UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

ESCUELA TÉCNICA
SUPERIOR INGENIEROS
INDUSTRIALES VALENCIA

TRABAJO FIN DE GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

LOCALIZACIÓN DE ROBOTS MÓVILES BASADO EN FILTROS DE PARTÍCULAS CON V-REP

AUTOR: ALEJANDRO MORA FONTELLES

TUTOR: LEOPOLDO ARMESTO ÁNGEL

COTUTOR: VICENT GIRBÉS JUAN

Curso Académico: 2013-14

ESCUELA TÉCNICA
SUPERIOR INGENIEROS
INDUSTRIALES VALENCIA

INDICE GENERAL

I. MEMORIA DESCRIPTIVA
II. PRESUPUESTO
III. ANEXOS DEL PROYECTO

ESCUELA TÉCNICA
SUPERIOR INGENIEROS
INDUSTRIALES VALENCIA

MEMORIA

Tabla de contenido

1 OBJETO DEL TRABAJO 5
2 INTRODUCCIÓN AL PROBLEMA 5
2.1 Antecedentes 5
2.1.1 Robots móviles 5
2.1.1.1 Robots con ruedas 6
2.1.1.2 Robots con patas 7
2.2 Motivación 7
2.3 Justificación académica 8
2.4 Alcance 8
3 ESTADO DEL ARTE 8
3.1 Problemas de localización 8
3.2 Métodos de localización 9
3.3 Tipos de sensores 11
3.3.1 Sensor ultra sonidos 11
3.3.2 Sensor por odometría 12
3.3.3 Sensor laser 12
3.3.4 Sensor de video 12
3.4 Algoritmos para la localización 12
3.4.1 Filtro de Bayes 13
3.4.2 Algoritmo de Markov 14
3.4.3 Algoritmo Filtro de Kalman 15
3.4.4 Algoritmo Filtro de Kalman Extendido 18
3.4.5 Algoritmo Grid 19
3.4.6 Algoritmo Montecarlo 21
4 SOLUCIONES ALTERNATIVAS 23
5 SOLUCION PROPUESTA 24
5.1 METODO MONTECARLO 24
5.1.1 Muestreo aleatorio de partículas 25
5.1.2 Calculo del desplazamiento aproximado por odometría 25
5.1.3 Ponderación de las muestras 29
5.1.4 Re muestreo de las partículas o actualización según el peso de cada partícula 35
6 SIMULACIÓN VREP 39

6.1 Entorno de Trabajo 39
6.1.1 Simulador V-REP 39
6.1.2 Robot utilizado 40
6.2 Modificaciones realizadas sobre el modelo predeterminado 41
6.3 Representación de las partículas 43
6.4 Implementación del Filtro Monte Carlo 43
6.4.1 Función para el muestreo aleatorio 43
6.4.2 Función para la odometría 45
6.4.3 Función para la ponderación de las partículas 49
6.4.4 Función para el re-muestreo de las partículas 51
6.5 Resultados 53
7 SIMULACION ROS-VREP 59
7.1 Herramienta ROS 59
7.2 Combinación ROS con VREP 59
7.3 Paquete de navegación AMCL 59
7.4 Conexión VREP y ROS 60
7.5 Preparación del entorno de simulación 61
7.6 Implementación del código de comunicación 61
7.7 Resultados 65
8 CONCLUSIÓN 68
9 BIBLIOGRAFIA 69

INDICE DE FIGURAS

Figura 2-1 Robot limpiador Scooba irobot 450 6
Figura 2-2 Robot militar anti-explosivos 6
Figura 2-3 Prototipo robot espacial 6
Figura 2-4 Robot médico RP-VITA 6
Figura 2-5 Robot araña Hexapod Phoenix........ 7
Figura 2-6 Robot humanoide ASIMO Honda 7
Figura 3-1 Encoder odometría 9
Figura 3-2 Sistema GPS........ 11
Figura 3-3 Método triangulación 11
Figura 3-4 Algoritmo basado en el filtro de Bayes 13
Figura 3-5 Algoritmo de localización Markov........ 14
Figura 3-6 Algoritmo basado en filtro de Kalman 16
Figura 3-7 Gráficas de la respuesta del algoritmo Filtro de Kalman 17
Figura 3-8 Evolución de la estimación de la posición con algoritmo EKF 19
Figura 3-9 Evolución de la estimación de la posición con algoritmo Grid 20
Figura 5-1 Algoritmo AMCL 24
Figura 5-2 Algoritmos para generación de una distribución aleatoria 25
Figura 5-3 Algoritmo sample_model_odometry........ 27
Figura 5-4 Transformaciones movimiento robot 27
Figura 5-5 Distribución de las posiciones tras movimiento 29
Figura 5-6 Algoritmo beam_range_finder........ 29
Figura 5-7 Distribución phit........ 31
Figura 5-8 Distribución pshort........ 32
Figura 5-9 Distribución pmax 33
Figura 5-10 Distribución prand........ 34
Figura 5-11 Distribución resultante de phit, pmax, prand, pshort 35
Figura 5-12 Función probabilidad acumulada........ 36
Figura 5-13 Re-muestreo de partículas........ 37
Figura 5-14 Evolución de las partículas en un problema de localización........ 37
Figura 6-1 Entorno V-REP 39
Figura 6-2 Robot Pioneer P3-dx real 40
Figura 6-3 Modelo 3d del robot Pioneer P3-dx........ 40
Figura 6-4 Algoritmo Brainterberg 41
Figura 6-5 Robot con láser incorporado........ 41
Figura 6-6 Laser Hokuyo URG........ 41
Figura 6-7 Sensor laser predeterminado........ 42
Figura 6-8 Sensor laser tras modificación 42
Figura 6-9 Función para generar número aleatorio........ 43
Figura 6-10 Código que marca parte de inicialización 44
Figura 6-11 Código para inicializar posición partículas........ 44
Figura 6-12 Distribución inicial partículas 45
Figura 6-13 Inicialización de los manejadores de cada objeto 45
Figura 6-14 Función normalizar ángulo........ 46

Figura 6-15 Función para calcular diferencia entre ángulos 47
Figura 6-16 Código para cálculo posición por odometría 47
Figura 6-17 Código algoritmo sample_model_odometry 48
Figura 6-18 Código para cambiar la posición de las partículas y actualizar valores 49
Figura 6-19 Función pshort 50
Figura 6-20 Función pmax 50
Figura 6-21 Función phit 50
Figura 6-22 Función prand 51
Figura 6-23 Normalización vector probabilidad de las partículas 51
Figura 6-24 Función cumprod y fliplr 52
Figura 6-25 Función resampling 52
Figura 6-26 Implementación función re-muestreo 53
Figura 6-27 Distribución inicial 50 partículas 54
Figura 6-28 Distribución 50 partículas tras varias etapas de re-muestreo 54
Figura 6-29 Distribución 50 partículas después de varias convergencias 55
Figura 6-30 Posición estimada por algoritmo AMCL con 50 partículas 55
Figura 6-31 Distribución inicial 120 partículas 56
Figura 6-32 Distribución 120 partículas tras varias convergencias 57
Figura 6-33 Distribución de las 120 partículas 57
Figura 6-34 Posición final estimada por AMCL 58
Figura 7-1 Publicación de tópicos 61
Figura 7-2 Publicación de tópicos para odometría 62
Figura 7-3 creación archivo launch de slam_gmapping 62
Figura 7-4 Comando para guardar el mapa en un archivo 63
Figura 7-5 Mapa del entorno 63
Figura 7-6 Archivo que contiene meta-datos del mapa 63
Figura 7-7 Archivo para iniciar nodo AMCL 64
Figura 7-8 Herramienta RVIZ 64
Figura 7-9 Resultado Instante inicial- VREP 65
Figura 7-10 Resultado Instante inicial- RVIZ 65
Figura 7-11 Resultados en entorno simulación VREP 66
Figura 7-12 Resultados en herramienta RVIZ 66
Figura 7-13 Resultado robot localizado VREP 67
Figura 7-14 Resultado robot localizado RVIZ 67

1 OBJETO DEL TRABAJO

El objetivo fundamental del presente proyecto consiste en el estudio, la implementación y la simulación de un algoritmo para la localización en un problema de robótica móvil. Además, como objetivos personales cabe destacar:

- Adquirir conocimientos en el campo de la robótica
- Aprender técnicas para documentarse
- Conocer nuevos entornos de programación
- Establecer objetivos concretos para la resolución de problemas

2 INTRODUCCIÓN AL PROBLEMA

La robótica está en un proceso de constante evolución en la que paso a paso se están consiguiendo robots cada vez más sofisticados y complejos. Esto se está consiguiendo gracias a la investigación en los distintos campos científicos que abarca la robótica. Existen tres tipos principales de robots: los robots estáticos, los robots semi-estáticos (permiten ciertos movimientos) y los robots móviles. Los robots estáticos y semi-estáticos son aquellos que se utilizan para realizar numerosas operaciones pero cuya actividad no depende de un desplazamiento íntegro del robot en sí. Los robots móviles autónomos son aquellos que operan en el mundo real, es decir, en un entorno dinámico. Suelen emplearse en plantas industriales para el transporte de mercancías y para la exploración de lugares de difícil acceso o muy distantes, como es el caso de la exploración espacial y de las investigaciones o rescates submarinos entre otros.

Para poder operar en un entorno indeterminista, como lo es el mundo real, los robots necesitan conocer su localización en ese entorno con el fin de poder desplazarse y realizar las funciones deseadas. Así pues, gran parte de los estudios en la robótica se centran en la mejora de este proceso de localización ya que es de vital importancia para el correcto funcionamiento de los robots móviles.

2.1 Antecedentes

2.1.1 Robots móviles

Como ya se ha comentado anteriormente, existen los robots llamados estáticos, semi-estáticos y los robots móviles. Los primeros y los segundos de ellos ya se usan en numerosas aplicaciones obteniendo resultados muy positivos. En cuanto a los terceros, los robots móviles, estos se encuentran todavía en una etapa de desarrollo que avanza mucho más lentamente debido a la complejidad del problema que presenta el desplazamiento. No obstante, existen en la actualidad una gran variedad de robots móviles capaces de realizar desde movimientos básicos hasta movimientos algo más complejos.

Dentro de la familia de los robots móviles podemos diferenciar dos tipos de robots, que son capaces de desplazarse, los robots con ruedas y los robots con patas.

2.1.1.1 Robots con ruedas

Los robots con ruedas son utilizados en numerosas aplicaciones como por ejemplo para la limpieza de locales, para la seguridad de algunos recintos, para hospitales, investigación espacial o para actividades militares. Dependiendo del campo en el que se utilizan así como del presupuesto disponible existen robots más o menos complejos. La ventaja de este tipo de robots es que al utilizar ruedas para el desplazamiento es fácil conseguir una buena estabilidad. Son capaces de desplazarse perfectamente por terrenos lisos. El problema puede aparecer cuando se trabaja en entornos donde el suelo posee cierta rugosidad y montículos.

Este problema se puede resolver adaptando el tipo de ruedas al entorno de trabajo del robot. Además, al igual que en los coches, el robot puede estar capacitado con un sistema básico de suspensión. Con esto la adaptación a un terreno algo más irregular es sencilla. El inconveniente de esta clase de robots, los robots móviles con ruedas, es que, generalmente, solo pueden desplazarse en planos horizontales. No son capaces por ejemplo de subir unas escaleras o cualquier otro tipo de montículo. Es por ello que se han desarrollado a la largo de la evolución de la robótica, robots capacitados con patas para solucionar este problema.

Figura 2-1 Robot limpiador Scooba irobot 450

Figura 2-2 Robot militar anti-explosivos

Figura 2-3 Prototipo robot espacial

Figura 2-4 Robot médico RP-VITA

2.1.1.2 Robots con patas

Como se ha comentado arriba, estos robots pueden llegar a ser capaces de superar obstáculos como podría ser unas escaleras. Lo que se intenta conseguir con estos robots es intentar imitar el movimiento del ser humano o de cualquier otro animal que posee patas. De esta forma se quiere conseguir que estos puedan ser capaces de imitar movimientos humanos. Uno de los problemas más evidentes, es lograr una buena estabilidad. Y es que así como los animales y el ser humano poseen centenares de músculos y articulaciones que les permiten mantenerse en equilibrio, en la robótica es muy difícil lograr este mismo efecto ya que el número de articulaciones que poseen y la cantidad de movimientos que son capaces de realizar son bastante limitados.

Figura 2-6 Robot humanoide ASIMO Honda

Figura 2-5 Robot araña Hexapod Phoenix

2.2 Motivación

Hoy en día, se utilizan cada vez más y más robots para realizar tareas repetitivas o peligrosas en las distintas industrias existentes. Un claro ejemplo son las fábricas de coches donde la mayor parte de las actividades son realizadas por brazos robóticos. De esta forma se ha conseguido acelerar los procesos disminuyendo notablemente los tiempos de producción. Estos tipos de robots son estáticos, es decir, no necesitan desplazarse para realizar las funciones para las cuales han sido diseñados.

No es el caso de los robots móviles cuyo principal fin es el de permitir el desplazamiento del mismo, ya sea gracias a unas ruedas o patas mecánicas integradas en la estructura robótica, por el entorno para el cual ha sido diseñados. Esto complica el problema ya que en este caso el robot necesitará saber dónde se encuentra para poder operar correctamente.

La localización constituye uno de los principales problemas en el campo de la robótica. El ruido que aparece en los distintos dispositivos sensoriales así como los inevitables errores y aproximaciones que se cometen en los modelos empleados hace que la auto-localización sea un problema difícil de resolver. Es entonces importante investigar esta área con el fin de conseguir mejoras en el funcionamiento de los robots. Dicho esto, es interesante conocer los distintos métodos que nos permiten conseguir unos resultados de auto-localización satisfactorios.

2.3 Justificación académica

El presente proyecto se ha realizado tanto para poder así obtener el título de graduado en Ingeniería de Tecnologías Industriales como para poder acceder al Master de Ingeniería de Tecnologías Industriales. Con la realización de este proyecto el alumno ha podido demonstrar su capacidad para realizar un proyecto dónde se han utilizado, y ampliado, los conocimientos adquiridos a lo largo del grado para la resolución del problema planteado.

2.4 Alcance

En el presente proyecto se ha planteado una solución al problema de la localización de robots móviles a través de la simulación de una situación real. Existen numerosas posibilidades de resolver este tipo de problema con distintos métodos. Las pruebas en simuladores suelen comprobarse posteriormente, en caso necesario, en el robot real. Es decir programar sobre el robot y realizar pruebas reales para posteriormente analizar los resultados obtenidos y contrastarlos con los resultados esperados. Sin embargo en este proyecto se realizará únicamente la simulación dejando de lado la parte de implementación y comprobación del algoritmo en el robot real. El trabajo en simuladores suele ser más cómodo y económico a la hora de comprobar soluciones a problemas dados.

3 ESTADO DEL ARTE

3.1 Problemas de localización

Existen diferentes problemas de localización en la robótica móvil. Estos se pueden clasificar según su grado de dificultad para resolverlo. La diferencia entre cada uno de estos es el conocimiento de la posición desde la cual empiezan a moverse. Dependiendo de si el robot conoce o no las coordenadas del punto desde el que parte se hablará de un tipo de problema u otro. Por ello es importante diferenciar cada uno de ellos:

- Localización local
- Localización global
- Problema del tipo “Kidnapped”

Por un lado, en la localización local el robot conoce su posición inicial. Es por ello que la resolución de este tipo de problemas es más sencilla. Cualquiera que sea el algoritmo que se utilice para resolver el problema, se puede en este caso establecer unas condiciones iniciales para que desde un principio el robot sepa dónde está. Se dice que es un problema de localización local porque el robot ha de localizarse en relación a ese punto de partida conocido.

Por otro lado, en los problemas de localización global la resolución puede ser algo más complicada. En este caso el robot no sabe dónde está en el instante inicial. Esto quiere decir que puede estar en cualquier sitio del mapa. Esto complica su resolución ya que la posibilidad de que se cometa un error en la aproximación es superior a la del problema de localización local.

Finalmente, los problemas del tipo “Kidnapped” son problemas dónde, cómo su nombre lo indica (kidnapped), el robot puede ser secuestrado de su posición. Es decir que estando el robot moviéndose por un entorno, alguien puede cogerlo y llevarlo a otro sitio del mapa. Este tipo de situación es de las más complejas de resolver. No todos los algoritmos que existen para resolver los problemas de localización son capaces de resolver este caso particular.

3.2 Métodos de localización

Existen numerosos sistemas de localización utilizados actualmente para conocer la posición de un dispositivo. Entre otros, destacan el “Global Positioning System” (GPS) la localización por sensores de posición o la localización por sensores no específicos de posición. Cada uno de estos métodos tiene sus ventajas y desventajas. Además, estos se pueden clasificar según su grado de dificultad que dependerá del número de parámetros que maneje el algoritmo de localización.

En primer lugar, el sistema de localización con el menor grado de dificultad es el que usa sensores de posición. En este caso, la posición nos la dan sensores de forma directa. Un ejemplo de este tipo de sensores son los encoders. La utilización de estos últimos es muy común cuando se desea conocer la localización un robot en interiores.

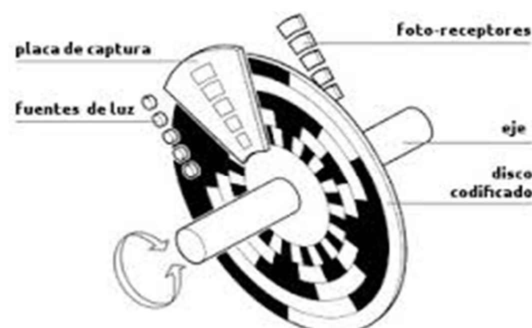


Figura 3-1 Encoder odometría

Si por el contrario deseamos localizar un dispositivo en el exterior, el método más usado es el de localización por GPS. Este sistema está basado en satélites girando alrededor de la tierra de manera continua. Está formado por un conjunto de 24 satélites que cubren toda la superficie de la tierra y que están constantemente mandando unas señales que indican, entre otros, la salud del satélite (indica si debe o no ser considerado para la toma de la posición), su posición en el espacio y su hora atómica. Son necesarios como mínimo cuatro satélites para conocer las coordenadas de la posición así como el tiempo.

Con este sistema es posible calcular las coordenadas de cualquier punto en la superficie terrestre. El GPS funciona como cualquier otro sistema de localización por balizas. Este método de ubicación puede resolver el problema en cuestión ya que permite al robot saber su posición ya sea por triangulación o por trilateración. En la triangulación el robot calcula las coordenadas de la posición basándose en los ángulos con los que el robot "ve" las balizas. Sin embargo, la trilateración consiste en la técnica a través de la cual el robot puede determinar las coordenadas de su posición basándose en la distancia que separa cada una de las balizas del robot.

Cada uno de estos dos métodos presenta ciertos errores de medida que deben de tenerse en cuenta a la hora de decidir que método usar según el grado precisión que se desee obtener.

El principal inconveniente de los sensores de posición, los encoders por ejemplo, es que al no disponer de medidas externas al robot el error es acumulativo. Esto quiere decir que a medida que el robot va avanzando de una posición a otra el error ira aumentando.

En cuanto al sistema de GPS, los errores no son acumulativos en este caso y son debidos a errores del reloj, errores en la órbita, errores en la modelización de la propagación en la troposfera y la ionosfera, errores debidos a rebotes de la señal y errores en el receptor o en la configuración del mismo. La precisión de un sistema de GPS suele rondar entorno a los 15 m generalmente aunque esta puede disminuir hasta 2-3metros según la según la calidad del dispositivo GPS utilizado.

En el caso de estudio del presente proyecto, el entorno en el cual se quiere localizar el robot puede llegar a ser una habitación de unos cuantos metros cuadrados. Es por ello que la utilización de un sistema GPS no es de gran utilidad ya que el error en comparación con las dimensiones del entorno es muy elevado.

Otra método de localización posible es usando sensores no específicos de posición (sensor laser, sensor visión, sensor ultra sonido etc). Los datos registrados por estos sensores se deben contrastar con un mapa que el robot debe disponer para poder así situarse dentro de ese mapa ficticio. El problema de este método es que se obtiene una localización local. Es decir, se podrá estimar la posición del robot solo en el caso de que se conozca la posición que ocupaba el robot en el instante inicial.

Figura 3-2 Sistema GPS

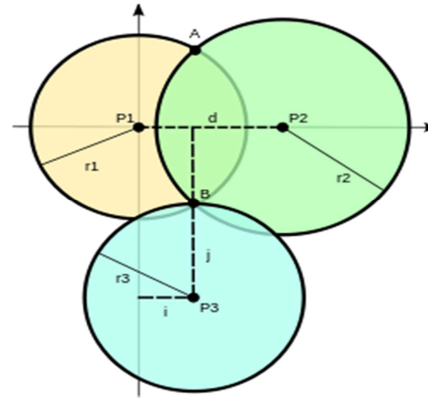


Figura 3-3 Método triangulación

3.3 Tipos de sensores

Los sensores por ultra sonidos, por luz, sensores de video o los sensores de odometría son algunos de los sensores más utilizados hoy en día en la robótica. Sería interesante pues explicar un poco el funcionamiento básico de cada uno de ellos.

3.3.1 Sensor ultra sonidos

Una de las posibles maneras de medir distancias en robótica es mediante el uso de sensores ultra sonidos. El modo de funcionamiento de estos es básicamente el de un sistema sonar cualquiera. Se utiliza un emisor que lanza pulsos de sonido ultrasónico. Estas ondas rebotaran en el caso de que exista algún objeto/obstáculo en la trayectoria de propagación. Si se da el caso de que la onda choca con algo, esta rebotará cambiando el sentido de su trayectoria. Es por ello que se necesita de un receptor que sea capaz de captar las ondas que rebotan. A través de un simple cálculo en el cual se relaciona la velocidad a la que se ha emitido las ondas, y el tiempo que tardan en ser detectadas por el receptor se puede conocer la distancia a la que se encuentra el obstáculo que ha provocado el desvió de dichas ondas. Existen robots en los cuales se tienen un emisor y un receptor que están separados el uno del otro, pero también los hay que poseen un sensor donde el emisor y el receptor son el mismo.

3.3.2 Sensor por odometría

Otro tipo de sensores son los que hemos nombrado en el apartado **Métodos de localización**, los sensores de odometría. Estos sensores nos permiten estimar la posición relativa a la localización inicial del robot. Las ventajas de este método son la sencillez de su implementación y el bajo coste que tiene. La odometría nos proporciona estimaciones bastante precisas a corto plazo. El inconveniente viene cuando se utiliza este sensor a largo plazo ya que a realizar las estimaciones a través de integraciones de estados anteriores, aparece de forma inevitable un error que va acumulándose con cada etapa de estimación. Es decir, que los errores que se producen en cada integración se van acumulando de manera que se llega a un punto en el cual la posición estimada está bastante lejos de la que realmente se encuentra el robot.

3.3.3 Sensor laser

Los sensores laser funcionan de forma similar a los sensores ultra sonidos. La diferencia es evidente, y es que en este tipo de sensores, el tipo de onda que genera el emisor es una onda lumínica en vez de sonora. La ventaja de los sensores laser es que al utilizar ondas de luz, la propagación de estas es mucho más rápida (solo hay que compara la velocidad de propagación del sonido con la velocidad de propagación de la luz). Es por esto que se pueden obtener medidas muchos más rápido y por ello la capacidad de reacción de un robot frente a obstáculos será pues mucho mejor.

3.3.4 Sensor de video

Los robots que utilizan sensores de video llevan incorporado una cámara electrónica que capta la luz y según la intensidad genera una señal con un voltaje dado. Este voltaje es generado por elementos fotosensibles que almacenan estas señales electicas en forma de patrón. Este último puede ser representado gráficamente posibilitando su visualización a modo de imagen. La calidad de las cámaras depende del número de elementos fotosensibles (pixeles) que estas poseen.

3.4 Algoritmos para la localización

Como se ha descrito anteriormente, existen numerosas aplicaciones científicas que necesitan operar según un proceso de estimación con el objetivo de conocer, aproximadamente, cuál va a ser el estado (la ubicación del robot en el problema de localización de robots) del objeto estudiado en el instante siguiente al que se encuentran.

Una de las posibles soluciones a este problema sería a través de la representación de espacios de estados. Este proceso consiste en la implementación de un modelo matemático para un sistema físico dado mediante un conjunto de entradas, variables de estado y salidas que dan como resultado una ecuación diferencial de primer orden.

En nuestro caso de estudio para poder resolver el problema se necesitara como mínimo dos modelos matemáticos. Uno de ellos se encargara de la evolución del estado en función del tiempo. En cuanto al segundo modelo, este será el responsable de relacionar las medidas realizadas con el estado.

Existen varios algoritmos que pueden ayudarnos a resolver el problema de localización procediendo de la forma en que hemos descrito anteriormente. Algunos de ellos son, el algoritmo de Markov, el algoritmo basado en un filtro de tipo gaussiano, como lo es, el algoritmo EKF (Extended Kalman Filter) y aquellos basados en filtros no paramétricos como lo son el algoritmo Grid o el AMCL (Algoritmo MonteCarlo). Todos ellos surgen de un mismo origen: el filtro de Bayes.

3.4.1 Filtro de Bayes

El algoritmo basado en el filtro de Bayes calcula la distribución de la estimación en función de las mediciones y de los datos de control. Este tipo de filtro es de carácter recursivo, es decir, la estimación en el tiempo t se calcula a partir de la estimación del estado (xt-1) a tiempo t-1. La entrada de este filtro sería, la creencia en el estado t-1 junto con los datos medidos por los sensores más recientes y el valor del estado anterior. La salida que nos proporciona este algoritmo será la estimación del estado (xt) en el tiempo t. De esta forma disponemos de una estimación para cada medida que se realiza.

Se puede dividir la estructura del algoritmo en dos partes (**Figura 3-4**): en la primera de ellas, se calcula la estimación de un estado xt en función de la estimación del estado xt-1 y los datos de control ut. Concretamente, la estimación que se asigna al estado xt se calcula integrando el producto de dos distribuciones: la que se asigna al estado anterior (xt-1) y la probabilidad de que el control ut induzca una transición desde xt-1 a xt. Se puede referir a esta primera parte del algoritmo como el paso de "predicción".

En la segunda parte del algoritmo, se multiplica la estimación del estado (xt) por la probabilidad de que la medida realizada zt coincida con la que se vería si el robot estuviese, de verdad, en la posición que cree estar. Se repite este proceso para cada supuesta posición donde el robot piensa que puede estar. Este paso recibe el nombre de "actualización de la medida". Como todo método recursivo, se necesitan unas condiciones iniciales para que el algoritmo funcione correctamente.

Algorithm Bayes_filter($bel(x_{t-1}), u_t, z_t$):
  for all x_t do
    $\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1})\, bel(x_{t-1})\, dx$
    $bel(x_t) = \eta\, p(z_t \mid x_t)\, \overline{bel}(x_t)$
  endfor
  return $bel(x_t)$

Figura 3-4 Algoritmo basado en el filtro de Bayes

Existen generalmente, dos casos posibles que han de tenerse en cuenta a la hora de definir las condiciones iniciales. Uno de ellos es cuando se conoce la posición exacta del robot (x0). En este caso se debe de inicializar la estimación del estado (x0) con una distribución dónde, se le asigna la máxima probabilidad al punto x0 y además se ha de asignar probabilidad cero al resto de puntos.

El otro caso es cuando desconocemos totalmente la posición inicial. En esta situación se deberá inicializar la estimación del estado (x0) usando una distribución uniforme.

Este algoritmo, implementado tal podemos observar en la figura 1.., solo sirve para resolver problemas de estimación muy simples.

3.4.2 Algoritmo de Markov

El algoritmo de Markov es la aplicación directa, de entre los algoritmos probabilísticos de localización, del filtro de Bayes para los problemas de localización de robots. El nombre de este algoritmo se debe al supuesto de que al filtro de Bayes se le aplica la propiedad de Markov. Esta propiedad indica que en un proceso estocástico, la distribución de probabilidad condicional sólo depende de la situación actual y no de la secuencia de los acontecimientos que la preceden.

Como se puede ver en la siguiente figura, el algoritmo es igual que el que se ha mostrado más arriba para implementar el filtro de Bayes. La única diferencia, es que en esta implementación se ha de añadirle una variable de entrada más, el mapa (m).

Algorithm Markov_localization($bel(x_{t-1}), u_t, z_t, m$**):**
for all x_t *do*
$\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1}, m)\, bel(x_{t-1})\, dx$
$bel(x_t) = \eta\, p(z_t \mid x_t, m)\, \overline{bel}(x_t)$
endfor
return $bel(x_t)$

Figura 3-5 Algoritmo de localización Markov

La **Figura 3-5** muestra cómo este algoritmo es capaz de resolver un problema de localización global. Se puede observar en el esquema (a) que en un instante inicial la posición del robot es totalmente desconocida, por ello, se inicializa todo el espacio de trabajo con la mismo probabilidad. En el esquema (b) los sensores del robot detectan una puerta, es por ello, que se le asigna un peso mayor a los espacios del mapa en los cuales hay una de ellas. En el siguiente paso se puede ver cómo el robot avanza de una puerta a otra. Al moverse las estimaciones que se habían realizado en el paso anterior van perdiendo peso progresivamente debido a la acumulación de errores. Cuando el robot

llega a la segunda puerta (esquema d) los sensores vuelven a detectarla, por ello, se vuelve a asignar un mayor peso a las zonas que tienen puerta. Como la segunda puerta ya tenía asignada mayor peso que el resto de puertas, la estimación final va a situar aproximadamente al robot dónde el realmente esta.

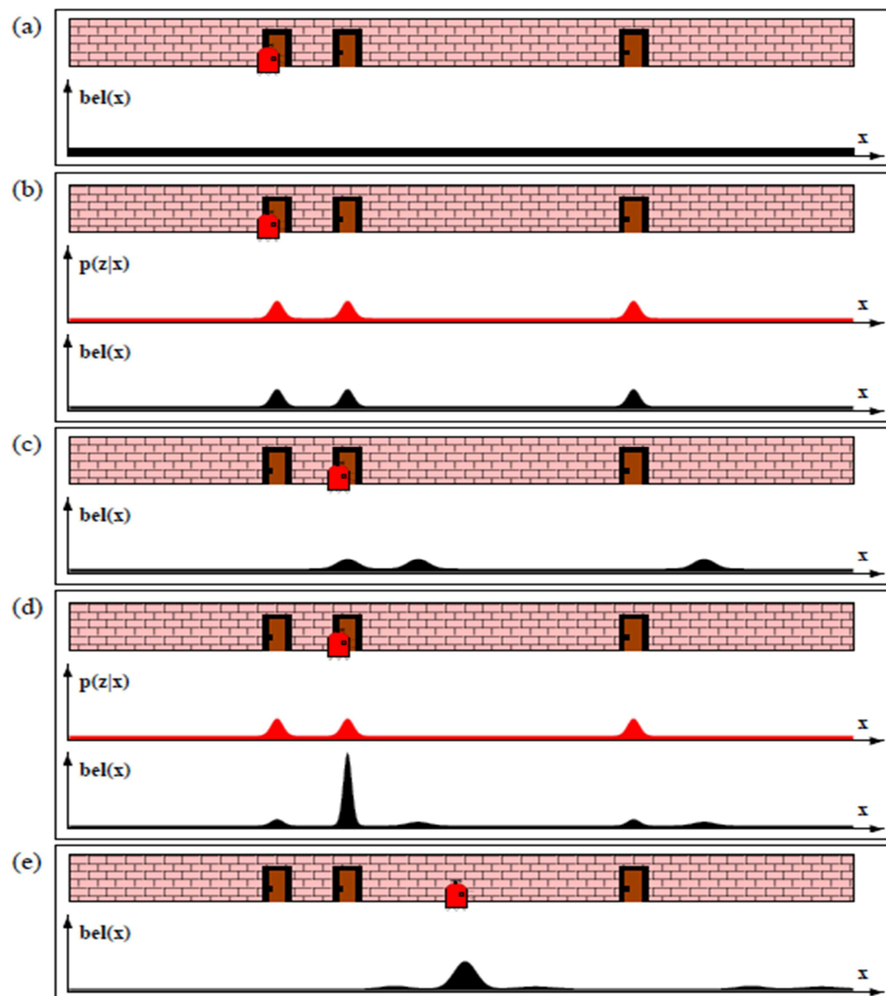


Figura 3- Evolución de la estimación de la posición con algoritmo Markov

3.4.3 Algoritmo Filtro de Kalman

El algoritmo basado en el filtro de Kalman es una las técnicas más utilizadas para implementar el filtro de Bayes. Este último fue inventado en 1950 por Rudolph Emil Kalman. Para poder hablar de filtro gaussiano se ha de cumplir las siguientes propiedades:

Se tiene que poder representar la probabilidad del estado siguiente en función de una ecuación lineal:

$$x_t = A_t x_{t-1} + B_t u_t + \varepsilon_t \; .$$

- Xt y x(t-1) son vectores de estado
- Ut es el vector control en el instante t
- At y Bt son matrices que sirven para linealizar la función
- εt representa una variable aleatoria que simula el ruido.

Además la función probabilidad de las medidas debe de ser también lineal:

$$z_t = C_t x_t + \delta_t$$

- Ct es una matriz
- Xt es el vector de estado
- δt es un valor aleatorio que se le añade a modo de ruido

Finalmente, la estimación inicial ha de ser una distribución normal de media μ0 y covarianza $\sum 0$:

$$bel(x_0) = p(x_0) = \det(2\pi\Sigma_0)^{-\frac{1}{2}} \exp\{-\tfrac{1}{2}(x_0-\mu_0)^T\Sigma_0^{-1}(x_0-\mu_0)\}$$

Algorithm Kalman_filter($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$):

$$\bar{\mu}_t = A_t\, \mu_{t-1} + B_t\, u_t$$
$$\bar{\Sigma}_t = A_t\, \Sigma_{t-1}\, A_t^T + R_t$$
$$K_t = \bar{\Sigma}_t\, C_t^T (C_t\, \bar{\Sigma}_t\, C_t^T + Q_t)^{-1}$$
$$\mu_t = \bar{\mu}_t + K_t(z_t - C_t\, \bar{\mu}_t)$$
$$\Sigma_t = (I - K_t\, C_t)\, \bar{\Sigma}_t$$

return μ_t, Σ_t

Figura 3-6 Algoritmo basado en filtro de Kalman

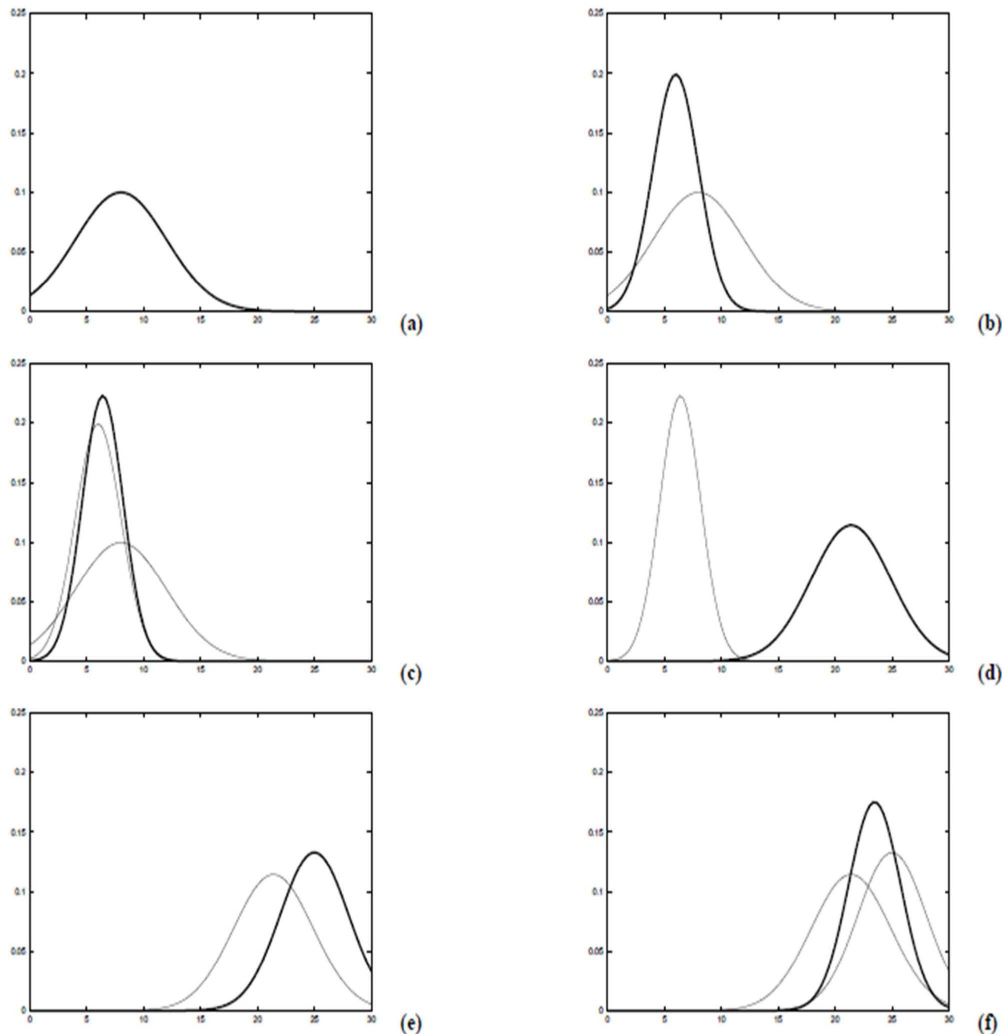


Figura 3-7 Gráficas de la respuesta del algoritmo Filtro de Kalman

En la **Figura 3-7** se puede observar el funcionamiento básico del filtro de Kalman. En primer lugar, en el gráfico (a) se puede ver la representación de la probabilidad del estado inicial según una distribución gaussiana uni-modal. En este caso se conoce una estimación de la posición donde se encuentra el robot. Esto quiere decir que se trata de un problema de localización de tipo local. En el gráfico (b) observa en este caso otra distribución que corresponde con la medida realizada por el robot. Al aplicar el filtro de Kalman se obtiene una nueva distribución que estima el estado del robot según, de nuevo, otra distribución gaussiana grafico (c). A continuación, el robot se mueve y cambia de posición generando esta acción dos nuevas distribuciones: una correspondiente al estado anterior, y otra correspondiente a la medida obtenida por el sensor del robot. Aplicando de nuevo el filtro de Kalman, se obtiene la distribución resultante que estima la nueva posición del robot.

3.4.4 Algoritmo Filtro de Kalman Extendido

La idea principal del filtro de Kalman Extendido (EKF) es la misma que la del filtro de Kalman con la diferencia que en este tipo de filtro las ecuaciones que relación un estado con otro no son lineales. Esta adaptación nace de la imposibilidad de aplicar el filtro de Kalman en problemas donde el robot describe movimientos que no son lineales. Un ejemplo de esto es un robot siguiendo una trayectoria circular. En este caso el filtro de Kalman no serviría para estimar la localización del robot.

La diferencia está pues en las ecuaciones que relacionan los estados, así como en las ecuaciones que estima la probabilidad de las medidas.

$$\begin{aligned} x_t &= g(u_t, x_{t-1}) + \varepsilon_t \\ z_t &= h(x_t) + \delta_t \ . \end{aligned}$$

Las funciones g y h sustituyen a las matrices At, Bt y Ct del filtro de Kalman. Son funciones no lineales. Para poder utilizar este filtro se ha de realizar una aproximación de la funcion no lineal a través de una que sí que lo sea. Es por ello que el filtro de kalman extendido puede dar más errores que el Filtro de Kalman. Esta aproximación se realiza a través de la utilización de expansiones en serie de Taylor. Con esto se conseguirá linealizar el problema no lineal. Existen otros métodos que también se puede usar para aproximar funciones no lineales a otras que sí que lo son.

La eficacia de este método dependerá, al ser una aproximación, del grado de complejidad del problema. Así, para ecuaciones con un grado alto de no linealidad, se obtendrán resultados bastante pobres. Sin embargo, si las funciones son aproximadamente lineales, en este caso sí que es obtendrá una buena aproximación.

Volviendo una vez más al problema del robot presentado en el apartado anterior, en la **Figura 3-8** se pueden observar algunos aspectos que difieren respecto al ya comentado anteriormente. La principal diferencia es que en este problema, en el instante inicial, no existe una distribución uniforme a lo largo de todo el espacio. Esto se debe a que el robot conoce aproximadamente su posición inicial. Se trata pues de un problema de localización local en el cual, el robot irá estimando su ubicación en el mapa a partir del conocimiento de las coordenadas iniciales. En la primera figura el robot está situado frente a una puerta están toda la distribución normal de probabilidades centrada en ese punto. Después el robot avanza hacia la derecha lo que provoca que la varianza de la distribución gaussiana aumente debido al aumento de error. Una vez llega a la segunda puerta, gracias a la lectura de los sensores, se le da una probabilidad alta a esa posición obteniendo como resultado una buena estimación. Al avanzar de nuevo hacia la derecha, la probabilidad va perdiendo peso de nuevo a medida que va aumentando el error.

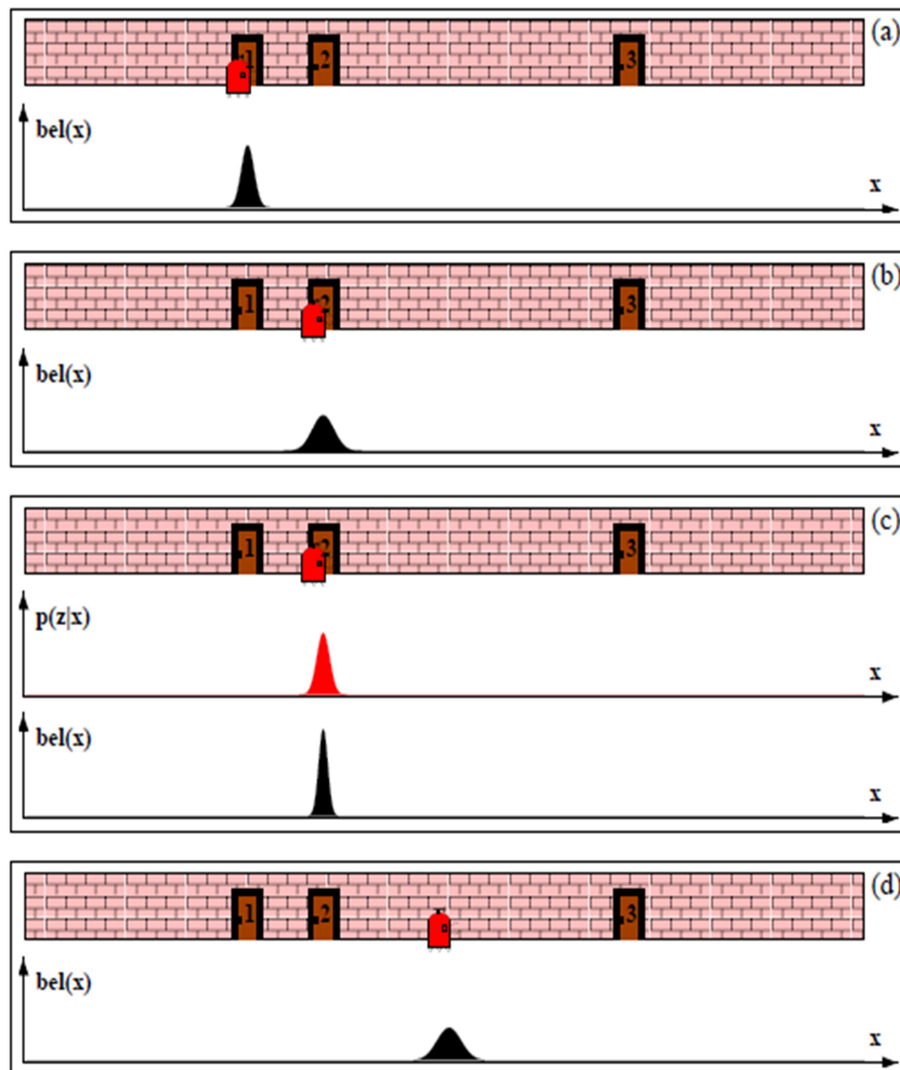


Figura 3-8 Evolución de la estimación de la posición con algoritmo EKF

El algoritmo EKF es muy utilizado para resolver problemas de localización local donde la incertidumbre de la posición inicial es baja. Sin embargo, tanto para los problemas de localización global como los problemas donde el robot puede ser "secuestrado" (kidnapped) de su posición, no se suele utilizar este tipo de algoritmo.

3.4.5 Algoritmo Grid

El algoritmo Grid utiliza un filtro de tipo no paramétrico conocido como "Histogram Filter". Lo que hace este algoritmo es dividir el espacio de estados posibles del robot en celdas. Según las medidas realizas, el desplazamiento del robot, y el mapa, el algoritmo pondera cada una de estas celdas con una probabilidad. Tanto el número de celdas como su tamaño es invariante una vez se ha iniciado el proceso de localización.

Sin embargo, estas variables pueden inicializarse con diferentes valores influyendo esto en la calidad de la aproximación de la siguiente manera:

- A mayor número de celdas y menor tamaño de cada una de ellas, mejor será la estimación pero los tiempos de computación aumentaran pudiendo incluso inhabilitar el funcionamiento del filtro.
- A menor número de celdas y mayor tamaño de cada una de ellas, se obtendrá una aproximación más pobre pero los tiempos de computación serán rápidos.

En base a esto, se ha de buscar una relación entre el número de celdas y su tamaño de tal forma que se obtenga la mejor aproximación con un tiempo de computación admisible. En la Figura 3-9 se puede observar el funcionamiento del algoritmo en el mismo caso de estudio que se ha utilizado para el resto de algoritmos.

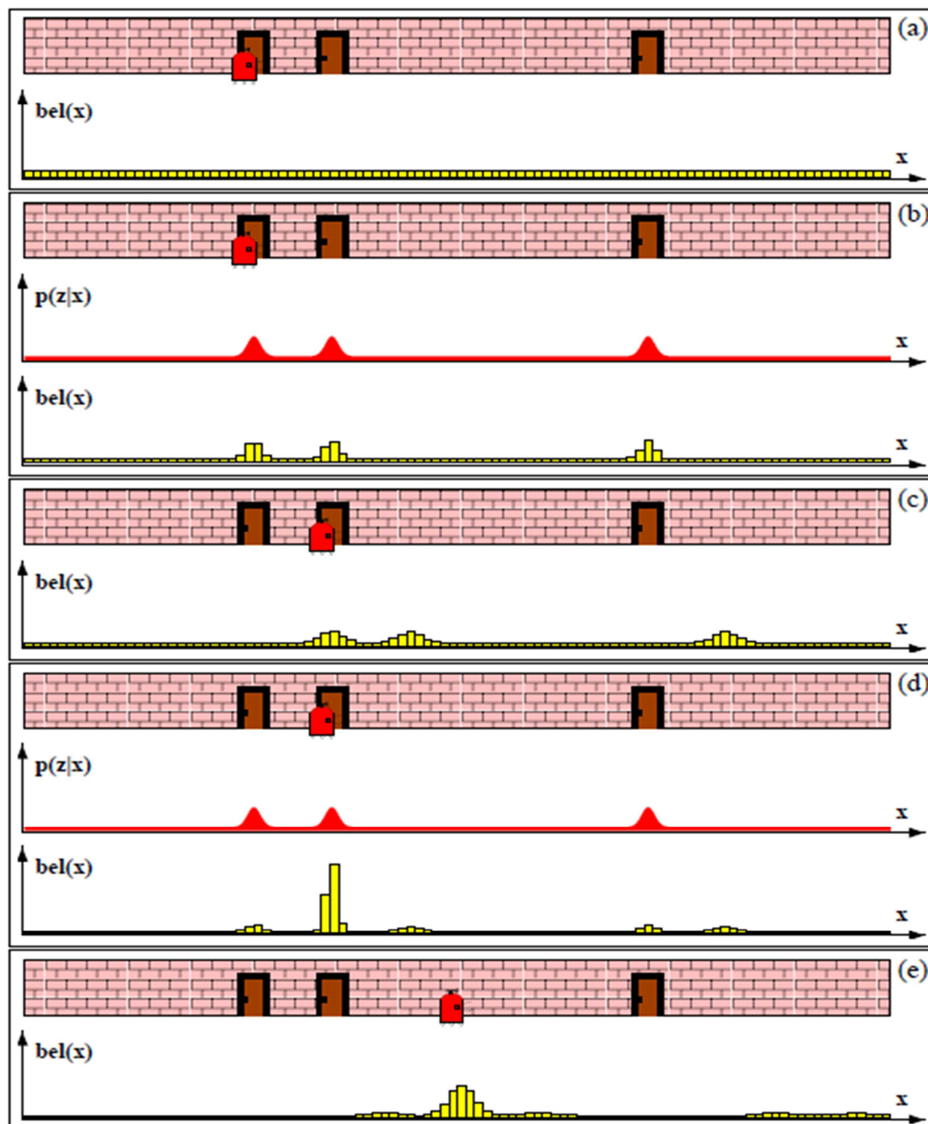


Figura 3-9 Evolución de la estimación de la posición con algoritmo Grid

Para este algoritmo, el problema del robot con las tres puertas se resuelve de forma similar. La diferencia es principalmente la forma en la que el robot realiza la estimación de su posición. Aquí, como ya se ha explicado en el algoritmo de Grid, se genera un número finito de celdas que tienen asignadas una probabilidad. En el instante inicial, debido al desconocimiento de la posición del robot, se divide todo el espacio en pequeñas celdas con la misma probabilidad. A medida que el robot va avanzando y detectando las puertas, se le va asignando a las celdas de la zona de las puertas mayor peso que al resto provocando que tras varias iteraciones el robot consiga posicionarse en el entorno.

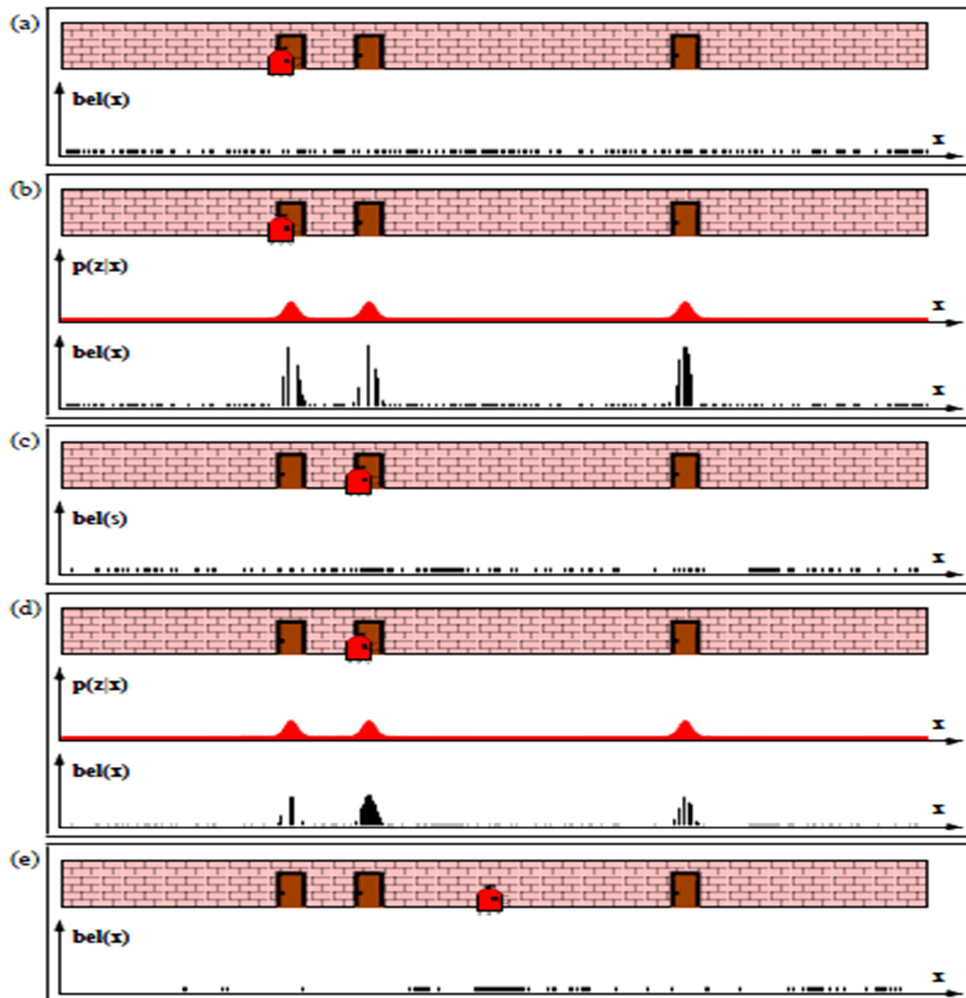
3.4.6 Algoritmo Montecarlo

Este algoritmo es uno de los más utilizados hoy en día para resolver los problemas de localización en robótica móvil. Es un filtro de tipo no paramétrico ya que no presenta ecuaciones que relacionen los estados a lo largo del tiempo. La estimación dependerá de las medidas de que se realicen en ese instante. El método Montecarlo utiliza un filtro de partículas para obtener la estimación de estados.

El modo de funcionamiento de este tipo de filtro es: generar una muestra de partículas que se reparten por todo el espacio de estados posibles. Al igual que en el algoritmo “Grid”, estas partículas se han de ponderar utilizando unas ecuaciones que asignan a cada una de ellas una probabilidad en función de la lectura de los sensores, del mapa y del movimiento del robot. De esta forma lo que se busca obtener es una convergencia hacia las partículas que tienen más probabilidad.

Hay que tener en cuenta varios aspectos:

- A mayor número de partículas, mejor será la estimación de la posición pero al igual que en el algoritmo Grid, cuando se utilizan celdas muy pequeñas, el tiempo de computación será muy elevado.
- Por otro lado, si el número de partículas es bajo es muy probable que la estimación que se realice sea erróneo debido a la rápida convergencia.


(a)
bel(x)
x
(b)
p(z|x)
x
bel(x)
x
(c)
bel(s)
x
(d)
p(z|x)
x
bel(x)
x
(e)
bel(x)
x

4 SOLUCIONES ALTERNATIVAS

Existen muchas formas de resolver los problemas de localización en la robótica móvil. Es posible que unos métodos sean mejores que otros, pero siempre se puede realizar una buena combinación entre los tipos de sensores y los algoritmos utilizados para tratar de adaptar la solución de manera óptima al problema en cuestión.
Con esto se quiere decir, que hay problemas para los cuales, por ejemplo, se necesita una alta precisión a la hora localizar el robot y otros en los cuales no se necesita tanta. Así mismo, puede también interesar que los tiempos de computación no sean muy altos.

Por ejemplo, para un problema en el que se necesita localizar a un robot conociendo aproximadamente la posición inicial y donde se necesita que los tiempos de computación sean bajos, se podría implementar el algoritmo EKF que cumpliría con estos requisitos. Además, en cuanto a la elección del aparato de medidas se podría utilizar un sensor de ultrasonidos el cual proporcionaría los datos necesarios al algoritmo.

5 SOLUCION PROPUESTA

5.1 METODO MONTECARLO

Tras el análisis de los distintos algoritmos que nos permiten auto localizar un robot, se obtara por el algoritmo de localización Montecarlo ya que tiene una sencilla implementación y permite resolver cualquier problema de localización únicamente variando la cantidad de partículas que se genere. Es un algoritmo que está empezando a ser muy utilizado por numerosas empresas para dotar a sus robots de un sistema de localización.

Como se ha descrito en el apartado de algoritmos, MCL es un algoritmo que calcula la estimación de un estado a través de la representación de partículas. Sirve para los problemas tanto de localización global como local. Su implementación es la siguiente:

Algorithm MCL($\mathcal{X}_{t-1}, u_t, z_t, m$):
  $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
  for $m = 1$ to M do
    $x_t^{[m]} = \textbf{sample_motion_model}(u_t, x_{t-1}^{[m]})$
    $w_t^{[m]} = \textbf{measurement_model}(z_t, x_t^{[m]}, m)$
    $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
  endfor
  for $m = 1$ to M do
    draw i with probability $\propto w_t^{[i]}$
    add $x_t^{[i]}$ to $\mathcal{X}_t$
  endfor
  return $\mathcal{X}_t$

Figura 5-1 Algoritmo AMCL

Las variables de entrada del algoritmo serán el vector que contiene la posición de las partículas (Xt), las medidas realizadas a través de la odometría del robot (ut), la medida realizada por el sensor (laser,sonar,video) y finalmente un vector con la lista de objetos que están presentes en el entorno de trabajo, es decir, un mapa (m).

Se puede dividir la estructura del algoritmo en cuatro partes principales:

- Muestreo aleatorio de partículas
- Calculo del desplazamiento hipotético por odometría
- Ponderación de las muestras
- Re-muestreo de las partículas o actualización según el peso de cada partícula.

5.1.1 Muestreo aleatorio de partículas

En esta parte del algoritmo lo que se busca es generar aleatoriamente una muestra de partículas por todo el entorno de acción del robot. Así en un primer instante existirán M partículas repartidas por todo el mapa. Cada una de estas partículas será un posible estado del robot (posible ubicación). El número de partículas M que se genera inicialmente dependerá del nivel de precisión que se busca. Cuantas más partículas generemos tendremos una mayor resolución a la hora de analizar los distintos lugares del mapa.

Para generar esta muestra de partículas aleatoriamente, se puede utilizar cualquiera de estas dos funciones:

Algorithm sample_normal_distribution(b):

$$\textit{return } \frac{b}{6} \sum_{i=1}^{12} \mathbf{rand}(-1,1)$$

Algorithm sample_triangular_distribution(b):

$$\textit{return } b \cdot \mathbf{rand}(-1,1) \cdot \mathbf{rand}(-1,1)$$

Figura 5-2 Algoritmos para generación de una distribución aleatoria

La diferencia entre ambas es que una genera números aleatorios según una distribución normal y la otra según una distribución triangular. A la hora de generar las M partículas aleatoriamente, también se ha de generar un vector orientación aleatorio para cada una de las muestras.

5.1.2 Calculo del desplazamiento aproximado por odometría

Aquí lo que se busca es obtener una estimación de cuanto se ha movido el robot. Utilizando las medidas realizas por los encoders de las ruedas, se puede obtener aproximadamente la distancia que ha avanzado el robot así como el ángulo. De esta forma se obtendrá las nuevas coordenadas de un punto suponiendo que se ha movido una distancia D y un ángulo (Θ) desde el punto anterior. Lo que se hará será aplicar este movimiento a cada una de las partículas de tal manera que cada partícula se desplazara hacia una nueva posición basándose en el supuesto desplazamiento del robot. La forma en que los encoders estiman el desplazamiento y el ángulo es a través de un simple cálculo de trigonometría.

En primer lugar se han de medir el número de vueltas que gira cada rueda. Una vez se conoce este dato multiplicándolo por 360, que es el ángulo de una vuelta (o 2Π si se

representa en radianes), y por el radio de la rueda, se puede obtener la distancia que se ha recorrido con dicha rueda. Esta operación ha de realizarse para cada una de las ruedas. En el caso de un robot de dos ruedas, para cada rueda se obtendrá Δs (rueda). Una vez se tiene el valor de cada Δs, realizando la siguiente operación se calculara la distancia que ha avanzado el robot.

$$\Delta s = \frac{\Delta s(\text{rueda derecha}) + \Delta s(\text{rueda izquierda})}{2}$$

Para calcular el ángulo de avance se utilizara la siguiente formula:

$$\Delta \Theta = \frac{\Delta s(\text{rueda derecha}) - \Delta s(\text{rueda izquierda})}{b}$$

Donde b es la distancia que hay entre las dos ruedas.

Una vez obtenidos los valores Δs y ΔΘ solo resta realizar un simple cálculo para calcular la distancia que se ha avanzado en el eje x (Δx) y la que se ha avanzado en el eje y (Δy).

$$\Delta x = \Delta s.\cos(\Theta + \Delta\Theta)$$

$$\Delta y = \Delta s.\sin(\Theta + \Delta\Theta)$$

Para obtener las nuevas coordenadas globales de cada punto se le sumara a cada coordenada x e y su respectivo incremento Δx y Δy obteniendo de esta forma unas nuevas coordenadas para cada punto.

Una vez realizados estos cálculos se pasara a implementar el siguiente algoritmo con el objetivo de aplicar el movimiento realizado por el robot, y registrado por los encoders, a cada una de las partículas que se ha generado aleatoriamente.

1: **Algorithm sample_motion_model_odometry**(u_t, x_{t-1}):

2: $\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$

3: $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$

4: $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

5: $\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}} + \alpha_2 \delta_{\text{trans}})$

6: $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3\ \delta_{\text{trans}} + \alpha_4(\delta_{\text{rot1}} + \delta_{\text{rot2}}))$

7: $\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}} + \alpha_2 \delta_{\text{trans}})$

8: $x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot1}})$

9: $y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot1}})$

10: $\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

11: *return* $x_t = (x', y', \theta')^T$

Figura 5-3 Algoritmo sample_model_odometry

Antes de comentar los distintos cálculos que se realizan en el algoritmo se ha de tener en cuenta que el desplazamiento que realiza el robot se puede aproximar por una rotación $\delta rot1$, una translación $\delta trans$ y otra rotación $\delta rot2$. Se puede ver más precisamente a qué corresponde cada una de estas transformaciones en la siguiente figura.

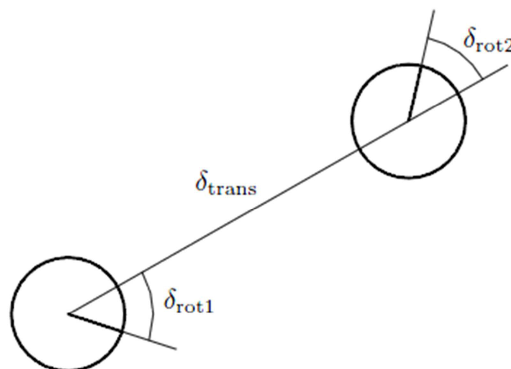


Figura 5-4 Transformaciones movimiento robot

Dicho esto, y volviendo de nuevo al algoritmo de la Figura 5-3, en las tres primeras líneas se calcula el valor de estos tres parámetros a través de las medidas realizadas por los encoders. Estos valores se introducen en el algoritmo como variable de entrada ut:

$$u_t = \begin{pmatrix} \bar{x}_{t-1} \\ \bar{x}_t \end{pmatrix}$$

Y a su vez cada una de estas dos variables realmente representa:

$$\bar{x}_{t-1} = (\bar{x} \ \ \bar{y} \ \ \bar{\theta})$$

$$\bar{x}_t = (\bar{x}' \ \ \bar{y}' \ \ \bar{\theta}')$$

Estos valores son actualizados a cada instante por la odometría del robot. La barra indica que se trata de medidas referenciadas a un sistema de coordenadas interno al robot cuya relación con el sistema de coordenadas general es desconocido. La otra variable de entrada del algoritmo es Xt-1. Este dato representa las coordenadas de la última posición calculada. En este caso, se tratará de las coordenadas (en t-1) de cada una de las partículas generadas en el paso “muestreo aleatorio de partículas”

Prosiguiendo con el análisis del algoritmo, en las líneas 5,6 y 7 se calculan unos nuevos valores dependientes de δrot1, δtrans y δrot2, y de una función “muestreo” que se le resta a estos últimos valores con el objetivo de obtener mayor robustez en el algoritmo. Al añadir la función “muestreo” que no es más que una función que genera un número aleatorio (se implementa con cualquiera de los dos algoritmos visto en el apartado 5.1.1) se está añadiendo una perturbación que simula al ruido que aparece en las mediciones.

Finalmente, a las coordenadas introducidas en el algoritmo a través de Xt-1 se les sumará el supuesto desplazamiento con ruido calculado en las líneas 5, 6,7. Se puede observar que existen unos parámetros constantes (α1, α2, α3, α4) en la función de muestro. Estos valores lo que hacen es modificar la forma que tendrá las distribución de las nuevas posibles posiciones. En la siguiente figura se puede observar como varía dicha distribución para valores distintos de α1, α2, α3, α4.

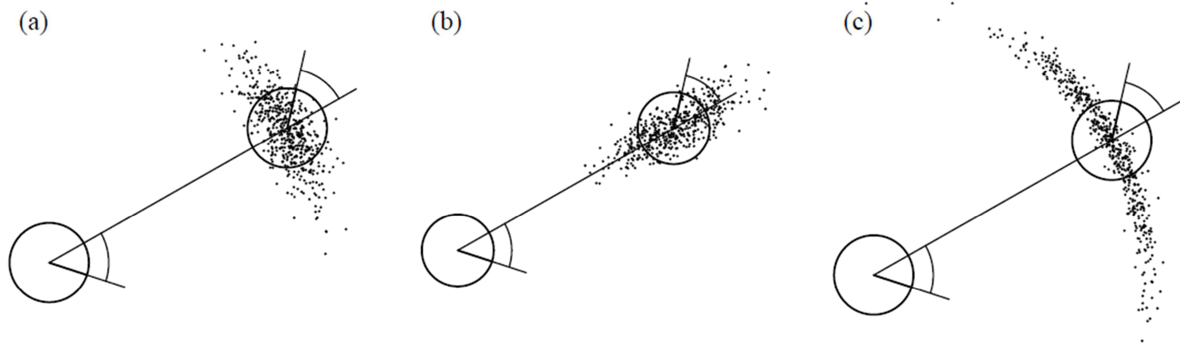


Figura 5-5 Distribución de las posiciones tras movimiento

Las figuras (a) y (c) se obtienen variando los valores de α1 y α2 ya que estos afectan al ruido que se le añade a la primera rotación δrot1 y a la segunda rotación δrot2 generando ese tipo de distribución. Sin embargo cuando variamos los valores de α3 y α4 (figura c), las partículas se distribuyen a lo largo de la trayectoria de la partícula ya que estos dos parámetros influyen en el valor de δtrans.

5.1.3 Ponderación de las muestras

Una vez se han generado las partículas (muestras) y se les ha aplicado el respectivo desplazamiento aproximado que ha realizado el robot, ahora lo que resta es utilizar un algoritmo para poder ir descartando partículas y obligando al sistema a que el conjunto de todas las partículas vaya convergiendo a cada iteración hacia la posición real del robot.

Para ello deberemos utilizar las medidas realizadas por el sensor del robot. Esta implementación tiene en cuenta que el sensor utilizado es de tipo laser. El objetivo del algoritmo siguiente algoritmo es el de comparar las medidas que realiza el robot con las medidas que se obtendrían si se tomaran medidas situándose sobre cada una de las partículas y teniendo en cuenta la orientación de estas. La implementación sería la siguiente:

Algorithm beam_range_finder_model(z_t, x_t, m):

$q = 1$

for $k = 1$ to K do

  compute z_t^{k*} for the measurement z_t^k using ray casting

  $p = z_{\text{hit}} \cdot p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}} \cdot p_{\text{short}}(z_t^k \mid x_t, m)$

    $+ z_{\max} \cdot p_{\max}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$

  $q = q \cdot p$

return q

Figura 5-6 Algoritmo beam_range_finder

Las variables de entrada que se le tienen que pasar a este algoritmo son:

- zt: vector que contiene las medidas realizadas por el sensor utilizado por el robot.
- xt: es el vector posición
- m: este vector contiene la información del mapa

Existe una variable interna que representa la medida "real" que ha realizado el sensor del robot denotada por zt (k)*. Será esta variable la que se utilizara posteriormente para compararla con la mediad que se obtendría tomando medidas en la posición y orientación de cada partícula.

Esta función devolverá una probabilidad q que dependerá de la coincidencia de las medidas del punto estudiado y de las medidas reales del robot. Este algoritmo se ha de aplicar, en el caso de estudio que se está estudiando, para cada una de las M partículas generadas. De esta forma se inicializa q=1 y se analiza cada medida realizada por cada uno de los rayos laser comparándola con "lo que mediría cada uno de esos rayos" si se situara el sensor laser sobre la posición de cada una de las partículas. La ecuación que calculará la probabilidad para cada una de estas medidas está compuesto por unas sub-funciones que tienen en cuenta cierto aspectos importantes que se van a explicar a continuación.

Este modelo incluye cuatro tipos de errores de medida que son todos ellos necesarios para que el modelo funcione correctamente.

Estos errores son los siguientes:

- Errores debidos al ruido que aparece en las medidas
- Errores debido a objetos inesperados (no aparecen en el mapa)
- Errores debido a los fallos de detección de objetos
- Errores debidos a la aparición de ruido aleatorio inexplicable.

4.1.3.1 Error debido al ruido presente en las medidas

Es necesario corregir el error debido al ruido presente en las medidas ya que el láser no siempre va a realizar una buena medida. Es decir, no siempre va devolver el valor exacto de la distancia a la que se encuentra el objeto/obstáculo contra el que ha chocado la onda lumínica. De hecho, aunque este sensor hubiera hecho una buena medida, esta misma estaría perturbada por el ruido que aparece en la señal medida cuya intensidad dependerá de la resolución del sensor utilizado.

Este ruido se puede modelar por una distribución de Gauss estrecha con media *zt* y desviación *σhit* como la mostrada en la siguiente figura.

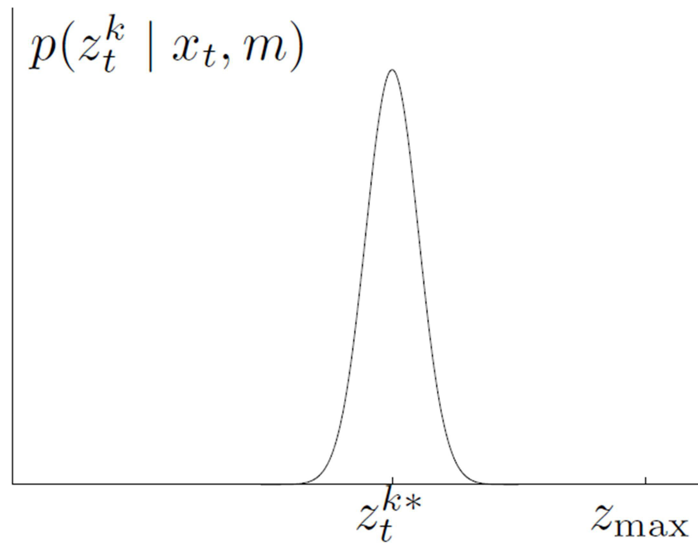


Figura 5-7 Distribución phit

zmax denota el valor máximo del rango que el sensor es capaz de medir.

Esta distribución se obtiene implementando la siguiente función:

$$phit(zt^k|xt,m) = \left\{ \frac{1}{\sqrt{2\Pi\sigma^2 \text{hit}}} e^{-\frac{1(zt^k - zt^{k*})^2}{2\sigma^2\text{hit}}} \quad si \quad 0 \le zt^k \le zmax \right\}$$

$$phit(zt^k|xt,m) = \{ \quad 0 \quad en\ cualquier\ otro\ caso \}$$

Los términos $\sigma^2\text{hit}$ y $zmax$ son valores constantes que se han de inicializar.

4.1.3.2 Errores debido a objetos inesperados

Los errores debido a objetos inesperados es otra de las posibles fuentes de errores que necesita corregirse. Cuando se habla de objetos inesperados, se está haciendo referencia a cualquier cosa que, pudiendo moverse libremente por el entorno del robot, pueda interponerse entre los objetos registrados en el mapa y el sensor del robot. Esto puede pasar ya que el robot posee un mapa que es estático, es decir, ese mapa no se genera "online", sino que se crea en un momento dado registrando pues los objetos que se encontraban en el instante en que el robot ha "dibujado" el mapa.

Por el contrario, las personas o cualquier objeto móvil que se encuentra en el entorno de acción del robot, no están registrados en el mapa, por lo que para el robot son totalmente desconocidos.

Esta probabilidad se puede representar matemáticamente cómo una distribución exponencial

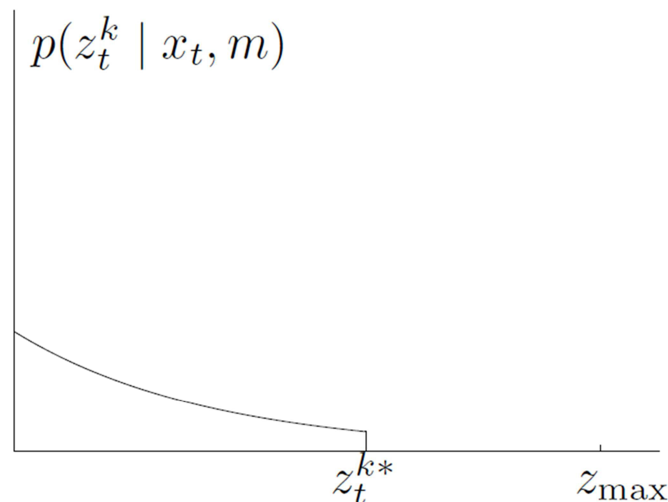


Figura 5-8 Distribución pshort

Cuya función que equivale a esta representación gráfica es:

$$pshort(zt^k|xt,m) = \left\{ \frac{1}{1-e^{-\lambda short.zt^{k*}}}.\lambda short.e^{-\lambda short.zt^k} \quad si \quad 0 \leq zt^k \leq zt^{k*} \right\}$$

$$pshort(zt^k|xt,m) = \{ \quad 0 \quad en\ cualquier\ otro\ caso \}$$

El término $\lambda short$ es un valor constante que se ha de inicializar.

4.1.3.3 Errores debido a los fallos de detección de objetos

Los errores debido a los fallos detección dependen del tipo de sensor utilizado. Un ejemplo es, cuando los sensores ultrasonidos emiten la onda y esta rebota de tal forma que no vuelve, en este caso, el receptor es incapaz de detectarla por lo que la medida que dará el sensor será errónea. Por otro lado, en los sensores laser, si el objeto contra el que rebota el rayo láser tiene un color muy oscuro, es posible que este absorba toda la onda lumínica de tal forma que el receptor, al igual que en el receptor del sensor ultrasonidos, no detecte la onda de retorno.

Este caso se puede modelar matemáticamente según una distribución puntual centrada en zmax.

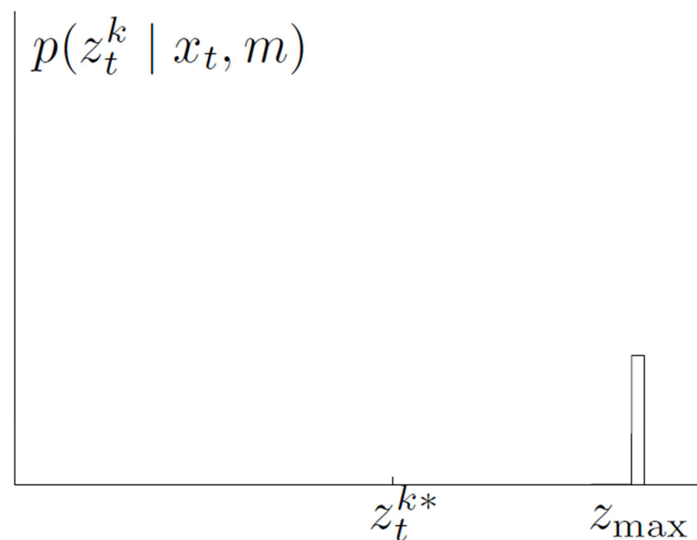


Figura 5-9 Distribución pmax

La implementación de esta distribución es bastante simple:

$$pmax(zt^k|xt,m) = \begin{cases} 1 & si \quad z = zmax \\ 0 & en\ cualquier\ otro\ caso \end{cases}$$

4.1.3.4 Errores debidos a la aparición de ruido aleatorio inexplicable.

A veces, los sensores de medida muestran valores que no corresponden a lo que realmente debería haberse medido. Ya sea por algún problema de interferencia o por el rebote de las ondas, existen medidas que son erróneas por lo que habrá que tenerlas en cuenta. Un forma de corregir este error es generando una distribución uniforme a lo largo de todo el rango de medida del sensor obteniendo la gráfica de la figura siguiente.

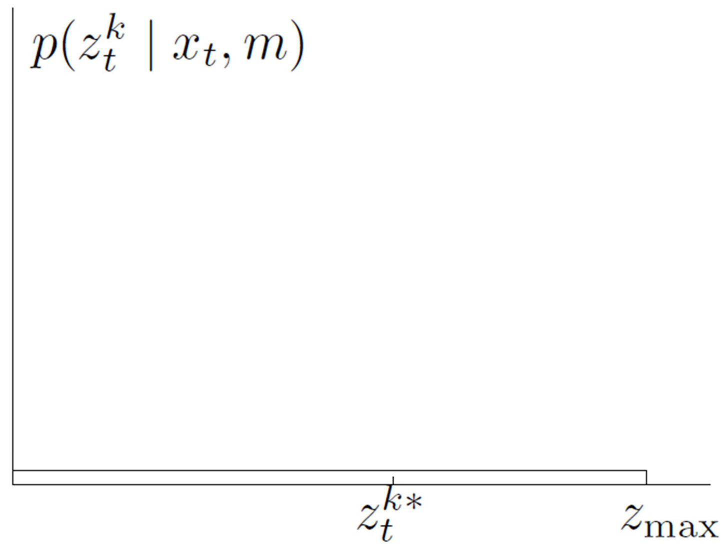


Figura 5-10 Distribución prand

Esta función se implementara de la siguiente manera:

$$prand(zt^k|xt,m) = \begin{cases} \dfrac{1}{zmax} & si \quad 0 \leq zt^k < zmax \\ 0 & en\ cualquier\ otro\ caso \end{cases}$$

Una vez implementadas las cuatro funciones $phit, pshort, pmax\ y\ prand$, ya podemos calcular la probabilidad (o peso) que tiene cada una de las partículas.

Para ello se ha de aplicar la ecuación:

q= $zhit * phit(zt^k|xt,m)$+zshort* $pshort(zt^k|xt,m)$+zmax* $pmax(zt^k|xt,m) +$ $zrand * \ prand(zt^k|xt,m)$

Teniendo en cuenta que zhit, zshort, zmax y zrand son parámetros que se han de inicializar y que han de cumplir la siguiente relación:

$$zhit + zshort + zmax + zrand = 1$$

La distribución resultante de la suma de las distintas distribuciones individuales da como resultado una nueva función que tendrá la forma de la siguiente figura:

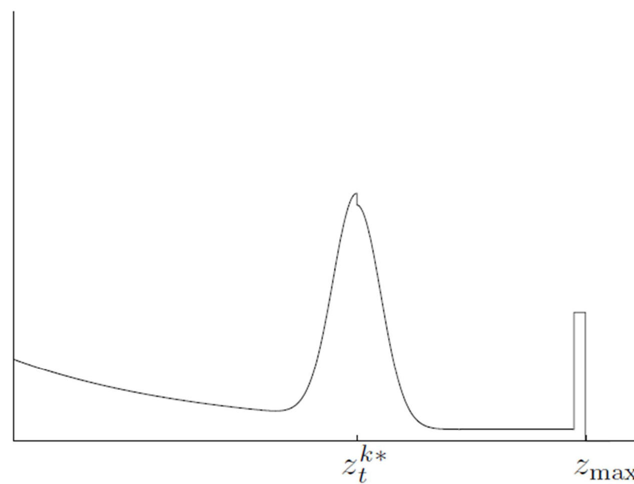


Figura 5-11 Distribución resultante de phit, pmax, prand, pshort

Una vez implementado este algoritmo ya se dispondrá de un vector con la ponderación de cada una de las partículas.

5.1.4 Re muestro de las partículas o actualización según el peso de cada partícula

El objetivo de esta parte del algoritmo es el de generar de nuevo M partículas aleatoriamente. La diferencia que existe entre este muestreo y el primer muestreo que se realizó en el apartado **Muestreo aleatorio de partículas**, es que en el primer paso todos los puntos del espacio de estudio tienen la misma probabilidad de ser elegidos por la función de muestreo aleatorio, sin embargo, en el "re muestreo" las partículas están ponderadas, es decir, unas partículas van a tener más probabilidad que otras de salir en el "sorteo ". De esta forma lo que se busca es que se generen partículas en las zonas próximas a las partículas que tiene más probabilidad. Obligando al sistema, paso a paso, a converger a hacia la posición real del robot

Existen varias formas de implementar el re-muestro de partículas. A continuación se explicara matemáticamente como se puede implementar esta función. Más adelante, cuando se hable del código de implementación del simulador se explicara concretamente la forma en la que se ha procedido para llegar al resultado buscado.

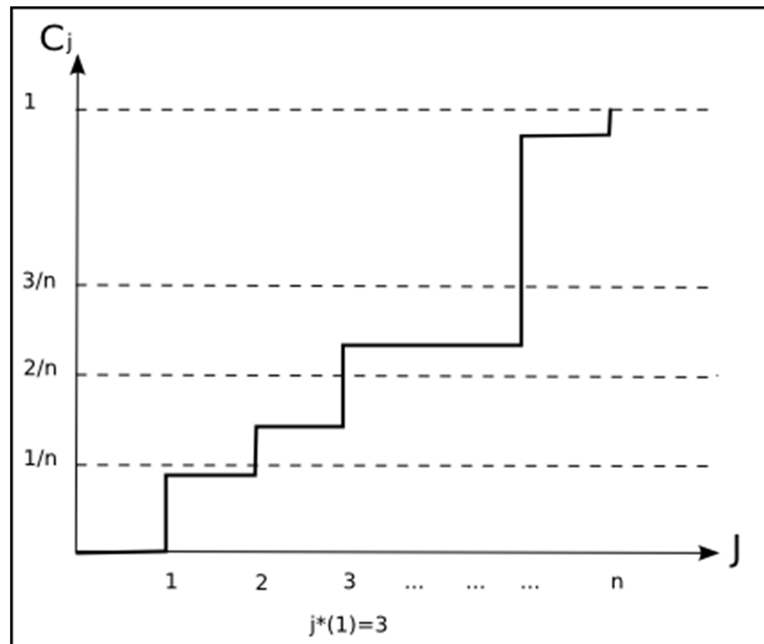


Figura 5-12 Función probabilidad acumulada

En la **Figura 5-12** se ha representado un ejemplo de una función que muestra la probabilidad acumulada para un numero N de partículas. En el eje de las abscisas se representa discretamente el número de partículas totales que se han generado. En el eje de las ordenadas se representa la probabilidad acumulada que tiene a cada una de las partículas. Para re muestrear las n partículas lo único que se ha de hacer es generar n números aleatorios entre 0 y 1 de tal forma que el número que se obtiene a cada iteración corresponderá directamente con una de las n partículas. Si se da el caso de que sale la partícula x, se cogerán las coordenadas de esa partícula x y se copiaran en un nuevo vector de tal forma que se obtendrá un vector con las coordenadas de las nuevas partículas. Si la función está bien implementada es lógico que se repitan más veces las partículas que tengan mayor probabilidad (peso).

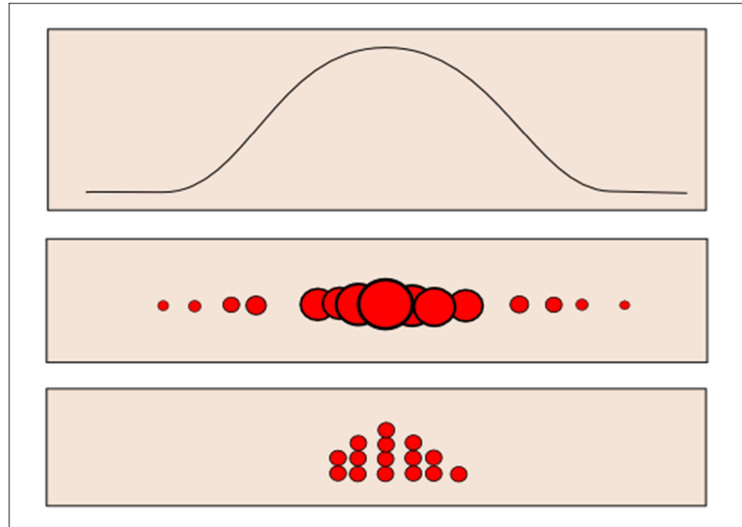


Figura 5-13 Re-muestreo de partículas

En la Figura 5-13 se puede observar como las partículas se van concentrando en los lugares ocupados por las partículas con mayor probabilidad.

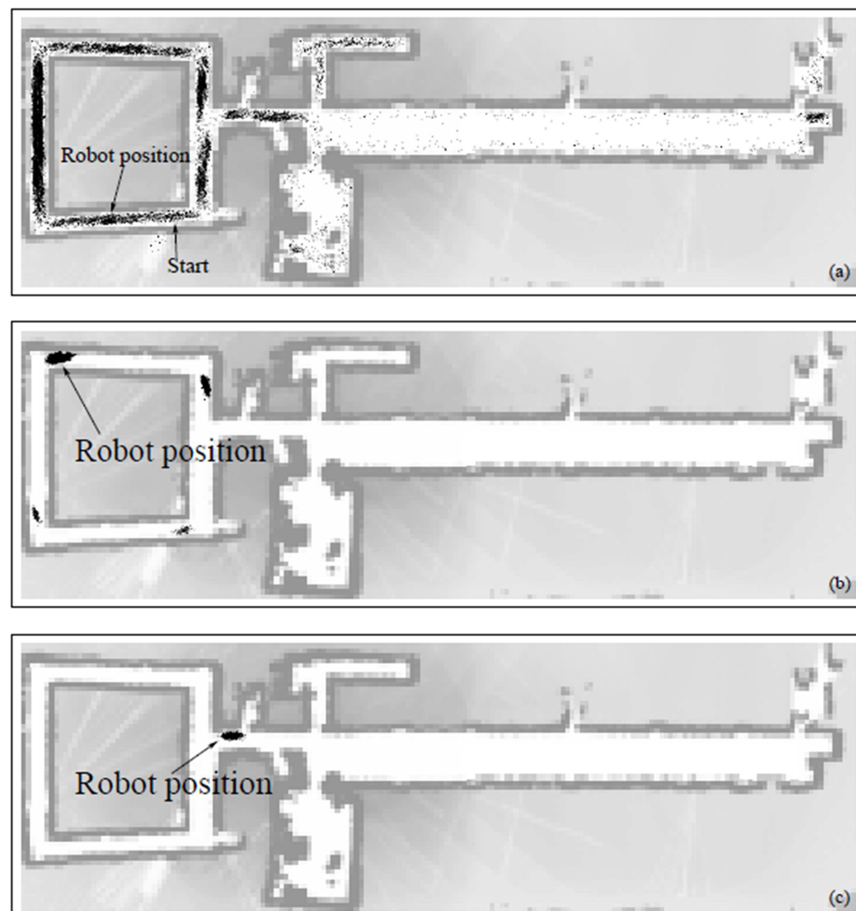


Figura 5-14 Evolución de las partículas en un problema de localización

En la Figura 5-14 se puede comprobar un ejemplo de re-muestreo de partículas. Arriba de la figura se puede observar el instante poco después de que el robot se ponga en marcha. Las partículas están repartidas por casi todo el entorno cercano a la posición del robot. A medida que el robot va avanzando, se puede ver como en número de partículas se va concentrado poco a poco en lugares concretos donde el robot piensa que puede estar. Esta agrupación en las esquinas se debe a la similitud de las cuatro posiciones del cuadrado del mapa. Una vez el robot sale del cuadrado, el algoritmo sitúa todas las partículas justo dónde está el robot, estimando correctamente su posición.

6 SIMULACIÓN VREP

Una vez se ha comprendido cada uno de los algoritmos necesarios para implementar el método Montecarlo, se puede pasar a la parte de la programación en el simulador. En esta parte se describirá todo el trabajo realizado en el programa así como los resultados obtenidos.

6.1 Entorno de Trabajo

6.1.1 Simulador V-REP

El programa que se ha utilizado para este trabajo es el V-REP. Este simulador de robots es una herramienta muy potente que permite desarrollar rápidamente algoritmos, simular procesos industriales, realizar prototipos y verificarlos o incluso para la enseñanza. Es un programa con una interfaz gráfica bastante intuitiva.

El funcionamiento de este simulador se basa en una arquitectura de control distribuido, es decir, cada uno de los objetos que se representen en el escenario de trabajo, pueden ser controlados individualmente a través de ya sea, una secuencia de comandos incrustada, un plug-in, un nodo de ROS, un cliente de API remota o incluso a través de una solución personalizada. Esto hace que V-REP sea muy versátil para simular procesos llevados a cabo por robots. Además, este simulador permite programa en distintos lenguajes como: Matlab, C, C++, Octave, Python, Lua, Java o Urbi.

En este trabajo todas las implementaciones que se han realizado han sido programadas con el lenguaje Lua.

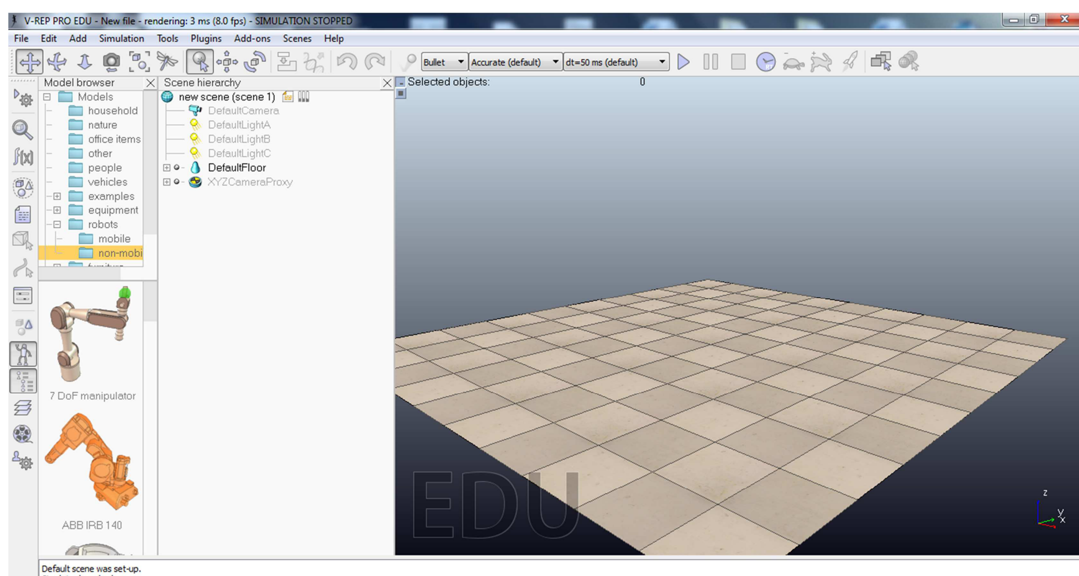


Figura 6-1 Entorno V-REP

6.1.2 Robot utilizado

Dentro del programa V-REP se puede escoger numerosos robots que vienen predeterminados con el simulador. En este trabajo, teniendo en cuenta el objetivo de este así como los requisitos necesarios para poder implementar el código se ha escogido el robot PIONEER P3-DX. Es un robot que se utiliza mucho en las investigaciones de la robótica móvil. Posee dos ruedas con sus respectivos encoders además de un sistema de sensores de ultra sonidos.

Figura 6-2 Robot Pioneer P3-dx real

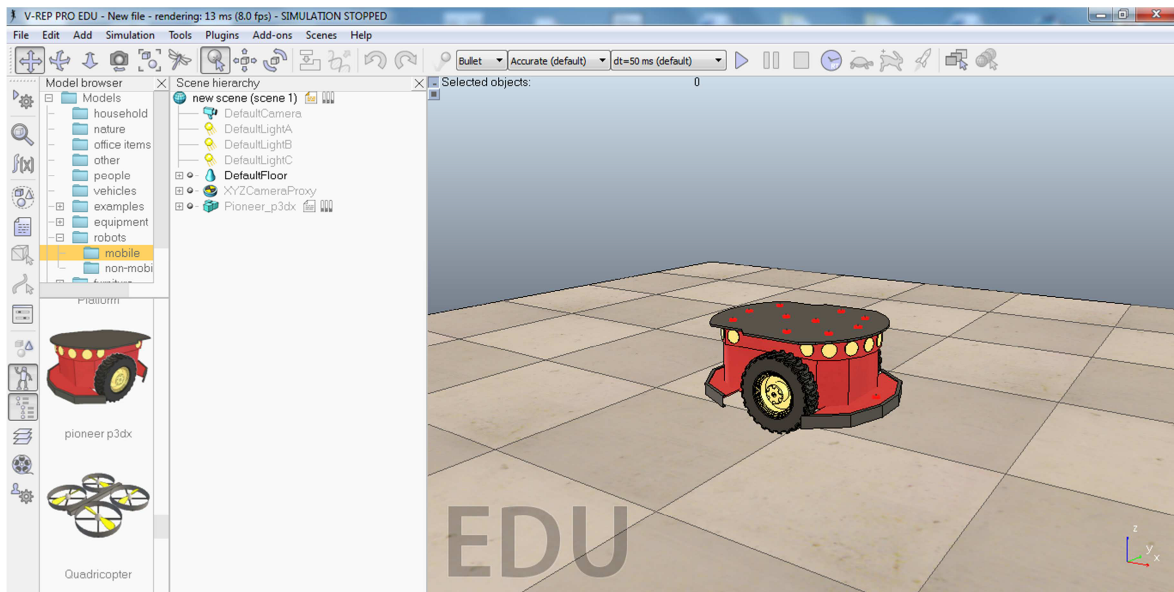


Figura 6-3 Modelo 3d del robot Pioneer P3-dx

Este robot lleva ya incorporado un sistema para evitar obstáculos basado en el algoritmo de Brainterberg. Para ello utiliza los sensores ultrasonidos para detectar los obstáculos de su entorno y modifica la velocidad de cada una de las ruedas según las mediciones realizadas.

```
--Evitacion Obstaculos
for i=1,16,1 do
    res,dist=simReadProximitySensor(usensors[i])
    if (res>0) and (dist<noDetectionDist) then
        if (dist<maxDetectionDist) then
            dist=maxDetectionDist
        end
        detect[i]=1-((dist-maxDetectionDist)/(noDetectionDist-maxDetectionDist))
    else
        detect[i]=0
    end
end

vLeft=v0
vRight=v0

for i=1,16,1 do
    vLeft=vLeft+braitenbergL[i]*detect[i]
    vRight=vRight+braitenbergR[i]*detect[i]
end
```

Figura 6-4 Algoritmo Brainterberg

6.2 Modificaciones realizadas sobre el modelo predeterminado

En el modelo predeterminado del programa, el robot PIONEER P3-DX solo dispone de sensores ultra sonidos (sonar). En este trabajo, se ha obtado por utilizar un sensor laser para realizar las medidas necesarias. Si estuviéramos trabajando con el robot real, se elegiría, en el caso de que se pudiera, el sensor laser ya que es más rápido que el sensor ultrasonido a la hora de tomar medidas. Aquí en el simulador la elección que se ha hecho tiene que ver con la posibilidad de visualizar los rayos laser a la hora de simular el proceso.

El sensor que se ha elegido es el Laser Hokuyo URG **Figura 6-6**

Figura 6-5 Robot con láser incorporado

Figura 6-6 Laser Hokuyo URG

Además de incorporar este laser se ha tenido que realizar unas pequeñas modificaciones. La **Figura 6-7** representa el sensor láser predeterminado que viene con el programa. En cambio en la **Figura 6-8** podemos ver el láser con sus respectivas modificaciones. Se puede observar que el número de rayos laser ha disminuido notablemente.

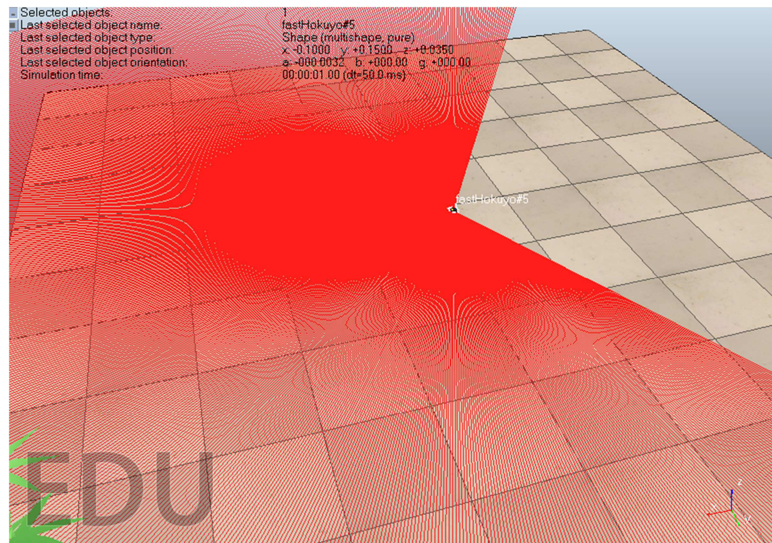


Figura 6-7 Sensor laser predeterminado

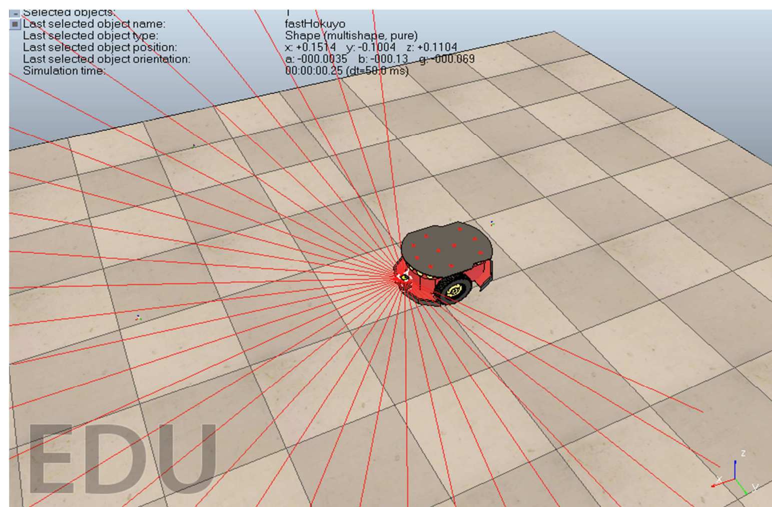


Figura 6-8 Sensor laser tras modificación

Esta modificación se ha realizado para reducir los tiempos de simulación. Como el entorno de trabajo que se va a crear va a ser un entorno sencillo, es decir, no van a haber objetos difíciles de detectar, podemos reducir el abanico de lecturas obteniendo igualmente los resultados deseados.

6.3 Representación de las partículas

Para la representación de las partículas, se han utilizado un tipo de objeto en el programa que se llama “dummy”. Con ello se le puede asignar a cada partícula unas coordenadas y una orientación. Además a cada partícula se le ha equipado con un sensor laser Hokuyo, el mismo que lleva el robot. De esta forma, utilizando estos sensores, se puede conocer en cada momento y para cada partícula que es lo que esta “viendo”. Esto facilitará la tarea a la hora de comparar las medidas realizadas por cada una de las partículas y la realizada por el robot.

6.4 Implementación del Filtro Monte Carlo

En este apartado se explicara cómo se ha implementado el algoritmo del filtro de partículas basado en el método Montecarlo.

6.4.1 Función para el muestreo aleatorio

Como ya se ha explicado en el apartado **METODO MONTECARLO**, el primer paso que hay que realizar para la implementación de este algoritmo es generar M partículas y distribuirlas aleatoriamente por todo el entorno donde el robot va a operar. Para ello se ha utilizado la distribución normal que se ha presentado en el apartado **Muestreo aleatorio de partículas** la fórmula matemática es la siguiente:

$$\frac{b}{6}\sum_{i=1}^{12} random\ (-1,1)$$

Lo que se ha hecho en el programa ha sido crear una función que simula la fórmula matemática representada arriba, de tal manera que siempre que se necesite generar números aleatorios según este tipo de distribución se hará una llamada a esta función definida en la cabecera del programa.

Se puede observar la forma en la que se ha implementado en la siguiente figura.

```
function randomFloat(lower, greater)
      aleatorio=0
     for i=1,12,1 do
      random= lower + math.random()  * (greater - lower)
      aleatorio=aleatorio+random
      end
      aleatorio=aleatorio*(greater/6)
    return aleatorio
end
```

Figura 6-9 Función para generar número aleatorio

Math.random () es una función que viene con las librerías de Lua. Lo que hace esta función exactamente, tal cual está escrita arriba, es generar números aleatorios entre 0 y 1.

Los valores "lower" and "greater" serán en este caso un valor (en metros) que dependerá de la cantidad de metros cuadrados que se quiera abarcar a la hora de distribuir las partículas por el mapa. Así, para repartir las muestras a lo largo de una distancia de 5 metros, el valor de "lower" deberá valer -2.5m y el de "greater" 2.5m. A continuación se explicara cómo se ha utilizado exactamente esta funciona para simular el muestreo inicial de todas las partículas.

Una vez se ha implementado la función que genera números aleatorios entre dos valores, ahora lo que falta es aplicar esa función a cada una de las coordenadas de cada partícula así como al vector orientación de tal manera que obtengamos la distribución deseada. A partir de ahora cuando se hable de etapa de inicialización significara que el código debe estar implementado dentro del siguiente bucle:

```
if (simGetScriptExecutionCount()==0) then

    end
```

Figura 6-10 Código que marca parte de inicialización

Todo el código que se encuentra dentro de ese bucle, significa que solo se ejecuta una vez, justo antes de que comience la simulación.

El código que realiza la distribución inicial de las partículas es el siguiente:

```
--Inicializacion particulas
for i=1,numParticulas,1 do

    x[i]=randomFloat(-1.25,1.25)
    y[i]=randomFloat(-1.25,1.25)
    tecta[i]=randomFloat(-0.25,0.25)
    posicioninicial[i]={x[i],y[i],0.1388}
    anguloinicial[i]={0,0,tecta[i]}

    simSetObjectPosition(handle[i],-1,posicioninicial[i])
    simSetObjectOrientation(handle[i],-1,anguloinicial[i])
end
```

Figura 6-11 Código para inicializar posición partículas

Con este código se obtiene la siguiente distribución de partículas.

Figura 6-12 Distribución inicial partículas

6.4.2 Función para la odometría

A continuación se va a explicar la forma en la que se ha realizado la lectura de los encoders de las ruedas además de como se ha utilizado estos datos para la implementación del algoritmo que aplica el desplazamiento aproximado del robot a cada una de las partículas. En la siguiente figura se muestra la parte del código que se ha de situar dentro del bucle de inicialización.

```
motorLeft=simGetObjectHandle("Pioneer_p3dx_leftMotor")
motorRight=simGetObjectHandle("Pioneer_p3dx_rightMotor")

leftEncoderID=simGetObjectHandle('Pioneer_p3dx_leftWheel')
rightEncoderID=simGetObjectHandle('Pioneer_p3dx_rightWheel')

pioneerID=simGetObjectHandle('Pioneer_p3dx')
childScriptHandle=simGetScriptHandle('MCL')


oldAngleL=simGetObjectOrientation(leftEncoderID,pioneerID)
oldAngleR=simGetObjectOrientation(rightEncoderID,pioneerID)

b=0.3310/2 -- wheel base
r=0.195/2 -- wheel radius
x1=0 -- X position in meters
y1=0 -- Y posiyion in meters
```

Figura 6-13 Inicialización de los manejadores de cada objeto

Lo que se está haciendo aquí es obtener en la etapa de inicialización, los distintos identificadores de motor izquierdo, derecho, rueda izquierda, rueda derecha, robot, y MCL.

Para ello, se ha utilizado la función *simGetObjectHandle* que viene predeterminada con la librería del programa. Además, se han utilizado las funciones *simGetScriptHandle* y *simGetObjectOrientation.* La primera de ellas sirve para guardar en una variable (en este caso "childScriptHandle") el identificador de un script (MCL es un objeto que contiene el conjunto de todas las partículas), en cambio la segunda de ellas sirve para obtener la orientación del objeto deseado. Por último, se puede observar que se han inicializado las variables que representan el radio y la distancia entre ruedas del robot.

Resumiendo, con este código lo que se ha hecho es lo siguiente:

- Asignar el identificador de cada una de las ruedas, cada uno de los motores, y el del robot mismo a su variable respectiva
- Obtener la orientación de cada una de las ruedas
- Inicializar los parámetros: radio y distancia entre rueda que se utilizarán más adelante para calcular los desplazamientos.

Tras inicializar los parámetros necesarios, lo siguiente que se ha de hacer es crear una función para normalizar los ángulos que se obtengan tras la lectura y otra función para realizar la resta entre el nuevo ángulo que tiene la rueda y el que tenía en el instante anterior. De esta forma lo que se está calculado es el incremento del ángulo de cada una de las ruedas.

Para normalizar los ángulos que se obtienen tras la lectura de la orientación de cada una de las ruedas, se ha implementado la siguiente función:

```
--FUNCION NORMALIZAR ANGULO

function (alpha,center)
alpha = (alpha-center+math.pi)%(2*math.pi) + center-math.pi
return alpha
end
```

Figura 6-14 Función normalizar ángulo

La segunda función, de la cual se ha hablado arriba, es la de realizar la diferencia entre dos ángulos y además normalizar el resultado. El código necesario para realizar esta operación es:

```
--FUNCION ANGLEDIFF

function angleDiff(angle1, angle2)
angle1 = normalizeAngle(angle1,math.pi)
angle2 = normalizeAngle(angle2,math.pi)
--compute difference and normalize in [-pi pi]
dif=normalizeAngle(angle2-angle1,0)
return dif
end
--------
```

Figura 6-15 Función para calcular diferencia entre ángulos

Después de haber creado las dos funciones, se ha de programar las ecuaciones para el cálculo del desplazamiento aproximado utilizando la medida realizada por cada uno de los encorders. Estas ecuaciones ya se han comentado en el apartado Calculo del desplazamiento aproximado por odometría

```
--IMPLEMENTACION ODOMETRIA

  angleL=simGetObjectOrientation(leftEncoderID,pioneerID)
  phiL=angleDiff(oldAngleL[3],angleL[3])

  angleR=simGetObjectOrientation(rightEncoderID,pioneerID)
  phiR=angleDiff(oldAngleR[3],angleR[3])

  deltaS=r2*(phiL+phiR)/2
  deltaTheta=r2*(phiR-phiL)/((b))
```

```
 xbarra_prima=xbarra+deltaS*math.cos(tectabarra)
 ybarra_prima=ybarra+deltaS*math.sin(tectabarra)
 tectabarra_prima=tectabarra+deltaTheta

 oldAngleL[3]=angleL[3]
 oldAngleR[3]=angleR[3]
```

Figura 6-16 Código para cálculo posición por odometría

Finalmente, cuando ya se conocen las nuevas coordenadas calculadas con el desplazamiento aproximado del robot, se puede pasar a calcular directamente el algoritmo que aplicara dicho desplazamiento a cada una de las partículas generadas. En la siguiente figura se puede observar una posible manera de implementarlo.

```
for i=1,numParticulas,1 do
    handle3[i]=simGetObjectHandle('Particula_#' .. i)
    posicionresampleada[i]=simGetObjectPosition(handle3[i],-1)
    orientacionresampleada3[i]=simGetObjectOrientation(handle3[i],-1)
    --handle[i]=simGetObjectHandle('Particula_#' .. i)
    --posicionresampleada[i]=simGetObjectPosition(handle[i],-1)
    --orientacionresampleada3[i]=simGetObjectOrientation(handle[i],-1)

    x[i]=posicionresampleada[i][1]
    y[i]=posicionresampleada[i][2]
    tecta[i]=orientacionresampleada3[i][3]

    delta_rot1=math.atan2(ybarra_prima-ybarra,xbarra_prima-xbarra)
    -tectabarra
    delta_trans=math.sqrt((xbarra-xbarra_prima)*(xbarra-xbarra_prima)
    +(ybarra-ybarra_prima)*(ybarra-ybarra_prima))
    delta_rot2=tectabarra_prima-tectabarra-delta_rot1

    b11=alfa1*delta_rot1+alfa2*delta_trans
    b12=alfa1*delta_rot2+alfa2*delta_trans
    b2=alfa3*delta_trans+alfa4*(delta_rot1+delta_rot2)

    delta_rot1_hat=delta_rot1-randomFloat(-b11,b11)
    delta_rot2_hat=delta_rot2-randomFloat(-b12,b12)
    delta_trans_hat=delta_trans-randomFloat(-b2,b2)

    x_prima[i]=x[i]+delta_trans_hat*math.cos(tecta[i]+delta_rot1_hat)
    y_prima[i]=y[i]+delta_trans_hat*math.sin(tecta[i]+delta_rot1_hat)
    tecta_prima[i]=tecta[i]+delta_rot1_hat+delta_rot2_hat
    angulo_euler_tectaprima[i]={0,0,tecta_prima[i]}
```

Figura 6-17 Código algoritmo sample_model_odometry

Las funciones *math.tan2* y *math.sqrt* que se utilizan en el código de arriba, vienen predeterminadas con la librería del programa. Sin embargo la función *randomFloat* es la misma que se ha implementado al principio y que se ha comentado en la página 43. Los valores de alfa1, alfa2, alfa3 y alfa4 son constantes que se inicializan al principio del programa. Estos valores afectan a la distribución que se va a obtener para el movimiento de cada partícula como ya se ha comentado en la página 29

Después de calcular las nuevas coordenadas a aplicar a cada punto, se ha de utilizar unas funciones para mover cada partícula a las coordenadas calculadas. Además, se ha de actualizar los datos de tal manera que la nueva posición que se ha calculado se guarde en la variable de la antigua posición para que en la siguiente iteración se haga el respectivo cálculo a partir de esas coordenadas. Para ello se ha procedido de la siguiente forma:

```
 angulo_euler_tectaprima[i]={0,0,tecta_prima[i]}
 nuevaposicion[i]={x_prima[i],y_prima[i],0.1388}
 nuevoangulo[i]=angulo_euler_tectaprima[i]

 handle4[i]=simGetObjectHandle('Particula_#' .. i)
 simSetObjectPosition(handle4[i],-1,nuevaposicion[i])
 simSetObjectOrientation(handle4[i],-1,nuevoangulo[i])

 x[i]=x_prima[i]
 y[i]=y_prima[i]
tecta[i]=tecta_prima[i]

end

 ybarra=ybarra_prima
 xbarra=xbarra_prima
 tectabarra=tectabarra_prima
```

Figura 6-18 Código para cambiar la posición de las partículas y actualizar valores

Como se ha comentado arriba, se necesitan dos funciones: una para modificar las coordenadas de la partícula, y la otra para modificar su orientación. La primera de ellas es *simSetObjectPosition* y la segunda *simSetObjectOrientation.* Los valores calculados por el algoritmo de la odometría, es decir: *ybarra*, *xbarra* e *tectabarra* también han de actualizarse.

Con este trozo de código finalizamos la primera parte de la implementación del método Montecarlo. Lo que se han conseguido con el conjunto de los códigos adjuntados arriba, es, en resumen, que cada una de las partículas imite el movimiento del robot.

6.4.3 Función para la ponderación de las partículas

Para esta parte del algoritmo, se han tenido que implementar varias funciones para poder realizar el cálculo de la probabilidad de cada partícula.

Para poder ponderar las partículas, se ha de comparar las lecturas de los sensores laser de cada una de ellas con la lectura del sensor que porta el robot. Los pasos que se han seguido para lograr esto son:

- Para el sensor robot: guardar la medida de cada uno de los rayos laser en un vector
- Para cada partícula: leer las medidas que realiza su propio sensor y compararlo con las tomadas por el sensor del robot
- Introducir estos valores en las funciones implementadas anteriormente
- Guardar la probabilidad de cada partícula en un vector de tamaño el número de partículas.

La función principal que se ha creado se llama probabilidad. Esta función lo que va a hacer es devolver la probabilidad que tiene cada una de las partículas. Dentro de esta función se han implementado las funciones de probabilidad que se explicaron en el apartado 5.1.3

```
--FUNCION PROBABILIDAD
function probabilidad(visionSensor1Handle, visionSensor2Handle, sensorRef, measuredData)
```

```
--IMPLEMENTACION PSHORT
function pshort(zt, zt_real, landa_short)
    resultado=0
    if (zt_real>=zt) and (zt>=0) then
        mu=(1/(1-math.exp(-landa_short*zt_real)))
        resultado=mu*landa_short*math.exp(-landa_short*zt_real)
    else
        resultado=0
    end
    print('pshort')
    print(resultado)
    return resultado
end
```

Figura 6-19 Función pshort

```
--IMPLEMENTACION PMAX
function pmax(zt, zmax)
    resultado=0
    if(zt==zmax) then
        resultado=1
    else
        resultado=0
    end
    print('pmax')
    print(resultado)
    return resultado
end
-----------------------------------
```

Figura 6-20 Función pmax

```
--IMPLEMENTACION PHIT
function phit(zt, zt_real, sigma_hit, zmax)
    resultado=0
    pi=3.141592653589793
    if (zmax>=zt_real) and (zt_real>=0)then
        nu=0
        dz=0.01
        for z=0,maxScanDistance_,dz do
            nu=nu+(math.exp(((-(z-zt_real)*(z-zt_real))
            /(2*sigma_hit*sigma_hit)))/(math.sqrt(2*pi*sigma_hit*sigma_hit)))*dz
            --nu=nu+(math.exp(((-(zt-z)*(zt-z))/(2*sigma_hit*sigma_hit))
            /(math.sqrt(2*pi*sigma_hit*sigma_hit)))*dz
        end
        resultado=(math.exp(((-(zt-zt_real)*(zt-zt_real))
        /(2*sigma_hit*sigma_hit)))/(math.sqrt(2*pi*sigma_hit*sigma_hit)))/nu
        --resultado=math.exp(((-(zt-zt_real)*(zt-zt_real))/(2*sigma_hit*sigma_hit)))
        /(math.sqrt(2*pi*sigma_hit*sigma_hit))
    else
        resultado=0
    end
    print('phit')
    print(resultado)
    return resultado
end
```

Figura 6-21 Función phit

```
--IMPLEMENTACION PRAND
function prand(zt, zmax)
    resultado=0
    if(zmax>zt) then
        resutado=1/zmax
    else
        resultado=0
    end
    print('prand')
    print(resultado)
    return resultado
end
----------------------------------
```

Figura 6-22 Función prand

Las funciones *pmax*, *prand*, *pshort* y *phit* necesitan de las medidas de los sensores de las partículas así como de las medidas del sensor del robot para poder compararlas y ponderar cada una de las partículas. Para ello se ha de realizar la lectura del sensor del robot y del sensor de cada una de las partículas por separado.

6.4.4 Función para el re-muestreo de las partículas

Una vez las partículas tienen cada una de ellas asignadas una probabilidad, lo que se ha de hacer a ahora es normalizar ese valor de tal forma que la suma de todo el vector de probabilidades sea uno. Esto se ha conseguido utilizando la siguiente función.

```
--Particulas normalizadas (automatico)
suma=0
for i=1,numParticulas,1 do
    qparticulavector[i]=probabilidad(visionSensor1Handle[i],visionSensor2Handle[i],sensorRef[i],measuredData)
    print(qparticulavector[i])
    suma=suma+qparticulavector[i]
end

for i=1,numParticulas,1 do
    qparticulavectornormalizado[i]=qparticulavector[i]/suma
    --print(qparticulavectornormalizado[i])
end
```

Figura 6-23 Normalización vector probabilidad de las partículas

Cuando ya se ha obtenido el vector con cada una de las probabilidades normalizadas, se utiliza una función llamada *resampling* que se ha creado y definido en la cabecera del programa. Esta función contiene un conjunto de funciones que permiten realizar la operación de elección de partículas en función de su probabilidad.

```
--funcion cumprod
function cumprod(variable)
    resultado={}
    N=#variable
    resultado[1]=variable[1]
    for i=1,N-1,1 do
        resultado[i+1]=resultado[i]*variable[i+1]
    end
    return resultado
end
----------------------------------
--funcion fliplr
function fliplr(u1)
    v1={}
    N=#u1
    for i=1,N,1 do
        v1[i]=u1[i]
    end
    --v1=u1
    for i=1,N,1 do
        u1[i]=v1[N+1-i]
    end
    return u1
end
```

Figura 6-24 Función cumprod y fliplr

```
--FUNCION RESAMPLING
function resampling(q)

    ----------------------------------
    --function cumsum
    function cumsum(q1)
        x1={}
        N=#q1
        --print(N)
        x1[1]=q1[1]
        for i=1,N-1,1 do
            x1[i+1]=x1[i]+q1[i+1]
        end
        return x1
    end
    ----------------------------------
```

Figura 6-25 Función resampling

El resultado que devuelve cada una de las funciones mostradas en la Figura 6-24 y Figura 6-25 es:

- La función *cunsum* genera un vector de valores de probabilidades acumuladas
- La función *cumprod* genera un vector con el producto acumulado de los valores aleatorios generados con *math.random* () en *valorcumprod*.
- *Fliplr* es una función que invierte el orden del vector que se le pasa como entrada.

En la última parte del código se compara el vector ordenado de valores aleatorios con el vector de probabilidad acumulada de las partículas. El valor que devuelve esta función es el índice de la posición de las partículas que “han salido en el sorteo”.

```
    u={}
    ut={}
    valorcumprod={}
    i={}
    P={}

    P=cumsum(q)
    N=#q

    for i=1,N,1 do
        valorcumprod[i]= math.pow(math.random(),(1/(N+1-i)))
    end

    u=cumprod(valorcumprod)
    ut=fliplr(u)

    --k=1
    for j=1,N,1 do
    k=1
        while (P[k]<ut[j]) do
            k=k+1
            if (k==N+1) then
                k=N
                break
            end
        end
        i[j]=k
    end

    return i
end
```

Figura 6-26 Implementación función re-muestreo

6.5 Resultados

A continuación se van a mostrar los resultados obtenidos en las pruebas que se han realizado en el simulador y además se comentarán con el objetivo de comparar las simulaciones que se han realizado: simulación con 50 partículas y simulación con 120 partículas.

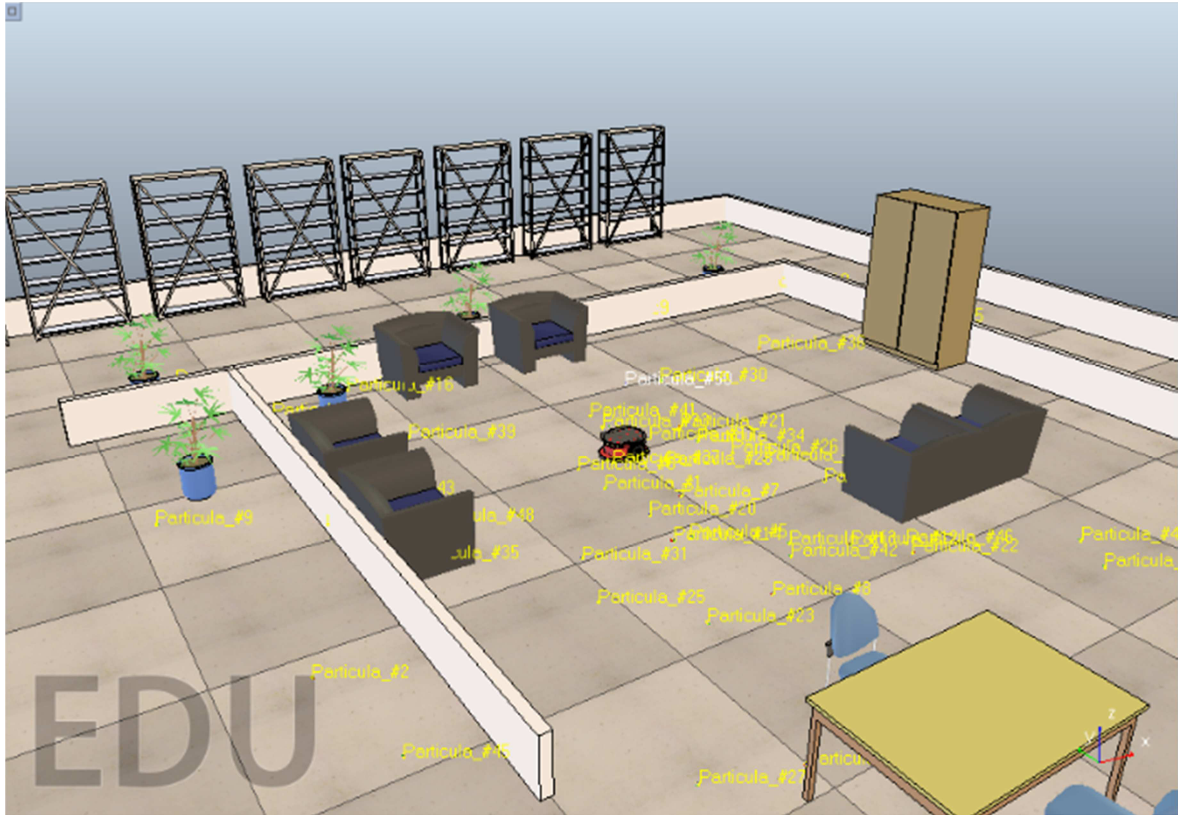


Figura 6-27 Distribución inicial 50 partículas

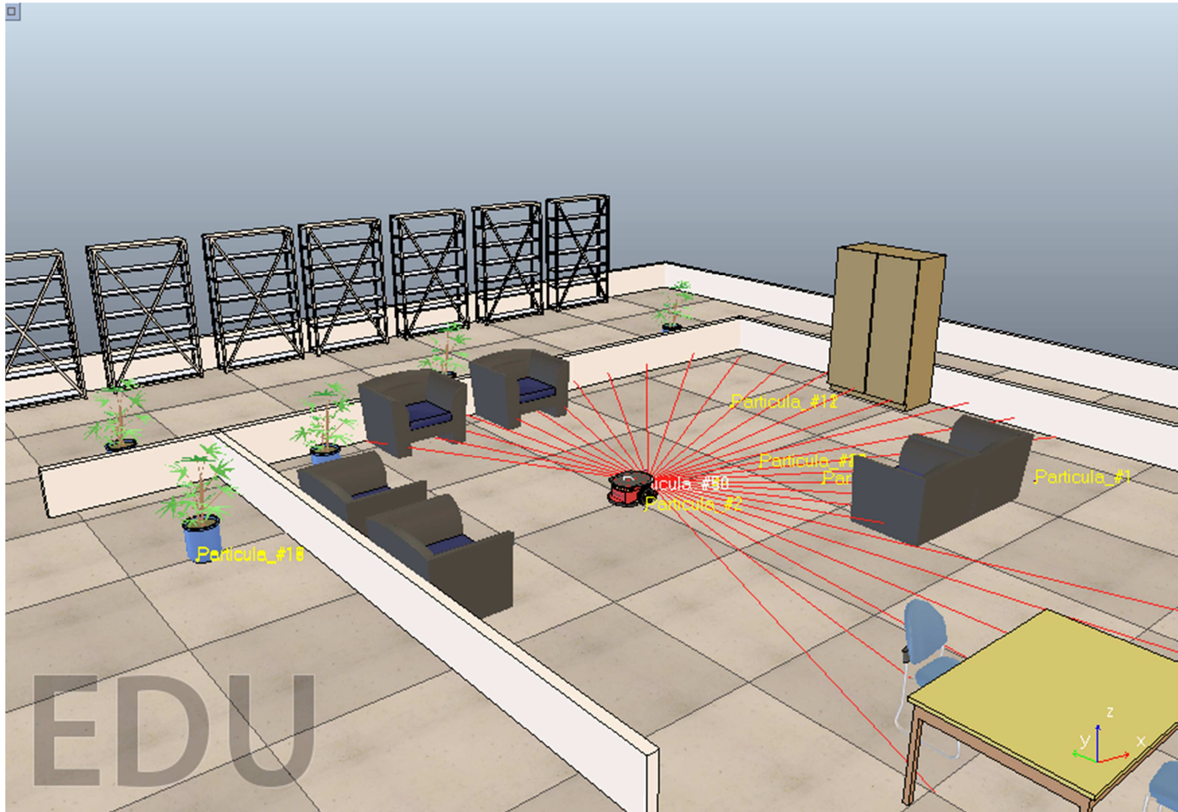


Figura 6-28 Distribución 50 partículas tras varias etapas de re-muestreo

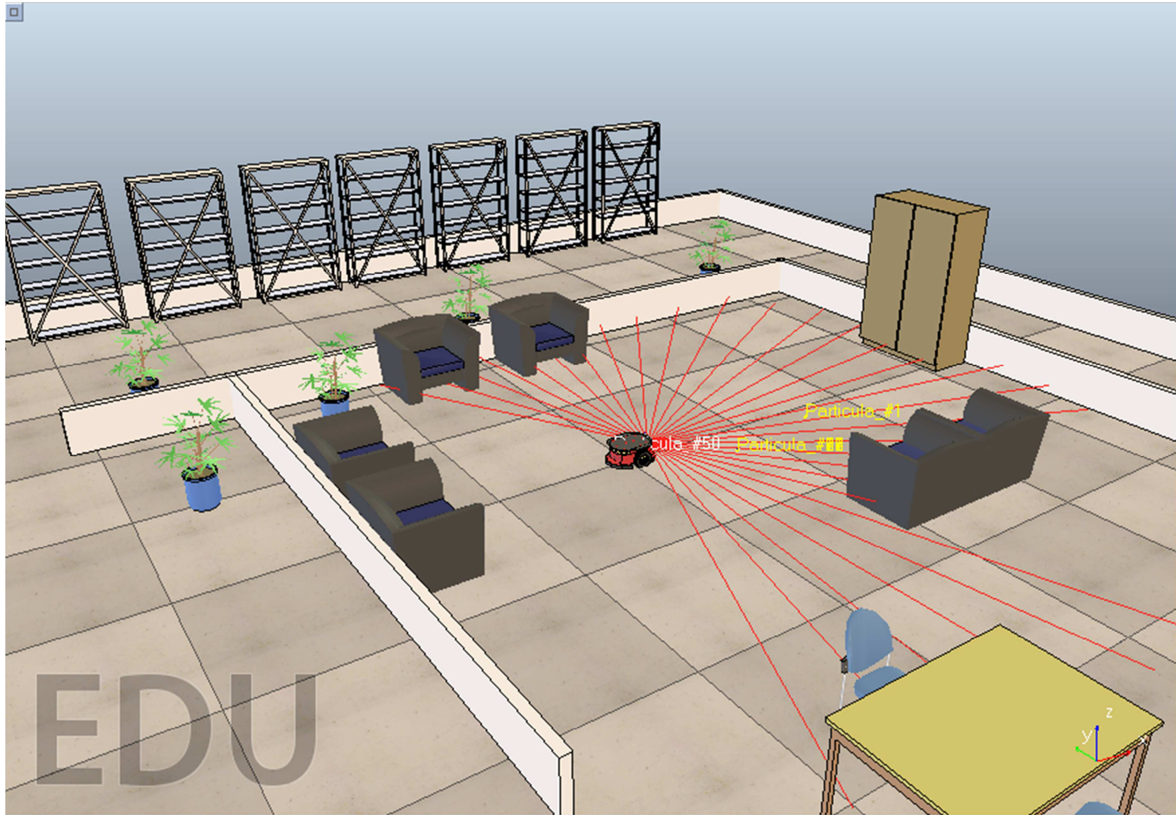


Figura 6-29 Distribución 50 partículas después de varias convergencias

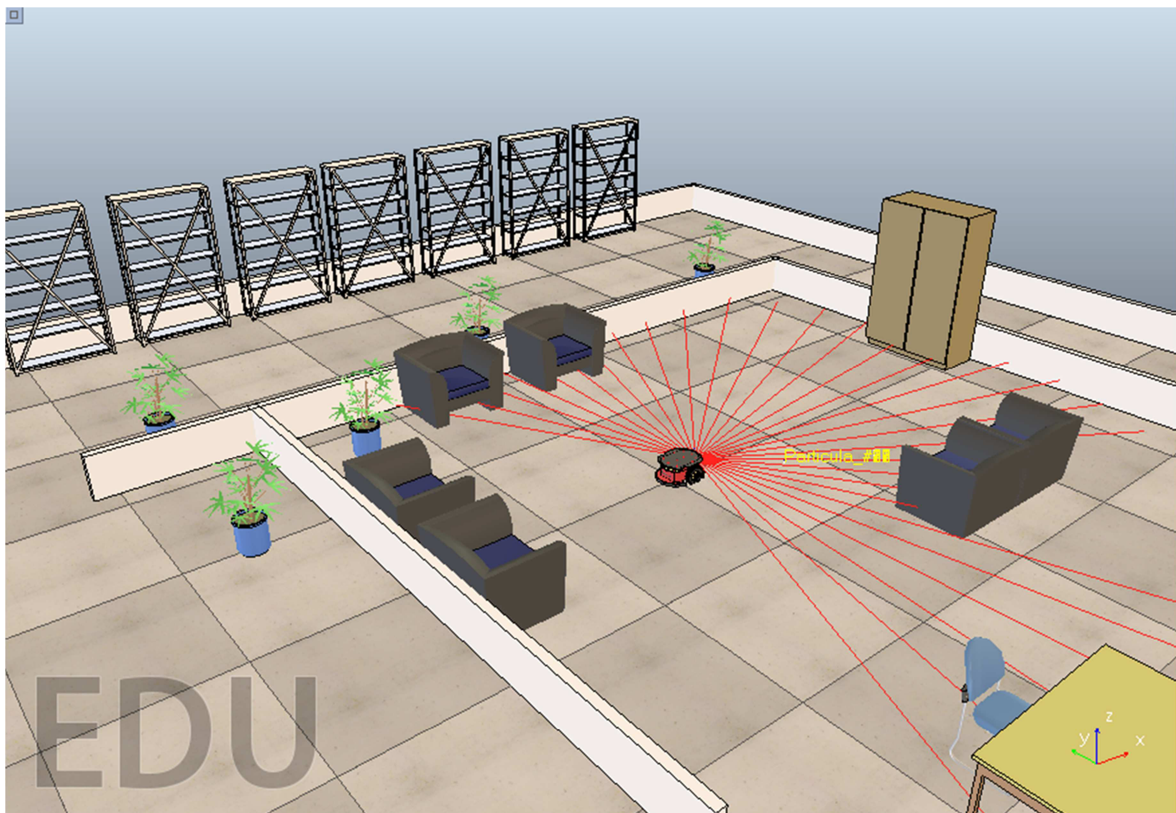


Figura 6-30 Posición estimada por algoritmo AMCL con 50 partículas

Como se puede observar el algoritmo no estima bien la posición del robot. Esto se debe a un problema de resolución que depende del número de partículas elegido. Para esta simulación se han utilizado 50 partículas. Teniendo en cuenta que se ha reducido el número de medidas que toma cada sensor laser y que el número de partículas no es suficiente, el algoritmo nos devuelve una posición estimada poco precisa

Se ha realizado otra prueba probando aumentar el número de partículas generado a 120 obteniendo unos resultados mejores pero sacrificando notablemente el tiempo de computación. Además, hay que añadir que a pesar de simularse muy lentamente, no siempre se obtenía el resultado deseado. Se podría pensar en aumentar aún más el número de partículas para de esta forma dotar al algoritmo de mayor robustez y precisión pero el problema está en que si se aumenta más la cantidad de partículas el simulador va demasiado lento o incluso puede dejar de funcionar.

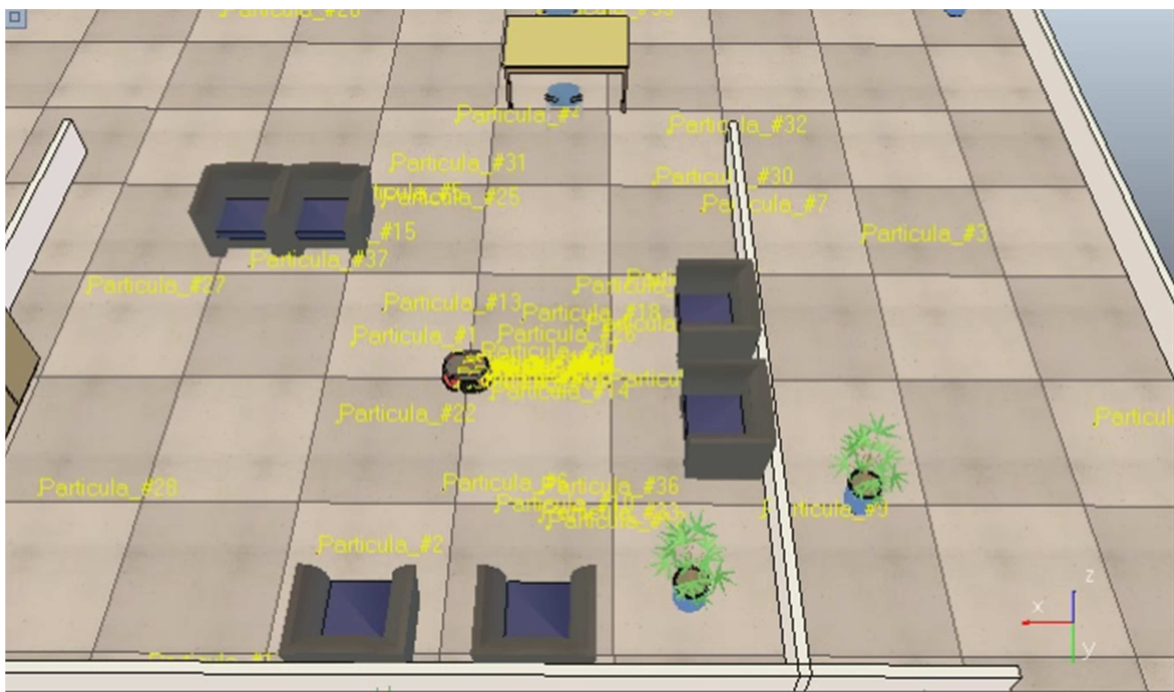


Figura 6-31 Distribución inicial 120 partículas

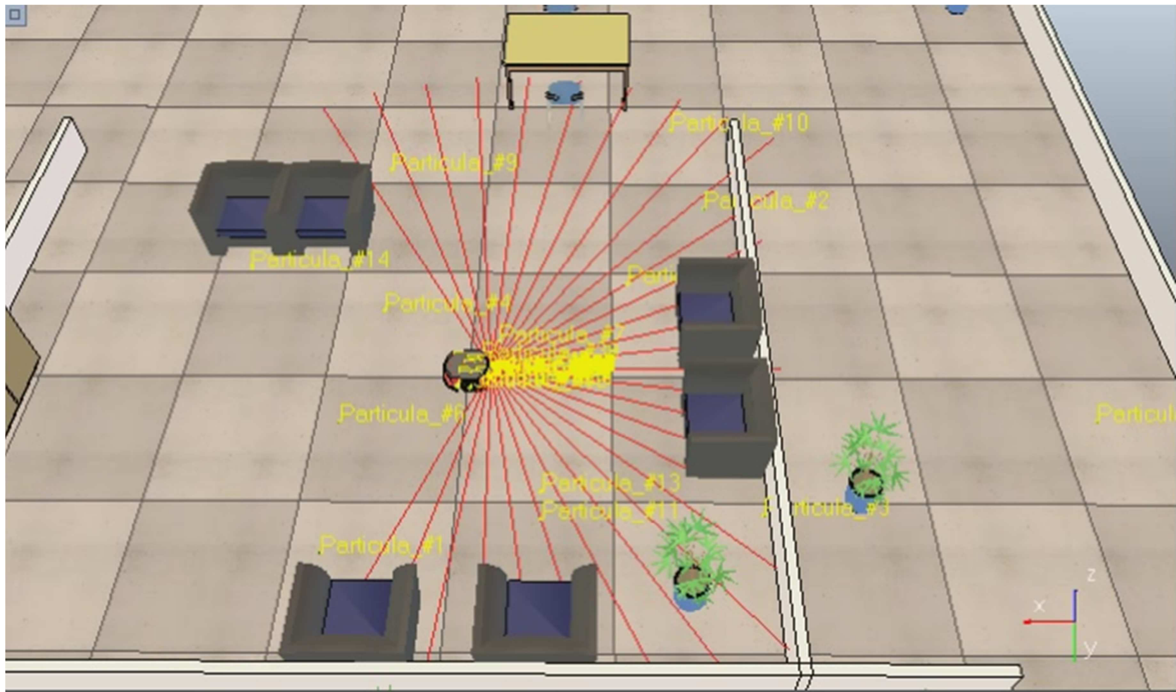


Figura 6-32 Distribución 120 partículas tras varias convergencias

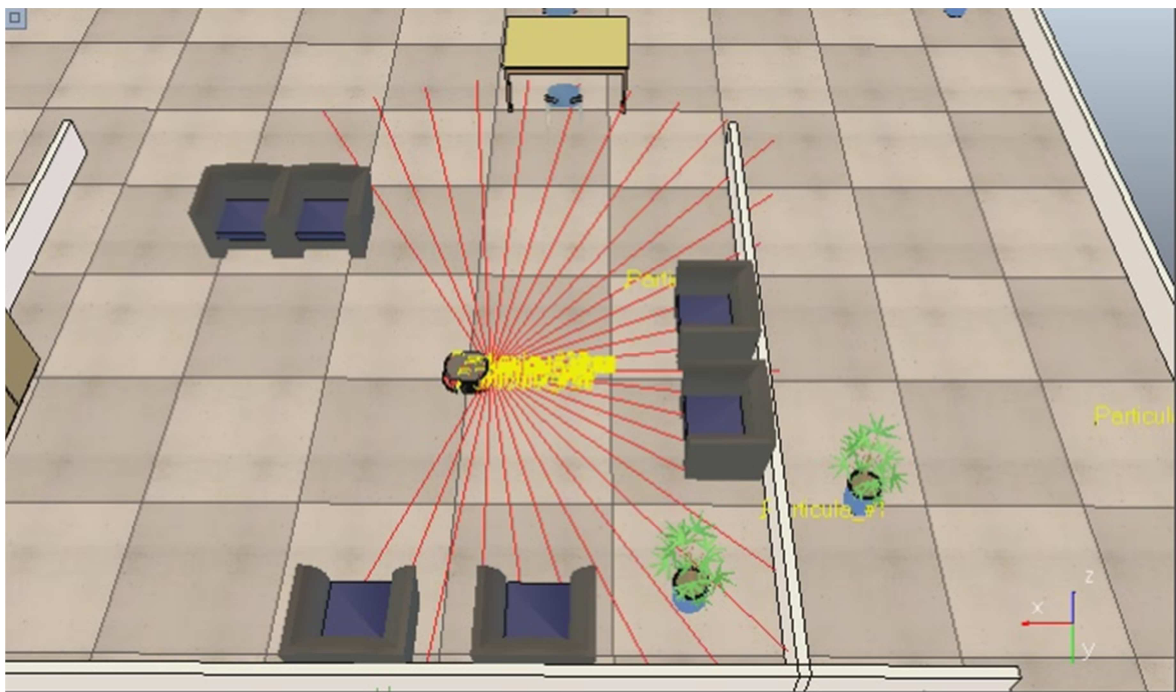


Figura 6-33 Distribución de las 120 partículas

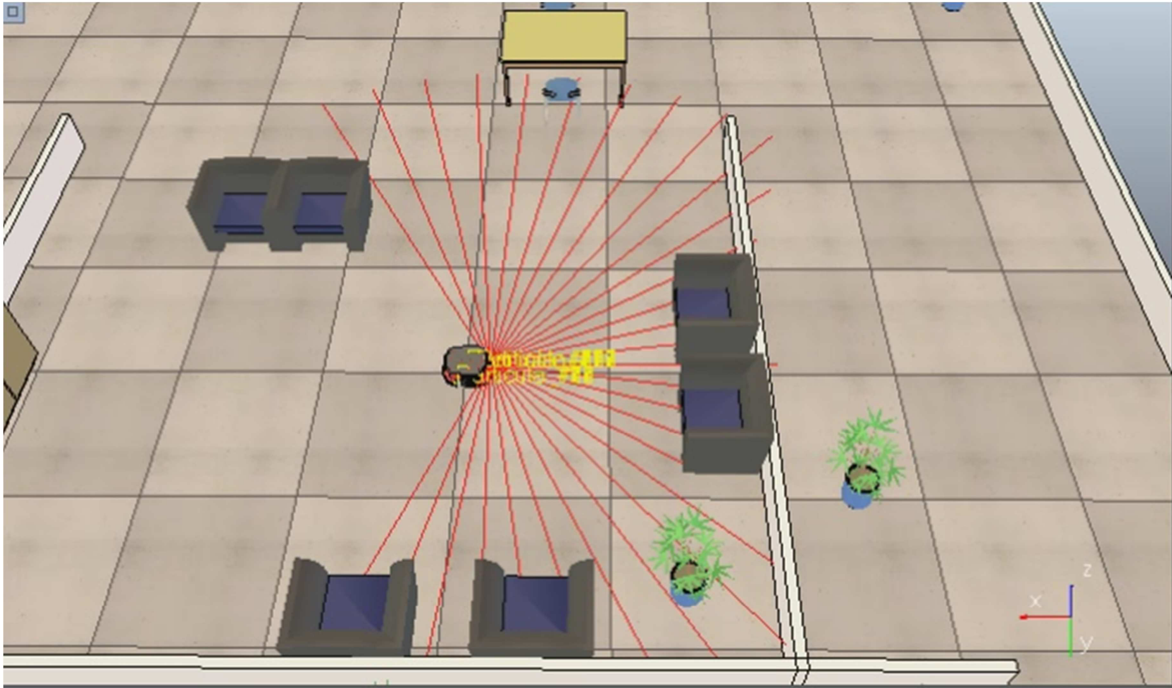


Figura 6-34 Posición final estimada por AMCL

Ya que no se ha podido simular bien el algoritmo de localización AMCL programándolo todo en V-REP, se plantea una alternativa que utiliza otro programa que se pueda comunicar con VREP para realizar los cálculos del algoritmo y el mismo simulador VREP para realizar la toma de medidas.

7 SIMULACION ROS-VREP

Como se ha comentado anteriormente, la simulación a través de VREP es demasiado lenta. Esto se debe a la forma en la que se ha implementado el algoritmo. El programa tiene que realizar muchas operaciones para realizar cada iteración del algoritmo. Por ello, en esta parte se ha propuesto una solución alternativa que utilizará un programa auxiliar para el cálculo de las operaciones necesarias, y el mismo programa utilizado anteriormente, VREP, para gestionar la parte de las mediciones.

7.1 Herramienta ROS

ROS es un meta-sistema operativo de código abierto para robots. Proporciona los servicios que uno espera de un sistema operativo, incluyendo la abstracción del hardware, control de bajo nivel de dispositivos, implementación de las funcionalidades comúnmente utilizadas, paso de mensajes entre procesos, gestión de paquetes, etc. También proporciona herramientas y librerías para obtener construir, escribir, y ejecutar código entre múltiples ordenadores. El grafo de ejecución de un sistema basado en ROS no es más que una red punto a punto de procesos que están parcialmente conectados utilizando la infraestructura de comunicación establecida por ROS a través de mensajes TCP/IP.

7.2 Combinación ROS con VREP

La principal ventaja de combinar ROS con VREP es que se pueden reducir los tiempos de simulación notablemente. Así como cuando se implementaba todo el algoritmo en VREP, el tiempo de ejecución del script era muy alto debido al número de operaciones que tenía que realizar y de la cantidad de objetos que tiene que representar, combinando ROS y VREP se consigue que la simulación sea bastante más rápida. Este aspecto es muy importante cuando se quiere trabajar sobre un algoritmo ya que cada prueba que se desee realizar se podrá ejecutar relativamente rápida. Sin embargo, trabajando solo con VREP, los tiempos de simulación son tan altos que se pierde mucho tiempo para simular un simple proceso.

7.3 Paquete de navegación AMCL

Dado que el código que se ha implementado en V-REP (código LUA) del algoritmo de Montecarlo no sirve para utilizarlo en ROS ya que este último trabaja con lenguaje C o Python, se ha descargado el paquete de navegación AMCL que ofrece ROS en su página de descargas. Este paquete contiene, básicamente, las funciones (en lenguaje C) que se han implementado en VREP además de archivos que permiten ejecutar aplicaciones auxiliares compatibles con el entorno de ROS.

Tanto el programa ROS como sus respectivos paquetes se han descargado de la siguiente dirección: *http://wiki.ros.org/indigo*

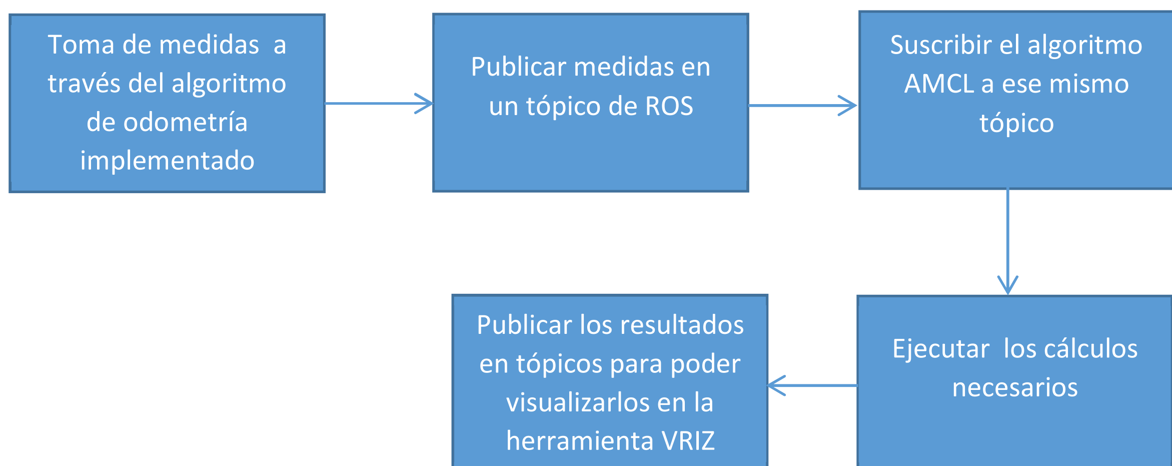
Una vez se tiene instalado todos los paquetes solo resta establecer la conexión entre VREP y ROS.

7.4 Conexión VREP y ROS

Es necesario establecer un "puente" de información entre ambos programas. En este aspecto VREP permite fácilmente la comunicación con programas externos. Existen funciones específicas para exportar e importar información. Una de las formas más utilizadas para compartir la información entre los distintos programas es a través de la creación de tópicos. Estos no son más que un mensaje asociado a un nombre de forma que dos nodos comunican datos el uno con el otro. Cuando uno publica mensajes de un tipo dado, el otro se suscribe al mismo tipo de mensajes y además el nombre del mensaje que pública/ suscribe es el mismo, esto es, el tópico. Para la conexión entre VREP y ROS se han utilizado únicamente este tipo de canal de comunicación que se acaba de describir.

La forma en la que se ha procedido para crear una conexión, de tal forma que el simulador tome las medidas y las envié a ROS donde se ejecutara el algoritmo y este de la misma forma reenviara los nuevos valores actualizados a VREP u a cualquier otra herramienta disponible en la librería de ros, es la siguiente:

- Se tendrá que crear un tópico para habilitar el canal de comunicación
- V-REP publicara las medidas realizadas con los sensores
- ROS recibe esas medidas, realiza los cálculos y devuelve los resultados a tráves de los tópicos.



7.5 Preparación del entorno de simulación

En cuanto al entorno de simulación se utilizara la mayor parte del trabajo realizado para la anterior simulación:

- El entorno será el mismo
- Se cambiara el tipo de laser
- Se utilizara el mismo robot Pioneer P3-dx

Se podrá eliminar todo el código que se implementó en la otra simulación, ya que ese algoritmo estará gestionado ahora por ROS, excepto el de la odometría que se necesitará también en este caso. El robot estará dotado con un sistema de evitación de obstáculos a través de la implementación del algoritmo de Braintenberg.

7.6 Implementación del código de comunicación

Antes de abrir el simulador es importante ejecutar ROS de tal manera que cuando V-REP se abra detecte la conexión entre estos dos programas.

Los pasos que hay que seguir para realizar una buena implementación del algoritmo basado en filtro de partículas son:

- Establecer los canales de comunicación entre VREP y ROS a través del uso de tópicos
- Lanzar la simulación para que el robot vaya moviéndose por el entorno y genere un archivo que contenga el mapa.
- Una vez obtenido el mapa, utilizar el paquete de navegación de ROS para localizar al robot en el mapa generado.
- Visualizar los resultados en la herramienta RVIZ

En primer lugar, para establecer la comunicación entre los dos programas se ha utilizado el siguiente código

```
sensorRef=simGetObjectHandle("base_laser_link")
baseRef=simGetObjectHandle("base_link")
topicName = simExtROS_enablePublisher('/base_scan',1
,simros_strmcmd_get_laser_scanner_data, sensorRef,-1, baseRef'')
tfname = simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform,
sensorRef, baseRef,'')
```

Figura 7-1 Publicación de tópicos

Esto nos permite generar dos tópicos: base_link y tf .El primero sirve para que VREP publique las medidas realizadas con el sensor. El segundo sirve para publicar las transformaciones entre los sitemas de referencia de la posición marcada por baseRef (posición estimada por odometría) y la del láser (sensorRef). Es necesario realizar esta transformación ya que se ha de tener en cuenta la distancia que hay entre la posición del láser y el robot ya que esto influirá en las medidas realizadas.

Además se ha de publicar los datos de la odometría ya que más adelante la herramienta RVIZ necesitara de estos valores para las representaciones. Se procede de la misma forma que para publicar los datos del láser.

```
odomID = simGetObjectHandle('odom')
baseLinkID = simGetObjectHandle('base_link')
topicName = simExtROS_enablePublisher('/odom',1
,simros_strmcmd_get_odom_data, baseLinkID, odomID,'')
tfname = simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform
,baseLinkID, odomID,'')
```

Figura 7-2 Publicación de tópicos para odometría

En segundo lugar, ROS dispone de varias herramientas para la construcción de mapas con diferentes tipos de sensores. Para sensores tipo laser, una de las más utilizadas es el paquete "*gmapping*". Esta herramienta es capaz de construir un mapa y localizar la posición del robot de manera simultánea.

Para ello se debe ejecutar el nodo "*slam_gmapping*" que está dentro del paquete "*gmapping*". Por defecto este nodo se suscribe a los tópicos "*tf*" y "*scan*". El primero de ellos contiene mensajes con las transformaciones entre sistemas de referencia y el segundo contiene los mensajes del sensor Laser. Por su parte el nodo *slam_gmapping* publica el mapa estimado a través del tópico *map*. ROS permiten ajustar múltiples parámetros como la calidad de los resultados, la resolución del mapa, el tamaño etc. Los parámetros que vienen predeterminados sirven para resolver un gran abánico de problemas por lo que solo hará falta ajustar algunos de ellos.

Para lanzar el nodo *slam_gmapping* hay que crear un fichero de tipo *launch*. Para ello se ha escribir el siguiente código con el editor de textos:

```
<launch>
<node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen">
<rosparam>
odom_frame: odommap_update_interval: 30.0
maxUrange: 8.0
particles: 80
xmin: -8.0
ymin: -8.0
xmax: 8.0
ymax: 8.0
delta: 0.05
base_frame: base_link
</rosparam>
<remap from="scan" to="base_scan"/>
</node>
</launch>
```

Figura 7-3 creación archivo launch de slam_gmapping

Donde:

- *Odom_frame* es el objeto que lleva incorporado el sistema de odometría
- *maxUrange* es el rango máximo del sensor
- *particles* es el número de partículas que se van ha utilizar
- *xmin* coordenada x mínima del mapa inicial
- *ymin* coordenada y mínima del mapa inicial
- *xmax* coordenada y máxima del mapa inicial
- *ymax* coordenada x máxima del mapa inicial
- *delta* es la resolución del mapa
- *base_frame* es el objeto móvil que representa la posición aproximada por odometría

Una vez se ha ejecutado el nodo, solo hace falta poner el simulador V-REP en marcha para que el robot vaya moviéndose por el entorno. A medidas que el sensor vaya realizando medidas, estas serán enviadas a través de los tópicos a ROS generando de esta forma el mapa. Para guardar el mapa se ha de utilizar un comando de ROS que genera dos archivos contiendo información sobre el mapa. Uno de ellos contiene el mapa en formato de imagen y el otro los datos como resolución, el tamaño etc.

```
rosrun map_server map_saver
```

Figura 7-4 Comando para guardar el mapa en un archivo

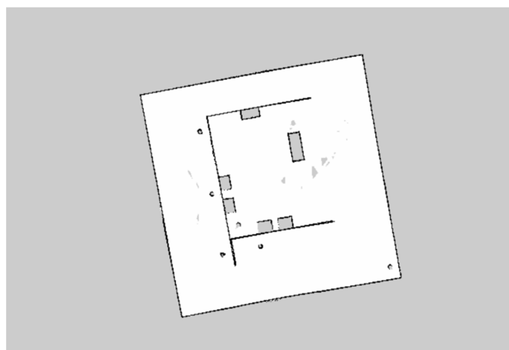


Figura 7-5 Mapa del entorno

```
image: map.pgm
resolution: 0.050000
origin: [-19.200000, -19.200000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figura 7-6 Archivo que contiene meta-datos del mapa

Finalmente, cuando ya se haya generado el mapa se puede trabajar con el algoritmo *AMCL* para que dibuje, sobre ese mapa, la posicione estimada del robot con el uso de una población de partículas. Para ello se ha de generar el siguiente archivo launch que permitirá iniciar el nodo AMCL.

```
<launch>
<node name="map_server" pkg="map_server" type="map_server" args="$(find alex)/map/map.yaml"/>
<node pkg="amcl" type="amcl" name="amcl" respawn="true">
<remap from="scan" to="base_scan"/>
<param name="initial_pose_x" value="0.0" />
<param name="initial_pose_y" value="0.0" />
<param name="initial_pose_a" value="0.0" />
<param name="odom_model_type" value="diff"/>
<param name="odom_frame_id" value="odom"/>
<param name="min_particles" value="100"/>
<param name="max_particles" value="1000"/>
</node>
</launch>
```

Figura 7-7 Archivo para iniciar nodo AMCL

Para poder visualizar los resultados, se utiliza la herramienta de visualización 3d, RVIZ, incluida en el paquete de ROS. Esta permite obtener una visión general en tiempo real de la conectividad entre los nodos además de facilitar la supervisión, adquisición y manipulación de datos. De esta forma, se le pasa el mapa generado y los valores calculados por el algoritmo AMCL y lo que hace RVIZ es representar la posición de cada una de las partículas.

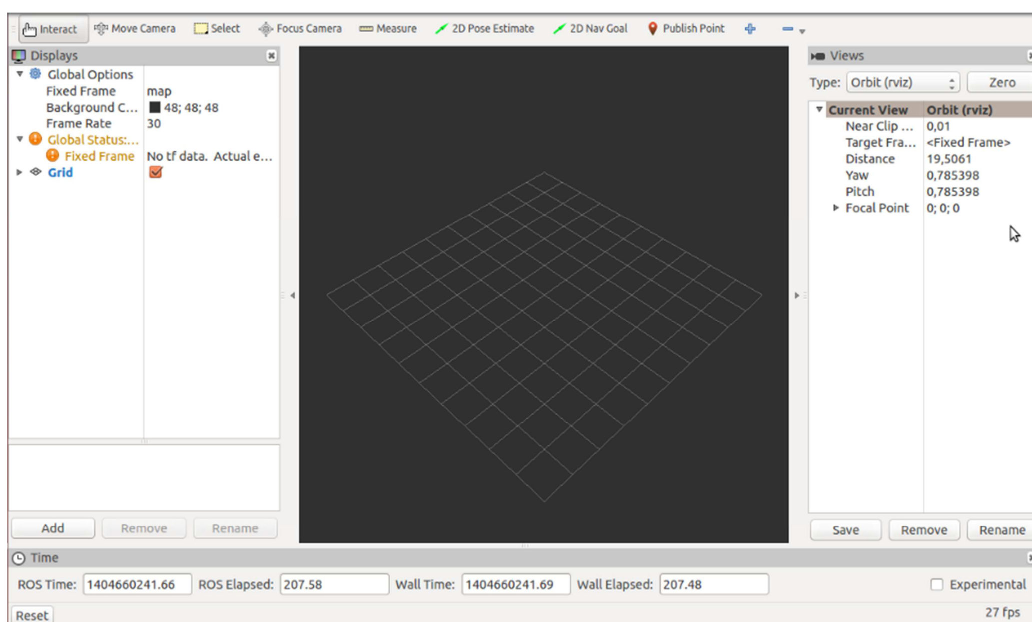


Figura 7-8 Herramienta RVIZ

7.7 Resultados

Tras la simulación del problema de localización del robot PIONEER P3-DX se representa a continuación los resultados obtenidos:

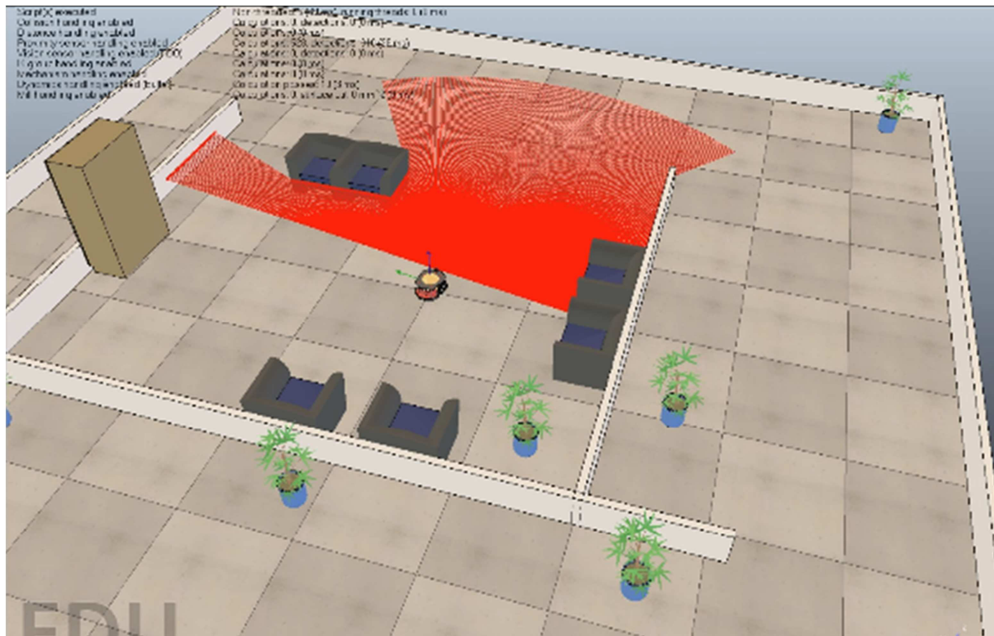


Figura 7-9 Resultado Instante inicial- VREP

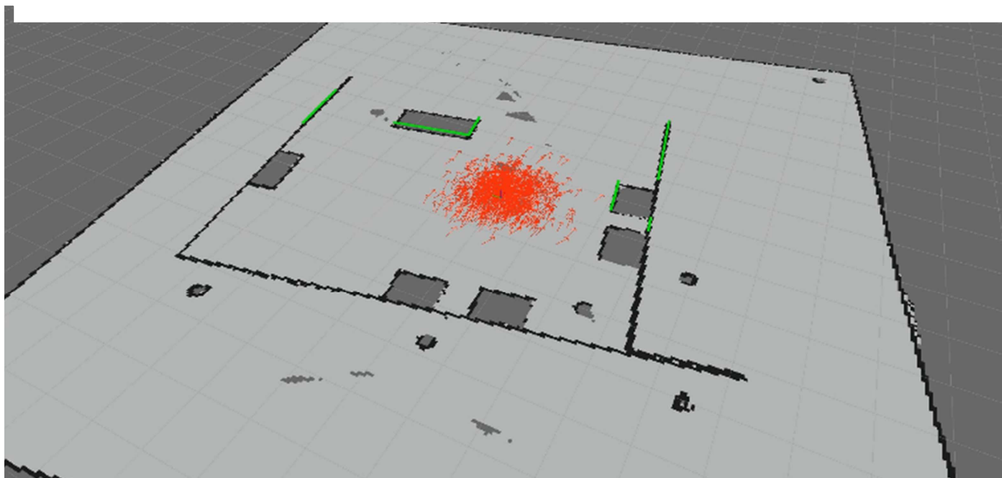


Figura 7-10 Resultado Instante inicial- RVIZ

La **Figura 7-9 Resultado Instante inicial- VREP**representa el instante inicial, justo cuando se pone en marcha la simulación. En la figura **Figura 7-10** se puede observar el entorno de la herramienta VRIZ. Esta permite dibujar la posición de las partículas generadas por el algoritmo AMCL. Se puede observar que en el instante inicial existe cierta dispersión de las partículas. Si bien se podría haber generado una distribución de partículas que ocupara todo el mapa (se hace para problemas de localización global) en este caso se ha generado la muestra de partículas con una dispersión no muy grande alrededor de la posición inicial (el robot conoce aproximadamente el área inicial donde se encuentra).

Las líneas verdes representan las medidas realizadas por el láser. Se puede observar como coinciden perfectamente con las líneas del mapa generado anteriormente.

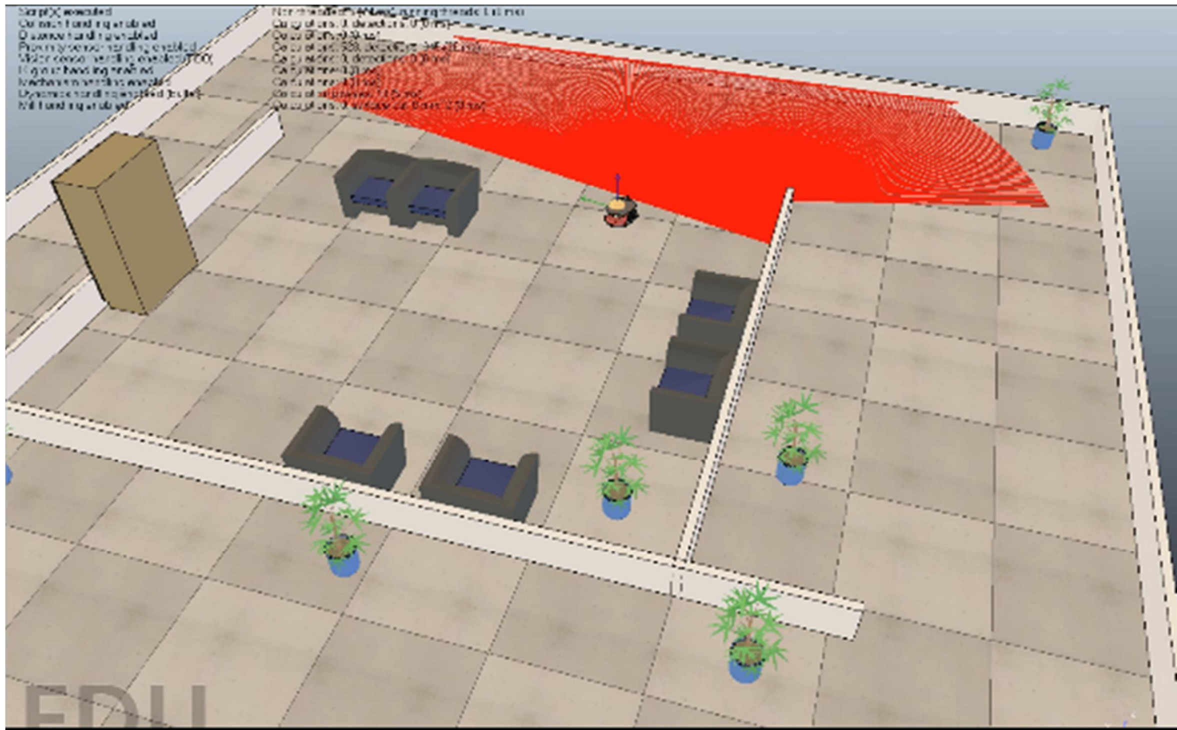


Figura 7-11 Resultados en entorno simulación VREP

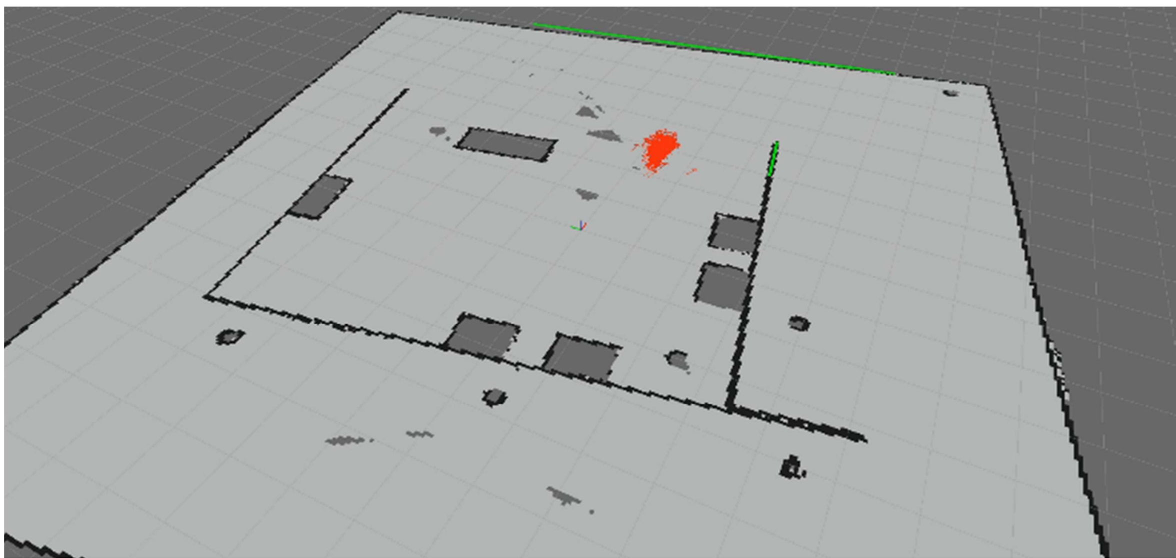


Figura 7-12 Resultados en herramienta RVIZ

Tras unos segundos de simulación se puede observar cómo la nube de partículas se va concentrando poco a poco y cómo va disminuyendo el área que abarca.

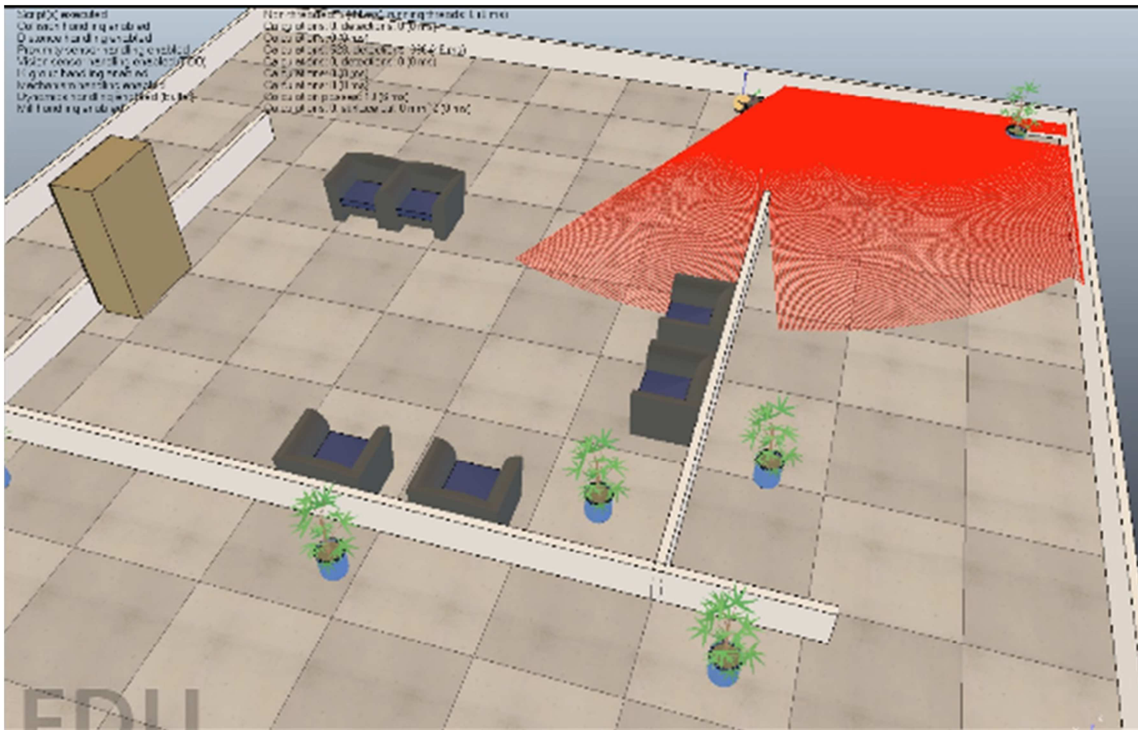


Figura 7-13 Resultado robot localizado VREP

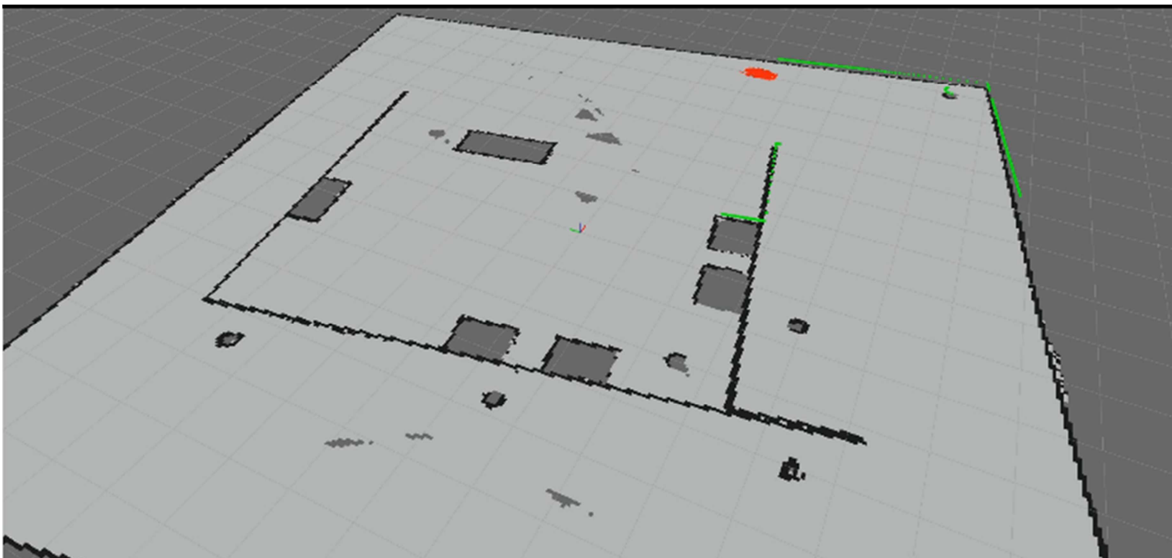


Figura 7-14 Resultado robot localizado RVIZ

Finalmente, unos segundos más tarde el robot ya se ha localizado muy satisfactoriamente. Se puede valorar este resultado comparando la posición que ocupa el robot en la **Figura 7-13** con la nube de partículas de la **Figura 7-14** Se aprecia que las partículas ocupan justo el área del robot.

A la vista de los resultados se puede decir que el algoritmo basado en el método Montecarlo ha funcionado muy bien para la localización del robot en este entorno preparado en V-REP.

8 CONCLUSIÓN

Con la realización del presente proyecto se ha alcanzado el objetivo principal de simular un proceso de localización en un problema de robótica móvil. Para ello, se ha hecho un análisis riguroso de los distintos métodos que existen para dotar a un robot de la capacidad de localizarse en un entorno dado. Una vez se ha escogido el algoritmo adecuado, se ha implementado en el simulador VREP con el objetivo de analizar su respuesta. Además se ha realizado otra prueba para intentar mejorar los resultados de la simulación. Esta consistía en el uso de un entorno externo al simulador para acelerar el tiempo de computación. El entorno externo utilizado ha sido el meta-sistema operativo ROS, una herramienta muy potente capaz de estructurar y conectar procesos a través de nodos. Analizando los resultados, se puede llegar a la conclusión de que por una parte, el algoritmo utilizado para la localización del robot funciona satisfactoriamente, y por otra parte, la combinación de ROS con el programa VREP resulta muy buena para reducir notablemente los tiempos de computación.

9 BIBLIOGRAFÍA

[1] *Beyond the Kalman Filter: Particle Filters for Tracking Applications .* (s.f.).

[2] Branko Ristic, S. A. (s.f.). *Beyond the Kalman Filter: Particle Filters for Tracking Applications.*

[3] ROS. (s.f.). Obtenido de http://www.ros.org/

[4]Sebastian Thrun, D. F. (2001). *Robust Monte Carlo localizaction for mobiles robots.*

[5] Sebastian THRUN, W. B. (1999-2000). *Probabilistic Robotics.*

[6] Tinne De Laet, J. D. (s.f.). Rigorously Bayesian Range Finder Sensor Model for Dynamic.

[7] *V-REP*. (s.f.). Obtenido de http://www.coppeliarobotics.com/

PRESUPUESTO

Contenido

1 CUADRO DE PRECIOS DESCOMPUESTO 3
2 MEDICIONES Y PRESUPUESTO 3
3 PRESUPUESTO DE EJECUCIÓN MATERIAL 4

1 CUADRO DE PRECIOS DESCOMPUESTO

CUADRO DE PRECIOS DESCOMPUESTOS.

CÓDIGO	UD.	DESCRIPCIÓN	Rdto	PRECIO	IMPORTE
1. CAP.01		ESTUDIO DE METODOS PARA LA LOCALIZACIÓN DE ROBOTS			
1.1	Ud.	Búsqueda de información			
		Estudio de las posibles formas en las que se puede implementar un algoritmo para la localización de robots			
	mes	Ingeniero Programador	0,3	2.400,00 €	720,00 €
	mes	Ingeniero Industrial especializado en automática	0,3	2.600,00 €	780,00 €
	ud	Libro Probabilistics Robotics de Sebastian THRUN, Wolfram BURGARD, Dieter FOX	1	55,68 €	55,68 €
	%	Costes directos complementarios	0,02	5.000,00 €	100,00 €
				Coste Total	1.655,68 €
2. CAP.02		PRUEBAS EN EL SIMULADOR			
2.1	mes	Elección del simulador			
		Busqueda y elección de un simulador para comprobar el algoritmo escogido			
	mes	Ingeniero Programador	0,2	2.400,00 €	480,00 €
	mes	Ingeniero Industrial especializado en automática	0,2	2.600,00 €	520,00 €
	%	Costes directos complementarios	0,02	5.000,00 €	100,00 €
				Coste Total	1.100,00 €
2.2	mes	Implementación del algoritmo elegido			
		Prueba del algoritmo escogido en el simulador y análisis de resultados			
	mes	Ingeniero Programador	0,5	2.400,00 €	1.200,00 €
	mes	Ingeniero Industrial especializado en automática	0,5	2.600,00 €	1.300,00 €
	%	Costes directos complementarios	0,02	5.000,00 €	100,00 €
				Coste Total	2.600,00 €

2 MEDICIONES Y PRESUPUESTO

MEDICIONES Y PRESUPUESTO

CÓDIGO	Ud.	DESCRIPCIÓN/ UNIDAD DE OBRA	MEDICIÓN	PRECIO	IMPORTE
CAP.01		ESTUDIO DE MÉTODOS PARA LA LOCALIZACIÓN DE ROBOTS			
1.1	Mes	Búsqueda de información	1	1.655,58 €	1.655,58 €
			Total Capitulo 01 :		1.655,58 €
CAP.02		PRUEBAS EN EL SIMULADOR			
2.1	Mes	Elección del simulador	1	1.100,00 €	1.100,00 €
2.2	Mes	Implementación del algoritmo elegido	1	2.600,00 €	2.600,00 €
			Total Capitulo 01 :		3.700,00 €

3 PRESUPUESTO DE EJECUCIÓN MATERIAL

PRESUPUESTO DE EJECUCIÓN MATERIAL

CAP.01	**ANÁLISIS PREVIO DE PRODUCTO Y PLANIFICACIÓN DEL PROYECTO**	**1.655,58 €**
CAP.02	**REDACCIÓN DE LA DOCUMENTACIÓN**	**3.700,00 €**
	TOTAL EJECUCIÓN MATERIAL :	**5.355,58 €**
	GASTOS GENERALES (13%) :	**696,23 €**
	BENEFICIO INDUSTRIAL (6%) :	**321,33 €**
	TOTAL EJECUCIÓN POR CONTRATA :	**6.373,14 €**
	IVA (21%) :	**1.338,36 €**
	PRESUPUESTO TOTAL (BASE DE LICITACIÓN):	**7.711,50 €**

ANEXOS DEL PROYECTO

Contenido

1 CÓDIGO IMPLEMENTACIÓN ALGORITMO AMCL 3
2 MANUAL FUNCIONES VREP 20

ESCUELA TÉCNICA
SUPERIOR INGENIEROS
INDUSTRIALES VALENCIA

1 CÓDIGO IMPLEMENTACIÓN ALGORITMO AMCL

```
--FUNCION RANDOMFLOAT
function randomFloat(lower, greater)
   aleatorio=0
   for i=1,12,1 do
      random= lower + math.random()*(greater - lower)
      aleatorio=aleatorio+random
   end
   return aleatorio;
end

----------------------------------

--FUNCION NORMALIZEANGLE
function normalizeAngle(alpha, center)
   alpha = (alpha-center+math.pi)%(2*math.pi) + center-math.pi
   return alpha
end

----------------------------------

--FUNCION ANGLEDIFF
function angleDiff(angle1, angle2)
   angle1 = normalizeAngle(angle1,math.pi)
   angle2 = normalizeAngle(angle2,math.pi)
   --compute difference and normalize in [-pi pi]
   dif = normalizeAngle(angle2-angle1,0)
   return dif
end

----------------------------------

--FUNCION PROBABILIDAD
function probabilidad(visionSensor1Handle, visionSensor2Handle, sensorRef,
measuredData)

   ----------------------------------
   --IMPLEMENTACION PHIT
```

```
function phit(zt, zt_real, sigma_hit, zmax)
   resultado=0
   pi=3.141592653589793
   if (zmax>=zt_real) and (zt_real>=0)then
      nu=0
      dz=0.01
      for z=0,maxScanDistance_,dz do
         nu=nu+(math.exp(((-(z-zt_real)*(z-zt_real))
         /(2*sigma_hit*sigma_hit)))
         /(math.sqrt(2*pi*sigma_hit*sigma_hit)))*dz
         --nu=nu+(math.exp(((-(zt-z)*(zt-z))/(2*sigma_hit*sigma_hit)))
         /(math.sqrt(2*pi*sigma_hit*sigma_hit)))*dz
      end
      resultado=(math.exp(((-(zt-zt_real)*(zt-zt_real))/(2*sigma_hit*sigma_hit)))
      /(math.sqrt(2*pi*sigma_hit*sigma_hit)))/nu
      --resultado=math.exp(((-(zt-zt_real)*(zt-zt_real))/(2*sigma_hit*sigma_hit)))
      /(math.sqrt(2*pi*sigma_hit*sigma_hit))
   else
      resultado=0
   end
   print('phit')
   print(resultado)
   return resultado
end
----------------------------------
--IMPLEMENTACION PSHORT
function pshort(zt, zt_real, landa_short)
   resultado=0
   if (zt_real>=zt) and (zt>=0) then
      mu=(1/(1-math.exp(-landa_short*zt_real)))
      resultado=mu*landa_short*math.exp(-landa_short*zt_real)
   else
      resultado=0
   end
   print('pshort')
   print(resultado)
   return resultado
end
----------------------------------
--IMPLEMENTACION PMAX
function pmax(zt, zmax)
   resultado=0
   if(zt==zmax) then
```

```
        resultado=1
    else
        resultado=0
    end
    print('pmax')
    print(resultado)
    return resultado
end
----------------------------------
--IMPLEMENTACION PRAND
function prand(zt, zmax)
    resultado=0
    if(zmax>zt) then
        resutado=1/zmax
    else
        resultado=0
    end
    print('prand')
    print(resultado)
    return resultado
end
----------------------------------

-- We skip the very first reading
r,t1,u1=simReadVisionSensor(visionSensor1Handle)
r,t2,u2=simReadVisionSensor(visionSensor2Handle)

--hay 66 valores en u1 y u2 y 15 en t
m1=simGetObjectMatrix(visionSensor1Handle,-1)
m01=simGetInvertedMatrix(simGetObjectMatrix(sensorRef,-1))
m01=simMultiplyMatrices(m01,m1)
m2=simGetObjectMatrix(visionSensor2Handle,-1)
m02=simGetInvertedMatrix(simGetObjectMatrix(sensorRef,-1))
m02=simMultiplyMatrices(m02,m2)
--q=1
if u1 then
q=1
    p={0,0,0}
    p=simMultiplyVector(m1,p)
    t={p[1],p[2],p[3],0,0,0}
    for j=0,u1[2]-1,1 do
        for i=0,u1[1]-1,1 do
            w=2+4*(j*u1[1]+i)
```

```
            zt_star=u1[w+4]
            zt=measuredData[i+1]
            p1=zhit*phit(zt,zt_star,sigma_hit,maxScanDistance_)
            +zshort*pshort(zt,zt_star,landa_short)
            +zmax*pmax(zt,maxScanDistance_)+zrand*prand(zt,maxScanDistance_);
            q=q*p1
         end
      end
   end
   if u2 then
      p={0,0,0}
      p=simMultiplyVector(m2,p)
      t={p[1],p[2],p[3],0,0,0}
      for j=0,u2[2]-1,1 do
         for i=0,u2[1]-1,1 do
            w=2+4*(j*u2[1]+i)
            zt_star=u2[w+4]
            zt=measuredData[i+u1[1]+1]
            p2=zhit*phit(zt,zt_star,sigma_hit,maxScanDistance_)
            +zshort*pshort(zt,zt_star,landa_short)+zmax*pmax(zt,maxScanDistance_)
            +zrand*prand(zt,maxScanDistance_);
            q=q*p2
         end
      end
   end
   qparticula=q
   return qparticula

end

----------------------------------

--FUNCION LECTURA SENSOR GENERAL
function lecturasensorgeneral(visionSensor1Handlegeneral,visionSensor2Handlegeneral
,sensorRefgeneral)

   measuredData={}
   -- We skip the very first reading
   simAddDrawingObjectItem(lines,nil)
   showLines=simGetScriptSimulationParameter(sim_handle_self,'showLaserSegments')
   rgeneral,t1general,u1general=simReadVisionSensor(visionSensor1Handlegeneral)
   rgeneral,t2general,u2general=simReadVisionSensor(visionSensor2Handlegeneral)
```

```
m1general=simGetObjectMatrix(visionSensor1Handlegeneral,-1)
m01general=simGetInvertedMatrix(simGetObjectMatrix(sensorRefgeneral,-1))
m01general=simMultiplyMatrices(m01general,m1general)
m2general=simGetObjectMatrix(visionSensor2Handlegeneral,-1)
m02general=simGetInvertedMatrix(simGetObjectMatrix(sensorRefgeneral,-1))
m02general=simMultiplyMatrices(m02general,m2general)
if u1general then
   p={0,0,0}
   p=simMultiplyVector(m1general,p)
   t={p[1],p[2],p[3],0,0,0}
   for j=0,u1general[2]-1,1 do
      for i=0,u1general[1]-1,1 do
         w=2+4*(j*u1general[1]+i)
         v1=u1general[w+1]
         v2=u1general[w+2]
         v3=u1general[w+3]
         v4=u1general[w+4]
         if (v4<maxScanDistance_) then
            table.insert(measuredData,v4)
         else
            table.insert(measuredData,maxScanDistance_)
         end
         if showLines then
            p={v1,v2,v3}
            p=simMultiplyVector(m1general,p)
            t[4]=p[1]
            t[5]=p[2]
            t[6]=p[3]
            simAddDrawingObjectItem(lines,t)
         end
      end
   end
end
if u2 then
   p={0,0,0}
   p=simMultiplyVector(m2general,p)
   t={p[1],p[2],p[3],0,0,0}
   for j=0,u2general[2]-1,1 do
      for i=0,u2general[1]-1,1 do
         w=2+4*(j*u2general[1]+i)
         v1=u2general[w+1]
         v2=u2general[w+2]
         v3=u2general[w+3]
```

```
          v4=u2general[w+4]
          if (v4<maxScanDistance_) then
            table.insert(measuredData,v4)
          else
            table.insert(measuredData,maxScanDistance_)
          end
          if showLines then
            p={v1,v2,v3}
            p=simMultiplyVector(m2,p)
            t[4]=p[1]
            t[5]=p[2]
            t[6]=p[3]
            simAddDrawingObjectItem(lines,t)
          end
        end
      end
  end

  return measuredData
end

----------------------------------

--FUNCION RESAMPLING
function resampling(q)

  ----------------------------------
  --function cumsum
  function cumsum(q1)
    x1={}
    N=#q1
    --print(N)
    x1[1]=q1[1]
    for i=1,N-1,1 do
      x1[i+1]=x1[i]+q1[i+1]
    end
    return x1
  end
  ----------------------------------
  --funcion cumprod
  function cumprod(variable)
    resultado={}
    N=#variable
```

```
    resultado[1]=variable[1]
    for i=1,N-1,1 do
        resultado[i+1]=resultado[i]*variable[i+1]
    end
    return resultado
end
----------------------------------
--funcion fliplr
function fliplr(u1)
    v1={}
    N=#u1
    for i=1,N,1 do
        v1[i]=u1[i]
    end
    --v1=u1
    for i=1,N,1 do
        u1[i]=v1[N+1-i]
    end
    return u1
end
----------------------------------

u={}
ut={}
valorcumprod={}
i={}
P={}

P=cumsum(q)
N=#q

for i=1,N,1 do
    valorcumprod[i]= math.pow(math.random(),(1/(N+1-i)))
end

u=cumprod(valorcumprod)
ut=fliplr(u)

--k=1
for j=1,N,1 do
k=1
    while (P[k]<ut[j]) do
        k=k+1
```

```
          if (k==N+1) then
             k=N
             break
          end
       end
       i[j]=k
    end

    return i
end

----------------------------------

--FUNCION INICIALIZACION
if (simGetScriptExecutionCount()==0) then

    ----------------------------------
    --Parametros Evitacion
    usensors={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}
    for i=1,16,1 do
       usensors[i]=simGetObjectHandle("Pioneer_p3dx_ultrasonicSensor"..i)
    end
    motorLeft=simGetObjectHandle("Pioneer_p3dx_leftMotor")
    motorRight=simGetObjectHandle("Pioneer_p3dx_rightMotor")
    noDetectionDist=0.5
    maxDetectionDist=0.2
    detect={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
    braitenbergL={-0.2,-0.4,-0.6,-0.8,-1,-1.2,-1.4,-1.6, 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
    braitenbergR={-1.6,-1.4,-1.2,-1,-0.8,-0.6,-0.4,-0.2, 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
    v0=2

    ----------------------------------
    --Numero de particulas
    numParticulas=120

    ----------------------------------
    --Manejadores de objetos
    leftEncoderID=simGetObjectHandle('Pioneer_p3dx_leftWheel')
    rightEncoderID=simGetObjectHandle('Pioneer_p3dx_rightWheel')
    pioneerID=simGetObjectHandle('Pioneer_p3dx')
    childScriptHandle=simGetScriptHandle('MCL')
    --odomID=simGetObjectHandle('odom')
    --base_linkID=simGetObjectHandle('base_link')
```

```
--laserID=simGetObjectHandle('base_laser_link')

--Manejador Sensor Particulas
handle={}
visionSensor1Handle={}
visionSensor2Handle={}
joint1Handle={}
joint2Handle={}
sensorRef={}
for i=1,numParticulas,1 do
   handle[i]=simGetObjectHandle('Particula_#' .. i)
   sensorRef[i]=simGetObjectHandle("fastHokuyo_ref" .. '#' .. i)
   joint1Handle[i]=simGetObjectHandle("fastHokuyo_joint1" .. '#' .. i)
   joint2Handle[i]=simGetObjectHandle("fastHokuyo_joint2" .. '#' .. i)
   visionSensor1Handle[i]=simGetObjectHandle("fastHokuyo_sensor1" .. '#' .. i)
   visionSensor2Handle[i]=simGetObjectHandle("fastHokuyo_sensor2" .. '#' .. i)
end

--Manejador Sensor General
sensorRefgeneral=simGetObjectHandle("fastHokuyo_ref")
joint1Handlegeneral=simGetObjectHandle("fastHokuyo_joint1")
joint2Handlegeneral=simGetObjectHandle("fastHokuyo_joint2")
visionSensor1Handlegeneral=simGetObjectHandle("fastHokuyo_sensor1")
visionSensor2Handlegeneral=simGetObjectHandle("fastHokuyo_sensor2")

----------------------------------
--Orientacion incial ruedas
oldAngleL=simGetObjectOrientation(leftEncoderID,pioneerID)
oldAngleR=simGetObjectOrientation(rightEncoderID,pioneerID)

----------------------------------
--Parametros del robot
b2=0.3310/2 -- wheel base
r2=0.195/2     -- wheel radius

--Posicion inicial
ybarra=0
xbarra=0
ybarra_prima=0
xbarra_prima=0
tectabarra=0

--Pesos nuevas particulas
```

```
alfa1=0.2
alfa2=0.2
alfa3=0.2
alfa4=0.2

----------------------------------
x={}
y={}
tecta={}
--xaleatorio={}
--yaleatorio={}
posicioninicial={}
anguloinicial={}
nuevaposicion={}
nuevoangulo={}
angulo_euler_tectaprima={}
data={}
data1={}
data2={}
datosiniciales1={}
math.randomseed(os.time())
x_prima={}
y_prima={}
qparticula={}

nuevasparticulas={}
qparticulavectornormalizado={}
qparticulavector={}
--handle1={}
--handle2={}
--handle3={}
--handle4={}
handle_new={}
name={}
suffix={}
m1={}
m01={}
m2={}
m02={}
L={}
t1={}
u1={}
u2={}
```

```
t2={}
posicionresampleada={}
anguloresampleado={}
orientacionresampleada={}
anguloresampleado1={}
orientacionresampleada3={}
vectornulo={}
measuredData={}

--Contador iteraciones
cnt=0

----------------------------------
--Posicion inicial
x1=0 --X position in meters
y1=0 --Y position in meters
ulaseranterior={0,0,0}

----------------------------------
--Inicializacion particulas
for i=1,numParticulas,1 do
   --xaleatorio[i]=randomFloat(-0.25,0.25)
   --yaleatorio[i]=randomFloat(-0.25,0.25)
   --x[i]=xaleatorio[i]
   --y[i]=yaleatorio[i]
   --tecta[i]=randomFloat(-0.25,0.25)
   x[i]=randomFloat(-0.25,0.25)
   y[i]=randomFloat(-0.25,0.25)
   tecta[i]=randomFloat(-0.25,0.25)
   --posicioninicial[i]={xaleatorio[i],yaleatorio[i],0.1388}
   posicioninicial[i]={x[i],y[i],0.1388}
   anguloinicial[i]={0,0,tecta[i]}
   --handle[i]=simGetObjectHandle('Particula_#' .. i)
   simSetObjectPosition(handle[i],-1,posicioninicial[i])
   simSetObjectOrientation(handle[i],-1,anguloinicial[i])
end

----------------------------------
--Parametros Ponderacion Particulas

sigma_hit=0.2
landa_short=0.1
```

```
    zhit=0.95
    zshort=0.1
    zrand=0.05
    zmax=0.05
--[[
    zhit=0.5
    zshort=0.05
    zrand=0.5
    zmax=0.05
]]--
    ----------------------------------
    --Sensor Particulas

    for h=1,numParticulas,1 do
        visionSensor1Handle[h]=simGetObjectHandle("fastHokuyo_sensor1" .. '#' .. h)
        visionSensor2Handle[h]=simGetObjectHandle("fastHokuyo_sensor2" .. '#' .. h)
        joint1Handle[h]=simGetObjectHandle("fastHokuyo_joint1" .. '#' .. h)
        joint2Handle[h]=simGetObjectHandle("fastHokuyo_joint2" .. '#' .. h)
        sensorRef[h]=simGetObjectHandle("fastHokuyo_ref" .. '#' .. h)

        --maxScanDistance=simGetScriptSimulationParameter(sim_handle_self,
        'maxScanDistance')
        maxScanDistance=5
        if maxScanDistance>1000 then maxScanDistance=1000 end
        if maxScanDistance<0.1 then maxScanDistance=0.1 end
        simSetObjectFloatParameter(visionSensor1Handle[h],1001,maxScanDistance)
        simSetObjectFloatParameter(visionSensor2Handle[h],1001,maxScanDistance)
        maxScanDistance_=maxScanDistance*0.9999

        --scanningAngle=simGetScriptSimulationParameter(sim_handle_self,'scanAngle')
        scanningAngle=240
        if scanningAngle>240 then scanningAngle=240 end
        if scanningAngle<2 then scanningAngle=2 end
        scanningAngle=scanningAngle*math.pi/180
        simSetObjectFloatParameter(visionSensor1Handle[h],1004,scanningAngle/2)
        simSetObjectFloatParameter(visionSensor2Handle[h],1004,scanningAngle/2)

        simSetJointPosition(joint1Handle[h],-scanningAngle/4)
        simSetJointPosition(joint2Handle[h],scanningAngle/4)
    end

    ----------------------------------
    --Sensor General
```

```
    visionSensor1Handlegeneral=simGetObjectHandle("fastHokuyo_sensor1")
    visionSensor2Handlegeneral=simGetObjectHandle("fastHokuyo_sensor2")
    joint1Handlegeneral=simGetObjectHandle("fastHokuyo_joint1")
    joint2Handlegeneral=simGetObjectHandle("fastHokuyo_joint2")
    sensorRefgeneral=simGetObjectHandle("fastHokuyo_ref")

    --maxScanDistance=simGetScriptSimulationParameter(sim_handle_self
    ,'maxScanDistance')
    maxScanDistance=5
    if maxScanDistance>1000 then maxScanDistance=1000 end
    if maxScanDistance<0.1 then maxScanDistance=0.1 end
    simSetObjectFloatParameter(visionSensor1Handlegeneral,1001,maxScanDistance)
    simSetObjectFloatParameter(visionSensor2Handlegeneral,1001,maxScanDistance)
    maxScanDistance_=maxScanDistance*0.9999

    --scanningAngle=simGetScriptSimulationParameter(sim_handle_self,'scanAngle')
    scanningAngle=240
    if scanningAngle>240 then scanningAngle=240 end
    if scanningAngle<2 then scanningAngle=2 end
    scanningAngle=scanningAngle*math.pi/180
    simSetObjectFloatParameter(visionSensor1Handlegeneral,1004,scanningAngle/2)
    simSetObjectFloatParameter(visionSensor2Handlegeneral,1004,scanningAngle/2)

    simSetJointPosition(joint1Handlegeneral,-scanningAngle/4)
    simSetJointPosition(joint2Handlegeneral,scanningAngle/4)

    ----------------------------------
    red={1,0,0}
    lines=simAddDrawingObject(sim_drawing_lines,1,0,-1,10000,nil,nil,nil,red)

    ----------------------------------
    if (simGetIntegerParameter(sim_intparam_program_version)<30004) then
      simDisplayDialog("ERROR","This version of the Hokuyo sensor is only supported from V-REP V3.0.4 and upwards.&&nMake sure to update your V-REP.",sim_dlgstyle_ok,false,nil,{0.8,0,0,0,0,0},{0.5,0,0,1,1,1})
    end

end

----------------------------------

--FUNCION PERIODICA
```

```
if (simGetScriptExecutionCount()~=0) then

    simHandleChildScript(sim_handle_all_except_explicit)

    ----------------------------------
    --Evitacion Obstaculos
    for i=1,16,1 do
        res,dist=simReadProximitySensor(usensors[i])
        if (res>0) and (dist<noDetectionDist) then
            if (dist<maxDetectionDist) then
                dist=maxDetectionDist
            end
            detect[i]=1-((dist-maxDetectionDist)/(noDetectionDist-maxDetectionDist))
        else
            detect[i]=0
        end
    end

    vLeft=v0
    vRight=v0

    for i=1,16,1 do
        vLeft=vLeft+braitenbergL[i]*detect[i]
        vRight=vRight+braitenbergR[i]*detect[i]
    end

    simSetJointTargetVelocity(motorLeft,vLeft)
    simSetJointTargetVelocity(motorRight,vRight)

    ----------------------------------
    --Modelo Odometria Ruedas

    handleWheel=simGetObjectHandle('Pioneer_p3dx')
    tecta_prima=simGetObjectOrientation(handleWheel,-1)
    --tecta_prima=simGetObjectOrientation(pioneerID,-1)

    angleL=simGetObjectOrientation(leftEncoderID,pioneerID)
    phiL=angleDiff(oldAngleL[3],angleL[3])
    angleR=simGetObjectOrientation(rightEncoderID,pioneerID)
    phiR=angleDiff(oldAngleR[3],angleR[3])

    deltaS=r2*(phiL+phiR)/2
    deltaTheta=r2*(phiR-phiL)/(2*b2)
```

```
xbarra_prima=xbarra+deltaS*math.cos(tectabarra)
ybarra_prima=ybarra+deltaS*math.sin(tectabarra)
tectabarra_prima=tectabarra+deltaTheta
--print('hhh',tectabarra_prima)
--print(xbarra_prima,ybarra_prima)
oldAngleL[3]=angleL[3]
oldAngleR[3]=angleR[3]

----------------------------------
--Modelo Odometria Montecarlo

for i=1,numParticulas,1 do
   --handle3[i]=simGetObjectHandle('Particula_#' .. i)
   --posicionresampleada[i]=simGetObjectPosition(handle3[i],-1)
   --orientacionresampleada3[i]=simGetObjectOrientation(handle3[i],-1)
   handle[i]=simGetObjectHandle('Particula_#' .. i)
   posicionresampleada[i]=simGetObjectPosition(handle[i],-1)
   orientacionresampleada3[i]=simGetObjectOrientation(handle[i],-1)

   --x[i]=posicionresampleada[i][1]
   --y[i]=posicionresampleada[i][2]
   --tecta[i]=orientacionresampleada3[i][3]

   delta_rot1=math.atan2(ybarra_prima-ybarra,xbarra_prima-xbarra)
   -tectabarra
   delta_trans=math.sqrt((xbarra-xbarra_prima)*(xbarra-xbarra_prima)
   +(ybarra-ybarra_prima)*(ybarra-ybarra_prima))
   delta_rot2=tectabarra_prima-tectabarra-delta_rot1

   b11=alfa1*delta_rot1+alfa2*delta_trans
   b12=alfa1*delta_rot2+alfa2*delta_trans
   b2=alfa3*delta_trans+alfa4*(delta_rot1+delta_rot2)

   delta_rot1_hat=delta_rot1-randomFloat(-b11,b11)
   delta_rot2_hat=delta_rot2-randomFloat(-b12,b12)
   delta_trans_hat=delta_trans-randomFloat(-b2,b2)

   x_prima[i]=x[i]+delta_trans_hat*math.cos(tecta[i]+delta_rot1_hat)
   y_prima[i]=y[i]+delta_trans_hat*math.sin(tecta[i]+delta_rot1_hat)
   tecta_prima[i]=tecta[i]+delta_rot1_hat+delta_rot2_hat
   angulo_euler_tectaprima[i]={0,0,tecta_prima[i]}
```

```
        nuevaposicion[i]={x_prima[i],y_prima[i],0.1388}
        nuevoangulo[i]=angulo_euler_tectaprima[i]

        --simSetObjectPosition(handle3[i],-1,nuevaposicion[i])
        --simSetObjectOrientation(handle3[i],-1,nuevoangulo[i])
        handle[i]=simGetObjectHandle('Particula_#' .. i)
        simSetObjectPosition(handle[i],-1,nuevaposicion[i])
        simSetObjectOrientation(handle[i],-1,nuevoangulo[i])

        x[i]=x_prima[i]
        y[i]=y_prima[i]
        tecta[i]=tecta_prima[i]
        --print(x[i],y[i],tecta[i])
    end

    ybarra=ybarra_prima
    xbarra=xbarra_prima
    tectabarra=tectabarra_prima
    --print(tectabarra)

    ----------------------------------
    --Lectura sensor robot
    measuredData=lecturasensorgeneral(visionSensor1Handlegeneral,
    visionSensor2Handlegeneral,sensorRefgeneral)

    ----------------------------------

    print('*************')
    print(cnt)
    print('*************')
    cnt=cnt+1;

--[[
    --Particulas normalizadas (automatico)
    suma=0
    for i=1,numParticulas,1 do
        qparticulavector[i]=probabilidad(visionSensor1Handle[i],
        visionSensor2Handle[i],sensorRef[i],measuredData)
        print(qparticulavector[i])
        suma=suma+qparticulavector[i]
    end

    for i=1,numParticulas,1 do
```

```
        qparticulavectornormalizado[i]=qparticulavector[i]/suma
        --print(qparticulavectornormalizado[i])
    end
]]--

    --Particulas normalizadas (manual)
    for i=1,numParticulas,1 do
        qparticulavectornormalizado[i]=1/numParticulas
    end

    ----------------------------------
    --Remuestreo particulas
    nuevasparticulas=resampling(qparticulavectornormalizado)

    for i=1,numParticulas,1 do
        --print(nuevasparticulas[i])
    end

    ----------------------------------

    --Actualizacion particulas
    for i=1,numParticulas,1 do

        handle_new[i]=simGetObjectHandle('Particula_#' .. nuevasparticulas[i])
        posicionresampleada[i]=simGetObjectPosition(handle_new[i],-1)
        anguloresampleado[i]=simGetObjectOrientation(handle_new[i],-1)
        anguloresampleado1[i]={0,0,anguloresampleado[i][3]}
        --handle2[i]=simGetObjectHandle('Particula_#' .. i)
        --simSetObjectPosition(handle2[i],-1,posicionresampleada[i])
        --simSetObjectOrientation(handle2[i],-1,anguloresampleado1[i])
        handle[i]=simGetObjectHandle('Particula_#' .. i)
        simSetObjectPosition(handle[i],-1,posicionresampleada[i])
        simSetObjectOrientation(handle[i],-1,anguloresampleado1[i])

        --x[i]=posicionresampleada[i][1]
        --y[i]=posicionresampleada[i][2]
        --tecta[i]=anguloresampleado[i][3]
        --print(x[i],y[i],tecta[i])
    end

end
```

2 MANUAL FUNCIONES VREP

simReadVisionSensor
remote API equivalent: simxReadVisionSensor
ROS API equivalent: simRosReadVisionSensor

Description	Reads the state of a vision sensor. This function doesn't perform detection, it merely reads the result from a previous call to simHandleVisionSensor (simHandleVisionSensor is called in the default main script). See also simCheckVisionSensor, simCheckVisionSensorEx and simResetVisionSensor.
C synopsis	simInt simReadVisionSensor(simInt visionSensorHandle,simFloat** auxValues,simInt** auxValuesCount)
C parameters	**visionSensorHandle**: handle of a vision sensor object **auxValues**: auxiliary values returned from the applied filters. By default V-REP returns one packet of 15 auxiliary values:the minimum of {intensity, red, green, blue, depth value}, the maximum of {intensity, red, green, blue, depth value}, and the average of {intensity, red, green, blue, depth value}. If additional filter components return values, then they will be appended as packets to the first packet. AuxValues can be NULL. The user is in charge of releasing the auxValues buffer with simReleaseBuffer(*auxValues). **auxValuesCount**: contains information about the number of auxiliary value packets and packet sizes returned in auxValues. The first value is the number of packets, the second is the size of packet1, the third is the size of packet2, etc. Can be NULL if auxValues is also NULL. The user is in charge of releasing the auxValuesCount buffer with simReleaseBuffer(*auxValuesCount). See simHandleVisionSensor for a usage example
C return value	detection state (0 or 1), or -1 in case of an error, or if simHandleVisionSensor was never called, or if simResetVisionSensor was previously called.
Lua synopsis	number result,table auxiliaryValuePacket1,table auxiliaryValuePacket2, etc.=simReadVisionSensor(number visionSensorHandle)
Lua parameters	**visionSensorHandle**: handle of a vision sensor object
Lua return values	**result**: detection state (0 or 1), or -1 in case of an error, or if simHandleVisionSensor was never called, or if simResetVisionSensor was previously called. **auxiliaryValuePacket1**: default auxiliary value packet (same as for the C-function) **auxiliaryValuePacket2**: additional auxiliary value packet (e.g. from a filter component) **auxiliaryValuePacket3**: etc. (the function returns as many tables as there are auxiliary value packets)

simGetObjectMatrix

Description	Retrieves the transformation matrix of an object. See also simSetObjectMatrix, simGetObjectPosition and simGetObjectOrientation.
C synopsis	simInt simGetObjectMatrix(simInt objectHandle,simInt relativeToObjectHandle,simFloat* matrix)
C parameters	**objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame we want the matrix. Specify -1 to retrieve the absolute transformation matrix, sim_handle_parent to retrieve the transformation matrix relative to the object's parent, or an object handle relative to whose reference frame we want the transformation matrix. **matrix**: pointer to 12 simFloat values (the last row of the 4x4 matrix (0,0,0,1) is not needed) The x-axis of the orientation component is (matrix[0],matrix[4],matrix[8]) The y-axis of the orientation component is (matrix[1],matrix[5],matrix[9]) The z-axis of the orientation component is (matrix[2],matrix[6],matrix[10]) The translation component is (matrix[3],matrix[7],matrix[11])
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	table_12 matrix=simGetObjectMatrix(number objectHandle,number relativeToObjectHandle)
Lua parameters	**objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame we want the matrix. Specify -1 to retrieve the absolute transformation matrix, sim_handle_parent to retrieve the transformation matrix relative to the object's parent, or an object handle relative to whose reference frame we want the transformation matrix.
Lua return values	**matrix**: table of 12 numbers (the last row of the 4x4 matrix (0,0,0,1) is not returned), or nil in case of an error. Table values in Lua are indexed from 1, not 0!

simGetInvertedMatrix

Description	Returns the inverse of a transformation matrix
C synopsis	See simInvertMatrix for the C-equivalent function
C parameters	-
C return value	-
Lua synopsis	table_12 invertedMatrix=simGetInvertedMatrix(table_12 matrix)
Lua parameters	**matrix**: table to 12 numbers (the last row of the 4x4 matrix (0,0,0,1) is not needed) The x-axis of the orientation component is (matrix[1],matrix[5],matrix[9]) The y-axis of the orientation component is (matrix[2],matrix[6],matrix[10]) The z-axis of the orientation component is (matrix[3],matrix[7],matrix[11]) The position component is (matrix[4],matrix[8],matrix[12])
Lua return values	**invertedMatrix**: inverse of the input matrix, or nil in case of an error

simMultiplyMatrices

Description	Multiplies two transformation matrices
C synopsis	simInt simMultiplyMatrices(const simFloat* matrixIn1,const simFloat* matrixIn2,simFloat* matrixOut)
C parameters	**matrixIn1**: the first input matrix **matrixIn2**: the second input matrix **matrixOut**: the output matrix (the result of the multiplication: matrixIn1*matrixIn2). A transformation matrix contains 12 values (the last row (0,0,0,1) is omitted): The x-axis of the orientation component is (matrix[0],matrix[4],matrix[8]) The y-axis of the orientation component is (matrix[1],matrix[5],matrix[9]) The z-axis of the orientation component is (matrix[2],matrix[6],matrix[10]) The position component is (matrix[3],matrix[7],matrix[11])
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	table_12 resultMatrix=simMultiplyMatrices(table_12 matrixIn1,table_12 matrixIn2)
Lua parameters	Same as C-function
Lua return values	**resultMatrix**: the result matrix (a table containing 12 values (the last row (0,0,0,1) is omitted)). Table values in Lua are indexed from 1, not 0!

simMultiplyVector

Description	Multiplies a vector with a transformation matrix (v=m*v). See simTransformVector for the C-API version.
C synopsis	-
C parameters	-
C return value	-
Lua synopsis	table_3 resultVector=simMultiplyVector(table_12 matrix,table_3 vector)
Lua parameters	**matrix**: the transformation matrix (a table containing 12 values (the last row (0,0,0,1) is not required)) **vector**: the original vector (a table containing 3 values (the last element (1) of the homogeneous coordinates is not required)
Lua return values	**resultVector**: the result vector (a table containing 3 values (the last element (1) of the homogeneous coordinates is omitted))

simAddDrawingObjectItem

Description	Adds an item (or clears all items) to a previously inserted drawing object. See also simAddDrawingObject and simRemoveDrawingObject
C synopsis	simInt simAddDrawingObjectItem(simInt objectHandle,const simFloat* itemData)
C parameters	**objectHandle**: handle of a previously added drawing object **itemData**: data relative to an item. If the item is a point item, 3 values are required (x;y;z). If the item is a line item, 6 values are required, and if the item is a triangle item, 9 values are required. Additional values (auxiliary values) might be required depending on the drawing object attributes. See the drawing object types and attributes for more information. If NULL the drawing object is emptied of all its items
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	number result=simAddDrawingObjectItem(number drawingObjectHandle,table itemData)
Lua parameters	Same as C-function
Lua return values	Same as C-function

simGetScriptExecutionCount

Description	Retrieves the number of times the current script was called and returned. Useful to set-up initial values when the function returns 0 (the script was called for the first time in the current simulation). Child scripts should always use this function instead of testing whether simGetSimulationState()==sim_simulation_advancing_firstafterstop (a child script might be created (through copy/paste operation for instance) in the middle of a simulation, in which case simGetSimulationState() will not return sim_simulation_advancing_firstafterstop). When a threaded child script executes and finishes several times in a same simulation run, then this function always returns 0.
C synopsis	-
C parameters	-
C return value	-
Lua synopsis	number executionCount=simGetScriptExecutionCount()
Lua parameters	None
Lua return values	**executionCount**: number of times the current script was called and returned, or -1 in case of an error

simGetObjectHandle
remote API equivalent: simxGetObjectHandle
ROS API equivalent: simRosGetObjectHandle

Description	Retrieves an object handle based on its name. The operation of this function depends on the current name suffix settings (see simGetNameSuffix, simSetNameSuffix, and the section on accessing general-type objects). See also simIsHandleValid and simGetObjectUniqueIdentifier.
C synopsis	simInt simGetObjectHandle(const simChar* objectName)
C parameters	**objectName**: name of object
C return value	handle of object or -1 if operation was not successful
Lua synopsis	number objectHandle=simGetObjectHandle(string objectName)
Lua parameters	Same as C-function
Lua return values	Same as C-function

simGetScriptName

Description	Retrieves a script's name based on its handle. A script doesn't have a name assigned, however if the script is a child script and associated with a scene object, then this function will retrieve the name of the associated scene object. If the script is not a child script or is not associated with a scene object, then the returned value is NULL
C synopsis	simChar* simGetScriptName(simInt scriptHandle)
C parameters	**scriptHandle**: handle of the script
C return value	buffer to the script's name if function was successful and the name is valid, NULL otherwise. The user is in charge of releasing the returned buffer with simReleaseBuffer
Lua synopsis	string scriptName=simGetScriptName(number scriptHandle)
Lua parameters	**scriptHandle**: handle of the script, or sim_handle_self for the handle of the current script
Lua return values	**scriptName**: name of the script if associated with a scene object, empty string if the script is the main script, or the name of the add-on if the script is an add-on.

simGetObjectOrientation
remote API equivalent: simxGetObjectOrientation
ROS API equivalent: simRosGetObjectPose

Description	Retrieves the orientation (Euler angles) of an object. See also simGetObjectQuaternion, simSetObjectOrientation, simGetObjectPosition and simGetObjectMatrix.
C synopsis	simInt simGetObjectOrientation(simInt objectHandle,simInt relativeToObjectHandle,simFloat* eulerAngles)
C parameters	**objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame we want the orientation. Specify -1 to retrieve the absolute orientation, sim_handle_parent to retrieve the orientation relative to the object's parent, or an object handle relative to whose reference frame you want the orientation. **eulerAngles**: Euler angles (alpha, beta and gamma)
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	table_3 eulerAngles=simGetObjectOrientation(number objectHandle,number relativeToObjectHandle)
Lua parameters	Same as C-function
Lua return values	**eulerAngles**: table of 3 values (Euler angles) or nil in case of an error

simSetObjectPosition
remote API equivalent: simxSetObjectPosition
ROS API equivalent: simRosSetObjectPosition

Description	Sets the position (x, y and z-coordinates) of an object. Dynamically simulated objects will implicitely be reset before the command is applied (i.e. similar to calling simResetDynamicObject just before). See also simGetObjectPosition, simSetObjectOrientation and simSetObjectMatrix.
C synopsis	simInt simSetObjectPosition(simInt objectHandle,simInt relativeToObjectHandle,const simFloat* position)
C parameters	**objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame the position is specified. Specify -1 to set the absolute position, sim_handle_parent to set the position relative to the object's parent, or an object handle relative to whose reference frame the position is specified. **position**: coordinates of the object (x, y and z)
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	number result=simSetObjectPosition(number objectHandle,number relativeToObjectHandle,table_3 position)
Lua parameters	Same as C-function
Lua return values	Same as C-function

simGetScriptSimulationParameter

Description	Retrieves a script's parameter from its simulation parameter list. Useful for simple interaction with the user, or for simple parameter exchange with other scripts. Only parameters from main or child scripts can be retrieved with this function. See also simSetScriptSimulationParameter, simUnpackInts and simUnpackFloats.
C synopsis	simChar* simGetScriptSimulationParameter(simInt scriptHandle,const simChar* parameterName,simInt* parameterLength)
C parameters	**scriptHandle**: handle of the script, or sim_handle_main_script or sim_handle_all. When scriptHandle is sim_handle_all, the function returns only one matching parameter encountered (other matching parameters might be different) **parameterName**: name of the parameter to retrieve **parameterLength**: the number of bytes that compose the value of the parameter (excluding the terminal zero)
C return value	value of the parameter or NULL if parameterName does not exist for the given script, or in case of an error. The user is in charge of releasing the returned value with simReleaseBuffer. The returned pointer points to parameterLength byte values, terminated by a terminal zero (the returned buffer may however contain several embedded zeros).
Lua synopsis	(1) boolean/number/string parameterValue=simGetScriptSimulationParameter(number scriptHandle,string parameterName,boolean forceStringReturn=false) 2) table parameterValues,table scriptHandles=simGetScriptSimulationParameter(number targetScripts,string parameterName,boolean forceStringReturn=false)
Lua parameters	(1) **scriptHandle**: handle of the script, or sim_handle_main_script or sim_handle_self. (2) **targetScripts**: sim_handle_all, sim_handle_tree or sim_handle_chain (with sim_handle_tree and sim_handle_chain the calling script is excluded). **parameterName**: name of the parameter to retrieve. **forceStringReturn**: forces the return of a string (i.e. raw value). False by default. If false, then the returned string will be converted to nil, false, true, a number or a string as appropriate (and in that order).
Lua return values	(1) **parameterValue**: value of the parameter, or nil in case of an error (or if that value is nil!). (2) **parameterValues**: table of parameter values or nil if no such parameter was found or in case of an error. **scriptHandles**: table of script handles associated with the parameter values (i.e. parameterValue[i] comes from the script with handle scriptHandles[i]) or nil if no such parameter was found or in case of an error. If the returned parameter value is a string, then it might contain any values (also embedded zeros)

simSetObjectPosition
remote API equivalent: simxSetObjectPosition
ROS API equivalent: simRosSetObjectPosition

Description	Sets the position (x, y and z-coordinates) of an object. Dynamically simulated objects will implicitely be reset before the command is applied (i.e. similar to calling simResetDynamicObject just before). See also simGetObjectPosition, simSetObjectOrientation and simSetObjectMatrix.
C synopsis	simInt simSetObjectPosition(simInt objectHandle,simInt relativeToObjectHandle,const simFloat* position)
C parameters	**objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame the position is specified. Specify -1 to set the absolute position, sim_handle_parent to set the position relative to the object's parent, or an object handle relative to whose reference frame the position is specified. **position**: coordinates of the object (x, y and z)
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	number result=simSetObjectPosition(number objectHandle,number relativeToObjectHandle,table_3 position)
Lua parameters	Same as C-function
Lua return values	Same as C-function

simSetObjectOrientation
remote API equivalent: simxSetObjectOrientation
ROS API equivalent: simRosSetObjectQuaternion

Description	Sets the orientation (Euler angles) of an object. Dynamically simulated objects will implicitely be reset before the command is applied (i.e. similar to calling simResetDynamicObject just before). See also simSetObjectQuaternion, simGetObjectOrientation, simSetObjectPosition and simSetObjectMatrix.
C synopsis	simInt simSetObjectOrientation(simInt objectHandle,simInt relativeToObjectHandle,const simFloat* eulerAngles)
C parameters	**objectHandle**: handle of the object **relativeToObjectHandle**: indicates relative to which reference frame the orientation is specified. Specify -1 to set the absolute orientation, sim_handle_parent to set the orientation relative to the object's parent, or an object handle relative to whose reference frame the orientation is specified. **eulerAngles**: Euler angles (alpha, beta and gamma)
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	number result=simSetObjectOrientation(number objectHandle,number relativeToObjectHandle,table_3 eulerAngles)
Lua parameters	Same as C-function
Lua return values	Same as C-function

simSetObjectFloatParameter
remote API equivalent: simxSetObjectFloatParameter
ROS API equivalent: simRosSetObjectFloatParameter

Description	Sets a floating-point parameter of a scene object or calculation object. See also simGetObjectFloatParameter, simSetObjectIntParameter and simSetObjectStringParameter
C synopsis	simInt simSetObjectFloatParameter(simInt objectHandle,simInt parameterID,simFloat parameter)
C parameters	**objectHandle**: handle of the object **parameterID**: identifier of the parameter to retrieve. See the list of all possible object parameter identifiers **parameter**: parameter value
C return value	-1 in case of an error, 0 if the parameter could not be set (e.g. because the parameterID doesn't exist, or because the specified object doesn't correspond to the correct type), or 1 if the operation was successful
Lua synopsis	number result=simSetObjectFloatParameter(number objectHandle,number parameterID,number parameter)
Lua parameters	Same as C-function
Lua return values	Same as C-function

simSetJointPosition
remote API equivalent: simxSetJointPosition
ROS API equivalent: simRosSetJointPosition

Description	Sets the intrinsic position of a joint. May have no effect depending on the joint mode. This function cannot be used with spherical joints (use simSetSphericalJointMatrix instead). See also simGetJointPosition and simSetJointTargetPosition.
C synopsis	simInt simSetJointPosition(simInt objectHandle,simFloat position)
C parameters	**objectHandle**: handle of the joint object **position**: position of the joint (angular or linear value depending on the joint type)
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	number result=simSetJointPosition(number objectHandle,number position)
Lua parameters	Same as C-function
Lua return values	Same as C-function

simGetIntegerParameter
remote API equivalent: simxGetIntegerParameter
ROS API equivalent: simRosGetIntegerParameter

Description	Retrieves an integer value. See the integer parameter identifiers. See also simSetIntegerParameter, simGetBooleanParameter, simGetFloatingParameter, simGetArrayParameter and simGetStringParameter.
C synopsis	simInt simGetIntegerParameter(simInt parameter,simInt* intState)
C parameters	**parameter**: integer parameter identifier **intState**: value of the parameter
C return value	-1 if operation was not successful. In a future release, a more differentiated return value might be available
Lua synopsis	number parameterValue=simGetIntegerParameter(number parameter)
Lua parameters	**parameter**: parameter identifier (sim_intparam_...)
Lua return values	**parameterValue**: value of the parameter or nil in case of an error

simHandleChildScript

Description	Calls one or several child scripts. Not available from the C-API. Only child scripts that are built onto the current script hierarchy will be called (that's why they are called "child scripts"!). Child scripts of a child script cannot be directly called, it is the responsibility of each child script to forward that call. For more details, see the description on how scripts are executed in V-REP. See also the simHandleSensingChildScripts function. This function is not available to add-ons.
C synopsis	-
C parameters	-
C return value	-
Lua synopsis	(1) number executedScriptCount=simHandleChildScript(sim_handle_all/sim_handle_all_except_explicit, <arguments>) (2) <return values>=simHandleChildScript(number childScriptHandle,<arguments>)
Lua parameters	(1) **Argument1**: sim_handle_all if all childscripts should be executed, also the ones marked as "explicit handling", or sim_handle_all_except_explicit if all childscripts should be executed, except the ones marked as "explicit handling". **Argument2**, etc.: <arguments> can be any arguments that you want to pass to the child scripts that will be called. Inside the called child scripts, the <arguments> can be retrieved with the "arguments=â€¦" command. <arguments> can be omitted when calling simHandleChildScript. (2) **Argument1**: childScriptHandle: handle of the childscript to be executed. **Argument2**, etc.: <arguments> can be any arguments that you want to pass to the child script that will be called. Inside the called child script, the <arguments> can be retrieved with the "arguments=â€¦" command. <arguments> can be omitted when calling simHandleChildScript.
Lua return values	(1) executedScriptCount: number of executed child scripts (including the called child scripts that are still in execution (e.g. if the childscript is marked as "threaded execution") (2) <returned values>: the values returned by the child script that was called. A child script can return values with following command: "return argument1, argument2, etc.". Values returned by a child script marked as "threaded execution" will be ignored (i.e. nil is returned instead)

simReadProximitySensor
remote API equivalent: simxReadProximitySensor
ROS API equivalent: simRosReadProximitySensor

Description	Reads the state of a proximity sensor. This function doesn't perform detection, it merely reads the result from a previous call to simHandleProximitySensor (simHandleProximitySensor is called in the default main script). See also simCheckProximitySensor, simCheckProximitySensorEx and simResetProximitySensor.
C synopsis	simInt simReadProximitySensor(simInt sensorHandle,simFloat* detectedPoint,simInt* detectedObjectHandle,simFloat* detectedSurfaceNormalVector)
C parameters	**sensorHandle**: handle of a proximity sensor object **detectedPoint**: coordinates of the closest detected point (x, y and z: detectedPoint[0]-detectedPoint[2]) relative to the sensor reference frame, and distance to the detected point (1 value: detectedPoint[3]). Can be NULL **detectedObjectHandle**: handle of the object that was detected. Can be NULL **detectedSurfaceNormalVector**: normal vector (normalized) of the detected surface. Relative to the sensor reference frame. Can be NULL
C return value	detection state (0 or 1), or -1 in case of an error, or if simHandleProximitySensor was never called, or if simResetProximitySensor was previously called.
Lua synopsis	number result,number distance,table_3 detectedPoint,number detectedObjectHandle,table_3 detectedSurfaceNormalVector=simReadProximitySensor(number sensorHandle)
Lua parameters	Same as C-function
Lua return values	**result**: detection state (0 or 1), or -1 in case of an error, or if simHandleProximitySensor was never called, or if simResetProximitySensor was previously called. **distance**: distance to the detected point if result is 1, nil otherwise **detectedPoint**: table of 3 numbers indicating the relative coordinates of the detected point if result is 1, nil otherwise **detectedObjectHandle**: handle of the object that was detected if result is 1, nil otherwise **detectedSurfaceNormalVector**: normal vector (normalized) of the detected surface. Relative to the sensor reference frame. Is nil if result is different from 1

simRosEnableSubscriber

Description	Enables a subscriber (i.e. V-REP will be listening to data streaming on a specific topic). Subscribers can only be enabled while simulation is running. Subscribers can be disabled with simRosDisableSubscriber. At simulation end, all subscribers are automatically disabled. A subscriber can also be enabled using the ROS plugin's exported Lua function: simExtROS_enableSubscriber. See also simRosEnablePublisher.
Input	**topicName (string)**: the desired topic name. **queueSize (int32)**: the desired queue size. **streamCmd (int32)**: the desired type of data to stream. **auxInt1 (int32)**: an auxiliary integer value (first) that might be needed to fully specify the desired data to stream **auxInt2 (int32)**: an auxiliary integer value (second) that might be needed to fully specify the desired data to stream **auxString (string)**: an auxiliary string value that might be needed to fully specify the desired data to stream
Output	**subscriberID (int32)**: a subscriber ID, or -1 in case of an error. The subscriber ID is needed to disable that subscriber with simRosDisableSubscriber.

simRosEnablePublisher

Description	Enables a publisher (i.e. V-REP will be streaming data on a specific topic). Publishers can only be enabled while simulation is running. If a same publisher (i.e. of the same type and applied to the same item) was already enabled previously, then a reference counter will be incremented and the topic name of that publisher is returned. Publishers can be disabled with simRosDisablePublisher. At simulation end, all publishers are automatically disabled. A publisher can also be enabled using the ROS plugin's exported Lua function: simExtROS_enablePublisher. See also simRosEnableSubscriber.
Input	**topicName (string)**: the desired topic name. **queueSize (int32)**: the desired queue size. **streamCmd (int32)**: the desired type of data to stream. **auxInt1 (int32)**: an auxiliary integer value (first) that might be needed to fully specify the desired data to stream **auxInt2 (int32)**: an auxiliary integer value (second) that might be needed to fully specify the desired data to stream **auxString (string)**: an auxiliary string value that might be needed to fully specify the desired data to stream
Output	**effectiveTopicName (string)**: the effective topic name that will be used to stream the desired data, or an empty string if there was an error. If the desired topic name is already in use for another type of publisher, then a new topic name is generated (i.e. "_2" appended). If the desired publisher is already enabled, then the topic name of that existing publisher is returned.