Universidad Zaragoza

Trabajo de Fin de Grado

Construcción de mapas 3D densos para navegación de un robot móvil

Autor
Richard Elvira López-Echazarreta

Director
Juan Domingo Tardós Solano

Codirector
Luis Miguel Riazuelo Latas

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura

2016

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Richard Elvira López-Echzarreta,

con nº de DNI 16618876-L en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado de Ingeniería Informática, (Título del Trabajo) Construcción de mapas 3D densos para navegación de un robot móvil,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 20 de Septiembre del 2016

Fdo: Richard Elvira López-Echazarreta

Índice

1 - Introducción y objetivos

1.1 - Introducción

Se quiere realizar la navegación y localización de un robot mediante técnicas de V-SLAM Visual Simultaneous Localization And Mapping) las cuales se fundamentan en la realización de mapas y localización del robot en el mundo mediante sensores de visión.

Se van a recoger los datos que proporciona el sensor de visión y procesarlos para obtener la información que necesita el robot para saber como es el entorno que le rodea, de forma que pueda tomar decisiones autónomas con las que llegar al destino designado previamente, evitando los obstáculos que puedan impedir el avance del robot. La información que requiere el robot es un mapa de ocupación del entorno y su posición en dicho mapa para poder planificar sus trayectorias.

Como base va a emplear ORBSLAM2, que es un algoritmo desarrollado en la Universidad de Zaragoza el cual reconoce puntos de interés de cada imagen para saber donde se encuentra el sensor respecto a las anteriores. Tiene la gran ventaja sobre otros programas de SLAM de tener un coste computacional pequeño y a pesar de ello una gran precisión.

Pero para realizar el mapa por el que navegue un robot no es suficiente con los puntos de interés que detecte ORBSLAM, por lo que se requiere de un sensor de profundidad (RGBD) con el que en cada captura se obtiene la información de donde está en 3D cada punto de la imagen. De esta forma se puede hacer una reconstrucción densa del entorno, incluyendo zonas lisas y sin textura en las que ORBSLAM no es capaz de detectar puntos de interés.

1.2 - Objetivos

El objetivo de este trabajo es desarrollar un programa que permita la creación de mapas a partir de un sensor de visión RGBD con los que un robot pueda planificar una ruta hasta su destino. A diferencia de otros sistemas, que utilizan un láser de barrido plano y solamente tienen en cuenta los obstáculos presentes a una cierta altura, nuestro sistema construye mapas 3D densos, y los trasforma en mapas 2D adecuados para navegar, teniendo en cuenta los obstáculos presentes a lo largo de toda la altura del robot, lo que permite planificar trayectorias óptimas y seguras.

1.2.1 - Requisitos

- Construcción de mapas en 2D a partir del sensor de visión, que tenga en cuenta los obstáculos presentes en toda la altura del robot.
- Localización del robot en tiempo real en referencia al mapa generado.
- Navegación autónoma de un robot por el mapa 2D.

2 - Tecnología

El proyecto se ha desarrollado en un ordenador con sistema operativo Ubuntu y con el entorno ROS **[6]**, que interacciona con el robot, y se ha programado en C++. Los grandes bloques tecnológicos utilizados como base se describen a continuación. Para más información del equipo en el que se va a montar el proyecto ir al anexo 1.

2.1 – ROS (Robot Operating System)

Entorno de dominio público que funciona bajo sistemas Unix y desarrollado para interaccionar con robots, como un sistema operativo, que permite la abstracción del hardware del robot de forma que la comunicación con los sensores y actuadores de bajo nivel se realiza de forma sencilla mediante comunicaciones entre procesos y envío de mensajes. La comunicación entre el programa y el robot se realiza mediante publicaciones y lecturas de temas (*topics*).

También proporciona nodos con los que realizar tareas como visualizar el mapa y la posición del robot, de forma que facilita mucho la interacción con cualquier robot. Además ROS incluye nodos para permitir la navegación autónoma sobre un mapa 2D, o la evitación de obstáculos, con lo que para utilizar estas funciones solo será necesarios desarrollar el software que construya el mapa 2D, y adaptar los datos a la forma que necesita ROS.

2.2 - Sensor RGBD

Para realizar una reconstrucción de los obstáculos se va a emplear una cámara RGBD, también conocido como cámara de profundidad (*Depth*), esta tecnología se basa en dos cámaras y un proyector de infrarrojos, una de las cámara es una convencional RGB mientras que la otra procesa la información del proyector que emite un patrón conocido con el que se puede calcular la distancia a los objetos del entorno gracias a las variaciones que se detectan en dicho patrón.

Para el trabajo se va a emplear un sensor Asus Xtion PRO Live cuyas características se pueden ver en el anexo 6.3.

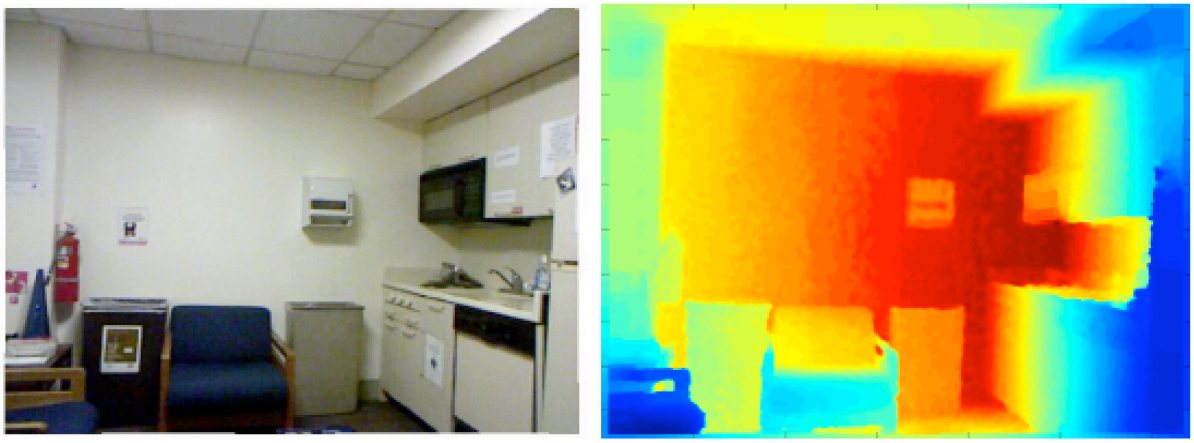


Figura 1: visión del sensor RGB (izquierda) y visión del sensor de profundidad (derecha).

2.3 - ORBSLAM2

Es un algoritmo de SLAM **[4]** desarrollado en la Universidad de Zaragoza en la tesis doctoral de Raúl Mur Artal, capaz de calcular con precisión la trayectoria que se ha seguido con un sensor de visión. El sistema funciona extrayendo puntos de interés ORB **[3]** que son puntos tipo “esquina” con un descriptor binario invariantes a la rotación y escala y con un bajo coste computacional. Emparejando los puntos de interés obtenidos en sucesivas imágenes, el sistema es capaz de obtener un mapa sencillo del entorno, formado por la posición 3D de los puntos emparejados, y de determinar con precisión la posición y orientación del sensor respecto de dicho mapa. Este cálculo de la localización resulta útil cuando se quiere lograr una navegación autónoma ya que es más fiable establecer la posición actual a partir de la escena que se ve que no a partir de los sensores internos del robot, como los encoders que miden el desplazamiento de cada rueda.

2.3.1 - Características

ORBSLAM2 está desarrollado en C++ y el código se encuentra accesible bajo licencia GPLv3 lo que quiere decir que el código es libre y se puede modificar para adaptar al uso que se le va a dar. Las principales razones por las que se va a usar son:

- Compatibilidad con cámara monocular, binocular y RGBD: implementa las funciones de localización con ese tipo de sensores, de los cuales se va a usar el RGBD (sensor de profundidad).
- Detección de cerrado de bucle: reconocimiento de escenas que previamente ya ha visitado para detectar cuando ha vuelto a una zona conocida y eliminar el error acumulado durante el movimiento, ajustando de mejor forma el recorrido y el mapa.
- Localización del sensor en el espacio en tiempo real: se puede ubicar la posición del sensor respecto a la posición inicial de forma que a medida que se mueve el sensor por el espacio se conoce en todo momento la posición y orientación que tiene.
- Compatibilidad con ROS: permite ser usado en el sistema operativo ROS que esta pensado para equipos robóticos, en este caso se va a preparar todo el sistema para que funcione en un robot real.

2.3.2 - Funcionamiento

Los componentes principales que se van a usar se representan con dos clases que se usaran en el resto del documento por lo que es importante conocer en que se diferencian:

- **Frame**, es una captura procedente del sensor que corresponde a la última imagen captada. Los puntos de interés extraídos en la imagen se comparan con el conjunto de puntos de interés del mapas, para decidir si la imagen tiene suficiente novedad para considerarla una imagen clave o “keyframe”

- **KeyFrame**, es una captura que después de realizar ciertas comprobaciones de emparejamiento se ha decidido guardar como una captura de interés para definir la trayectoria que ha seguido el sensor. Cuando se guarda un nuevo *KeyFrame* se guardan sus puntos de interés en una bolsa de puntos de interés para identificar la posición de los nuevos *Frames* a partir de estos.

El programa inicia 4 hilos diferentes para realizar todo el proceso, estos hilos se comunican entre ellos para lograr que todo funcione correctamente.

- **Main**, lanza el resto de procesos y captura la imagen del sensor para que sea procesada. Obtiene la posición respecto al inicio y comprueba si es necesario que se guarde como un nuevo *KeyFrame*.
- **LocalMapping**, gestiona los nuevos *KeyFrame* que se han agregado. Establece qué puntos se guardan en la bolsa de puntos de interés, de forma que no se encuentren datos repetidos.
- **LoopCloser**, se encarga de realizar comprobaciones periódicas con los últimos *KeyFrames* para ver si hay coincidencias con otros más antiguos y reconocer un cerrado de bucle con el que minimizar el error acumulado que se haya producido durante el ciclo.
- **Viewer**, muestra por pantalla dos visores:
  - **Visor del sensor:** proporciona la vista del sensor de visión en escala de grises y señala los puntos de interés reconocidos con dos colores. Los puntos verdes son los que se han emparejado con puntos de interés de la bolsa, mientras que los azules son puntos que se han reconocido pero no han sido emparejados.
  - **Visor 3D:** proporciona una vista en 3D en la que se puede interactuar. Los puntos rojos son puntos de interés que actualmente se están observando mientras que los puntos negros son puntos reconocidos que ahora no se están visualizando. El cuadro verde representa un *KeyFrame* y el azul la posición actual. Para identificar el recorrido se unen los *KeyFrames* consecutivos con una linea verde y a cerrar el bucle se unen los que han sido emparejados.

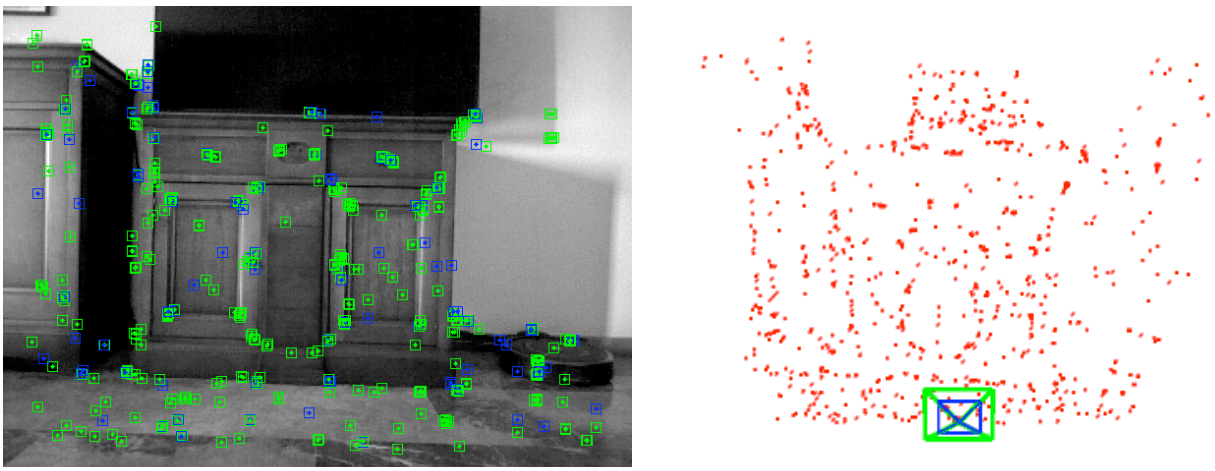


Figura 2: visión del sensor con puntos de interés (izquierda) y puntos de interés representados en visor 3D (derecha).

2.4 - TurtleBot2

El robot que se va a utilizar para realizar todos los experimentos es la segunda versión de TurtleBot, el cual es un robot con tracción diferencial, lo que le permite girar sobre sí mismo. Ademas incluye diferentes plataformas como se muestra en la imagen para poner los diferentes sensores. En nuestro caso solo se va a poner el sensor de visión a la misma altura que se puede observar en la figura 3. La plataforma superior se usará para colocar el ordenador que realiza todos los cálculos requeridos para la navegación.

La comunicación de los diferentes componentes del robot con el ordenador se realizan mediante ROS para simplificar las tareas.

Para más información ir al anexo 2.

Figura 3: robot TurtleBot2 con plataformas.

3 - Implementación

3.1 - Estructura del proyecto

Para abordar el trabajo a realizar se va a dividir en subtareas más abordables de forma que se centren los esfuerzos en cada uno de ellas hasta lograr los resultados para después unirlos y lograr alcanzar los objetivos.

Se va a realizar una programación sobre el código de ORBSLAM2 para incluir las funciones que se requerirán para formar un mapa a partir de la navegación con las imágenes captadas por el sensor Para lo cual se va a usar un ordenador con sistema operativo Ubuntu y se va a realizar en C++.

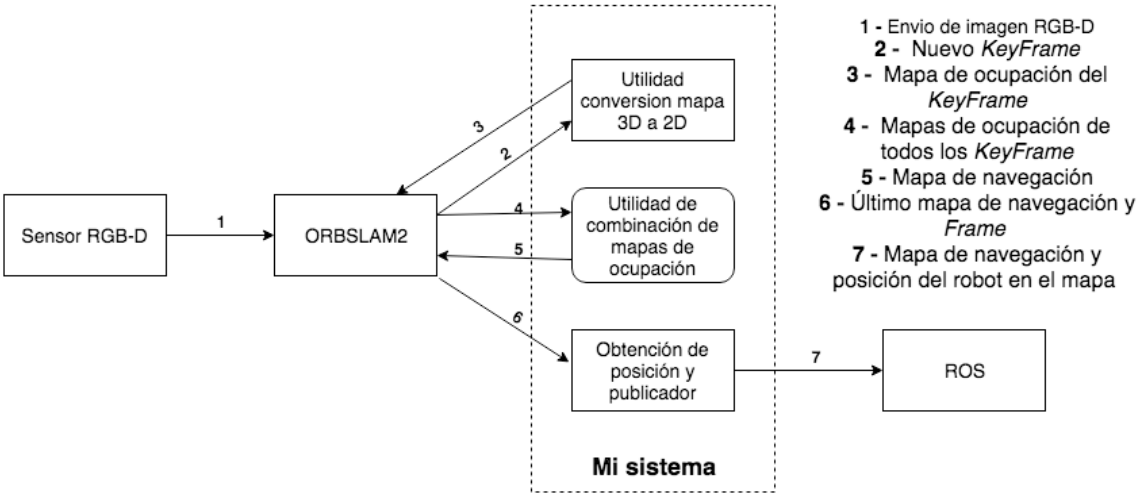


Figura 4: estructura del sistema y comunicación con otras herramientas.

3.2 - Construcción de mapas

Se quiere realizar un mapa con la visión de la cámara por lo que se va a guardar los rangos de profundidad de cada *KeyFrame* ya que con solo esos elementos se puede reconstruir la visión que ha tenido el robot durante toda la navegación. Estos puntos se verán en el visualizador de *ORBSLAM2* para comprobar si el mapa 2D que se genera coincide con la visión del sensor.

3.2.1 - Visión del sensor RGBD

En este apartado se busca capturar los puntos de profundidad de cada escena guardada y procesar la información que contiene para saber el espacio que ocupa cada uno de ellos, para lo que se usara el visor de *ORBSLAM2* el cual se ha modificado para mostrar dichos puntos en el espacio tridimensional. Estos puntos se pintan en azul para poder distinguirlos de los puntos rojos que son los puntos de interés que *ORBSLAM2* esta visualizando para calcular su posición.

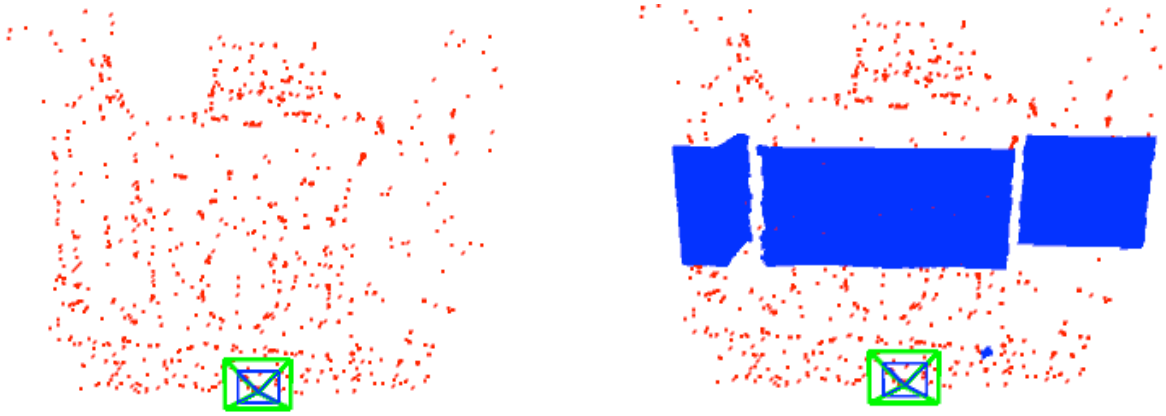


Figura 5: visión 3D de puntos de interés con ORBSLAM2 (izquierda) y con puntos de profundidad (derecha).

Esta información se extrae del sensor *RGBD*, el cual proporciona dos imágenes, una con la imagen *RGB* y otra con una imagen de profundidad la cual indica la distancia del objeto en cada pixel con un valor numérico de tal forma que se puede obtener la posición 3D de cada punto.

3.2.2 - Calculo de los obstáculos y zonas libres

Una vez se tienen los valores de profundidad 3D de cada *KeyFrame* se tiene que pasar la información a un mapa de ocupación 2D para poder realizar la navegación. Para ello se ha estudiado el formato que tienen los mapas de ocupación de ROS, que están formados por un conjunto de casillas de dimensión predefinida, y un vector en el que se señala para cada casilla, si la casilla es desconocida (-1), esta libre (0) u ocupada (100). Para más información sobre el mapa de ocupación ir al anexo 6.4.

La mayor parte de sistemas de construcción de mapas de ocupación construyen mapas estáticos, con lo que, al acumularse deriva en la posición del robot, se pierde mucha precisión, sobre todo al revisitar una zona que ya está en el mapa. En este trabajo queremos aprovechar los cerrados de

bucles de ORBSLAM2, que son capaces de corregir la posición de cada *KeyFrame* reduciendo la deriva. Para ello, es necesario rehacer el mapa de ocupación 2D después de las optimizaciones realizadas por ORBSLAM2, y sobre todo después de los cerrados de bucle.

Al iniciar este trabajo se pensó en realizar la transformación de todas las imágenes de profundidad al mapa 2D cada vez que se capturase un nuevo *KeyFrame*, pero esta solución es demasiado ineficiente debido al coste computacional que tiene esta tarea y al número de *KeyFrames* con los que se acaba trabajando. Además una vez que se captura la imagen de profundidad de un *KeyFrame* la información que proporciona no se modifica, únicamente es posible modificar la posición desde la cual se ha tomado el *KeyFrame* mediante los ajustes de localización que realiza el optimizador, por lo que se estaría extrapolando la misma imagen de profundidad continuamente pero con la posición de origen de la imagen modificada, y estos cálculos no se pueden realizar en tiempo real con todos los *KeyFrames* a la vez.

Para evitar repetir operaciones se ha decidido calcular un mapa de ocupación 2D estático por cada *KeyFrame*, a partir de la información de su imagen de profundidad. Cada cierto tiempo, el cual se establece mediante un fichero de configuración, estos mapas de ocupación se fusionan en un mapa de ocupación global, que es el que se utiliza para navegar.

Una vez definida la forma de representar la información se ha dividido la tarea en cuatro de menor complejidad.

- **Información del mapa e inicialización**: se debe establecer el tamaño del mapa de ocupación que se va a representar para cada *KeyFrame*, estos componentes son el alto y ancho del mapa, que se pueden calcular a partir de los valores de la imagen de profundidad. Si se establece un mapa de ocupación más pequeño de lo necesario se pierde información, mientras que si es muy grande se empeora el coste computacional al realizar más operaciones de las necesarias. El calculo de los valores iniciales es el siguiente:
    - Ancho, se calcula en función a la distancia máxima detectada en la imagen de profundidad siempre que no sea superior a un valor máximo que se establece por parámetro, en ese caso se usa el valor máximo.
    - Alto, se calcula a partir de la profundidad máxima que se ha obtenido en el cálculo del ancho del mapa.
    - Vector, de tamaño ancho*alto para que se pueda representar toda la información y cada celda se inicializa como una celda desconocida (valor -1).
- **Representación de los obstáculos**: para representar los obstáculos se utiliza la imagen de profundidad la cual proporciona una visión 3D en un fotograma 2D. Esto es gracias a que en cada pixel en vez de ver el color del objeto se tiene un valor que representa la profundidad a la que se encuentra. De esta forma, sabiendo la posición del sensor puede saber la altura a la que esta el objeto y la distancia a la que se encuentra. Para procesar un dato como obstáculo se deben cumplir ciertos requisitos:

- Lo primero es la distancia a la que se encuentra la cual se debe encontrar dentro del rango de visión óptimo del sensor, ya que como indica el fabricante a partir de cierta distancia los datos contienen mucho error y empeoran el resultado. Esta distancia se obtiene a partir del valor de la imagen de profundidad, si es no esta dentro del rango se deshecha.
- Después se comprueba la altura a la que se encuentra el objeto respecto al sensor, se calcula a partir de la distancia a la que se encuentra y la posición del punto dentro del fotograma, si el objeto no esta a la altura del robot se considera que dicho objeto no entorpece la navegación y se ignora.
- Si el objeto se encuentra dentro del rango de profundidad admitido y a una altura en la que entorpezca la navegación se procede a tratar como un obstáculo de forma que se calcula su posición en el vector que representa el mapa de ocupación y se pone el valor de ocupado (100) en dicha casilla.

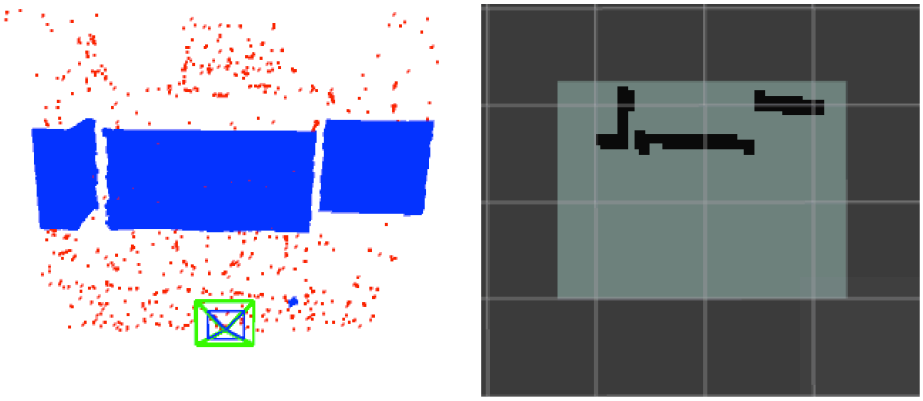


Figura 6: Mapa de profundidad 3D (izquierda) y mapa con obstáculos 2D (derecha)

- **Representación de celdas libres**: al identificar un objeto como obstáculo se pasa a calcular las casillas libres que se encuentran entre el obstáculo y el sensor de visión, conociendo la posición del obstáculo y el punto del sensor de visión se traza una linea entre los dos puntos mediante una función para recorrer dicha linea hasta encontrar un obstáculo o llegar al punto del sensor de visión marcando todas las casillas que son desconocidas(-1) como libres(0). Para hacerlo de una forma eficiente se debe tener en cuenta que una linea tiene infinitos puntos que se pueden visitar por lo que se establece la distancia de un punto a otro de la linea a partir de la resolución del mapa.

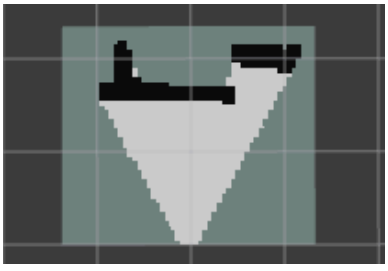


Figura 7: Mapa con obstáculos y celdas libres

- **Posicionamiento del mapa de ocupación:** una vez el mapa de ocupación se ha calculado, se debe establecer la posición del mapa a partir de la posición del sensor de visión, por lo que se debe calcular donde se encuentra el mapa y la orientación que tiene para lo que se utiliza la matriz de rotación y posición que guarda cada *KeyFrame* para identificar la posición y orientación del sensor. Con esta información y el ancho y alto del mapa que se ha calculado previamente se calculan los datos de posición del mapa.

3.2.2.1 - Problemas y mejoras

A medida que se iban realizando pruebas se han observados varios problemas que no se habían previsto.

- **Deformación en los bordes de la imagen del profundidad**: se produce una deformación de cojín en la imagen lo que significa que las esquinas se estiran produciendo que la información que se encuentra en ellas no sea muy fiable. Este tipo de distorsión se trata de dos formas, se puede procesar para corregir el error o se pueden eliminar los bordes de la imagen. En este caso se ha optado por eliminar los bordes de las imágenes ya que tiene menor coste computacional y el borde a eliminar es una región muy pequeña de la imagen, ya que al estar el sensor calibrado no hay mucha distorsión. En la figura 8 se puede observar el tipo de distorsión y como se produce en los bordes del mapa de profundidad. En este caso una pared plana parece que se curva hacia atrás, además se puede ver como las capturas sucesivas del mapa hace que se vean varios picos en las zonas que ha ocurrido el mismo problema.

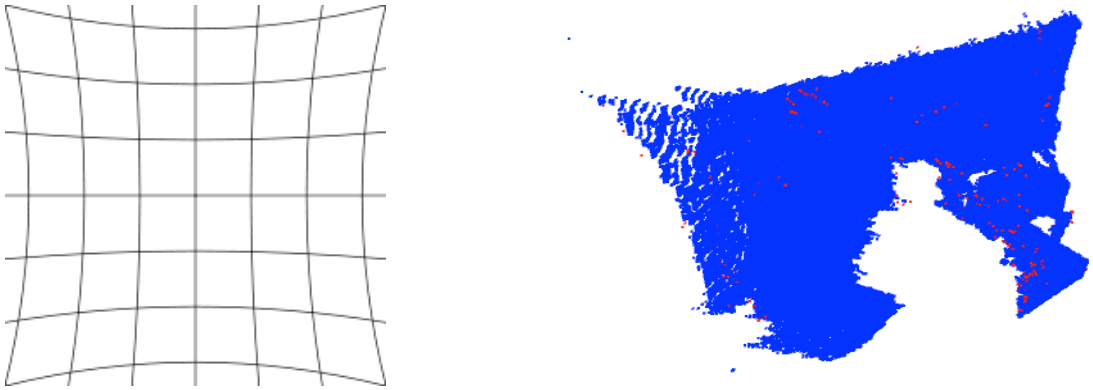


Figura 8: distorsión de cojín (izquierda) y mapa de profundidad con distorsión (derecha).

- **Distorsión en largas distancias:** realizar capturas con un sensor de visión nos da una imagen de lo que ha visto, pero el sensor de profundidad se basa en una tecnología que tiene un rango de captura para la escena de profundidad y si no se encuentra en dicho rango los datos pueden no ser fiables. Durante las primeras pruebas no se tenía en cuenta este rango por lo que se obtenían datos de profundidad que a medida que se alejaba de dicho rango se observa como los datos no eran consistentes y daba saltos en la profundidad. Se puede apreciar en la figura 9 que a pesar de ser una pared totalmente recta se observa que en el

final los datos se encuentran escalonados e introducen ruido que luego entorpece cuando se quiere construir el mapa. Para la versión final se ha introducido el rango del sensor, el cual se puede facilitar por parámetro en el fichero de configuración.

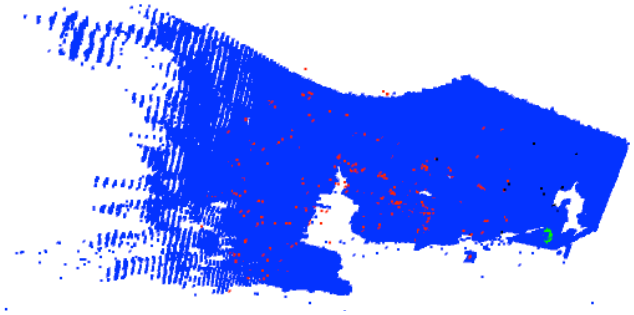


Figura 9: mapa de profundidad con error en distancias largas.

- **Rendimiento insatisfactorio:** una vez ya se ha realizado la creación del mapa para cada *KeyFrame,* se ha estudiado el rendimiento del programa. La primera versión no ha sido muy satisfactoria ya que el tiempo para generar cada *KeyFrame* se incrementó demasiado, dando lugar a que para generar la información del mapa de profundidad se necesitaban unos 500 milisegundos, por lo que el visor daba saltos de imagen y con giros muy abruptos se perdía la localización. Esto se debe a que este tiempo entorpece la correcta ejecución del programa que debe procesar 30 capturas por segundo (30 FPS) para garantizar el correcto funcionamiento, ya que si se insertan 2 *KeyFrames* nuevos al generar la información del mapa de ocupación de ambos se ocuparía un segundo de tiempo de computo, provocando que no puedan procesar mas imágenes. Este coste desmedido se encontraba en una mala optimización de la función que calcula los bloques libres, la cual se realizaba continuamente. Se ha solucionado realizando comprobaciones antes de lanzar la tarea, de forma que si se sabe de antemano que una nueva ejecución no va a agregar mas información al mapa de ocupación no se realizan los cálculos. De esta forma el resultado final del mapa de ocupación es el mismo pero el tiempo de creación de cada *KeyFrame* se ha reducido a 17 milisegundos, dando como resultado que si se procesan 30 imágenes por segundo aunque todas sean identificadas como *KeyFrames* se puede calcular la información de todas ellas a tiempo.
- **Zonas libres sin obstáculos**: a medida que se han realizado pruebas se ha observado un caso particular que aunque el sensor ve una zona libre el programa no la marca como tal y la deja como desconocida, esto es debido a que para marcar una zona libre se busca antes una zona con obstáculos aunque se encuentren fuera de rango y todo lo que este marcado como desconocido entre ese obstáculo y el sensor se marca como casilla libre, por lo que si no se ha encontrado ningún obstáculo la sección se queda como desconocida. Para solucionar de una forma óptima se ha optado por recorrer la imagen por columnas ya que el mapa de ocupación se construye de forma que es una proyección de la información desde arriba, por lo que al recorrer una columna entera sin encontrar un obstáculo se da por supuesto que en

ninguna altura se ha encontrado algo que entorpezca la navegación por lo que se marca desde el punto en el que se encuentra la imagen hasta el sensor como celdas libres.

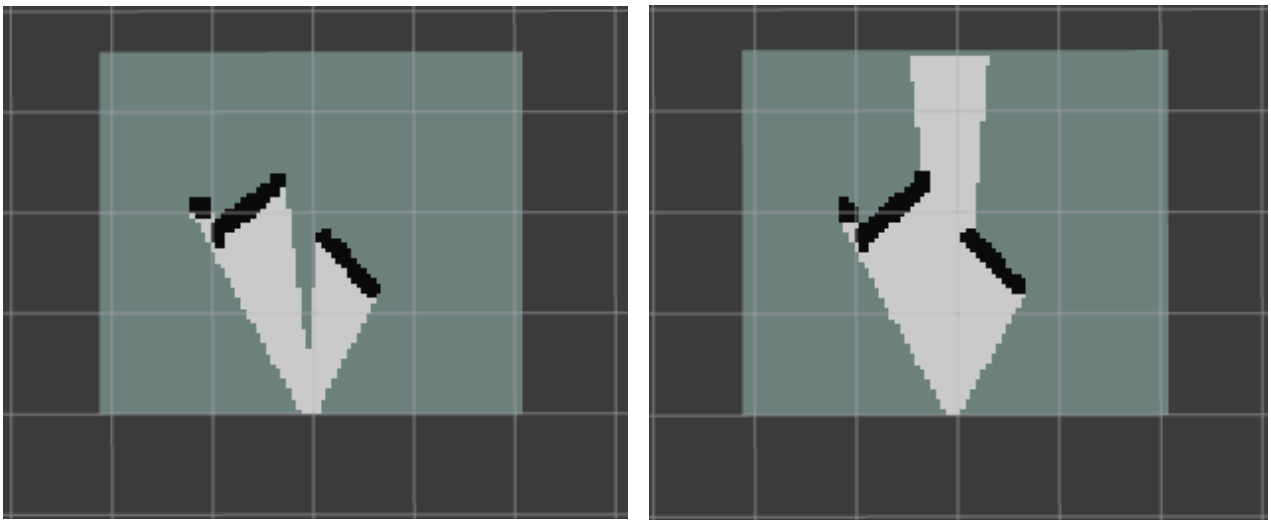


Figura 10: mapa sin algoritmo de zonas libres sin obstáculos (izquierda) y con algoritmo (derecha).

3.2.3 - Fichero de configuración

Para no dejar parámetros de navegación que dependen de los dispositivos que se van a usar en variables que luego requieran la compilación del proyecto, se ha decidido proporcionar un fichero de configuración en el que se establecen los datos dependientes de la navegación. Este fichero se ha de pasar en el comando de invocación del programa y tiene el siguiente formato:

- **Navigation.resolution,** resolución del mapa de navegación, se refiere al tamaño de las celdas del mapa.
- **Navigation.heightTop,** altura del robot por encima del sensor.
- **Navigation.heightBottom,** altura del robot por debajo del sensor.
- **Navigation.depthMin,** valor de profundidad mínimo para ser considerado obstáculo.
- **Navigation.depthMax,** valor de profundidad máximo para ser considerado obstáculo.

```
%YAML:1.0

#--------------------------------------------------------------------
# Navigation Parameters.
#--------------------------------------------------------------------

#Map resolution (in metres)
Navigation.resolution: 0.05

#Robot height from camera position (in metres)
Navigation.heightTop: 0.13
Navigation.heightBottom: 0.3

#Depth sensor range (in metres) Asus min:0.8, max:3.5
Navigation.depthMin: 0.4
Navigation.depthMax: 3.5

#Save data from camera capture(1=true, 0=false)
Navigation.saveDepthImage: 0
Navigation.saveDepthCloud: 1

#Generation map parameters(nombre, tiempo refresco)
Navigation.name: map
Navigation.milisec: 500

#Public map position for navigation(1=true, 0=false)
Navigation.public_pos: 1
```

Figura 14: fichero de configuración.

- **Navigation.saveDepthImage,** marca si se guarda la imagen de profundidad.
- **Navigation.saveDepthCloud,** marca si se guardan los puntos de profundidad, si no se guardan no se pueden visualizar en el visor 3D de *ORBSLAM2*.

- **Navigation.name,** nombre del tema (*topic*) en el que se publica el mapa de navegación para ROS.
- **Navigation.milisec,** cantidad de milisegundos que se debe esperar entre la creación de un mapa de navegación y el siguiente, si no se puede crear no hace nada.
- **Navigation.public_pos,** marca si se debe publicar la posición del robot por el tema correspondiente, solo es necesario para la navegación.

3.3 - Generación de mapa de navegación

Una vez se ha conseguido generar los mapas de ocupación de cada *KeyFrame* y establecer su posición y orientación correcta se debe generar el mapa de navegación que se obtiene al combinar la información de los mapas de ocupación **[2]** de todos los *KeyFrame*. Esta tarea se debe realizar de forma periódica creando el mapa desde cero cada vez, ya que *ORBSLAM2* se dedica a optimizar la posición de los *KeyFrames* para minimizar el error y al detectar un cerrado de bucle se puede observar una corrección mucho mayor. Estos ajustes afectan a la posición del mapa de ocupación de cada *KeyFrame* y hace que se tenga que generar de nuevo el mapa de navegación para tener la información más fiable posible debido a que al mezclar todos los mapas de ocupación no se puede saber de que *KeyFrame* es cada sección para modificarla.

Para que se realice esta tarea de forma periódica se ha creado un hilo que se lanza al ejecutar el programa y se encarga de mantener el mapa de navegación actualizado y se realiza de la siguiente forma:

1. Se obtiene el vector de *KeyFrames*.
2. Se coge el mapa de ocupación de cada uno de ellos y se inserta en un vector.
3. Se combinan todos los mapas de ocupación que contiene el vector en un mapa de navegación auxiliar.
    i. Se combinan los datos de todos los mapas para obtener los datos del mapa combinado de forma que se conoce la altura, anchura, posición y orientación del mapa a representar.
    ii. Se inician los valores del vector que representa el mapa de ocupación combinado a desconocido (-1).
    iii. Se recorre el vector de mapas de ocupación y se pasa la información al mapa combinado de forma que siempre tienen prioridad los obstáculos sobre las zonas libres.
4. Se actualiza el mapa de navegación final garantizando la exclusión mutua en las operaciones de leer y escribir en el acceso al mapa.

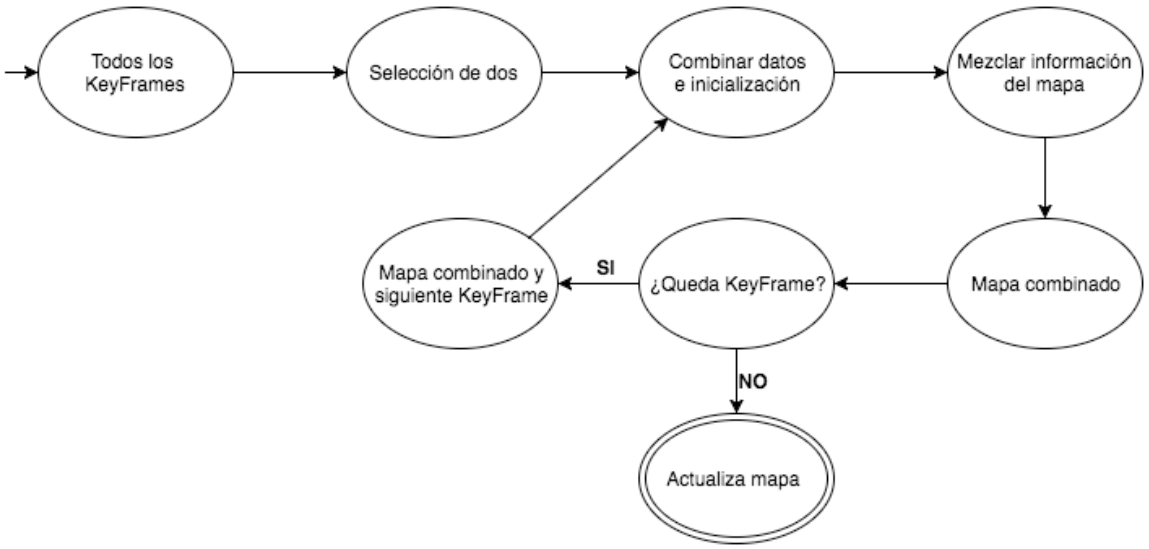


Figura 11: flujo de combinación de mapas

Los resultados que aporta la solución son muy fidedignos a los datos que se visualizan en el visor 3D de *ORBSLAM2*, tal y como se puede comprobar en la figura 12.

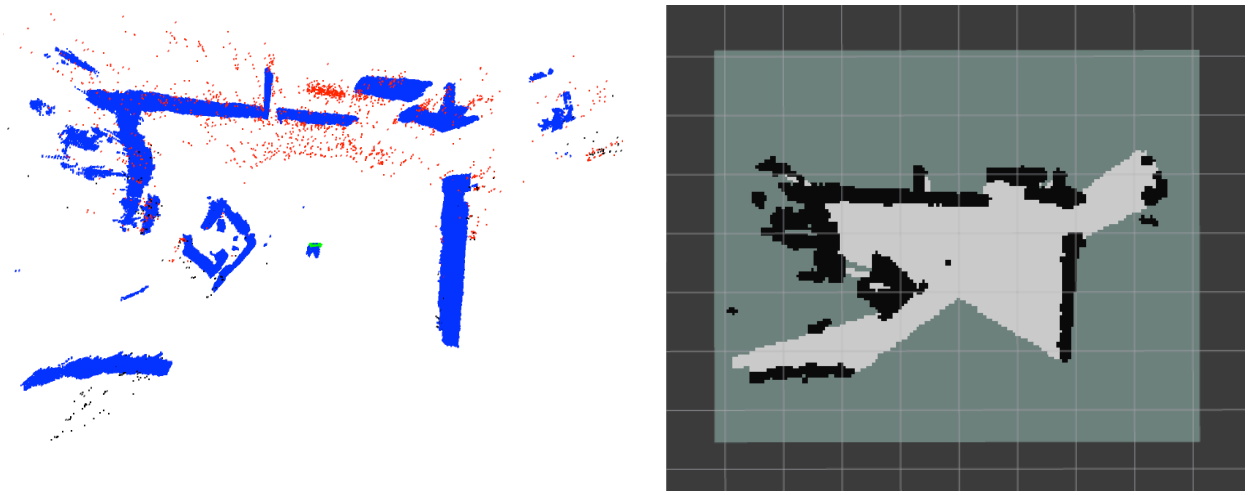


Figura 12: visión 3D de obstáculos de ORBSLAM2 (izquierda) y mapa de navegación generado (derecha).

Pero a pesar de los buenos resultados que proporciona se han tomado medidas de tiempos para evaluar el rendimiento de generar un mapa nuevo y cuando el número de *KeyFrames* crece mucho se puede observar como el coste de generar el mapa aumenta, haciendo que se navegue hasta un par de segundos con información obsoleta. Para darse este caso se deben tener muchos *KeyFrames* de los que extraer información, además el que no se haya terminado de calcular el mapa no quiere decir que no se pueda navegar ya que siempre se esta publicando el último mapa disponible que se ha generado.

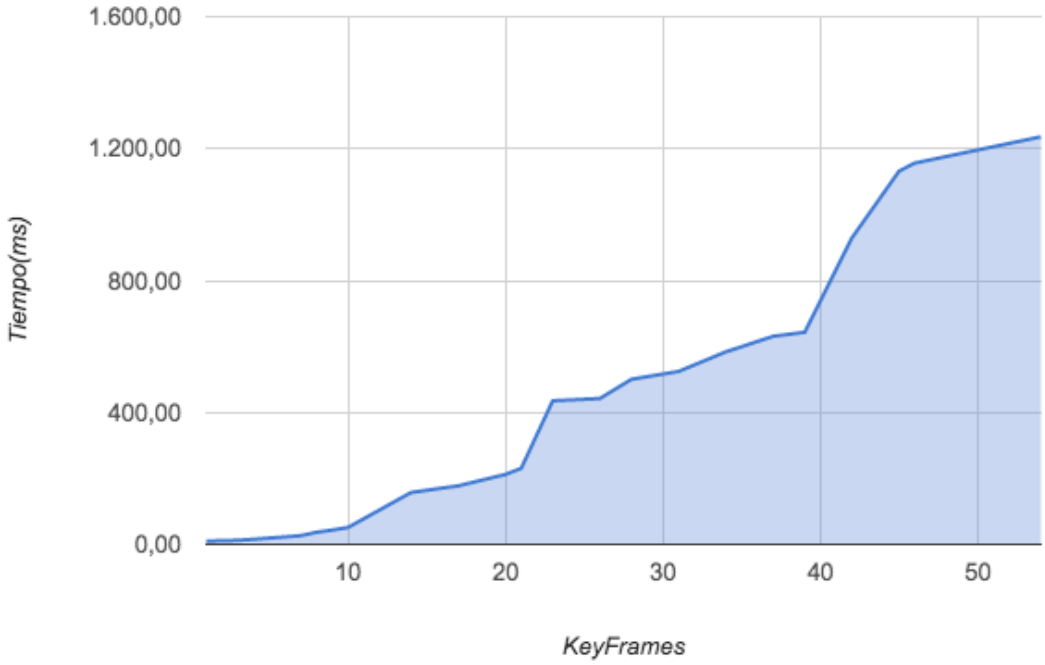


Figura 13: Tiempo (ms) de generación del mapa con determinado número de KeyFrames.

3.4 - Navegación del robot

Una vez el mapa de navegación se publica ya se puede tratar de navegar con el robot. Para ello, se necesita conectar y realizas las primeras pruebas, para comprobar si el mapa aparece bien representado respecto al robot. Además al localizar la posición del robot gracias a sistema de localización de *ORBSLAM2*, se necesita publicar esta información.

3.4.1 - Navegación

Con el mapa ya construido la principal tarea que se debe realizar es publicar la información de posición del robot **[5]**, para ello hay que entender como funcionan los temas (*topics*) que utiliza ROS para establecer la posición del robot en el mapa.

La posición se puede tomar en función del último Frame o KeyFrame pero se ha decidido tomar como referencia el Frame ya que es mas preciso al actualizar la información 30 veces por segundo de forma que tiene la posición más actualizada.

Para establecer la posición en el mapa del robot se deben utilizar los temas que se publican. Cada una con dos datos:

1. La posición con un vector de 3 elementos.
2. La orientación para la que se emplean cuaternios por lo que es un vector de 4 elementos.

Las referencias entre los temas se nombran con letras que contienen los datos con los que se opera para obtener la posición.

- **A,** posición del robot con respecto a la posición actual proporcionada por la odometría.
- **B,** posición del robot medido con los sensores de odometría.
- **C,** posición actual del robot en el mapa, esta referencia es simbólica para expresar los cálculos.

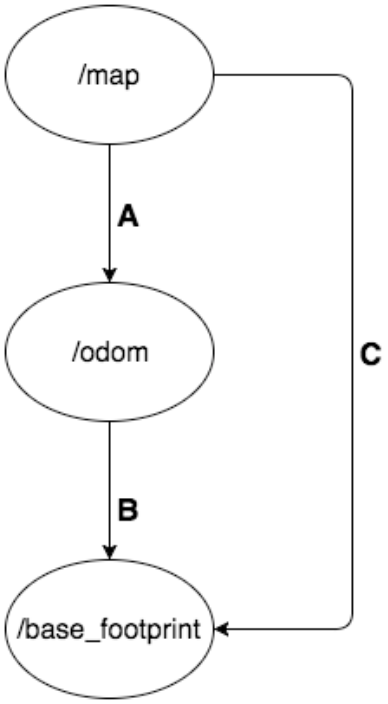


Figura 15: composición de posición

Al obtener la posición actual mediante *ORBSLAM2*, se establece que dicha posición es C. Sin embargo no se puede publicar esta posición y que el robot la admita ya que se calcula de la siguiente forma:

$$C = A * B$$

Siendo B una información que proporciona el robot y no se puede modificar, sin embargo se puede leer de forma que conociendo C y B se calcula cual es el valor de A que se debe propagar para obtener la posición deseada.

$$A = C * B^{-1}$$

De forma que se obtiene el valor de A y al publicarlo se asegura que el resultado final es la posición proporcionada por el programa.

3.4.2 – Transformación de referencias

Para establecer la posición del robot en el mapa se tiene que realizar una transformación de la posición y orientación. ORBSLAM2 proporciona estos datos con una matriz de transformación en la que se encuentra la información, pero ROS requiere que se le mande la información de traslación en un vector de 3 elementos y la orientación en un vector de 4 elementos llamado cuaternio.

La transformación de la matriz al vector de traslación es inmediata y no se debe cambiar nada mas que el eje de referencia, ya que aunque el eje X de ORBSLAM y ROS es el mismo, mientras que el eje Y es el eje Z del otro sistema.

Transformar la matriz de rotación a un cuaternio tiene algo más de complejidad, ya que solo se permiten rotaciones del robot en el eje Z del sistema ROS, por lo que el valor de X e Y siempre es 0. Para que las rotaciones se ejecuten de forma correcta se debe tener en cuenta las siguientes reglas:

$$W = coseno(ángulo)$$

$$X+Y+Z = seno(ángulo)$$

Como la rotación en el mapa solo se produce sobre el eje Z queda de la siguiente forma:

$$W = coseno(ángulo)$$

$$Z = seno(ángulo)$$

De esta forma se cumple la regla y el giro que se produce del robot en el mapa de navegación es el correcto. Si no se cumple la regla de arriba el giro no se realiza correctamente y a medida que se aleja del punto de origen se empieza a acumular error, dando lugar a que la posición sea incorrecta.

4 - Resultados

Se va a exponer uno de los multiples experimentos que se han realizado. se puede observar los pasos que sigue el programa para construir el mapa en el cual se encontrara un cerrado de bucle para apreciar como se corrige el error acumulado, luego se puede usar la información recabada para navegar por él.

4.1 - Construcción de mapas

Al iniciar el proceso de construcción de mapas se crean los primeros trozos del mapa con la visión de la cámara actual, tal y como se puede ver en la figura 16.

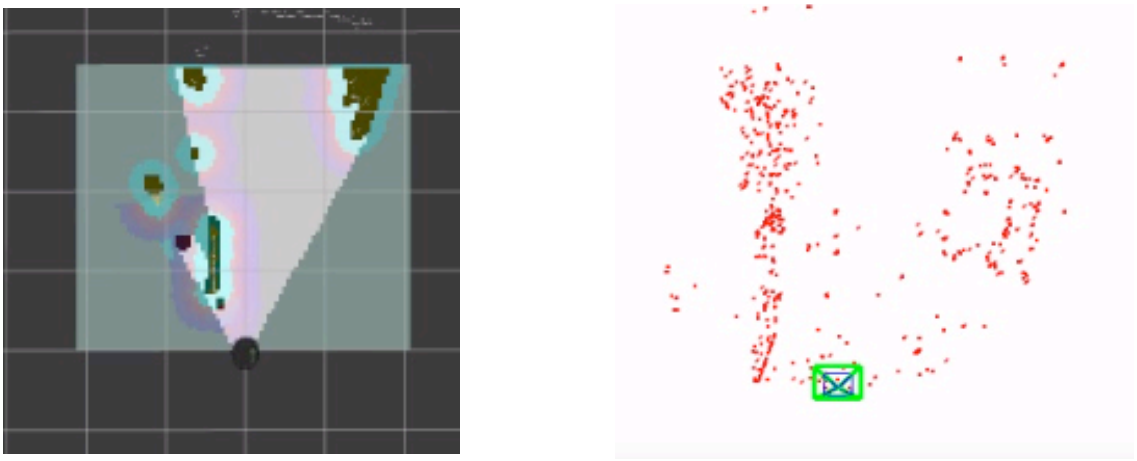


Figura 16: inicio de la construcción de mapas (izquierda) e inicio de visor de puntos te interés (derecha).

A medida que el robot se desplaza por el mundo, se agrega la información que va recabando sobre el entorno, de forma que el mapa aumenta de tamaño y proporciona más zonas para realizar la navegación. En la figura 17 se puede ver como crece el mapa y la trayectoria de navegación.

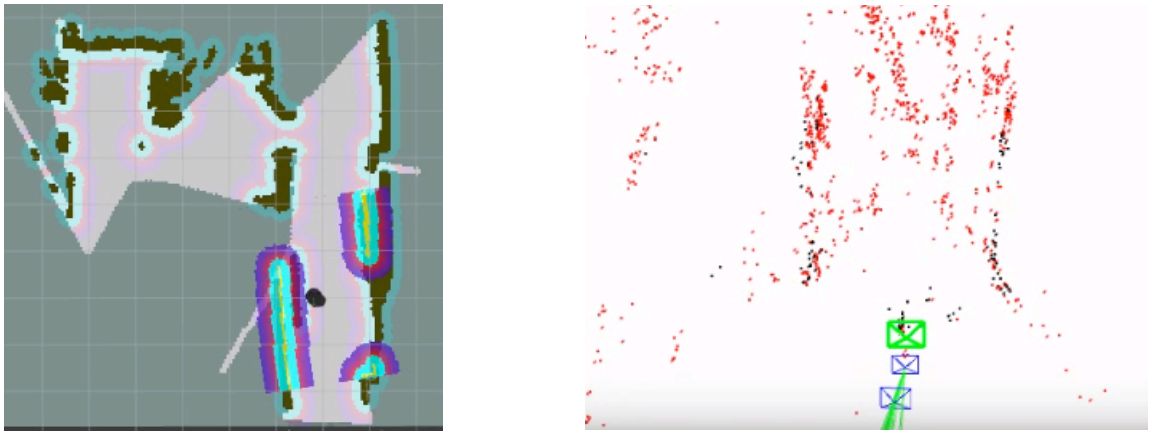


Figura 17: mapa de navegación (izquierda) y trayectoria de navegación de ORBSLAM2 (derecha)

Sobre los obstáculos que están en negro se puede observar un borde azul, que es un engorde que se aplica para evitar colisiones del robot. Se explicará en profundidad en la navegación.

4.2 – Cerrado de bucle

La creación del mapa no esta exenta de errores, los cuales se dan en la posición de robot y lejos de desaparecer, se van acumulando, de forma que el resultado final no es fiel con la realidad. En la figura 18 se puede ver como el robot al vuelto al punto de origen y tiene un deriva en la posición que ocasiona que el mapa de navegación no sea del todo correcto.

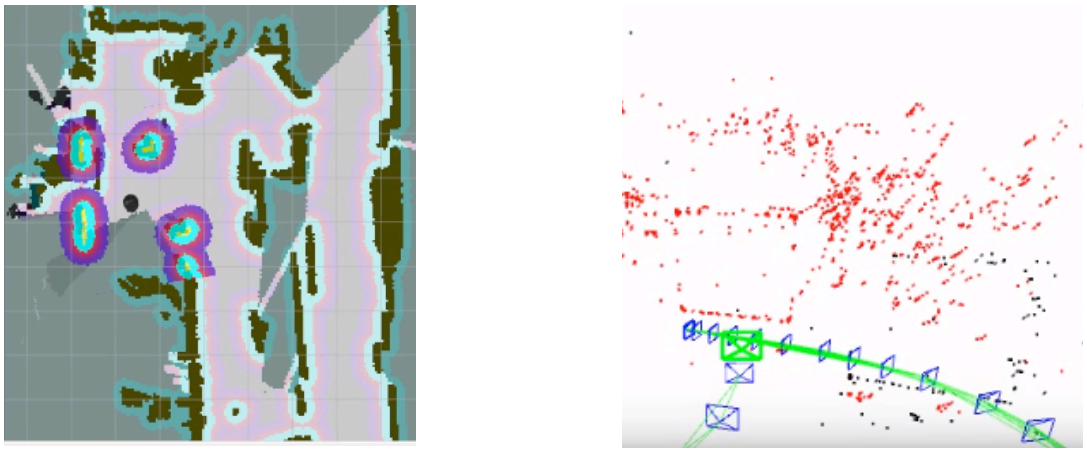


Figura 18: mapa de navegación sin cerrar el bucle (izquierda) y navegación a punto de origen (derecha).

No obstante gracias a que ORBSLAM2 detecta cerrados de bucle y corrige las posiciones de los *KeyFrames*, de forma que se elimina la deriva. Al calcular de nuevo el mapa de navegación con el error corregido se traza un nuevo mapa más fiel a la realidad, como se puede ver en la figura 19.

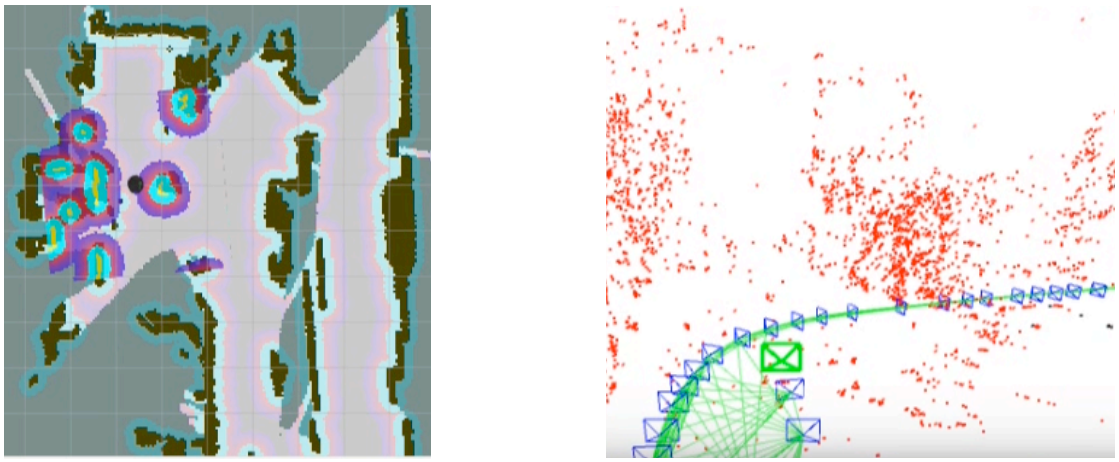


Figura 19: mapa de navegación corregido (izquierda) y enlazado de KeyFrames (derecha).

Estas correcciones permiten una mejor visión del entorno y localización del robot en el mapa para que la navegación sea más precisa.

4.3 - Navegación

El proceso de navegación se realiza con un nodo de ROS el cual calcula la ruta óptima al destino. Para lo que usa un mapa de costes, el cual se representa como un cuadrado coloreado alrededor del robot y los colores más cálidos indican la ruta con menor coste, mientras que los colores fríos tienen un coste mayor.

Además los obstáculos que entran en el mapa de costes adquieren varios colores, el color amarillo representa el obstáculo el cual es envuelto por un color azul que realiza la función de engordar el obstáculo en función del tamaño del robot, de forma que si hay un solo pixel libre de estos colores, la zona se considera transitable. También se ve color del rojo al morado alrededor del engorde, estas zonas son transitables pero consideradas zonas de riesgo por lo que intenta evitarlas si es posible y sino se puede evitar, la maniobra se realiza con más cuidado.

Se establece la meta mediante el cursor en el pasillo y el navegador establece la ruta óptima con el mapa conocido, como se ve en la figura 20.

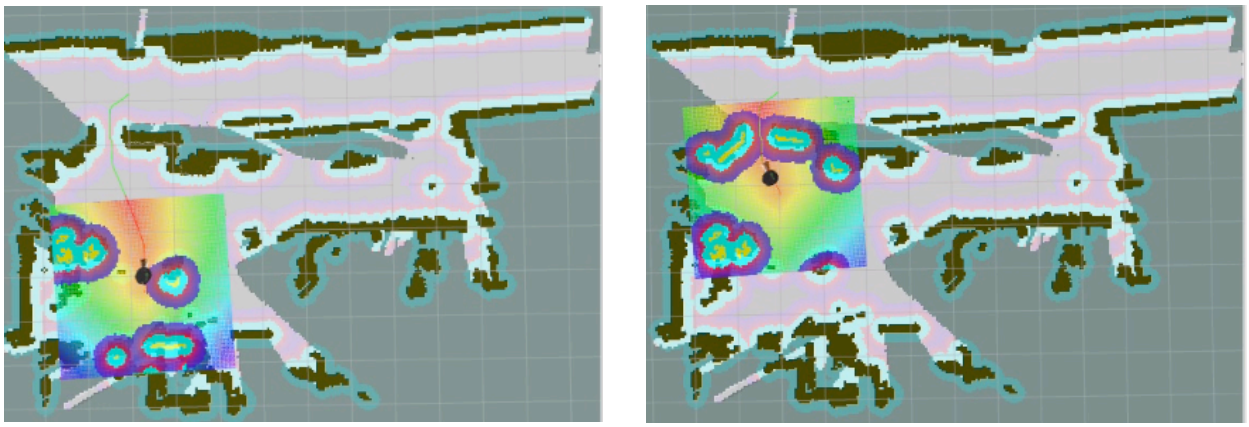


Figura 20: navegación desde el punto inicial (izquierda) y navegación en zona de riesgo (derecha).

5 – Conclusiones

Después de múltiples pruebas se ha logrado alcanzar los objetivos propuestos y se han observado muchas mejoras que aplica frente a los métodos convencionales.

Otros sistemas de navegación y construcción de mapas en tiempo real emplean radares láser para conocer la ubicación de los obstáculos, pero este tipo de tecnología solo otorga la visión de un plano, por lo que todos los elementos que se encuentren por encima o debajo de ese plano no son representados. En el sistema que se ha desarrollado se emplea un sensor de visión, de tal forma que se tiene una visión de toda la escena y se representan los obstáculos en el rango de altura que puede entorpecer la navegación del robot. Esto permite realizar mejores planificaciones y evitar colisiones con objetos que no se encuentren a la altura del sensor láser.

El hecho de usar ORBSLAM2 le da mucha potencia a esta utilidad, ya que al navegar siempre se acumula error en la localización, por lo que un sistema convencional representa los obstáculos con el error que arrastra el robot y no se corrige posteriormente. ORBSLAM2 tiene un forma de corregir la deriva que aparece al navegar al reconocer zonas ya visitadas, momento en el que cierra el bucle y corrige el error de posición, en ese momento también se corrige el mapa de navegación y se elimina el error acumulado.

Al pasar por zonas ya visitadas y corregir el error se evita introducir obstáculos duplicados en diferentes posiciones, mientras que en los sistemas convencionales al no poder eliminar la deriva en la posición del robot se puede detectar el mismo obstáculo en otra posición y al agregarlo como si fuera un nuevo obstáculo bloquear zonas transitables.

6 - Posibles mejoras

Al acabar todo se han planteado mejoras que se pueden implementar para hacer que el programa funcione mejor:

- Sistema distribuido, con cálculos complejos de combinación de mapas en servidor usando la información de diversos nodos de forma que a los nodos se les quita la mayor carga de trabajo y se logran mejores mapas al tener información de diferentes nodos móviles.
- Implementar diferentes políticas de combinación de mapas de forma que se pueda detectar de mejor forma los obstáculos fijos de los móviles.

7 - Anexos

7.1 - Especificaciones del ordenador

Se va a trabajar con un ordenador portátil MacBook Pro de 15 pulgadas de finales del 2011 con las siguientes especificaciones:

- Procesador: Intel Core I7-2675QM.
  - Frecuencia: 2.2GHz.
  - Cores: 4.
  - Hilos/Core: 2.
- Memoria RAM: 16GB DDR3 a 1333MHz.
- Sistema operativo: Ubuntu 14.04.
- Versión ROS: Indigo.

7.2 - Especificaciones de TurtleBot2

El robot empleado durante todos los experimentos ha sido la segunda versión del conocido TurtleBot **[7]**, el cual esta construido desde una base Kobuki a diferencia de la primera versión que era una versión modificada de iRobot (Roomba).

- Máxima velocidad: 0.65m/s.
- Dimensiones: 35.4 x 35.4 x 42 cm.
- Peso: 6.3Kg.
- Peso de carga máximo: 5Kg.
- Api: ROS.

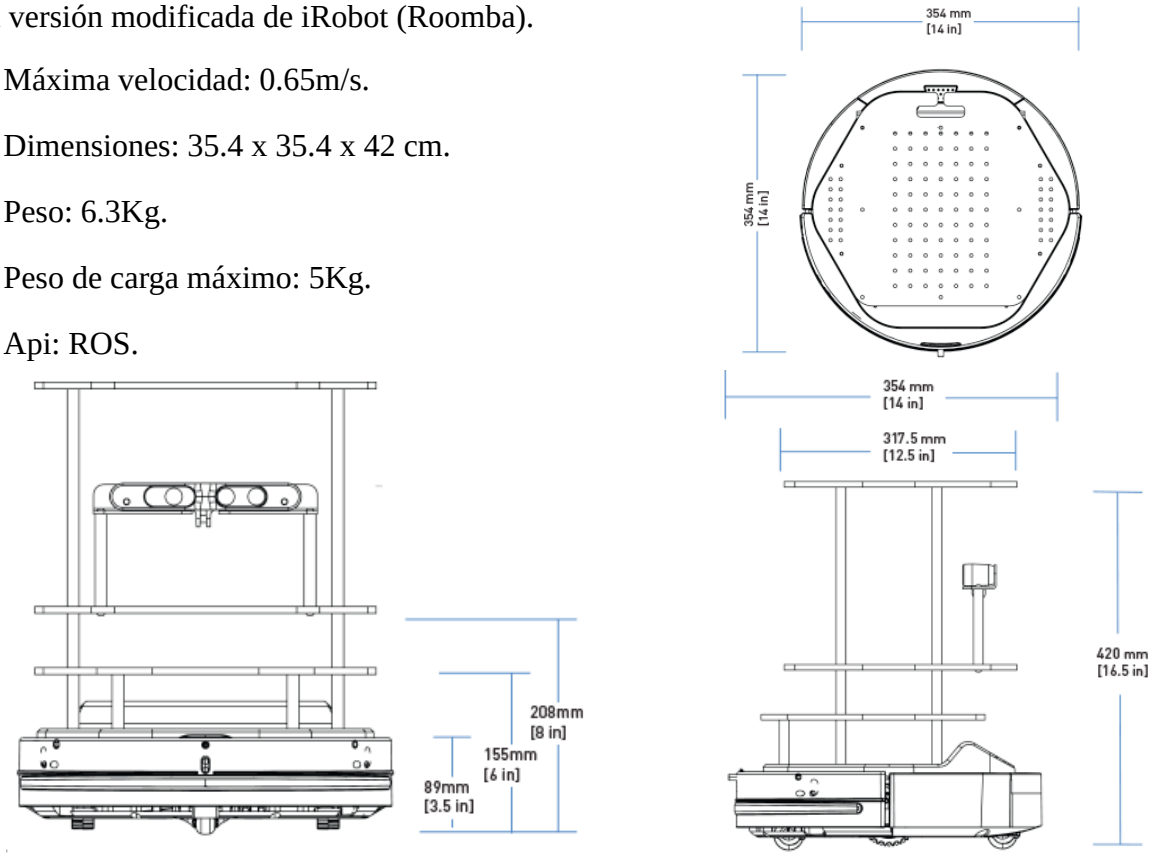


Figura 21: visión frontal del robot (izquierda), lateral (derecha) y desde arriba (arriba a la derecha).

7.3 – Especificaciones del sensor de visión

Se ha utilizado el sensor RGBD de Asus "Xtion PRO Live" **[1]**.

Figura 21: Sensor de visión empleado.

- Sensor: RGB y profundidad.
- Resolución: SXGA(1280x1024).
- Profundidad del tamaño de la imagen.
  - VGA (640x480) 30fps.
  - QVGA (320x240) 60fps.
- Campo de visión: 58° H, 45° V, 70° D (Horizontal, Vertical, Diagonal).
- Distancia de uso: Entre 0.8 m y 3.5 m.
- Interfaz: USB 2.0.
- Software: Kit de desarrollo de software (OPEN NI SDK bundled).
- Ambiente de operación: interiores.
- Dimensiones: 18 x 3.5 x 5 cm.

7.4 - Formato del mapa de ocupación

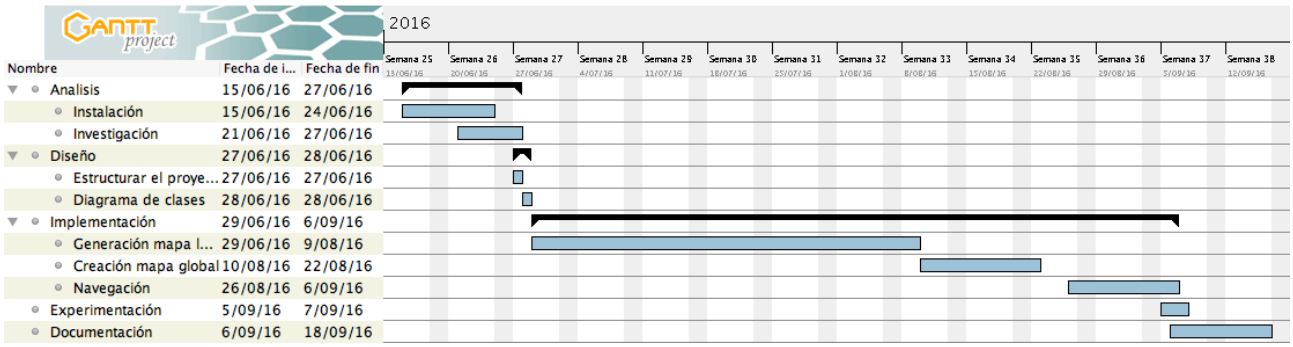
El formato para que ROS pueda entender el mapa de ocupación es del tipo OccupancyGrid que contiene la siguiente información:

- **header**, cabecera del mapa.
  - seq, número que identifica el mapa de ocupación, se utiliza una secuencia incremental para no repetirse.
  - stamp, tiempo en el que se ha creado el mapa de ocupación.
  - frame_id, identifcador del tipo de mapa.
- **Info**, información del mapa de ocupación.
  - map_load_time, tiempo en el que el mapa se ha cargado.
  - resolution, resolución de las celdas en metros/celda.
  - width, ancho del mapa en número de celdas.
  - height, alto del mapa en número de celdas.
  - origin, posición y orientación del mapa en el mundo.

- **position**, posición del mapa en el mundo 3D.
  - x, posición del mapa en la coordenada x.
  - y, posición del mapa en la coordenada y.
  - z, posición del mapa en la coordenada z.
- **orientation**, orientación del mapa representada con un cuaternio.
  - x, rotación sobre el eje x.
  - y, rotación sobre el eje y.
  - z, rotación sobre el eje z.
  - w, número complejo que representa el total de grados girados.

- data, vector de enteros de 8bits que contiene el valor de cada celda, siendo: -1 celda desconocida, 0 celda libre y 100 celda ocupada.

7.5 - Diagrama de ocupación



Tiempo de dedicación aproximado 380h.

7.6 - Referencias

[1] - Asus Xtion PRO Live → https://www.asus.com/es/3D-Sensor/Xtion_PRO_LIVE visitado en 16/06/2016

[2] – Occupancy Grid Utils → http://wiki.ros.org/occupancy_grid_utils visitado en 09/08/2016

[3] - ORB → http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_orb/py_orb.html visitado en 16/06/2016

[4] - ORBSLAM2 → http://webdiis.unizar.es/~raulmur/orbslam visitado en 16/06/2016

[5] - ROS Navigation → http://wiki.ros.org/navigation/Tutorials/RobotSetup visitado en 13/08/2016

[6] - ROS → http://wiki.ros.org visitado en 16/09/2016

[7] - TurtleBot → http://www.turtlebot.com visitado en 13/09/2016