



**Universidad  
Rey Juan Carlos**

**Grado en Ingeniería en Telemática**

Escuela Técnica Superior de Ingeniería Telecomunicación

Curso académico 2016/2017

**Trabajo Fin de Grado**

Servidor de mapas dinámico para la navegación en entornos  
domésticos

**Autor:** Jonathan Ginés Clavero **Tutor:** Francisco Martín Rico



*A*



# Agradecimientos



# Resumen



# Índice general

<b>1. Introducción</b>	<b>1</b>
<b>2. Objetivos y Metodología</b>	<b>2</b>
2.1. Descripción del problema y requisitos . . . . .	2
2.2. Objetivo del proyecto . . . . .	2
2.3. Metodología de desarrollo . . . . .	3
2.4. Plan de trabajo . . . . .	5
<b>3. Servidor de mapas dinámico</b>	<b>6</b>
3.1. Costmap_2D . . . . .	6
3.2. Tipos de mapas . . . . .	8
3.2.1. Mapa estático . . . . .	8
3.2.2. Mapa de corto plazo . . . . .	8
3.2.3. Mapa de largo plazo . . . . .	12
3.3. Construcción del mapa final . . . . .	15
<b>4. Aplicaciones</b>	<b>17</b>
4.1. AMCL . . . . .	17

# Índice de figuras

2.1. Modelo en espiral . . . . .	4
3.1. Ejemplo visual de un costmap . . . . .	6
3.2. A la izquierda el frame map y a la derecha los frames de odom y base_footprint . . . . .	7
3.3. Mapa estático . . . . .	8
3.4. Visión del simulador, (a) y (b), y mapa a corto plazo (c) . . . . .	10
3.5. Añadimos un objeto al escenario . . . . .	10
3.6. Eliminamos un objeto del escenario . . . . .	11
3.7. Añadimos un objeto al mapa de largo plazo . . . . .	13
3.8. Borramos un objeto del mapa de largo plazo . . . . .	14
4.1. Inicialización del filtro de partículas . . . . .	18
4.2. El robot se va acercando a la posición ideal . . . . .	18
4.3. El robot se encuentra totalmente localizado . . . . .	19

# Capítulo 1

## Introducción

# **Capítulo 2**

## **Objetivos y Metodología**

### **2.1. Descripción del problema y requisitos**

La navegación en entornos dinámicos, como puede ser una casa o una institución pública, supone un gran reto que superar ya que las personas o las mascotas pueden acercarse o cruzarse delante del robot o podríamos encontrarnos objetos del mobiliario que han sido movidos de su posición original y se encuentran en nuestro camino. En este escenario el robot no debería nunca ni chocar ni perderse en el entorno y debe llegar al destino impuesto por la mejor ruta disponible.

Por ejemplo, si nuestro robot está lleno desde el salón a la cocina de nuestra casa pero en su camino habitual y más óptimo se encuentra un mueble, el robot debe darse cuenta rápidamente, esquivarlo, proseguir con su camino y además recordarlo para que cuando volvamos al salón de vuelta podamos esquivarlo más fácilmente. Por otro lado en nuestra casa también habrá personas, estas personas se moverán casi constantemente por la estancia por lo que no será del todo correcto tenerlas en cuenta a la hora de planificar nuestra ruta para navegar de un sitio a otro de la casa pero si que será muy importante no chocar con ellas.

Para analizar el entorno del robot usaremos el sensor laser, este sensor destaca por su alta precisión y su corto tiempo de procesado.

### **2.2. Objetivo del proyecto**

Se quiere diseñar un algoritmo que genere un mapa en tiempo real, el cual será usado por el nodo de navegación de ROS para navegar por un entorno doméstico, ya sea indicando una posición x,y en el mapa o indicándole una estancia a la que navegar.

Este mapa se construirá a partir de la mezcla de 3 mapas, mapa estático, mapa de largo plazo y mapa de corto plazo.

En primera instancia el algoritmo se validará haciendo uso de un simulador en el que se representa una casa con varios tipos de muebles, ya que resulta más fácil de depurar un algoritmo en un entorno virtual, que en un entorno real. Posteriormente el algoritmo se probará en distintas recreaciones de escenarios reales, y se harán las modificaciones oportunas para adaptarlo al entorno real, y por último se llevará a la competición.

Para simplificar la resolución del problema se ha dividido el proyecto en varios subobjetivos:

1. Se usarán las herramientas por defecto que nos ofrece ROS para crear el mapa de corto plazo en referencia a las mediciones tomadas por el laser. En un primer paso solo añadiremos los diferentes objetos que percibamos.
2. Se ampliará el algoritmo anterior para poder añadir y eliminar objetos que aparezcan o desaparezcan del entorno.
3. Se desarrollará un algoritmo para mezclar los mapas entre sí y así generar tanto el mapa de largo plazo como el mapa que usaremos para la navegación.
4. Se creará un servidor de mapas dinámicos que se inicializará con los mapas estático y de largo plazo y que generará el mapa final mezclando los mapas de largo plazo y de corto plazo.
5. Se usará el mapa final como parámetro del paquete de navegación de ROS, *move base*, y del paquete de localización de ROS *amcl*.
6. se generará y usará un mapa semántico, en el que cada nivel de gris se asocie con una etiqueta para después ordenar al robot que navegue a dichas etiquetas.

## 2.3. Metodología de desarrollo

En el desarrollo del sistema descrito, el modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos. Este modelo permite desarrollar el proyecto de forma incremental, aumentando la complejidad progresivamente y haciendo posible la generación de prototipos funcionales. Este planteamiento permite obtener productos parciales al final de cada ciclo que pueden ser evaluados, ya sea total o parcialmente. Esto facilita la adaptación a los cambios en los requisitos, circunstancia muy común en

los proyectos de investigación.

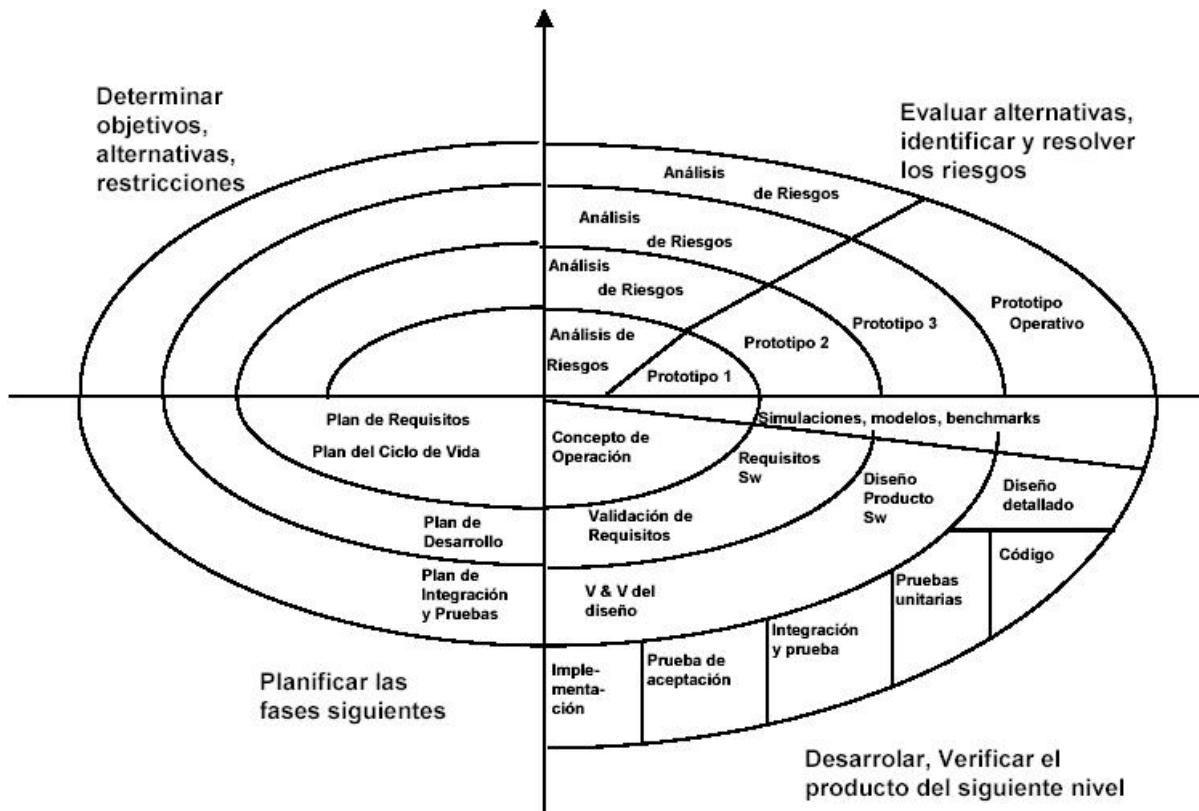


Figura 2.1: Modelo en espiral.

El ciclo de vida de este modelo está dividido en ciclos. Cada ciclo representa una fase del proyecto y está dividido, a su vez, en 4 partes. Cada una de las partes tiene un objetivo distinto:

- **Determinar objetivos.** Se establecen las necesidades que debe cumplir el sistema en cada iteración teniendo en cuenta los objetivos finales. Según avanzan las iteraciones aumenta el coste del ciclo y su complejidad.
- **Evaluar alternativas.** Se determinan diferentes formas de alcanzar los objetivos que se han establecido en la fase anterior. Se aborda el problema desde distintos puntos de vista, como, por ejemplo, el rendimiento del algoritmo en tiempo y espacio. Además, se consideran explícitamente los riesgos, intentando mitigarlos al máximo.

- **Desarrollar y verificar.** Se desarrolla el producto siguiendo la mejor alternativa para poder alcanzar los objetivos del ciclo. Una vez diseñado e implementado el producto, se realizan las pruebas necesarias para comprobar su funcionamiento.
- **Planificar.** Teniendo en cuenta los resultados de las pruebas realizadas, se planifica la siguiente iteración, se revisan los errores cometidos y se comienza un nuevo ciclo de la espiral.

## 2.4. Plan de trabajo

Para poder abordar el problema se han marcado una serie de subobjetivos ha completar. Dichos hitos son los siguientes:

1. Estudio y comprensión de la composición de un mapa y como construirlo. Nos apoyaremos en las herramientas ofrecidas por ROS y que resultaran básicas para este fin, dichas herramientas son *TF*<sup>1</sup> y *Costmap*<sup>2</sup>.
2. Primer subobjetivo. Una vez conocido como funcionan los *costmap* procederemos a crear un pequeño nodo en el que se cree un mapa con las observaciones instantáneas que percibimos con el laser.
3. Segundo subobjetivo. Extender el algoritmo anterior para poder añadir y eliminar objetos según entren o salgan de la escena.
4. Tercer subobjetivo. Modificar el paquete *map\_server* para que acepte varios mapas como entrada y estudiar la manera de mezclar los mapas entre sí.
5. Fase de pruebas. Se le pasará al paquete de navegación de ROS, *move\_base*, el mapa resultante y se harán pruebas de navegación en el simulador y en el robot real.
6. Cuarto subobjetivo. Crear el mapa semántico, especificando las etiquetas naturales que tendrá una casa, salón, cocina, habitación... y usarlo para poder ir a la estancia que le indiquemos.

---

<sup>1</sup><http://wiki.ros.org/tf>

<sup>2</sup>[http://wiki.ros.org/costmap\\_2D](http://wiki.ros.org/costmap_2D)

# Capítulo 3

## Servidor de mapas dinámico

En este capítulo se explica la construcción y el funcionamiento del servidor de mapas dinámico. En primer lugar se explica que es, para que se utiliza y como se construye un *costmap*. En segundo lugar se explica como se construyen los mapas que componen el algoritmo y por último se explica como se combinan estos mapas para generar el mapa usado en la navegación.

### 3.1. Costmap\_2D

Un *costmap* es una estructura de datos ofrecida por ROS y compuesta por un grid de ocupación y los metadatos de este grid. Cada celda del grid toma valores entre 0 y 255, donde 0 corresponde a una celda vacía, los valores entre 1 y 254 representan la probabilidad de que una celda está ocupada y el valor 255 se reserva para el desconocimiento. Cada valor se asocia con un nivel de gris, como se puede ver en la imagen.

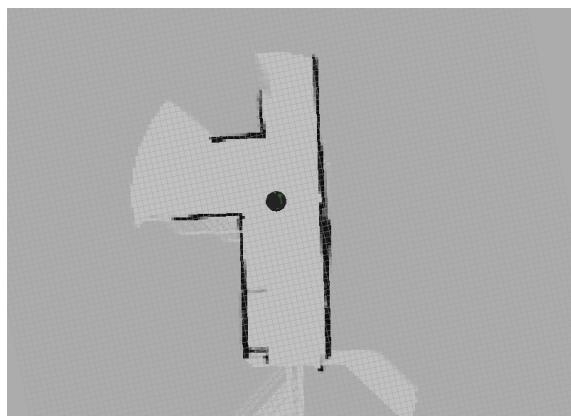


Figura 3.1: Ejemplo visual de un costmap.

Para poder representar la ocupación de un objeto en un *costmap* es necesario hacer uso de las transformadas entre frames que nos ofrece ROS.

**tf**

Cualquier robot está compuesto por multitud de piezas móviles, como puede ser la propia base del robot o la pinza de un brazo robótico. Cada una de estas piezas se pueden representar con un *frame*. Además existen también otros *frames* que pueden interesarnos, como puede ser el *frame* de world o el *frame* de map.

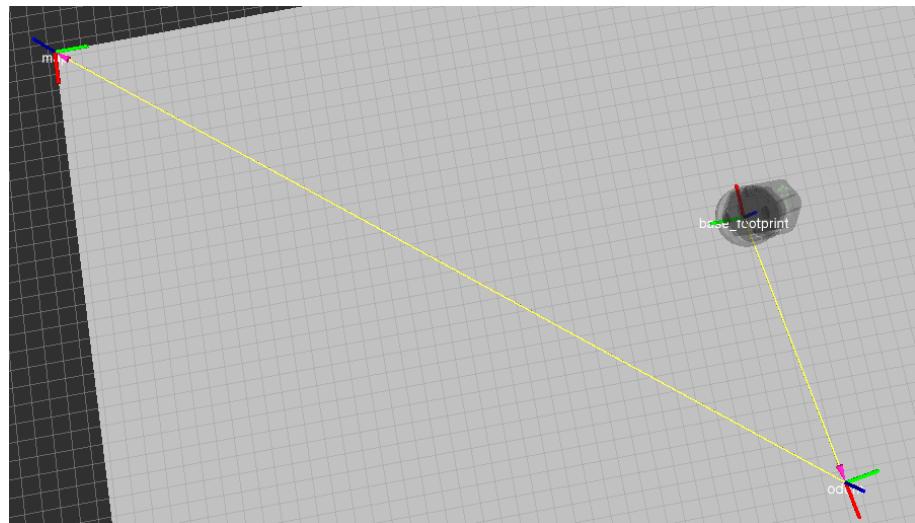


Figura 3.2: A la izquierda el frame map y a la derecha los frames de odom y base\_footprint

Usamos las *tf* para poder representar información relativa a uno de estos frames. Esto puede sernos de utilidad, por ejemplo, si queremos conocer la posición de un objeto que hemos cogido con nuestra pinza respecto a la base de nuestro robot, o cuál es la posición relativa de un objeto que estamos percibiendo con el laser respecto a nosotros o respecto al mapa.

Cuando trabajamos con mapas es importante que todo lo que se representa en él sea respecto al frame map. De este modo nuestro mapa puede ser usado por otros nodos, como el nodo de navegación, o en cualquier otro escenario.

## 3.2. Tipos de mapas

En este apartado se describirá la metodología seguida para la construcción de cada uno de los tres mapas usados por el algoritmo.

### 3.2.1. Mapa estático

El mapa estático se caracteriza por incluir las partes inmutables del escenario, como son las paredes o las puertas. La mejor manera de construirlo es medir todo el escenario y crear el mapa usando una herramienta de diseño gráfico. En este caso se ha usado *Gimp*. Este mapa nos servirá como base para crear el mapa de largo plazo.

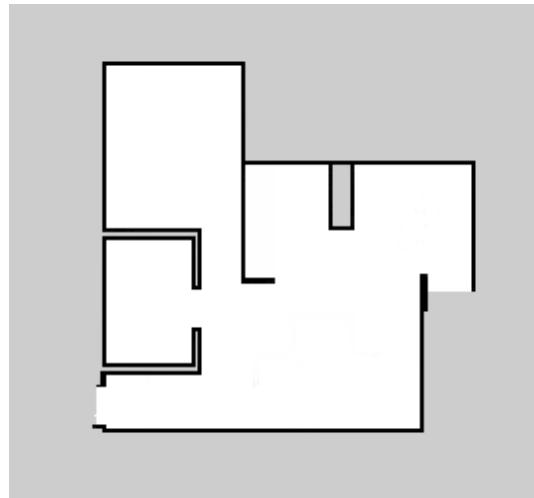


Figura 3.3: Mapa estático

### 3.2.2. Mapa de corto plazo

El mapa de corto plazo se caracteriza por ser un mapa en el que se representa los objetos que el robot va percibiendo. Este mapa se inicializa con el valor 255, lo que indica una incertidumbre total. En el instante en el que el algoritmo de construcción del mapa comienza a iterar comenzarán a corregirse estos valores iniciales, asignando el valor 0 a las celdas que corresponden con zonas libres e incrementando desde 0 hasta 254 el valor de las celdas que se perciben como ocupadas.

Código 3.1: Inicialización del cost\_map correspondiente al mapa de corto plazo

---

```

metadata = getMetadata();
tf::TransformListener tf(ros::Duration(10));
cells_size_x = metadata.width;
cells_size_y = metadata.height;
resolution = metadata.resolution;
origin_x = metadata.origin.position.x;
origin_y = metadata.origin.position.y;
default_value = 255;
scan_ready = false;
pos_ready = false;
cost_map.resizeMap(cells_size_x,cells_size_y, resolution, origin_x,
origin_y);
cost_map.setDefaultValue(default_value);
cost_map.resetMap(0,0,cost_map.getSizeInCellsX(),
cost_map.getSizeInCellsY());

```

---

El algoritmo propuesto destaca por la capacidad de no solo añadir objetos al mapa, si no ademas eliminarlos si los objetos desaparecen del lugar que ocupaban. Para ello se compara cada muestra de datos con el mapa que estamos generando y si en dicha muestra existen celdas libres que en el mapa están ocupadas se decrementa el valor de dicha celda en el mapa. La cuantia del decremento se puede modelar, consiguiendo asi que el robot olvide más lentamente o más rápidamente los objetos que desaparecen del escenario.

Código 3.2: Función que actualiza el mapa en cada iteración

---

```

void
ObstacleDetector::updateCostmap(){
    if(scan_ready && pos_ready){
        incrementCostProcedure();
        decrementCostProcedure();
    }
    pVectorList_.clear();
    pointList_.clear();
}

```

---

La figura 3.4 fué captada al iniciar el algoritmo. Observamos que la mayor parte del mapa de corto plazo se encuentra en una posición de desconocimiento y que se han ido incluyendo en este las zonas libres, las paredes y la estantería. Los puntos morados y verdes corresponden a la representación de las muestras tomadas por el laser.

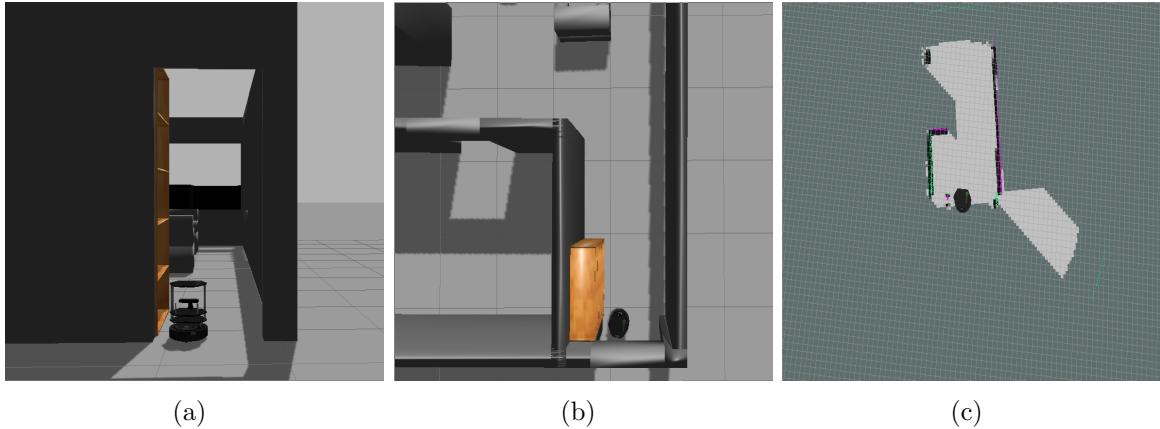


Figura 3.4: Visión del simulador, (a) y (b), y mapa a corto plazo (c).

Tras el inicio del algoritmo se añadió un objeto nuevo al escenario. Esto se representa en la figura 3.5. Vemos como el algoritmo añade el objeto al mapa y lo sitúa en una posición coherente respecto a la posición que ocupa el objeto en el escenario simulado.

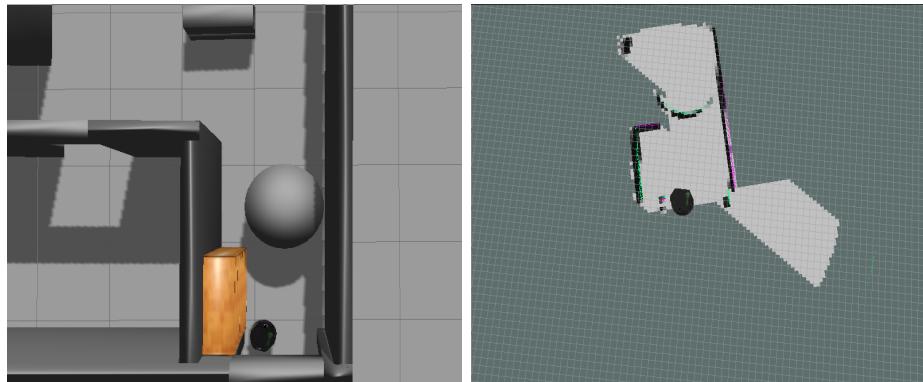


Figura 3.5: Añadimos un objeto al escenario

Una vez que el algoritmo ha incluido el objeto en el mapa procedemos a eliminarlo del escenario simulado. Esto se representa en la figura 3.6. Vemos como el algoritmo ha comenzado a borrar el objeto, por lo que el valor de las celdas que estaban ocupadas por el objeto ahora es mucho menor.

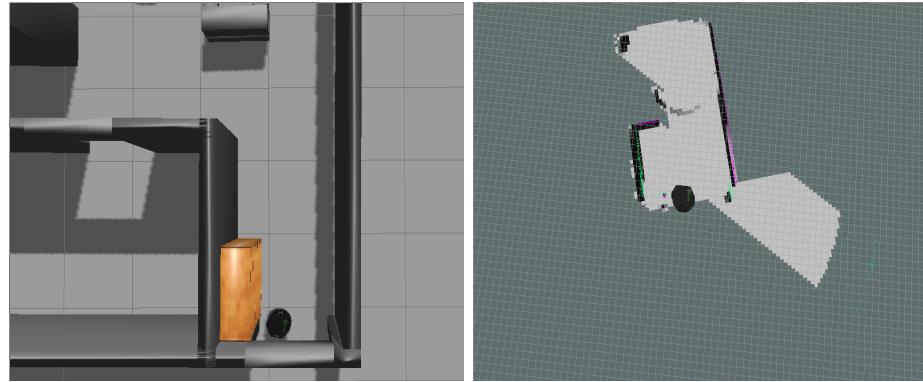


Figura 3.6: Eliminamos un objeto del escenario

El proceso de añadir o eliminar un objeto puede ser más o menos rápido dependiendo de los valores de las variables que modelan estas operaciones. Para el caso de nuestro algoritmo contamos un fichero con extensión yaml en el que modificamos los valores de los parámetros.

Código 3.3: Fichero de configuración `obstacle_detector.yaml`

---

```
cost_inc: 4
cost_dec: 1
min_length: 0.23
max_length: 2.5
```

---

Los valores asociados a `cost_inc` y `cost_dec` representan el incremento o decremento del valor de una celda por cada iteración del algoritmo. Los otros valores configurables representan la longitud máxima y mínima que se tiene en cuenta para cada medida que el laser nos proporciona. Las muestras obtenidas del laser que estén fuera de estos valores son ignoradas.

Explicar como se borran los objetos? (decremento)

Al final de cada iteración del algoritmo el mapa de corto plazo es publicado en un topic para que pueda ser usado por el resto de nodos que conforman el servidor de mapas dinámico. Para llevar a cabo esta operación usamos una estructura de datos proporcionada por ROS, `Costmap2DPublisher`<sup>1</sup>. Este publicador publica nuestro mapa en el topic `costmap_auto`.

---

<sup>1</sup>[http://docs.ros.org/indigo/api/costmap\\_2d/html/classcostmap\\_2d\\_1\\_1Costmap2DPublisher.html](http://docs.ros.org/indigo/api/costmap_2d/html/classcostmap_2d_1_1Costmap2DPublisher.html)

Código 3.4: Step del nodo obstacle\_detector

---

```

void
ObstacleDetector::step(){
    updateCostmap();
    cost_map_publisher_.publishCostmap();
}

int
main(int argc, char** argv)
{
    ros::init(argc, argv, "obstacle_detector"); //Inicializa el nodo
    ObstacleDetector obs;
    ros::Rate loop_rate(5);

    while (ros::ok()){
        obs.step();
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}

```

---

Existe un pequeño cambio en los valores del mapa que se publica. El tipo de mensaje que se usa para publicar un costmap es *nav\_msgs::OccupancyGrid*<sup>2</sup>. Este tipo de mensaje contiene, ademas de los metadatos asociados al mapa, un array de datos que representa al mapa. La principal diferencia con un costmap es que los valores en un OccupancyGrid van de 0 a 100 y -1 para las celdas desconocidas,y no de 0 a 255 como en un costmap.

### 3.2.3. Mapa de largo plazo

El mapa de largo plazo se inicializa con los valores del mapa estático y se caracteriza por incluir los objetos que tienen un valor muy alto en el mapa de corto plazo, por tanto tenemos un mapa con objetos que han perdurado en el mapa a corto plazo durante un largo periodo de tiempo y que podemos considerar que nos van a influir a la hora de planificar nuestra ruta por el escenario. Este mapa tambien es dinámico, por lo que tambien elimina los objetos del mapa si estos desaparecen o cambian su posición.

---

<sup>2</sup>[http://docs.ros.org/indigo/api/nav\\_msgs/html/msg/OccupancyGrid.html](http://docs.ros.org/indigo/api/nav_msgs/html/msg/OccupancyGrid.html)

Código 3.5: Procedimiento para añadir un objeto al mapa de largo plazo

---

```

void
MapController::updateLongTermMap(nav_msgs::OccupancyGrid s_map){
    for(int i = 0;i<s_map.data.size();i++){
        if(s_map.data[i] > 95){
            longTerm_map.data[i] = s_map.data[i];
            .
            .
            .
        }
    }
}

```

---

En el código mostrado en 3.5 muestra el proceso de adición de un objeto al mapa de largo plazo. Para ello se recorren los valores del mapa de corto plazo y si alguno tiene un valor mayor a 95, valor que consideramos lo suficientemente alto como para que sea muy fiable que esa celda está ocupada, se incluye con ese valor al mapa.

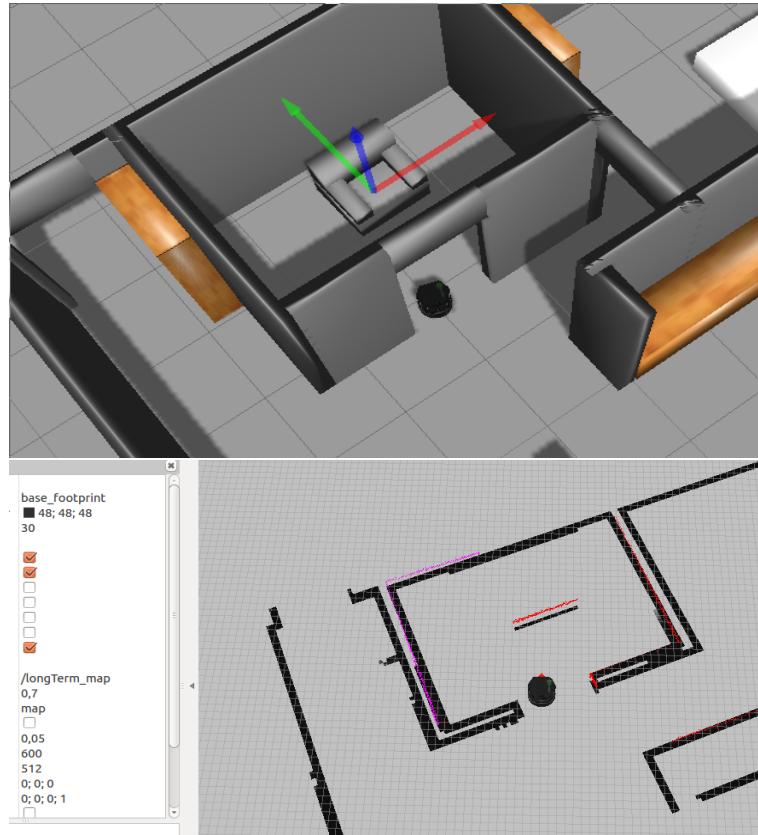


Figura 3.7: Añadimos un objeto al mapa de largo plazo

Código 3.6: Procedimiento para eliminar un objeto al mapa de largo plazo

---

```

void
MapController::updateLongTermMap(nav_msgs::OccupancyGrid s_map){
    for(int i = 0;i<s_map.data.size();i++){
        .
        .
        .

    }else if(s_map.data[i] < 5 && s_map.data[i] >= 0 &&
        longTerm_map.data[i] >= longterm_cost_dec && static_map.data[i] <
        95){
        longTerm_map.data[i] = longTerm_map.data[i] - longterm_cost_dec;
    }
}
}
}

```

---

En el código mostrado en 3.6 se compara el mapa de largo plazo con el mapa de corto plazo. Si una celda en el mapa de largo plazo tiene un valor que indica que está ocupada y en el mapa de corto plazo esa misma celda tiene un valor que indica que está libre, siempre y cuando esa celda no pertenezca a una celda de una pared, se decrementa su valor en el mapa de largo plazo. La cuantía de este decremento también puede modelarse, `longterm_cost_dec`, regulando así la memoria que tenemos de los objetos que hemos añadido a este mapa.

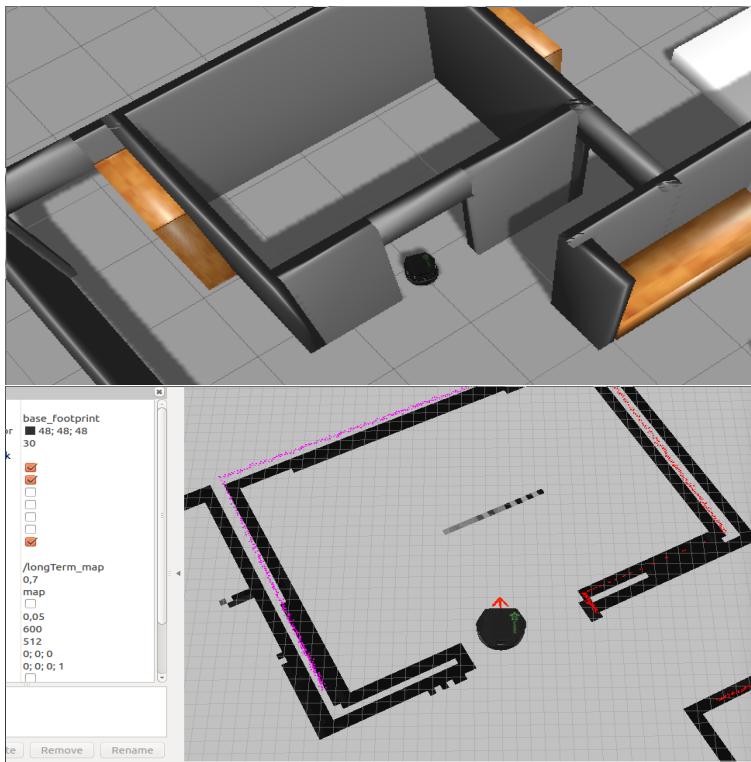


Figura 3.8: Borramos un objeto del mapa de largo plazo

Por último publicamos el mapa en el topic `/longTerm_map`.

Código 3.7: Publicación del mapa de largo plazo

---

```
longTermMap_pub = nh_.advertise<nav_msgs::OccupancyGrid>("/longTerm_map",
    5);

void
MapController::publishAll(){
    .
    .
    .
    longTerm_map.header.stamp = ros::Time::now();
    longTermMap_pub.publish(longTerm_map);
}
```

---

Adicionalmente este mapa se guarda cada cierto tiempo en un fichero, usando el nodo por defecto de ROS para este fin, `map_saver`<sup>3</sup>. Así por ejemplo podemos tener en cuenta una mesa dentro del salón de una casa para una futura ejecución del algoritmo. Esto nos permitiría planificar una ruta mejor para la navegación, ya que podríamos esquivar esta mesa con facilidad y llegar a nuestro destino lo más rápido posible. Si no tuvieramos esta mesa en cuenta el algoritmo podría planificar una ruta a través de las celdas ocupadas por la mesa. Seguramente el robot no llegaría a chocar, ya que el planificador de rutas usa un pequeño mapa local para evitar estos problemas, pero es seguro que tardaría más en alcanzar su destino ya que al encontrarse frente a la mesa el robot se pararía y tendría que recalcular una nueva ruta.

### 3.3. Construcción del mapa final

El mapa final será la composición de los mapas de largo plazo, que ya incluye el mapa estático, y de corto plazo. Este mapa será usado por el nodo de la navegación y por el nodo de la localización para navegar y localizar al robot en el escenario. La composición del mapa final o mapa efectivo será el resultado de la operación de máximo entre los mapas de corto y de largo plazo, como vemos en el código del apartado 3.8. De esta manera incluiremos en el mapa final toda la información relacionada con los objetos que llevan un tiempo en la escena y también incluiremos, aunque con un menor valor, personas o cosas que acaban de entrar en las inmediaciones del robot.

---

<sup>3</sup>[http://wiki.ros.org/action/fullsearch/map\\_server#map\\_saver](http://wiki.ros.org/action/fullsearch/map_server#map_saver)

Código 3.8: Composición del mapa final

---

```

void
MapController::buildEffectiveMap(nav_msgs::OccupancyGrid
    s_map,nav_msgs::OccupancyGrid l_map){
for(int i = 0;i<s_map.data.size();i++){
    effective_map.data[i] = std::max(s_map.data[i],l_map.data[i]);
}
}

```

---

Tras la creación del mapa este es publicado en el topic */map*.

Código 3.9: Publicación del mapa final

---

```

effectiveMap_pub = nh_.advertise<nav_msgs::OccupancyGrid>("/map", 5);

void
MapController::publishAll(){
    effective_map.header.stamp = ros::Time::now();
    effectiveMap_pub.publish(effective_map);
    .
    .
    .
}

```

---

Este topic es usado por defecto por las aplicaciones que necesitan un mapa como parámetro de entrada. En nuestro caso las aplicaciones que usarán nuestro mapa son el nodo de localización y el nodo de navegación, como veremos en el siguiente capítulo.

# Capítulo 4

## Aplicaciones

En este capítulo se enumeran y describen las aplicaciones que usarán el mapa que construimos dinámicamente. Estas aplicaciones serán, principalmente, el nodo de localización y el nodo de navegación.

### 4.1. AMCL

*AMCL*<sup>1</sup> es un paquete de localización que implementa un algoritmo de Monte Carlo, el cual usa un filtro de partículas para localizar al robot sobre un mapa que previamente le proporcionamos.

#### Filtro de partículas

El algoritmo del filtro de partículas se divide en 4 etapas: Inicialización, actualización, estimación y predicción. En la etapa de inicialización se “lanzan” una serie de partículas cercanas a la posición inicial del robot. Estas partículas ademas de una posición en el espacio tambien tendrán una dirección. Podemos ver un ejemplo gráfico en la imagen 4.1

Vemos como se han generado muchas partículas alrededor del robot y que cada una tiene una dirección más o menos acertada con la dirección del robot.

En este punto del algoritmo se compara la percepción del laser del robot en el punto en el que se encuentra con la percepción que tendría si tuviera la posición y la dirección de cada una de las partículas que generamos. Cuanto más acertada sea la suposición anterior, más valor se le dá a esa partícula. Así nos encontraremos que las partículas que están más cercanas a la posición del robot cobran más valor y las que están más lejos y con una dirección totalmente errónea tienen menos valor. Esta fase es la llamada de Actualización.

---

<sup>1</sup><http://wiki.ros.org/amcl>

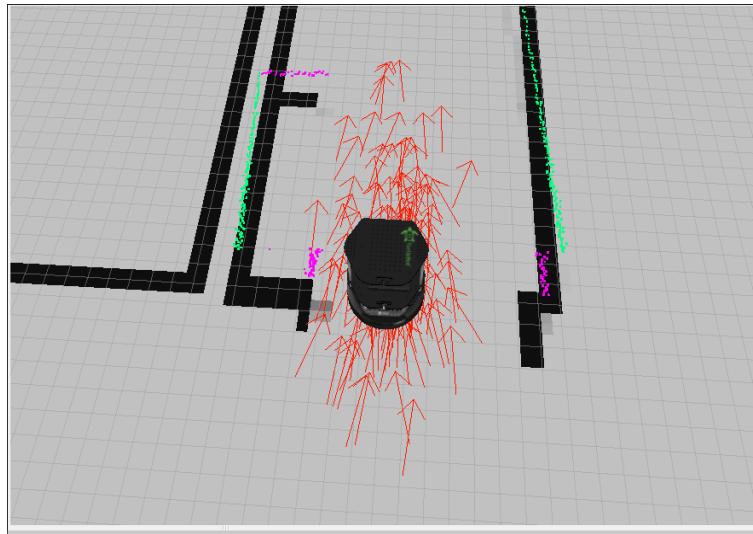


Figura 4.1: Inicialización del filtro de partículas

En la siguiente fase, llamada de Estimación, nos quedamos con las partículas que más valor tenían para volver a lanzarlas en la siguiente fase del algoritmo.

En la última fase, fase de Precicción, lanzamos las partículas de nuevo con el valor que tenían y su posición, añadiéndole un pequeño ruido. En este punto del algoritmo también se corrige la posición del robot a la posición de la partícula con más valor. Una vez completado el algoritmo se vuelve a la fase de Actualización y se repite hasta que el robot esté perfectamente localizado en el mapa.

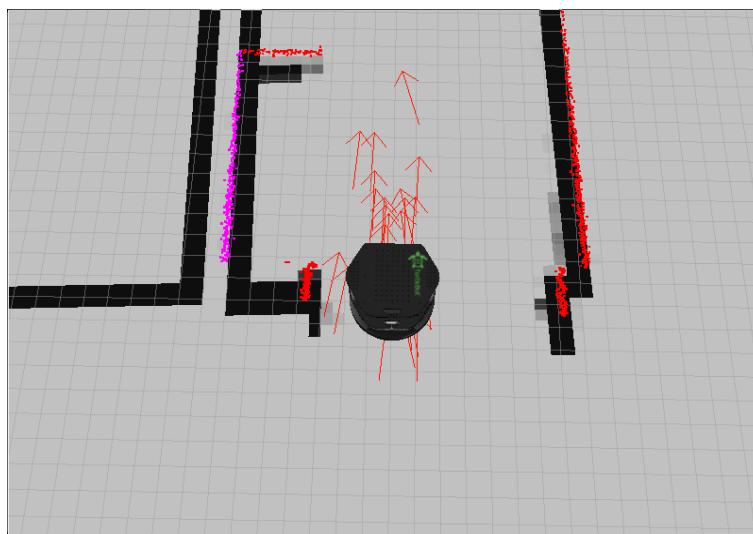


Figura 4.2: El robot se va acercando a la posición ideal

Observamos como la linea color de la parte derecha de la imagen 4.2, correspondiente a las muestras tomadas con el laser, está más cerca de la linea del mapa que en la imagen 4.1 que se aprecia que no está alineada. Esto es fruto de la corrección que se va haciendo de la posición. Tambien observamos que hay menos partículas, esto es fruto de la convergencia hacia la posición correcta.

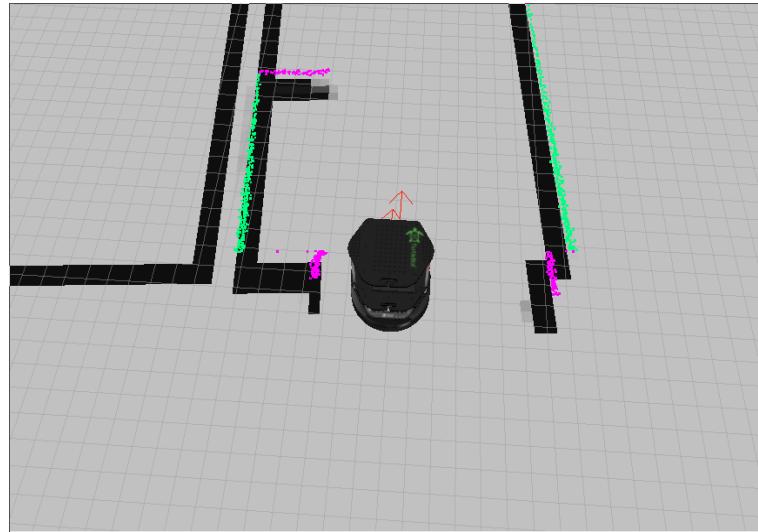


Figura 4.3: El robot se encuentra totalmente localizado

En la imagen 4.3 vemos como el número de partículas se ha reducido mucho, ya que se ha llegado a casi una estimación de la posición del robot muy cerca de la posición real. Vemos tambien que las lineas de color correspondientes a las muestras del laser están alineadas con el mapa.

Para el uso del paquete *amcl* en nuestro algoritmo fué necesaria la realización de una pequeña modificación. Esta modificación se refiere a que el paquete por defecto solo usa un mapa y lo obtiene al principio de la ejecución del algoritmo. Si le llegaba un nuevo mapa reiniciaba por completo el algoritmo. Esto nos generaba un problema, ya que en el algoritmo propuesto se publica un mapa por cada iteración y el paquete por defecto se reiniciaba constantemente. El efecto que producía es que el robot siempre se encontraba en la posición inicial y aunque lo movieramos siempre ocupaba la misma posición en el mapa. En nuestro *amcl* modificado se usa el mapa que se obtiene en cada iteración y sobre el se calcula la posición del robot, sin reiniciar en ningú momento el algoritmo.