Universitat de les Illes Balears

escola politècnica superior

Trabajo Fin de Grado

INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Estudio e implementación del algoritmo de navegación reactiva Tangent Bug en el entorno ROS

PASQUAL JOAN RIBOT LACOSTA

Tutores

Javier Antich Tobaruela Alberto Ortiz Rodríguez

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, 3 de julio de 2018

Agradezco a mis tutores Javier Antich y Alberto Ortiz la oportunidad que me han dado de trabajar en este proyecto. Dar las gracias a la Universitat de les Illes Baleares y a todos los profesores del Grado en Electrónica Industrial y Autómatica, por su dedicación y empeño en mi educación. Por último, agradecer a mi familia y a Alicia todo el apoyo y ánimos que me han dado estos años.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Resumen

Los robots terrestres ya forman parte de muchos elementos cotidianos en las sociedades modernas. Uno de los principales retos para conseguir mejorar su funcionamiento es la navegación de los entornos. Existen multitud de algoritmos de navegación que constantemente se refinan y se adaptan a la aparición de mejores sensores y actuadores. La investigación en este campo ofrece resultados muy eficientes, con robots que, sin conocimiento previo del entorno de navegación, son capaces de evitar los obstáculos y encontrar el camino más corto hacia el objetivo.

Uno de estos algoritmos de navegación terrestre es *Tangent Bug*, que será el centro de este estudio. Forma parte de la familia de algoritmos *Bug*, pero dónde los demás se basan en sensores por contacto, este algoritmo trata con datos del entorno recibidos por sensores con rango. Además, los datos del sensor se usan para el así llamado *Local Tangent Graph*, un análisis de los datos que logra acortar mucho las distancias recorridas por el robot.

Estos elementos hacen de *tangent bug* un algoritmo con alto potencial si se implementa correctamente y se adapta a condiciones reales. Para simular estas condiciones se utilizará *MORSE* que se combina con *ROS* para la comunicación entre simulador y algoritmo. El algoritmo se implementará en *Python*, mientras que los entornos serán diseñados con *Blender*.

CAPÍTULO 1

INTRODUCCIÓN

En esta introducción se expondrán las motivaciones y objetivos de este trabajo. También se resumen los algoritmos de navegación y sus características, explicando en concreto la familia de algoritmos a la que pertenece *tangent bug*.

1.1 Motivación

En la actualidad uno de los campos en la tecnología con mayor auge y proyección es el de la robótica. Aún siendo una herramienta ampliamente extendida en la industria de todo tipo, no se puede decir que por el momento exista un uso generalizado en otros ámbitos.

Dentro de la robótica, uno de los problemas fundamentales es la navegación, como se ve en la figura 1.1. Es necesario evitar los obstáculos, pero también mejorar la eficiencia reduciendo el camino que el robot recorre. Un algoritmo de navegación interesante es *tangent bug*, que presenta un alto rendimiento proporcionando caminos muy cercanos al más corto posible. Este algoritmo es el centro de este trabajo.

La propuesta de realizar esta implementación surge de la falta de recursos encontrados sobre este algoritmo. Todas las implementaciones del algoritmo encontradas en la web aplican el algoritmo con fallos importantes. Además, no existe ninguno basado en *ROS* y *MORSE*, que es el software de simulación elegido para esta implementación. El código se implementa y usando el lenguaje *Python*. Hemos usado *Python* para aprovecharnos de su librería gráfica *Turtle graphics*.

1.2 Objetivo

El objetivo de este trabajo de fin de grado es la implementación del algoritmo de navegación para robots terrestres *Tangent Bug*. Es un algoritmo perteneciente a la familia de algoritmos *Bug*, que se encuentran entre los algoritmos de navegación más populares en robots de este tipo.

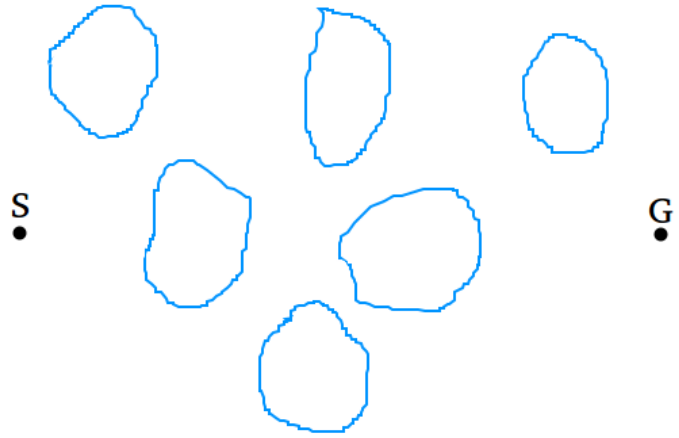

Figura 1.1: Situación en la que el robot situado en el punto S no puede alcanzar el punto G objetivo sin emplear un algoritmo de navegación. Los obstáculos se presentan por las formas azules.

Para lograr el objetivo propuesto es necesario llevar a cabo una implementación correcta del algoritmo respetando la idea original y cuando sea necesario hacer adaptaciones para que funcione en entornos realistas. Para validar dicha implementación, los resultados serán comprobados en entornos complejos que se aproximen a posibles situaciones que el robot se encontraría en el mundo real.

1.3 Algoritmos de navegación

Existen multitud de algoritmos de navegación en robótica que se diferencian por varios factores como los sensores del robot, la planificación u otras propiedades. En función de las diversas maneras de planificar el recorrido, los algoritmos de navegación se clasifican de acuerdo a los tres siguientes paradigmas [5]:

- *Deliberativos*: Basados en el paradigma *Sense-Plan-Act* pueden disponer de un mapa del entorno previamente a la ejecución o puede que este mapa lo construya el algoritmo durante la ejecución. En ambos casos el algoritmo usa un mapa del entorno para planear las órdenes.

- *Reactivos*: En estos algoritmos no existe planificación global, ni se utiliza mapa del entorno. Solamente mediante *Sense-Act*, estos algoritmos reaccionan a la información de los sensores de manera inmediata, sin planificación global.

- *Híbridos*: Una mezcla de los dos paradigmas anteriores, ejecutan una capa deliberativa de manera paralela a una capa de reactiva. Su paradigma sería *Plan, Sense-Act*. De esta manera los algoritmos híbridos se aprovechan de lo mejor de los dos otros tipos de algoritmo, realizan una planificación global pero tiene una rápida capacidad de reacción local.

Además del tipo de paradigma en el que un algoritmo se basa, los resultados de navegación dependen también del tipo de sensor con las que el algoritmo obtiene la información del entorno por el que navega el robot. En los algoritmos *Bug*, los dos tipos de sensores más relevantes son los siguientes:

- *Sensor de contacto*: Detectan los obstáculos al entrar en contacto físico con ellos. Por tanto, ofrecen un campo de visión nulo o cero.
- *Sensor de rango*: Pueden funcionar mediante diferentes tecnologías: láser, ultrasonidos etc. En todos estos casos, los sensores dan información sobre el entorno cercano al robot, con un campo de visión variable en función de la longitud del rango de detección del sensor.

Otra propiedad importante de los algoritmos de navegación es la convergencia. Se dice que un algoritmo es convergente si de existir un camino posible entre el robot y el objetivo, el algoritmo asegura que el robot acabará alcanzando dicho objetivo.

El algoritmo escogido, *Tangent Bug*, es un algoritmo reactivo. La decisión de implementar un algoritmo con este paradigma es por su rápida ejecución. Ya que no planifica a nivel global, el algoritmo *Tangent Bug* se puede usar sobre robots con pocos recursos computacionales, con baja memoria y poca velocidad de procesador. Estos robots de bajo coste son los más extendidos y por estas razones hemos escogido un algoritmo reactivo.

Seguidamente, describiremos las principales características del conjunto de algoritmos de tipo *Bug*.

1.4 Algoritmos *Bug*

Los algoritmos de la familia *Bug* reciben este nombre por su comportamiento parecido al de los insectos, que por su campo de visión limitado han de ir siguiendo el contorno de un obstáculo hasta encontrar un camino libre en la dirección en la que quieren ir [1].

Estos algoritmos sólo necesitan como información previa a su ejecución las coordenadas globales del objetivo, con la que podemos calcular la distancia que separa el robot del objetivo en todo momento. Durante la ejecución, se usa la información del entorno que le proporcionan los sensores del robot para dirimir las órdenes.

Siguiendo la clasificación establecida, todos los algoritmos de tipo *Bug* son reactivos y convergentes, con excepción del algoritmo *Bug* 0 que no es convergente. Por ello, son un grupo de algoritmos muy usados en navegación terrestre.No obstante, estos algoritmos también tienen una serie de inconvenientes, siendo los principales los que se enumeran a continuación:

- Suponen que el robot ocupa un único punto en el espacio. A la hora de implementarlos para usos prácticos, esto hace que tengan que ser modificados en algunos aspectos para incorporar el radio del robot al algoritmo, respetando siempre la estructura principal.
- Suponen que los sensores son perfectos proporcionando siempre información precisa, sin errores, del entorno que rodea el robot. Al ser esta suposición poco

realista, se deberán añadir ciertos márgenes de tolerancia a las condiciones impuestas por estos algoritmos.

- Los entornos por los que navegan los robots usando estos algoritmos han de ser estáticos, es decir, se asume que ningún obstáculo se mueve.

De manera general, la ejecución de estos algoritmos se puede dividir en dos comportamientos diferentes:

- *Motion to Goal*: El robot se dirige al objetivo si existe un camino libre hacia él.
- *Boundary Following*: El robot evita los obstáculos recorriendo su contorno hasta encontrar un camino libre hacia el objetivo.

Estos comportamientos se implementan de manera diferente en cada algoritmo de tipo *Bug*. Veamos a continuación una descripción de los principales algoritmos *Bug*.

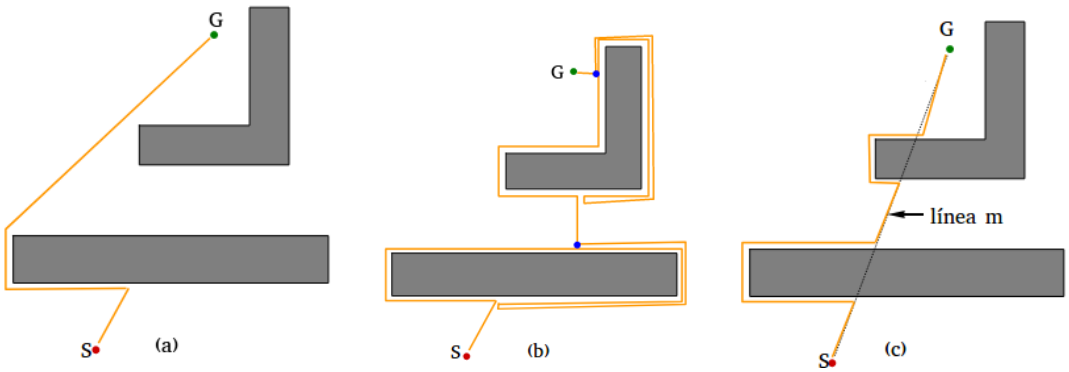

Figura 1.2: Caminos generados por los algoritmos: (a)Bug 0, (b)Bug 1, (c)Bug 2. Imagen extraída de [1].

1.4.1 El algoritmo *Bug* 0

El robot se mueve hacia el objetivo hasta que encuentra un obstáculo. Entonces, entra en seguimiento del contorno del obstáculo (con dirección arbitraria) hasta que pueda ir hacia el objetivo de nuevo. Un recorrido resultado del algoritmo *Bug* 0 se puede observar en la figura 1.2.a.

1.4.2 El algoritmo *Bug* 1

La siguiente versión de estos algoritmos denominada *Bug* 1, requiere que el robot tenga cierto grado de memoria. Al llegar a un obstáculo se recorre su contorno por completo recordando el punto en el que el robot estuvo más cerca del objetivo,tal y como muestra la figura 1.2.b. Una vez el robot se encuentra en la posición en la que había empezado a seguir el obstáculo, se dirige hacia ese punto mínimo para luego moverse en línea recta hacia el objetivo.

1.4.3 El algortimo *Bug* 2

En *Bug* 2 se usa la línea que une el punto de inicio del robot con el objetivo como referencia; a esa línea se le llama línea *m*. Iniciando el recorrido sobre esta línea, el robot se desvía de ella cuando un obstáculo bloquea su camino. En este algoritmo, se deja de seguir el contorno del obstáculo una vez el robot vuelve a pasar por la línea *m*.

1.4.4 El algoritmo *Tangent Bug*

Todos los algoritmos anteriores utilizan información obtenida mediante sensores de contacto. Parece claro que con sensoresque tengan un rango de detección mayor se podrían mejorar los resultados. El algoritmo que lleva esto acabo es el centro de este este estudio: *Tangent Bug*. La razón por la que se ha decidido estudiar e implementar este algoritmo queda clara observando la figura 1.3. En general, el algoritmo *Tangent Bug* genera caminos al objetivo mucho más cortos que los algoritmos *Bug* 0, *Bug* 1 y *Bug* 2. En la figura 1.3.b se representan dos resultados del algoritmo *Tangent Bug*. La

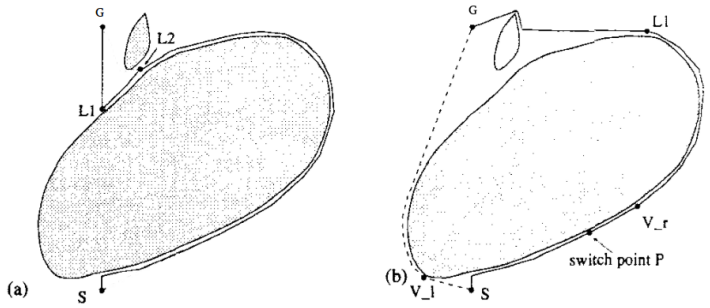

Figura 1.3: Camino generados por los algoritmos (a)*Bug*2 ,(b)*Tangent Bug*. Imagen extraída de [2].

línea continua corresponde con el camino que se obtendría con un robot que tuviera un rango de detección de cero, que equivale a decir que el robot dispone de sensores de contacto. Por otro lado, la línea discontinua representa el camino que se obtendría con un robot que tuviera un mayor rango de detección. Como se puede observar, al aumentar el rango de detección, el camino mejora significativamente En la figura 1.4vemos el resultado del algoritmo *Bug* 2 en un entorno de tipo oficina. Usando este mismo entorno, podemos compararlo con las trayectorias generadas por el algoritmo *Tangent Bug* con diversos rangos de detección/visión.

En la figura 1.5,el rango de visión aumenta de izquierda a derecha. Comparando la figura 1.4 con la figura 1.5 vemos como, incluso con un rango de visión muy limitado, el algoritmo *Tangent Bug* reduce la distancia recorrida obtenida por *Bug*2.A modo de resumen, podemos decir que el algoritmo *Tangent Bug* es capaz de encontrar caminos más eficientes que los del resto de la familia de algoritmos *Bug* en la mayoría de los casos. Este hecho justifica el papel central que tiene el algoritmo *Tangent Bug* en este trabajo de final de grado.

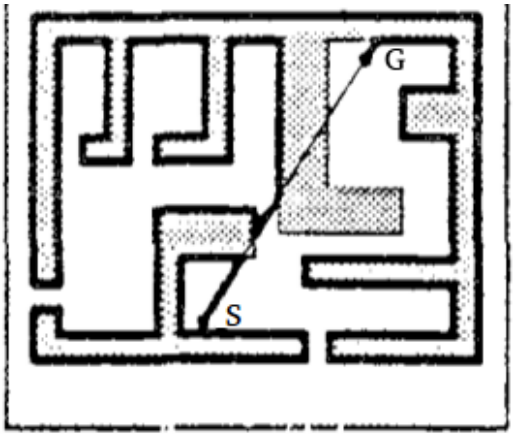

Figura 1.4: Camino generado por el algoritmo *Bug* 2 en un entorno complejo. Imagen extraída de [2].

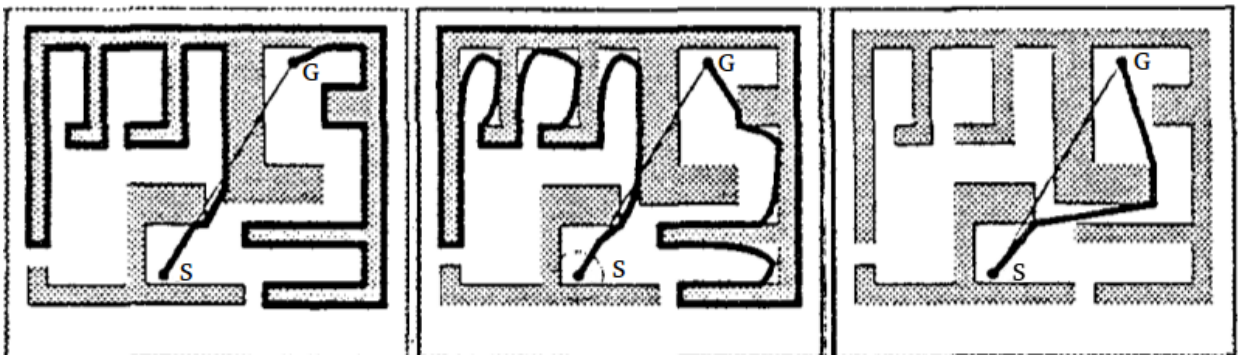

Figura 1.5: Camino generado por por el algoritmo *Tangent Bug* en un entorno complejo utilizando diferentes rangos de detección. Imagen extraída de [2].

1.5 Estructura del documento

Esta memoria se divide en varios capítulos. En el capítulo2 se describe el algoritmo *Tangent Bug*, explicando su estructura y funcionamiento. En el capítulo 3 se expone la implementación realizada con una introducción del *software* utilizado, una exposición de los principales problemas y una explicación de todos los métodos empleados y sus funciones. Posteriormente, se exponen los resultados experimentales en el capítulo 4, en el que se presentan y analizan las pruebas realizadas. Por último, en el capítulo 5 de conclusiones se valora el desarrollo del trabajo y los resultados obtenidos.

CAPÍTULO 2

DESCRIPCIÓN DEL ALGORITMO *Tangent Bug*

Tangent Bug fue introducido por Ishay Kamon, Ehud Rivlin y Elon Rimon. En el artículo *A New Range-Sensor Based Globally Convergent Navigation Algorithm for Mobile Robots* sentaron las bases para este algoritmo [2].

Para su planificación local, *Tangent Bug* utiliza los datos que le proporciona su sensor de rango para calcular una dirección local óptima basándose en una estructura llamada *Local Tangent Graph* (LTG). Este grafo se usa tanto en su comportamiento *Motion to Goal* como en su comportamiento *Boundary Following*.

2.1 *Local tangent graph*

El grafo se genera a partir de los obstáculos detectados en el rango de visión y conociendo la posición actual del robot (R) y del objetivo (G). Los rangos en los que hay detección son divididos en diferentes obstáculos por los puntos de discontinuidad y cada uno se modela como una pared fina. Esto conlleva una subestimación del tamaño del obstáculo, pero un modelo más preciso requiere cálculos más complicados. Los puntos *Oi* son los que delimitan los obstáculos y a partir de ellos se calculan las distancias para elegir el camino óptimo.

Sobre un grafo como el representado en la figura 2.2 podemos calcular las distancias entre los *Ois*, el robot y el objetivo. Estas distancias y su relación entre ellas dictan las órdenes en los comportamientos *Motion to Goal* y *Boundary Following* [6].

2.2 El comportamiento *Motion to Goal*

Durante este primer comportamiento, el robot se dirige directo al objetivo siempre y cuando no detecte ningún obstáculo que bloquee ese camino. De ser así, el algoritmo asigna a los *Oi* que delimitan los obstáculos una distancia heurística que se calcula de la siguiente manera:

$$D_{heur} = d(R,Oi) + d(Oi,G) \tag{2.1}$$

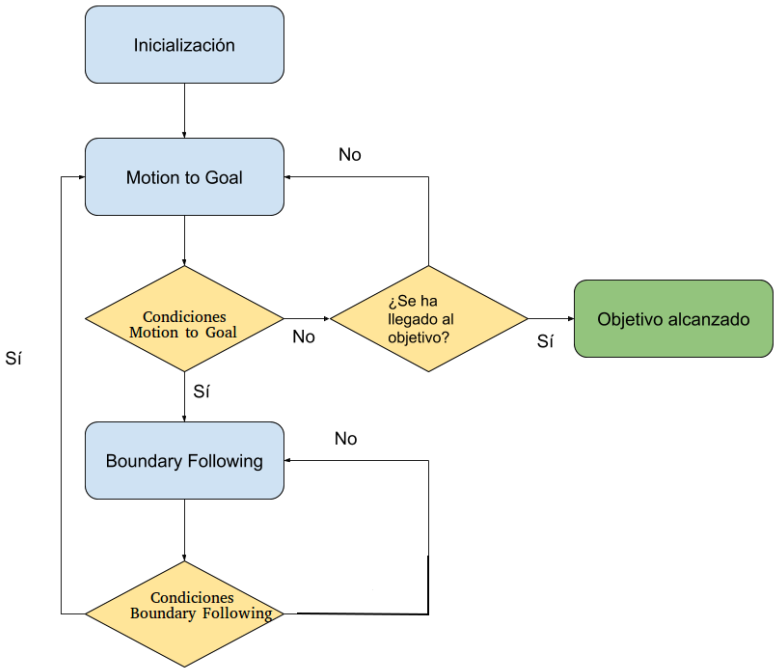

Figura 2.1: En este diagrama vemos como el algoritmo *Tangent Bug* funciona alternando los dos comportamientos principales: *Motion to Goal* y *Boundary Following*. Las condiciones de cambio de cada comportamiento se exponen en las explicaciones de este capítulo.

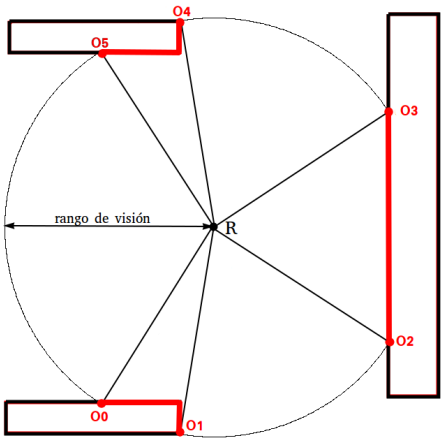

Figura 2.2: En esta figura se observa un ejemplo de *Local Tangent Graph*. Los puntos *Oi* delimitan los obstáculos detectados las partes del obstáculo en rojo son las que el robot detecta y en negro las que están fuera del rango. Este rango se representa con el círculo y en el centro se encuentra el robot.Las líneas que conectan el robot con los *Oi*, representan los rangos en los que el robot detecta obstáculos.

La distancia heurística, como muestra la ecuación 2.1, se calcula mediante la suma de la distancia euclídea entre la posición global del robot y la posición global del *Oi* más

la distancia euclídea entre la posición global del *Oi* y la posición global del objetivo.

El *Oi* que presente menor distancia heurística será el que marque la dirección del robot, que se dirigirá hacia él. Esto ocurrirá siempre que esta distancia vaya disminuyendo, aunque el *Oi* con menor heurística cambie. Es por eso que este comportamiento también conlleva algunas situaciones en las que se sigue el contorno del obstáculo, pero no es un cambio de modo aún, ya que puede cambiar de obstáculo al que sigue [7] [8].

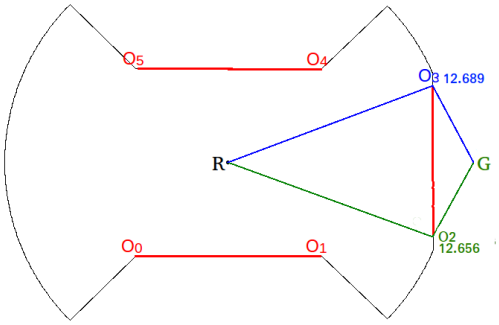

Figura 2.3: LTG en un momento durante la ejecución del comportamiento *Motion to Goal*.Los *Oi* que que son válidos tienen su distancia heurística representada y se les asigna un color: verde al que tenga la distancia heurística más corta y azul a los demás.*O3* se representa en azul y su distancia heurística es de 12.689 metros. *O2* se representa en verde y su distancia heurística es de 12.656 metros.

Como se ve en la figura 2.3, no todos los *Oi* son válidos para dirigirse hacia ellos. Esto se debe al filtrado que se aplica sobre ellos. En este caso, *O0* y *O5* están por detrás del robot y, por tanto, más alejados del objetivo que él. *O1* y *O4* están más cerca del objetivo que el robot, pero existe un obstáculo frente al robot que interseca con la línea entre estos *Oi* y el objetivo. Después del filtrado quedan *O2* y *O3*,de los cuales *O2* tiene la menor distancia heurística. Cualquier *Oi* que esté más alejado del objetivo que el robot o los *Oi* que tengan un obstáculo entre ellos y el objetivo, serán descartados para el cálculo de la posible dirección [9].

En la figura 2.4 la trayectoria del robot se representa con la línea verde y los obstáculos de colo rojo.Se observa un recorrido realizado mediante el comportamiento *Motion to Goal* donde se ve como el robot esquiva los obstáculos siguiendo la trayectoria más fluida y directa.

En cualquier instante de la ejecución, si el camino hacia el objetivo se despeja, *Motion to Goal* dirige el robot hacia el objetivo, sin tener en cuenta ningún *Oi* ni distancias heurísticas. En el instante en que la distancia heurística aumenta o ningún *Oi* supera el filtrado se cambiará el modo de ejecución a *Boundary Following*.

2.3 El comportamiento *Boundary Following*

El segundo comportamiento del algoritmo *Tangent Bug* se inicia siempre partiendo de que las condiciones para mantenerse en *motion to goal* no se cumplen. Primeramente, la selección de la dirección en la que el robot seguirá el contorno del obstáculo se hace

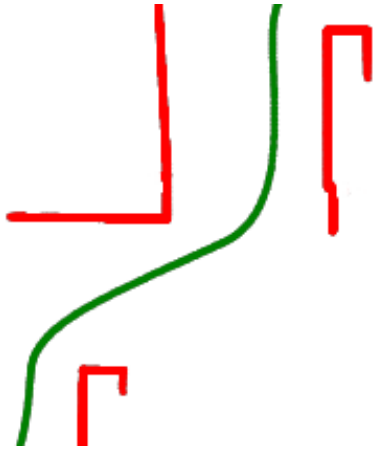

Figura 2.4: Ejemplo del recorrido resultante en una ejecución del algoritmo *Tangent Bug* durante el comportamiento *Motion to Goal*.

en función del *Oi* al que se dirigía antes de entrar en este modo. Si el cambio se produce por falta de un *Oi* válido, la dirección será la del último *Oi* que sí lo era.

Existen dos distancias, denominadas *dmin* y *dleave*, que se necesitan calcular durante el funcionamiento del comportamiento *Boundary Following*:

- *dmin* es la distancia entre un punto M y el objetivo. El punto M pertenece al obstáculo que bloquea el camino libre y es el que está más cerca del objetivo. El punto M se calcula sólo una vez al inicio del comportamiento *Boundary Following* y permanece inalterado durante toda su ejecución.

- *dleave* es la distancia entre un punto L y el objetivo. El punto L se define de igual forma que el punto M, es decir, es el punto del obstáculo que sigue el robot que se encuentra más próxima al objetivo. La principal diferencia entre los puntos M y L es que este último se calcula en cada instante de tiempo, y no sólo una vez como el punto M. La distancia *dleave* se calcula de una forma especial cuando existe un camino libre de obstáculos entre el robot y el objetivo. Cuando esto ocurre, *dleave* se redefine como la distancia entre un punto T y el objetivo. Este punto T se sitúa en el límite del campo de visión en dirección al objetivo.

Durante *Boundary Following* el robot se dirige hacia un *Oi* igual que en el comportamiento *Motion to Goal*. El primer *Oi* al que se dirige es el último *Oi* válido durante *Motion to Goal*,posteriormente se dirigirá hacia el *Oi* que tiene una posición más cercana a la posición del *Oi* al que se dirigía en el instante anterior. De esta manera se hace un seguimiento del contorno del obstáculo pero aprovechándose de los atajos que surgen resultado del *Local Tangent Graph*.

$$dleave < dmin \tag{2.2}$$

Como se observa en la condición 2.2, para pasar al comportamiento *motion to goal*la distancia *dleave* tendrá que ser menor que *dmin*. Si esta condición no se cumple y el robot ha completado un recorrido por todo el perímetro del obstáculo, el objetivo es inalcanzable.

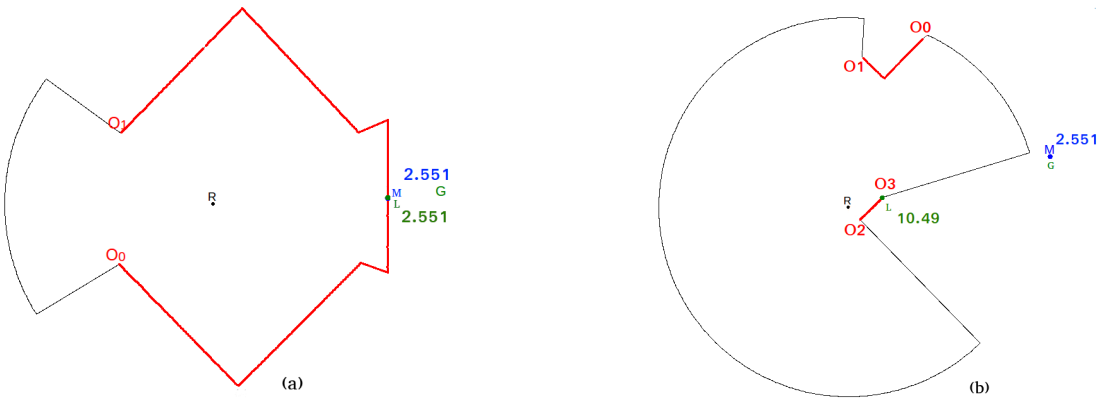

Figura 2.5: Dos momentos durante la ejecución del comportamiento *Boundary Following*.

Analizando la figura 2.5, vemos como,en el momento en el que el comportamiento *Boundary Following* se activa, el punto M y L coinciden y las distancias *dmin* y *dleave* son iguales, con un valor de 2.551 m. Más tarde en la ejecución, el punto M deja de estar en el campo de visión del sensor y se dibuja en azul en la dirección en la que se encuentra, fuera del rango. El punto L se dibuja en verde.*dmin* sigue valiendo 2.551 m y sigue siendo no mayor que *dleave*, 10.49 m.

En el ejemplo de la figura 2.6 la distancia *dleave* se calcula con el punto T.

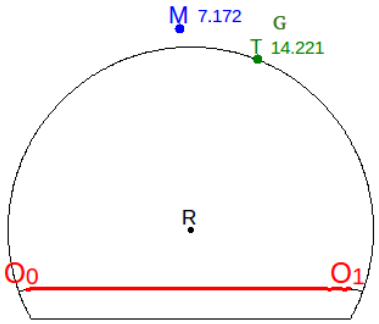

Figura 2.6: Ejecución del *boundary following* con punto T.

La condición 2.2 y el funcionamiento del comportamiento *Boundary Following* aseguran que el algoritmo *Tangent Bug* sea convergente. Es decir, garantizan que el robot acabará alcanzando el objetivo, siempre que el objetivo sea alcanzable.

En la figura 2.7 se observan los beneficios de usar el LTG en este comportamiento, ya que se atajan las esquinas acortando las longitud total recorrida.

2.4 Ejemplos de ejecuciones del algoritmo *Tangent Bug*

A continuación se pueden ver algunos ejemplos de los recorridos resultantes del algoritmo *Tangent Bug*. Estos resultados son analizados a fondo en el capítulo de resultados

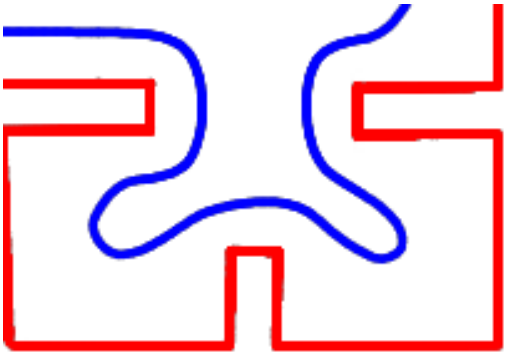

Figura 2.7: Ejemplo del recorrido resultante en una ejecución del algoritmo *Tangent Bug* durante el comportamiento *Boundary Following*.

experimentales.

En las representaciones de la figura 2.8 los obstáculos que han sido detectados aparecen en rojo, mientras que el recorrido del robot es verde para los tramos realizados por el comportamiento *Motion to Goal* y en azul se representan los tramos con comportamiento *Boundary Following*. Los puntos R y G están marcados además de un círculo en la posición inicial del robot que marca su campo de visión.

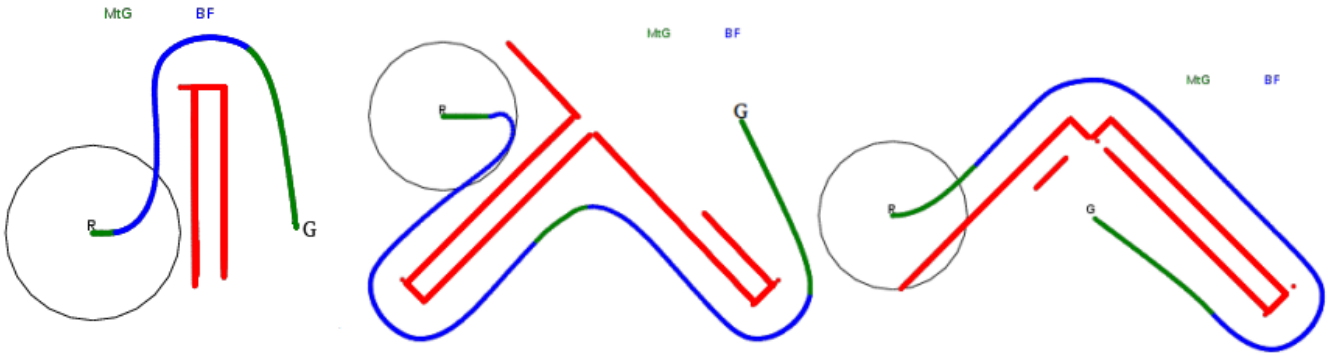

Figura 2.8: Recorridos obtenidos mediante el algoritmo *Tangent Bug*.

2.5 Diagrama de flujo

La siguiente figura 2.9 representa el diagrama de flujo del algoritmo *Tangent Bug* y facilita la comprensión de su funcionamiento.

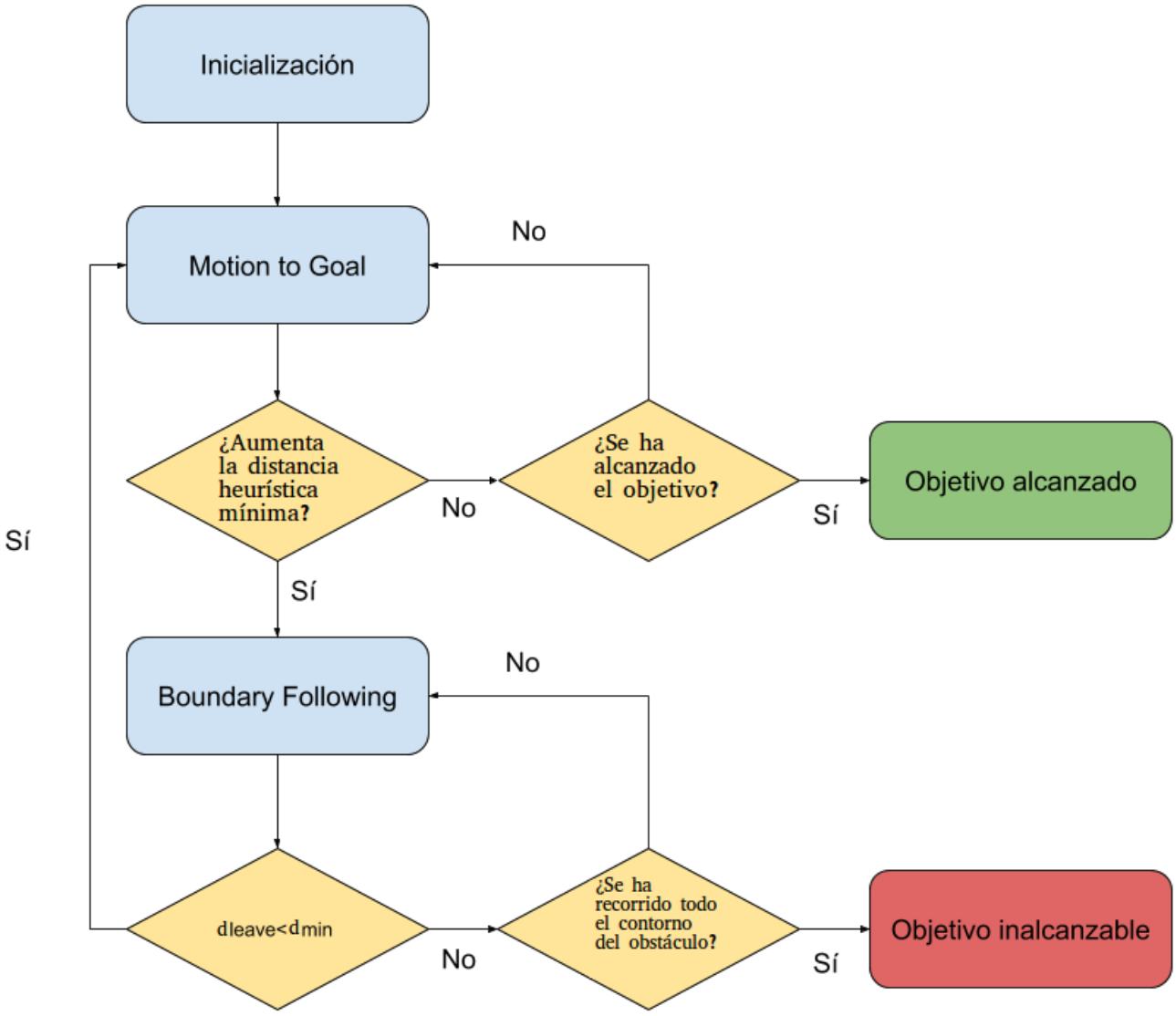

Figura 2.9: Diagrama de flujo detallado del algoritmo *Tangent Bug*.

CAPÍTULO 3

IMPLEMENTACIÓN

La implementación del algoritmo *tangent bug* se ha realizado en *Python*. Para simular se utiliza *MORSE* con un entorno gráfico creado mediante *Blender*. La comunicación se hace a través de *ROS* que funciona mediante publicación y suscripción de la información de los sensores y los actuadores.

Inicialmente, en este capítulo analizaremos las implementaciones previas del algoritmo *Tangent Bug*. Después, se introduce el *software* utilizado, *ROS* y *MORSE*, y los elementos usados en la simulación. Se exponen los problemas encontrados durante la implementación y como se han solucionado. Finalmente, se explican los métodos y los parámetros del programa. Además, se muestra un diagrama de flujo de la ejecución del programa.

3.1 Implementaciones previas

Una de las razones de ser de este trabajo de final de grado es la falta de implementaciones públicas del algoritmo *Tangent Bug* usando el *software* elegido. Además, las que encontramos en otros entornos, tal como *MATLAB* y páginas *HTML*, no funcionan correctamente en todas las situaciones y no simulan el algoritmo en condiciones cercanas a las reales como lo hace *MORSE*. Las simulaciones del algoritmo *Tangent Bug* que se han analizado de forma previa a nuestro desarrollo han sido las siguientes.

3.1.1 *JavaScript*, Baran Kahyaoglu

Basada en el lenguaje *JavaScript*, la implementación del algoritmo *Tangent Bug* de Baran Kahyaoglu está acompañada de una explicación teórica del algoritmo. El entorno gráfico es muy informativo de las trayectorias y decisiones que toma el robot. En las mismas explicaciones del algoritmo ya se subrayan algunos fallos que presenta esta implementación [3]. La figura 3.1.a representa una correcta ejecución, pero en la figura 3.1.b vemos un fallo en el que el robot no es capaz de acabar la ejecución, debido a la indecisión sobre que dirección tomar que le genera un entorno estrecho como un

pasillo. En el capítulo 4, veremos como nuestra implementación del algoritmo *Tangent Bug* consigue llegar al objetivo en el mismo entorno que el de la figura 3.1.b.

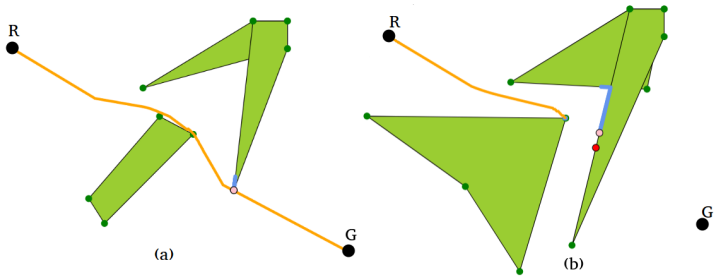

Figura 3.1: Simulación del algoritmo *Tangent Bug* en un entorno web. Imagen extraída de [3].

3.1.2 *ROS/STAGE*,Filipe Correia

Esta versión del algoritmo utiliza *Tangent Bug* un entorno *ROS* al igual que en nuestro caso, pero el simulador es *STAGE*, un simulador también muy usado en robótica pero que sólo puede simular entornos dos dimensiones. El algoritmo está implementado en *C++*, el otro lenguaje, a parte de *Python* soportado en *ROS*.Un ejemplo de simulación en *STAGE* se puede ver en la figura 3.2 donde se observa el robot (en rojo) y el campo de visión (en azul).

En el archivo *Read Me* que acompaña al código se señalan los problemas que tiene esta implementación como que ocurren colisiones e indecisiones que pueden llevar a un bucle infinito en el que robot no sabe qué hacer. Estos fallos no se dan en la implementación realizada en este trabajo; las colisiones son inexistentes y las indecisiones se evitan gracias a un bloqueo de las decisiones tomadas cuándo el cambio de decisión ocurre dentro de un margen establecido [10].

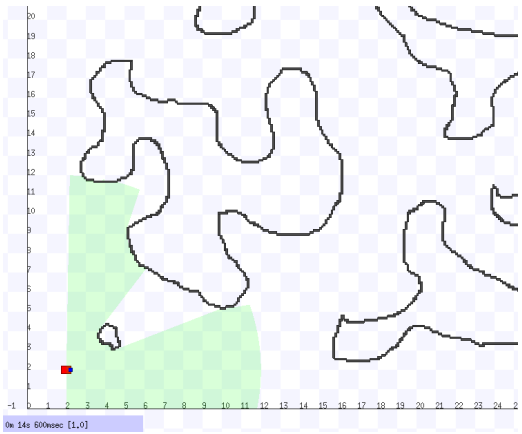

Figura 3.2: Entorno 2D en *Stage*.

3.1.3 *MatLab*, Mehmet Mutlu

Esta implementación en *MATLAB* logra un funcionamiento bastante correcto del algoritmo *Tangent Bug*, sin fallos graves de ejecución. Utiliza un simulador teórico, sin físicas y asume que el robot es un punto.

En las figura 3.3, en la que se representa en rojo los tramos de la trayectoria del robot generada en el modo *Motion to Goal* y en azul aquellos tramos generados en el modo *Boundary Following*, se ven dos ejecuciones correctar del algoritmo *Tangent Bug*. En la figura 3.3.b se observa como, al cambiar de modo a *Boundary Following*, se decide seguir el contorno del obstáculo hacia la izquierda, cuando habría sido mejor decisión hacerlo hacia la derecha. En el capítulo 4 veremos como nuestra implementación del algoritmo *Tangent Bug* toma mejor este tipo de decisiones, generando así caminos más cortos al objetivo [11].

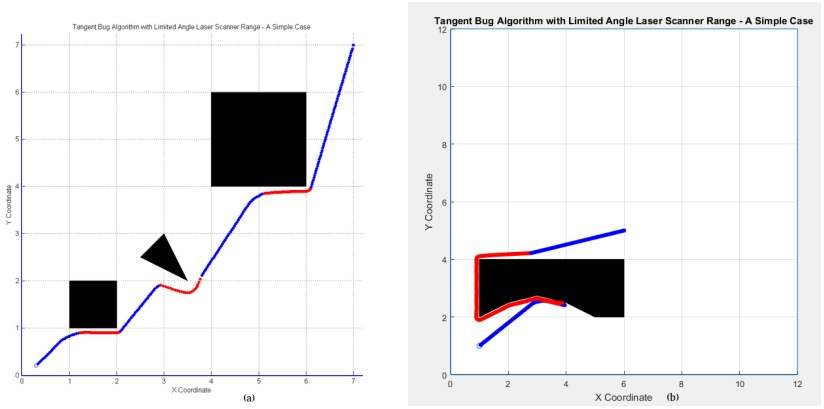

Figura 3.3: Simulación del algoritmo *Tangent Bug* en *MATLAB*.

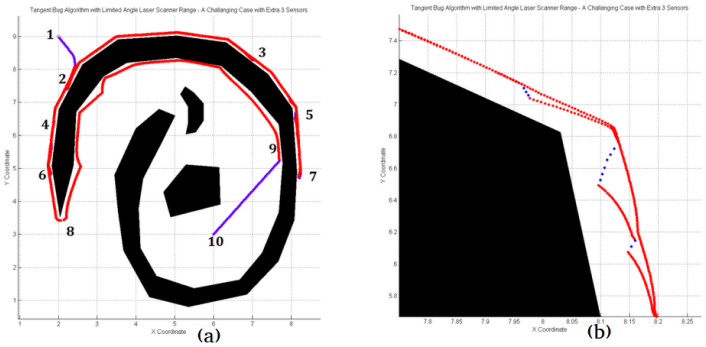

Figura 3.4: Continuos cambios entre los comportamientos *Motion to Goal* y *Boundary Following* que provocan una ejecución incorrecta del algoritmo *Tangent Bug*.

En la ejecución de la figura 3.4 se ve un ejemplo de fallo en esta implementación. En la figura 3.4.a se ve al ejecución total y en la figura 3.4.b se muestra el detalle de la situación donde falla. El error se produce porque el robot cambia muchas veces de modo en poco tiempo. Esto se debe a que se han implementado las condiciones que determinan los cambios de modo de manera estricta; nosotros hemos flexibilizado estas condiciones para evitar problemas.

3.2 *ROS y MORSE*

ROS (*Robot Operating System*) y *MORSE* (*Modular Open Robots Simulation Engine*) son el *software* sobre los hemos realizado la implementación del algoritmo *Tangent Bug*. Cabe destacar algunas propiedades importantes de este *software* :

- Publicar y Suscribir: La forma de intercambiar información entre el simulador *MORSE* y el algoritmo *Tangent Bug* sigue un esquema de publicación y suscripción. La información se publica a un canal y los suscriptores de ese canal reciben una interrupción con la información publicada.
- *Topics*: Los topics son los canales por los que se publica la información. Los *topics* se definen con un nombre único y con un determinado tipo de datos.
- *MORSE y Blender*: *MORSE* se encarga de simular el robot, con sus sensores y actuadores, y el entorno en el que se moverá, definido por un archivo *.blend* creado mediante el *software* de diseño gráfico *Blender*. Los elementos que intervienen en una simulación se definen en el archivo *default.py* [12] [13] [14].

3.2.1 Elementos de la simulación

Para entender como hemos implementado el algoritmo *Tangent Bug* sobre el entorno *ROS/MORSE* es necesario conocer primero las principales características del robot que hemos simulado. Dicho robot es un modelo de la empresa *iRobot* denominado *All Terrain Robot Vehicle* o ATRV (ver figura 3.5) [4]. *MORSE* dispone de multitud de sensores y actuadores. Nuestro robot ATRV incorpora dos sensores y un actuador.

Figura 3.5: El robot real sobre el que se basan nuestras simulaciones en*MORSE*. Imagen extraída de [4].

- Láser *Sick*: Este sensor láser es el que proporciona información del entorno al robot. El sensor ofrece un rango de detección que configurable por parámetro. Otros parámetros de este sensor que se pueden configurar son: cobertura, frecuencia de publicación, y resolución(grados que separan cada haz de láser). El

objeto publicado es de tipo *laser* y contiene información sobre el rango de detección, los ángulos máximos y mínimos que delimitan la cobertura, el incremento de ángulo entre cada haz de láser y las medidas de distancia a los obstáculos de que haz.La figura 3.6 muestra un mensaje publicado por el sensor láser. En este caso, la cobertura del láser estaba configurada a 360 grados, su rango de detección era de 3 m y su resolución de 0.5 grados. El array *ranges* contiene todas las medidas tomadas por el sensor láser.

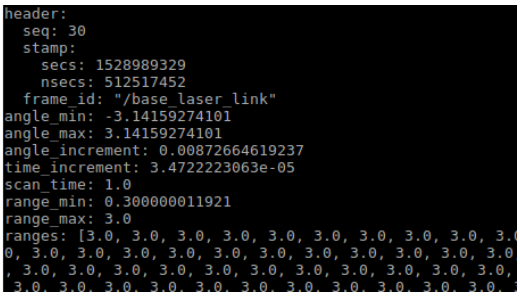

Figura 3.6: Mensaje que publica el sensor láser.

- *Pose*: Con un sensor de posición podemos obtener la posición y orientación del robot en coordenadas globales. La posición se publica en un vector (x,y,z) y la orientación en un cuaternión (x,y,z,w). En la figura 3.7 se ve que aunque el robot este en el origen, el sensor no da lecturas constantes y ninguna es 0 exactamente. Esto es una muestra de que *MORSE* no simula sensores perfectos, tal y como asume el algoritmo *Tangent Bug*.

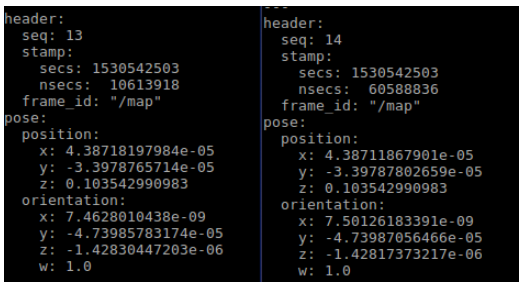

Figura 3.7: Mensaje que publica el sensor de posición.

- *MotionVW*: Actuador de velocidad lineal y angular. Se publica un objeto de tipo *Twist* que contiene dos vectores con valores (x,y,z), uno para cada velocidad, que definen el desplazamiento o la rotación sobre los ejes en m/s o rad/s, respectivamente.

Además de los elementos propios del robot, *MORSE* nos permite definir el entorno y algunos parámetros de simulación como la posición de la cámara, renderizar o no las texturas para agilizar la ejecución o definir la estrategia del tiempo de simulación para que ésta sea o no acorde con el tiempo real. Un ejemplo de simulación con todos los elementos comentados se muestra en la figura 3.8.

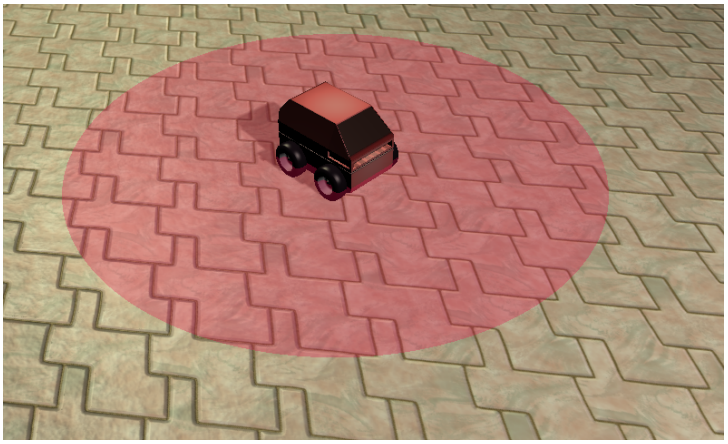

Figura 3.8: El robot ATRV equipado con un sensor láser con coberture de 360 grados en un entorno sin obstáculos.

3.3 Problemas encontrados y soluciones dadas durante la implementación

Inicialmente, nuestra intención era implementar el algoritmo *Tangent Bug* tal y como se plantea en el artículo original. Sin embargo, esto no ha sido posible debido a que el algoritmo no está preparado para algunas situaciones reales que el simulador*MORSE* sí que considera. Se han tenido que realizar muchos cambios y ajustes sobre el algoritmo *Tangent Bug* original para poder conseguir que funcionase en la simulaciones de *MORSE*. Seguidamente, se describen todos estos cambios.

3.3.1 Hinchado de los obstáculos

La suposición teórica que hace el algoritmo *Tangent Bug* original de que el robot es un punto en el espacio es claramente falsa. Esta suposición puede dar lugar a que un robot real colisione con los obstáculos del entorno. Para solucionar este problemas decidimos hinchar los obstáculos en función del radio del robot. Esto se implementa en nuestro método *Inflate Obstacles*. Los resultados son los que se observan en la figura 3.9.

En la figura 3.9.a se observa que el robot se dirigirá hacia G directamente pasando así por el espacio que hay entre los dos obstáculos.Sin embargo, este espacio no es suficiente para que el robot pase y, por tanto, el robot acabará chocando con uno de esos obstáculos.

En la figura 3.9.b los obstáculos sí se han hinchado en función del radio del robot. Como se puede ver, el camino que se elegía en la figura 3.9.a ya no existe por lo que el robot tendrá que seguir los *Oi* que limitan el nuevo obstáculo que forma la fusión de los dos anteriores.

3.3.2 Desplazamiento del objetivo

Si dirigimos al robot directamente a los puntos *Oi* situados sobre el límite del obstáculo hinchado, puede ocurrir que el robot acabe entrando dentro de la zona "hinchada",

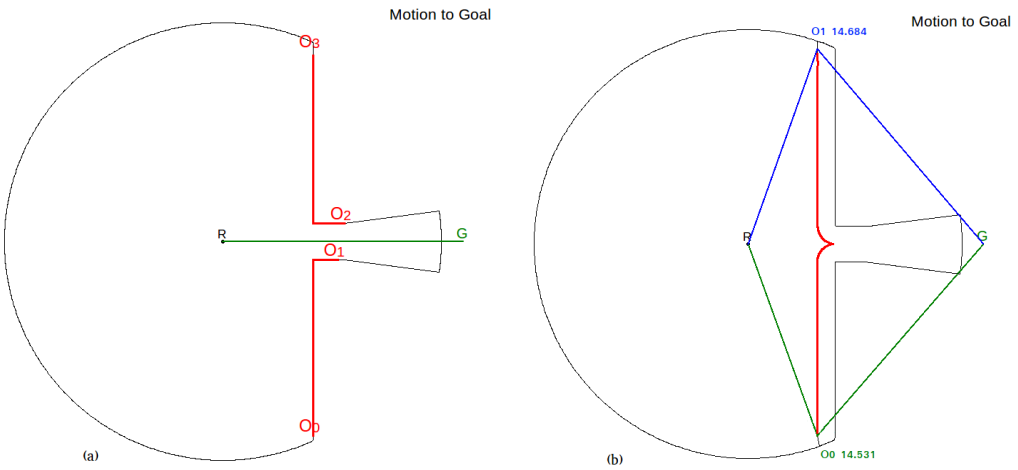

Figura 3.9: LTG con los obstáculos (a) sin hinchar e (b) hinchados.

es decir, la zona comprendida entre el obstáculo real y el obstáculo hinchado. Si esto ocurre, se producen muchos errores en los cálculos de distancias e intersecciones, así que es necesario evitarlo. Para ello, se elige un punto *waypoint* que se obtiene a partir del *Oi* que ha sido seleccionado. En el método *move target* se arregla este problema, pero hace variando un poco la idea original del algoritmo *Tangent Bug* ya que el robot no se dirige a los *Oi* exactamente, sino a un *waypoint*. Los resultados se observan en la figura 3.10.

Mover el punto al que el robot se dirige supone también la aplicación de un nuevo filtro sobre los *Oi*. Aquellos *Oi* en los que entre su *waypoint* correspondiente y el robot tengan un obstáculo, no serán válidos. Esta comprobación no era necesaria anteriormente para los *Oi*, ya que por su naturaleza siempre existía un camino libre entre el robot y el *Oi*.

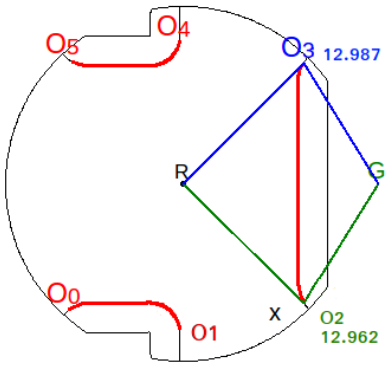

Figura 3.10: Para este LTG, el *Oi* seleccionado es *O2* y su correspondiente *waypoint* es *x*.

3.3.3 Indecisión o *zig-zag*

Una situación de *zig-zag* se da cuando el robot cambia de decisión sobre a que *Oi* se dirige en instantes consecutivos. En el algoritmo *Tangent Bug* se pueden dar estas situaciones debido a pequeños errores en las medidas tomadas por el sensor láser. Durante la ejecución del comportamiento *Motion to Goal* se ha evitado que el robot cambie de *Oi* al que se dirige si éste pertenece al mismo obstáculo que el *Oi* seleccionado en el paso de ejecución anterior. Esto ocurría especialmente cuando el robot se encontraba perpendicular a un obstáculo, como muestra la figura 3.11. Bloqueando el *Oi* seleccionado se evitan situaciones de *zig-zag* indeseadas.

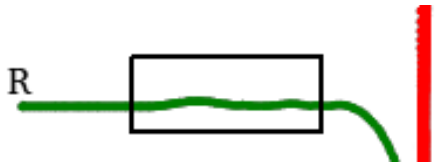

Figura 3.11: Ejemplo de una situación de indecisión. Con un recuadro negro se remarca la zona en la que se produce la indecisión.

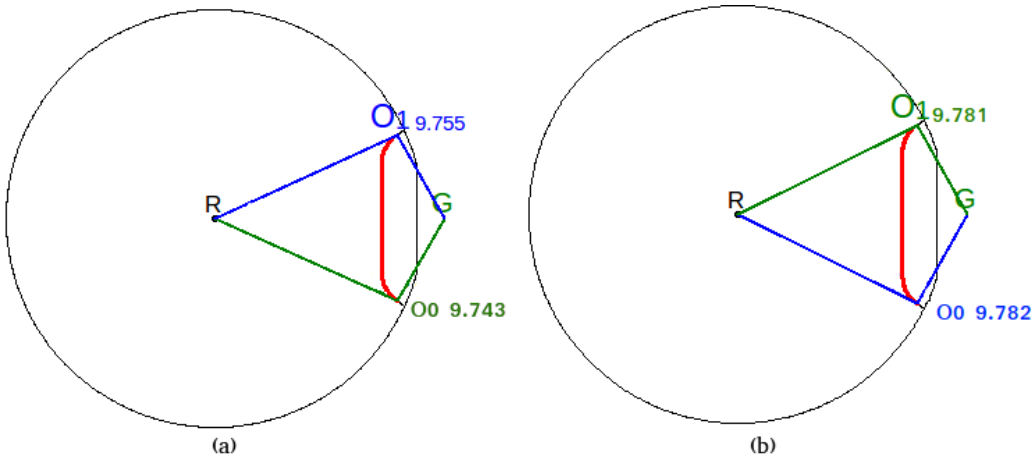

Figura 3.12: LTG durante la indecisión.

Analizando la figura 3.12, en el instante (a) el algoritmo escoge *O0* al tener una distancia heurística menor. En el siguiente instante (ver figura 3.12.b), el *Oi* con menor heurística es *O1* aunque sólo sea por un milímetro. Cuando esta alternancia en la selección de un *Oi* u otro (en este casa, *O0* y *O1*) se repite varias veces provoca que el robot se mueva según un patrón de *zig-zag*. Para evitar la situación de la figura 3.12, nuestra implementación del algoritmo *Tangent Bug* en el instante (a) guardaría *O0*. En el instante siguiente de tiempo, al ser *O1* un *Oi* perteneciente al mismo obstáculo que *O0*, se mantendría *O0* como *Oi* a seguir.

3.3.4 Cambios de modo

El algoritmo *Tangent Bug* establece que deberá cambiarse de modo *Motion to Goal* a *Boundary Following* cuando el mínimo valor de la heurística empiece a crecer entre

un paso de ejecución y el siguiente. Ese crecimiento se puede producir de forma no deseada debido a pequeños errores en las medidas tomadas por el sensor láser que pueden provocar aumentos o disminuciones momentáneas en los valores dela heurística que se calculan durante la ejecución del comportamiento *Motion to Goal*. Para evitar este problemas se ha añadido un contador previo al cambio de modo. Si la heurística crece durante N instantes seguidos (nosotros hemos tomado N=5), entonces ya sí se da por válida la condición de cambio y se lleva a cabo.

También se realizó un ajuste para el otro cambio, de *Boundary Following* a *Motion to Goal*. En este caso, la condición 3.1 se ha cambiado por la condición 3.2.

$$dleave < dmin \tag{3.1}$$

$$dleave + R < dmin \tag{3.2}$$

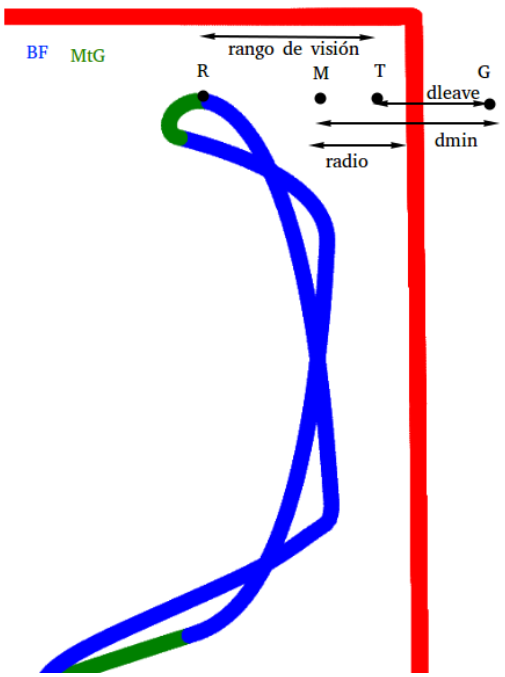

Figura 3.13: Cambio de modo incorrecto. Se ven las distancias y los puntos que afectan al cambio de modo de *Boundary Following* a *Motion to Goal* colocados en la situación del instante en que ocurre el error. *dleave* es la distancia del punto T a G. *dmin* la distancia entre M y G. Entre M y el obstáculo hay una distancia igual al radio del robot, ya que M se sitúa sobre el obstáculo hinchado (el obstáculo de la figura no está hinchado). Por último entre el robot y T la distancia es igual al rango máximo de visión. Como se observa, al ser *dleave* menor que *dmin* se produce este cambio indeseado.

Este cambio es necesario debido a que el punto M con el que se calcula la distancia*dmin* se sitúa los obstáculos hinchados. Si en algún momento ese mismo obstáculo deja de estar en el campo de visión del robot, la distancia *dleave* quedará representada por el punto T que será colocado sobre el límite del campo de visión y estará más cerca del objetivo que M, ya que no se hincha el obstáculo si no se detecta. Esto hace

que el cambio de modos tarde más en hacerse, pero se asegura de hacerse cuando es necesario. De lo contrario, ocurre el fallo que se observa en la figura 3.13, en el que el robot da la vuelta en vez de continuar siguiendo el contorno del obstáculo como debería.

3.3.5 Sincronización de los *callbacks*

Para explicar el problema que nos ha surgido en relación a los *callbacks*, es necesario primero introducir que son los *callbacks*. Los métodos *callback* son aquellos que se ejecutan mediante interrupción cada vez que se publica información de un tema al que se está suscrito. Son propios del funcionamiento de *ROS* y tratan con los datos de los sensores. En nuestro caso, tenemos dos *callbacks*: *callbackPose*, ligado al sensor de posición y orientación, y *callbackLaser*, que recibe las lecturas del sensor láser.

Nuestro programa siempre recoge primero las lecturas del sensor de posición y después las del sensor láser. Teniendo en cuenta esto, para tener sincronizadas ambas informaciones (es decir, para que la lectura del láser recibida en el *callbackLaser* haya sido tomada en la posición indicada por el *callbackPose*) es necesario que el código ejecutado en el *callback* del sensor *Pose* sea lo más corto posible. En el caso de no ser así se obtendrían resultados como el de la figura 3.14, donde una pared recta aparece algo torcida cunado representamos el entorno en nuestro código *DrawPath.py* que será explicado en el capítulo 4.

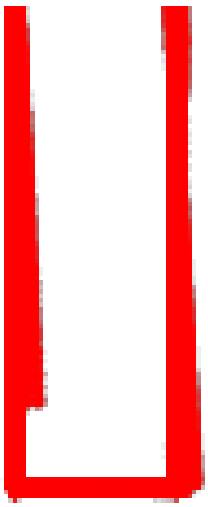

Figura 3.14: Error en la detección de un obstáculo.

3.4 Estructura del código

La estructura del código se rige por la misma estructura que el algoritmo. Dos métodos por cada uno de los comportamientos, *Motion to Goal* y *Boundary Following*. Para simplificar el código se genera una clase *tbug* y en ella se desarrollan todos lo métodos que

implementan el algoritmo *Tangent Bug*. Existen diversas variables globales definidas como *default* que serán los parámetros que se podrán ajustar del algoritmo.

3.4.1 Métodos

Los métodos se agrupan en varias categorías dependiendo de su función: para tratar los datos de los sensores, calcular las órdenes para los actuadores, representar gráficamente el *Local Tangent Graph* u otros cálculos necesarios. A continuación se resumen los métodos, sus funciones y como se relacionan entre ellos.

Callbacks

- *callbackPose*: En este método se guarda la posición y rotación actual del robot. Es importante que este método dure lo menos posible a fin de que el método *callbackLaser* se ejecute poco después.
- *callbackLaser*: Aunque este también sea llamado por interrupción, se bloquea su ejecución hasta que termina *callbackPose*, ya que se necesita la información de la posición y orientación actual del robot para tratar los datos del sensor láser. En este método se recibe la información del entorno en forma de distancia a la que se encuentran los obstáculos y se transforma a puntos en coordenadas globales para calcular las distancias necesarias y representar gráficamente el entorno.

La ejecución de cada *callback* se bloquea después de ser ejecutados y se libera *callbackPose* una vez se han mandado las órdenes al robot. *CallbackLaser* se desbloquea al finalizar *callbackPose*.

Métodos para los comportamientos

Una vez ejecutados los dos *callbacks* y habiendo guardado y tratado la información de los sensores, se procede al cálculo de las órdenes y las comprobaciones de las condiciones del algoritmo. Estas tareas serán diferentes en función del comportamiento y así están distribuidos los métodos:

- *Motion to Goal*: En este método se concentran varias llamadas a otros métodos, pero significativamente aquí se publican las órdenes de velocidad lineal y angular. Además en este instante se comprueba la condición de llegada al objetivo. De cumplirse esta condición, se publica la parada del robot y se finaliza la ejecución.
- *MtG distances*: Se calcula si existe un obstáculo que bloquea el camino directo del robot al objetivo. De ser así, se filtran los *Oi* ,se calculan las distancias heurísticas y se selecciona el *Oi* al que dirigirse, que será el que tenga una distancia heurística menor. Si se calcula que no hay obstáculo que bloquea el camino libre entre el robot y el objetivo, el robot se dirige hacia el objetivo.

  Para realizar estas operaciones se requieren de otros métodos que se expondrán posteriormente. En este instante se comprueba la condición de cambio de *Motion to Goal* a *Boundary Following* y si se cumple se ejecuta el cambio de modo, indicando que obstáculo el robot deberá seguir. También en el instante en que se cambia de modo se sitúa el punto M y se calcula la distancia *dmin*.

- *Boundary Following*: Siguiendo la misma estructura que en el modo *Motion to Goal*, en este método se realizan varias llamadas a otros métodos, para después publicar las órdenes correspondientes. Si el objetivo el inalcanzable, aquí se realizan las comprobaciones necesarias para reconocerlo, y parar el robot y la ejecución.

- *BF distances*: En este método se realizan los cálculos propios del modo *Boundary Following*. Se sitúa el punto L o T y se calcula la distancia *dleave*. También se asigna el *Oi* a seguir, que será el que esté más cerca de la posición en la que se encontraba el *Oi* al que el robot se dirigía en el instante de ejecución anterior. Una vez se calcula *dleave*, se compara con *dmin* para comprobar la condición de cambio. Si la condición de cambio se cumple, se pasa a *Motion to Goal*.

Métodos de cálculo geométrico

Los métodos expuestos anteriormente son la estructura principal del código, pero para funcionar necesitan otros métodos que realizan diferentes cálculos. Son los siguientes:

- *Transform angle*: En este método se recibe un punto al cual el robot debe dirigirse, el *waypoint*. Necesitamos calcular el ángulo que el robot deberá rotar para que quede mirando hacia *waypoint*. Mediante *atan2* se calcula el ángulo entre este punto y el robot, el cuál se denomina *target angle*. El problema es que este ángulo no tiene en cuenta la rotación *yaw* del robot. Para arreglar esto,se realizan las operaciones necesarias para transformarlo, que varían en función del cuadrante de *target angle*. Este ángulo global será *motion angle*, el ángulo que tendrá que rotar el robot sobre el eje Z para apuntar hacia *waypoint*.

- *Intersection*: Se reciben 3 puntos: P1, P2 y C. El método calculará si existe intersección entre la recta (P1,P2) y una circunferencia con centro en C. El cálculo geométrico es el siguiente:

$$y = m * x + d \tag{3.3}$$

$$m = \frac{y_2 - y_2}{x_2 - x_1} \tag{3.4}$$

$$d = y_1 - m * x_1 \tag{3.5}$$

  Calculamos la recta (P1,P2) (ver ecuaciones 3.3,3.4 y 3.5) y a continuación calculamos una recta perpendicular a (P1,P2) que pasa por C. El punto donde se cortan se obtiene de la manera que muestra las ecuaciones 3.6, 3.7, 3.8 y 3.9:

$$m_p = -\frac{-1}{m} \tag{3.6}$$

$$d_p = cy - m_p * c_x \tag{3.7}$$

$$x_{inter} = \frac{d_p - d}{m - m_p} \tag{3.8}$$

$$y_{inter} = x_{inter} * m + d \tag{3.9}$$

  Una vez obtenido este punto, podemos calcular la distancia entre el punto C y el punto donde intersecan las dos rectas. Si esta distancia es menor que el radio de

la circunferencia, entonces existe intersección. El radio viene definido por uno de los dos modos posibles de este método.

  - *Simple*:Este modo se usa durante el proceso de hinchado de los obstáculos, para detectar que puntos han ser hinchados. En este modo el radio de la circunferencia será igual al radio del robot y el punto C irá rotando por todos los puntos en los que se ha detectado obstáculo.
  - *Between*:Este modo se utiliza una vez los obstáculos ya están hinchados y es para reconocer si existe intersección con algún obstáculos entre el robot, los *Oi* y el objetivo. El radio de la circunferencia es el valor D, que es la distancia máxima que puede haber entre dos puntos de la detección del sensor láser. Su valor depende de los parámetros del sensor como su rango máximo o su resolución. Este modo nos reconoce la intersección solo si se produce en el segmento de la línea que va entre P1 y P2.

- *Inflate Obstacles*: En este método de nuevo se calcula la intersección entre una circunferencia y una recta. En este caso, debemos guardar los puntos en que está intersección ocurra, ya que serán los nuevos puntos de los obstáculos hinchados. Los puntos usados son: P1, la posición actual del robot, y P2, que irá rotando entre todos los puntos que el sensor láser ha recibido. El centro de la circunferencia C irá rotando entre todos los puntos detectados cómo obstáculos.

  La recta se calcula como aparece en las ecuaciones 3.3,3.4 y 3.5. La ecuación 3.10 es la del círculo.

$$(x - x_c)^2 + (y - y_c)^2 = R^2 \tag{3.10}$$

  Combinando la ecuación de la recta y el círculo existen las siguientes ecuaciones(ver 3.11, 3.12 y 3.13) para encontrar los dos puntos en que intersecan la recta y el círculo. Si se cumple la condición 3.14, entonces no existe intersección.

$$\rho = R^2 * (1 + m^2) - (y_c - m * a - d)^2 \tag{3.11}$$

$$x_{inter} = \frac{m * (y_c - d) + x_c \pm \sqrt{\rho}}{1 + m^2} \tag{3.12}$$

$$y_{inter} = m * x_{inter} + d \tag{3.13}$$

$$\rho < 0 \tag{3.14}$$

  El método ejecuta un bucle por todos los valores de P2, es decir, por todas las lecturas del sensor de láser. Para cada P2 se ejecuta un bucle de todos los puntos C , que son los puntos en que se ha detectado un obstáculo. Esto permite que algunos valores que antes no se consideraban obstáculos ahora sí. Esto ocurre en los puntos en los que el haz de láser que había detectado ese punto pasa a una distancia menor que el radio del robot por algún punto de los obstáculos. Finalmente se actualizan los valores de las coordenados de los puntos de los obstáculos a la nuevas coordenadas hinchadas.

- *Move target*: El método recibe el *Oi* al que el robot necesita dirigirse y se calcula un nuevo punto llamado *waypoint*, que se encuentra cerca de este *Oi* pero desplazado una cierta distancia de seguridad del obstáculo. Esta distancia de seguridad se define mediante dos parámetros, uno para cada comportamiento. El *waypoint* y el *Oi* seleccionado se encuentran a la misma distancia del robot. Sólo se necesita comprobar hacia que lado se desplaza el *waypoint*, mirando en que dirección de desplazamiento no existe intersección con un obstáculo entre el robot y el *waypoint*.

- *Blocking Obs*: En el momento de filtrar los *Oi* en *Motion to Goal* se comprueba que no exista intersección entre los *Oi* y el objetivo. Esa intersección solo deberá afectar al comportamiento si se produce con un obstáculo al cual el *Oi* del que se comprueba la intersección no pertenece. En este método se utiliza el método *Intersection* en modo *Between*, para comprobar la intersección entre cada *Oi* y el objetivo con todos los puntos de obstáculos menos con los que pertenecen al mismo obstáculo que ese *Oi*.

- *PointisBetween*: Este método reciben tres puntos A, B y C. El método devuelve *True* si C se encuentra dentro del segmento de la recta que une A y B. Para ello se calcula el producto vectorial de (B-A) con (C-A), como aparece en la ecuación 3.15.

$$(B-A)\times(C-A) = (y_c - y_a) * (x_b - x_a) - (x_c - x_a) * (y_b - y_a) \tag{3.15}$$

  Si este es diferente de 0 es que C no forma parte de la recta AB. Si es igual a 0 ,se calcula el producto escalar y el módulo del vector AB (ver ecuación 3.16 y 3.17).

$$(B-A)\cdot(C-A) = (x_c - x_a) * (x_b - x_a) + (y_c - y_a) * (y_b - y_a) \tag{3.16}$$

$$|AB| = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2} \tag{3.17}$$

  Si se cumple la condición 3.18 el punto C esta situado entre A y B:

$$0 < (B-A)\cdot(C-A) < |AB| \tag{3.18}$$

- *Distance*: Un sencillo método que devuelve el módulo del vector que forman los dos puntos que recibe, como muestra la ecuación 3.19.

$$|AB| = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2} \tag{3.19}$$

Métodos de dibujo

Con los métodos expuestos hasta ahora se realiza la implementación del algoritmo *Tangent Bug*. Por motivos de depuración de errores y comprensión de la situación en cada instante del robot se utiliza un método para dibujar el *Local Tangent Graph*

- *Draw Surroundings*: Utilizando la librería *Turtle*, una herramienta de dibujo de *Python*, se genera un grafo con información relevante del entorno. Se dibujan lo siguientes elementos:

- Los puntos R en la posición del robot (siempre representada en el centro del grafo) y G que se dibuja en su posición si entra en el rango de visión o en la dirección en la que se encuentra.
- Los puntos de los rangos de detección del láser tanto hinchados como sin hinchar.
- Los puntos de los obstáculos, únicamente los hinchados, de color rojo.
- Los puntos *Oi*, de color rojo por defecto. Durante *Motion to Goal* aparecen en azul los *Oi* que pasan el filtrado y en verde el *Oi* que tiene menor heurística.
- Las distancias relevantes. En *Motion to Goal* aparecen todas las distancia heurísticas de los *Oi* que pasan el filtrado así como una línea del color correspondiente(azul o verde) que va del robot al *Oi* y después al objetivo. Si existe camino libre se dibuja una línea verde del robot al objetivo. Durante *Boundary Following* se representan los puntos M(en color azul) y L o T(en color verde) junto a las distancias *dmin* y *dleave* respectivamente.
- Se señala el modo en que se encuentra el robot en ese instante.

3.4.2 Parámetros

Los siguientes parámetros se definen al inicio del programa y pueden ser ajustados:

- *DEFAULT WALL DISTANCE* : Distancia a la que desplaza el punto *waypoint* del *Oi* durante *Boundary Following*.
- *DEFAULT OI DISTANCE*: Distancia a la que desplaza el punto *waypoint* del *Oi* durante *Motion to Goal*.
- *DEFAULT MAX V*: Velocidad lineal en m/s.
- *DEFAULT MAX W*: Regulador de velocidad angular, de 0 a 1. El valor que se atribuya a esta variable será multiplicado por *motion angle*, que es el ángulo que el robot debe recorrer para mirar hacia el objetivo, y ese valor será la velocidad angular aplicada, en rad/s.
- *DEFAULT RADIUS*: El radio del robot. Al ser el ATRV un robot rectangular se aproxima su radio a 0.5 m.
- *DEFAULT OBSTACLE JUMP*: Distancia a partir de la cual se reconocen obstáculos diferentes. Durante la detección es importante reconocer los diferentes obstáculos aunque se sitúen de forma continua. Si existe una diferencia entre los rangos de los obstáculos entre dos puntos seguidos mayor a este valor, se registran dos obstáculos diferentes.
- *DEFAULT DRAW MULTIPLIER*: Multiplicador aplicado a todas los puntos en el grafo realizado por *draw surroundings*. Este valor debe ser modificado en función del rango de visión que tenga el láser, para que la representación se ajuste de manera deseada.

3.5 Diagrama de flujo

El siguiente diagrama de flujo representa lo mismo que el de la figura 2.9, el funcionamiento del algoritmo. En este caso los pasos están marcados por los diferentes métodos de la implementación, para entender el flujo de ejecución del programa. Notar que en la condiciones *Mode* se comprueba el modo actual, que puede cambiar en los métodos *MtG distances* y *BF distances*.

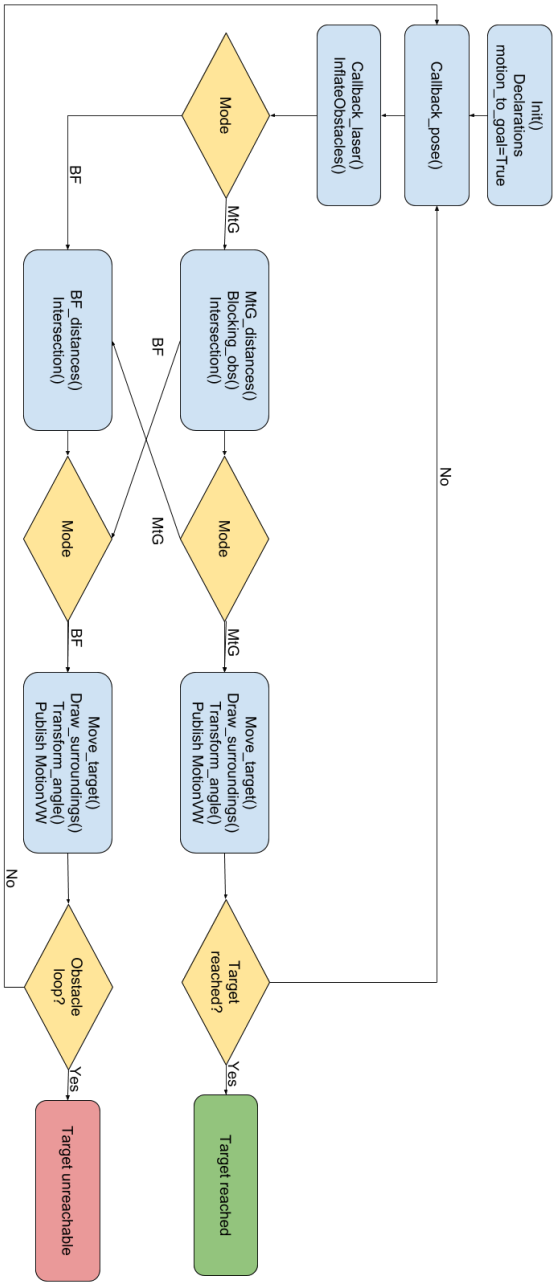

Figura 3.15: Diagrama de flujo de la implementación.

CAPÍTULO 4

RESULTADOS EXPERIMENTALES

Los resultados experimentales se han realizado en varios entornos creados con *Blender* y divididos entre simples y complejos. Para analizar los resultados obtenidos hemos creado un código de dibujo para representar los recorridos conseguidos.

4.1 *Draw Path*

Para analizar los resultados obtenidos se puede observar el movimiento del robot durante la ejecución, *Blender* muestra en tiempo real la simulación además del dibujo del *local tangent graph* que genera nuestro programa. Normalmente las ejecuciones suelen ser largas así que se implementó un código para generar un dibujo con*turtle* del camino recorrido una vez recopilada la información de toda a ejecución, *DrawPath.py*.

Durante la ejecución se escribe la información necesaria en tres ficheros diferentes:

- *Position.txt*: Se guardan todas las posiciones del robot cada vez que se obtienen. Se hace en *callbackLaser* a fin de reducir el tiempo de ejecución de *callbackPose*.

- *Obstacles.txt*: Para dibujar un mapa del entorno en necesario guardar sólo los puntos de los obstáculos detectados ya que si se guardaran todos los del láser, también se representarían los puntos de camino libre y sería un mapa imposible de interpretar. Se guardan los rangos de los obstáculos sin hinchar, a fin de representar fielmente el entorno. La escritura también se realiza *callbackLaser* pero únicamente se guarda una de cada 10 veces que se ejecuta el *callback*, de otra manera se guardarían demasiados puntos iguales y el tiempo de dibujado sería muy elevado Los puntos de obstáculo se representan en rojo.

- *Changes.txt*: Se guarda la posición del robot en los momentos que cambia de modo. Esto se hace para poder dibujar el camino recorrido de diferentes colores en función del modo en que se hizo cada tramo. En verde *motion to goal* y en azul *boundary following*.

Además se escriben unas leyendas relacionando los colores con su modo, se muestra el valor numérico de la distancia recorrida y se dibuja un círculo al inicio indicando el campo de visión del robot.

4.2 *Blender*

Para la generación de los entornos por los que se mueve el robot se utiliza el *software Blender*. La manera de crearlos ha sido mediante un objeto ya creado en las librerías de entornos de *MORSE*, un prisma rectangular, que es usado varias veces en cada entorno para crear los obstáculos. Además también se han utilizado 5 planos que delimitan en el entorno actuando como suelo y paredes.

Debido a que es una simulación que incorpora elementos de la realidad física como el rozamiento o la gravedad, para colocar los elementos es prudente dejar una mínima distancia entre ellos a fin de evitar que interactúen los obstáculos entre si. En 4.1, se ve como el entorno se forma a partir de varios objetos *blue box*, modificando su tamaño, posición y rotación.

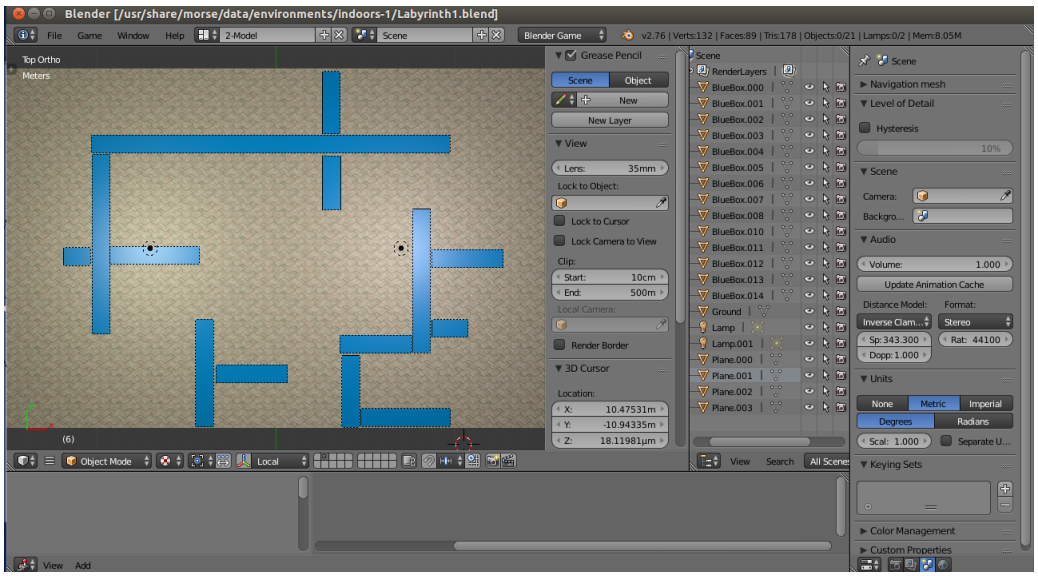

Figura 4.1: Ventana de *Blender* durante la creación del entorno complejo #1.

4.3 Entornos simples

Las primeras pruebas se realizaron en entornos más simples, donde el comportamiento del robot es predecible y se puede comprobar fácilmente si se ajusta a lo debido.

4.3.1 Entorno simple # 1

Como se observa en la figura 4.2, por la forma del obstáculo, el robot no necesitaría entrar en el modo *Boundary Following*.

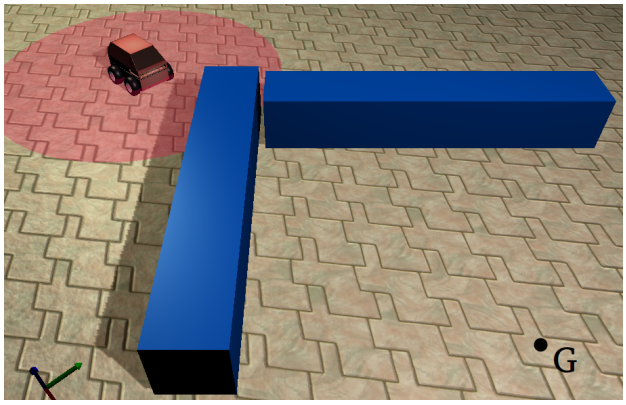

Figura 4.2: Aspecto del entorno simple # 1.

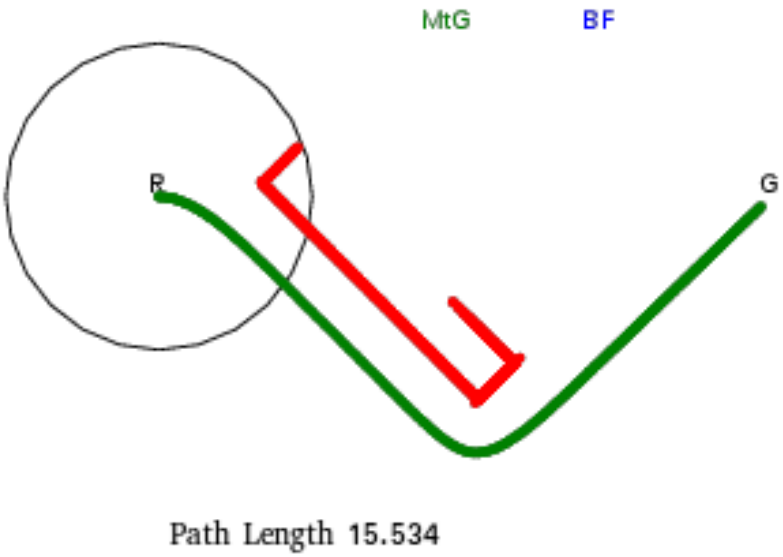

Figura 4.3: Camino recorrido por el robot en el entorno simple # 1 con un rango 3 m.

El algoritmo cumple las con lo esperado,como se observa en la figura 4.3, ya que no requiere del uso del comportamiento *Boundary Following* para completar el recorrido. En este primer entorno, el rango del sensor láser no afecta a la distancia recorrida por el robot y, por tanto, sólo se adjunta un ejemplo.

4.3.2 Entorno simple # 2

Si se invierte el sentido de la navegación en el entorno anterior(es decir, el punto objetivo G y la posición inicial del robot), tenemos las condiciones necesarias para que se deba entrar en el modo *Boundary Following*.El nuevo entorno aparece en la figura 4.4.

En este caso, sí que afectará el rango de visión del sensor láser al recorrido, tal y como muestra la figura 4.5.

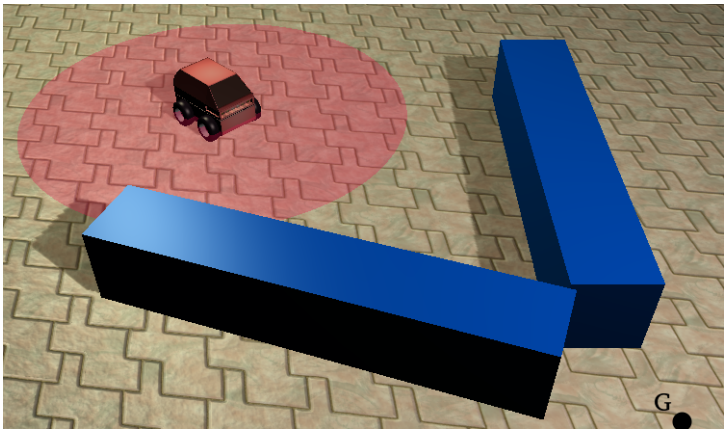

Figura 4.4: Aspecto del entorno simple # 2.

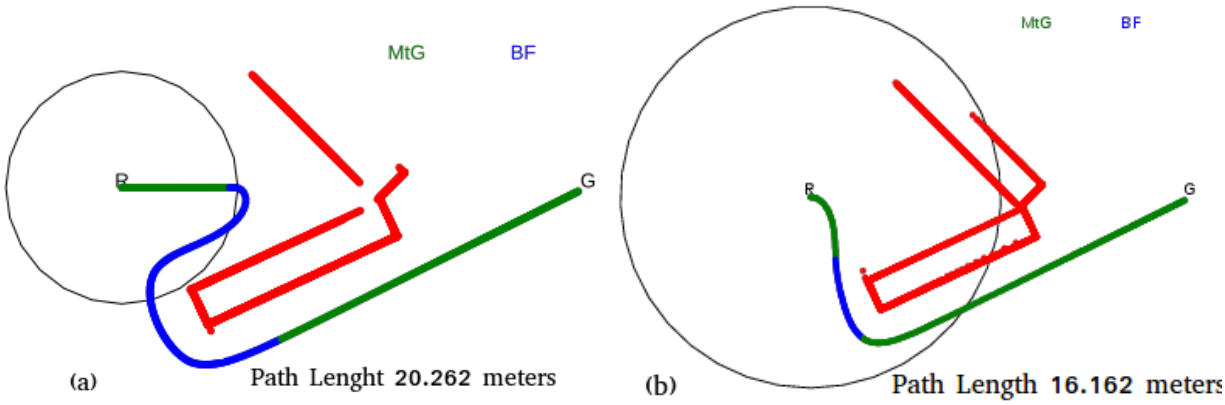

Figura 4.5: Caminos recorridos por el robot en el entorno simple # 2. Rangos de visión : (a) 3 m ,(b)6 m.

Como se observa en la figura 4.5, el algoritmo *Tangent Bug* cumple con la hipótesis planteada sobre la necesidad de usar el modo *Boundary Following*.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	20.262	-	-
6	16.162	4.1	20.23

Tabla 4.1: Resultados numéricos de las simulaciones en el entorno simple # 2.

En la tabla 4.1, vemos los rangos de visión y la distancia recorrida por el robot en cada ejecución. También se muestra la diferencia entre la distancia recorrida por el robot con rango de visión de 3 metros y con rango de visión de 6 metros.Esta diferencia se expresan en metros y en porcentaje sobre el valor de la distancia recorrida por el robot con rango de visión de 3 metros.

La tabla 4.1 demuestra que se consigue reducir notablemente la distancia recorrida con un mayor rango de visión.

4.3.3 Entorno simple # 3

Veamos otro entorno simple en el que será necesario usar los dos modos, *Motion to Goal* y *Boundary Following*. Este entorno, representado en la figura 4.6, es una versión más compleja del entorno # 2 y permitirá ver un recorrido más largo con cada uno de los comportamientos anteriormente mencionados.

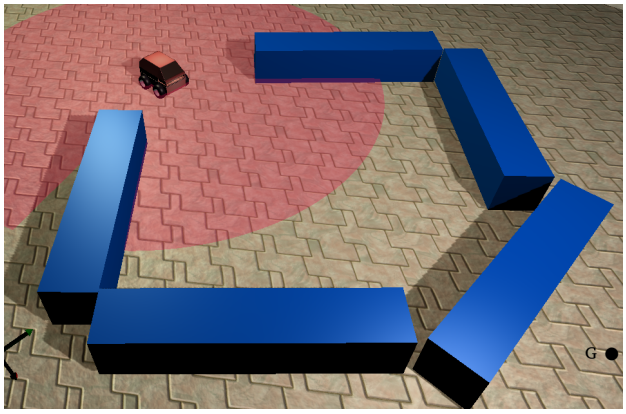

Figura 4.6: Aspecto del entorno simple # 3.

La figura 4.7 muestra los resultados obtenidos con diferentes rangos de visión.

En el caso de la figura 4.7 se observa con mejor claridad el camino recorrido por cada comportamiento : *Motion to Goal* y *Boundary Following*.Adicionalmente, se aprecian los cortes que se produce en las esquinas cuando se ejecuta *Boundary Following*; estos cortes se debe al uso del *Local Tangent Graph* durante este modo.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	36.68	-	-
6	28.32	8.36	22.79
9	23.18	13.5	36.80

Tabla 4.2: Resultados numéricos de las simulaciones en el entorno simple # 3.

Como se observa en la tabla 4.2, la reducción de la longitud del camino recorrido al aumentar el rango de visión es clara, aunque esa reducción no es lineal (ver figura 4.8).

Es importante notar que la función de la figura 4.8 no es extrapolable a otros entornos, ya que, aunque por norma general el aumento del rango de visión reduce la distancia recorrida, también afectan otros aspectos como son los cambios de modo y las formas de los obstáculos.

Además, es fácil ver que para cada entorno existe un rango de visión que reduce la distancia lo máximo posible y aumentarlo más que ese rango ya no afectará a la trayectoria del robot.

4.3.4 Entorno simple # 4

Este entorno ha sido elegido para verificar el correcto funcionamiento de nuestro algoritmo *Tangent Bug* en una situación como la de figura 4.9, en la que la implementación

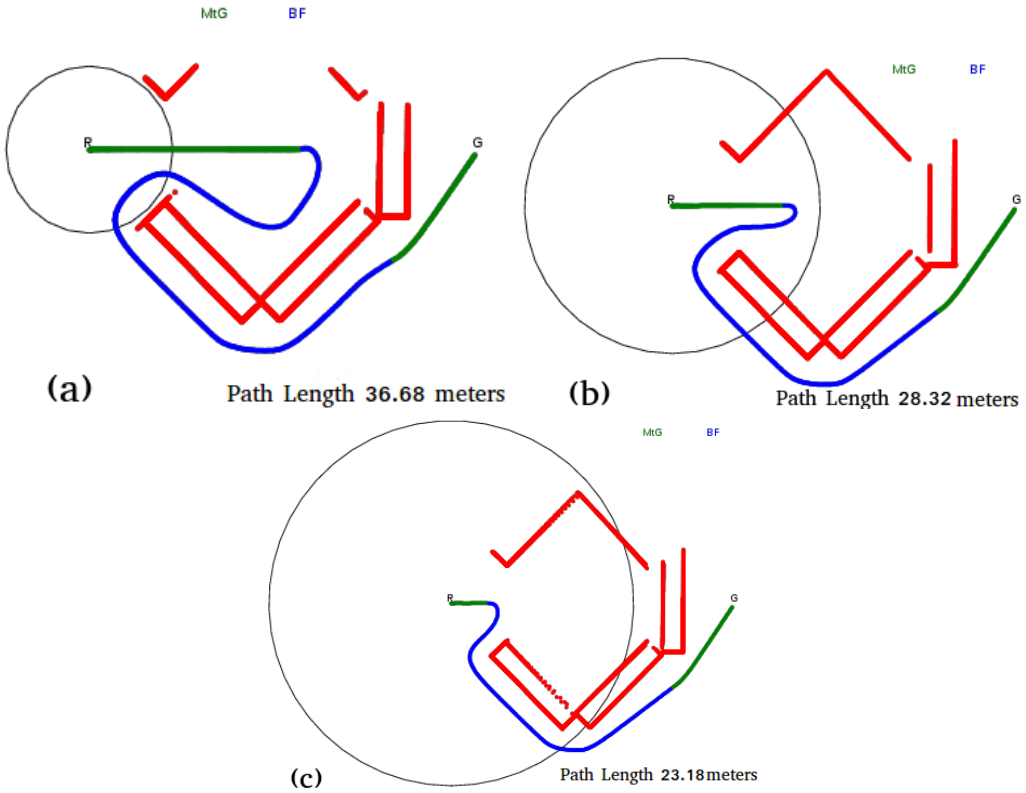

Figura 4.7: Caminos recorridos por el robot en el entorno simple # 3. Rangos de visión: (a) 3 m (b) 6 m (c) 9 m.

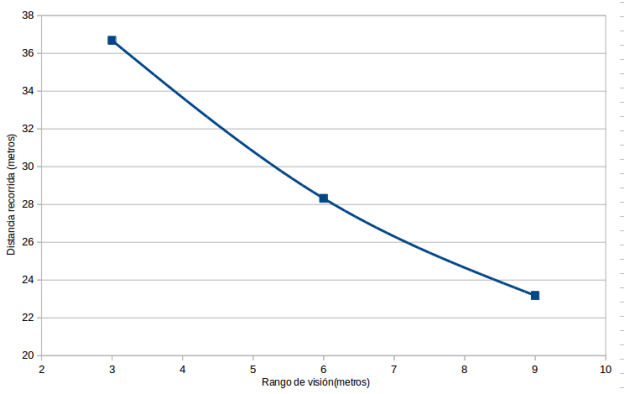

Figura 4.8: Representación gráfica de los resultados obtenidos en el entorno simple # 3.

realizada sobre por Baran Kahyaoglu en *JavaScript* fallabada (ver figura 3.1.b).

Como se observa en la figura 4.10, el robot alcanza de manera correcta el objetivo. Los fallos de la implementación en *JavaScript* venían por realizar un mal bloqueo de los *Oi*. En nuestra implementación, esta cuestión se gestiona de manera adecuada y evitando así estos fallos.

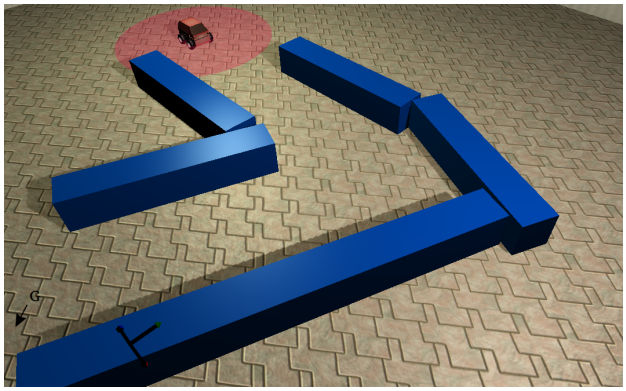

Figura 4.9: Entorno creado siguiendo el de la figura 3.1.b.

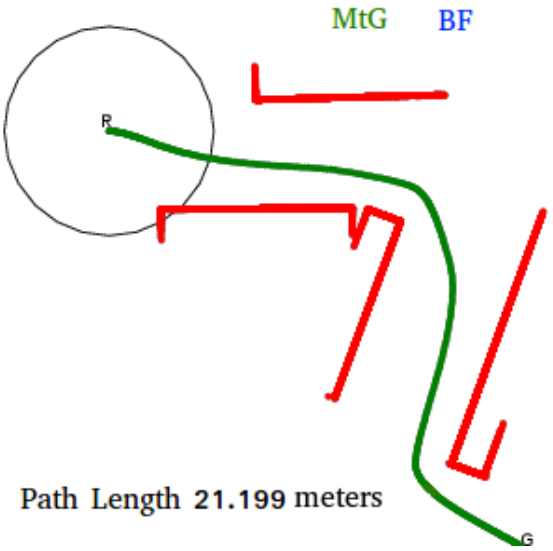

Figura 4.10: Camino recorrido por el robot en el entorno simple # 4 con rango de visión 3 m.

4.3.5 Entorno simple # 5

Este entorno ha sido elegido para comprobar el correcto funcionamiento de nuestro algoritmo *Tangent Bug* en un situación en la que la implementación realizada por Mehmet Mutlu en *MATLAB* (ver figura 3.3.b) generaba un camino largo al escoger erróneamente la dirección de seguimiento del contorno del obstáculos.

La figura 4.11, al igual que la figura 3.3.b, presentan un entorno muy similar y puntos de inicio y final aproximados. Con las mismas condiciones, nuestra implementación (ver figura 4.12) lleva al robot por un camino más corto al elegir la dirección de seguimiento del contorno del obstáculo de acuerdo con la dirección del *Oi* que fuerza el cambio de modo de *Motio to Goal* a *Boundary Following*, y no de manera arbitraria como ocurre en la implementación sobre *MATLAB*.

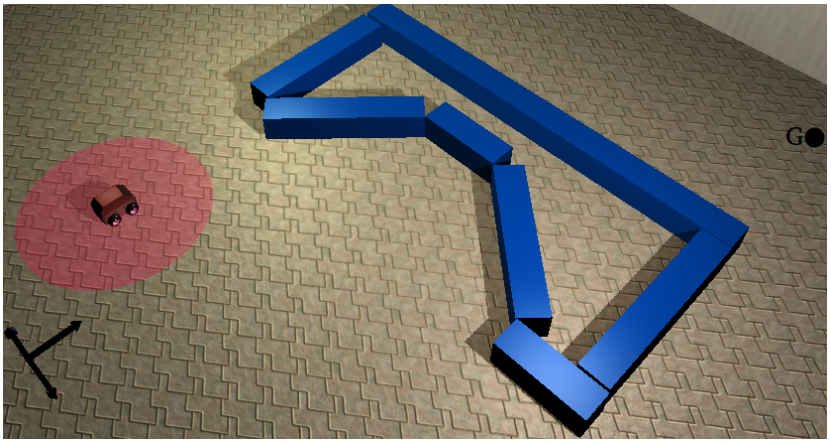

Figura 4.11: Entorno creado siguiendo el de la figura 3.3.b.

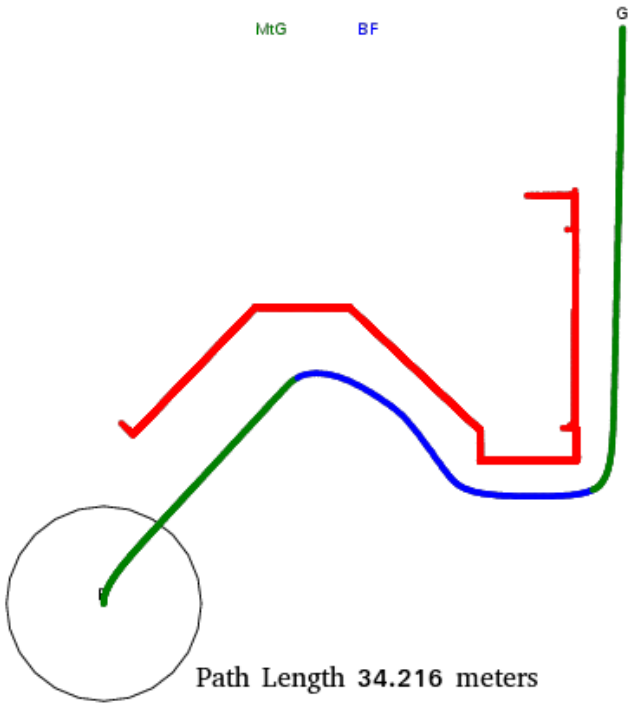

Figura 4.12: Contorno recorrido por el robot en el entorno simple # 5 con un rango de visión de 3 m.

4.3.6 Entorno simple # 6

En este entorno se busca una ejecución en la que el objetivo sea imposible de alcanzar.

En el entorno de la figura 4.13 se realizarán dos pruebas: una con el robot dentro del obstáculo y el objetivo fuera, y otra al revés.

En la figura 4.14 se ven dos casos en los que el robot no puede alcanzar el objetivo. Como se ha visto durante la implementación, el reconocimiento de que el objetivo es inalcanzable siempre ocurre durante el comportamiento *Boundary Following*, ya

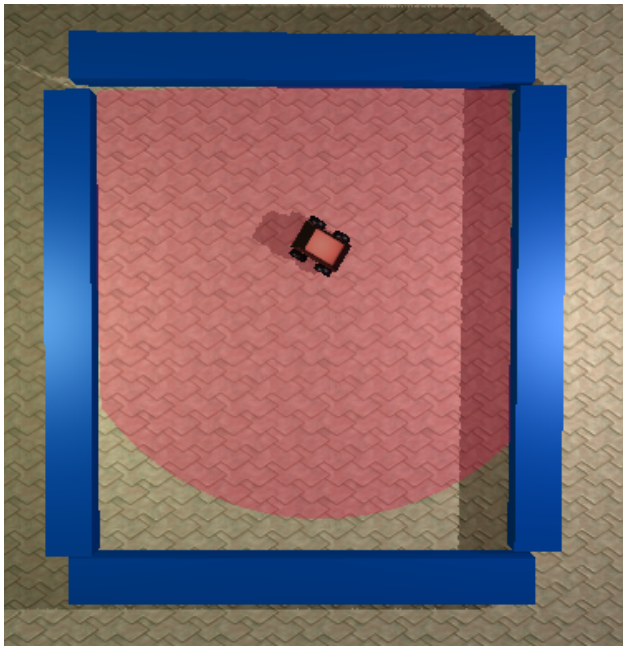

Figura 4.13: Aspecto del entorno simple 6.

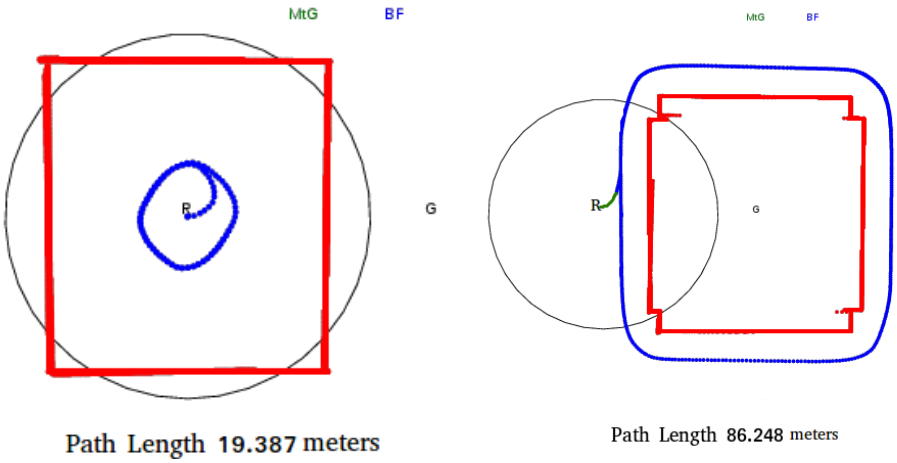

Figura 4.14: Resultados obtenido en el entorno simple # 6.

que de estar en el otro modo significaría que existe un camino que acerca el robot al objetivo.

Una vez se realizado el recorrido completo del contorno del obstáculo, el algoritmo detecta que el objetivo es inalcanzable y se acaba la ejecución.

4.4 Entornos complejos

Para la validación de nuestra implementación del algoritmo *Tangent Bug* en entornos más complejos, se han creado cinco modelos en *Blender*.

4.4.1 Entorno complejo # 1

El primer entorno complejo tiene un forma de laberinto, tal y como se observa en la figura 4.15.

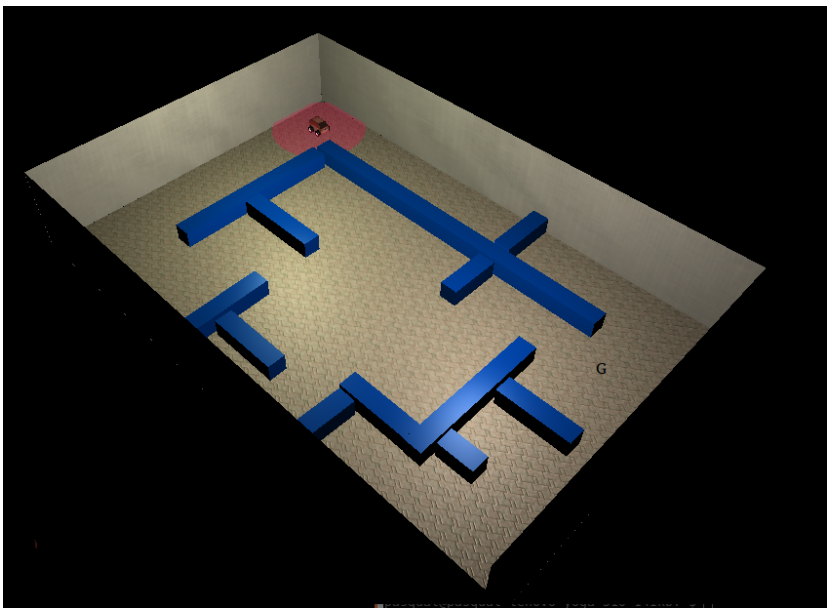

Figura 4.15: Aspecto del entorno complejo # 1.

Se observa en la figura 4.16 cómo el robot avanza durante un tramo por un pasillo sin salida bajo el comportamiento *Motion to Goal*. Esto se debe a que el robot no tiene manera de ver que ese pasillo no tiene salida hasta que se encuentra al final del mismo.

Cuando esto ocurre, se inicia el comportamiento *Boundary Following*,al no existir ningún *Oi* que cumpla las condiciones del filtrado. Este modo se mantendrá hasta que se cumpla la condición de cambio. En el tramo final bajo el comportamiento *Motion to Goal* siguen existiendo obstáculos que bloquean el camino al objetivo, pero en este caso no es necesario cambiar de modo para alcanzarlo.

La simulación de la figura 4.17 transcurre de manera parecida a la de la figura 4.16, pero se aprecia una mejora gracias al aumento del campo de visión del robot. El camino recorrido en el pasillo sin salida se acorta mucho, ya que se alcanza a ver su final antes.El último cambio de modo también se realiza antes ya que durante *Boundary Following* el robot necesita recorrer menos distancia del contorno que con un rango de visión de 3 metros, para cambiar al modo *Motion to Goal*.

Los resultados numéricos de la simulaciones aparecen en el cuadro 4.3.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	90.764	-	-
6	62.332	28.432	31.33

Tabla 4.3: Resultados numéricos de las simulaciones en el entorno complejo # 1.

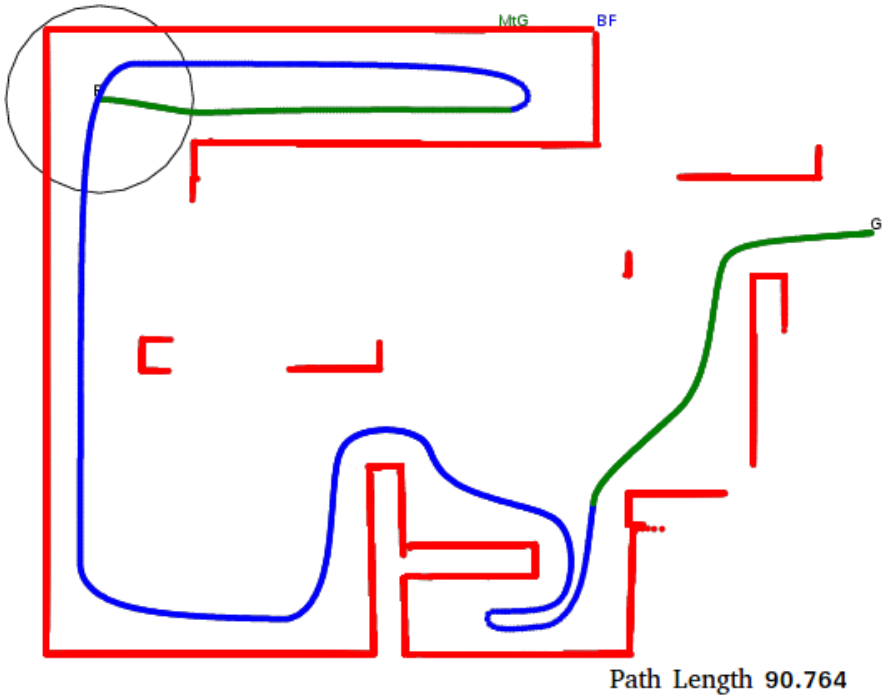

Figura 4.16: Camino recorrido por el robot en el entorno complejo # 1 un rango de visión de 3 m.

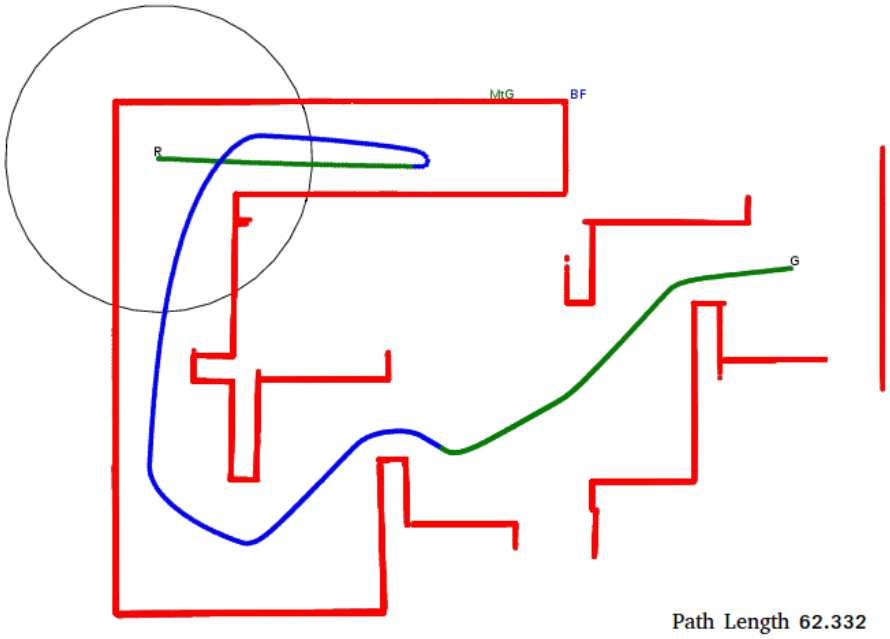

Figura 4.17: Camino recorrido por el robot sobre el entrono complejo # 1 con un rango visión de 6 metros.

4.4.2 Entorno complejo # 2

Este modelo recrea un entorno de tipo oficina (ver figura 4.18), con estancias y pasillos que conectan los espacios.

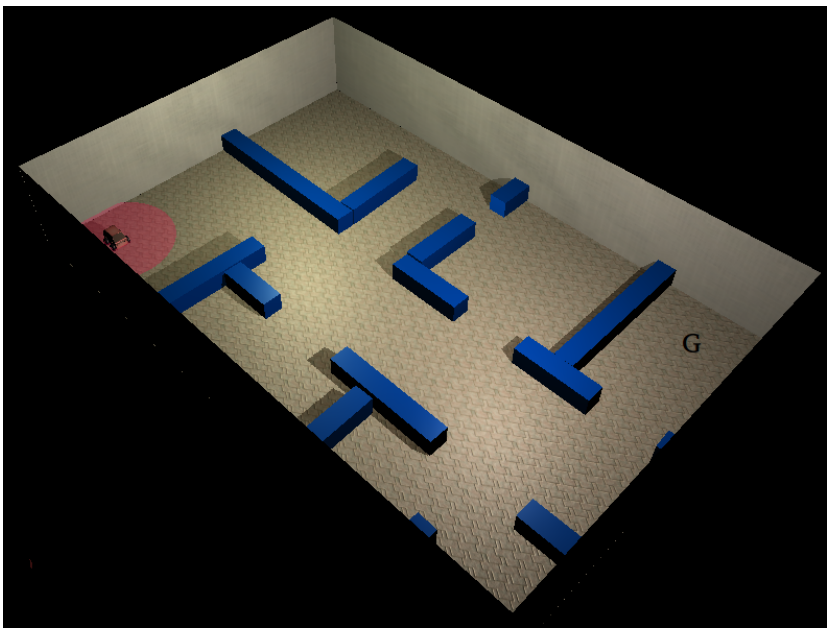

Figura 4.18: Aspecto del entorno complejo # 2.

Para este entorno se ha optado por definir dos situaciones de inicio y final diferentes.

Primera simulación

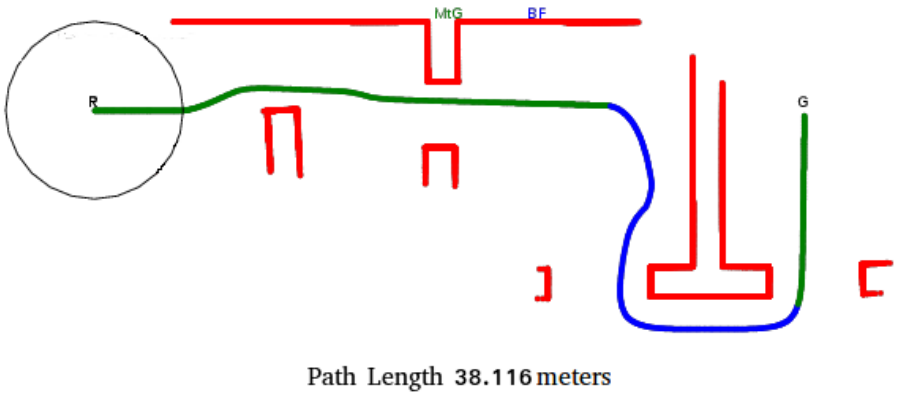

Figura 4.19: Camino recorrido por el robot en el entorno complejo # 2 durante la primera simulación con rango de visión 3.

En la figura 4.19 vemos como el robot es capaz de recorrer buena parte del camino en el modo *Motion to Goal*, evitando un obstáculo durante ese tramo. AL encontrar

una pared larga cuando está cerca del objetivo, el robot se ve obligado a activar el comportamiento *Boundary Following*. Durante la ejecución de este comportamiento, el robot toma la dirección correcta de giro, lo que hace que la distancia recorrida sea mucho menor de lo que sería si hubiese ido en la otra dirección. Finalmente, se vuelve a cambiar de modo cuando detecta camino libre hacia el objetivo.

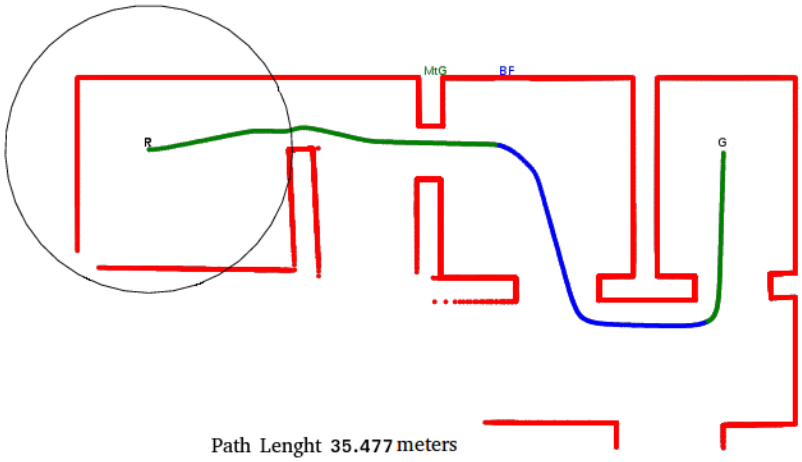

Figura 4.20: Camino recorrido en el entorno complejo # 2 durante la primera simulación 1 con un rango rango de visión 6m.

En el caso de la figura 4.20, el movimiento es muy similar al de la figura 4.19, pero el camino se acorta durante el tramo de *Boundary Bollowing* gracias a que, con un campo de visión mayor, se puede atajar más durante el seguimiento del contorno del obstáculo.

Los resultados de las ejecuciones anteriores se presentan en la tabla 4.4.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	38.116	-	-
6	35.477	2.639	6.92

Tabla 4.4: Resultados numéricos para la primera simulación en el entorno complejo # 2.

Los resultados se ciñen a lo esperado, aunque es destacable decir que, por las características del entorno y la selección hecha de los puntos de inicio y objetivo, la distancia recorrida con un rango de 3 metros ya es cercana a la distancia mínima posible entre dichos puntos. Aún así, se logra mejorar casi un 7 % duplicando el rango de visión.

Segunda simulación

En esta simulación, se eligen dos puntos de inicia y final los más separados posibles para intentar forzar una ejecución más complicada que en la primera simulación.

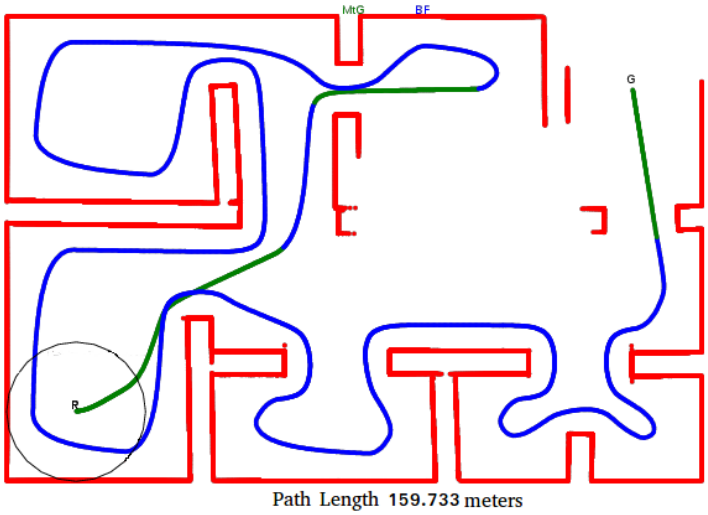

Figura 4.21: Camino recorrido por el robot en el entorno complejo # 2 durante la segunda simulación con un rango de visión de 3 m.

En el caso de la figura 4.21, el robot consigue acercarse a su objetivo mediante dos de tramos de *Motion to Goal* y uno de *Boundary Following*. En el último cambio de modo a *Boundary Following*, el robot elige seguir el contorno del obstáculo hacia la izquierda, no siendo esta dirección la más adecuada. Este error se produce debido a que el robot estaba separado del objetivo por un obstáculo ubicado perpendicularmente entre ellos y según la visión local del robot la dirección escogida para el seguimiento del contorno de obstáculo podría haber sido buena. Todo esto provoca que el robot siga gran parte del contorno del obstáculo hasta encontrar camino libre al objetivo. Se observan claramente los atajos que el algoritmo *Tangent Bug* es capaz de encontrar para no seguir las esquinas de las habitaciones.

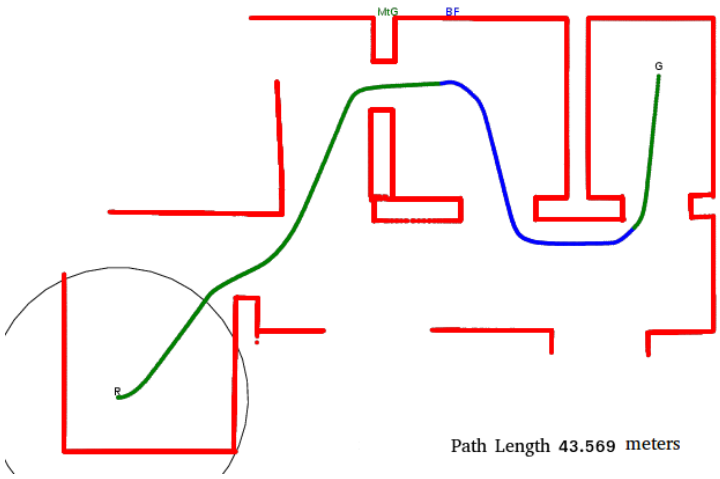

Figura 4.22: Camino recorrido por el robot en el entorno complejo # 2 durante la segunda simulación con un rango de visión de 6 m.

Como se observa en la figura 4.22, en este caso el camino resultante es muy eficiente. Con un tramo de *Motion to Goal* inicial el robot alcanza el último obstáculo que bloquea el objetivo. Ahora, debido a que su campo de visión es mayor que en el ejemplo anterior, el robot decide recorrer el contorno de ese último obstáculo hacia la derecha, siendo ésta una decisión mucho más acertada. Esto hace que el robot encuentre un camino libro rápidamente, evitando un recorrido mayor.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	50.835	-	-
6	43.569	7.266	14.29

Tabla 4.5: Resultados numéricos de las segunda simulación en el entorno complejo # 2.

Los resultados obtenidos para esta segunda simulación se resumen en la tabla 4.5. En este caso en concreto, la reducción del camino recorrido es muy significativa gracias a que se ha elegido correctamente la dirección de seguimiento del contorno del obstáculo en el cambio a *Boundary Following* con el rango de visión de 6 m. Aún si se hubiera tomado la dirección contraria, el camino recorrida se habría reducido gracias a los atajos que genera el *Local Tangent Graph* durante el comportamiento *Boundary Following*.

4.4.3 Entorno complejo # 3

En éste entorno también se imita la estructura de una oficina, pero esta vez las estancias y pasillos son más estrechos y alargados, como se observa en la figura 4.23.

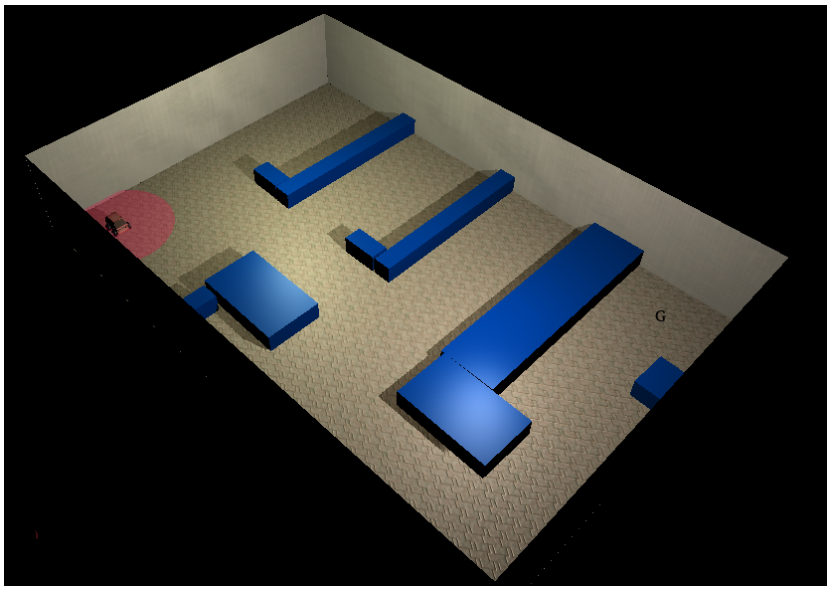

Figura 4.23: Aspecto del entorno complejo # 3.

En la figura 4.24 se observa como el comportamiento *Motion to Goal* permite al robot acercarse bastante al objetivo. También, en este modo, se aprecian muy claramente

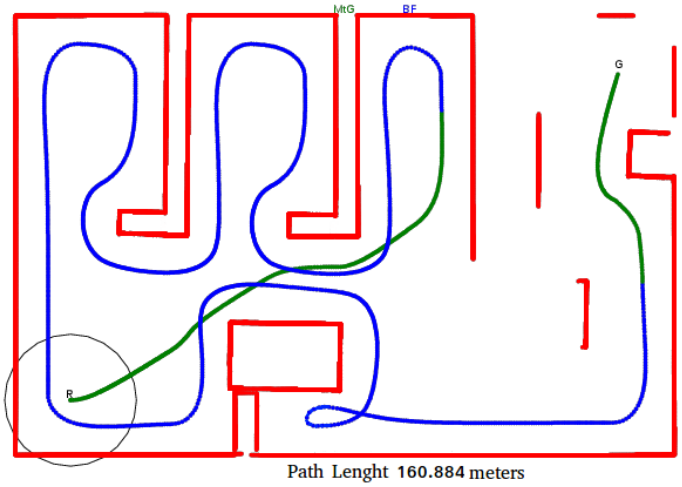

Figura 4.24: Camino recorrido por el robot en el entorno complejo # 3 con un rango de visión de 3 m.

diversos tramos en el los que el robot sigue el contorno de diferentes obstáculos. Al entrar en *Boundary Following*, el tipo de entorno con estancias alargadas, hace que el camino recorrido sea largo, pero el robot acaba encontrando un punto de obstáculo más próximo a G que aquel donde se había iniciado el comportamiento *Boundary Following*. Finalmente, el último obstáculo que bloquea el camino al objetivo es superado en modo *Motion to Goal*.

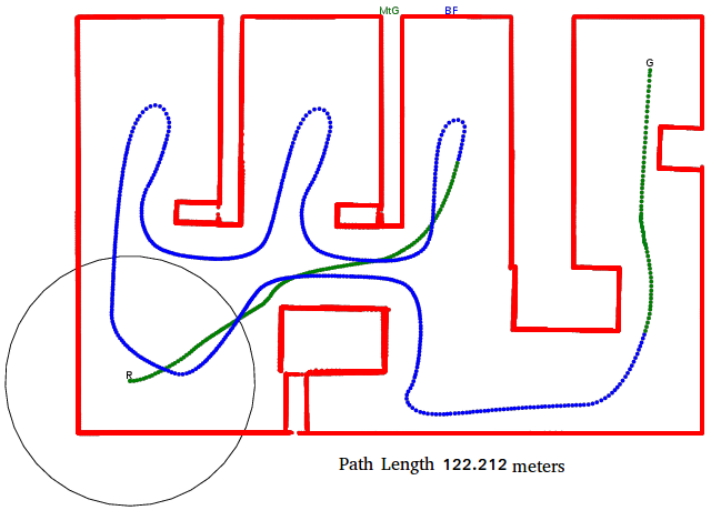

Figura 4.25: Camino recorrido por el robot en el entorno complejo # 3 con un rango de visión de 6 m.

En el caso de la figura 4.25 vemos como el inicio es muy similar, sólo varía en que con mayor rango de visión, el robot necesita seguir menos tiempo el obstáculo durante el comportamiento *Motion to Goal*, para darse cuenta de tiene que cambiar de modo. El tramo largo de *Boundary Following* sigue lo previsto. Notar que alguna porciones

de la trayectoria del robot aparecen punteadas. Esas porciones de la trayectoria se corresponden con situaciones donde el robot se ha movido a una velocidad elevada; fundamentalmente, en los giros de 180 grados realizados para salir de las habitaciones y cuando se ha encontrado un camino totalmente libre hacia el objetivo.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	160.884	-	-
6	122.212	38.672	23.07

Tabla 4.6: Resultados numéricos de las simulaciones en el entorno complejo # 3.

En la tabla 4.6 vemos los resultados numéricos. La mejora existe, aunque no es tan sustancial como la de la tabla 4.5, ya que ambas simulaciones han necesitado un seguimiento casi completo del entorno para alcanzar el objetivo.

4.4.4 Entorno complejo # 4

Este entorno intenta conseguir que el algoritmo *Tangent Bug* haga múltiples cambios de modo. El entorno creado con para este objetivo se observa en la figura 4.26.

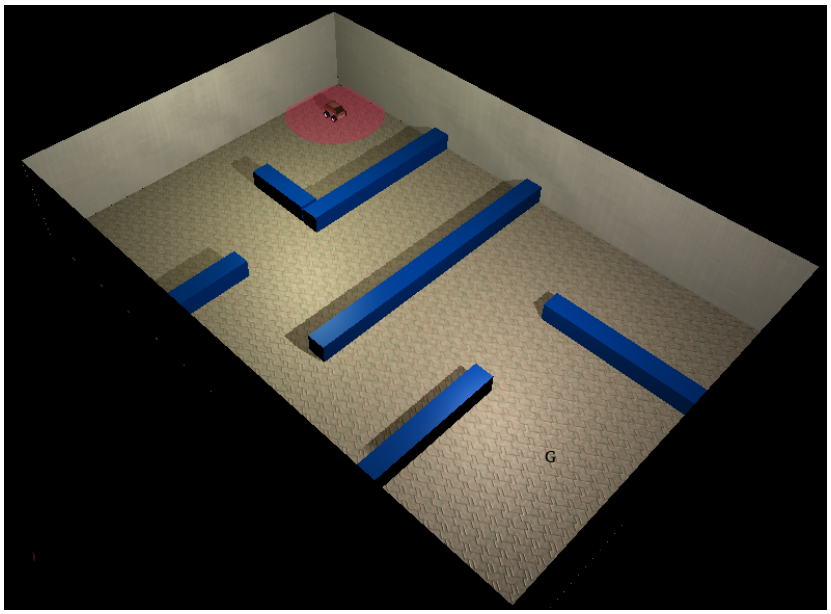

Figura 4.26: Entorno complejo # 4.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	48.869	-	-
6	42.328	6.541	13.38

Tabla 4.7: Resultados numéricos de las simulaciones en el entorno complejo # 4.

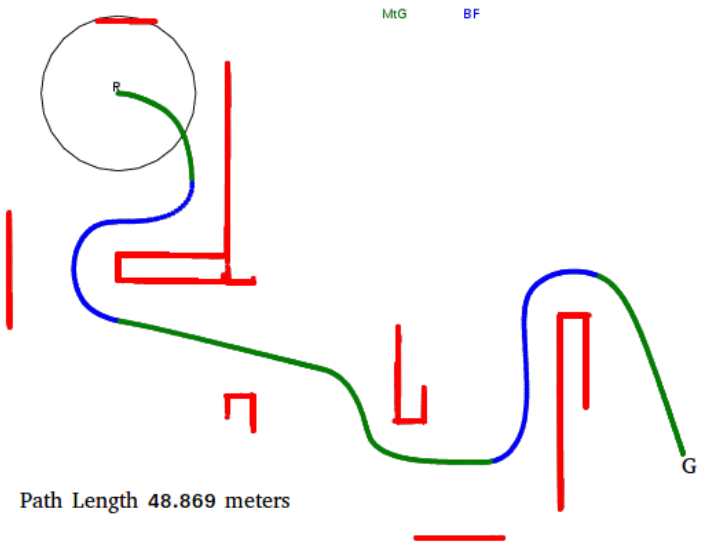

Figura 4.27: Camino recorrido por el robot en el entorno complejo # 4 con rango de visión de 3 m.

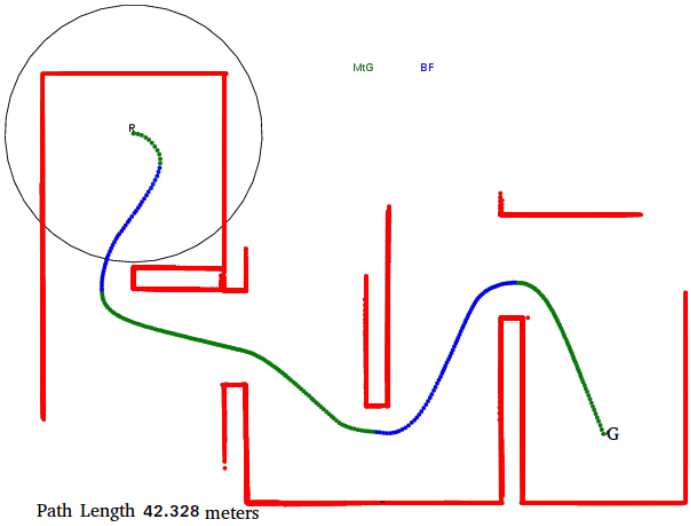

Figura 4.28: Camino recorrido por el robot en el entorno complejo # 4 con rango de visión de 6 m.

Como se observa en los resultados de las figuras 4.27 y 4.28, en ambos casos se utilizan tres tramos de *Motion to Goal* y dos de *Boundary Following* para alcanzar el objetivo. El utilizar un mayor rango de visión no acorta notablemente el camino recorrido como queda patente en la tabla 4.7.

4.4.5 Entorno complejo # 5

El entorno complejo # 5 está diseñado en forma de espiral. Por la forma del entorno, podemos predecir que casi todo el camino tendrá que ser recorrido bajo el compor-

tamiento *Boundary Following*, y que el rango de visión tendrá menor efecto en el resultado final.

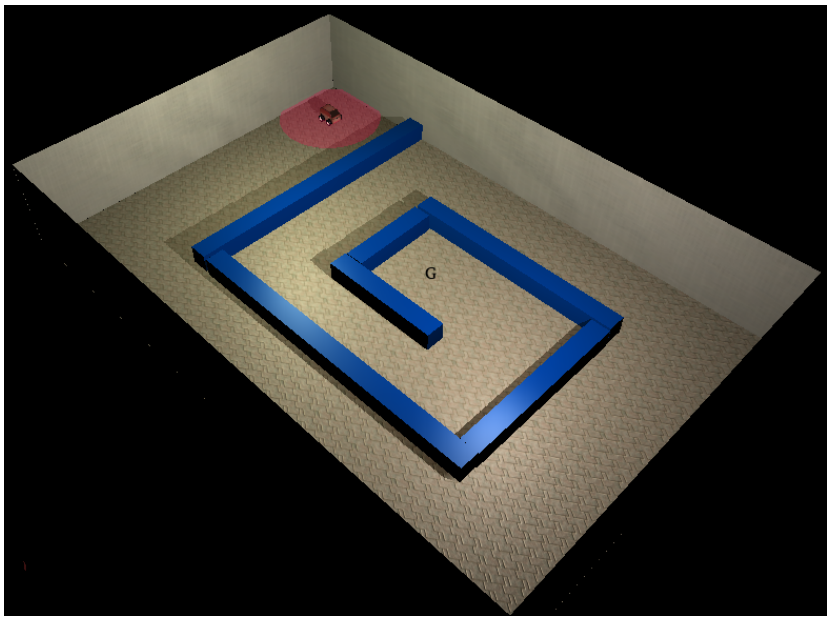

Figura 4.29: Entorno complejo # 5.

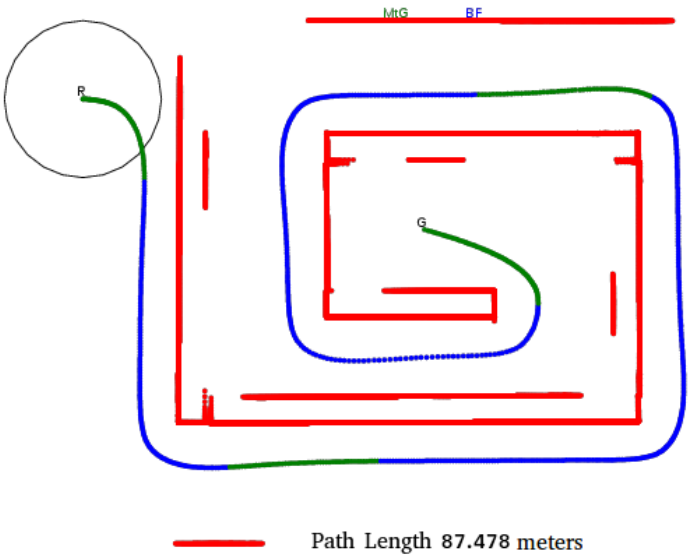

Figura 4.30: Camino recorrido por el robot en el entorno complejo # 5 con un rango de visión de 3 m.

Visto los resultados de las figuras 4.30 y 4.31 y de la tabla 4.8, se observa como el rango de visión en este caso tiene muy poca influencia sobre la trayectoria final. La poca diferencia entre las distancias recorridas ocurre al inicio, ya que, con menor rango de visión, el robot recorre más distancia en la primera activación del comportamiento *Motion to Goal*. Este ejemplo demuestra que hay casos en los que aumentar el rango de

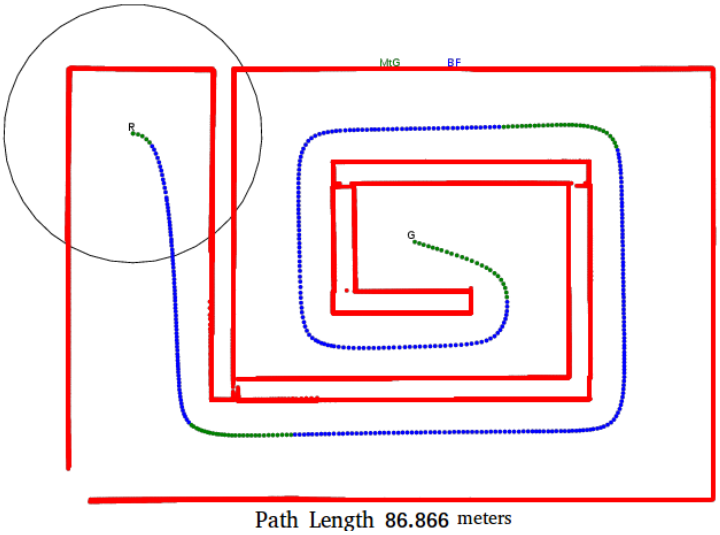

Figura 4.31: Camino recorrido por el robot en el entorno complejo # 5 con un rango de visión de 6 m.

Rango de visión (metros)	Distancia recorrida (metros)	Diferencia (metros)	%
3	87.478	-	-
6	86.866	0.612	0.7

Tabla 4.8: Resultados numéricos de las simulaciones en el entorno complejo # 5.

visión del robot no aporta mejoras, al menos significativas, con respecto a la distancia que recorre el robot hasta alcanzar su objetivo.

CAPÍTULO 5

CONCLUSIONES

Para concluir esta memoria analizaremos el trabajo realizado durante el desarrollo, fijándonos en los problemas de cada etapa y en cómo se podrían haber mejorado. También comentaremos los resultados obtenidos de manera general, complementado el análisis a fondo que se ha hecho de los resultados en el apartado anterior. Por último, expondremos posibles futuras ampliaciones del trabajo realizado.

5.1 Desarrollo

El propósito de este trabajo de final de grado ha sido implementar el algoritmo *Tangent Bug*. Para ello, el trabajo realizado se dividió en las siguientes tres fases:

- Estudio: En esta primera fase se recopiló toda la información posible sobre el algoritmo *Tangent Bug* para poder entender completamente su funcionamiento. Durante este análisis, se decidió utilizar el software *ROS/MORSE*, al no haber ninguna implementación pública en este entorno y porque la herramienta *MORSE* simula condiciones cercanas a la realidad. Esta primera etapa duró cerca de un 1 mes pero la investigación de material en la web fue constante durante todo el desarrollo.

- Implementación: La fase más larga del desarrollo, empezó con la instalación del *software ROS/MORSE*, además del sistema operativo *Ubuntu*. Al contrario de lo esperado, este paso no requirió mucho tiempo y pudimos iniciar la implementación del algoritmo poco después.

  Inicialmente, se programó el dibujo del *Local Tangent Graph*, al que posteriormente se añadieron diferentes elementos en función del modo. Cuando los dos modos de funcionamiento del algoritmo *Tangent Bug* estuvieron completados, se empezó con las pruebas de *debug*. Estas primera pruebas fueron las que mostraron la necesidad de inflar los obstáculos, desplazar los *waypoints*, filtrar los *Oi* y flexibilizar las condiciones.

Un problema importante que surgió durante la implementación del algoritmo *Tangent Bug* fue el de interpretar cómo debía operar el comportamiento *Boundary Following*. Nuestra primera implementación consistió en un controlador PD para el seguimiento de paredes [15]. El modo *Boundary Following* así funcionaba correctamente, pero más adelante nos dimos cuenta que no se ajustaba a la descripción original del algoritmo *Tangent Bug*. En el artículo que describa el algoritmo se comenta que en el modo *Boundary Following* se debe sacar provecho a los atajos generados por el *Local Tangent Graph* (ver figura 5.1). Esto nos llevó a la conclusión de que era necesario rehacer todo este modo, pero se pudo hacer sin mucha dificultad ya que se trataba de adaptar parte de la funcionalidad del comportamiento *Motion to Goal*.

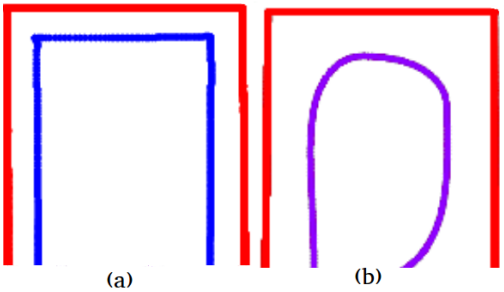

Figura 5.1: *Boundary following*:(a) sin atajos (b) con atajos.

En total la implementación del algoritmo duró 2 meses. Durante este tiempo se tuvo que aprender a usar *Python*, al ser la primera vez que se programaba con este lenguaje.

- *Test*: Una vez completada toda la implementación, se procedió a hacer pruebas con *MORSE* para validar el correcto funcionamiento. Para analizar los, resultados se escribió el programa *DrawPath.py* y se diseñaron entornos simple y complejos utilizando *Blender*. Durante esta fase, sólo se tuvieron que ajustar los parámetros del algoritmo para una óptima ejecución, como son las velocidades de movimiento del robot y las distancias de seguridad. Esta última fase del desarrollo necesitó aproximadamente 1 mes para completarse.

5.2 Resultados

Los resultados obtenidos corroboran que nuestra implementación del algoritmo *Tangnet Bug* es correcta. Las pruebas mostradas en este informe, tanto las simples como las complejas, muestran un comportamiento que se ajusta a la teoría del algoritmo. Como ya se ha explicado, las modificaciones del algoritmo aplicadas para poder simular en *MORSE*, por ser un simulador realista, no han cambiado los resultados esperados. El hecho de añadir filtros y bloqueos de algunos *Oi* hace que los recorridos siempre sean fluidos, sin movimientos bruscos ni giros indeseados.

Cabe decir que las simulaciones realizadas son muy largas en el tiempo, algunas de las que usan entornos complejos excediendo los 30 minutos. Esto se debe a la representación gráfica que se realiza en cada instante de la ejecución y a que *Python* es

un lenguaje de programación interpretado. Seguramente otro factor que ralentiza la ejecución sea el ordenador sobre el que se han hecho las pruebas, un portátil de gama media.

5.3 Posibles ampliaciones

A continuación se enumeran algunas posibles ampliaciones de este trabajo:

- *Reducir el tiempo de ejecución.* La implementación realizada tiene el inconveniente de que sus ejecuciones son muy largas en el tiempo. Si se usará un lenguaje no interpretado, cómo es *C++*, y se empleará una librería gráfica más eficiente, se podría reducir el tiempo de ejecución.

- *Aplicación sobre un robot real.* El paso lógico siguiente de este desarrollo sería probar el algoritmo *Tangent Bug* sobre un robot real. Teniendo en cuneta todas las modificaciones ya hechas al algoritmo para hacerlo funcionar sobre *MORSE* y el elevado realizamos de este simulador, pensamos que no será necesario un gran esfuerzo para hacer funcionar nuestro algoritmo *Tangent Bug* en un entorno real.

- *Creación de una web.* Uno de los ejemplos de implementaciones previas del algoritmo *Tangent Bug* es el creado por Baran Kahyaoglu. Esta implementación ya se ha analizado y visto sus errores, pero su presentación gráfica es muy clara e intuitiva. Pensamos que sería interesante convertir nuestra implementación del algoritmo *Tangent Bug* al lenguaje *JavaScript* para hacer más fácil su uso sin la necesidad de instalar *software* como *ROS, MORSE* y *Ubuntu.* Con un simple navegador, cualquier persona podría ejecutar el algoritmo sobre diferentes entornos.

APÉNDICE A

CÓDIGO *tbug.py*

Código principal del algoritmo.

```
# PASQUAL JOAN RIBOT LACOSTA 2018
# Tangent Bug algorithm for ROS/MORSE.
#!/usr/bin/env python3
import rospy
from std_msgs.msg import *
from sensor_msgs.msg import *
from geometry_msgs.msg import *
from tkinter import *
import turtle
import pymorse
import numpy
import math
import keyboard
import os
import copy
import time
#DECLARATIONS
DEFAULT_WALL_DISTANCE = 0.5# Distance from the wall
DEFAULT_OI_DISTANCE=0.5 #Distance form Oi is in MtG
DEFAULT_MAX_V= 0.5 # Maximum speed of robot m/s
DEFAULT_MAX_W=0.5 # Not rad/s, but from 0 to 1 as a regulator
DEFAULT_RADIUS=0.5 # Robot's radius
DEFAULT_OBSTACLE_JUMP=1 # Jump between obstacle ranges to recognise it
    as different obstacles
DEFAULT_DRAW_MULTIPLIER=50 # Multiplier to determine draw size.

#FILES FOR RECORDING
f= open("tbuglog.txt",'w') # log for relevant data
pos=open("positions.txt",'w')# Record of all positions
obs=open("obstacles.txt",'w')# Record of obstacles detected
changes=open("changes.txt",'w') #Record of mode changes
```

```
# pos, obs and changes are used in DrawPath.py
class tbug():


  motion = rospy.Publisher("/atrv/motion",Twist,queue_size=10)

  #Default declarations
  radius=DEFAULT_RADIUS

  #Callback declarations
  obstacle_ranges=[] #Ranges where an obstalce is detected
  ois=[]# List of indexes for the Oi points.
  laser_angles=[]#angle for each laser
  laser_x=[]#global coordinates for laser points
  laser_y=[]
  laser_ranges=[]#distance reading from laser
  laser_incr=0 #angle incremente between each laser
  infl_x=[]#inflated coordinates
  infl_y=[]
  laser_range_max=0 #laser maximum range
  total_lasers=0 #total number of laser
  yaw=0 #yaw rotation of robot
  current_position=Vector3(0,0,0) #robot's position
  current_orientation=Quaternion(0,0,0,0)#robot's rotation
  obs_behind=False
  laser_record=False

  #Motion to Goal declarations
  ois_heur=[] # List of Ois with valird heuristic distance
  waypoint_x=math.inf #waypoint result of Move_target()
  waypoint_y=math.inf
  minheur_old=math.inf# Oi with minimum heuristc last cycle
  minheur_indx=-1 #Index for that Oi
  heur_dist=[] # List of all heuristic distances
  cond_count=0 #counter for the MtG to BF condition
  D=0 # distance used in Intersection() with "Between" mode
  minheur_x=math.inf# Coordinates for Oi with minimum heuristic.
  minheur_y=math.inf
  arrived=False #Condition of target reached
  change_x=math.inf #Coordinates where MtG to BF occurs.
  change_y=math.inf
  obs_detected=False # Obstacle detected flag
  motion_to_goal=True #Motion to goal flag
  boundary_following=False#Boundary Following flag

  # Boundary Following declarations
  oiBF=-1 #Index for the followed Oi
  oiBF_x=math.inf #Coordinates for OiBF
  oiBF_y=math.inf
  loop_x=math.inf # Coordinates to check if a loop through an obstacle
      has been completed
  loop_y=math.inf
  leave_x=math.inf # L point coordinates
```

```
leave_y=math.inf
dleave=math.inf # dleave distance
min_x=math.inf # M point coordinates
min_y=math.inf
dmin=math.inf #dmin distance
freepathBF=False # check to distinguish when to use L or T
unreachable=False # unreachalbe condition
check_unreachable=False # flag to start checking the unreachable
    condition
check_loop=False #flag to check for a complete loop when checking the
    unreachalbe condition
BFcycles=0
def __init__(self):

  rospy.init_node("Listener")
  rospy.Subscriber("/atrv/laser", LaserScan,
      self.callback_laser,DEFAULT_OBSTACLE_JUMP)
  rospy.Subscriber("/atrv/pose", PoseStamped, self.callback_Pose)
  self.callbackDoneLaser = False
  self.callbackDonePose = False

def callback_Pose(self,pose):

  if self.callbackDonePose==False:
    self.current_position=pose.pose.position
    self.current_orientation=pose.pose.orientation
    self.callbackDonePose=True

def callback_laser(self,laser,jump):

  if self.callbackDoneLaser == False and self.callbackDonePose==True:
    #pose operations
    siny = 2*(self.current_orientation.w * self.current_orientation.z +
        self.current_orientation.x * self.current_orientation.y)
    cosy = 1-2*(self.current_orientation.y * self.current_orientation.y
        + self.current_orientation.z * self.current_orientation.z)
    self.yaw = math.atan2(siny,cosy)
    pos.write("{0}\n".format(self.current_position.x))
    pos.write("{0}\n".format(self.current_position.y))
    #list clearing
    self.laser_x.clear()
    self.laser_y.clear()
    self.infl_x.clear()
    self.infl_y.clear()
    self.laser_ranges.clear()
    self.laser_angles.clear()
    self.ois.clear()
    self.obstacle_ranges.clear()
    self.ois_heur.clear()
    self.heur_dist.clear()
    self.laser_record=self.laser_record+1
    self.D=math.sin(laser.angle_increment/2)*(laser.range_max*2)*2
    self.laser_range_max=laser.range_max
```

```
self.laser_incr=laser.angle_increment
self.total_lasers=len(laser.ranges)

#laser raw data treatment
for  i in range (0,len(laser.ranges)-1):
  angle=i*laser.angle_increment+laser.angle_min+self.yaw
  self.laser_x.append(math.cos(angle)*laser.ranges[i]
  +self.current_position.x)
  self.laser_y.append(math.sin(angle)*laser.ranges[i]
  +self.current_position.y)
  self.laser_angles.append(angle)
  self.laser_ranges.append(laser.ranges[i])

  if -jump<(laser.ranges[i+1]-laser.ranges[i])<jump and
  (laser.ranges[i]<laser.range_max):
    self.obstacle_ranges.append(i)
    if self.laser_record>=5:
      obs.write("{0}\n".format(self.laser_x[i]))
      obs.write("{0}\n".format(self.laser_y[i]))

if self.laser_record>=5:
  self.laser_record=0
angle=laser.angle_max+self.yaw
self.laser_x.append(math.cos(angle)*laser.ranges[len(laser.ranges)-1]
+self.current_position.x)
self.laser_y.append(math.sin(angle)*laser.ranges[len(laser.ranges)-1]
+self.current_position.y)
self.laser_angles.append(angle)
self.laser_ranges.append(laser.ranges[len(laser.ranges)-1])

#infalting obstacles and saving new obstacle ranges
tb.InflateObstacles()
self.obstacle_ranges.clear()
self.laser_ranges.clear()
for i in range (0,self.total_lasers):
    self.laser_ranges.append(tb.distance(self.current_position.x,
self.current_position.y,self.infl_x[i],self.infl_y[i]))

for i in range(0,len(self.laser_ranges)-1):
  if -jump<(self.laser_ranges[i+1]-self.laser_ranges[i])<jump and
      (self.laser_ranges[i]<laser.range_max*0.98):
    self.obstacle_ranges.append(i)

#changes to Oi points for when there's an obstacle behind the robot.
if(len(self.obstacle_ranges)!=0):
    self.obs_detected=True
    obs_x1=self.infl_x[self.obstacle_ranges[0]]
    obs_y1=self.infl_y[self.obstacle_ranges[0]]
    obs_x2=self.infl_x[self.obstacle_ranges[len(self.obstacle_ranges)-1]]
    obs_y2=self.infl_y[self.obstacle_ranges[len(self.obstacle_ranges)-1]]
    dist=tb.distance(obs_x1,obs_y1,obs_x2,obs_y2)
    if dist>self.D:
      self.obs_behind=False
```

```
        else:
          self.obs_behind=True
        if self.obs_behind==False:
          self.ois.append(self.obstacle_ranges[0])
        j=1
        while j<len(self.obstacle_ranges)-1:
          if(self.obstacle_ranges[j+1]-self.obstacle_ranges[j]!=1):
            self.ois.append(self.obstacle_ranges[j])
            self.ois.append(self.obstacle_ranges[j+1])
            j=j+2
          else:
            j=j+1
        if self.obs_behind==False:
          self.ois.append(self.obstacle_ranges[len(self.obstacle_ranges)-1])
        if self.obs_behind==True and len(self.ois)!=0:
          self.ois_old=copy.copy(self.ois)
          self.ois[0]=self.ois_old[len(self.ois_old)-1]
          for i in range (1,len(self.ois)):
            self.ois[i]=self.ois_old[i-1]
      else:
        self.obs_behind=False
        self.obs_detected=False
      self.callbackDoneLaser = True

  def MotiontoGoal(self,goal):

    if self.callbackDoneLaser==True and self.callbackDonePose==True:
      #check conditions and calculate distances
      tb.MtG_distances(goal)
      if self.motion_to_goal==True:
        tb.Move_target(self.minheur_indx,goal)
        turtle.clearscreen()
        tb.draw_surroundings(goal,DEFAULT_DRAW_MULTIPLIER)
        # message to publish
        msg = Twist()
        msg.linear.x=DEFAULT_MAX_V
        motion_angle=tb.transform_angle(self.waypoint_x,self.waypoint_y)
        msg.angular.z=motion_angle*DEFAULT_MAX_W

        #check reached condition
        if(goal.x-0.2<=self.current_position.x<=goal.x+0.2 and
            goal.y-0.2<=self.current_position.y<=goal.y+0.2):
          msg.angular.z=0
          msg.linear.x=0
          for i in range (0,5):
            self.motion.publish(msg)
          self.motion_to_goal=False
          self.arrived=True
        else:
          self.motion.publish(msg)
          self.callbackDonePose = False
          self.callbackDoneLaser = False
```

```
def MtG_distances(self,goal):

  minheur=math.inf
  p1x=self.current_position.x
  p1y=self.current_position.y
  p2x=goal.x
  p2y=goal.y
  blocking_obs_MtG=False
  dist_block=math.inf
  oi_blocked=math.inf
  oi_alt=math.inf
  alt_ok=False
  wp_inter=False
  #check for free path
  for i in range (0,len(self.obstacle_ranges)):
    blocking_obs_MtG=tb.Intersection(p1x,p1y,p2x,p2y,self.infl_x[self.obstacle_ranges[i]],
    self.infl_y[self.obstacle_ranges[i]],"Between")
    if blocking_obs_MtG==True:
      break
  if blocking_obs_MtG==True:
    d_old=math.inf
    #set up blocking Oi in case its need it (to avoid zig-zag)
    if self.minheur_indx>=0:
      for i in range (0,len(self.ois)):
        d_new=tb.distance(self.infl_x[self.ois[i]],self.infl_y[self.ois[i]],
        self.minheur_x,self.minheur_y)
        if d_new<d_old:
          d_old=d_new
          oi_alt=i
      if oi_alt % 2==0:
        oi_blocked=oi_alt+1
      else:
        oi_blocked=oi_alt-1
      oialt_x=self.infl_x[self.ois[oi_alt]]
      oialt_y=self.infl_y[self.ois[oi_alt]]
      dist_block=tb.distance(oialt_x,oialt_y,self.infl_x[self.ois[oi_blocked]],
      self.infl_y[self.ois[oi_blocked]])

    # Calculating heuristic distances
    for i in range (0,len(self.ois)):

      dx_oi=tb.distance(self.current_position.x,self.current_position.y,
      self.infl_x[self.ois[i]],self.infl_y[self.ois[i]])
      doi_goal=tb.distance(self.infl_x[self.ois[i]],
      self.infl_y[self.ois[i]],goal.x,goal.y)
      dx_goal=tb.distance(self.current_position.x,
      self.current_position.y,goal.x,goal.y)
      heur=round(dx_oi+doi_goal,3)

      tb.Move_target(i,goal)
      for j in range(0,len(self.obstacle_ranges)-1):
        wpg_x=self.waypoint_x+self.current_position.x
        wpg_y=self.waypoint_y+self.current_position.y
```

```
      wp_inter=tb.Intersection(p1x,p1y,wpg_x,wpg_y,
            self.infl_x[self.obstacle_ranges[j]],
            self.infl_y[self.obstacle_ranges[j]],"Between")
      if wp_inter==True:
        break

    #Oi filtering
    if doi_goal<dx_goal and tb.blocking_obs(i,p2x,p2y)==False and
        wp_inter==False:

      if heur<=minheur:
        minheur=heur
        self.heur_dist.append(heur)
        self.minheur_indx=i
        self.ois_heur.append(self.ois[i])
        self.minheur_x=self.infl_x[self.ois[i]]
        self.minheur_y=self.infl_y[self.ois[i]]
        if i==oi_alt:
          alt_ok=True
      else:
        if i==oi_alt:
          alt_ok=True
        self.ois_heur.append(self.ois[i])
        self.heur_dist.append(heur)
#Check for Oi blocking
if self.minheur_indx==oi_blocked and dist_block>0.3:
  for i in range (0,len(self.ois_heur)-1):
    if self.ois_heur[i]==oi_alt:
      alt_ok=True
      break
  if alt_ok==True:
    self.minheur_indx=oi_alt
    self.minheur_x=self.infl_x[self.ois[oi_alt]]
    self.minheur_y=self.infl_y[self.ois[oi_alt]]
    dx_oi=tb.distance(self.current_position.x,self.current_position.y,
    self.infl_x[self.ois[oi_alt]],self.infl_y[self.ois[oi_alt]])
    doi_goal=tb.distance(self.infl_x[self.ois[oi_alt]],
    self.infl_y[self.ois[oi_alt]],goal.x,goal.y)
    minheur=round(dx_oi+doi_goal,3)
  else:
    #Oi to change to when blocking is no valid
    self.minheur_old=0
    self.cond_count=6

# If no Oi goes through filtering, MtG->BF
if len(self.ois_heur)==0:
  self.minheur_old=0
  self.cond_count=6

# MtG->BF condition with counter
if minheur>self.minheur_old:
  self.cond_count=self.cond_count+1
  if self.cond_count>5:
```

```
        # starting M,L,dmin and dleave
        self.dmin=math.inf
        for i in range (0,len(self.obstacle_ranges)):
          dmin_prov=tb.distance(self.infl_x[self.obstacle_ranges[i]],
          self.infl_y[self.obstacle_ranges[i]],goal.x,goal.y)
          if dmin_prov<self.dmin:
            self.dmin=dmin_prov
            self.min_x=self.infl_x[self.obstacle_ranges[i]]
            self.min_y=self.infl_y[self.obstacle_ranges[i]]
        self.leave_x=self.min_x
        self.leave_y=self.min_y
        self.dleave=self.dmin
        # If a change happens whit no Oi currently
        #valid, we take the last one that was valid.
        if self.minheur_indx<0 or len(self.ois_heur)==0:
          dist=math.inf
          for i in range(0,len(self.ois)-1):
            dist_new=tb.distance(self.current_position.x,
            self.current_position.y,self.infl_x[self.ois[i]],self.infl_x[self.ois[i]])
            if dist_new<dist:
              dist=dist_new
              self.oiBF_x=self.infl_x[self.ois[i]]
              self.oiBF_y=self.infl_y[self.ois[i]]
              self.oiBF=i
        else:
          self.oiBF_x=self.infl_x[self.ois[self.minheur_indx]]
          self.oiBF_y=self.infl_y[self.ois[self.minheur_indx]]
          self.oiBF=self.minheur_indx
        f.write("minheur'{0}'\r\n".format(minheur))
        f.write("minheur old'{0}'\r\n".format(self.minheur_old))
        f.write(" MtG->BF\r\n")
        changes.write("{0}\n".format(self.current_position.x))
        changes.write("{0}\n".format(self.current_position.y))
        self.cond_count=0
        self.motion_to_goal=False
        self.callbackDonePose = False
        self.callbackDoneLaser = False
        self.change_x=self.current_position.x
        self.change_y=self.current_position.y
        self.boundary_following=True
      else:
        # continue in MtG
        self.minheur_old=minheur
        self.cond_count=0
    else:
      #straight to goal
      self.minheur_indx=-1

  def BoundaryFollowing(self,goal):

    if self.callbackDoneLaser==True and self.callbackDonePose==True:
      #check for conditions and get Oi to follow
      tb.BF_distances(goal)
```

```
    if self.boundary_following==True:
      tb.Move_target(self.oiBF,goal)
      turtle.clearscreen()
      tb.draw_surroundings(goal,DEFAULT_DRAW_MULTIPLIER)
      motion_angle=tb.transform_angle(self.waypoint_x,self.waypoint_y)
      #message to publish
      msg = Twist()
      msg.linear.x=DEFAULT_MAX_V
      msg.angular.z=motion_angle*DEFAULT_MAX_W
      #To check unreachable, first we get to a distance from the change
          point and then we get another point.
      #After seperating a distance again from that point, we can start
          checking if its unreachalbe.
      dist_change=tb.distance(self.current_position.x,self.current_position.y,
      self.change_x,self.change_y)
      if dist_change>1 and self.check_loop==False:
        self.check_loop=True
        self.loop_x=self.current_position.x
        self.loop_y=self.current_position.y
      dist_loop=tb.distance(self.current_position.x,self.current_position.y
      ,self.loop_x,self.loop_y)
      if self.check_loop==True and dist_loop>1:
        self.check_unreachable=True
      if self.check_unreachable==True and dist_loop<0.1:
        msg.linear.x=0
        msg.angular.z=0
        for i in range (0,5):
          self.motion.publish(msg)
        self.boundary_following=False
        self.unreachable=True
      else:
        self.motion.publish(msg)
        self.callbackDoneLaser=False
        self.callbackDonePose=False

def BF_distances(self,goal):

  p1x=self.current_position.x
  p1y=self.current_position.y
  p2x=goal.x
  p2y=goal.y
  blocking_obs_BF=False
  self.dleave=math.inf
  d_old=math.inf
  old_oiBF=self.oiBF
  for i in range (0,len(self.ois)):
    #the new OiBF will be the closest to the last one. We block a
        change of OiBF when the current and last Oi belong to the same
        obstacle.
    d_new=tb.distance(self.infl_x[self.ois[i]],self.infl_y[self.ois[i]],
    self.oiBF_x,self.oiBF_y)
    if d_new<d_old:
      self.oiBF=i
```

```
    d_old=d_new
if old_oiBF % 2==0 and self.oiBF==old_oiBF+1:
  self.oiBF=old_oiBF
elif old_oiBF % 2 !=0 and self.oiBF==old_oiBF-1:
  self.oiBF=old_oiBF
self.oiBF_x=self.infl_x[self.ois[self.oiBF]]
self.oiBF_y=self.infl_y[self.ois[self.oiBF]]

#Calculatin dleave.
for i in range (0,len(self.obstacle_ranges)):
  blocking_obs_BF=tb.Intersection(p1x,p1y,p2x,p2y,self.infl_x[self.obstacle_ranges[i]],
  self.infl_y[self.obstacle_ranges[i]],"Between")
  if blocking_obs_BF==True:
    break
if blocking_obs_BF==False:
  vx=p2x-p1x
  vy=p2y-p1y
  ux=vx/math.sqrt(vx**2+vy**2)
  uy=vy/math.sqrt(vx**2+vy**2)
  self.leave_x=self.current_position.x+(ux*self.laser_range_max)
  self.leave_y=self.current_position.y+(uy*self.laser_range_max)
  self.dleave=tb.distance(self.leave_x,self.leave_y,goal.x,goal.y)
  self.freepathBF=True

else:
  self.freepathBF=False
  if self.oiBF %2 ==0:
    closest_oi=self.oiBF
  else:
    closest_oi=self.oiBF-1
  for i in range (0,len(self.obstacle_ranges)):
    if closest_oi==0 and self.obs_behind:
      if self.ois[0]<self.obstacle_ranges[i]<self.total_lasers-1 or
          0<self.obstacle_ranges[i]<self.ois[1]:
        dleave_prov=tb.distance(self.infl_x[self.obstacle_ranges[i]],
        self.infl_y[self.obstacle_ranges[i]],goal.x,goal.y)
        if dleave_prov<self.dleave:
          self.dleave=dleave_prov
          self.leave_x=self.infl_x[self.obstacle_ranges[i]]
          self.leave_y=self.infl_y[self.obstacle_ranges[i]]
    else:
      if self.ois[closest_oi]<self.obstacle_ranges[i]
      <self.ois[closest_oi+1]:
        dleave_prov=tb.distance(self.infl_x[self.obstacle_ranges[i]],
        self.infl_y[self.obstacle_ranges[i]],goal.x,goal.y)
        if dleave_prov<self.dleave:
          self.dleave=dleave_prov
          self.leave_x=self.infl_x[self.obstacle_ranges[i]]
          self.leave_y=self.infl_y[self.obstacle_ranges[i]]

#BF->MtG condition
if self.dleave+self.radius<self.dmin:
  f.write("dmin'{0}'\r\n".format(self.dmin))
```

```
    f.write("dleave'{0}'\r\n".format(self.dleave))
    f.write("BF->MtG \r\n")
    changes.write("{0}\n".format(self.current_position.x))
    changes.write("{0}\n".format(self.current_position.y))
    self.minheur_indx=-1
    self.minheur_x=math.inf
    self.minheur_y=math.inf
    self.minheur_old=math.inf
    self.check_unreachable=False
    self.check_loop=False
    self.boundary_following=False
    self.motion_to_goal=True
    self.callbackDonePose = False
    self.callbackDoneLaser = False

def transform_angle(self,p1x,p1y):
  #transforming angle so we can know how much the robot has to turn to
      point to the desired point.
  target_angle=math.atan2(p1y,p1x)
  if(math.radians(90)>target_angle>=math.radians(0) or
      math.radians(180)>target_angle>=math.radians(90)):
    if(self.yaw>0):
      motion_angle=target_angle-self.yaw
    else:
      if(math.radians(180)-math.fabs(self.yaw)<target_angle):
        motion_angle=-(math.radians(180)-math.fabs(self.yaw))-(math.radians(180)-target_angle)
      else:
        motion_angle=target_angle+math.fabs(self.yaw)
  else:
    if(self.yaw>0):
      if(self.yaw>math.radians(180)-math.fabs(target_angle)):
        motion_angle=(math.radians(180)-self.yaw)+(math.radians(180)-math.fabs(target_angle))
      else:
        motion_angle=-(self.yaw+math.fabs(target_angle))
    else:
      if(self.yaw<target_angle):
        motion_angle=math.fabs(self.yaw)-math.fabs(target_angle)
      else:
        motion_angle=-(math.fabs(target_angle)-math.fabs(self.yaw))
  return motion_angle

def draw_surroundings(self,goal,mult):
    #Robot's position
    turtle.ht()
    turtle.up()
    turtle.pencolor("black")
    turtle.title("Robot's surroundings")
    turtle.speed(0)
    turtle.tracer(0,0)
    turtle.setposition(0,0)
    turtle.write("R",align="center",font=("Arial", 13, "normal"))
    turtle.dot()
```

```
#Goal's position

goal_rel=Vector3(goal.x-self.current_position.x,goal.y-self.current_position.y,0)
dx_goal=tb.distance(goal.x,goal.y,self.current_position.x,self.current_position.y)

if(self.laser_range_max>dx_goal):
  G=Vector3(goal_rel.x*mult,goal_rel.y*mult,0)
else:
  G=Vector3((goal_rel.x/dx_goal)*self.laser_range_max*(mult+5)
  ,(goal_rel.y/dx_goal)*self.laser_range_max*(mult+5),0)

turtle.setposition(G.x,G.y)
turtle.pencolor("green")
turtle.write("G",align="center",font=("Arial", 15, "normal"))
turtle.up()
# Mode
turtle.pencolor("black")
turtle.setposition(self.laser_range_max*mult,self.laser_range_max*mult)
if self.motion_to_goal==True:
  turtle.write("Motion to Goal",align="center",font=("Arial", 16,
      "normal"))
else:
  turtle.write("Boundary Following",align="center",font=("Arial",
      16, "normal"))
# All ranges
turtle.pencolor("black")
turtle.setposition((self.laser_x[0]-self.current_position.x)*mult,
(self.laser_y[0]-self.current_position.y)*mult)
turtle.down()
for i in range(1,len(self.laser_x)):
  turtle.setposition((self.laser_x[i]-self.current_position.x)*mult
  ,(self.laser_y[i]-self.current_position.y)*mult)

turtle.up()
#Inflated ranges
turtle.setposition((self.infl_x[0]-self.current_position.x)*mult
,(self.infl_y[0]-self.current_position.y)*mult)
turtle.down()
for i in range(1,len(self.infl_x)):
  turtle.setposition((self.infl_x[i]-self.current_position.x)*mult
  ,(self.infl_y[i]-self.current_position.y)*mult)
turtle.up()

if self.obs_detected==True:

  #Obstacle ranges
  turtle.pencolor("red")
  turtle.pensize(3)
  turtle.up()
  h=1
  px=self.infl_x[self.obstacle_ranges[0]]-self.current_position.x
  py=self.infl_y[self.obstacle_ranges[0]]-self.current_position.y
  turtle.setposition(px*mult,py*mult)
```

```
    turtle.down()

    while h <len(self.obstacle_ranges)-1:
      if self.obstacle_ranges[h]-self.obstacle_ranges[h-1]==1:
        px=self.infl_x[self.obstacle_ranges[h]]-self.current_position.x
        py=self.infl_y[self.obstacle_ranges[h]]-self.current_position.y
        turtle.setposition(px*mult,py*mult)
        h=h+1
      else:
        px=self.infl_x[self.obstacle_ranges[h]]-self.current_position.x
        py=self.infl_y[self.obstacle_ranges[h]]-self.current_position.y
        turtle.up()
        turtle.setposition(px*mult,py*mult)
        turtle.down()
        h=h+1
    turtle.up()

  #Oi labels

    turtle.pensize(2)
    k=0
    for x in range (0,len(self.ois)):
      if k < len(self.ois_heur):
        if self.ois[x]==self.ois_heur[k]:
          k=k+1
          if self.ois[x]==self.ois[self.minheur_indx]:
            turtle.pencolor("green")
          else:
            turtle.pencolor("blue")
        else:
          turtle.pencolor("red")
      else:
        turtle.pencolor("red")
      if self.minheur_indx<0 or self.boundary_following==True:
        turtle.pencolor("red")
      turtle.up()
      px=(self.infl_x[self.ois[x]])-self.current_position.x
      py=(self.infl_y[self.ois[x]])-self.current_position.y
      turtle.setposition(px*mult,py*mult)
      turtle.down()
      turtle.write("O",align="right",font=("Arial", 17, "normal"))
      turtle.write(x,align="left",font=("Arial", 14, "normal"))
      turtle.up()

  if self.motion_to_goal==True:

    if self.minheur_indx<0:
      #line straight to the goal when free
      turtle.pencolor("green")
      turtle.setposition(0,0)
      turtle.down()
      turtle.setposition(G.x,G.y)
    else:
```

```
        # x at the current waypoint
        px=self.waypoint_x
        py=self.waypoint_y
        turtle.setposition(px*mult,py*mult)
        turtle.pencolor("black")
        turtle.write("x",align="center",font=("Arial", 15, "normal"))

      #Heuristic distances
        for j in range(0,len(self.ois_heur)):
          if self.ois_heur[j]==self.ois[self.minheur_indx]:
            turtle.pencolor("green")
          else:
            turtle.pencolor("blue")
          turtle.up()
          turtle.setposition(0,0)
          turtle.down()
          px=self.infl_x[self.ois_heur[j]]-self.current_position.x
          py=self.infl_y[self.ois_heur[j]]-self.current_position.y
          turtle.setposition(px*mult,py*mult)
          turtle.up()
          turtle.setposition((px+1)*mult,py*mult)
          turtle.write(self.heur_dist[j],align="right",font=("Arial",
              11, "normal"))
          turtle.setposition(px*mult,py*mult)
          turtle.down()
          turtle.setposition(G.x,G.y)
    else:
      # M and dmin
      px=self.waypoint_x
      py=self.waypoint_y
      turtle.setposition(px*mult,py*mult)
      turtle.pencolor("black")
      turtle.write("x",align="center",font=("Arial", 15, "normal"))
      turtle.up()
      turtle.pencolor("blue")
      turtle.pensize(4)
      dx_m=tb.distance(self.min_x,self.min_y,
      self.current_position.x,self.current_position.y)
      if dx_m>self.laser_range_max:
        px=((self.min_x-self.current_position.x)/dx_m)*self.laser_range_max*(mult+5)
        py=((self.min_y-self.current_position.y)/dx_m)*self.laser_range_max*(mult+5)
      else:
        px=(self.min_x-self.current_position.x)*mult
        py=(self.min_y-self.current_position.y)*mult
      turtle.setposition(px,py)
      turtle.down()
      turtle.dot()
      turtle.write("M",align="center",font=("Arial", 15, "normal"))
      turtle.up()
      turtle.setposition(px+(3*mult),py)
      turtle.write(round(self.dmin,3),align="right",font=("Arial", 11,
          "normal"))
      # L/T and dleave
```

```
      turtle.up()
      turtle.pencolor("green")
      px=self.leave_x-self.current_position.x
      py=self.leave_y-self.current_position.y
      turtle.setposition(px*mult,py*mult)
      turtle.down()
      turtle.dot()
      if self.freepathBF:
        turtle.write("T",align="center",font=("Arial", 13, "normal"))
      else:
        turtle.write("L",align="center",font=("Arial", 13, "normal"))
      self.freepathBF=False
      turtle.up()
      turtle.setposition((px+3)*mult,py*mult)
      turtle.write(round(self.dleave,3),align="right",font=("Arial",
          11, "normal"))

    turtle.pensize(2)
    turtle.pencolor("black")
    turtle.up()
    turtle.setposition(0,0)

    turtle.update()

def Intersection(self,p1x,p1y,p2x,p2y,cx,cy,mode):
  # True if there's an intersection between the line P1P2 and a circle
      with center at C.
  #y=mx+d
  m=(p2y-p1y)/(p2x-p1x)
  d=p1y-m*p1x
  if m==0:
    m_per=math.inf
  else:
    m_per=-(1/m)
  d_per=cy-m_per*cx
  interx=(d_per-d)/(m-m_per)
  intery=interx*m+d
  dist=tb.distance(cx,cy,interx,intery)

  if mode=="Between":
    #Cirle radius is D and also check is the intersection is on the
        line segement P1P2
    if dist<=self.D and
        tb.PointisBetween(p1x,p1y,p2x,p2y,interx,intery):
      return True
    else:
      return False

  elif mode=="Simple":
    #circle radius is R
    if dist<=self.radius:
      return True
    else:
```

```
      return False
    else:
      return False


  def InflateObstacles(self):
    #Inflating obstacles to the robot's radius to avoid collision.
    p1x=self.current_position.x
    p1y=self.current_position.y
    for i in range (0,self.total_lasers):

      mindist=math.inf
      p2x=self.laser_x[i]
      p2y=self.laser_y[i]
      minx=p2x
      miny=p2y
      for j in range (0,len(self.obstacle_ranges)):
        cx=self.laser_x[self.obstacle_ranges[j]]
        cy=self.laser_y[self.obstacle_ranges[j]]
        if
            self.radius>tb.distance(self.current_position.x,self.current_position.y,cx,cy):
          f.write("ERROR, DENTRO DEL HINCHADO")
        if tb.Intersection(p1x,p1y,p2x,p2y,cx,cy,"Simple"):
          #y=mx+d
          m=(p2y-p1y)/(p2x-p1x)
          d=p1y-m*p1x
          #(x-a)*(x-a)+(y-b)*(y-b)=R*R
          a=cx
          b=cy
          ro=(self.radius**2)*(1+m**2)-(b-m*a-d)**2
          if ro>=0:
            xinter1=(a+b*m-d*m+math.sqrt(ro))/(1+m**2)
            xinter2=(a+b*m-d*m-math.sqrt(ro))/(1+m**2)
            yinter1=m*xinter1+d
            yinter2=m*xinter2+d
            dist1=tb.distance(self.current_position.x,self.current_position.y,
            xinter1,yinter1)
            dist2=tb.distance(self.current_position.x,self.current_position.y,
            xinter2,yinter2)
            Pmax_x=(math.cos(self.laser_angles[i])*self.laser_range_max)
            +self.current_position.x
            Pmax_y=(math.sin(self.laser_angles[i])*self.laser_range_max)
            +self.current_position.y
            inter1ok=tb.PointisBetween(self.current_position.x,self.current_position.y,
            Pmax_x,Pmax_y,xinter1,yinter1)
            inter2ok= tb.PointisBetween(self.current_position.x,
            self.current_position.y,Pmax_x,Pmax_y,xinter2,yinter2)
            if dist1<=dist2:
              if dist1<=mindist and inter1ok:
                mindist=dist1
                minx=xinter1
                miny=yinter1
            else:
```

```
            if dist2<=mindist and inter2ok:
              mindist=dist2
              minx=xinter2
              miny=yinter2
    self.infl_x.append(minx)
    self.infl_y.append(miny)

def Move_target(self,i,goal):
  # Move current target to a new waypoint in order to not entering
      inflated ranges.
  if i<0:
    self.waypoint_x=goal.x-self.current_position.x
    self.waypoint_y=goal.y-self.current_position.y
  else:
    way1=False
    way2=False
    if self.motion_to_goal:
      ang=DEFAULT_OI_DISTANCE/(self.laser_ranges[self.ois[i]])
    else:
      ang=DEFAULT_WALL_DISTANCE/(self.laser_ranges[self.ois[i]])
    if ang>1:
      ang=1
    elif ang<-1:
      ang=-1
    alpha=2*math.asin(ang)
    lasers=round(alpha/self.laser_incr)

    if self.ois[i]-lasers<=0:
      waypoint1_x=(math.cos(self.laser_angles[0])*self.laser_ranges[self.ois[i]])
      waypoint1_y=(math.sin(self.laser_angles[0])*self.laser_ranges[self.ois[i]])
    else:
      waypoint1_x=(math.cos(self.laser_angles[self.ois[i]-
      lasers])*self.laser_ranges[self.ois[i]])
      waypoint1_y=(math.sin(self.laser_angles[self.ois[i]-
      lasers])*self.laser_ranges[self.ois[i]])

    if self.ois[i]+lasers>=len(self.laser_angles)-1:
      waypoint2_x=(math.cos(self.laser_angles[len(self.laser_angles)-1])
      *self.laser_ranges[self.ois[i]])
      waypoint2_y=(math.sin(self.laser_angles[len(self.laser_angles)-1])
      *self.laser_ranges[self.ois[i]])
    else:
      waypoint2_x=(math.cos(self.laser_angles[self.ois[i]
      +lasers])*self.laser_ranges[self.ois[i]])
      waypoint2_y=(math.sin(self.laser_angles[self.ois[i]
      +lasers])*self.laser_ranges[self.ois[i]])

    p1x=self.current_position.x
    p1y=self.current_position.y
    p2x=waypoint1_x+self.current_position.x
    p2y=waypoint1_y+self.current_position.y
    p3x=waypoint2_x+self.current_position.x
    p3y=waypoint2_y+self.current_position.y
```

```
    for j in range (0,len(self.obstacle_ranges)-1):
      inter1=tb.Intersection(p1x,p1y,p2x,p2y,self.infl_x[self.obstacle_ranges[j]],
      self.infl_y[self.obstacle_ranges[j]],"Between")
      inter2=tb.Intersection(p1x,p1y,p3x,p3y,self.infl_x[self.obstacle_ranges[j]],
      self.infl_y[self.obstacle_ranges[j]],"Between")
      if inter1==True:
        self.waypoint_x=waypoint2_x
        self.waypoint_y=waypoint2_y
        break
      elif inter2==True:
        self.waypoint_x=waypoint1_x
        self.waypoint_y=waypoint1_y
        break

  def blocking_obs (self,i,p2x,p2y):
    #check if theres a blocking obstacle between Oi and G, not counting
       the Oi's obstacle
    p1x=self.infl_x[self.ois[i]]
    p1y=self.infl_y[self.ois[i]]
    if len(self.ois)>2:
      if self.obs_behind and (i==0 or i==1):
        for j in range (0,len(self.obstacle_ranges)-1):
          if (self.obstacle_ranges[j]<self.ois[0] and
              self.obstacle_ranges[j]>self.ois[1]):
            blocking_obs=tb.Intersection(p1x,p1y,p2x,p2y,
            self.infl_x[self.obstacle_ranges[j]],
            self.infl_y[self.obstacle_ranges[j]],"Between")
            if blocking_obs==True:
              return True
        return False
      else:
        for j in range (0,len(self.obstacle_ranges)-1):
          if i % 2==0 and (self.obstacle_ranges[j]<self.ois[i] or
              self.obstacle_ranges[j]>self.ois[i+1]):
            blocking_obs=tb.Intersection(p1x,p1y,p2x,p2y,
            self.infl_x[self.obstacle_ranges[j]],
            self.infl_y[self.obstacle_ranges[j]],"Between")
            if blocking_obs==True:
              return True

          elif i % 2!=0 and (self.obstacle_ranges[j]<self.ois[i-1] or
              self.obstacle_ranges[j]>self.ois[i]):
            blocking_obs=tb.Intersection(p1x,p1y,p2x,p2y,
            self.infl_x[self.obstacle_ranges[j]],
            self.infl_y[self.obstacle_ranges[j]],"Between")
            if blocking_obs==True:
              return True
        return False
    else:
      return False

  def PointisBetween(self,ax,ay,bx,by,cx,cy):
```

```
    # True if C is between A and B
    crossproduct=(cy-ay)*(bx-ax)-(cx-ax)*(by-ay)
    if abs(round(crossproduct,3))!=0:
      return False

    dotproduct =(cx-ax)*(bx-ax)+(cy-ay)*(by-ay)
    if round(dotproduct,3)<= 0:
      return False

    squaredlengthba = (bx-ax)*(bx-ax)+(by-ay)*(by-ay)
    if dotproduct >= squaredlengthba:
      return False
    return True


  def distance(self,ax,ay,bx,by):
    # distance between two points
    dx = bx-ax
    dy = by-ay
    return math.sqrt(dx**2 + dy**2)

if __name__ == '__main__':
  try:
    tb = tbug()
    goal=Vector3(12.5,7.5,0)
    #Main loop
    while True:
      if tb.motion_to_goal==True:
        tb.MotiontoGoal(goal)
      elif tb.boundary_following==True:
        tb.BoundaryFollowing(goal)
      else:
        if tb.arrived==True:
          f.write("Arrived")
          break
        elif tb.unreachable==True:
          f.write("Goal is unreachable")
          break
        else:
          break
    f.close()
    changes.close()
    pos.close()
    obs.close()
    rospy.spin()
  except rospy.ROSInterruptException: pass
```

APÉNDICE B

CÓDIGO *default.py*

Código para definir robot,sensores,actuadores y entorno de *MORSE*.

```
# PASQUAL JOAN RIBOT LACOSTA 2018
# MORSE builder for an ATRV with pose and laser sensors.
from morse.builder import *
# Land robot
atrv = ATRV()
atrv.properties(GroundRobot=True)
atrv.translate(x=-12.5,y=7.5,z=0)
atrv.rotate(x=0,y=0,z=0)
#Laser range sensor
laser = Sick()
laser.properties(Visible_arc = True)
laser.properties(laser_range = 3)
laser.properties(resolution = 0.5)
laser.properties(scan_window = 360.0)
laser.frequency(20)
laser.translate(x=0,y=0,z=0.1)
atrv.append(laser)
#Position sensor
pose=Pose()
pose.frequency(20)
atrv.append(pose)
#Motion actuator
motion=MotionVW()
atrv.append(motion)

# Add interface for our robot's components
atrv.add_default_interface('ros')

#Environment setup
env = Environment('indoors-1/Labyrinth1',fastmode=False,
    auto_tune_time=False)
```

```
env._camera_location=(0,0,30)
env._camera_rotation=(0, 0, 0)
env.set_time_strategy(TimeStrategies.BestEffort)
```

APÉNDICE C

CÓDIGO *DrawPath.py*

Código para dibujar el recorrido realizado al final de una ejecución.

```
#PASQUAL JOAN RIBOT LACOSTA
# Drawing with Turtle of the tbug.py results.
#!/usr/bin/env python3

import rospy
from std_msgs.msg import *
from sensor_msgs.msg import *
from geometry_msgs.msg import *
import turtle
import numpy
import math
import os
import copy
import time
pos=[]
obs=[]
chg=[]
from numpy import loadtxt
#File reading
positions = open("positions.txt","r")
obstacles = open("obstacles.txt","r")
changes= open("changes.txt","r")
for line in positions:
    pos.append(float(line))
for line in obstacles:
    obs.append(float(line))
for line in changes:
    chg.append(float(line))
#parameters
mult=25
R_start=Vector3(-12.5,7.5,0)
```

```
Goal=Vector3(12.5,7.5,0)
laser_range=3
mtg=True
bf=False
#turtle set up
turtle.ht()
turtle.title("Robot's Path")
turtle.speed(0)
turtle.tracer(0,0)
turtle.up()
#Robot starting position and range
turtle.setposition(R_start.x*mult,R_start.y*mult-laser_range*mult)
turtle.down()
turtle.circle(laser_range*mult)
turtle.up()
turtle.setposition(R_start.x*mult,R_start.y*mult)
turtle.write("R",align="center",font=("Arial", 13, "normal"))
#Goal position
turtle.setposition(Goal.x*mult,Goal.y*mult)
turtle.write("G",align="center",font=("Arial", 13, "normal"))
# Mode color caption
turtle.setposition(0,10*mult)
turtle.pencolor("green")
turtle.write("MtG",align="center",font=("Arial", 13, "normal"))
turtle.setposition(3*mult,10*mult)
turtle.pencolor("blue")
turtle.write("BF",align="center",font=("Arial", 13, "normal"))
turtle.pencolor("red")
#Drawing of obstacles
for i in range (0,len(obs)-1,2):
  turtle.setposition(obs[i]*mult,obs[i+1]*mult)
  turtle.dot()
turtle.pencolor("green")
k=0
dist=0
#Drawing of robot's path
for i in range (0,len(pos)-1,2):
  if len(chg)!=0:
    if pos[i]==chg[k] and pos[i+1]==chg[k+1]:
      if mtg==True:
        turtle.pencolor("blue")
        mtg=False
        bf=True
        if k==len(chg)-2:
          k=0
        else:
          k=k+2
      elif bf==True:
        turtle.pencolor("green")
        mtg=True
        bf=False
        if k==len(chg)-2:
          k=0
```

```
        else:
          k=k+2

  turtle.setposition(pos[i]*mult,pos[i+1]*mult)
  turtle.dot()
#Calculating path's length
for i in range (0,len(pos)-4,2):
  dist=dist+(math.sqrt(((pos[i+2]-pos[i])**2)+((pos[i+3]-pos[i+1])**2)))
turtle.setposition(0,-10*mult)
turtle.pencolor("Black")
turtle.write("Path Length",align="center",font=("Arial", 13, "normal"))
turtle.setposition(4*mult,-10*mult)
turtle.write(round(dist,3),align="center",font=("Arial", 13, "normal"))
turtle.update()
ts = turtle.getscreen()
ts.getcanvas().postscript(file="Path.eps")
```

BIBLIOGRAFÍA

[1] H. Choset. (Último acceso: 26-06-2018) Robotic motion planning:bug algorithms. [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf

[2] Kamon, Ishay and Rivlin,Ehud end Rimon,Elon, "A New Range-Sensor Based Globally Convergent Navigation Algorithm for Mobile Robotss," in *IEEE International Conference on Robotics and Automation*, Minneapolis, Minnesota, April 1996.

[3] B. Kahyaoglu. (Último acceso: 26-06-2018). [Online]. Available: http://barankahyaoglu.com/robotics/tangent/

[4] Laas. (Último acceso: 26-06-2018). [Online]. Available: http://homepages.laas.fr/matthieu/robots/dala.shtml

[5] AutoRob. [Online]. Available: http://web.eecs.umich.edu/~ocj/courses/autorob/autorob_12_bugs.pdf

[6] Liu,Yun-Hui and Arimoto,Suguru, "Proposal of Tangent Graph and Extended Tangent Graph for Path Planning of Mobile Robots," in *IEEE International Conference on Robotics and Automation*, Sacramento, California, April 1991.

[7] S. Tully. (Último acceso: 26-06-2018) Robotic motion planning. [Online]. Available: https://www.cs.cmu.edu/~motionplanning/student_gallery/2006/st/hw2pub.htm

[8] I. Gentili. (Último acceso: 26-06-2018) Robotic motion planning. [Online]. Available: http://www.cs.cmu.edu/~motionplanning/student_gallery/2007/Iacopo/HW2_files/Tangent%20Bug.pdf

[9] H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementation*, 2010.

[10] F. Correia. (Último acceso: 26-06-2018). [Online]. Available: https://github.com/filipelbc/tangentbug

[11] M. Mutlu. (Último acceso: 26-06-2018) Tangent bug algorithm with a sensor constraint. [Online]. Available: https://sites.google.com/site/mutlumehmetmetuceng786/ceng786/bug-algorithms

[12] O. S. R. Foundation. (Último acceso: 4-05-2018). [Online]. Available: http://wiki.ros.org/

[13] O. Robots. (Último acceso: 4-05-2018). [Online]. Available: https://www.openrobots.org/wiki/morse

[14] B. Foundation. (Último acceso: 4-05-2018). [Online]. Available: https://www.blender.org

[15] Syrotek. (Último acceso: 26-06-2018). [Online]. Available: https://syrotek.felk.cvut.cz/course/ROS_CPP_INTRO/exercise/ROS_CPP_WALLFOLLOWING

[16] I. T. U. Group, *L A TEX Tutorials*, 2003.

[17] I. Noda, A. Noriakim, D. Brugali, and J. Kuffner, *Simulation, Modeling and Programming for Autonumous Robots*. Springer, 2012.