UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

SIMULADOR DE ROBOTS MÓVILES

T E S I S

QUE PARA OPTAR POR EL GRADO DE:

MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

P R E S E N T A:

R E Y N A L D O M A R T E L L A V I L A

DIRECTOR DE TESIS:

DR. JESÚS SAVAGE CARMONA

FACULTAD DE INGENIERÍA, UNAM

CIUDAD UNIVERSITARIA, CD. MX. ENERO, 2017

Agradecimientos.

A mis padres Isabel y Reynaldo por todo su cariño y apoyo que me han brindado, sin ustedes no podría haber cumplido esta gran meta en mi vida.

A mis hermanas Isabel y Alma por su afecto y consejos que me brindaron, son las mejores hermanas que puedo pedir.

A mis sobrinos Gonzalo, Héctor, Eli, Nicolas, Miguel y Yarisma por ser el impulso que necesitaba para concluir este logro.

A Stephanie por ser el amor de mi vida, agradezco cada momento que me disté, das y seguirás dando, por todos tus consejos y tu amor inquebrantable.

A toda mi familia por el apoyo incondicional que nunca me han dejado de dar.

Al Dr. Jesús Savage por darme la oportunidad de pertenecer a su equipo de trabajo, por su confianza y el tiempo invertido en la mejora de este trabajo.

A Dios por permitirme haber llegado hasta aquí.

A mis amigos ingenieros por su amistad que a pesar de los años aún perdura.

A mis amigos de la maestría Hugo, Isra, Julio, Marco y Pedro por hacer que mi estancia en la maestría haya sido más grata.

A la UNAM por ser mi alma mater.

Agradezco al Posgrado en Ciencia e Ingeniería de la Computación por el apoyo, los conocimientos y enseñanzas que me ha brindado.

A mis compañeros del laboratorio de Bio-Robótica con quienes he pasado buenos momentos.

A mis sinodales Dr. Miguel Padilla, Dr. Pablo Perez, M.I. Abel Pacheco y Mat. Ana Luisa Solís por sus conocimientos y recomendaciones para la mejora de mi proyecto de investigación.

A CONACYT por el apoyo proporcionado durante mis estudios de maestría.

Se agradece a la DGAPA-UNAM por el apoyo proporcionado para la realización de esta tesis a través del proyecto PAPIIT IG100915 "Desarrollo de técnicas de la robótica aplicadas a las artes escénicas y visuales".

Índice general.

Índice de figuras.

Índice de tablas.

Capítulo 1. Introducción.

Robots móviles es el área de investigación que se ocupa del control de vehículos autónomos y semiautónomos [1]. El gran crecimiento en las investigaciones de robots móviles desde los años 70 ha dado origen a utilizar y desarrollar novedosos algoritmos que proporcionan la habilidad para navegar. La habilidad de navegar es fundamental para muchos animales y organismos inteligentes [1].

Existen diferentes paradigmas de modelos para el control de robots móviles. En primer lugar tenemos los modelos basados en comportamientos reactivos, los cuales tienen como objetivo evaluar las entradas al sistema para producir un resultado, como por ejemplo, se tienen campos potenciales, redes neuronales, algoritmos genéticos, etc [2]. Estos algoritmos tienen la particularidad de procesar información masivamente, en forma paralela, lo que conlleva a que sean rápidos. En segundo lugar tenemos los modelos tradicionales, los cuales hacen uso del medio ambiente para poder tomar decisiones con base en el entorno en donde se encuentre, es decir, el robot actúa y modifica el estado del medio ambiente que se tiene representado. Cuando se combinan los modelos reactivos y los tradicionales se le denominan modelos híbridos, así se puede tener y aprovechar las características de ambos modelos, como por ejemplo, un robot que utiliza algoritmos de búsqueda en una red topológica para encontrar la ruta optima (modelo tradicional); por otra parte, emplear la planeación de trayectorias guiadas por campos potenciales (modelo reactivo) en la que cada nodo obtenido del modelo tradicional se convierte en el campo atractivo que dirige el movimiento.

Crear ambientes simulados para robots, en el cual se puedan visualizar los modelos mencionados anteriormente, es de gran importancia para los desarrolladores dedicados a la programación de algoritmos encargados de proporcionar inteligencia artificial a los robots, debido a que ofrecen herramientas que pueden ser utilizadas para iniciar o mejorar la autonomía de los robots.

Hoy en día los simuladores se enfocan en el aprendizaje de una tarea en específico, por ejemplo; los simuladores de vuelos se especializan en el entrenamiento de pilotos capaces de operar un avión en un corto periodo de tiempo, evitando así gastos en comparación con el uso de un avión real. En cuanto a la robótica es algo muy parecido ya que a menudo los robots implican gastos que un estudiante no puede realizar, o que en ocasiones, se desea tener una estimación rápida de cómo funciona una tarea programada en un agente autónomo.

1.1. Objetivo.

Desarrollar un simulador 3D para robots móviles, mediante el uso de librerías para la comunicación entre proceso y herramientas de visualización que proporciona el marco de trabajo ROS, el cual apoye al desarrollo y aprendizaje de los algoritmos empleados en la navegación y planeación de movimiento de robots móviles.

Para el cumplimiento de este objetivo, se plantearon las siguientes actividades.

- Familiarizarse con el simulador *Roc2* que servirá como base para el diseño y desarrollo del nuevo simulador.
- Obtener los conocimientos de las librerías y herramientas que proporciona *ROS*.
- Desarrollar los algoritmos que permitan la detección de colisiones entre el robot y los obstáculos del medio ambiente virtual.
- Diseñar e implementar los algoritmos que simulen la funcionalidad de un sensor láser.
- Utilizar las librerías y herramientas de ROS para el desarrollo de la nueva versión del simulador.
- Desarrollar la simulación de movimientos y comportamiento utilizando campos potenciales empleando el sistema y así poder probarlo experimentalmente.
- Desarrollar un algoritmo que permita hacer la planeación de movimientos de un robot móvil utilizando un mapa topológico.
- En base a un mapa de rejillas de ocupación espacial, implementar un algoritmo de búsqueda para encontrar la ruta entre los nodos de navegación

1.2. Hipótesis.

La hipótesis que se generó al inicio y durante el desarrollo del presente trabajo de investigación es:

> *"Se consigue desarrollar un simulador de robots móviles que permita implementar prototipos de algunas de las técnicas empleadas para el control autónomo de Robots Móviles"*

Para finalizar el presente trabajo y validar la hipótesis mencionada anteriormente, se implementaron una serie de algoritmos para la planeación de movimiento de un robot móvil. Respecto al alcance, se considera valida la hipótesis si el simulador desarrollado permite visualizar la simulación de estos algoritmos.

1.3. Motivación.

Uno de los problemas que surgen durante el aprendizaje de robótica es la necesidad de contar con un robot que ya haya sido construido y que cuente con una arquitectura ya definida con la capacidad de desarrollar los comandos de movimientos, lecturas de sensores, módulo de visión, etc. Esto conlleva a tener una preparación enfocada al diseño y construcción de robots que tenga las capacidades necesarias para así llevar a cabo los algoritmos de navegación.

Lo que se demuestra con lo descrito anteriormente es que el aprendizaje de robótica requiere de un gran número de recursos, como herramientas, personal especializado, espacio de laboratorio, etc. [3]. Debido a esto, el uso de herramientas tales como los simuladores de robots son de gran importancia, ya que proporciona rápidamente un prototipo de aplicaciones, comportamientos, ambientes; así como una herramienta para poder probar las tareas que llevan a cabo los robots a un alto nivel [4]. Los simuladores juegan un papel muy importante para el aprendizaje de robótica, debido a que ofrecen una poderosa herramienta que permite desarrollar y experimentar sin tener el robot real [4].

Los simuladores 2D a menudo son muy útiles ya que proporcionan la evaluación de un comportamiento con diferentes robots y ambientes; sin embargo, esto no siempre es suficiente, ya que constantemente se requiere que el ambiente en el cual se pretende probar el buen funcionamiento del comportamiento se lleve a cabo en un ambiente 3D [4].

En la actualidad existen varios simuladores como: *Roc2, USARSim, Stage, Gazebo, Sim, Webots, Morse*. *Roc2* es un simulador de robots móviles que se utilizó para el aprendizaje de la materia de robots móviles y agentes inteligentes, impartida por la Facultad de Ingeniería. Este simulador fue desarrollado por el laboratorio de Bio-Robótica en el año 2009 y ya no se utiliza debido a que está desactualizado y funciona en plataformas que ya no son compatibles.

Los simuladores deben contar con un cierto número de características como son:

- Flexibilidad. Deben permitir la simulación con diferentes robots y ambientes.
- Realismo Físico. Los robots y los ambientes virtuales deben ser capaces de modelarse a través de las leyes físicas de cuerpos rígidos.
- Realismo Visual. La apariencia del sistema debe ser lo más exacto y así garantizar la precisión de la lectura de los sensores.
- Eficiencia. Deben correr en tiempo real con una visualización mínima de 30 cuadros por segundo.
- Modularidad. Se deben agregar y modificar características de acuerdo al ambiente y a los robots, lo cual incluye a los sensores de entrada y salida.
- Interfaz de programación. Debe ser flexible de tal forma que la programación sea como en el robot real [4].

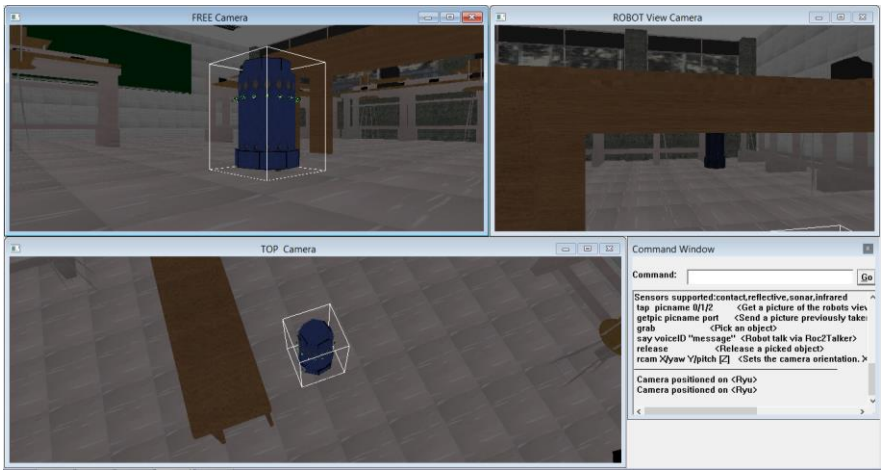


Figura 1.1: Simulador Roc2.

1.4. Organización de la tesis.

El presente trabajo está integrado por siete capítulos, los cuales se describen a continuación.

En el segundo capítulo, *Marco Teórico,* se presentan conceptos y definiciones que le dan soporte a este trabajo.

El tercer capítulo, *Herramientas de software,* expone las herramientas que se utilizarán para el desarrollo del simulador.

En el cuarto capítulo, *Simulador de robots móviles*, se detalla los componentes (modelo del robot, representación del mundo, movimientos básicos, simulación del láser e interacción) del simulador.

El quinto capítulo, *Implementación de comportamientos,* describe el uso del sistema desarrollado, paquetes que proporciona ROS y paquetes del laboratorio de Bio-Robótica de la Faculta de Ingeniería para la implementación de la planeación de movimientos utilizando un mapa topológico, así como un mapa de rejillas de ocupación espacial.

En el sexto capítulo *Pruebas y resultados* se exponen los resultados obtenidos de la utilización del sistema, así como un análisis de estos.

El séptimo capítulo *Conclusiones y trabajo a futuro* se hace un análisis de las pruebas realizadas a los algoritmos implementados y de los beneficios de utilizar el sistema.

Capítulo 2. Marco teórico.

2.1. El robot Justina.

Justina es un robot de servicio diseñado y construido en el laboratorio de Bio-Robótica de la Facultad de Ingeniería de la UNAM. Justina participa en la competencia internacional de robótica *RoboCup* que se lleva a cabo anualmente.

El robot Justina actualmente cuenta con 4 motores, dos de estos están colocados lateralmente, uno en cada lado del robot y los otros dos se encuentran distribuidos en la parte delantera y trasera de la base móvil.

Justina cuenta con dos brazos manipuladores colocados lateralmente, uno en cada lado, contando ambos brazos con siete grados de libertad. Otro componente importante del robot Justina es la cabeza mecatrónica, con dos grados de libertad, los cuales están basados en los correspondientes movimientos de la cabeza humana (*pan* y *tilt*). El último componente de Justina es el torso, el cual permite controlar la altura del robot, y es de gran utilidad para el reconocimiento y manipulación de objetos. En la Figura 2.1 se muestra el robot Justina.

Figura 2.1: Robot Justina.

Para coordinar las diferentes tareas de la simulación del robot, se utiliza una arquitectura basada en el sistema Virbot [5], en el cual se distribuyen diferentes módulos especializados para realizar tareas conjuntamente. En la Figura 2.2 se observan los diferentes módulos y la interconexión entre ellos.

Los módulos de la arquitectura *Virbot* que son utilizados para el desarrollo del simulador son: *External sensor, World Model, Motion Planner, Behavior Methods, Control Algorithms* y *Actuators*. El motivo por el cual son utilizados estos módulos es debido a la separación de los procesos que conforman el simulador en: módulo sensorial, módulo para la representación del mundo, módulo de planeación de movimientos, módulo para controlar comportamientos y módulo para la simulación de control y actuadores.

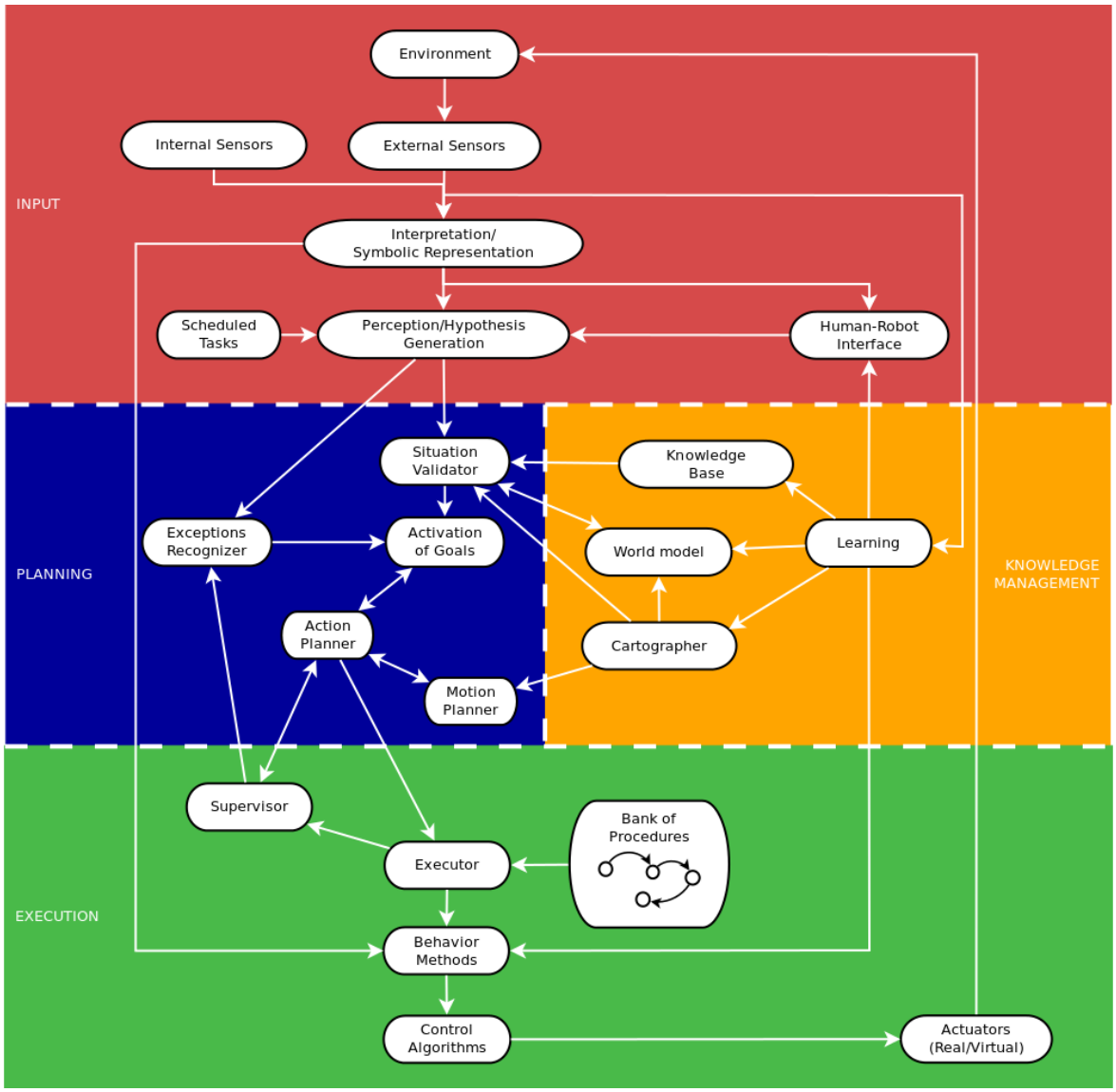


Figura 2.2: Diagrama de bloques de la arquitectura VirBot.

2.2. Modelos para el control de robots móviles.

En este apartado se detallan los conceptos que se relacionan con los modelos de control para robots móviles y sus características, para así tener un panorama general de cuáles comportamientos pueden ser aplicados a determinadas tareas.

El motivo por el cual se abordan estos temas es que posteriormente se aplicarán para desarrollar la simulación de los comportamientos de navegación de robots móviles, teniendo como base que, el simulador se usará para visualizar el funcionamiento de los comportamientos.

2.2.1. Comportamientos tradicionales.

En los comportamientos tradicionales, el robot tiene definido conocimiento del medio ambiente, para así tomar en cuenta esta información y poder determinar llevar a cabo una secuencia de acciones. Usualmente, la organización por módulos de los comportamientos tradicionales consta de 5 módulos: un módulo sensorial que permite captar la información obtenida por los sensores, un módulo de modelado, un módulo de planificación, un módulo de evaluación de valores y un módulo de ejecución [6].

Esta separación nos permite contar con un pipeline de funcionalidad con el fin de poder separar operaciones en cada uno de los módulos sin que sean dependientes unos de otros, es decir, que se pueda modificar un módulo sin afectar al otro. Un problema típico de estos modelos radica en la planificación de acciones, la cual consiste en realizar una búsqueda de las posibles secuencias de acciones dado un determinado estado del robot. La planificación de acciones es el componente principal de la inteligencia artificial.

El proceso para que el robot lleve a cabo la planificación de acciones requiere obtener la lectura de los sensores, planificación de movimientos, planificación de acciones y llevarlas a cabo.

Anteriormente se hace mención de que en los comportamientos tradicionales existe una representación del medio ambiente, esto nos permite hacer predicciones de acciones en los estados del robot con el fin de obtener planes. Lo que se observa es que la representación del medio ambiente siempre debe estar bien definida y actualizada, tomando en cuenta estas dos condiciones de la representación del medio ambiente, se pueden obtener planes óptimos para el estado en el que se encuentre el robot.

Existen dos problemas con la representación del medio ambiente: se tiene que muchas de las veces la representación del ambiente no es ideal causado por la existencia de la gran cantidad de ruido en ellos, los ambientes son dinámicos esto quiere decir que no necesariamente se cuenta con una representación del medio ambiente actualizada. Imaginar que el robot tiene la representación topológica del ambiente, pero por alguna razón el ambiente cambia teniendo otros objetos que el robot desconoce. Para solucionar estos problemas se desarrollaron otros modelos que proporcionan acciones mucho más rápidas y que son ideales para ambientes dinámicos del mundo real.

2.2.2. Comportamientos reactivos.

Los comportamientos reactivos son aquellos que utilizan únicamente las entradas sensoriales para llevar a cabo una acción. En los comportamientos reactivos no interviene ningún otro tipo de información, esto con el fin de obtener una respuesta rápidamente para ambientes dinámicos [7]. Los comportamientos reactivos se basan en un paradigma de estímulo-respuesta, a diferencia de los modelos tradicionales que no requieren de la generación de ningún tipo de información que estructure el ambiente.

Los comportamientos reactivos logran procesar las entradas de los sensores y generar la respuesta en tiempo real, construyendo un controlador de acciones que permite crear reglas del tipo if-then; por ejemplo, si choca-detenerse, si detiene-reversa, si reversa-detente, etc. Además, se pueden tener diversos métodos reactivos corriendo en paralelo y contar con un módulo que decida cuál método es el que mejor se ajusta en un determinado estado [7]. Estas características hacen que los modelos reactivos sean aptos para ambientes dinámicos, además que no cuenten con una estructura del ambiente, ya que contar con una estructura del ambiente no es una opción realista.

El poder de cómputo que utilizan los modelos reactivos es mínimo en comparación con los modelos tradicionales, lo que permite responder rápidamente a los estímulos que se reciban.

Los métodos reactivos tienen como base el comportamiento de los insectos, estimulo-respuesta y son un método muy poderoso; sin embargo, tiene su limitación en el hecho de que no guardan ningún tipo de estado previo, ni representación del medio ambiente, lo que impide tener una limitación en el aprendizaje durante un periodo de tiempo [8].

Analizando lo comentado anteriormente, se concluye que los modelos reactivos son muy potentes cuando no se requiera ningún tipo de información estructural del ambiente y no se requiere memoria ni aprendizaje. Si dentro de los requerimientos del problema, se requiere

lo comentado anteriormente, los comportamientos reactivos no son adecuados para la solución del problema.

2.2.3. Comportamientos híbridos.

Los comportamientos híbridos combinan las mejores características de los comportamientos reactivos y tradicionales: estimulo-respuesta y cuentan con conocimiento del medio ambiente para la toma de decisiones. Durante la ejecución de los comportamientos híbridos se debe tener un componente intermedio que coordine ambos componentes, como por ejemplo, la tarea de llegar de un punto a otro, lo cual se realiza con un algoritmo de planeación (utilizando la información del medio ambiente). Este algoritmo obtiene los lugares que debe visitar el robot, mientras no haya un objeto desconocido en medio de los lugares a visitar, los comportamientos tradicionales funciona perfectamente; sin embargo, en el instante que los sensores detecten objetos desconocidos, los comportamientos reactivos deben entrar en acción dejando a un lado la trayectoria que inicialmente debían seguir; en consecuencia la parte reactiva evade el objeto y cuando ésta finalice, el robot debe seguir su trayectoria nuevamente con el modelo tradicional.

Como se puede observar, se debe tener un control en la coordinación de los comportamientos con el fin de que no entren en conflicto, es decir, si la parte reactiva está ejecutándose y entra en acción el modelo tradicional, al finalizar debe informar al modelo reactivo para retomar el flujo normal de la ejecución. Del mismo modo sucede cuando la parte tradicional se encuentra en ejecución y haya una ejecución por la parte reactiva, ésta debe informarle a la componente tradicional cuando haya finalizado y tome nuevamente el control de la tarea.

2.3. Transformaciones geométricas.

Una transformación es una función f que toma un punto (o vector) y hace el mapeo a otro punto (o vector). Si se utilizan coordenadas homogéneas se tiene representado el vector o punto en una matriz con una columna de cuatro dimensiones. Se puede obtener una clase de transformación muy usada si colocamos una restricción en f. Esa restricción es la linealidad. Una función f es una función lineal si y solo si, para cualquier escalares α, β y dos vértices (o vectores) p, q se cumple lo siguiente [9].

$$f(\alpha p + \beta q) = \alpha f(p) + \beta f(q) \tag{2.1}$$

Entonces, una transformación lineal mapea la representación de un punto o vector en otra representación punto (o vector) y puede ser definida en términos de dos representaciones u y v como multiplicación de matrices como se muestra en la ecuación (2.2).

$$v = Cu \tag{2.2}$$

Donde C es una matriz cuadrada y si se trabaja en coordenadas homogéneas es de dimensión 4x4.

2.3.1. Translación.

La translación es una operación que desplaza un punto en una dirección. Para realizar una translación se necesita especificar un vector de desplazamiento d. La transformación de translación tiene tres grados de libertad debido a que sólo se especifican tres componentes del vector de desplazamiento. La translación de un punto P con un vector d de desplazamiento está dada por:

$$P' = P + d \tag{2.3}$$

2.3.2. Rotación.

Este tipo de transformación es más difícil en comparación con la translación debido a que depende de más parámetros. Considerar la rotación de un punto p en dos dimensiones. El punto p en (x, y) se quiere rotar un ángulo θ generando una nueva posición (x', y'), considerar la representación polar de los puntos (x, y) y (x', y'):

$$x = \rho \cos \phi \tag{2.4}$$

$$x = \rho \sin \phi \tag{2.5}$$

$$x' = \rho \cos(\theta + \phi) \tag{2.6}$$

$$y' = \rho \sin(\theta + \phi) \tag{2.7}$$

Expandiendo las ecuaciones anteriores usando identidades trigonométricos del seno y coseno de sumas de dos ángulos se tiene:

$$x' = \rho \cos \phi \cos \theta - \rho \sin \phi \sin \theta = x \cos \theta - y \sin \theta \tag{2.8}$$

$$y' = \rho \cos \phi \sin \theta - \rho \sin \phi \cos \theta = x \sin \theta + y \cos \theta \tag{2.9}$$

Estas ecuaciones se pueden representar en forma matricial como se muestran en la siguiente ecuación:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.10}$$

La ecuación (2.10) solo define la rotación de un punto en dos dimensiones; sin embargo, este mismo análisis se puede expandir para tres dimensiones tomando en cuenta lo siguiente:

- Existe un punto que no cambia al que se le conoce como origen de la transformación.
- Las rotaciones positivas están dadas de acuerdo a la regla de la mano derecha en el sistema de referencia de tres dimensiones, para el caso de dos dimensiones cuando $z = 0$ la rotación positiva se da en sentido inverso a las manecillas del reloj.
- La rotación en dos dimensiones cuando el plano $z = 0$ es equivalente a la rotación en tres dimensiones alrededor del eje z [9].

La matriz de rotación en tres dimensiones es la que se muestra en la siguiente ecuación:

$$R_z = R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

Para las matrices de rotación alrededor del eje x y el eje y se utiliza la ecuación (2.11).

$$R_x = R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.12}$$

$$R_y = R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.13}$$

Estas ecuaciones conservan el sentido del giro positivo de acuerdo a la regla de la mano derecha.

2.3.3. Escalamiento.

En la transformación de escalamiento, al igual que la transformación de rotación, existe un punto que no cambia llamado origen de la transformación. Una matriz de escalamiento permite escalar en los ejes coordenados. Las tres ecuaciones de escalamiento en cada uno de sus ejes son:

$$x' = \beta_x x \tag{2.14}$$

$$y' = \beta_y y \tag{2.15}$$

$$z' = \beta_z z \tag{2.16}$$

Estas ecuaciones pueden ser agrupadas en su forma matricial usando coordenadas homogéneas.

$$S = s(\beta_x, \beta_y, \beta_z) = \begin{pmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.17}$$

2.4. Modelo cinemático de un robot móvil.

Cinemática es el estudio del comportamiento de un sistema de mecanismos, pero en robots móviles la idea es entender el comportamiento del mecanismo para diseñarlo de acuerdo a las tareas que el robot realiza. Los robots móviles no son los únicos que necesitan este análisis, también los manipuladores han sido objeto de estudio por más de treinta años [10]. Los robots manipuladores son mucho más complejos que los robots móviles debido a que el grado de complejidad del mecanismo es mucho mayor.

Ambos tipos de mecanismos comparten conceptos en común, como por ejemplo, el espacio de trabajo del robot, el cual es de gran importancia debido que define el rango de posibles posiciones; otro punto en común es el control que se implementa en un brazo robótico, en éste se define el engranaje de motores que es usado para mover el manipulador a una posición en el espacio de trabajo, por otro lado, para un robot móvil el control define las posibles rutas y trayectorias en el espacio de trabajo [10]. En primera instancia se analiza la representación de la posición del robot móvil, éste es visto como un cuerpo rígido con ruedas, los grados de libertad de un robot en el plano es de tres, dos para la posición en el plano y una para la orientación sobre el eje perpendicular al plano.

Para especificar la posición del robot en el plano se define la relación que existe entre el sistema de referencia global y local del robot. La Figura 2.3 muestra el sistema de referencia global y local del robot.

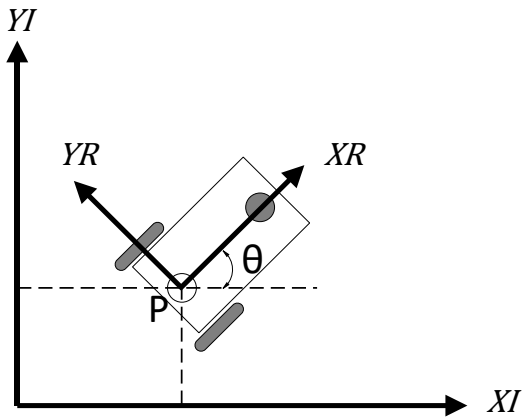


Figura 2.3: Sistema de referencia global y sistema de referencia del robot [10].

Para describir el movimiento de un robot se necesita hacer el mapeo del movimiento del sistema de referencia global al sistema de referencia local del robot. La ecuación que se usa para mapear el movimiento del sistema de referencia global (X_I, Y_I) en términos del sistema de referencia local (X_R, Y_R) es la que se muestra en la ecuación (2.18) [10].

$$R(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

La ecuación (2.18) es usada para transformar el movimiento del robot del sistema de referencia (X_I, Y_I) al sistema de referencia (X_R, Y_R). A esta operación se le denota por:

$$\dot{\xi}_R = R(\theta)\dot{\xi}_I \tag{2.19}$$

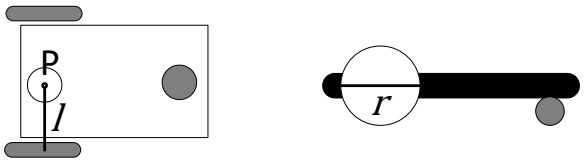
Donde:

$$\dot{\xi}_I = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} \tag{2.20}$$

La ecuación (2.19) indica que dada una velocidad $(\dot{x}, \dot{y}, \dot{\theta})$ en el sistema de referencia global, se transforman las componentes del movimiento del robot en sus ejes locales (X_R, Y_R).

La ecuación (2.19) es insuficiente para representar el movimiento cinemático del robot, ya que no contiene parámetros relacionados con su geometría y velocidades de las llantas del robot.

El modelo diferencial de un robot consiste de dos ruedas montadas en un eje común, como se muestra en la Figura 2.4, cada rueda tiene un diámetro r, son independientes entre sí y tiene la libertad de moverse hacia adelante o atrás. Dado un punto *P* entre las dos llantas, cada llanta tiene una distancia l a P.



a) Dado un punto P entre las dos llantas, cada llanta tiene una distancia l a P.

b) Diámetro de la ruda.

Figura 2.4: Parámetros del modelo cinemático.

Con estas definiciones se representa el modelo cinemático de un robot diferencial en términos de $r, l, \theta, \dot{\phi}_r, \dot{\phi}_l$, donde $\dot{\phi}_d, \dot{\phi}_i$ son las velocidades de giro de cada llanta. La estrategia cosiste en primero calcular la contribución de las dos llantas en el sistema de referencia del robot $\dot{\xi}_R$.

Antes de llegar al modelo cinemático de un robot diferencial hay que hacer algunas consideraciones: la contribución de la velocidad de giro en cada llanta a la velocidad de translación de *P* se realiza únicamente en dirección de $+X_R$; debido a que *P* se encuentra a la mitad de las dos llantas, éste se moverá a la mitad de la velocidad $\dot{x}_{rd} = \frac{r\dot{\phi}_d}{2}$ y $\dot{x}_{ri} = \frac{r\dot{\phi}_\iota}{2}$, estas dos contribuciones son agregadas para calcular la componente $\dot{x}_R$ de $\dot{\xi}_R$, considerando un robot diferencial, el giro de cada llanta con la misma velocidad pero en sentido contrario una de otra resulta un movimiento estacionario, se espera que $\dot{x}_R$ sea cero en este caso, mientras el valor de $\dot{y}_R$ es simple de calcular ya que $\dot{y}_R$ siempre será cero. La velocidad angular w_d de la llanta derecha puede ser calculada debido a que la llanta se mueve a lo largo de un círculo de radio $2l$ como se muestra en la ecuación (2.21) [10].

$$w_d = \frac{r\dot{\phi}_d}{2l}, w_i = \frac{-r\dot{\phi}_i}{2l} \tag{2.21}$$

La idea es calcular la contribución de las dos llantas en el sistema de referencia del robot $\dot{\xi}_R$, teniendo como resultado la siguiente ecuación.

$$\dot{\xi}_I = R(\theta)^{-1}\dot{\xi}_R \tag{2.22}$$

Combinando las ecuaciones (2.21) y (2.22) se llega al modelo cinemático de un robot móvil de par diferencial.

$$\dot{\xi}_1 = R(\theta)^{-1}\begin{pmatrix} \frac{r\dot{\phi}_d}{2} + \frac{r\dot{\phi}_i}{2} \\ 0 \\ \frac{r\dot{\phi}_d}{2l} + \frac{-r\dot{\phi}_i}{2l} \end{pmatrix} \tag{2.23}$$

2.5. Sensores.

En esta sección de la investigación se analiza los tipos de sensores utilizados comúnmente en un robot. Una de las tareas más importantes para cualquier tipo de agente autónomo es obtener la información de cómo está representado el medio ambiente, lo cual se logra tomando mediciones de algún tipo de sensor que extrae la información [11].

Existe una variedad de sensores usados en un robo móvil; algunos son empleados para medir valores simples como la temperatura interna del robot o el movimiento rotacional en las ruedas de los motores. Los sensores más sofisticados son usados para obtener la información del ambiente o directamente medir la posición global del robot.

Considérese la situación en la que el robot realiza la tarea de llegar de un punto a otro. Cualquier robot que se mueva en el mundo real debe ser capaz de evadir obstáculos para así llegar a una posición indicada; sin embargo, en la práctica los objetos tienen un sin fin de formas; por lo tanto, el robot debe conocer con cierta precisión la posición y orientación de los objetos del medio ambiente.

A continuación se muestra la clasificación de sensores de acuerdo al tipo de medición y la forma que es obtenida:

- Sensores propioceptivos: Miden valores internos al sistema del robot, por ejemplo: velocidad de motores, voltaje de la batería, ángulos de giro en las articulaciones de los brazos, etc.
- Sensores exteroceptivos: Adquieren información del ambiente en el que se encuentra el robot, por ejemplo: medición de distancia, intensidad de luz, amplitud de sonido; por lo tanto, las mediciones de los sensores exteroceptivos son interpretadas por el robot con el fin de extraer características significativas del ambiente.
- Sensores pasivos: Miden la energía entrante del medio ambiente, por ejemplo: temperatura, micrófonos, cámaras CCD o CMOS, etc.
- Sensores activos: Emiten energía hacia el medio ambiente obteniendo la medición de acuerdo a la reacción del ambiente. Este tipo de sensores es sensible al ruido, que influye al resultado de la medición [10].

La siguiente tabla muestra la clasificación de los sensores más usados en robots móviles.

Clasificación.	**Uso común.**	**Sensor.**	**PC o EC**	**A o P**
Sensores de contacto.	Detección de contacto físico o de proximidad.	Fin de carrera.	EC	P
		Sensores ópticos.	EC	A
		Sensores de proximidad.	EC	A
Sensores de motores.	Velocidad y posición de los motores.	Encoders.	PC	P
		Potenciómetros.	PC	P
		Encoders ópticos.	PC	A
		Encoders magnéticos.	PC	A
Sensores de orientación.	Orientación del robot en relación al sistema de referencia.	Compás.	EC	P
		Giroscopio.	PC	P
		Clinómetro.	EC	A/P
Sensores de localización.	Localización en el sistema de referencia.	GPS.	EC	A
		Radiobalizas ópticos o de RF.	EC	A

		Radiobalizas ultrasónicos.	EC	A
		Radiobalizas reflectivos.	EC	A
Sensores de rango.	Reflectividad, tiempo de vuelo y triangulación geométrica.	Sensores reflectivos.	EC	A
		Sensores ultrasónicos.	EC	A
		Sensor láser.	EC	A
		Triangulación óptica.	EC	A
		Luz estructurada.	EC	A
Sensores de visión.	Análisis de imágenes, segmentación, reconocimiento de objetos.	Cámaras CCD/CMOD	EC.	P

Tabla 2.1: Clasificación de sensores. A-activo, P-pasivo, PC-propioceptivos, EC-exteroceptivos [10].

Para esta investigación se simula únicamente un sensor láser, el cual mide la distancia que existe entre el robot y los objetos a su alrededor. Los sensores láser consisten en un transmisor que ilumina el objetivo con un haz que se refleja en el objeto y un receptor que recibe la componente de luz reflejada. Los sensores láser miden la distancia en base al tiempo que necesita el haz de luz en llegar al objetivo y regresar.

2.6. Líneas, rayos y segmentos.

Una línea se define como un conjunto de puntos que se pueden expresar como una combinación lineal de dos puntos distintos A y B [12].

$$L(t) = (1 - t)A + tB \tag{2.24}$$

Donde $-\infty < t < \infty$. El segmento de línea $\overline{AB}$ es la porción finita de A y B, limitada por la reducción del rango $0 \leq t \leq 1$. Los segmentos tienen dirección y están definidos dependiendo del orden de A y B. Un rayo es una media línea al infinito, que de igual forma se define con la ecuación (2.24), pero limitada únicamente por $t \geq 0$. La Figura 2.5 muestra las diferencias entre una línea, un segmento de línea y un rayo.

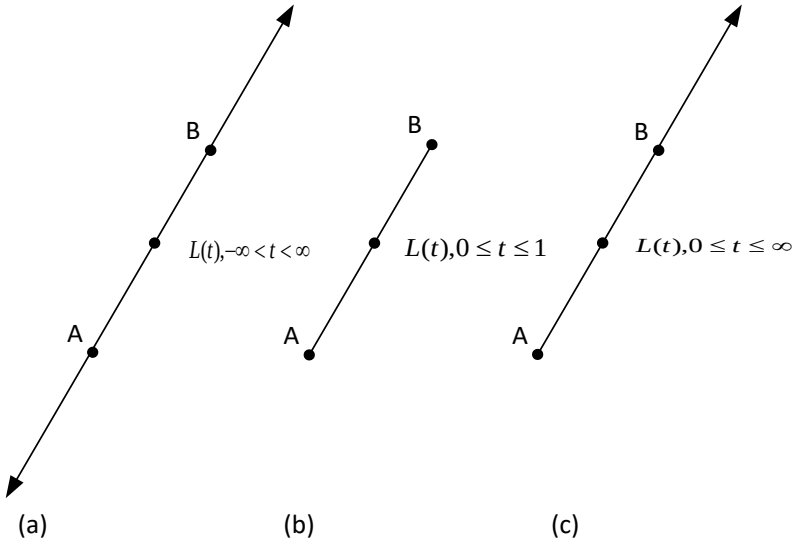


Figura 2.5: (a) Línea, (b) Segmento y (c) Rayo [12].

Otra forma de representar una línea, segmento o rayo es por medio de su forma paramétrica como se muestra en la ecuación (2.25).

$$L(t) = A + tV \text{ , } donde\ V = B - A \tag{2.25}$$

2.7. Planos y semiplanos.

Un plano en el espacio de tres dimensiones es una superficie plana extendiéndose indefinidamente en todas las direcciones. Un plano se define de múltiples formas, por ejemplo:

- Tres puntos no colineales (forman un triángulo sobre el plano).
- La normal y un punto sobre el plano.
- La normal y una distancia al origen [12].

En el primer caso, los tres puntos $A, B\ y\ C$, permiten expresar al plano P en su forma paramétrica como se muestra en la ecuación (2.26).

$$P(u,v) = A + u(B - A) + v(C - A) \tag{2.26}$$

Para el segundo y tercer caso la normal del plano es un vector no nulo perpendicular a cualquier vector sobre el plano. Existen dos posibles normales en direcciones opuestas; sin embargo, se toma la convención de tomar los tres puntos en orden inverso a las manecillas del reloj, haciendo que la normal tenga la dirección hacia el observador. La normal es calculada con el producto cruz de los segmentos dirigidos entre los tres puntos que definen el plano como se muestra en la ecuación (2.27).

$$n = (B - A)X(C - A) \tag{2.27}$$

Dada la normal n y un punto p sobre el plano, todos los puntos X sobre el plano pueden ser definidos por el vector $X - P$ siendo perpendicular a n. Por otra parte dos vectores son perpendiculares si su producto punto es igual a cero, lo que implica la ecuación del plano en su forma punto-normal [12].

$$n * (X - P) = 0 \tag{2.28}$$

Debido a que el producto punto es un operador lineal, permite distribuir la expresión como se muestra en la ecuación (2.29).

$$n * X = d \tag{2.29}$$

Donde.

$$d = n * P \tag{2.30}$$

Cuando dos planos no paralelos entre si se intersectan dan como resultado una línea, del mismo modo para tres planos no paralelos su intersección es un solo punto.

Los planos en una dimensión arbitraria son conocidos como hiperplanos, en 2D un hiperplano corresponde a una línea, en 3D corresponde a un plano. Cuando un hiperplano divide el espacio en dos conjuntos de puntos infinitos se les llama semiplanos [12].

2.8. Polígonos.

Un polígono es una figura cerrada con n lados que es definido por un conjunto de tres o más puntos en el plano, de tal forma que cada punto está conectado al siguiente (el último al primero) con un segmento de línea [13]. A los segmentos de línea que definen el borde de un polígono se le conocen como aristas y a los puntos se les denomina vértices del polígono, dos vértices son adyacentes si estos forman una arista [9].

Un polígono es simple si no existen dos aristas que se crucen entre sí. En la Figura 2.6 se muestran los componentes de un polígono. Normalmente se refiere un polígono a la frontera y su interior.

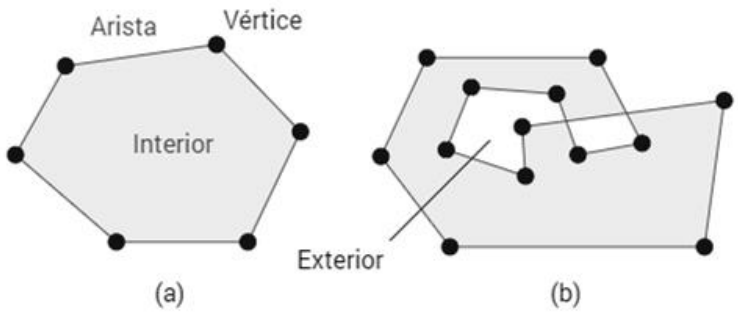


Figura 2.6: Componentes de un polígono. (a) Polígono Simple, (b) Polígono no simple.

Un vértice es convexo si el ángulo interior que se forma entre las conexiones del vértice miden igual o menor que 180. Si el ángulo que se forma es mayor a 180 es un vértice cóncavo [12].

Un polígono P es un polígono convexo si todos los segmentos de línea entre dos puntos de P están totalmente contenidos en el polígono; un polígono que no es convexo se le llama polígono cóncavo. La Figura 2.7 muestra gráficamente las definiciones que se mencionan anteriormente.

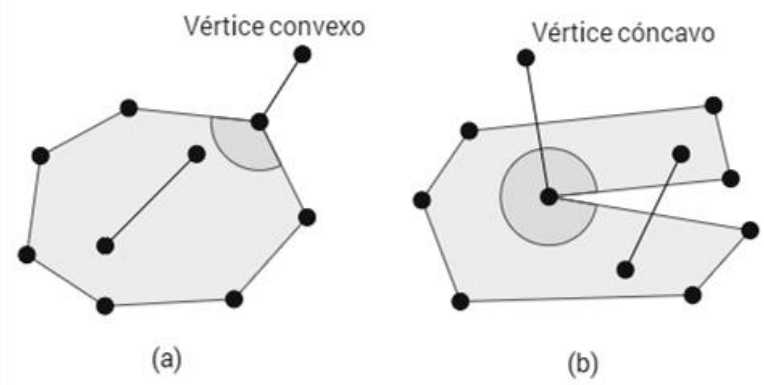


Figura 2.7: (a) Muestra un polígono convexo, el segmento de línea que conecta a cualquier par de puntos del polígono debe estar totalmente contenida en el polígono. (b) Si dos puntos pueden ser encontrados tales que el segmento está parcialmente contenido fuera del polígono, el polígono es cóncavo.

2.9. Volúmenes envolventes.

Realizar la prueba de superposición entre dos objetos puede ser costoso, especialmente cuando los objetos están formados de cientos de polígonos; para optimizar este costo se utiliza un volumen envolvente del objeto, el cual es usado para realizar la prueba de superposición antes de realizar pruebas de intersecciones geométricas.

Un volumen envolvente es simplemente un volumen que encapsula uno o más objetos, usualmente se utilizan cajas y esferas como volúmenes envolventes. La idea es que estos volúmenes de objetos complejos sean sometidos a pruebas de superposición de bajo costo Figura 2.8.

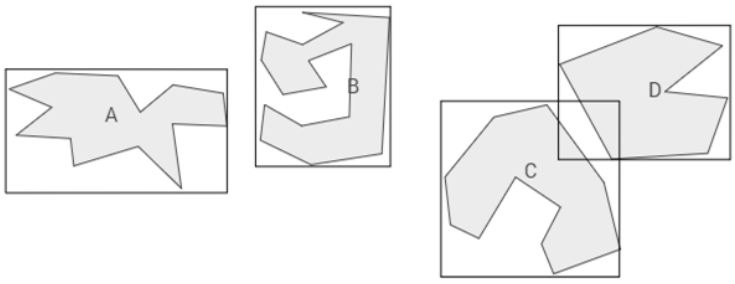


Figura 2.8: El volumen envolvente de A y B no se superponen; por lo tanto, no hay intersección entre A y B. El volumen de C y D puede haber intersección debido a que el volumen envolvente de C y D se superponen.

Los volúmenes envolventes tienen las siguientes propiedades:

- Prueba de intersección de bajo costo.
- Cálculo de bajo costo.
- Fácil de rotar y trasladar.
- Usan poca memoria [12].

Los volúmenes envolventes más usados que cumplen las características mencionadas anteriormente se describen a continuación:

- *Axis-aligned Bounding Boxes* (*AABBs*). Es una caja rectangular de seis lados, caracterizada por tener sus caras orientadas de tal forma que las normales son simultáneamente paralelas a los ejes del sistema de coordenadas. Su mejor característica es que las pruebas de intersección son fáciles de realizar; existen tres formas de representar una *AABB*: almacenar la coordenada mínima y máxima, utilizar una coordenada mínima y el ancho de la caja, por último, utilizar el centro de la caja. Para las dos primeras se deben especificar valores mínimos y máximos en los tres ejes coordenados si estamos en el espacio de tres dimensiones.
- Esferas. Utilizando esferas como volúmenes envolventes se pueden realizar pruebas de intersección con un bajo costo, también tienen la ventaja de ser invariantes a la rotación, haciendo que sea muy fácil aplicarles una transformación.
  Una esfera envolvente es la más eficiente en cuanto a memoria debido a que solo se requiere almacenar el centro y el radio para representar la esfera. La prueba de intersección entre esferas es muy simple, consiste únicamente en comparar la suma de sus radios, esto con el fin de evitar costosas operaciones como la raíz cuadrada.
- Cajas envolventes orientadas (*OBBs*). Una caja envolvente orientada es una caja *AABB* pero puede tener cualquier orientación; existe una variedad de formas para representar una caja orientada como es: un arreglo de ocho vértices, un arreglo de seis planos, una colección de planos paralelos, etc.

2.10. Diagrama de Voronoi.

Imagínese la situación en la que se desea abrir una nueva oficina de servicio postal en una ciudad, tomando en cuenta que existen lugares que proporcionan el mismo tipo de servicio a los cuales se les denominan sitios. Primero se tiene que realizar un análisis de la

rentabilidad de establecer la oficina postal haciendo algunas suposiciones relacionadas con la calidad del servicio que reciben los consumidores:

- El precio del servicio es el mismo que en otros sitios.
- El costo de obtener un buen servicio es igual al precio del servicio, más el costo de transportación al sitio.
- El costo de transportación al sitio es igual a la distancia euclidiana entre el lugar de origen y el sitio de la oficina postal.
- Los consumidores tratan de minimizar el costo de adquirir un buen servicio.

Normalmente estas suposiciones no son del todo acertadas, ya que algunos sitios son más baratos que otros y el costo de transportación entre los puntos probablemente no sean líneas rectas; sin embargo, se puede dar un modelo aproximado que indique las áreas de preferencias.

Este análisis conlleva a obtener una interpretación geométrica, la cual es un modelo que subdivide las áreas comerciales de los sitios, denominadas regiones, de tal forma que las personas que vive en la misma región sean consumidoras de un sitio, las suposiciones implican que las personas simplemente decidan ir al sitio más cercano.

De lo anterior se puede concluir que existe una región para un sitio que contiene a todos los puntos que son más cercanos al sitio y no a otro, a esta región se le conoce como región de Voronoi, la subdivisión que resulta de este modelo se le denomina diagrama de Voronoi. A partir del diagrama de Voronoi se puede obtener todo tipo de información acerca de las áreas comerciales y su relación; por ejemplo, si las regiones de dos sitios comparten una frontera común, estos dos sitios estarán compitiendo directamente por los consumidores que viven en la frontera de la región [13].

Se define la distancia euclidiana entre dos puntos p y q por:

$$dist(p,q) = \sqrt{(p_x - q_x)^2 + \left(p_y - q_y\right)^2} \tag{2.31}$$

Sea el conjunto $P \coloneqq \{p_1, p_2, \ldots, p_n\}$ de n distintos puntos en el plano, a estos puntos se les llaman sitios. Se define un diagrama de Voronoi de P como la subdivisión del plano en n celdas, una para cada sitio de P, con la propiedad que un punto q caen dentro de la celda correspondiente al sitio p_i, si y solo si $dist(q,p_i) < dis(q,p_j)$ para cada $P_j \in P$ con $j \neq i$.

Se denota el diagrama de Voronoi de P por $Vor(p)$, el cual es usado para referirse a los vértices y aristas de la subdivisión. La celda de Voronoi de p_i se denota por $V(p_i)$ [13].

Antes de ver como luce un diagrama de Voronoi, se analiza el caso más básico, el cual cosiste de dos puntos p y q en el plano. Se define un bisector de p y q como una línea perpendicular al segmento $\overline{pq}$ que está a la misa distancia de p y q. Este bisector separa al plano en dos semiplanos, al semiplano que contiene a p se denota por $h(p,q)$ y el semiplano que contiene a q por $h(q,p)$.

Es importante mencionar que $r \in h(p,q)$ si y solo si $dist(r,p) < dist(r,q)$. Tomando en cuenta estas definiciones, se llega a la siguiente observación $V(p_i) = \cap_{1 \le j \le n, j \ne i} h(p_i, p_j)$. Esto significa que $V(p_i)$ es la intersección de $n-1$ semiplanos y por lo tanto un polígono convexo abierto delimitado de al menos $n-1$ vertices y a lo más $n-1$ aristas, algunas aristas son segmentos de líneas y otras son rayos o semirrectas [13].

Sea un conjunto P de n sitios en el plano; si todos los sitios son colineales entonces $Vor(P)$ está formado de $n-1$ lineas paralelas, en caso contrario $Vor(P)$ es conexo y sus aristas son segmentos de líneas o semirrectas [13]. La Figura 2.9 muestra los elementos que forman un diagrama de Voronoi.

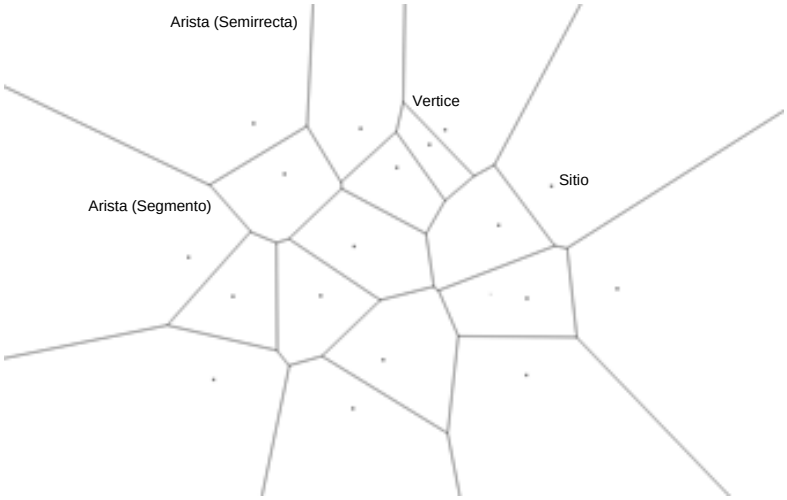


Figura 2.9: Diagrama de Voronoi.

Un diagrama de Voronoi $Vor(P)$ de un conjunto de puntos P tiene las siguientes propiedades:

- Un punto q es un vértice de $Vor(p)$ si y sólo si el circulo $C_p(q)$ contiene tres o más sitios sobre su frontera y no contiene a ningún otro sitio en su interior.
- El bisector entre los sitios p_i y p_j definen una arista de $Vor(p)$ si y sólo si existe un punto q sobre el bisector tal que $C_p(q)$ contiene a $p_i \, y \, p_j$ sobre su frontera y no contiene a otro sitio en su interior [13].

Los diagramas de Voronoi tienen múltiples aplicaciones como en física, computación gráfica, robótica, etc. En el área de conocimientos de robots móviles los diagramas de Voronoi son usados para la planeación de movimientos de un robot; utilizando un modelo tradicional, el robot necesita tener la representación del mundo, una de las formas es utilizando formas geométricas básicas con el fin de obtener el espacio libre y el espacio ocupado. Una vez determinado el espacio por donde el robot puede navegar, se obtiene la secuencia de posiciones que debe visitar el robot para llegar de un punto a otro.

Por lo que se refiere a los objetos, estos son representados por primitivas geométricas básicas, las cuales son puntos y líneas. Durante el análisis realizado previamente en la definición de un diagrama de Voronoi y sus propiedades, únicamente se utilizaron puntos en el plano llamados sitios; sin embargo, un diagrama Voronoi puede ser extendido a puntos y líneas Figura 2.10.

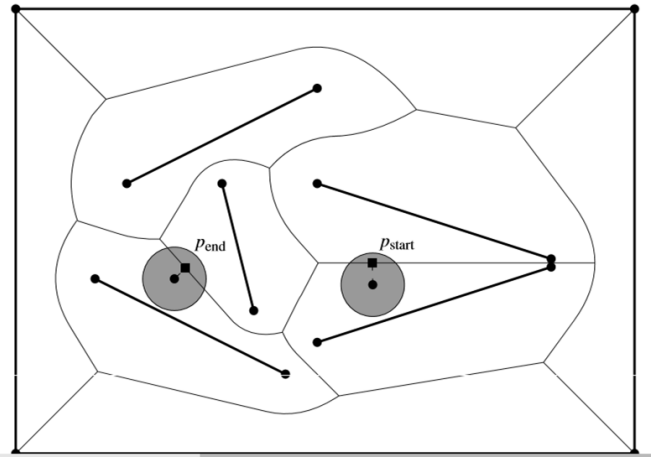


Figura 2.10: Diagrama de Voronoi de segmentos de línea [13].

Mientras el bisector de dos puntos es simplemente una línea, el bisector de dos segmentos de línea disjuntos tiene una forma más compleja. Éste consiste de siete partes, donde cada parte puede ser un segmento de línea o un arco parabólico. La Figura 2.11 muestra las partes de un bisector de dos segmentos de línea.

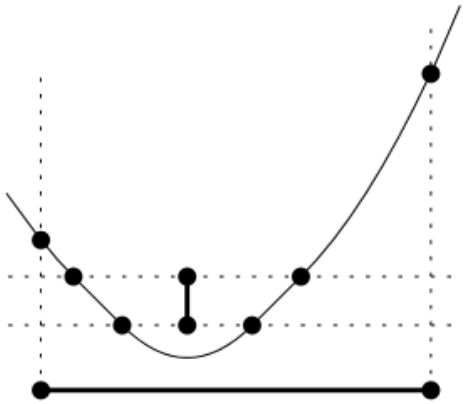


Figura 2.11: Bisector de dos líneas, cada parte puede ser o un segmento de línea o un arco parabólico [13].

Capítulo 3. Herramientas de Software.

3.1. Sistema operativo para robots.

Sistema operativo para robots (ROS) es un marco de trabajo que frecuentemente es usado en robótica. La filosofía de ROS es hacer programas modulares que puedan ser usados en otros robots haciendo pequeños cambios en el código. Esto significa que se puede compartir y usar funcionalidades en otros robots sin tener que reinventar la rueda.

ROS fue desarrollado originalmente en 2007 por Stanford Artificial Inteligence Laboratory (SAIL) con el apoyo de proyectos Stanford AI Robot. Desde el 2008 el principal desarrollador es Willow Garage, un instituto de investigaciones en robótica con más de 20 institutos colaborando entre sí para darle soporte a las nuevas versiones [14].

Una gran cantidad de institutos de investigación ha empezado a desarrollar proyectos en ROS, agregando hardware y compartiendo sus códigos a la comunidad de investigación en robótica, así como compañías han empezado a utilizar ROS en sus productos comerciales. El software del robot Justina ya se encuentra desarrollado en ROS. En cuanto a los sensores y actuadores que son usados en robots, ROS tiene software dedicado al control de estos dispositivos, con el paso del tiempo aumenta el número de dispositivos compatibles con ROS [14].

ROS proporciona un sistema de operación estándar que facilita la abstracción de hardware, control de dispositivos, implementación de funcionalidades usadas comúnmente y una arquitectura que permite la comunicación de procesos por medio de mensajes. Como se mencionó anteriormente ROS cuenta con una arquitectura centralizada donde los procesos se ejecutan en nodos que pueden recibir o solicitar datos como: múltiples lecturas de sensores, sistema de control, máquina de estados, planeación, actuadores, etc.

ROS es un sistema compatible con sistemas *UNIX*, oficialmente el sistema operativo empleado para alojar a ROS es *Ubuntu*, aunque se puede utilizar otros sistemas *UNIX* no se recomienda utilizarlos. Para el presente trabajo de investigación se utilizó la versión del sistema operativo *Linux Ubuntu* 14.04.2 y la versión de *ros-indigo* [14].

Los paquetes *-ros-pkg están dentro de un repositorio para el desarrollo de librerías; muchas de las librerías y herramientas con las que cuenta ROS tales como librería de navegación o la herramienta de visualización RVIZ se pueden encontrar en este repositorio. Un paquete de ROS es la estructura física de archivos y directorios.

Hay que destacar que el objetivo de utilizar ROS es tener procesos independientes llamados nodos comunicándose con otros nodos por medio del envío de información. ROS proporciona dos tipos de comunicación: síncrona y asíncrona, estos dos tipos de comunicación están directamente relacionados con la sincronización de la comunicación.

En la comunicación asíncrona, el proceso que realiza la petición no espera a que el proceso receptor termine su ejecución, es decir, no existe una respuesta de parte de la ejecución del nodo que atiende la solicitud. Por otro lado se tienen la comunicación síncrona en la cual el nodo que realiza la petición debe esperar hasta que el nodo receptor termine y envié una respuesta del resultado de su operación, a este tipo de comunicación se le conoce como cliente-servidor.

La comunicación asíncrona se realiza por medio de tópicos que son usados sin tener una dirección entre los nodos de comunicación, haciendo que la producción y consumo de los datos esté desacoplada. Un tópico puede tener varios subscriptores, cada tópico está fuertemente ligado al tipo de mensaje de ROS, este tipo mensaje es usado por el productor de mensajes para publicar en este tópico y los nodos llamados subscriptores, reciben el mensaje únicamente si su tipo corresponde al del tópico que está subscrito. Los tópicos son trasmitidos usando los protocolos *TCP/IP* y *UDP*.

Por otro lado, en la comunicación síncrona entre los procesos, el nodo que atiende la petición se le denomina servicio, mientras al nodo que realiza la solicitud se le llama cliente. Del mismo modo que los tópicos, a los servicios también se le asocia un tipo, el cual es un archivo de extensión .srv que está alojado en el paquete. En este archivo se define la estructura de los datos que se deseen enviar, así como también los datos que retorne el servicio.

En un sistema demasiado complejo existen casos en los que se requiere enviar una petición a un nodo para la realización de una tarea y recibir una respuesta de la solicitud, lo cual se puede lograr utilizando un servicio; sin embargo, en algunos casos la tarea es demasiado tardada y el cliente necesita estar informado de cuál es el estado de dicha tarea, por lo que se requiere un mecanismo que proporcione la comunicación asíncrona con retroalimentación u obtener el resultado de la tarea. Esto se puede lograr utilizando diferentes tópicos del lado del consumidor para informarle de su ejecución al productor, no obstante ROS proporciona un paquete de nombre *actionlib* que proporciona herramientas para crear servicios cuya ejecuciones sean demasiadas tardadas y así mantener informado al nodo que solicitó realizar la tarea [14].

3.2. El visualizador RVIZ.

Debido a la necesidad de contar con una utilería para observar por medio de la computadora el estado del robot, ROS proporciona una herramienta de nombre RVIZ.

RVIZ se utiliza para desplegar en pantalla el mundo en 3D, por ejemplo: nube de puntos, sensores láser, modelo del robot, etc. Adicionalmente, RVIZ cuenta con una interfaz gráfica con la cual, el desarrollador puede interactuar con opciones para agregar dinámicamente los diferentes elementos, herramienta de medición, diferentes tipos de cámara para poder moverse en el ambiente virtual, posicionamiento del mouse dentro de la ventana de graficación para así realizar una acción [14].

Los elementos gráficos que RVIZ proporciona al desarrollo del presente trabajo se muestran a continuación:

- Markers. Son elementos gráficos tales como puntos, líneas, triángulos, esferas y cubos con el fin de representar una posición en el ambiente virtual.
- RobotModel. Muestra una representación visual del robot en cierta posición.
- Laser Scan. Visualiza los datos obtenidos del sensor láser.
- Map. Muestra un mapa sobre el plano del mundo.
- Path. Muestra la ruta obtenida para la navegación del robot.
- TF. Visualiza la herencia de transformaciones.

Capítulo 4. Simulador de robots móviles.

4.1. Modelo visual del robot.

Un aspecto importante para el simulador es la forma de obtener y visualizar el modelo del robot, el modelo del robot se refiere a sus componentes y su interconexión, por ejemplo: la base móvil, brazo manipulador, cabeza, torso, etc. Para ello se utiliza un componente de ROS que permite especificar por medio de un *XML* cuáles son los componentes que conforman el robot y la dependencia de estos; a este descriptor del modelo del robot se le conoce como archivo *URDF*.

Antes de analizar los componentes y la estructura del modelo de un robot con un archivo *URDF*, se deben conocer los dos elementos básicos que conforman un modelo cinemático de un mecanismo. En primer lugar se tiene los eslabones a los cuales se les conoce como links y a las uniones de los eslabones (articulaciones) se les conoce como joints.

El archivo *URDF* contiene una estructura que define los eslabones que están interconectados entre sí por medio de articulaciones. En la Figura 4.1 se muestra la representación jerárquica de un archivo *URDF*, en la cual un nodo puede tener múltiples nodos dependientes unidos por su respectiva articulación, de este modo se crea una herencia de componentes, los cuales al aplicarles una transformación afecta a los nodos dependientes del nodo padre.

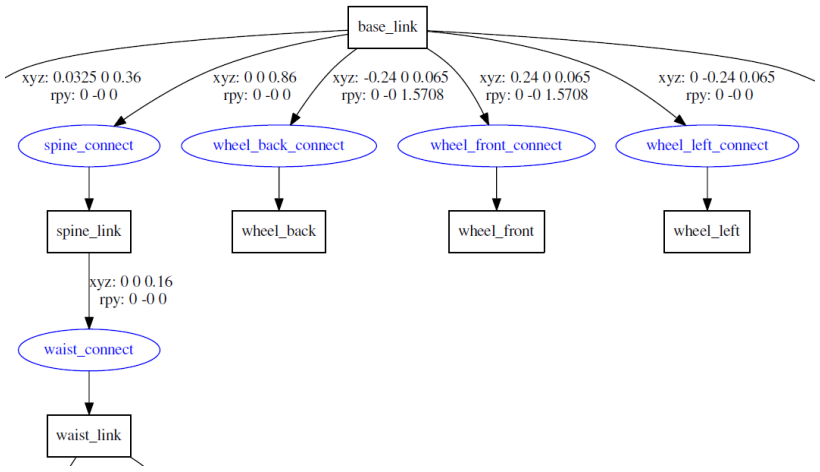


Figura 4.1: Representación jerárquica de la descripción lógica y visual de un modelo URDF.

A los eslabones se le asocia una forma que puede ser: una caja, cilindro, esfera o malla, la cual es la forma geométrica que RVIZ dibuja en su panel de visualización.

El archivo *URDF* utilizado en esta investigación es el que define la estructura del robot de servicio Justina. Este archivo se toma como base teniendo en cuenta que puede cambiar de tal forma que se ajuste a las necesidades y capacidades del robot que se desee representar.

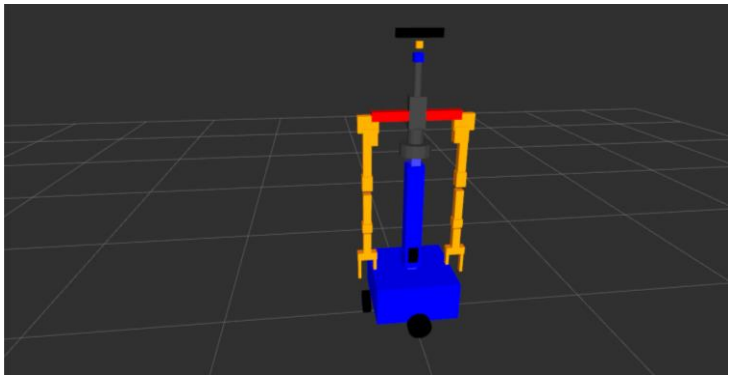


Figura 4.2: Modelo del robot Justina.

Una vez ya creado el modelo del robot, se le indica a RVIZ cuál es el archivo que debe cargar para su visualización, esto se realiza colocando el nombre del archivo *URDF* en un parámetro llamado *robot_description* dentro del archivo de lanzamiento de ROS.

4.2. Representación del mundo.

La representación de los obstáculos en el ambiente virtual debe de ser flexible en cuanto a su configuración, es decir, la distribución espacial de los obstáculos necesita estar encapsulada en un archivo, que sirve para representar el ambiente virtual, en el cual el robot interactúa.

Antes de especificar la forma y estructura que debe tomar esta configuración se necesitan tener en cuenta los siguientes puntos:

- La cinemática de un robot móvil únicamente cuenta con tres grados de libertad: dos para el desplazamiento en el plano $Z = 0$ y uno para el giro sobre el eje Z. De esta forma únicamente se limita a tener figuras geométricas en $2D$ que formen los objetos y así poder determinar el espacio libre y ocupado para la navegación del robot.

- Los objetos tienen la forma de polígonos simples, los cuales están formados por vértices que se interconectan entre sí por medio de segmentos de línea.
- Con el objetivo de obtener la visualización de los obstáculos en tres dimensiones, únicamente se establece una altura en el eje Z.
- Existen dos tipos de obstáculos a los cuales se debe de hacer una distinción: se tienen los objetos y las paredes, esto se realiza debido a que se necesita visualizar de diferente forma los objetos y las paredes, proporcionándoles a estas últimas una altura diferente que a los objetos.

Analizando los puntos mencionados anteriormente basta con tener un arreglo de polígonos cuyo contenido sea un conjunto de vértices que representen el obstáculo (objeto o pared).

Esta representación es utilizada de dos formas distintas: enviar al visualizador RVIZ primitivas gráficas con el fin de obtener la visualización de los objetos en el ambiente virtual y obtener la distribución espacial para utilizarla en la lógica de la simulación.

Con el fin de proporcionar la representación gráfica y lógica de la distribución espacial de los obstáculos, el simulador cuenta con un nodo de ROS con las siguientes características:

- Realiza el parseo del archivo almacenando en memoria las posiciones de cada vértice que forman al obstáculo.
- RVIZ cuenta con diferentes tipos de primitivas visuales, la más adecuada para la representación de polígonos con alturas son triángulos; por lo tanto, este nodo es el encargado de publicar las primitivas conforme a la estructura que tiene el ambiente virtual.
- Debido a que otros nodos necesitan la información de distribución espacial de los objetos, es necesario contar con servicio de ROS que proporcione la posición de los obstáculos en el ambiente virtual; por ejemplo, el nodo encargado de obtener las lecturas de los sensores utiliza la información para determinar la distancia a la que se encuentran los obstáculos del robot.

Es preciso destacar que los obstáculos están definidos por polígonos no convexos en el plano $z = 0$, un ejemplo de cómo se visualizan los obstáculos en el plano $z = 0$ se muestra en la Figura 4.3. El objetivo es crear una representación de éstos aumentándoles una altura y así tener una apariencia visual en tres dimensiones. No obstante esta representación puede ser de utilidad para otras funciones que debe realizar el robot móvil, como por ejemplo, la simulación del reconocimiento y manipulación de objetos. Estos algoritmos se basan en encontrar los planos que ayudan a la segmentación de los objetos por encima de la mesa, en el presente trabajo no se consideran la simulación de los algoritmos de reconocimiento de objetos y manipulación, pero se plantea como un problema a resolver como trabajo a futuro.

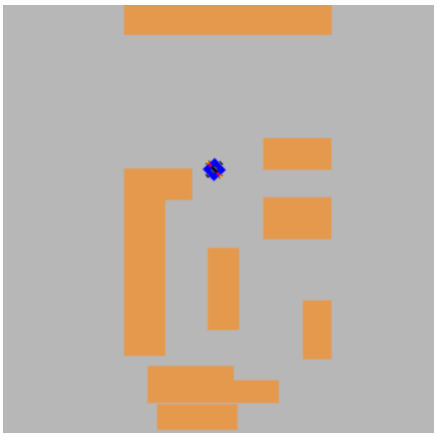


Figura 4.3: Polígonos del simulador vistos desde una perspectiva ortogonal.

La creación de los triángulos que forman los objetos, se realiza tomando como entrada los polígonos que los definen, la idea principal consiste en obtener una proyección de los vértices de cada polígono por encima del plano $z = 0$, es decir, la componente en z es igual a la altura que define el obstáculo.

Una vez obtenido el conjunto de vértices de los obstáculos, lo siguiente es crear los triángulos que conforman la tapa inferior y la tapa superior. Es importante mencionar que únicamente se tienen los vértices de los polígonos por lo que se necesita realizar una triangulación de estos vértices y así obtener figuras solidas en el ambiente gráfico.

Antes de proporcionar el algoritmo que construye la triangulación del conjunto de vértices que definen a los polígonos, se dará la definición formal del término triangulación.

Sea P un polígono simple con n vértices, una diagonal es un segmento que conecta a dos vértices de p y está contenido en el interior de p. Una descomposición del polígono en triángulos por un conjunto maximal de diagonales que no se intersectan entre si se le llama triangulación del polígono [13]. La Figura 4.4 muestra un polígono simple y una posible triangulación de éste.

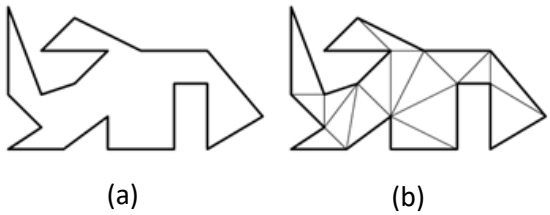


Figura 4.4: (a) Polígono simple, (b) Una posibles triangulación de (a) [13].

Para realizar la triangulación de un polígono simple primero se analiza el caso menos complejo, el cual consiste en la triangulación de un polígono convexo. La triangulación de un polígono convexo se realiza tomando un vértice del polígono y trazando una diagonal a otro vértice a excepción de sus vecinos, de esta forma el algoritmo tiene complejidad lineal. Ahora que pasa con los polígonos cóncavos, para esto se usa un método de geometría computacional que consiste en hacer particiones convexas del polígono simple, es decir, siempre se encuentra subdivisiones convexas del polígono.

Existe un algoritmo llamado aproximación de Greene que permite obtener particiones convexas con una complejidad $O(nlogn)$ y espacio $O(n)$ [15]. Este algoritmo utiliza una partición *y-monotone*. Un polígono simple es llamado monótono con respecto a una línea l si para una línea l' perpendicular a l la intersección del polígono con l' es conexa, es decir que la intersección es una segmento de línea, un punto o vacío [13]. Un polígono que es monótono con el eje y es llamado *y-monotone*. Un polígono *y-monotone* tiene la siguiente característica: si se recorre su frontera desde el vértice superior hasta el inferior, entonces siempre se mueve hacia abajo u horizontalmente y nunca hacia arriba [13].

Una vez que se calculan las particiones convexas del polígono, lo siguiente es obtener la triangulación de los vértices de las particiones convexas; estos algoritmos ya están implementados en la librería llamada CGAL. CGAL es una librería *OpenSource* de C++ que proporciona algoritmos de geometría computacional [16].

La implementación del algoritmo que obtiene la triangulación de los polígonos que forman los obstáculos se muestra en Algoritmo 1.

Algoritmo 1 Implementación de la triangulación de los vértices de un arreglo de polígonos.

Entradas

P – Arreglo de polígonos de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

Salidas

$ListaTriangulos$ – Lista de triángulos correspondiente a la triangulación de P.

```
1:  Function GetTrianglesFromPolygons(P).
2:      for i = 1 to N do
3:          Obtener partición y-monotone.
4:          Dada la partición y-monotone obtener las particiones convexas de P.
5:          K ← Número de particiones convexas.
6:          for j = 1 to K do
7:              Computar la triangulación de la partición convexa j.
8:              L ← Número de triángulos de la partición convexa j.
9:              for n = 1 to L do
```

10:	$T \leftarrow$ Triangulo n de la partición convexa j.
11:	Agregar el triángulo T a $ListaTriangulos$.
12:	**return** $ListaTriangulos$

Los triángulos obtenidos del Algoritmo 1 son utilizados para crear las tapas inferiores y superiores de los obstáculos, en primer lugar se tienen las tapas inferiores cuyo valor en su componente en el eje z es igual a cero, mientras la componente z de las tapas superiores es igual a la altura que tiene el obstáculo.

Para crear los triángulos que forman las tapas laterales de los obstáculos, se asume que sus vértices vecinos están ordenados en el arreglo. Como entrada de este algoritmo, se tiene el arreglo de polígonos, cada polígono del arreglo tiene su arreglo de vértices. Para cada polígono se recorre el arreglo de sus vértices creando dos triángulos por cada iteración: el primer triángulo se forma tomando el vértice en el que se encuentre la iteración, el segundo vértice corresponde a su vecino en el arreglo, el tercero es el vértice actual con la componente z igual a la altura del obstáculo.

El segundo triangulo se crea tomando como primer elemento el vértice vecino de la iteración actual, el segundo elemento corresponde a este mismo vértice pero su componente z igual a la altura del obstáculo y el último elemento del triángulo es igual al vértice de la iteración actual con su componente z igual a la altura del obstáculo, es importante mencionar que cuando se haya alcanzado el último elemento del arreglo, el vecino de éste se convierte en el primer elemento del arreglo. Este algoritmo es una especie de cosido entre los vértices de la tapa superior y la tapa inferior. El Algoritmo 2 muestra resumidamente en que consiste este algoritmo.

Algoritmo 2 Implementación del algoritmo para obtener los triángulos de las tapas laterales.

Entradas

P – Arreglo de polígonos de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

Salidas

$ListaTriangulos$ – Lista de triángulos correspondiente a las tapas laterales de los obstáculos.

1:	**Function** getTrianglesLateralPolygons(P).
2:	**for** $i = 1$ to N **do**
3:	$H \leftarrow$ Altura del obstáculo
4:	**for** $j = 1$ to $P[i].vertex\_size$ **do**
5:	$v_1 \leftarrow P[i].vertex[j]$
6:	**if** $j < P[i].vertex\_size$ **then**
7:	$v_2 \leftarrow P[i].vertex[j + 1]$

8: **else**
9: $v_2 \leftarrow P[i].vertex[1]$
10: $e_1 \leftarrow Vertex(v_1.x, v_1.y, 0)$
11: $e_2 \leftarrow Vertex(v_2.x, v_2.y, 0)$
12: $e_3 \leftarrow Vertex(v_1.x, v_1.y, H)$
13: $e_4 \leftarrow Vertex(v_2.x, v_2.y, 0)$
14: $e_5 \leftarrow Vertex(v_2.x, v_2.y, H)$
15: $e_6 \leftarrow Vertex(v_1.x, v_1.y, H)$
16: $t_1 \leftarrow Triangle(e_1, e_2, e_3)$
17: $t_2 \leftarrow Triangle(e_4, e_5, e_6)$
18: Agregar el triángulo t_1 a $ListaTriangulos$
19: Agregar el triángulo t_2 a $ListaTriangulos$
20: **return** $ListaTriangulos$

La aplicación del Algoritmo 1 y Algoritmo 2 se muestra en la Figura 4.5.

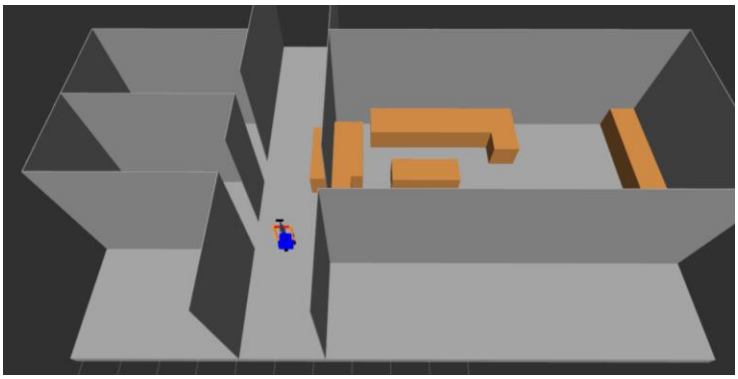


Figura 4.5: Primitivas visuales de los obstáculos del simulador.

4.3. Simulación de hardware.

La simulación de los componentes de hardware depende de la configuración del robot que se desee simular; por ejemplo, un robot puede tener una base holonómica, tener un solo brazo manipulador, etc. La configuración del robot depende de la descripción del robot dada en la sección 4.1, para este trabajo se utiliza el modelo del robot Justina. Es importante mencionar que si se desea realizar la simulación con la configuración de otro robot, se tiene que desarrollar el software encargado de simular sus componentes.

Para utilizar el modelo de par diferencial, considerando las cuatro direcciones de movimiento, se utiliza la ecuación (2.23), en la cual se agregan las contribuciones de las

velocidades lineales de la rueda delantera $\dot{\phi}_f$ y trasera $\dot{\phi}_b$ como se muestra en la ecuación (4.1).

$$\dot{\xi}_I = R(\theta)^{-1} \begin{pmatrix} \frac{r\dot{\phi}_d}{2} + \frac{r\dot{\phi}_i}{2} \\ \frac{r\dot{\phi}_f}{2} + \frac{r\dot{\phi}_b}{2} \\ \frac{r\dot{\phi}_d}{2l} + \frac{-r\dot{\phi}_i}{2l} + \frac{r\dot{\phi}_f}{2l} + \frac{-r\dot{\phi}_b}{2l} \end{pmatrix} \tag{4.1}$$

Para la simulación de la ecuación (4.1), se tiene un nodo de ROS que recibe, por medio de un tópico, las velocidades lineales en cada llanta, se hace el cálculo de cuál es la velocidad lineal y angular del robot obteniendo la posición del robot al aplicarles dichas velocidades en los motores.

El laboratorio de Bio-Robótica de la facultad de Ingeniería me proporcionó el software que simula el funcionamiento de los componentes: brazos manipuladores, cabeza mecatrónica y torso. Esta investigación no detalla la implementación de los componentes, ya que únicamente está orientado a los movimientos del robot; sin embargo, el uso de estos nodos de simulación se plantea como un problema de trabajo a futuro.

4.4. Movimientos del robot.

Una vez que se cuenta con las funciones de más bajo nivel que permite enviarle valores de velocidades lineales a las llantas, se describen funciones de más alto nivel que hacen uso del nodo de ROS detallado en la sección 4.3, estas funciones son encargadas de realizar trayectorias de acuerdo a diferentes tipos de movimientos. Los tipos de movimientos básicos que se implementaron son:

1. Mover el robot una distancia hacia adelante o atrás.
2. Girar en sentido de las manecillas del reloj o en sentido inverso.
3. Moverse a una posición deseada.
4. Desplazarse a un arreglo de posiciones.
5. Desplazarse lateralmente hacia la izquierda o derecha.

Las funciones descritas anteriormente utilizan una ecuación de control para llegar a un punto deseado. Las entradas del sistema de control son: las coordenadas del punto objetivo al cual el robot debe navegar $Pg = [X_g \quad Y_g]^T$.

Las velocidades lineales de la llanta izquierda y derecha se calculan con las ecuaciones (4.2) y (4.3) respectivamente [17].

$$\dot{\phi}_i = v_{max} e^{-\frac{e_a^2}{\alpha}} + \frac{D}{2} w_{max} \left(\frac{2}{1 + e^{-\frac{e_\alpha}{\beta}}} - 1 \right) \tag{4.2}$$

$$\dot{\phi}_d = v_{max} e^{-\frac{e_a^2}{\alpha}} - \frac{D}{2} w_{max} \left(\frac{2}{1 + e^{-\frac{e_\alpha}{\beta}}} - 1 \right) \tag{4.3}$$

$$e_p = P_g - P = [X_g - X \quad Y_g - Y] = \begin{bmatrix} e_x & e_y \end{bmatrix} \tag{4.4}$$

$$e_a = atan2(e_y, e_x) - \theta \tag{4.5}$$

Donde D es el diámetro del robot y $P = [X \quad Y \quad \theta]^T$ es la posición y orientación actual del robot. Las variables v_{max}, w_{max}, β y α son constantes de diseño mayores a cero [17].

Estas ecuaciones únicamente se aplican cuando no son movimientos laterales, es decir para todos excepto el último tipo de movimiento, el cual utiliza las llantas delantera y trasera. El control de velocidades que se requiere es idéntico al que se presentó en las ecuaciones (4.2) y (4.3); pero considerando ahora las llantas delantera y trasera como se muestra en las ecuaciones (4.6) y (4.7).

$$\dot{\phi}_b = v_{max} e^{-\frac{e_a^2}{\alpha}} + \frac{D}{2} w_{max} \left(\frac{2}{1 + e^{-\frac{e_\alpha}{\beta}}} - 1 \right) \tag{4.6}$$

$$\dot{\phi}_f = v_{max} e^{-\frac{e_a^2}{\alpha}} - \frac{D}{2} w_{max} \left(\frac{2}{1 + e^{-\frac{e_\alpha}{\beta}}} - 1 \right) \tag{4.7}$$

Las ecuaciones de control que se mostraron anteriormente tienen como entrada la posición objetivo del robot, para los tipos de movimientos 3 y 4 es trivial ya que se asume que la posición o posiciones deseadas se envían como parámetro; sin embargo, para los casos 1, 2 y 5 se debe calcular en base a los parámetros de entrada, los cuales son: distancia, ángulo

de rotación y si el movimiento es lateral. Las ecuaciones (4.8) y (4.9) muestran la forma de calcular la posición deseada $Pg = [\,X_g \quad Y_g]^T$ cuando se usan las llantas laterales.

$$X_g = X + d * \cos(\theta + \theta') \tag{4.8}$$

$$Y_g = Y + d * \sin(\theta + \theta') \tag{4.9}$$

Donde d es la distancia que avanza el robot, θ' el giro y $P = [X \quad Y \quad \theta]^T$ es la posición y orientación actual del robot.

Para los movimientos laterales se tiene que hacer una rotación de noventa grados siguiendo la regla de la mano derecha obteniendo las siguientes ecuaciones:

$$X_g = X + d * \cos\left(\theta + \frac{\pi}{2}\right) \tag{4.10}$$

$$Y_g = Y + d * \sin\left(\theta + \frac{\pi}{2}\right) \tag{4.11}$$

4.5. Colisión del robot.

La detección de colisiones es parte fundamental de los sistemas de realidad virtual, animación por computadora, modelado de sistemas físicos, visualización científica, simuladores de robots y desarrollo de videojuegos.

La detección de colisiones consiste en realizar pruebas para determinar si dos objetos se superponen. Existen múltiples algoritmos para la detección de colisiones ya sea para dos dimensiones o tres dimensiones, siendo algunos de los algoritmos empleados en dos dimensiones una particularidad de los algoritmos en tres dimensiones. Como se ha mencionado anteriormente, el robot únicamente se desplaza en el plano $z = 0$ y es por ello que los algoritmos utilizados para la detección de colisiones se realizan en dos dimensiones.

Para saber que algoritmos deben ser implementados se analizan las diferentes configuraciones que puede tener el robot; el robot puede tener cualquier forma y dimensión, por lo que se plantean formas geométricas que se ajusten a cualquier configuración de la base del robot que es la encargada de limitar los movimientos. Las formas que se ajustan al modelo

del robot son círculos o cajas envolventes. Los círculos son empleados para representar las bases cilíndricas, mientras las cajas envolventes para representar las bases cuadradas o cualquier otra configuración geométrica.

Por otro lado, se requiere saber contra qué se debe realizar la prueba de intersección; para esto se analiza de qué forma está representado el mundo, el cual se detalló en el capítulo 4.2. En esta representación del mundo, los obstáculos están formados por polígonos simples. Existen varios algoritmos que determinan de manera eficiente si existe colisión con polígonos convexos; sin embargo, esta condición de convexidad no se cumple y por lo tanto se plantea una prueba de intersección considerando las aristas de los polígonos.

Las pruebas de colisión entre el robot y los obstáculos se realiza con las aristas de los polígonos y la forma geométrica del robot, como se observa en la Figura 4.6; por lo que los algoritmos que se detallan a continuación son: las pruebas de intersección circulo vs segmento y caja vs segmento.

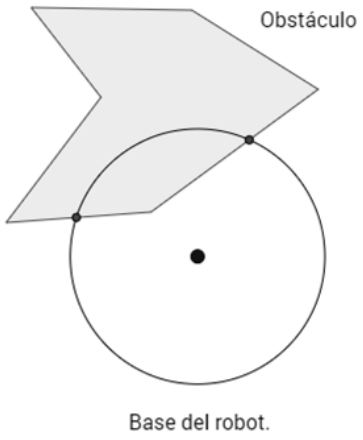


Figura 4.6: Intersección de la base del robot y un obstáculo.

Primeramente se describe la prueba de intersección entre un círculo y un segmento. Un círculo está definido por la ecuación (4.12), donde r es el radio del círculo y h, k son las coordenadas de su centro.

$$(x-h)^2 + (y-k)^2 = r^2 \tag{4.12}$$

En la sección 2.6 se definió una recta por su ecuación paramétrica. Descomponiendo la ecuación (2.25) en cada uno de sus términos, se tiene como resultado las ecuaciones (4.13) y (4.14).

$$x = dx * t + x_0 \tag{4.13}$$

$$y = dy * t + y_0 \tag{4.14}$$

Donde $dx = x_1 - x_0, dy = y_1 - y_0$.

Si existe intersección, las ecuaciones (4.13) y (4.14) debe satisfacer a (4.12); por lo tanto, sustituyendo las ecuaciones en (4.12) y agrupando los términos se tiene la ecuación (4.15).

$$(dx^2 + dy^2)t^2 + \big(2dx(x_0 - h) + 2dy(y_0 - k)\big)t + (x_0 - h)^2 + (y_0 - k)^2 - r^2 = 0 \tag{4.15}$$

La ecuación (4.15) tiene la forma de la ecuación general de segundo grado.

$$at^2 + bt + c = 0 \tag{4.16}$$

$$a = (dx^2 + dy^2) \tag{4.17}$$

$$b = 2dx(x_0 - h) + 2dy(y_0 - k) \tag{4.18}$$

$$c = (x_0 - h)^2 + (y_0 - k)^2 - r^2 \tag{4.19}$$

La solución de la ecuación (4.16) está dada por:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{4.20}$$

El discriminante $d = b^2 - 4ac$ determina cuántos valores reales puede tener la solución. Si $d < 0$, ésta no tiene ningún valor real; por lo tanto, el circulo no intersecta a la recta. Si $d = 0$, entonces solo existe una solución; por lo tanto, la línea es tangente al círculo. Si $d > 0$, entonces hay dos soluciones reales, es evidente que para este caso existen dos coordenadas de intersección de la recta con el círculo. La Figura 4.7 muestra los casos del discriminante d. Encontrando la solución de la ecuación (4.15) se obtiene el parámetro t, el cual se utiliza para obtener las coordenadas x, y con las ecuaciones (4.13) y (4.14).

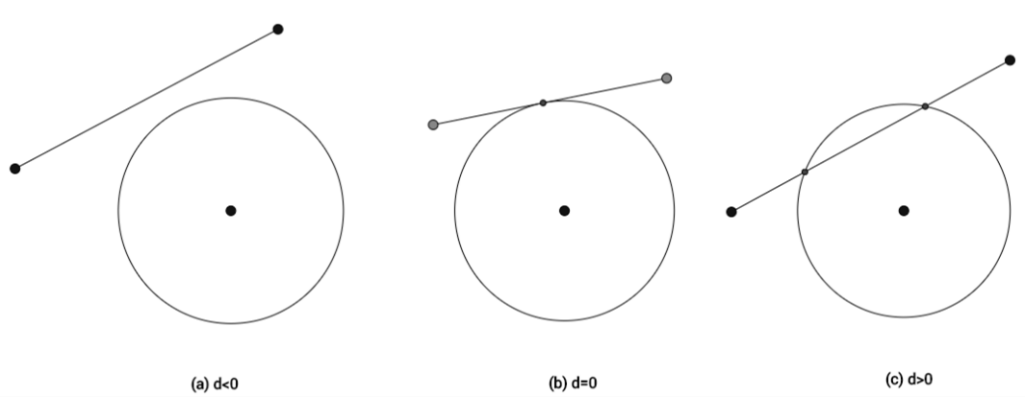


Figura 4.7: (a) d<0 no existe intersección, (b) d=0 la recta es tangente al círculo, (c) d> 0 la ecuación (3.14) tiene dos soluciones por la tanto existen dos intersecciones.

Debido a que el objetivo es realizar la prueba de intersección con un segmento de línea, se necesita realizar un paso extra, el cual consiste en validar si algún punto de intersección está contendió en el segmento, este algoritmo se basa en el hecho de que el punto está sobre la línea y el segmento es una parte acotada de la recta; por lo tanto, la prueba consiste en definir una caja envolvente únicamente con sus coordenadas $x_{min}, x_{max}, y_{min}, y_{max}$ que corresponden al mínimo y máximo de la coordenadas x, y de los puntos que definen el segmento, como se muestra en la Figura 4.8. Una vez ya definido estos valores sólo queda validar si el punto está contenido en ese intervalo. El Algoritmo 3 muestra cómo se realiza esta prueba.

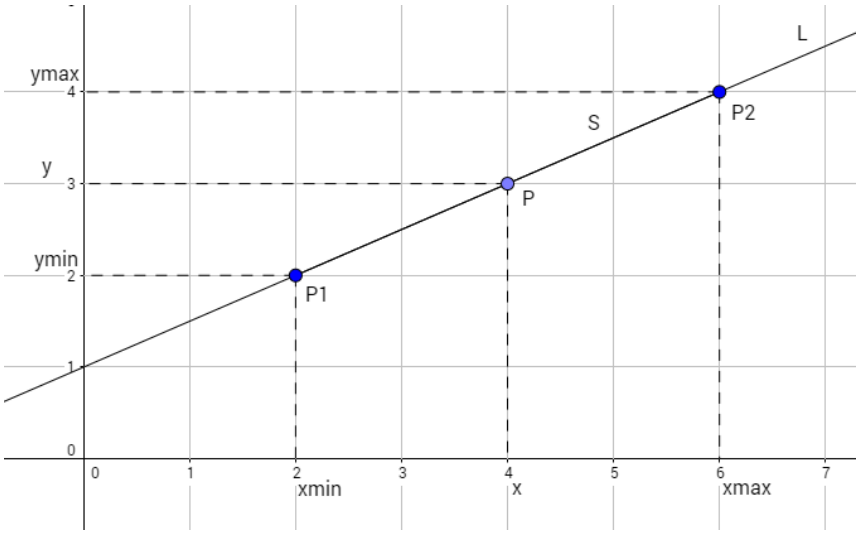


Figura 4.8: Prueba para determinar si el punto P que pertenece a la recta L, está contenido en el rango del segmento S.

Algoritmo 3 Prueba para determinar si el punto P que pertenece a la recta L está contenido en el rango del segmento S.

Entradas

S – Segmento de recta definido por P_1 y P_2.
P – Punto sobre la línea L, L se forma con P_1y P_2.

Salidas

B – Bandera que indica si P pertenece a S.

1: **Function** isInRange(P, S)
2: **if** $S.P_1.x < S.P_2.x$ **then**
3: $x_{min} \leftarrow S.P_1.x$
4: $x_{max} \leftarrow S.P_2.x$
5: **else**
6: $x_{min} \leftarrow S.P_2.x$
7: $x_{max} \leftarrow S.P_1.x$
8: **if** $S.P_1.y < S.P_2.y$ **then**
9: $y_{min} \leftarrow S.P_1.y$
10: $y_{max} \leftarrow S.P_2.y$
11: **else**
12: $y_{min} \leftarrow S.P_2.y$
13: $y_{max} \leftarrow S.P_1.y$
14: **if** $P.x \geq x_{min}$ ***and*** $P.x \leq x_{max}$ ***and*** $P.y \geq y_{min}$ ***and*** $P.y \leq y_{max}$ **then**
15: $B \leftarrow$ **true**
16: **else**
17: $B \leftarrow$ **false**
18: **return** B

El Algoritmo 4 describe resumidamente en que consiste la prueba de intersección de un círculo y un segmento de línea.

Algoritmo 4 Algoritmo para determinar si un circulo y una recta se intersectan.

Entradas

C – Circulo definido por su centro C_i y radio r.

S – Segmento de recta definido por P_1y P_2.

Salidas

B – Bandera que indica si existe intersección entre C y S.

Function testSegmentCircleIntersect(C, S)
  $dx \leftarrow S.P_2.x - S.P_1.x$
  $dy \leftarrow S.P_2.y - S.P_1.y$
  $h \leftarrow C.C_i.x$
  $k \leftarrow C.C_i.y$
  $r \leftarrow C.r$
  $x_0 \leftarrow S.P_1.x$
  $y_0 \leftarrow S.P_1.y$
  $a \leftarrow dx^2 + dy^2$
  $b \leftarrow 2 * dx * (x_0 - h) + 2 * dy * (y_0 - k)$
  $c \leftarrow (x_0 - h)^2 + (y_0 - k)^2 - r^2$
  $disc = b^2 - 4ac$
  if $disc \geq 0$ **then**
    $t_1 \leftarrow \frac{-b+\sqrt{disc}}{2a}$
    $t_2 \leftarrow \frac{-b-\sqrt{disc}}{2a}$
    $xint_1 \leftarrow dx * t_1 + x_0$
    $yint_1 \leftarrow dy * t_1 + y_0$
    $xint_2 \leftarrow dx * t_2 + x_0$
    $yint_2 \leftarrow dy * t_2 + y_0$
    if $disc > 0$ **then**
      $B \leftarrow isInRange(P(xint_1, yint_1), S)$
      if $B =$ **false then**
        $B \leftarrow isInRange(P(xint_2, yint_2), S)$

  else
    $B \leftarrow$ **false**
  return B

El siguiente tipo de colisión que se describe a continuación, es la intersección entre un segmento de recta y una caja envolvente alineada con los ejes coordenados que fue detallada

en la sección 2.9. La caja envolvente representa a la base del robot y el segmento de línea a cada una de las aristas que forman los polígonos.

En primer lugar se dará una definición del termino slab como el espacio entre un par de planos paralelos. A partir de esta definición, el volumen rectangular de una caja puede ser visto como la intersección de tres slabs formando ángulos rectos entre ellos. Un punto está contenido en la caja si y sólo si éste se encuentra dentro de los tres slabs; por su parte, una semirrecta intersecta a la caja si y sólo si la intersección entre la semirrecta y todos los slabs se superponen, el rayo no puede estar dentro de los slabs al mismo tiempo y por lo tanto no existe intersección con el volumen formado por la intersección de los slabs [12], este principio se aplica para la intersección de una semirrecta con una caja. La Figura 4.9 muestra las condiciones que se cumplen para la intersección de una semirrecta y una caja en dos dimensiones. En el caso de dos dimensiones únicamente se tiene dos slabs; por lo tanto, la condición se cumple cuando la intersección entre el segmento y los slabs x, y se superponen.

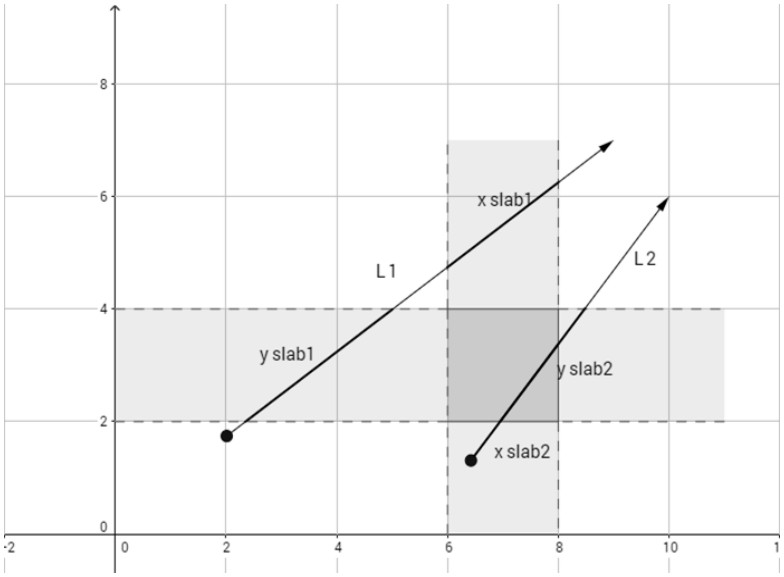


Figura 4.9: La semirrecta L_1 no intersecta la caja debido a que la intersección con x slab1 así como y slab2 no se superpone. La semirrecta L_1 intersecta a la caja debido a que la intersección de sus slabs se superpone.

De lo anterior se puede deducir que la prueba de intersección de una semirrecta y una caja consiste en obtener los intervalos de intersección de la semirrecta con los planos de los slabs y realizar operaciones simples de comparación; pero, se necesita verificar si existe intersección con todos los intervalos para que el segmento de recta y la caja se intersecten.

La intersección de los intervalos del segmento de recta con los slabs se obtiene con la intersección de la semirrecta y los planos de los slabs. Tomando la ecuación paramétrica de

la semirrecta (2.25), junto con la ecuación del plano (2.29) y resolviendo para t, se obtiene, como resultado la ecuación (4.21).

$$t = \frac{d - P * n}{v * n} \tag{4.21}$$

Analizando las propiedades de una caja envolvente alineada con sus ejes coordenados se sabe que dos de sus componentes de la normal del plano que definen las caras son cero, adicional se sabe que $P = (p_x, p_y, p_z)$ y $v = (v_x, v_y, v_z)$, considerando el plano que es perpendicular al eje x queda simplificada la ecuación (4.21) como $t = \frac{d - p_x}{v_x}$, donde d corresponde a la posición del plano a lo largo del eje x [12]. Para evitar la división entre cero, cuando la semirrecta es paralela a un slab, se maneja el caso sustituyendo esta condición por la de verificar si el origen de la semirrecta está contenida en el slab. El Algoritmo 5 muestra lo descrito anteriormente.

Algoritmo 5 Algoritmo para determinar si las intersecciones de la semirrecta con los slab se superpone.

Entradas

p – Componente del origen del segmento.
v – Componente de la dirección de la recta.
min – Componente mínimo de la caja.
max – Componente máximo de la caja.
$tmin$ – Valor mínimo de intersección con algún slab.
$tmax$ – Valor máximo de intersección con algún slab.

Salidas

B – Bandera que indica si existe superposición entre la semirrecta y el slab.

```
1:  Function testSLABPlane(p, v, min, max, tmin, tmax)
2:      if abs(v) < 0.01 then
3:          if p ≥ min and p ≤ max then
4:              B ← true
5:          else
6:              B ← false
7:          return B
8:      ood ← 1/v
9:      t1 ← (min − p) * ood
10:     t2 ← (max − p) * ood
11:     if t1 > t2 then
12:         aux ← t1
```

```
13:        t1 ← t2
14:        t2 ← aux
15:    if t1 > tmin then
16:        tmin ← t1
17:    if t2 < tmax then
18:        tmax ← t2
19:    if tmin > tmax then
20:        B ← true
21:    else
22:        B ← false
23:    return B
```

Para que exista la intersección entre la caja y la semirrecta se deben cumplir las condiciones en los ejes x, y utilizando el Algoritmo 5. Como resultado de este algoritmo se obtiene $tmin$, cuyo valor es el valor t de la ecuación (2.25) y el cual es empleado para encontrar el punto más cercano de intersección entre la semirrecta y la caja envolvente. Este algoritmo funciona tomando en cuenta una semirrecta y una caja; sin embargo, se desea obtener la prueba de intersección con un segmento de recta y una caja, para ello se tiene que agregar otra condición más, la cual consiste en validar que el punto de intersección de la semirrecta pertenezca al segmento que la define, que ya se describió en el Algoritmo 3. El Algoritmo 6 muestra las condiciones que se deben de cumplir para que exista intersección entre un segmento y una caja envolvente alineada con los ejes.

Algoritmo 6 Algoritmo para determinar si existe intersección entre un segmento de recta y una caja envolvente alineada con los ejes.

Entradas

P_1 – Punto origen que define el segmento.
P_2 – Punto fin que define el segmento.
A – AABB.

Salidas

B – Bandera que indica si existe intersección entre un segmento de recta y una caja envolvente alineada con los ejes.

```
1:  Function intersectSegmentAABB(P1, P2, A)
2:      tmin ← −FLT_MAX
3:      tmax ← FLT_MAX
4:      D ← P2 − P1
5:      if testSLABPlane(P1.x, D.x, A.minx, A.maxx, tmin, tmax) = false then
6:          return false
7:      if testSLABPlane(P1.y, D.y, A.miny, A.maxy, tmin, tmax) = false then
8:          return false
9:      Q.x ← P1.x + D.x ∗ tmin
```

10: $Q.y \leftarrow P_1.y + D.y * tmin$
11: $B \leftarrow isInRange(Q, Segment(P_1, P_2))$
12: **return** B

Existe otro proceso que se tiene que realizar ya que se dijo anteriormente que la caja alineada con los ejes representa al robot, pero el robot tiene orientación lo cual hace que la caja ya no cumpla esta condición. Para poder utilizar el Algoritmo 5 y el Algoritmo 6, es un requisito que la caja esté alineada con los ejes, lo cual implica que los puntos que forman los segmentos estén transformados del sistema de coordenadas del mundo al sistema de coordenadas del robot.

A continuación en el Algoritmo 7 se muestra el método que se utilizó para detectar las colisiones del robot con los obstáculos, tomando en cuenta el proceso de transformación del sistema de coordenadas del mundo al sistema de coordenadas del robot.

Algoritmo 7 Algoritmo para detección de colisiones del robot con los obstáculos.

Entradas

P – Arreglo de polígonos de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.
O – Matriz de orientación del robot.
V – Posición del robot.
W – Dimensión del robot en el eje x.
H – Dimensión del robot en el eje y.

Salidas

B – Bandera que indica si existe colisión entre el robot y los obstáculos.

1: **Function** testAABBWithPolygons(P, O, V, W, H)
2: $C = O^{-1} * V$
3: **for** $i = 1$ to N **do**
4: **for** $j = 1$ to $P[i].vertex\_size$ **do**
5: $v_1 \leftarrow P[i].vertex[j]$
6: **if** $j < P[i].vertex\_size$ **then**
7: $v_2 \leftarrow P[i].vertex[j + 1]$
8: **else**
9: $v_2 \leftarrow P[i].vertex[1]$
10: $v_1 \leftarrow O^{-1} * v_1$
11: $v_2 \leftarrow O^{-1} * v_2$
12: $minx \leftarrow C.x - \frac{W}{2}$
13: $maxx \leftarrow C.x + \frac{W}{2}$
14: $miny \leftarrow C.y - \frac{H}{2}$
15: $maxy \leftarrow C.y + \frac{H}{2}$

```
            B = intersectSegmentAABB(v1, v2, AABB(minx, maxx, miny, maxy))
            if B = true
                break
        if B = true
            break
    return B
```

4.6. Simulación de sensor láser.

El objetivo de la simulación de un sensor láser es obtener la distancia a la que se encuentran los objetos alrededor de éste. Los objetos en el ambiente virtual están compuestos por polígonos, que a su vez están formados por segmentos de recta; por lo tanto, el método utilizado consiste en trazar múltiples rayos haciendo un barrido circular alrededor del eje z, desde la posición del sensor láser, con el fin de encontrar los segmentos que intersectan con los rayos.

Para realizar el barrido se considera: el número de rayos, ángulo inicial de barrido, rango de apertura, alcance mínimo y máximo. Estos 5 parámetros permiten crear un barrido como el que se muestra en la Figura 4.10.

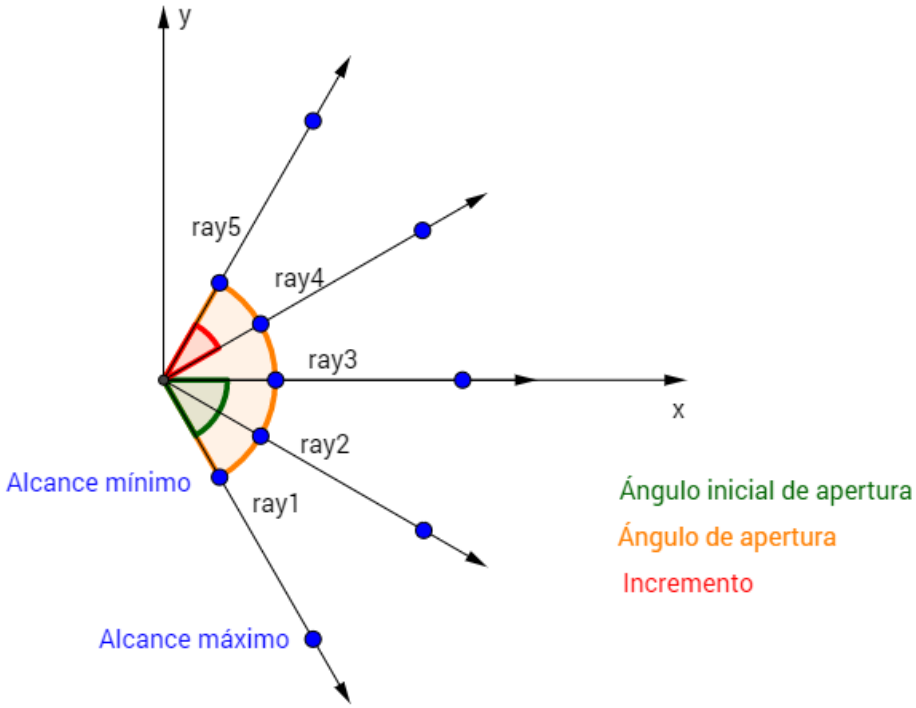


Figura 4.10: Barrido para generar los rayos que simulan un sensor láser.

El incremento del ángulo depende de dos parámetros: el número de rayos y el ángulo de apertura, la ecuación para determinar el incremento se muestra en la ecuación (4.22).

$$inc_{angle} = \frac{range_{angle}}{num_{rays} - 1} \tag{4.22}$$

Es importante mencionar que se maneja una condición para evitar la indeterminación de la ecuación (4.22) cuando $num_{rays} = 1$. En la Figura 4.10 se observa que el rayo se origina en la posición del sensor láser con la orientación definida por el ángulo inicial de barrido, el ángulo de cada rayo es la suma del ángulo acumulado y el valor de inc_{angle}. Los valores de alcance mínimo y máximo, que se usan para obtener los puntos que definen el rango del rayo generado, se encuentran utilizando las ecuaciones (4.23) y (4.24).

$$x = spos_x + rang * \cos(angle_{ray}) \tag{4.23}$$

$$y = spos_y + rang * \sin(angle_{ray}) \tag{4.24}$$

Donde $spos_x, spos_y$ es la posición del sensor, $rang$ es el alcance mínimo o máximo y $angle_{ray}$ es el ángulo del rayo.

Es necesario realizar pruebas de intersección entre las aristas de los polígonos y los rayos generados, ya que cuando existe intersección de una arista y un rayo es posible obtener el punto de intersección y en él medir la distancia euclidiana entre el origen del rayo y el punto de intersección.

Sean los segmentos $L_1 = \{A, B\}$ y $L_2 = \{C, D\}$ como se muestran en la Figura 4.11, existe intersección entre L_1 y L_2 si y sólo si C está a la izquierda y D a la derecha de L_1, además A está a la izquierda y B a la derecha de L_2.

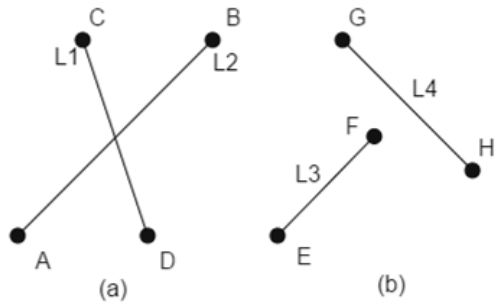


Figura 4.11: (a) Existe intersección entre L_1 y L_2 debido a que D está a la derecha y C a la izquierda de L_2; por su parte, B está a la derecha y A a la izquierda de L_1. (b) No existe intersección entre L_3 y L_4 ya que E y F están a la izquierda de L_4.

El método que determina de qué lado está un punto de un segmento de recta consiste en obtener el signo del determinante de los dos puntos del segmento de recta y el punto en cuestión. Se requiere determinar de qué lado de $\overline{AB}$ se encuentra D; para ello se crean dos vectores $\overline{AB}$ y $\overline{AD}$ como se muestra en la Figura 4.12, se sabe que el producto cruz entre $\overline{AD}$ y $\overline{AB}$ obtiene un vector perpendicular a ambos vectores, tratándose del plano $z = 0$ unicamente este vector cuenta con la componente en k, el signo de k representa el sentido del vector paralelo al eje z, para la operación $\overline{AD}x\overline{AB}$ de los vectores de la Figura 4.12 el signo es positivo; por lo tanto, D está a la derecha de $\overline{AB}$, si se analiza ahora para C el signo de la operación $\overline{AC}x\overline{AB}$ es negativo; por lo tanto, el punto está a la izquierda del segmento $\overline{AB}$.

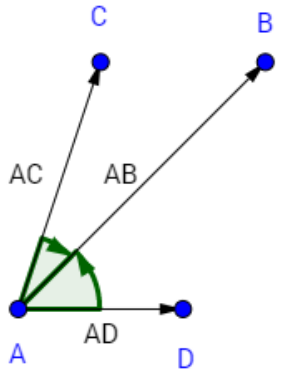


Figura 4.12: La componente k de $\overline{AD}x\overline{AB}$ es positiva; por lo tanto, el punto D está a la derecha de $\overline{AB}$; por su parte, la componente en k de $\overline{AC}x\overline{AB}$ es negativa; por lo tanto, el punto está a la izquierda de $\overline{AB}$.

Esta interpretación es equivalente a obtener el determinante de los tres puntos, esto significa que las ecuaciones son equivalentes entre sí. La ecuación para determinar de qué lado se encuentra un punto de un segmento de recta se muestra en la ecuación (4.25).

$$det = b_x d_y + a_x b_y + a_y d_x - a_y b_x - a_x d_y - b_y d_x \tag{4.25}$$

El algoritmo que determina si dos segmentos se intersectan se muestra resumidamente en el Algoritmo 8.

Algoritmo 8 Algoritmo para detectar la intersección entre dos segmentos.

Entradas

S_1 – Primer segmento para realizar la prueba de intersección, el segmento se define por dos puntos P_1, P_2.

S_2 – Segundo segmento para realizar la prueba de intersección, el segmento se define por dos puntos P_1, P_2.

Salidas

B – Bandera que indica si existe intersección entre dos segmentos.

```
1:  Function testSegmentIntersect(S1, S2)
2:      det1 ← getDeterminant(S1.P1, S1.P2, S2.P1)
3:      det2 ← getDeterminant(S1.P1, S1.P2, S2.P2)
4:      if (det1 < 0 and det2 > 0) or (det1 > 0 and det2 < 0) then
5:          det1 ← getDeterminant(S2.P1, S2.P2, S1.P1)
6:          det2 ← getDeterminant(S2.P1, S2.P2, S1.P2)
7:          if (det1 < 0 and det2 > 0) or (det1 > 0 and det2 < 0) then
8:              B ← true
9:          else
10:             B ← false
11:         return B
12:     if det1 = 0 and det2 = 0 then
13:         B ← true
14:     else
15:         B ← false
16:     return B
```

Cuando la prueba de intersección entre dos segmentos resulta verdadera, se procede a encontrar el punto de intersección, para esto considerar la ecuación (4.26) que define a una recta.

$$Ax + By = C \tag{4.26}$$

Donde:

$$A = y_2 - y_1 \tag{4.27}$$

$$A = x_1 - x_2 \tag{4.28}$$

$$C = Ax_1 + By_1 \tag{4.29}$$

Las ecuaciones (4.30) y (4.31) representan a las dos rectas que se intersectan.

$$A_1x + B_1y = C_1 \tag{4.30}$$

$$A_2x + B_2y = C_2 \tag{4.31}$$

Si existe intersección entre las dos rectas, las ecuaciones (4.30) y (4.31) deben satisfacerse, para obtener las ecuaciones del punto de intersección se considera un sistema de ecuaciones donde en primera instancia se resuelve para x y posteriormente para y obteniendo las siguientes ecuaciones.

$$x = \frac{B_2C_1 - B_1C_2}{A_1B_2 - A_2B_1} \tag{4.32}$$

$$y = \frac{A_1C_2 - A_2C_1}{A_1B_2 - A_2B_1} \tag{4.33}$$

Es importante mencionar que el denominador de las ecuaciones (4.32) y (4.33) es el determinante de las dos rectas, si este determinante es igual a cero significa que las rectas son paralelas.

El Algoritmo 9 muestra brevemente como se realiza la simulación del sensor láser.

Algoritmo 9 Algoritmo para la simulación de sensores láser.

Entradas

num_{rays} – Cantidad de rayos.
$angle_{min}$ – Angulo inicial de barrido.
$range_{min}$ – Alance mínimo.
$range_{max}$ – Alcance máximo.
$range_{angle}$ – Angulo de apertura.
P – Arreglo de polígonos de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.
a – Angulo del sensor láser.
V – Posición del sensor láser.

Salidas

$LaserScan$ – Lecturas del sensor láser.

1: **Function** simulaSensor($num_{rays}, angle_{min}, range_{min}, range_{max}, range_{angle}, P, a, V$)
2: **if** $num_{rays} = 1$ **then**
3: $inc_{angle} \leftarrow 1$
4: **else**
5: $inc_{angle} \leftarrow \frac{range_{angle}}{num_{rays}-1}$
6: **for** $i = 1$ to $num_{rays} - 1$ **do**
7: $ray_{angle} \leftarrow a + (i-1) * inc_{angle} + angle_{min}$
8: $s_x \leftarrow V.x + range_{min} * \cos(ray_{angle})$
9: $s_y \leftarrow V.y + range_{min} * \sin(ray_{angle})$
10: $e_x \leftarrow V.x + range_{max} * \cos(ray_{angle})$
11: $e_y \leftarrow V.y + range_{max} * \sin(ray_{angle})$
12: **for** $j = 1$ to N **do**
13: **for** $k = 1$ to $P[i].vertex\_size$ **do**
14: $v_1 \leftarrow P[j].vertex[k]$
15: **if** $j < P[i].vertex\_size$ **then**
16: $v_2 \leftarrow P[j].vertex[k + 1]$
17: **else**
18: $v_2 \leftarrow P[j].vertex[1]$
19: $S_1 \leftarrow Segment\left(Point(s_x, s_y), Point(e_x, e_y)\right)$
20: $S_2 \leftarrow Segment(v_1, v_2)$
21: $testIntersect \leftarrow testSegmentIntersect(S_1, S_2)$
22: **if** $testIntersect =$ **true then**
23: $ray_{int} \leftarrow$ **true**
24: $intersection \leftarrow getSegmentIntersection(S_1, S_2)$

```
25:                 d ← getEuclideanDistance(Point(s_x, s_y), intersection)
26:                 if min < 0 then
27:                     min ← d
28:                 else if d < min then
29:                     min ← d
30:         if ray_int = true
31:             LaserScan[i] ← min
32:     return LaserScan
```

Capítulo 5. Implementación de comportamientos.

5.1. Planeador usando campos potenciales.

Durante el paso del tiempo los campos potenciales para evadir obstáculos han tomado gran popularidad en la investigación de robots móviles. La idea fue desarrollada por Andrews y Hogan [18], la cual consiste en fuerzas imaginarias actuando sobre el robot, en donde los obstáculos ejercen fuerzas repulsivas sobre el robot; mientras la posición objetivo ejerce una fuerza de atracción. Un robot puede seguir un camino libre obteniendo el vector de movimiento a partir del campo de fuerzas superpuestas. Las razones del por qué es muy utilizado este método es debido a su simplicidad, rápida implementación y la capacidad de actuar del robot en ambientes dinámicos en tiempo real. La fuerza de atracción se calcula con la ecuación (5.1) [19].

$$\bar{F}_{atr} = K_{atr}\left(\frac{\bar{q}_{act} - \bar{q}_{goal}}{|\bar{q}_{act} - \bar{q}_{goal}|}\right) \tag{5.1}$$

Donde $\bar{q}_{goal}$ es la posición deseada, $\bar{q}_{act}$ es la posición actual del robot, K_{atr} es una constante de diseño que indica cuál es la influencia que tiene la fuerza de atracción.

La fuerza de repulsión total ejercida sobre el robot, es la suma de todas las fuerzas repulsivas ejercidas por los obstáculos sobre el robot. Estas fuerzas repulsivas pueden obtenerse de manera puramente reactiva o con la representación del mundo; para el presente trabajo únicamente se hace el análisis utilizando comportamientos reactivos, debido a que el objetivo es crear un comportamiento dinámico, de tal forma que el robot pueda actuar si se le presenta una situación desconocida.

La fuerza de repulsión que ejerce cada obstáculo únicamente contribuye a la fuerza total repulsiva si este obstáculo se encuentra a una distancia menor que d_{rep}. La ecuación que obtiene la fuerza de repulsión de cada objeto sobre el robot está dada por:

$$\bar{F}_{rep} = -K_{rep}\left(\frac{1}{|\bar{q}_{act} - \bar{q}_{obs}|} - \frac{1}{d_{rep}}\right) * \frac{(\bar{q}_{act} - \bar{q}_{obs})}{|\bar{q}_{act} - \bar{q}_{obs}|^2 |\bar{q}_{act} - \bar{q}_{obs}|} \tag{5.2}$$

Donde $\bar{q}_{obs}$ es la posición del obstáculo, d_{rep} es la distancia mínima en la que el obstáculo tiene influencia en la fuerza de repulsión. Para obtener la posición del obstáculo reactivamente, se utilizan las mediciones de los sensores, por lo cual cada rayo que obtiene información de los obstáculos representa una posición que es usada para obtener una fuerza de repulsión. Las ecuaciones (5.3) y (5.4) sirven para encontrar la posición del obstáculo utilizando un haz de la lectura del sensor láser.

$$q_{obs_x} = q_{act_x} + range_i * \cos\left(ray_{angle}\right) \tag{5.3}$$

$$q_{obs_y} = q_{act_y} + range_i * \sin\left(ray_{angle}\right) \tag{5.4}$$

La fuerza total de repulsión que ejercen los obstáculos sobre el robot, se calcula sumando todas las fuerzas calculadas por cada uno de los rayos de la lectura del sensor láser, a continuación se muestra la ecuación con la que se obtiene la fuerza total de repulsión [19]:

$$\bar{F}_{Trep} = \frac{1}{N}\sum_{n=1}^{N} F_{rep_n} \tag{5.5}$$

Donde K_{rep} es una constante de diseño que indica la influencia que tiene el vector de fuerzas repulsivas sobre el robot, N es el número de rayos de la lectura del sensor láser.

La fuerza total se obtiene sumando todas las fuerzas que actúan sobre el robot, es decir la fuerza de atracción y la fuerza total de repulsión, esta fuerza se calcula utilizando la ecuación (5.6) [19].

$$\bar{F}_T = \bar{F}_{atr} + \bar{F}_{Trep} \tag{5.6}$$

Considerando a q_{act} como la posición actual del robot, q_{t+1} la siguiente posición que se desea alcanzar y el vector de dirección de $\bar{F}_T$, se puede calcular q_{t+1} en base a la ecuación (5.7) [19], se toma el vector unitario de $\bar{F}_T$ debido a que únicamente importa la dirección de esta fuerza.

$$q_{t+1} = q_{act} - \delta \frac{\bar{F}_T}{|\bar{F}_T|} \tag{5.7}$$

5.2. Planeador utilizando la representación del mundo.

La planeación de movimientos es la principal tarea que realiza un robot móvil y tiene como objetivo encontrar un camino libre para el robot de una posición inicial a una posición objetivo. El método basado en campos potenciales, analizado en la sección 5.1, es un enfoque que a menudo es visto como una herramienta que puede ser embebida dentro de otra arquitectura. La planeación de movimientos que utiliza las características de los modelos tradicionales se basa en crear una red de nodos, los cuales el robot puede visitar sin chocar con algún obstáculo.

La planeación de movimientos utilizada en el presente trabajo consiste en obtener la red de nodos donde el robot puede navegar, crear las conexiones entre los nodos de tal forma que no exista obstáculo entre dos nodos conectados y encontrar la secuencia de nodos que se deben visitar para llegar de un punto a otro.

5.2.1. Polígonos aumentados.

El espacio libre por donde el robot consiga navegar puede ser calculado de diversas formas, el método planteado consiste en aumentar los polígonos que representan a los obstáculos de acuerdo a las dimensiones del robot. Un primer método que obtiene los polígonos aumentados consiste en aplicarles una transformación de escalamiento. Para lograr el escalamiento de los polígonos, se obtiene el centro geométrico de cada uno de estos, posteriormente se le aplica una transformación de translación a los segmentos que forman las aristas, con el fin de obtener las coordenadas de los puntos en el sistema de coordenadas locales, esta transformación consiste en trasladar el origen del sistema de referencia al centro geométrico del obstáculo. Finalmente, cada vértice transformado es multiplicado por una matriz de escalamiento y nuevamente trasladado al sistema de referencia global.

Un inconveniente con este método es que no considera las dimensiones del robot, además los obstáculos que están demasiados alargados y angostos tienen formas desproporcionadas, por tal razón se plantea un método que consiste en aumentar los polígonos acorde a las dimensiones del robot obteniendo formas proporcionadas de los polígonos.

Este método consiste en obtener líneas paralelas a cada arista del polígono original. Para cada arista se calculan dos líneas paralelas cuya distancia entre cualquier punto de las nuevas

rectas y la arista del polígono sea igual al factor que se desea aumentar a los polígonos. La meta es calcular dos líneas paralelas a cada segmento L. Para lograr esto, primero se encuentra una línea recta L'_1 perpendicular al segmento de la arista. La pendiente de esta línea se calcula fácilmente, ya que corresponde al reciproco opuesto de la pendiente de L. Una vez obtenida la pendiente de la recta solo se necesita conocer dos puntos que definan a la línea perpendicular. El primer punto corresponde a un punto P_1 del segmento L y el segundo punto P_3 es desconocido; sin embargo, se puede calcular tomando un valor $p_{3_x} > p_{1_x}$ y sustituyendo este valor en la ecuación de la recta, obteniendo el valor de la coordenada y de P_3.

Ya encontrados los puntos P_1 y P_3 que definen a la recta, el siguiente paso es encontrar la intersección de esta línea con un círculo cuyo centro está en P_1 y radio es la proporción que se desea aumentar el polígono, esta prueba de intersección ya se detalló en la sección 4.5. Existen dos puntos Pi_1 y Pi_2 de intersección de la línea con el círculo, que corresponden a los dos primeros puntos de las rectas paralelas a la arista. Los otros dos puntos son calculados tomando el otro extremo del segmento. La Figura 5.1 muestra gráficamente como se obtienen dos líneas paralelas al segmento que define la arista del polígono.

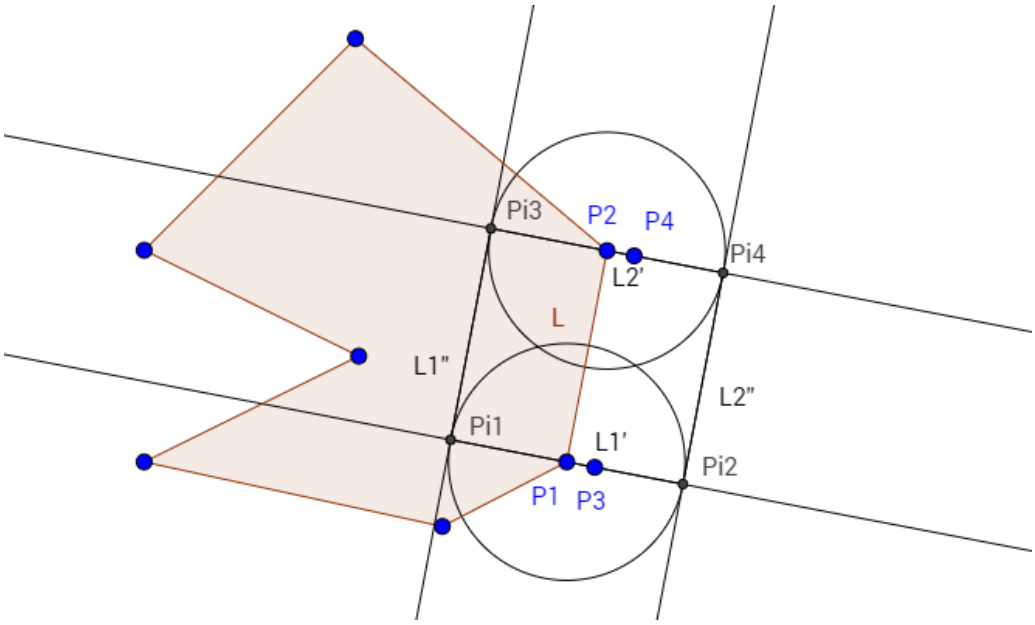


Figura 5.1: Calculo de las rectas L''_1 y L''_2 que son paralelas a L.

Dado que se está manejando la ecuación cartesiana de la recta, la pendiente puede indeterminarse para líneas verticales u horizontales, en estos casos el cálculo de las líneas paralelas se obtiene con una condición que aumenta y resta el valor que se desea aumentar a

la coordenada x en el caso de líneas verticales; por otro lado, cuando se trata de líneas paralelas al eje x se aumenta y resta el valor que se desea aumentar a la coordenada y.

El Algoritmo 10 muestra los pasos que se siguen para obtener las líneas paralelas de un segmento de recta.

Algoritmo 10 Obtiene las rectas L_1 y L_2 que son paralelos al segmento S, la distancia de un punto de L_1 o L_2 al segmento S es igual a r.

Entradas

S – Segmento de recta definido por P_1 y P_2 de S.
r – Distancia de cualquier punto de L_1 a L.

Salidas

L_1 – Segmento paralelo a S.
L_2 – Segmento paralelo a S.

1: **Function** computeParallelLines(S, r)
2: $dx \leftarrow S.P_{2_x} - S.P_{1_x}$
3: $dy \leftarrow S.P_{2_y} - S.P_{1_y}$
4: **if** $dx = 0$ **then**
5: $P_1 \leftarrow \left(S.P_{1_x} - r, S.P_{1_y}\right)$
6: $P_2 \leftarrow \left(S.P_{2_x} - r, S.P_{2_y}\right)$
7: $P_3 \leftarrow \left(S.P_{1_x} + r, S.P_{1_y}\right)$
8: $P_4 \leftarrow \left(S.P_{2_x} + r, S.P_{2_y}\right)$
9: **else if** $dy = 0$ **then**
10: $P_1 \leftarrow \left(S.P_{1_x}, S.P_{1_y} - r\right)$
11: $P_2 \leftarrow \left(S.P_{2_x}, S.P_{2_y} - r\right)$
12: $P_3 \leftarrow \left(S.P_{1_x}, S.P_{1_y} + r\right)$
13: $P_4 \leftarrow \left(S.P_{2_x}, S.P_{2_y} + r\right)$
14: **else**
15: $inv \leftarrow -\frac{dx}{dy}$
16: $R_1 \leftarrow \left(S.P_{1_x}, S.P_{1_y}\right)$
17: $x_1 \leftarrow S.P_{1_x} + 100 * r$
18: $y_1 \leftarrow inv * \left(x_1 - S.P_{1_x}\right) + S.P_{1_y}$
19: $R_2 \leftarrow (x_1, y_1)$
20: $(t_1, t_2) \leftarrow getLineCircleIntersection(Segment(R_1, R_2), Circle(S.P_1, r))$
21: $P_1 \leftarrow (R_2 - R_1) * t_1 + S.P_{1_x}$
22: $P_2 \leftarrow (R_2 - R_1) * t_2 + S.P_{1_x}$

23: $R_3 \leftarrow \left(S.P_{2_x}, S.P_{2_y}\right)$
24: $x_2 \leftarrow S.P_{2_x} + 100 * r$
25: $y_2 \leftarrow inv * \left(x_2 - S.P_{2_x}\right) + S.P_{2_y}$
26: $R_4 \leftarrow (x_2, y_2)$
27: $(t_3, t_4) \leftarrow getLineCircleIntersection(Segment(R_3, R_4), Circle(S.P_2, r))$
28: $P_3 \leftarrow (R_4 - R_3) * t_1 + S.P_{2_x}$
29: $P_4 \leftarrow (R_4 - R_3) * t_2 + S.P_{2_x}$
30: $S_1 \leftarrow Segment(P_1, P_2)$
31: $S_2 \leftarrow Segment(P_3, P_4)$
32: **return** S_1, S_2

El Algoritmo 10 obtiene dos rectas paralelas al segmento; sin embargo, únicamente se considera la recta cuyos puntos que la define estén a la derecha del segmento, esta prueba ya fue descrita en la sección 4.6. Este proceso se repite para todos los segmentos de los polígonos, con ello se encuentran un conjunto de líneas que delimitan al polígono aumentado como se muestra en la Figura 5.2.

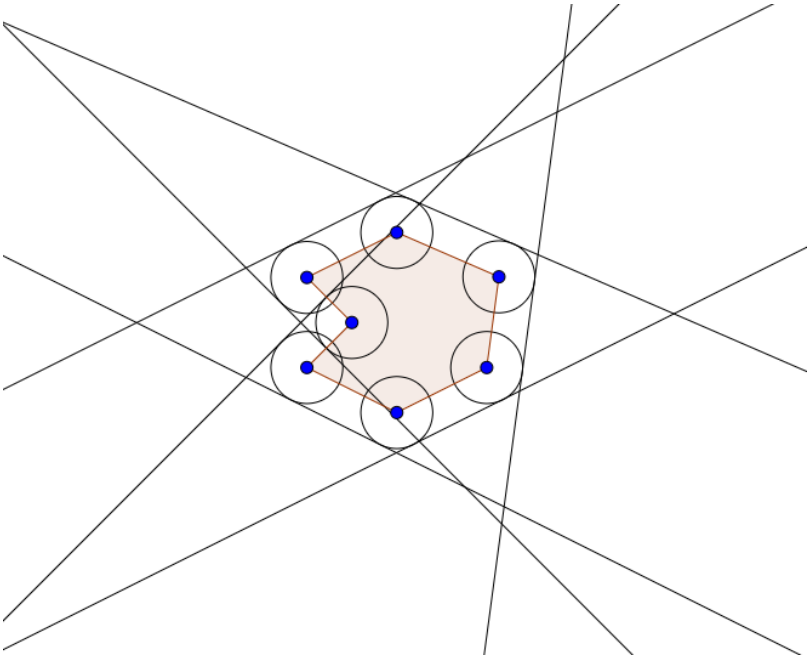


Figura 5.2: Líneas paralelas a los segmentos del polígono.

Finalmente, para obtener los segmentos del polígono aumentado, se calculan las intersecciones entre líneas consecutivas como se muestra en la Figura 5.3. La intersección entre dos líneas ya se ha detallado en la sección 4.6.

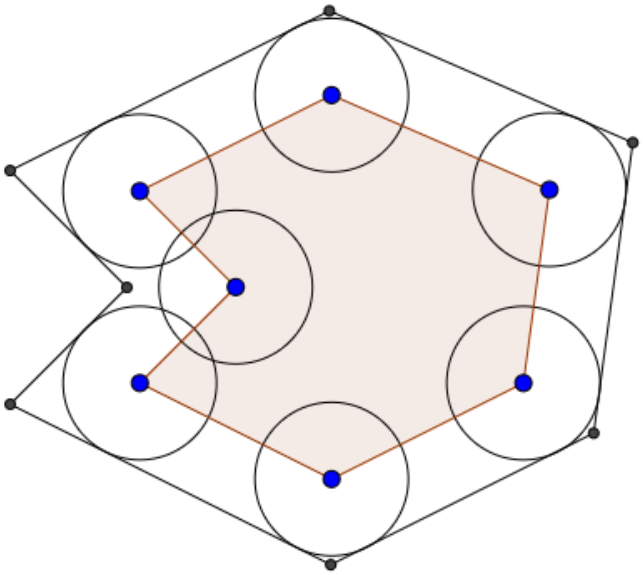


Figura 5.3: Polígono aumentado.

A continuación se muestra el algoritmo que obtiene los polígonos aumentados.

Algoritmo 11 Algoritmo para aumentar los polígonos.

Entradas

P – Arreglo de polígonos de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

r – Radio de expansión.

Salidas

G – Arreglo de polígonos aumentados de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

```
1:  Function grownPolygons(P, r)
2:      for i = 1 to N do
3:          for j = 1 to P[i].vertex_size do
4:              v1 ← P[i].vertex[j]
5:              if j < P[i].vertex_size then
6:                  v2 ← P[i].vertex[j + 1]
7:              else
8:                  v2 ← P[i].vertex[1]
9:              S ← Segment(v1, v2)
10:             (S1, S2) ← computeParallelLines(S, r)
11:             if getDeterminant(S.P2, S.P1, S1.P1) > 0 then
12:                 E[i][j] ← S1
```

13: **else if** $getDeterminant(S.P_2, S.P_1, S_2.P_1) > 0$ **then**
14: $E[i][j] \leftarrow S_2$
15: **for** $i = 1$ to N **do**
16: **for** $j = 1$ to $P[i].vertex\_size$ **do**
17: $Se_1 \leftarrow E[i][j]$
18: **if** $j = 1$ **then**
19: $Se_2 \leftarrow E[i][P[i].vertex\_size]$
20: **else**
21: $Se_2 \leftarrow E[i][j - 1]$
22: $intersection \leftarrow getSegmentIntersection(Se_1, Se_2)$
23: $G[i][j] \leftarrow intersection$
24: **return** G

5.2.2. Creación de mapa topológico.

Para crear el mapa topológico se realiza la conexión de los vértices de un polígono aumentado con los vértices del resto de los polígonos tomando en cuenta las siguientes consideraciones:

- Un vértice se conecta a otro si existe un camino libre en línea recta que el robot pueda alcanzar sin chocar con ningún obstáculo.
- Para verificar si existe camino libre sin obstáculo en línea recta, se obtienen dos líneas paralelas al segmento, cuya distancia de cualquier punto de éstas al segmento que une a los dos vértices sea igual al radio del robot.
- El robot puede navegar entre dos vértices si no existe intersección entre las líneas paralelas y las aristas de los polígonos originales, además se debe cumplir esta condición entre el segmento que une a los dos vértices y las aristas de los polígonos originales.
- Es posible la interconexión entre los vértices del mismo polígono aumentado siempre y cuando no exista intersección con las aristas de los obstáculos.
- Un vértice no se conecta a otro si éste se encuentra entre el polígono y el polígono aumentado.

La interconexión de nodos donde el robot puede navegar es utilizada con el fin de crear una matriz de adyacencia entre nodos que posteriormente se utiliza para encontrar la ruta óptima entre dos puntos.

El Algoritmo 12 muestra resumidamente las condiciones para que un vértice de un polígono aumentado se conecte con otro.

Algoritmo 12 Determinar si dos vértices pueden unirse sin que el robot choque.

Entradas

P – Arreglo de polígonos de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

P_g – Arreglo de polígonos aumentados de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

p_1 – Vértice que se desea conectar.

p_2 – Vértice que se desea conectar.

r – Radio del robot.

Salidas

B – Bandera que indica si existe unión entre p_1 y p_2.

1: **Function** connectVertex(P, P_g, p_1, p_2, r)
2: $S \leftarrow Segment(p_1, p_2)$
3: $(S_1, S_2) \leftarrow computeParallelLines(S, r)$
4: **for** $i = 1$ to N **do**
5: **for** $j = 1$ to $P[i].vertex\_size$ **do**
6: $v_1 \leftarrow P[i].vertex[j]$
7: $gv_1 \leftarrow P_g[i].vertex[j]$
8: **if** $j < P[i].vertex\_size$ **then**
9: $v_2 \leftarrow P[i].vertex[j + 1]$
10: $gv_2 \leftarrow P_g[i].vertex[j + 1]$
11: **else**
12: $v_2 \leftarrow P[i].vertex[1]$
13: $gv_2 \leftarrow P_g[i].vertex[1]$
14: $S_T \leftarrow Segment(s_1, s_2)$
15: $d_1 \leftarrow getDeterminant(v_1, v_2, p_1)$
16: $d_2 \leftarrow getDeterminant(v_2, gv_2, p_1)$
17: $d_3 \leftarrow getDeterminant(gv_2, gv_1, p_1)$
18: $d_4 \leftarrow getDeterminant(gv_1, v_1, p_1)$
19: $d_5 \leftarrow getDeterminant(v_1, v_2, p_2)$
20: $d_6 \leftarrow getDeterminant(v_2, gv_2, p_2)$
21: $d_7 \leftarrow getDeterminant(gv_2, gv_1, p_2)$
22: $d_8 \leftarrow getDeterminant(gv_1, v_1, p_2)$
23: **if** $(d_1 < 0$ ***and*** $d_2 < 0$ ***and*** $d_3 < 0$ $d_4 < 0)$ ***or***
$(d_5 < 0$ ***and*** $d_6 < 0$ ***and*** $d_7 < 0$ $d_8 < 0)$ ***or***
$testSegmentIntersect(S, S_t)$ ***or*** $testSegmentIntersect(S_1, S_t)$ ***or***
$testSegmentIntersect(S_2, S_t)$ **then**
24: $B \leftarrow$ **true**
25: **break**
26: **if** $B =$ **true then**
27: **break**
28: **return** B

Para cada vértice del polígono aumentado se le debe realizar la prueba de conexión con cada vértice del resto de los polígonos. Cuando se trata del mismo polígono únicamente se hace la prueba entre vértices consecutivos del mismo polígono (aristas del polígono). En el Algoritmo 13 se muestra como se obtiene la matriz de adyacencia que determina las conexiones entre los vértices de los polígonos aumentados.

Algoritmo 13 Obtiene el mapa topológico entre los vértices de los polígonos aumentados.

Entradas

P – Arreglo de polígonos de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

P – Arreglo de polígonos aumentados de tamaño N, cada polígono tiene un arreglo de vértices de tamaño M.

r – Radio del robot.

Salidas

A – Matriz de adyacencias de tamaño $N * M$ x $N * M$.

```
1:  Function computeMapTopologic(P, G, r)
2:      sumSizeVertex ← 0
3:      A ← 0
4:      for i = 1 to N do
5:          offset[i] ← sumSizeVertex
6:          sumSizeVertex ← sumSizeVertex + P[i].vertex_size
7:      for i = 1 to N do
8:          for j = i to N do
9:              if i ≠ j then
10:                 for k = 1 to N do
11:                     v1 ← P[i].vertex[k]
12:                     for l = 1 to P[i].vertex_size do
13:                         v2 ← P[j].vertex[l]
14:                         B ← connectVertex(P, v1, v2, r)
15:                         if B = false then
16:                             A[offset[i] + k][ offset[j] + l] = 1
17:                             A[ offset[j] + l][offset[i] + k] = 1
18:             else
19:                 for k = 1 to P[i].vertex_size do
20:                     if k < P[i].vertex_size then
21:                         v2 ← P[i].vertex[k + 1]
22:                     else
23:                         v2 ← P[i].vertex[1]
24:                     B ← connectVertex(P, v1, v2, r)
25:                     if B = false then
26:                         if k < P[i].vertex_size then
27:                             A[offset[i] + k][ offset[i] + k + 1] = 1
```

28: $A[offset[i]+k+1][\,offset[i]+k] = 1$
29: **else**
30: $A[offset[i]+k]\big[\,offset[i]\big] = 1$
31: $A\big[offset[i]\big][\,offset[i]+k] = 1$
32: **return** A

5.2.3. Algoritmo de Dijkstra para encontrar la ruta más óptima.

El algoritmo para encontrar la ruta de un origen a un destino consiste en agregar los nodos de inicio y fin al arreglo que contiene a los nodos del mapa, además lleva a cabo las conexiones con el resto de los nodos y actualiza la matriz de adyacencias. La conexión de un nodo de inicio o fin con otro nodo se realiza si existe un camino en línea recta que conecte a ambos nodos sin que el robot choque con algún obstáculo, lo cual se determina creando dos segmentos de recta paralelos al segmento que une a los dos nodos, cuya distancia de éste con ambos segmentos es igual al radio del robot. Existe unión si los tres segmentos no intersectan con los polígonos de los obstáculos, este método ya fue detallado en la sección 5.2.1.

El algoritmo de Dijkstra proporciona una solución al problema de la ruta más óptima en un grafo conexo. El grafo que se crea con la matriz de adyacencias obtenida anteriormente es un grafo dirigido, el cual todos los pesos entre las conexiones de los nodos del grafo son positivos. En el presente trabajo se hace uso de este algoritmo para encontrar la secuencia de posiciones que el robot debe visitar para llegar de un punto a otro. La implementación del algoritmo de Dijkstra se encuentra resumida en el Algoritmo 14.

Algoritmo 14 Implementación del algoritmo de Dijsktra.

Entradas

V – Arreglo de nodos del grafo de tamaño N.
W – Matriz de pesos de tamaño N x N, donde N es el número de nodos del grafo.
s – Índice de V correspondiente a la posición de origen.
d – Índice de V correspondiente a la posición destino.

Salidas

P – Arreglo de índices de los nodos de la ruta encontrada.

1: **Function** dijkstra(V, W, s)
2: $D[s] \leftarrow 0$
3: **for all** $v \in V - \{s\}$ **do**
4: $D[v] \leftarrow \infty$
5: $P[v] \leftarrow -1$
6: $S[v] \leftarrow 0$

```
7:    start ← s
8:    S[start] ← 1
9:    while S[d] = 0 do
10:       u ← ∞
11:       m ← −1
12:       for all v ∈ V − {s} do
13:          d ← D[start] + W[start][v]
14:          if d < dis[v] and S[v] = 0 then
15:             D[v] ← d
16:             P[v] ← start
17:          if min > D[v] and S[v] = 0 then
18:             u ← D[v]
19:             m ← v
20:       start ← m
21:       S[start] ← 1
```

5.3. Planeación con rejilla de ocupación.

Otro enfoque que se le puede dar a la planeación de movimientos es utilizar una rejilla de ocupación espacial. Una rejilla de ocupación consiste en realizar la discretizacion del mundo en celdas, cada celda que componen a la rejilla de ocupación define si está libre de obstáculo o no; en otras palabras, el objetivo es crear una matriz de NxM en la que cada elemento representa una celda. La celda que está libre de obstáculo tiene el valor de 0 y cuando la celda está ocupada o desconocida tiene un valor diferente de cero.

Para generar la rejilla de ocupación espacial en el simulador se hace uso de las lecturas del láser que se detallaron en la sección 4.6 y de un nodo de ROS llamado *gmapping*, el cual crea un mapa de rejillas de ocupación espacial en base a las lecturas del sensor láser. La Figura 5.4 muestra cómo se utilizan las lecturas de los sensores y el nodo de ROS para crear el mapa de rejillas de ocupación.

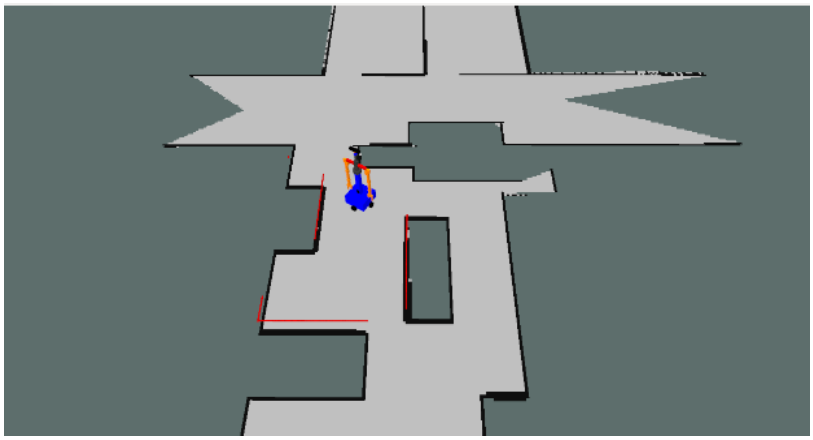


Figura 5.4: Rejilla de ocupación generada con las lecturas del sensor y el nodo *gmapping*.

Una vez creada la celda de ocupación espacial, se utiliza otro paquete de ROS llamado *costmap_2d*, el cual proporciona una estructura configurable que almacena la información de las celdas libres por donde el robot puede navegar en forma de rejilla de ocupación; es decir, este nodo crea una estructura que se adapta a las dimensiones del robot de tal forma que determina cuales celdas pueden ser visitadas sin que el robot choque. La Figura 5.5 muestra el funcionamiento del paquete de ROS *costmap_2d*.

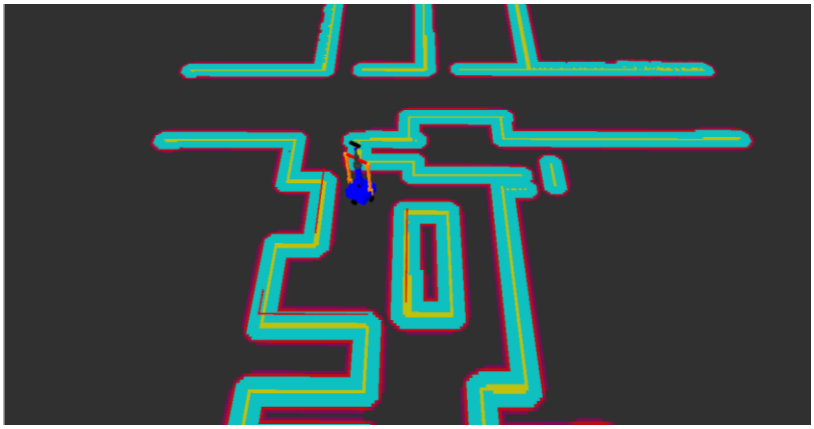


Figura 5.5: Mapa generado con el nodo *costmap_2d*, rejilla de ocupación considerando las dimensiones del robot.

Con este método se crea un grafo donde las celdas son vértices y las aristas son las celdas vecinas libres con las que se conecta. Ya obtenido este grafo se realiza la búsqueda de las celdas que debe visitar el robot para llegar de un punto a otro; sin embargo, esta búsqueda se reduce utilizando el diagrama de Voronoi de la rejilla de ocupación espacial. El diagrama de

Voronoi de la rejilla de ocupación se obtiene mediante un paquete de ROS llamado Dynamic Voronoi, que es una implementación de un eficiente algoritmo para actualizar el diagrama de Voronoi en ambientes dinámicos [20]. La Figura 5.6 muestra el diagrama de Voronoi de una rejilla de ocupación espacial.

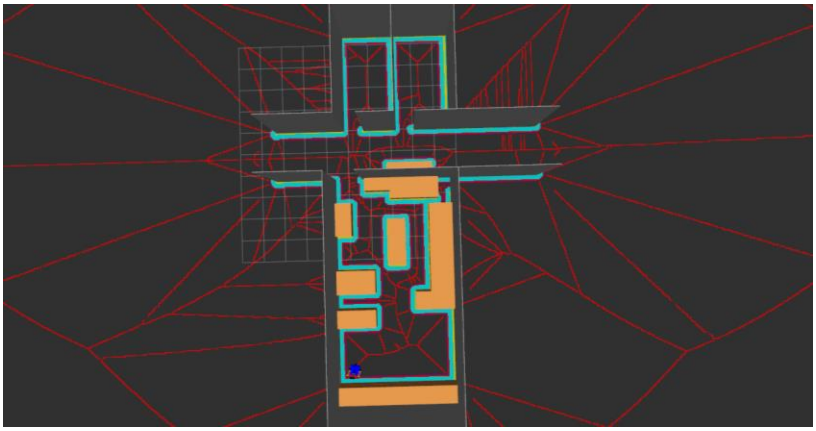


Figura 5.6: Diagrama de Voronoi de la rejilla de ocupación.

La búsqueda del camino de un punto a otro se simplifica ya que únicamente se pueden recorrer las celdas que forman el diagrama de Voronoi. El algoritmo implementado consiste en obtener las celdas correspondientes a los puntos de inicio y fin, esto se realiza usando las propiedades que proporciona la rejilla de ocupación como son: el origen del mapa, su resolución y su dimensión. El origen O del mapa es simplemente las coordenadas x, y desde donde se empezó a crear el mapa, la resolución r es un factor que indica cuál es el tamaño de cada celda, la dimensión de la rejilla es el número de celdas horizontales y verticales que la forman. Para obtener la celda correspondiente a un punto P en plano $z = 0$, se utilizan las ecuaciones (5.8) y (5.9). Adicional a esta conversión se le agrega la validación de que las celdas obtenidas estén contendidas en las dimensiones de la rejilla de ocupación.

$$celda_x = \frac{(p_x - o_x)}{r} \tag{5.8}$$

$$celda_y = \frac{(p_y - o_y)}{r} \tag{5.9}$$

Ya que se tienen las celdas del origen y destino se utiliza el algoritmo de búsqueda de costo uniforme que encuentra la ruta de la celda origen a la celda más cercana del diagrama

de Voronoi, es importante mencionar que este proceso se repite de la misma forma con la celda correspondiente al destino. Las celdas del diagrama de Voronoi obtenidas son utilizadas posteriormente para realizar nuevamente una búsqueda con $A*$, con el fin de encontrar las celdas del diagrama de Voronoi que debe visitar para llegar de un extremo a otro.

El algoritmo $A*$ utiliza dos listas: la lista abierta y la lista cerrada. El propósito de la lista abierta es almacenar los candidatos a mejor ruta de nodos que aún no han sido considerados, ésta es inicializada con el nodo de inicio. Si la lista abierta está vacía entonces no es posible encontrar una ruta. La lista cerrada es inicializada vacía y contiene a todos los nodos que ya han sido visitados.

Este algoritmo itera sobre la lista abierta seleccionado el nodo con menor costo estimado para alcanzar el objetivo. Si el nodo seleccionado no es el nodo destino, se insertan todos los nodos vecinos validos en la lista abierta y se repite este proceso. Cabe destacar que este algoritmo crea un nodo manteniendo la referencia a su nodo padre, de esta forma es posible regresar al nodo que lo generó. En un nodo se almacena la posición i, j de cada celda, la referencia a su padre y tres valores numéricos asociados con este, estos valores determinan como el algoritmo considera primero a un nodo, estos valores son:

- $costo_g$ es el costo incremental que toma llegar del nodo inicio al nodo actual, éste es igual al costo de su nodo padre más el costo que toma llegar del nodo padre al nodo actual, este valor depende del tipo de vecino que es el nodo actual. Cuando son desplazamiento laterales el costo que toma desplazarse entre celdas es igual a la unidad, en caso de que sea una celda diagonal se toma la distancia utilizando el teorema de Pitágoras, el cual corresponde a 1.414.
- $costo_h$ es el costo estimado que toma llegar del nodo actual al nodo objetivo, este valor depende en gran medida de las propiedades del grafo en el que se esté haciendo la búsqueda, en el presente trabajo se eligió utilizar la distancia "*Manhattan*" que es ideal para la búsqueda en rejillas de ocupación, el costo estimado se obtiene con la expresión $costo_h = |n_i - goal_i| + |n_j - goal_j|$, donde n es la celda actual y $goal$ es la celda objetivo.
- $costo_f$ es simplemente la suma de los costos descritos anteriormente $costo_f = costo_g + costo_h$.

Por último la ruta se encuentra tomando el nodo destino y retrocediendo al padre de cada nodo.

En el Algoritmo 15 se muestra el funcionamiento del algoritmo de búsqueda $A*$ descrito anteriormente en un diagrama de Voronoi.

Algoritmo 15 Implementación del algoritmo de A*.

Entradas

C_i – Celda de inicio.

C_g – Celda objetivo.

GDV – Rejilla de ocupación del diagrama de Voronoi.

Salidas

P – Ruta que debe seguir el robot para llegar a la celda objetivo.

Function a-star(C_i, C_g)
    $ClosedList \leftarrow \emptyset$
    $start_g \leftarrow 0$
    $start_f \leftarrow start_{acumulado} + \left|C_{i_x} - C_{g_x}\right| + \left|C_{i_y} - C_{g_y}\right|$
    $OpenList.Agregar(start)$
    while $OpenList \neq \emptyset$ **do**
        $current \leftarrow$ Elemento de $OpenList$ con menor costo total.
        if $current = goal$ **then**
            return $construct\_path(goal)$
        $OpenList.Remover(current)$
        $ClosedList.Agregar(current)$
        for all $n_i \in neighbors(current)$ **do**
            if $n_i \notin GDV$ **then**
                continue
            if n_i $isOccupied$ **then**
                continue
            if $n_i \notin ClosedList$ **then**
                if n_i ***is*** $lateral$ **then**
                    $n_{i_g} = current_g + 1$
                else if n_i ***is*** $diagonal$ **then**
                    $n_{i_g} = current_g + 1.414$
                $n_{i_f} = n_{i_g} + \left|n_{i_x} - C_{g_x}\right| + \left|n_{i_y} - C_{g_y}\right|$
                if $n_i \notin OpenList$ **then**
                    $OpenList.Agregar(n_i)$
                else
                    $on_i = n_i$ ***in*** $OpenList$
                    if $n_{i_g} < on_{i_g}$ **then**
                        $on_{i_g} = n_{i_g}$
                        $on_{i_{parent}} = n_{i_{parent}}$

5.4. Comportamientos híbridos.

El comportamiento mostrado en el apartado 5.1 corresponde a una arquitectura reactiva, es preciso destacar que éste utiliza la información de la lectura del sensor láser para actuar instantáneamente a estímulos. En lo referente a los comportamientos tradicionales, los cuales se mostraron en los apartados 5.2 y 5.3; en ambos modelos existe una serie de ventajas y desventajas que se mostraron en la sección 2.2, en el caso de los comportamientos tradicionales la desventaja es la ausencia de información que mantenga actualizado el medio ambiente impidiendo actuar cuando se presentan objetos desconocidos; por su parte, la desventaja de utilizar campos potenciales es que el campo de fuerzas imaginarias que actúan sobre el robot conlleven a que entre en estado de un mínimo local.

Para solucionar estas problemáticas se utiliza un comportamiento hibrido; por consiguiente, los campos potenciales y la planeación empleando la representación del mundo se combinan de la siguiente forma:

- Para llegar de un punto a otro se obtiene la ruta empleando la planeación de movimientos utilizando el mapa topológico, cada elemento de la ruta se utiliza como la posición objetivo para campos potenciales.
- Se detecta el mínimo local cuando el movimiento no ha sido significativo en un periodo de tiempo establecido.
- Si el robot se encuentra en un mínimo local, se consideran las lecturas del sensor para crear los obstáculos que estén suficientemente cercanos en forma de polígonos.
- Con el fin de alejarse de dicho estado, se activa la máquina de estados que se muestra en la Figura 5.7, la cual realiza la cuantificación de la lectura de los sensores con el objetivo de obtener la posición de los obstáculos: enfrente, izquierda o derecha. La Figura 5.8 muestra un robot móvil que evade obstáculos.
- Para encontrar si existe, la nueva ruta por donde el robot puede navegar, se utiliza la unión de los obstáculos conocidos y desconocidos.

Este mismo principio se aplica haciendo uso de la planeación empleando una rejilla de ocupación; sin embargo, en este caso la celda vecina a cada elemento de la ruta se encuentra muy cercana una de otra, lo que implica que el movimiento entre celdas sea muy lento, no obstante la estrategia utilizada para la planeación empleando una rejilla de ocupación consiste en:

- Detectar los obstáculos no conocidos utilizando las lecturas del sensor láser, en el caso de existir obstáculos suficientemente cercanos, se activa la máquina de estados que se muestra en la Figura 5.7, al igual que la planeación empleando un mapa topológico realiza la cuantificación de la lectura de los sensores con el objetivo de obtener la posición de los obstáculos: enfrente, izquierda o derecha.

- El paquete *costmap_2d* detallado en la sección 5.3 se actualiza dinámicamente haciendo uso de la lectura del sensor láser, esta actualización sucede cuando el robot está a una distancia determinada del obstáculo.
- Ejecuta la re-planeación empleando la celda de ocupación actualizada.

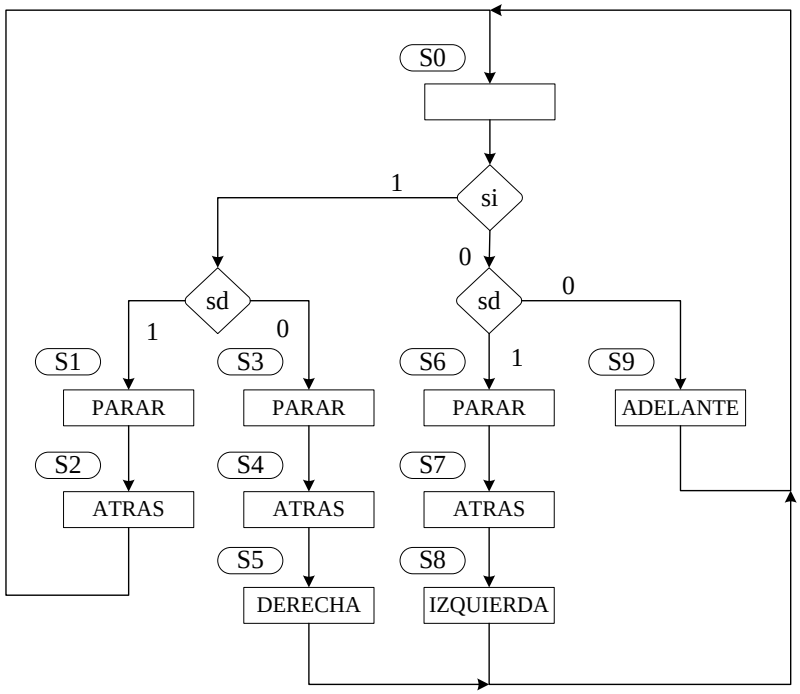


Figura 5.7: Maquina de estados que se ejecuta previamente a la re-planeación de la ruta.

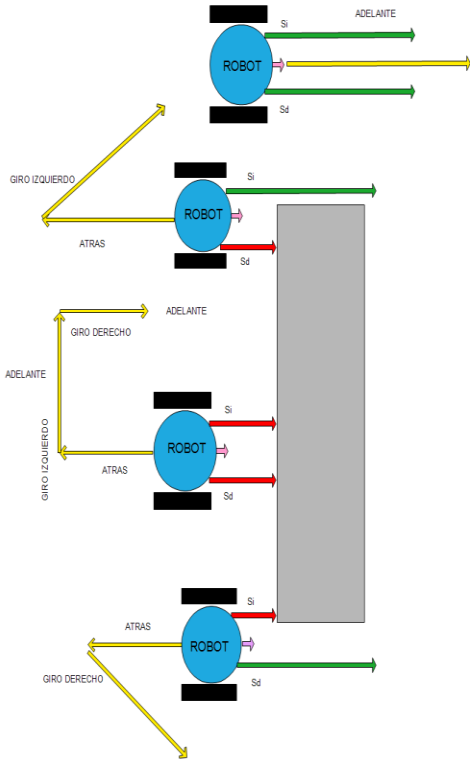


Figura 5.8: Robot Móvil que evade obstáculos.

Capítulo 6. Pruebas y resultados.

6.1. Pruebas del simulador.

Para verificar el buen funcionamiento del simulador fue necesario la implementación de los algoritmos detallados en el Capítulo 5 cuyo objetivo es:

- Probar el conjunto de funciones con las que cuenta el robot Justina para realizar movimientos.
- Probar los algoritmos de colisión implementados.
- Probar la simulación del sensor láser.

Las pruebas del conjunto de funciones del simulador usando el modelo del robot Justina, consistieron en probar en forma individual cada una de las funciones que proporcionan los movimientos u alguna funcionalidad a la simulación. Estas pruebas se realizaron con ayuda de los comandos que ROS provee para el envío de mensajes por medio de la terminal de *Linux Ubuntu*. Las funciones que se probaron se muestran en la siguiente tabla:

Función	**Descripción**
Movimiento de la base	Función que recibe las velocidades lineales de las llantas.
Movimientos básicos	Funciones relacionadas con los movimientos básicos del robot, las cuales se detallaron en la sección 4.4.
Sensor	Función que obtiene las lecturas de los sensores.
Ambiente	Función que obtiene la estructura del medio ambiente.
Campos potenciales	Función para iniciar el comportamiento de campos potenciales.
Planeador de movimientos	Funciones para realizar la planeación de movimientos.

Tabla 6.1: Pruebas realizadas a funciones del simulador.

Las pruebas fueron realizadas bajo diferentes condiciones, obteniendo resultados homogéneos. En la Figura 6.1 y Figura 6.2 se presentan la distribución y configuración de los distintos ambientes, donde los objetos de color café representan los obstáculos, mientras los objetos de color gris las paredes.

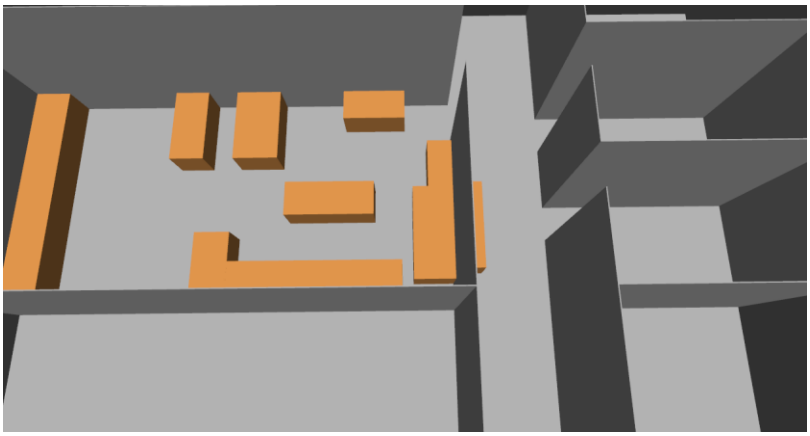


Figura 6.1: Distribución espacial de objetos del laboratorio de Bio-Robótica.

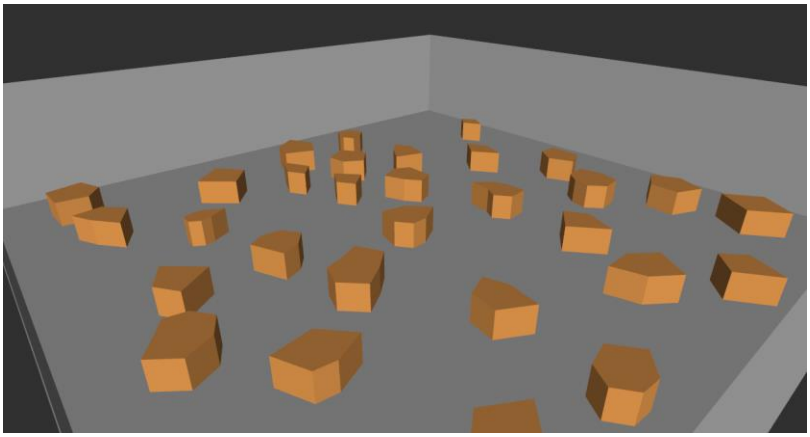


Figura 6.2: Distribución de objetos con diferentes formas.

La representación del mundo resultó de gran utilidad ya que es de fácil creación y ofrece una estructura eficiente para determinar la colisión del robot con los obstáculos, tomando cada arista del polígono y realizando la prueba de superposición con la geometría de la base del robot. Cabe mencionar que la detección de colisiones ayudó a que la simulación tuviera un grado más alto de realismo, siendo evidente que el uso de la lectura de los sensores para evadir obstáculos permite estimar en que momento el robot choca; sin embargo, existen condiciones en la que la proximidad del robot con los obstáculos impide la ejecución del comando deseado, y es por ello que la integración de los algoritmos de colisión ayudó a obtener una mejor aproximación de cómo funcionará en un ambiente real.

Otra forma en la que se utilizó la estructura de los polígonos fue para la simulación del sensor láser, esta estructura permitió la detección de intersección entre líneas de manera eficiente. Debido a la distribución de funcionalidades en el simulador fue imprescindible determinar la frecuencia a la que funcionan los nodos que proporcionan la relación de los puntos de referencia de la base del robot y la del sensor láser, ambos nodos trabajan a una

frecuencia de 30 Hz. La Figura 6.3 muestra la simulación grafica del sensor láser, donde las esferas rojas representan las lecturas del sensor.

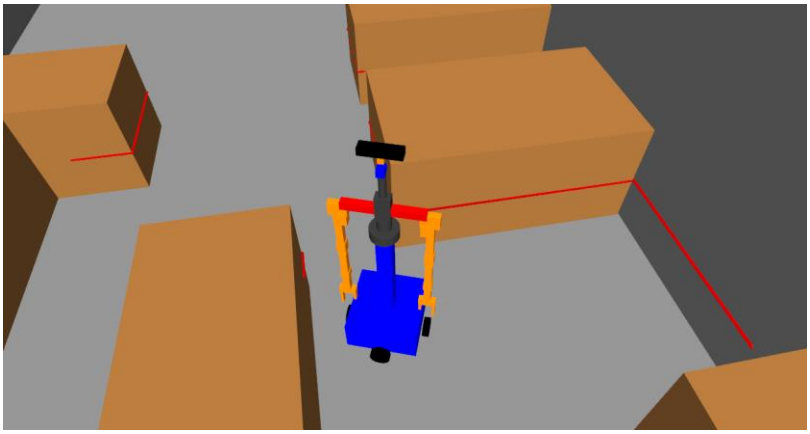


Figura 6.3: Simulación del sensor láser.

La configuración del sensor láser que fue simulado corresponde al que tiene el robot Justina. Este sensor es el modelo Hokuyo UHG-08LX. La Tabla 6.2 muestra los parámetros utilizados en la simulación de dicho sensor.

Parámetro	Valor
num_{rays}	768
$angle_{min}$	135°
$range_{min}$	0 m
$range_{max}$	8 m
$range_{angle}$	270°

Tabla 6.2: Parámetros del sensor láser simulado.

Las respectivas implementaciones se realizaron en el lenguaje de programación C++, es importante mencionar que se utilizó el paradigma orientado a objetos con el fin de abstraer funcionalidades que permitieron posteriormente integrar a la arquitectura distribuida con la que cuenta ROS.

Las características del equipo de cómputo en la que se probó el simulador y la implementación de los algoritmos se presentan en la Tabla 6.3. En todos los equipos de cómputo se tienen resultados similares debido a que existe un control de frecuencia que es manejada en cada nodo implementado.

Equipo	Equipo 1	Equipo 2	Equipo 3
Procesador	AMD E2-1800 Dual-Core a 1.7 GHz	Intel Core i5 Quad-Core a 1.6 GHz.	Intel Core i7 Quad-Core a 2.4 GHz.
Memoria RAM	6 GB	8 GB	8 GB
Gráficos	AMD Radeon HD 7340 Graphics	NVIDIA Geforce 840M	NVIDIA Geforce GTX 770

Tabla 6.3: Características del equipo de cómputo.

6.2. Resultados.

En esta sección se presentan los resultados obtenidos de la implementación de los algoritmos presentados en el Capítulo 5 haciendo uso del simulador. Los detalles de los resultados de estas implementaciones se muestran a continuación.

6.2.1. Campos potenciales.

Con la implementación de campos potenciales se obtuvo un comportamiento puramente reactivo que proporcionó al robot la capacidad de evadir obstáculos. Los parámetros utilizados, que controlan las fuerzas involucradas, se muestran en la Tabla 6.4. Estos parámetros se obtuvieron experimentalmente realizando pruebas para ajustar la influencia que tenían las fuerzas de atracción y repulsión; cabe mencionar que los parámetros pueden variar para otro modelo del robot. En la Figura 6.4 se observa la evasión de obstáculos empleado estos valores.

Parámetro	Valor
$\boldsymbol{d_{rep}}$	0.59
$\boldsymbol{K_{rep}}$	3.65
$\boldsymbol{d_{atr}}$	0.28
$\boldsymbol{K_{atr}}$	1.65
$\boldsymbol{\delta}$	0.1

Tabla 6.4: Parámetros empleados para el comportamiento de campos potenciales utilizando el modelo del robot Justina.

Durante las pruebas realizadas se identificó una problemática utilizando este algoritmo. Los campos potenciales son propensos a entrar en estado de un mínimo local, lo cual significa que el robot entra en una situación en la que el campo de fuerzas actúa de tal forma que el robot no realiza movimiento alguno.

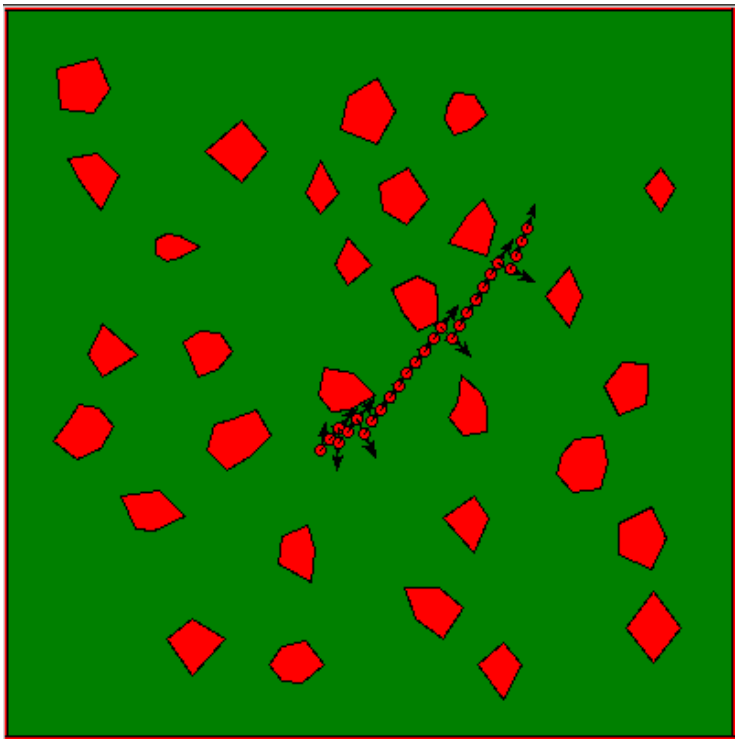


Figura 6.4: Ejemplo de trayectoria empleando campos potenciales.

6.2.2. Planeación utilizando mapa topológico.

La implementación de la planeación utilizando el mapa topológico consistió de tres etapas la primera fue encontrar los polígonos aumentados, la segunda crear la red de nodos del mapa topológico y por ultimo determinar la ruta que debe seguir para llegar de un punto a otro utilizando el mapa topológico.

La primera prueba que se realizó fue la creación de los polígonos aumentados. Estos polígonos se ajustaron a la dimensión del robot Justina, el cual tiene una base rectangular de $0.24x0.24\ m$. Para que pueda realizar libremente los movimientos se tomó el valor de ajuste a la máxima distancia que existe entre el centro y cualquier esquina, por lo que esta distancia

corresponde a $\sqrt{0.24^2 + 0.24^2} = 0.34$. En la Figura 6.5 se observa el caculo de los polígonos aumentados que se obtuvo para la distribución espacial del laboratorio de Bio-Robótica, mientras en la Figura 6.6 se muestra el resultado del cálculo de los polígonos aumentados para la distribución de objetos con diferentes formas.

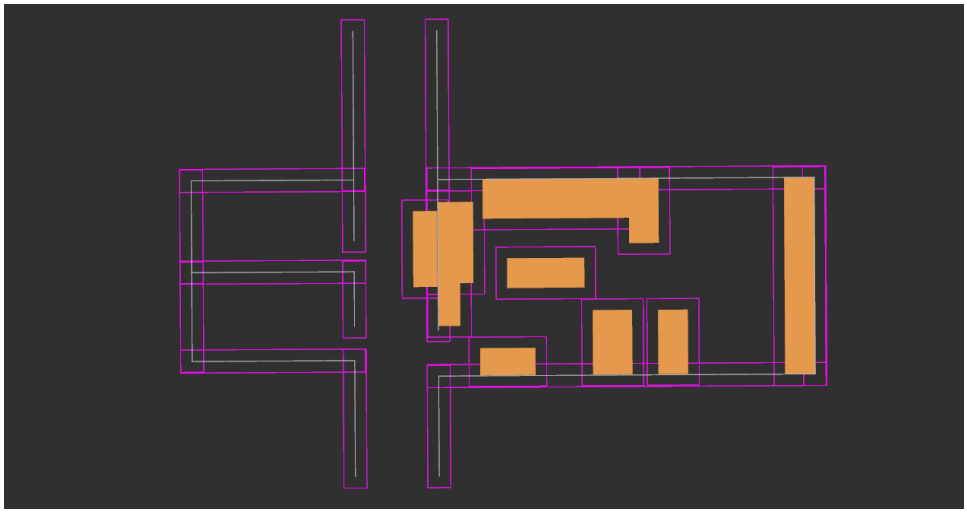


Figura 6.5: Polígonos aumentados de la distribución de objetos conforme al laboratorio de Bio-Robótica.

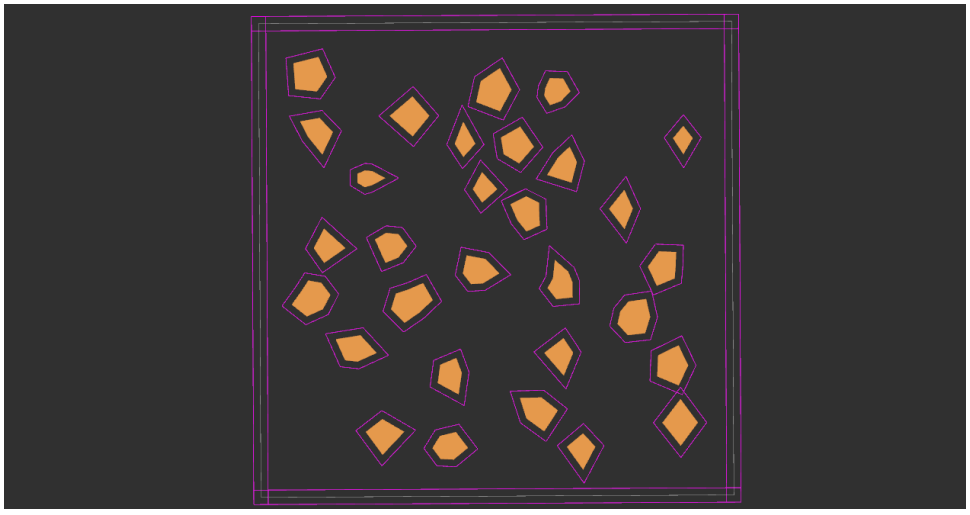


Figura 6.6: Polígonos aumentados de la distribución de objetos con múltiples formas.

Posteriormente se procedió a obtener el mapa topológico. Los algoritmos que calculan este mapa toman como parámetro el radio del robot, en este caso se utilizó el valor de $0.24\ m$ que corresponde al radio del robot Justina. La Figura 6.7 muestra el mapa obtenido para la

distribución de objetos acorde al laboratorio de Bio-Robótica; por otro lado, en la Figura 6.8 se observa el mapa para la distribución de objetos con diferentes formas.

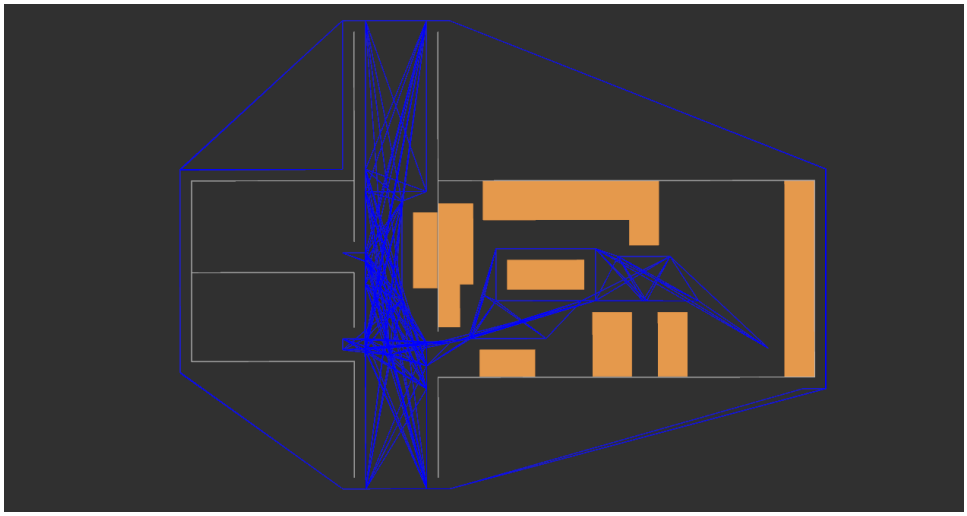


Figura 6.7: Mapa de la distribución de objetos conforme al laboratorio de Bio-Robótica.

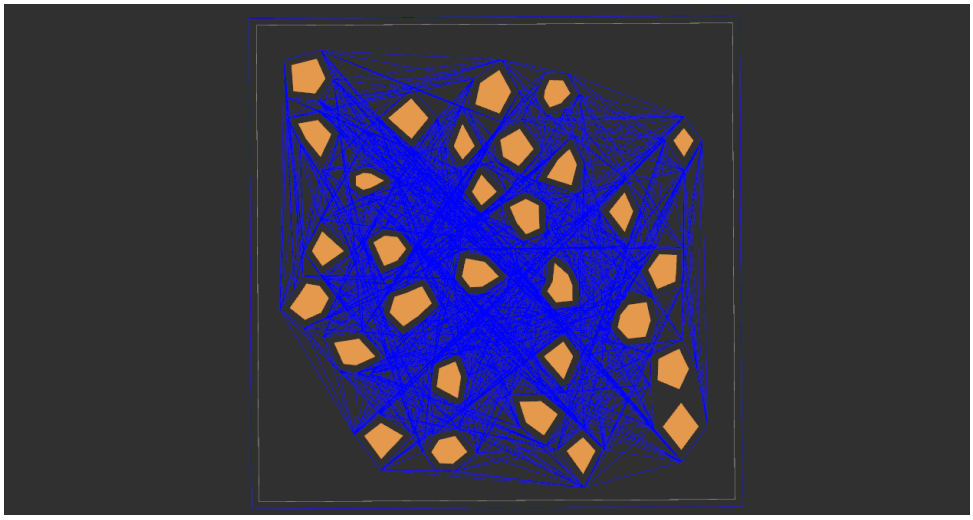


Figura 6.8: Mapa de la distribución de objetos con múltiples formas.

Para el cálculo de la ruta óptima entre la posición del robot y un destino se hizo uso de posiciones predefinidas que se observan de color azul celeste en la Figura 6.9 y Figura 6.10, para las dos distribuciones respectivas. Se hicieron varias pruebas tomando diferentes combinaciones, las cuales se muestran en la Tabla 6.5.

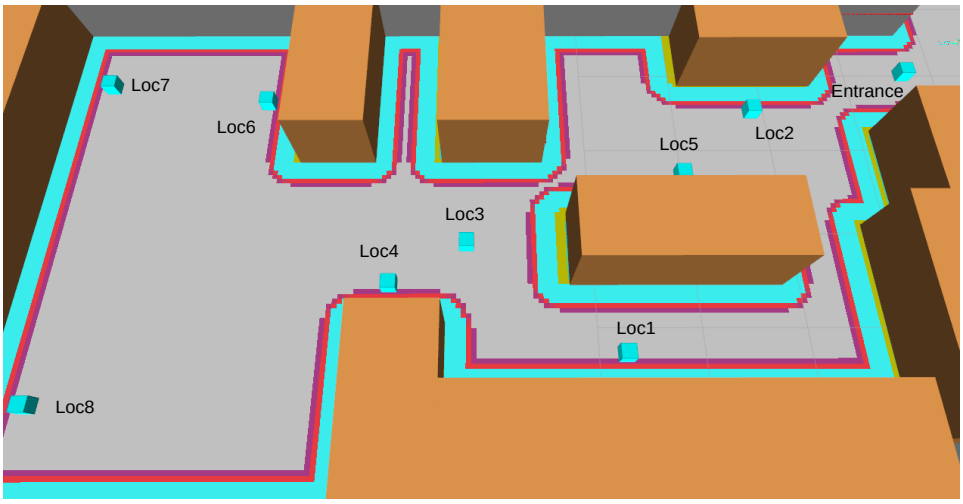


Figura 6.9: Posiciones predefinidas de la distribución del laboratorio de Bio-Robótica.

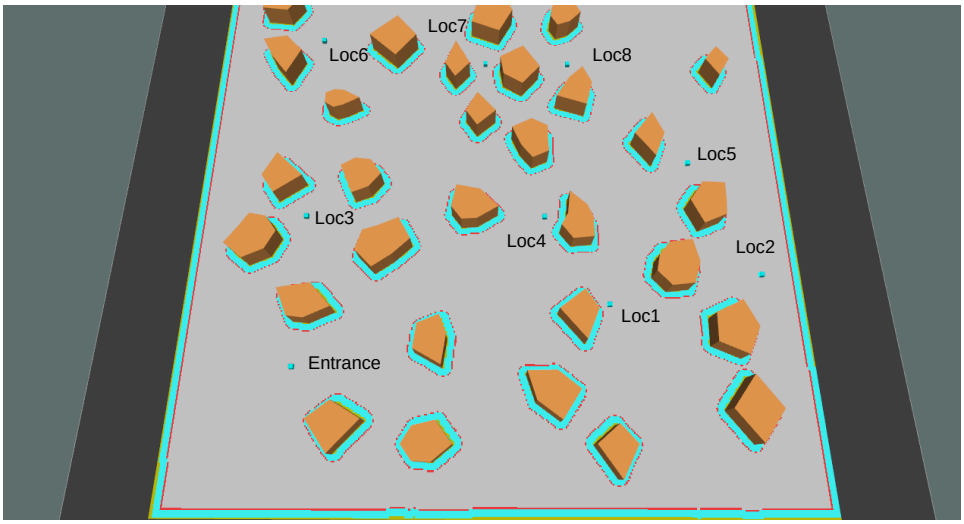
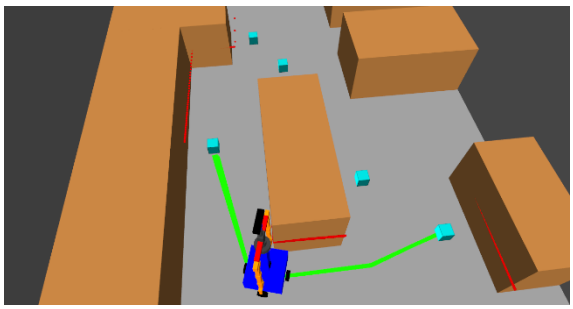


Figura 6.10: Posiciones predefinidas de la distribución con diferentes formas.

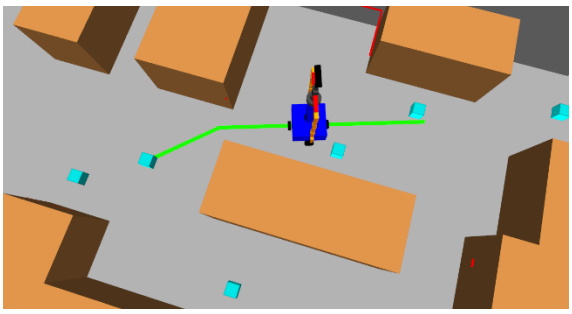
Origen	**Destino**
Loc1	Loc2
Loc2	Loc3
entrance	Loc1
Loc4	Loc5
Loc2	Loc6
Loc1	Loc8
Loc7	entrance

Tabla 6.5: Combinaciones de planeación de rutas.

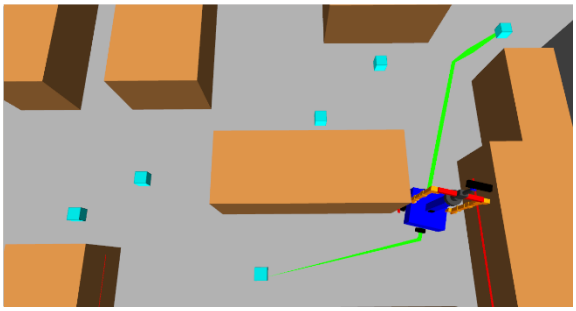
Los resultados de estas pruebas se muestran en la Figura 6.11 y Figura 6.12, para las dos respectivas configuraciones.



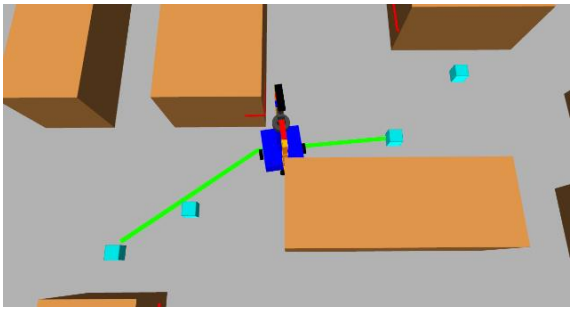
a) Ruta entre Loc1 y Loc2.



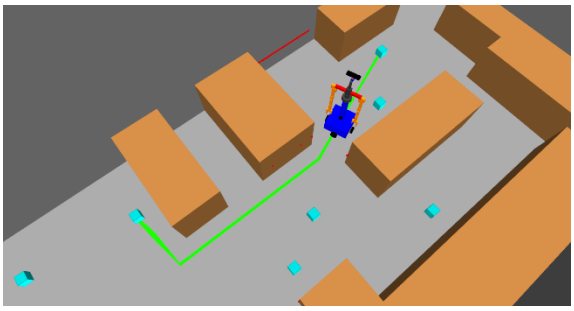
b) Ruta entre Loc2 y Loc3.



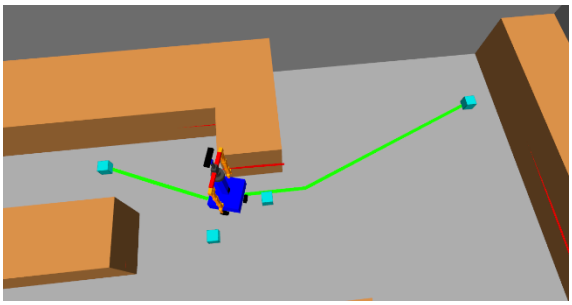
c) Ruta entre entrance y Loc1.



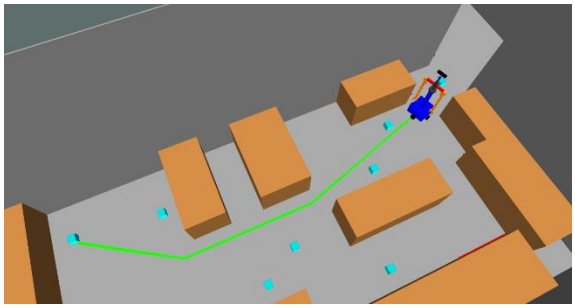
d) Ruta entre Loc4 y Loc5.



e) Ruta entre Loc2 y Loc6.

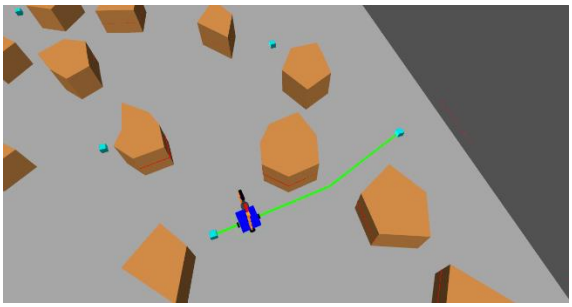


f) Ruta entre Loc1 y Loc8.

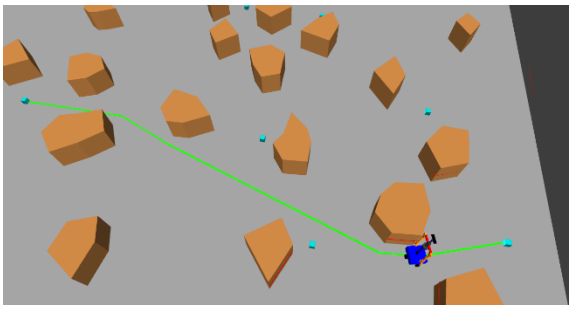


g) Ruta entre Loc7 y entrance.

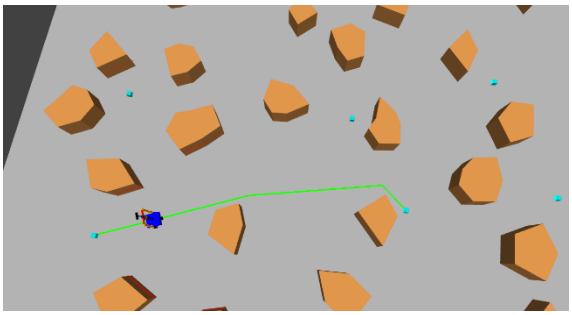
Figura 6.11: Rutas de la distribución de objetos conforme al laboratorio de Bio-Robótica.



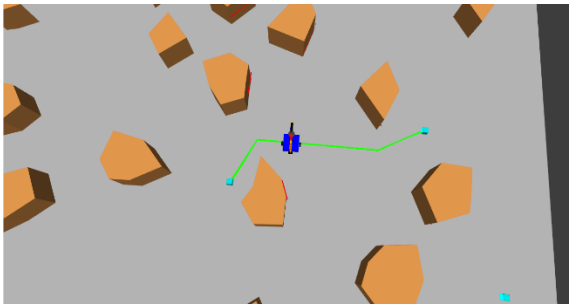
a) Ruta entre Loc1 y Loc2.



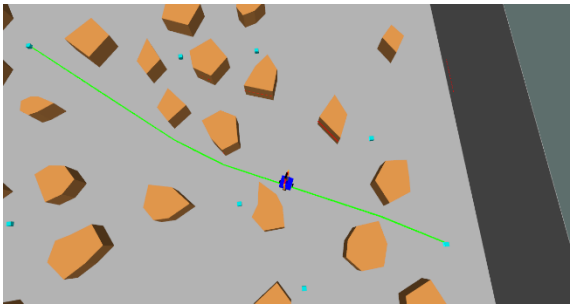
b) Ruta entre Loc2 y Loc3.



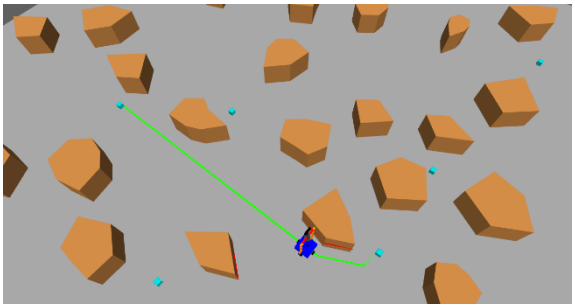
c) Ruta entre entrance y Loc1.



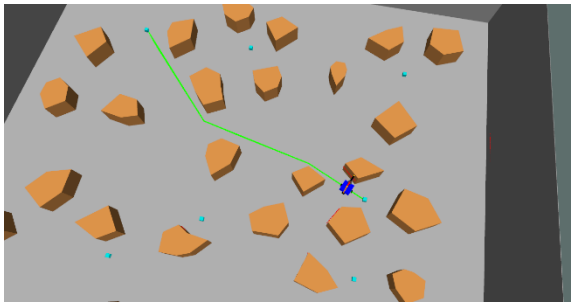
d) Ruta entre Loc4 y Loc5.



e) Ruta entre Loc2 y Loc6.



f) Ruta entre Loc1 y Loc8.



g) Ruta entre Loc7 y entrance.

Figura 6.12: Rutas de la distribución con diferentes formas.

6.2.3. Planeación utilizando rejilla de ocupación.

Con los diagramas de Voronoi correspondientes a las dos configuraciones que se muestran en la Figura 6.13 y Figura 6.14, se utilizó el algoritmo de $A*$ para encontrar las rutas entre las posiciones predefinidas de la Tabla 6.5. La Figura 6.15 y Figura 6.16 muestran el resultado de estas pruebas para los dos respectivos ambientes. La configuración utilizada del nodo de ROS se basó en la geometría del robot Justina, al igual que en las pruebas anteriores se emplearon dos tipos de distribución espacial de objetos: configuración del laboratorio de Bio-Robotica y la que contiene diferentes formas.

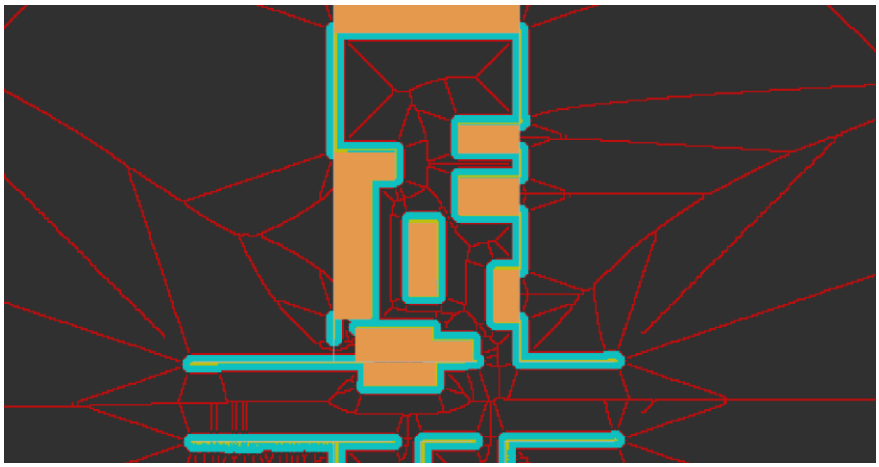


Figura 6.13: Diagrama de Voronoi de la distribución del laboratorio de Bio-Robotica.

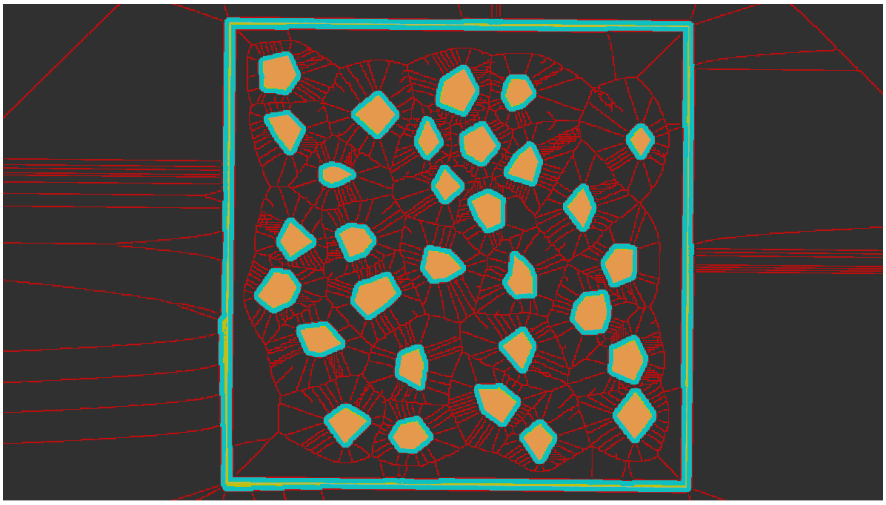
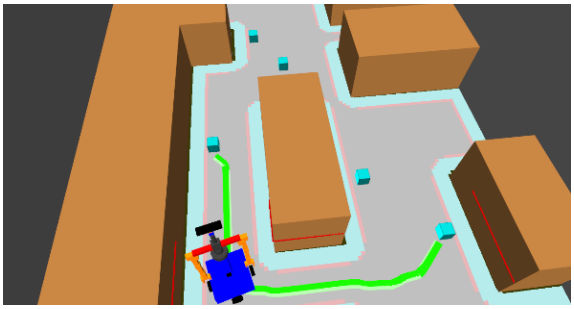
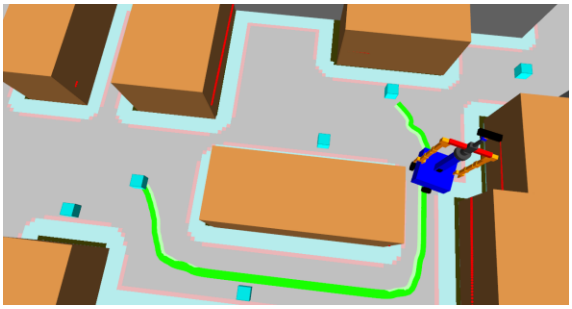


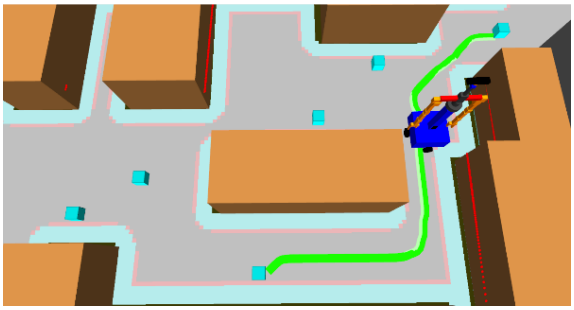
Figura 6.14: Diagrama de Voronoi de la distribución con diferentes formas.



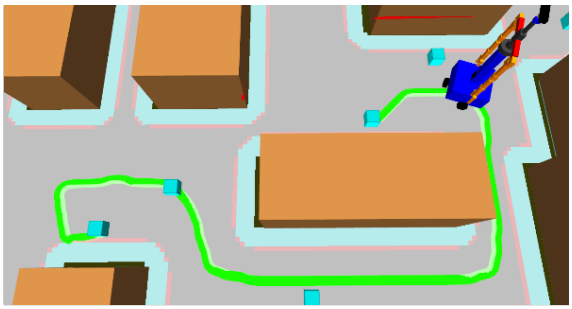
a) Ruta entre Loc1 y Loc2.



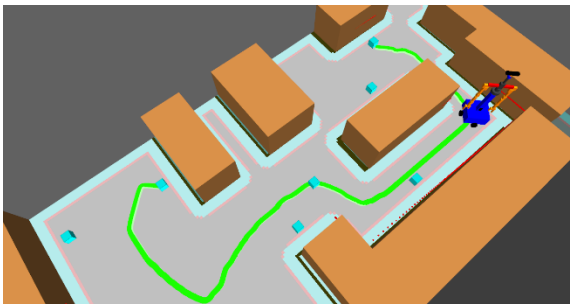
b) Ruta entre Loc2 y Loc3.



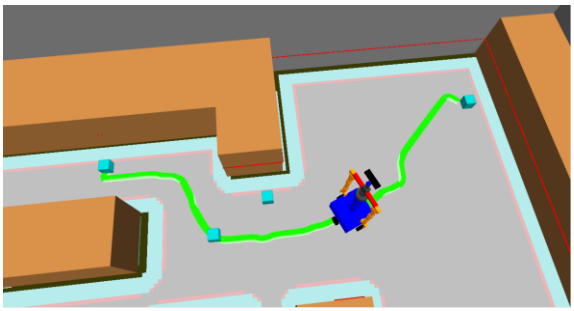
c) Ruta entre entrance y Loc1.



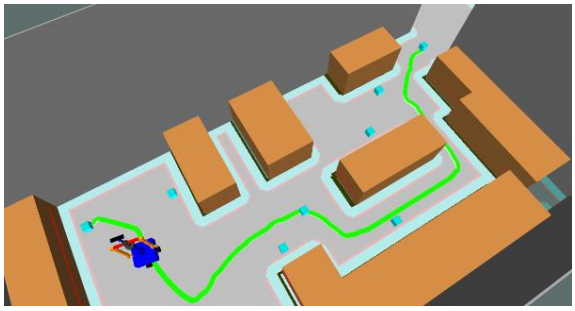
d) Ruta entre Loc4 y Loc5.



e) Ruta entre Loc2 y Loc6.

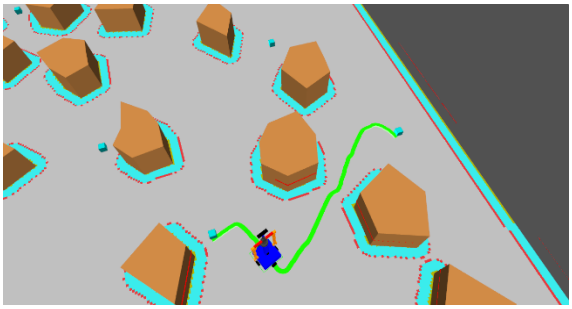


f) Ruta entre Loc1 y Loc8.

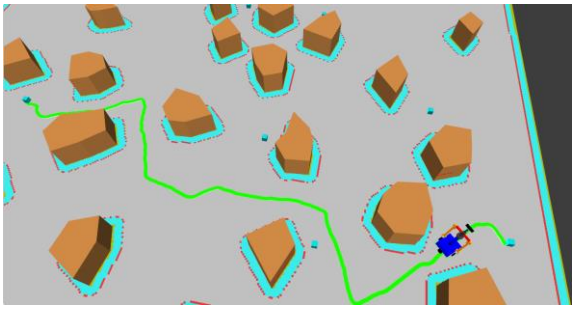


g) Ruta entre Loc7 y entrance.

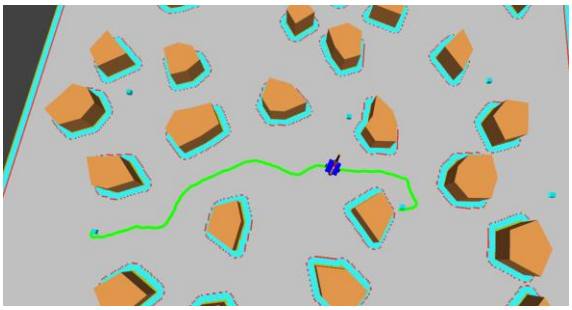
Figura 6.15: Rutas empleando el diagrama de Voronoi de la distribución de objetos conforme al laboratorio de Bio-Robótica.



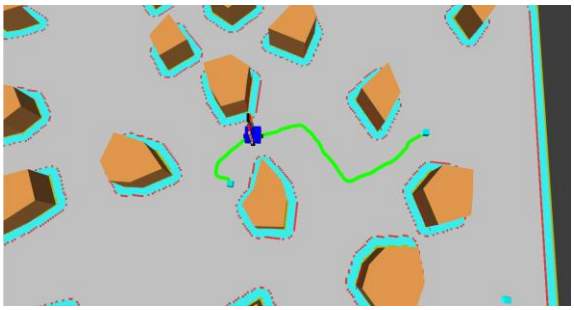
a) Ruta entre Loc1 y Loc2.



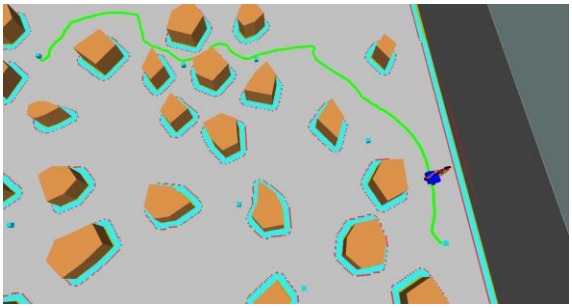
b) Ruta entre Loc2 y Loc3.



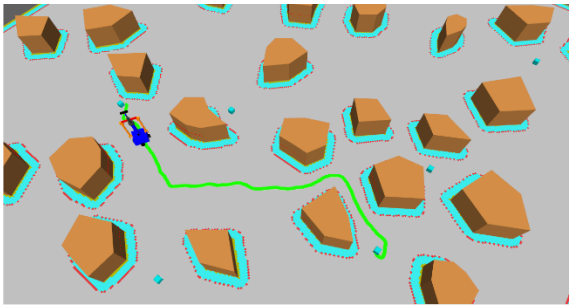
c) Ruta entre entrance y Loc1.



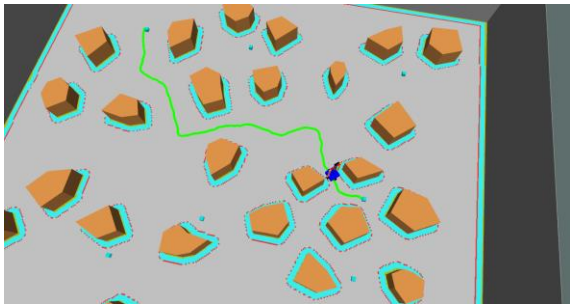
d) Ruta entre Loc4 y Loc5.



e) Ruta entre Loc2 y Loc6.



f) Ruta entre Loc1 y Loc8.

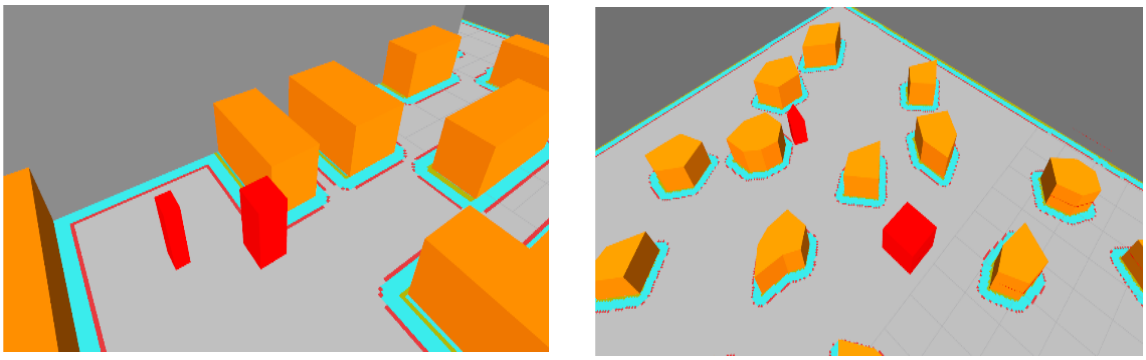


g) Ruta entre Loc7 y entrance.

Figura 6.16: Rutas empleando el diagrama de Voronoi de la distribución con diferentes formas.

6.2.4. Comportamientos híbridos.

Para estas pruebas se colocaron dos objetos desconocidos en las distribuciones del medio ambiente: el primer obstáculo se colocó entre la trayectoria que se deseaba alcanzar, mientras que el segundo obstáculo se colocó muy cercano a un obstáculo conocido, como se observa en la Figura 6.17.

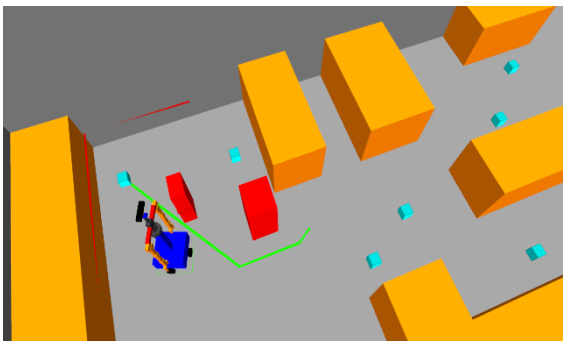


a) Obstáculos desconocidos acorde a la distribución de objetos del laboratorio de Bio-Robótica.

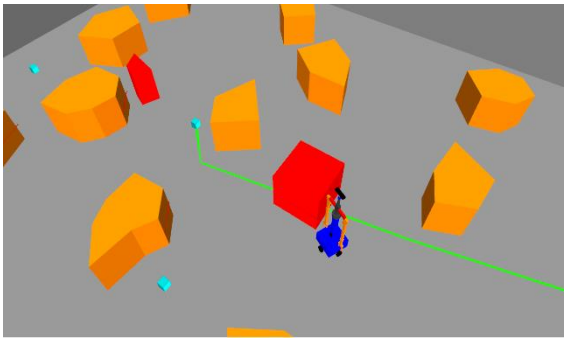
b) Obstáculos desconocidos de la distribución con diferentes formas.

Figura 6.17: Obstáculos desconocidos.

En lo referente a las pruebas realizadas que combinan campos potenciales y planeación de rutas empleando el mapa topológico, se aprecia la evasión de obstáculos desconocidos en la Figura 6.18. Mientras que en la Figura 6.19 se muestra la condición del mínimo local cuando se detecta que no hay movimiento en un periodo establecido. Es preciso destacar que otra condición que se presentó, en la que el robot entra en estado de un mínimo local es cuando el punto objetivo se encuentra contenido en el obstáculo desconocido o muy cercano a este, el robot realiza la planeación optando por seguir otra ruta como se observa en la Figura 6.20.

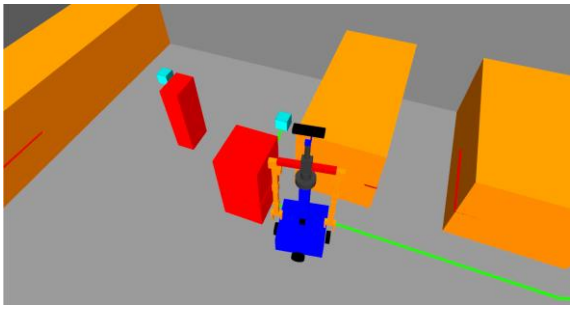


a) Evasión de obstáculos en la distribución de objetos conforme al laboratorio de Bio-Robótica.

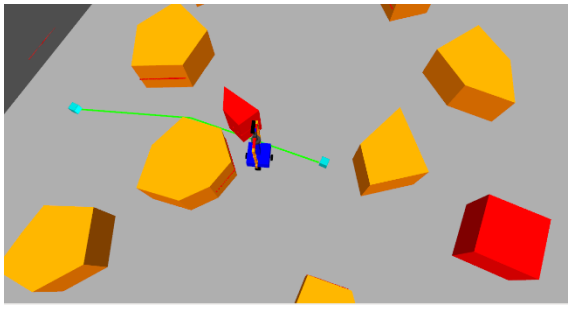


b) Evasión de obstáculos en la distribución de la distribución con diferentes formas.

Figura 6.18: Evasión de obstáculos utilizando campos potenciales.

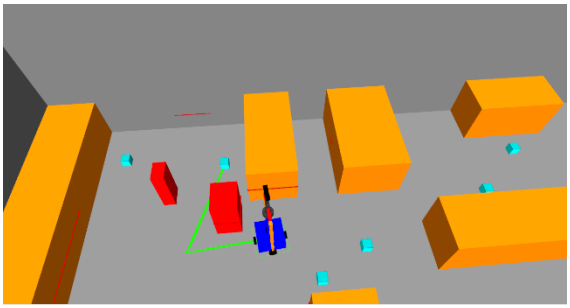


a) Detección de mínimo local en la distribución de objetos conforme al laboratorio de Bio-Robótica.

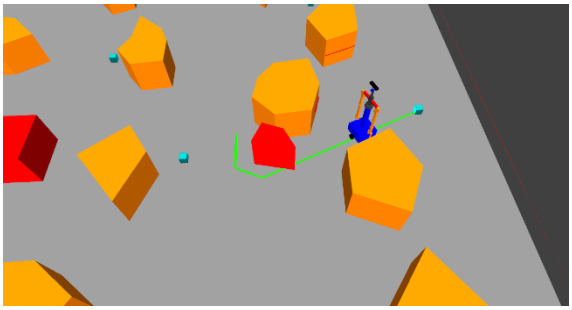


b) Detección de mínimo local en la distribución de la distribución con diferentes formas.

Figura 6.19: Detección de mínimos locales.



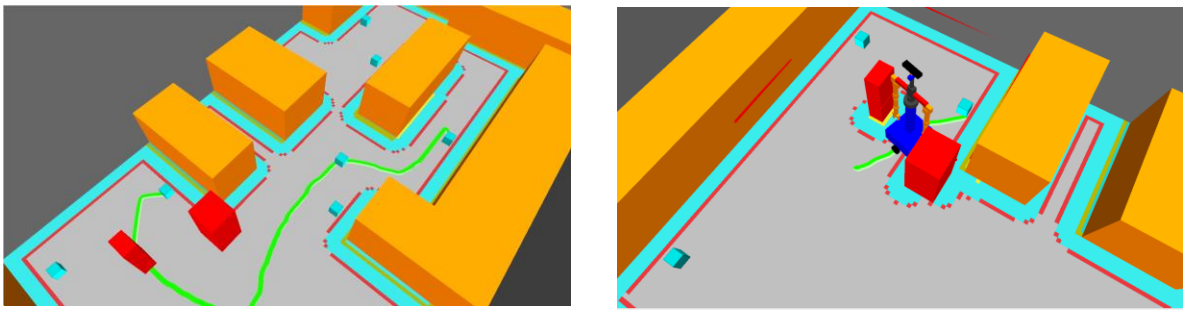
a) Re-planeación en la distribución de objetos conforme al laboratorio de Bio-Robótica.



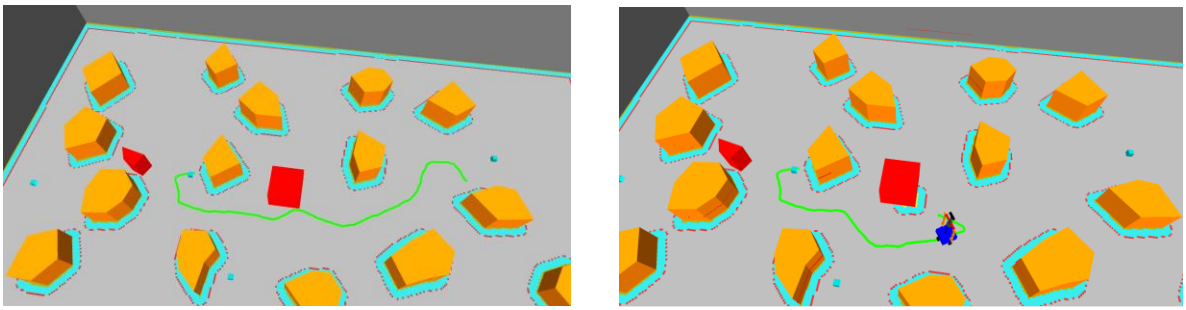
b) Re-planeación en la distribución de la distribución con diferentes formas.

Figura 6.20: Re-planeación de ruta.

Para las pruebas del comportamiento hibrido utilizando una rejilla de ocupación, el cual no hace uso de campos potenciales, el robot evadió el obstáculo desconocido detectándolos por medio del sensor láser; es importante destacar que en este comportamiento considera el mapa actualizado para ejecutar la re-planeación de la ruta. La Figura 6.21 muestra la ejecución de este comportamiento.



a) Re-planeación cuando el obstáculo desconocido impide el recorrido de la ruta utilizando la distribución de objetos conforme al laboratorio de Bio-Robótica.



b) Re-planeación cuando el obstáculo desconocido impide el recorrido de la ruta utilizando la distribución de la distribución con diferentes formas.

Figura 6.21: Re-planeación de ruta empleando la rejilla de ocupación actualizada.

6.3. Pruebas en Justina.

Para comprobar el buen funcionamiento de los algoritmos simulados en el robot Justina, se utilizó únicamente el mapa del laboratorio de Bio-Robótica. Lo que se hizo fue cambiar la configuración del descriptor de lanzamiento de ROS para que se utilizaran los componentes del hardware del robot real, estos componentes son: la base móvil y el sensor láser.

En estas pruebas se le indico a Justina las posiciones deseadas acorde a las pruebas realizadas en las simulaciones mostradas en el apartado 6.2. Los resultados obtenidos concuerdan con la simulación de estos algoritmos. La Figura 6.22 muestra a Justina navegando en el laboratorio de Bio-Robótica.

Figura 6.22: Justina navegando.

Capítulo 7. Conclusiones y trabajo a futuro.

7.1. Conclusiones.

Utilizando el paradigma, arquitectura y herramientas que proporciona ROS, se desarrolló un simulador, en el cual se permite implementar modelos para el control autónomo de robots móviles. Este simulador hace uso de software que el laboratorio de Bio-Robótica de la facultad de ingeniería proporcionó, el cual provee de funciones de bajo nivel para el control simulado de los distintos componentes del robot de servicio Justina.

En lo referente a las implementaciones de la planeación de movimientos utilizando el mapa del medio ambiente, se desarrollaron dos métodos: el que utiliza un mapa topológico y el que hace uso de una rejilla de ocupación. Cada uno tiene sus ventajas y desventajas. Las ventajas de emplear un mapa topológico son: los movimientos entre nodos son largos lo que hace que los movimientos sean más rápidos y siempre se encuentra un mapa que sirve para navegar, el cual se adapta a las dimensiones del robot. La ventaja de utilizar la rejilla de ocupación y el diagrama de Voronoi, es que siempre se encuentra una ruta a la mitad de dos objetos adaptándose a las dimensiones del robot.

Por otra parte las desventajas de utilizar un mapa topológico y campos potenciales son: en el mundo real es difícil contar con la distribución de obstáculos y medidas del mapa, existiendo la posibilidad de que el robot entre en un estado de mínimo local cuando se emplean campos potenciales. Mientras que las desventajas de utilizar una rejilla de ocupación son: las celdas están muy cercanas, lo que implica que los movimientos sean más lentos en comparación en los obtenidos al utilizar un mapa topológico y la unión entre la celda origen y destino con el diagrama de Voronoi no implica que se encuentre la mejor ruta, ya que es posible la existencia de una ruta en donde esta unión no sea la de menor costo.

La mayor ventaja de utilizar el simulador reside en que se puede tener el prototipo de un algoritmo para así tener una aproximación de cómo se comporta en el mundo real. De lo anterior se desprende que, la detección de colisiones, proporciona al desarrollador del algoritmo la capacidad de verificar si la implementación u configuración de éste es acorde al objetivo planteado. En términos generales el objetivo de esta implementación es proveer al robot la cualidad de navegar autónomamente sin que choque. Cabe destacar que una vez desarrollado el simulador, únicamente se enfocó al aprendizaje de las técnicas de inteligencia artificial para el control autónomo de un robot móvil, evitando la necesidad de contar con el robot real y facilitando este aprendizaje.

En general, la hipótesis planteada al inicio se validó completamente, puesto que, mediante el uso del simulador propuesto, se desarrollaron algoritmos para el control del robot de servicio Justina. El control le proporcionó a Justina la habilidad de navegación autónoma, tanto en un ambiente virtual como en el mundo real.

7.2. Trabajo a futuro.

En este apartado se lleva a cabo un análisis de ideas acerca de los puntos que permitan continuar con el desarrollo de la simulación de un robot de servicio. Estas ideas que, por el alcance del presente trabajo, no pudieron ser presentadas y desarrolladas.

Como se observó, la presente investigación se enfocó al desarrollo de un simulador para robots móviles, que posteriormente fue empleado para implementar técnicas de inteligencia artificial utilizadas en la navegación de un robot móvil, las cuales se probaron con la configuración del robot de servicio Justina. No obstante, un robot de servicio no solo es un robot móvil, ya que existen otros componentes con los que cuenta Justina, tales como: cabeza, torso, brazo manipulador y sistema de visión. Estos elementos permiten llevar a cabo tareas de reconocimiento y manipulación de objetos, reconocimiento de personas, seguimiento de personas, etc.

En términos generales, estas son tareas que están relacionadas con pruebas de la competencia internacional de robótica *RoboCup*, que ayudaría en gran medida a tener una herramienta que simule una previa visualización del funcionamiento y desempeño de Justina en estas pruebas.

Apendice A. Interfaz de programación del simulador.

A.1 Navegación.

La clase **NavigationUtil** contiene métodos para la navegación del robot, la cual puede ser utilizada agregando la cabecera **#include "common/NavigationUtil.h"**. En ésta se encuentran los siguientes métodos:

- **void initRosConnection(ros::NodeHandle * n)**

Descripción.

Este método inicializa la referencia del manejador de nodos para la comunicación entre una instancia de este objeto con ROS.

Parámetros.

n – Apuntador al manejador de ROS.

- **void getCurrPose(float &x, float &y, float &theta)**

Descripción.

Método que obtiene la posición y orientación actual del robot.

Parámetros.

x – Coordenada en x correspondiente a la posición actual del robot.

y – Coordenada en y correspondiente a la posición actual del robot.

theta – Angulo de orientación en el eje z.

- **void asyncMoveDist(float dist, bool moveLateral)**

Descripción.

Este método permite realizar movimientos asíncronos frontales o laterales, esto quiere decir que únicamente envía la señal de inicio e inmediatamente se obtiene el control de

ejecución de la tarea sin tener que esperar a la finalización del movimiento. El sentido positivo para el movimiento lateral es a la izquierda.

Parámetros.

dist – Distancia en metros que el robot avanza, si el valor es mayor a cero avanza para adelante y en caso contrario avanza hacia atrás.

moveLateral – Bandera que indica si el movimiento es frontal o lateral, si el valor es 0 el robot avanza de manera frontal y si es igual a 1 avanza lateralmente.

- **bool syncMoveDist(float dist, bool moveLateral, int timeout)**

Descripción.

Este método permite realizar movimientos síncrona frontales o laterales, esto quiere decir que no se retoma el control de ejecución hasta que termine de ejecutar la tarea o se agote el tiempo de espera establecido. El sentido positivo para el movimiento lateral es a la izquierda.

Parámetros.

dist – Distancia en metros que el robot avanza, si el valor es mayor a cero avanza para adelante y en caso contrario avanza hacia atrás.

moveLateral – Bandera que indica si el movimiento es frontal o lateral, si el valor es 0 el robot avanza de manera frontal y si es igual a 1 avanza lateralmente.

timeout – Tiempo en milisegundos para que termine de ejecutarse la tarea, si este valor es igual a cero no regresa el control de ejecución hasta que la tarea se haya completado.

Resultado.

Bandera que indica si la tarea fue exitosa o no.

- **void asyncMoveDistAngle(float dist, float theta)**

Descripción.

Método para llevar a cabo movimientos asíncronos del tipo girar y avanzar, lo cual significa que únicamente envía la señal de inicio e inmediatamente se obtiene el control de ejecución de la tarea sin tener que esperar la finalización del movimiento.

Parámetros.

dist – Distancia en metros que el robot avanza, si el valor es mayor a cero avanza para adelante y en caso contrario avanza hacia atrás.

theta – Angulo que el robot gira, si este valor es positivo el giro es en sentido inverso a las manecillas del reloj y en caso contrario es en sentido horario.

- **bool syncMoveDistAngle(float dist, float theta, int timeout)**

Descripción.

Método para llevar a cabo movimientos síncronos del tipo girar y avanzar, esto quiere decir que no se retoma el control de ejecución hasta que termine de ejecutar la tarea o se agote el tiempo de espera establecido.

Parámetros.

dist – Distancia en metros que el robot avanza, si el valor es mayor a cero avanza para adelante y en caso contrario avanza hacia atrás.

theta – Angulo que el robot gira, si este valor es positivo el giro es en sentido inverso a las manecillas del reloj y en caso contrario es en sentido horario.

timeout – Tiempo en milisegundos para que termine de ejecutar la tarea, si este valor es igual a cero no regresa el control de ejecución hasta que la tarea se haya completado.

Resultado.

Bandera que indica si la tarea fue exitosa o no.

- **void asyncMovePose(float goalX, float goalY, float goalTheta, bool correctAngle)**

Descripción.

Este método permite la navegación asíncrona del robot a una posición en el plano z = 0 respecto al sistema de referencia global; además tiene la opción de establecer una orientación al llegar al punto establecido.

Parámetros.

goalX – Coordenada en x correspondiente a la posición deseada del robot.

goalY – Coordenada en y correspondiente a la posición deseada del robot.

goalTheta – Orientación deseada del robot.

correctAngle – Bandera que indica si se desea considerar el ángulo de orientación, si este valor es igual a 0 únicamente intenta llegar al punto establecido y cuando es igual a 1 el robot se colocará con la orientación que se definió en el parámetro **goalTheta**.

- **bool syncMovePose(float goalX, float goalY, float goalTheta, bool correctAngle, int timeout)**

Descripción.

Este método permite la navegación síncrona del robot a una posición en el plano z = 0, respecto al sistema de referencia global; además tiene la opción de establecer una orientación al llegar al punto establecido.

Parámetros.

goalX – Coordenada en x correspondiente a la posición deseada del robot.

goalY – Coordenada en y correspondiente a la posición deseada del robot.

goalTheta – Orientación deseada del robot.

correctAngle – Bandera que indica si se desea considerar el ángulo de orientación, si este valor es igual a 0 únicamente intenta llegar al punto establecido y cuando es igual a 1 el robot se colocará con la orientación que se definió en el parámetro **goalTheta**.

timeout – Tiempo en milisegundos para que termine de ejecutar la tarea, si este valor es igual a cero no regresa el control de ejecución hasta que la tarea se haya completado.

Resultado.

Bandera que indica si la tarea fue exitosa o no.

- **void asyncMovePath(nav_msgs::Path path)**

Descripción.

Método que permite la navegación asíncrona del robot a una secuencia de posiciones.

Parámetros.

path – Secuencia de posiciones a las cuales el robot debe navegar.

- **bool syncMovePath(nav_msgs::Path path, int timeout)**

Descripción.

Método que permite la navegación síncrona del robot a una secuencia de posiciones.

Parámetros.

path – Secuencia de posiciones a las cuales el robot debe navegar.

timeout – Tiempo en milisegundos para que termine de ejecutar la tarea, si este valor es igual a cero no regresa el control de ejecución hasta que la tarea se haya completado.

Resultado.

Bandera que indica si la tarea fue exitosa o no.

- **void asyncPotentialFields(float goalX, float goalY)**

Descripción.

Este método sirve para iniciar la navegación asíncrona utilizando campos potenciales. El robot alcanza la posición deseada desde su posición actual.

Parámetros.

goalX – Coordenada en x correspondiente a la posición deseada del robot.

goalY – Coordenada en y correspondiente a la posición deseada del robot.

- **bool syncPotentialFields(float goalX, float goalY, int timeout)**

Descripción.

Este método sirve para iniciar la navegación síncrona utilizando campos potenciales. El robot alcanza la posición deseada desde su posición actual.

Parámetros.

goalX – Coordenada en x correspondiente a la posición deseada del robot.

goalY – Coordenada en y correspondiente a la posición deseada del robot.

timeout – Tiempo en milisegundos para que termine de ejecutar la tarea, si este valor es igual a cero no regresa el control de ejecución hasta que la tarea se haya completado.

Resultado.

Bandera que indica si la tarea fue exitosa o no.

- **bool finishedCurrMotionDist()**

Descripción.

Método para monitorear el estado de ejecución del método **asyncMoveDist**.

Resultado.

Bandera para monitorear el estado de la última tarea en ejecución.

- **bool finishedCurrMotionDistAngle()**

Descripción.

Método para monitorear el estado de ejecución del método **asyncMoveDistAngle**.

Resultado.

Bandera para monitorear el estado de la última tarea en ejecución.

- **bool finishedCurrMotionPose()**

Descripción.

Método para monitorear el estado de ejecución del método **asyncMovePose**.

Resultado.

Bandera para monitorear el estado de la última tarea en ejecución.

- **bool finishedCurrMotionPath()**

Descripción.

Método para monitorear el estado de ejecución del método **asyncMovePath**.

Resultado.

Bandera para monitorear el estado de la última tarea en ejecución.

- **bool finishedCurrMotionPF()**

Descripción.

Método para monitorear el estado de ejecución del método **asyncPotentialFields**.

Resultado.

Bandera para monitorear el estado de la última tarea en ejecución.

- **void stopMotion()**

Descripción.

Este método detiene todas las tareas de navegación que se encuentran en ejecución.

A.2 Planeación de movimientos.

La clase **PathPlanningUtil** contiene métodos para obtener un plan de movimientos entre una posición inicial y una final, existe dos opciones disponibles para obtener el plan: utilizando un mapa topológico y una celda de ocupación. Ésta puede ser utilizada agregando la cabecera **#include "common/PathPlanningUtil.h"**, y en la cual se encuentran los siguientes métodos:

- **void initRosConnection(ros::NodeHandle * n)**

Descripción.

Este método inicializa la referencia del manejador de nodos para la comunicación entre una instancia de este objeto con ROS.

Parámetros.

n – Apuntador al manejador de ROS.

- **nav_msgs::Path planPathSymbMapDjsk(float startX, float startY, float goalX, float goalY, std::vector<geometry_msgs::Polygon> polygons, bool& success)**

Descripción.

Método que obtiene un plan de movimientos que debe seguir el robot desde una posición inicial hasta una final, el cual utiliza un mapa topológico, éste además recibe una arreglo de polígonos que se deseen considerar para obtener dicho plan.

Parámetros.

startX – Coordenada en x correspondiente a la posición inicial.

startY – Coordenada en y correspondiente a la posición inicial.

goalX – Coordenada en x correspondiente a la posición deseada.

goalY – Coordenada en y correspondiente a la posición deseada.

polygons – Arreglo de polígonos que se deseen incluir en el mapa.

success – Bandera que indica si fue posible encontrar un plan de movimientos.

Resultado.

Arreglo de posiciones para llegar de un punto a otro.

- **nav_msgs::Path planPathGridMap(float startX, float startY, float goalX, float goalY, bool& success)**

Descripción.

Método que obtiene un plan de movimientos que debe seguir el robot desde una posición inicial hasta una final, el cual utiliza una celda de ocupación espacial.

Parámetros.

startX – Coordenada en x correspondiente a la posición inicial.

startY – Coordenada en y correspondiente a la posición inicial.

goalX – Coordenada en x correspondiente a la posición deseada.

goalY – Coordenada en y correspondiente a la posición deseada.

success – Bandera que indica si fue posible encontrar un plan de movimientos.

Resultado.

Arreglo de posiciones para llegar de un punto a otro.

A.3 Adquisición de las lecturas del sensor láser.

La clase **LaserScanUtil** contiene métodos para obtener los datos actualizados del sensor láser, la cual puede ser utilizada agregando la cabecera **#include <common/LaserScanUtil.h>**. En ésta se encuentran los siguientes métodos:

- **void initRosConnection(ros::NodeHandle * n)**

Descripción.

Este método inicializa la referencia del manejador de nodos para la comunicación entre una instancia de este objeto con ROS.

Parámetros.

n – Apuntador al manejador de ROS.

- **biorobotics::LaserScan * getLaserScan()**

Descripción.

Método que obtiene la lectura más actualizada del sensor láser.

Resultado.

Lectura del sensor láser.

A.4 Obtención de estructura del medio ambiente.

La clase **EnvironmentUtil** contiene métodos para obtener la estructura del medio ambiente, la cual puede ser utilizada agregando la cabecera **#include "common/EnvironmentUtil.h"**. En ésta se encuentran los siguientes métodos:

- **void initRosConnection(ros::NodeHandle * n)**

Descripción.

Este método inicializa la referencia del manejador de nodos para la comunicación entre una instancia de este objeto con ROS.

Parámetros.

n – Apuntador al manejador de ROS.

- **std::vector<geometry_msgs::Polygon> getKnownPolygons()**

Descripción.

Método que obtiene únicamente la estructura de la representación del medio ambiente que ya es conocido.

Resultado.

Arreglo de polígonos que representan el medio ambiente.

- **std::vector<geometry_msgs::Polygon> getAllPolygons()**

Descripción.

Método que obtiene la estructura de la representación del medio ambiente, el cual está compuesto por la estructura ya conocida y la simulación de objetos no conocidos.

Resultado.

Arreglo de polígonos que representan el medio ambiente.

Bibliografía.

[1] G. Dudek y M. Jenkin, Computational principles of mobile robotics, Cambridge university press., 2010.

[2] R. C. Arkin, Behavior-based robotics, MIT press, 1998.

[3] S. Carpin, M. Lewis, J. Wang, S. Balakirsky y C. Scrapper, «USARSim: a robot simulator for research and education. Robotics and Automation,» *IEEE International Conference on Robotics and Automation*, 2007.

[4] M. Zaratti, M. Fratarcangeli y L. Locchi, «A 3D simulator of multiple legged robots based on USARSim.,» *Robocup 2006: Robot Soccer World Cup*, pp. 13-24, 2007.

[5] J. Savage, M. Billinghurst y A. Holden, «The virbot: a virtual reality robot driven with multimodal commands.,» *Expert Systems with Applications*, pp. 413-419, 1998.

[6] J. Albus, «Outline for a theory of intelligence. Systems, Man and Cybernetics,» *IEEE Transactions on*, pp. 480-501, 1991.

[7] R. A. Brooks y J. Connell, Asynchronous Distributed Control System for a Mobile Robot, Storage and Retrieval for Image and Video Databases, 1986.

[8] V. Vonásek, D. Fišer, K. Košnar y L. Přeučil, «A Light-Weight Robot Simulator for Modular Robotics,» *Modelling and Simulation for Autonomous Systems*, pp. 206-216, 2014.

[9] E. Angel y D. Shreiner, Interactive Computer Graphics (6th Edition), Addison-Wesley, 2011.

[10] R. Siegwart y I. Nourbakhsh, Introduction to Autonomous Mobile Robots, The MIT Press, 2004.

[11] H. R. Everett, Sensors for mobile robots, A K Peters, Ltd, 1995.

[12] E. Christer, Real-Time Collision Detection, Elsevier, 2005.

[13] M. Berg, C. Otfried, M. Van Kreveld y M. Overmars, Computational Geometry Algorithms and Applications (Third Edition), Springer, 2008.

[14] «ROS wiki,» 14 Noviembre 2015. [En línea]. Available: http://wiki.ros.org/.

[15] D. H. Greene, «The decomposition of polygons into convex parts,» *Computational Geometry*, vol. volume 1 of Adv. Comput. Res., p. 235–259, 1983.

[16] «The Computational Geometry Algorithms Library,» [En línea]. Available: http://www.cgal.org/. [Último acceso: 3 Mayo 2016].

[17] M. Negrete, J. Savage, J. Cruz y J. Márquez, «Behavior-Based Navigation System for a Service Robot with a Mechatronic Head.,» *XVI Congreso Latinoamericano de Control Automático*, 2014, Cancún, México..

[18] J. R. Andrews y N. Hogan, «Impedance Control as a Framework for Implementing Obstacle Avoidance in a Manipulator,» de *Control of Manufacturing Processes and Robotic Systems*, Boston, Eds. Hardt, D. E. and Book W., ASME, 1983, pp. 243-251.

[19] F. Arambula y M. A. Padilla, «Autonomous Robot Navigation using Adaptive Potential Fields,» *Mathematical and Computer Modelling*, pp. 1141-1156, 2003.

[20] C. Sprunk, B. Lau y W. Burgard, «Efficient Grid-Based Spatial Representations for Robot Navigation in Dynamic Environments,» *Robotics and Autonomous Systems*, pp. 1116-1130, 2013.

[21] H. L. Akin, N. Ito, A. Jacoff, A. Kleiner, J. Pellez y J. Visser, «Robocup rescue and simulation leagues,» *AI Magazine*, pp. 78-86, 2012.

[22] R. C. Arkin, «Motor Schema-Based Mobile Robot Navigation,» *The International Journal of Robotics Research*, pp. 92-112, August 1989.

[23] R. A. Brooks, «A Robust Layered Control System for a Mobile Robot,» *IEEE Journal of Robotics and Automation*, vol. 1, pp. 14-23, 1986.

[24] OpenSLAM.org, «OpenSLAM,» [En línea]. Available: http://openslam.org/gmapping.html. [Último acceso: 2 Junio 2016].