based Resonant Magnetometer,» *Sensors (ISSN 1424-8220). Open Access Journal,* nº 9, pp. 6951-6966, 2009.

[38] W. Burgard, C. Stachniss, M. Bennewitz y K. Arras, «Uni Freiburg,» [En línea]. Available: http://ais.informatik.uni-freiburg.de/teaching/ss11/robotics/slides/18-robot-motion-planning.pdf. [Último acceso: 25 Abril 2016].

[39] M. v. Kreveld, «Utrecht University,» [En línea]. Available: http://www.cs.uu.nl/docs/vakken/ga/slides7.pdf. [Último acceso: 2 Abril 2016].

[40] M. L. Wager. [En línea]. Available: http://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/integrated4/micha_voronoi.pdf. [Último acceso: 2 Abril 2016].

[41] P. B. Solanki, G. H. V. Reddy y A. Mukerjee, «IIT Kanpur,» [En línea]. Available: http://cse.iitk.ac.in/users/cs365/2013/submissions/~prabhanu/cs365/project/report.pdf. [Último acceso: 5 Abril 2016].

[42] S. Garrido, L. Moreno, D. Blanco y P. Jurewicz, «Path Planning for Mobile Robot Navigation using Voronoi Diagram and Fast Marching,» *International Journal of Robotics and Automation (IJRA),* vol. 2, nº 1, pp. 42-64, 2011.

[43] J. O'Rourke, Computational Geometry in C, Cambridge University Press.

[44] Y.-J. Ho y J.-S. Liu, «Institute of Information Science,» [En línea]. Available: http://www.iis.sinica.edu.tw/papers/liu/8846-F.pdf. [Último acceso: 8 Junio 2016].

[45] S. M. LaValle, Planning Algorithms, Cambridge University Press, 2006.

[46] T. Yüksel y A. Sezgin. [En línea]. Available: http://www.emo.org.tr/ekler/c90885b28e58d1f_ek.pdf. [Último acceso: 25 Abril 2016].

[47] A. Doucette y W. Lu. [En línea]. Available: http://worldcomp-proceedings.com/proc/p2015/ICA3006.pdf. [Último acceso: 25 Abril 2016].

[48] S. Thrun y P. Norvig, «Intro to Artificial Intelligence,» Udacity, [En línea]. Available: https://www.udacity.com/course/intro-to-artificial-intelligence--cs271. [Último acceso: 25 Abril 2016].

[49] S. K. Ghosh, «Tata Institute of Fundamental Research,» [En línea]. Available: http://www.tcs.tifr.res.in/~ghosh/robot-motion.pdf. [Último acceso: 26 Abril 2016].

[50] C. Taylor, «Robotics: Computational Motion Planning,» Coursera, [En línea]. Available: https://es.coursera.org/learn/robotics-motion-planning/. [Último acceso: 27 Abril 2016].

[51] M. B. Subramanian, D. K. Sudhagar y G. RajaRajeswari, «Intelligent Path Planning Of Mobile Robot Agent By Using Breadth First Search Algorithm,» *International Journal of Innovative Research in Science, Engineering and Technology,* vol. 3, nº Special Issue 3, pp. 1951-1955, 2014.

[52] «School of Mathematics and Statistics. University of Melbourne,» [En línea]. Available: http://www.ms.unimelb.edu.au/~moshe@unimelb/620-261/dijkstra/dijkstra.html. [Último acceso: 29 Abril 2016].

[53] D. Ferguson, M. Likhachev y A. Stentz, «Carnegie Mellon University,» [En línea]. Available: http://www.cs.cmu.edu/~maxim/files/hsplanguide_icaps05ws.pdf. [Último acceso: 30 Abril 2016].

[54] A. Andriën, «TU/e Eindhoven University of Technology,» 12 Diciembre 2012. [En línea]. Available: https://www.tue.nl/uploads/media/2012_MN-420707_ARP_Andrien.pdf. [Último acceso: 30 Abril 2016].

[55] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz y S. Thrun, «Carnegie Mellon University,» 23 Octubre 2007. [En línea]. Available: http://www.cs.cmu.edu/~maxim/files/ad_aij08_preprint.pdf. [Último acceso: 30 Abril 2016].

[56] N. Sariff y N. Buniyamin, «An Overview of Autonomous Mobile Robot Path Planning Algorithms,» de *4th Student Conference on Research and Development*, Selangor, 2006.

[57] H. Choset, «Carnegie Mellon University,» [En línea]. Available: http://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf. [Último acceso: 30 Abril 2016].

[58] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz y S. Thrun, «Carnegie Mellon University,» 2005. [En línea]. Available: http://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf. [Último acceso: 7 Mayo 2016].

[59] D. Ferguson y A. Stentz, «Carniege Mellon. The Robotics Institute,» [En línea]. Available: https://www-preview.ri.cmu.edu/pub_files/pub4/ferguson_david_2006_3/ferguson_david_2006_3.pdf. [Último acceso: 06 Mayo 2016].

[60] N. James y T. Bräunl, «University of Western Australia. Robotics & Automation Lab,» [En línea]. Available: http://robotics.ee.uwa.edu.au/theses/2005-Navigation-Ng.pdf. [Último acceso: 6 Mayo 2016].

[61] C. Saranya, K. K. Rao, M. Unnikrishnan, D. V. Brinda, D. V. Lalithambika y M. Dhekane, «Implementation of D* Path Planning Algorithm with NXT LEGO Mindstorms Kit for Navigation through Unknown Terrains,» de *International Conference on Control, Communication and Power Engineering*, Bangalore, 2013.

[62] I. A. Sucan, M. Moll y L. E. Kavraki, «OMPL. The Open Motion Planning Library,» 23 Noviembre 2015. [En línea]. Available: http://ompl.kavrakilab.org/OMPL_Primer.pdf. [Último acceso: 27 Mayo 2016].

[63] P. M. Newman, «University of Oxford. Information Engineering,» Octubre 2003. [En línea]. Available: http://www.robots.ox.ac.uk/~pnewman/Teaching/C4CourseResources/C4BMobileRobots.pdf. [Último acceso: 3 Junio 2016].

[64] T. Liddy, T.-F. Lu, P. Lozo y D. Harvey, «ARAA. Australian Robotics & Automation Asociation,» [En línea]. Available: http://www.araa.asn.au/acra/acra2008/papers/pap122s1.pdf. [Último acceso: 2 Junio 2016].

[65] B. Kuipers, «The University of Texas at Austin. Computer Science,» [En línea]. Available: http://www.cs.utexas.edu/~pstone/Courses/395Tfall05/resources/. [Último acceso: 2 Junio 2016].

[66] R. Lonsdale, «Demonstrating Simultaneous Localization and Mapping Using LEGO Mindstorms,» University of Leeds. School of Computing Studies, Leeds, 2011-2012.

[67] S. Brown, «Using SLAM with LEGO Mindstorms to Explore and Map an Environment,» University of Leeds. School of Computing Studies, Leeds, 2013-2014.

[68] H. Durrant-Whyte y T. Bailey, «Simultaneous Localisation and Mapping (SLAM): Part I The Essential Algorithms,» *IEEE Robotics & Automation Magazine,* vol. 13, nº 2, pp. 99-110, 2006.

[69] R. Kala, «Rahul Kala, Indian Institute of Information Technology, Allahabad,» [En línea]. Available: http://rkala.in/codes.php. [Último acceso: 16 Junio 2016].