Universidad Zaragoza
1542

Trabajo Fin de Grado

Título del trabajo:
Navegación autónoma de un robot utilizando mapas de elevación

English title:
Robot autonomous navigation using elevation maps

Autor/es

Fco. Javier Oíza Cólera

Director/es

Luis Montano Gella

Escuela de Ingeniería y Arquitectura de Zaragoza
2018

Repositorio de la Universidad de Zaragoza – Zaguan
http://zaguan.unizar.es

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. Fco. Javier Oíza Cólera,

con nº de DNI 20491377X 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, (Título del Trabajo) Navegación autónoma de un robot utilizando mapas de elevación

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

Zaragoza, Junio de 2018

Fdo: Fco. Javier Oíza Cólera

Navegación autónoma de un robot utilizando mapas de elevación

RESUMEN

El proyecto tiene como objetivo la localización y navegación autónoma de un robot en un entorno en el que hay obstáculos no conocidos a priori. Para ello es necesario detectar e identificar los obstáculos, etiquetándolos para construir un mapa de transitabilidad. Dependiendo del tipo y capacidades de navegación del robot y de los obstáculos, alguno de estos obstáculos no serán transitables, pero otros sí, como por ejemplo rampas. A su vez algunas de estas rampas serán transitables o no por el robot disponible. Se trata de etiquetar los obstáculos del entorno de acuerdo con las capacidades de los robots, y por tanto el software desarrollado deberá ser parametrizable en función de estas capacidades (p.e. pendientes navegables o no por cada robot).

Un aspecto fundamental para la navegación es la localización en el escenario. Para ello se utilizarán técnicas de localización basadas en mapa. Este mapa contendrá información del entorno que permitirá realizar esta localización con precisión, y por otro lado contendrá las diferentes zonas etiquetadas de acuerdo con la capacidad del robot para navegar por ellas.

Para la realización de estas tareas se utilizará una plataforma robótica, el robot Turttlebot, que tiene embarcados diferentes sensores. Para este trabajo se utilizarán dos de ellos: 1) un escáner láser Hokuyo, que permitirá la localización del robot en el entorno mediante técnicas conocidas, “gmapping” o “hector_mapping”; 2) Un sensor RGB-D, en este caso una cámara Asus Xtion Pro Live que permitirá construir mapas de elevación del terreno, que servirá para el etiquetado de los obstáculos y la definición de las zonas transitables y no transitables.

Una vez creados los mapas para localización y transpirabilidad, se adaptará a ellos una técnica de navegación autónoma, la Ventana Dinámica (Dynamic Window Approach, DWA), permite navegar evitando colisiones en base a los mapas de navegación que se construyen simultáneamente en tiempo real.

Además del equipamiento mencionado, se utiliza un simulador, Gazebo, y un entorno de desarrollo para robótica, ROS (Robot Operating System). Estas herramientas software permiten poner a punto los algoritmos desarrollados y el ajuste de parámetros, en escenarios simulados, previamente a su implementación en el robot real.

Índice

Índice de figuras

1. Introducción

1.1 Contexto o trabajo previo

La robotización de aplicaciones es un proceso que día a día se va introduciendo en todos los sectores industriales y de servicio. Muchas de estas aplicaciones están relacionadas con la navegación autónoma de robots en escenarios reales interiores o exteriores, que presentan obstáculos para la navegación, algunos de ellos salvables y otros insalvables. La posibilidad y capacidad para poder navegar se llama transitabilidad, que se representa en mapas donde aparecen las zonas transitables y las no transitables. Ello a su vez depende de las capacidades de navegación del robot que se utilice; unos robots serán capaces de subir rampas de 45º y otros no, de sobrepasar o no bordes de acera, etc. Ello deber reflejarse en el mapa que el robot utiliza para navegar, y debe ser adaptado a sus capacidades. Como en la Figura 1 que se ve un robot navegando sobre un terreno irregular. Esquivando los posibles obstáculos que pueda encontrar.

En este trabajo se desarrollan los algoritmos que permiten construir y etiquetar mapas de navegabilidad o transitabilidad, y utilizarlos para la navegación autónoma de robots. Los mapas se construirán a partir de la información capturada por algunos de los sensores embarcados. Este trabajo ha sido realizado en el laboratorio de Robótica en el Instituto de Investigación de Ingeniería de Aragón (I3A) en la Universidad de Zaragoza, utilizando el equipamiento hardware y software del Grupo de Robótica, Percepción y tiempo real del Instituto.

Figura 1. Robot en terreno irregular

1.2 Tareas realizadas

En este proyecto lo que se desea conseguir es un robot que consiga localizarse y navegar en un mapa mientras lo va creando, Para ello se han realizado los siguientes pasos:

1. Estudio, instalación y familiarización de los entornos de ROS y Gazebo
2. Documentación y estudio de la utilización de las funciones de ROS: navegación, mapas, localización, sensor RGB-D (Asus Xtion Pro Live), escáner Hokuyo
3. Localización en un mapa por medio del escáner Hokuyo
4. Creación de mapas de elevación a partir del sensor RGB-D
5. Clasificación y etiquetado en el mapa de las zonas transitables u obstáculos para la creación de un mapa de obstáculos a partir del de elevación
6. Navegación autónoma del robot en el mapa creado
7. Ajuste de parámetros, puesta a punto final y experimentación en diferentes escenarios simulados y reales
8. Documentación de las diversas pruebas realizadas y de los diversos problemas encontrados y como solucionarlos

1.3Técnicas y herramientas utilizadas

En esta sección se explicaran las técnicas y herramientas más importantes utilizadas para la realización de este proyecto dividido en los 4 puntos principales del trabajo:

-Localización en el mapa

Por medio del “hector_mapping” [ver Anexo F (Hector-SLAM)], implementado en ROS [ver Anexo A (ROS)], utilizando barridos del láser embarcado en el robot Turtlebot [ver Anexo B (Turtlebot)], se detectan los obstáculos situados en un ángulo de unos 140º y una distancia de hasta unos 4m, Esta información se representa en el mapa para la localización. Este sistema, en un principio, se realizó utilizando la técnica “gmapping” [ver Anexo E (Gmapping y AMCL)] implementada en ROS, y con la misma cámara Asus Xtion Pro Live [ver Anexo D (Cámara Asus Xtion Pro Live)], Posteriormente se utilizó la técnica “hector_mapping”, también implementada en ROS, ya que con ella se podía incorporar información de inclinación (Pitch) del robot en el terreno, a partir de la información de un sensor inercial (IMU). A partir de esta información se crea un mapa adicional de elevación, con lo cual se dispone de una información completa de localización en el mapa global absoluto, esto es, las coordenadas x-y-z del robot, su orientación (ángulo Yaw, ψ), y la inclinación (ángulo Pitch, θ).

-Creación del mapa de elevación

Para la creación del mapa de elevación se requieren los datos de la cámara Asus Xtion Pro Live y la pose (x,y,z,θ,ψ) del robot. Ésta es obtenida mediante "hector_mapping" y una estimación de la altura, a partir de la pose 2D (x,y,ψ) del robot y la inclinación(θ). El "elevation_mapping" crea un mapa/retícula (grid) de elevación con información de altura asociada a cada celda del grid como se ve en la Figura 2 y en la Figura 3. Son distintas vistas de este mapa de elevación. También incluye una escala, para ver qué color corresponde a una mayor o menor altura.

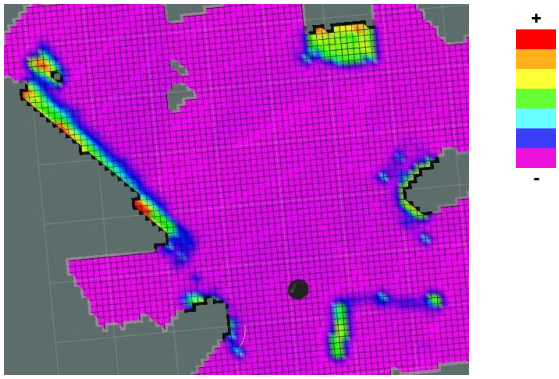


Figura 2. Vista superior mapa de elevación

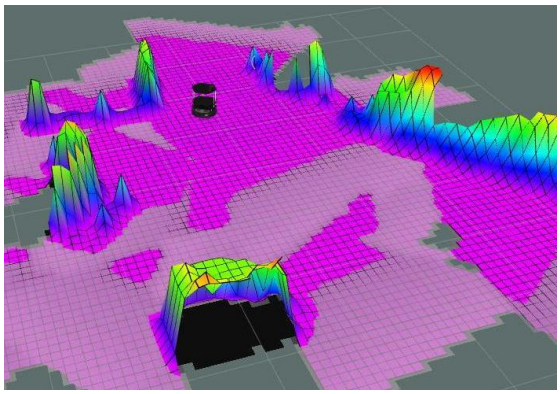


Figura 3. Vista lateral mapa de elevación

-Procesamiento del mapa de obstáculos

La construcción del mapa de obstáculos se obtiene a partir del mapa de elevación. La diferencia de altura entre celdas vecinas en el mapa, permite el etiquetado de los obstáculos, transitables o no. Las celdas correspondientes no traspasables se etiquetarán como no transitables. Aquellas que correspondan a rampas, se etiquetarán como transitables o no, en función del robot disponible y su capacidad de subir rampas. El Turtlebot utilizado en este trabajo tiene una capacidad de subir rampas con muy poca inclinación, pero otros robots pueden subir rampas con mayor inclinación. Este será un parámetro ajustable en función del robot utilizado. Por ejemplo en la Figura 4 se ven 2 tipos de rampas con diferente inclinación. Y en la Figura 5 se ve como se crea el mapa de obstáculos. La rampa de mayor inclinación la toma como obstáculo, mientras que la otra la ve como transitable.

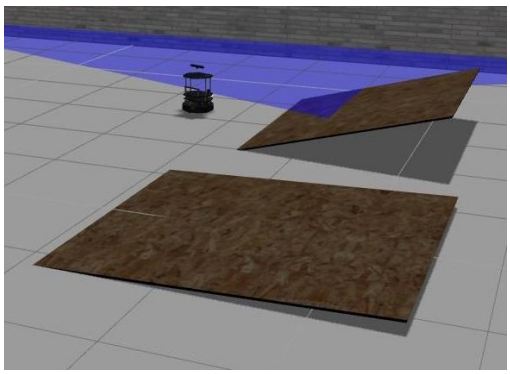


Figura 4. Diferentes pendientes

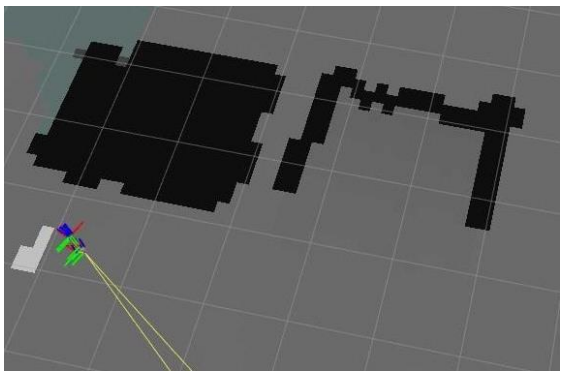


Figura 5. Mapa de obstáculos de pendientes

-Navegación en el mapa

Las técnicas de navegación utilizadas se aplican sobre estos mapas de transitabilidad, denominados “costmaps”. Los controles aplicados al robot en cada periodo de muestreo o control, usualmente consignas de velocidad lineal (v) y angular (ω), se calculan para obtener un movimiento que evite los obstáculos. Es decir, utiliza la información del “costmap” para generar las trayectorias más cortas hacia el objetivo, evitando las colisiones con los obstáculos representados en el mapa. El costmap local obtenido en el campo de visión de los sensores se integra en el mapa global que mantiene y actualiza en todo momento el sistema, obteniendo sobre él, el camino o trayectoria al objetivo. A continuación se ve una secuencia de decisiones de navegación que utiliza el robot para su planificación y navegación.

1. Considera las posibles velocidades (v, ω).
2. Restringe a las velocidades seguras, con las que tenga tiempo de detenerse antes de llegar al obstáculo más cercano.
3. Restringe a las velocidades alcanzables con las características del robot.
4. Elige la consigna de velocidad (v, ω) óptima teniendo en cuenta:
    a. La posición (x,y, θ) del robot
    b. La distancia con el obstáculo más cercano

1.4 Estructura del proyecto:

La memoria de este proyecto está estructurada de la siguiente forma:

-**Capítulo 2. Localización y mapeo.** En este capítulo se presenta las diferentes alternativas analizadas y evaluadas para la construcción de mapas y la localización en ellos, justificando la elección final realizada.

-**Capítulo 3. Creación de mapa de transitabilidad y navegación en él.** Se presenta inicialmente la construcción del mapa de transitabilidad a partir del mapa de elevación, y posteriormente la navegación autónoma utilizando dicho mapa.

-**Capítulo** 4. Implementación**.** En este capítulo se explica cómo se ha llevado a cabo la implementación en ROS de los puntos más importantes del trabajo.

-Capítulo 5. Nodos ROS. En este capítulo se ven los nodos que utiliza ROS con una breve explicación de cada uno de ellos y los topics más importantes.

-**Capítulo 6. Evaluación experimental.** En este capítulo se presentan las pruebas y evaluaciones que se han hecho para la puesta a punto del sistema.

-**Capítulo7. Conclusiones.** Se presentan las conclusiones finales del trabajo, las limitaciones, los problemas encontrados, y algunas soluciones a plantear en futuros trabajos. Para cerrar el capítulo acabo con una valoración personal de lo que he aprendido y lo que me ha aportado este trabajo.

2. Localización y mapeo

2.1 Resumen del proceso

Esta primera parte del trabajo consiste en la localización de un robot dentro de un mapa (el cual se va construyendo en tiempo real conforme el robot va avanzando). Por tanto para esta primera parte se requiere alguna de las técnicas de localización que ofrece ROS. Tal como “AMCL” o “gmapping” [Anexo E (Gmapping y AMCL)] o el “hector_mapping” [Anexo F (Hector-SLAM)]. Con estas técnicas se puede obtener la posición del robot en un mapa, con una incertidumbre debida a las diferentes fuentes de error (precisión de los sensores, error de localización acumulado, error de predicción, etc.). Al mismo tiempo el robot amplia la información de este mapa. Esta información proviene de los barridos láser del propio sensor embarcado en el robot. Estos sistemas están todos diseñados para la navegación del robot en 2D, pero en este caso, se necesita un sistema en 3D que permita al robot navegar en terrenos con una cierta inclinación o altura respecto al terreno horizontal. Ello conlleva el modificar el sistema de referencias básico del robot, como se explica con más detalle en el capítulo 2.2 Sistemas de referencia y transformaciones.

2.2 Sistemas de referencia y transformaciones

Las técnicas de mapeo y localización exigen el manejo de transformaciones relativas entre sistemas de referencia, que representan localización de objetos, sensores o puntos relevantes para esta localización. La Figura 6 representa las principales referencias y la relación entre ellas, utilizadas en la implementación. El árbol de transformaciones del robot es algo muy importante para poder conseguir la altura y la inclinación debido a que inicialmente el árbol de transformadas que tiene el robot es el básico como se ve en la Figura 6. Donde se ven un conjunto de transformaciones:

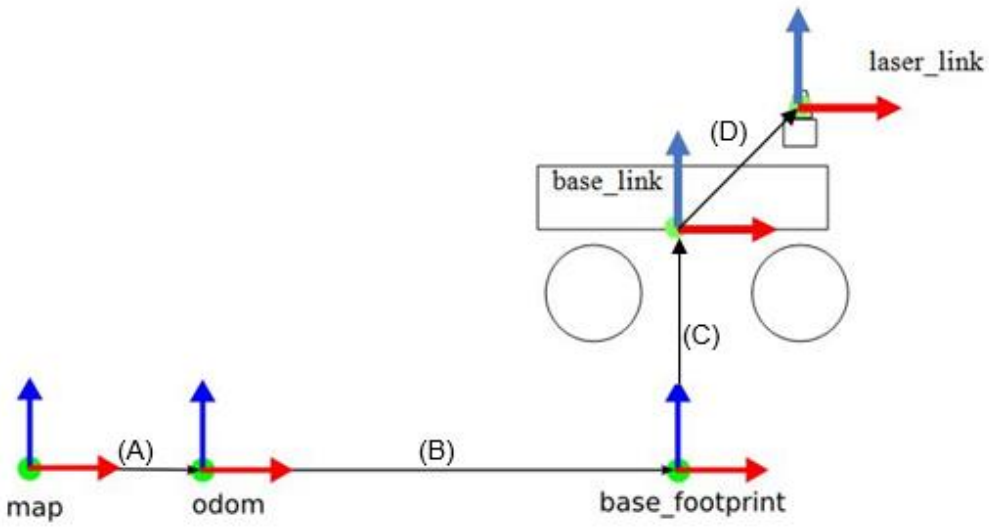


Figura 6. Referencias y transformaciones básicas del robot en ROS para navegación en 2D

(A) relaciona “map” con “odom” y representa la corrección que hace el sistema de localización a la odometría del robot en relación al mapa del que se dispone.

(B) relaciona “odom” con “base_footprint” y representa la información obtenida de la odometría del robot.

(C) relaciona “base_footprint” con “base_link” y representa la posición de la base del robot respecto al suelo.

(D) relaciona “base_link” con “laser_link” y representa la posición del láser respecto a la base donde se referencian también los demás sensores que se necesiten.

En estas relaciones las alturas son fijas. Y dependen de la altura que tenga la base respecto al suelo o la altura que tenga el láser respecto a la base.

Por ello si se quiere que el robot pueda navegar con una inclinación y con cierta altura respecto al terreno, hay que diseñar otro árbol de transformaciones como el que se ve en la Figura 7. En él se ven un conjunto de transformaciones similar al anterior pero dividiendo la transformación (C) en dos transformaciones diferentes.

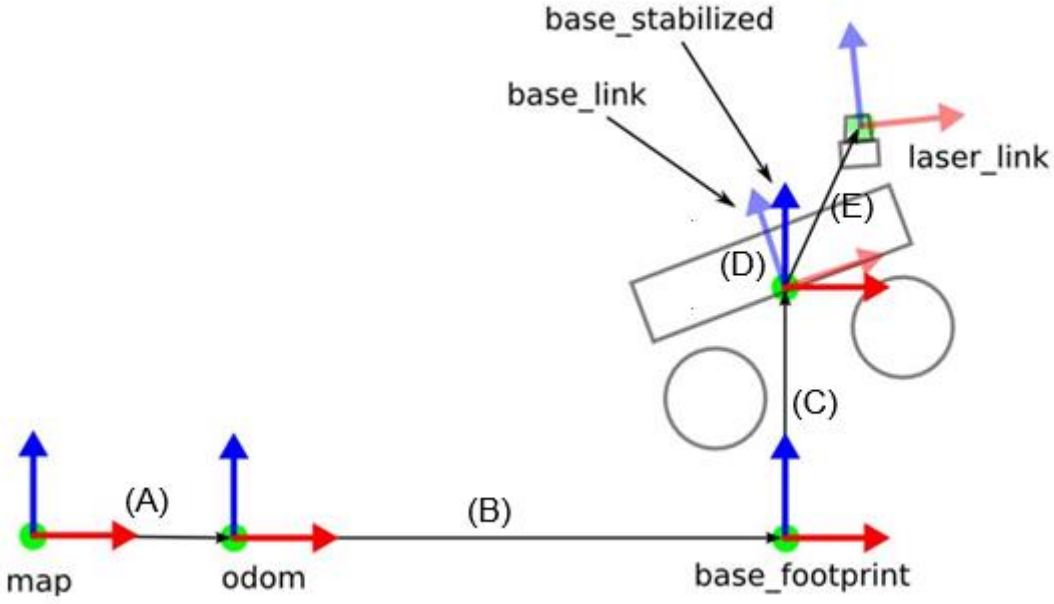


Figura 7. Referencias y transformaciones básicas del robot en ROS para navegación en 3D

(A) relaciona “map” con “odom” y representa la corrección que hace el sistema de localización a la odometría del robot en relación al mapa del que se dispone.

(B) relaciona “odom” con “base_footprint” y representa la información obtenida de la odometría del robot.

(C) relaciona “base_footprint” con “base_stabilizer” y representa la altura del robot.

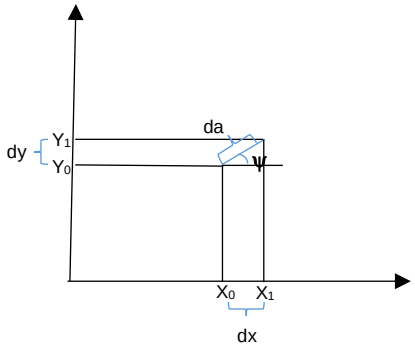
(D) relaciona “base_stabilizer” con “base_link” y representa la inclinación que tiene el robot.

(E) relaciona “base_link” con “laser_link” y representa la posición del láser respecto a la base donde se referencian también los demás sensores que se necesiten.

En este caso la “base_footprint” sería la base del robot proyectada en el espacio 2D y con la altura se obtendrá “base_stabilizer”, que sería la representación de la “base_link” pero sin la inclinación.

La inclinación del robot se obtiene de un sensor de medida inercial (IMU) que da la orientación del robot, además de otras medidas que no interesan para este trabajo.

La altura se obtendrá a partir de la posición del robot en 2D (actual y anterior) y de los datos de orientación obtenidos del IMU. En este caso aunque la trayectoria del robot sea circular se puede estimar como recta, ya que las distancias son pequeñas y el error es mínimo al ser un periodo de muestreo muy bajo. Como se ve en la Figura 8 Se quiere obtener la distancia que recorre el robot en 2D (da).



dx: representa el avance en x entre el instante anterior y el actual.

dy: representa el avance en y entre el instante anterior y actual.

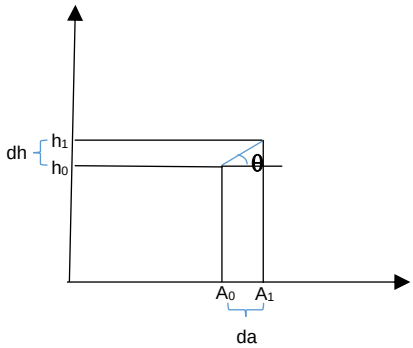
mYaw (ψ): es la media del ángulo Yaw que tiene el robot durante esos instantes.

Figura 8. Recorrido del robot en 2D

Para calcular da se utiliza la fórmula matemática:

$$da = \frac{dx}{\cos \psi}$$

Para la estimación de la altura solo se tiene en cuenta el ángulo Pitch (θ de inclinación, debido a que el robot no puede moverse en dirección transversal. Como se ve en la Figura 9, se quiere obtener la diferencia de altura del robot entre el instante anterior y el actual.



da: representa la distancia recorrida por el robot en 2D entre el instante anterior y actual.

mPitch (θ): es la media del Ángulo Pitch que tiene el robot entre esos instantes.

dh: es la diferencia de altura del robot entre el instante anterior y actual.

Figura 9. Diferencia de altura del robot

$$dh = da * \tan \theta$$

Una vez se obtenga la *dh* con la fórmula anterior, solo se tendrá que sumar la altura anterior, y ya se obtiene una estimación de la altura del robot en cada momento. Finalmente, la transformación entre “map” y “base_link” será la obtenida de la información de posición del robot en 3D.

La información relacionada con la implementación de las transformaciones en ROS se encuentra en el capítulo 4.1 Obtención de la posición en 3D.

3. Creación de mapa de transitabilidad y navegación en él.

3.1 Resumen del proceso

En este capítulo se trata de la navegación del robot, la creación del mapa de elevación y su procesamiento para crear un mapa de obstáculos. Este mapa es necesario para la planificación y ejecución del movimiento del robot. Primero se necesita la creación del mapa de elevación para lo cual se utiliza el "elevation_mapping" [ver Anexo G (Elevation_mapping)] que crea un mapa en 3D a partir de la cámara RGB-D Asus Xtion Pro Live y de la pose del robot. Este mapa de elevación, representa la información de altura de cada punto del mapa. Como se ve en la Figura 10, en el que hay una rampa con diferentes colores según la altura. Una vez se tenga el mapa de elevación, se necesita transformarlo en un mapa de obstáculos. Para que el método de navegación pueda planificar el mejor camino (el más corto, el de menor tiempo, etc.) y mande la orden de movimiento al robot. La forma de crear y transformar este mapa de elevación en un mapa de obstáculos se explica en el capítulo 3.2 Procesamiento del mapa.

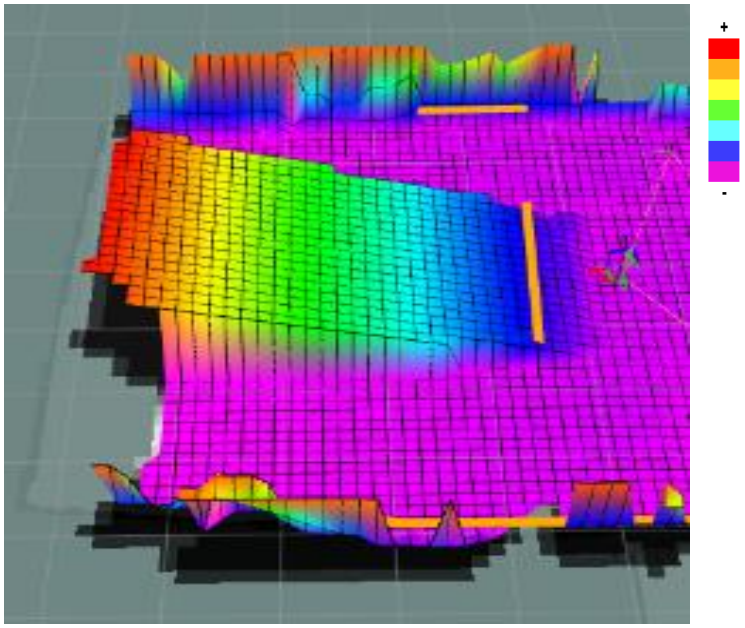


Figura 10. Mapa de elevación con diferentes alturas

3.2 Procesamiento del mapa

En este capítulo se pretende realizar el procesamiento del mapa de elevación, para conseguir un mapa de obstáculos. Una manera simple de realizar el mapa de obstáculos, sería que todos los puntos con altura mayor de un determinado umbral se consideren obstáculo. Sin embargo, en terrenos irregulares con elevaciones no es posible aplicarlo, ya que consideraría, por ejemplo, las rampas como obstáculos. Dependiendo de las capacidades del robot algunas rampas serán transitables y otras, con pendiente eleveda, no lo serán. Es necesario por tanto dotar al sistema de capacidad para distinguir y representar adecuadamente estas situaciones. En la Figura 11 se representan diferentes obstáculos: (A) Un muro, que claramente es un obstáculo. (B) Un

pallet, que es un obstáculo para este robot que no puede subir aunque sean pequeños escalones. (C) Una rampa, que en este caso sería obstáculo porque el robot solo puede subir rampas de menor inclinación como la (D), que si sería transitable. El caso de las rampas se tendrá que hacer parametizable en función del robot que se vaya a utilizar y sus características de navegación.

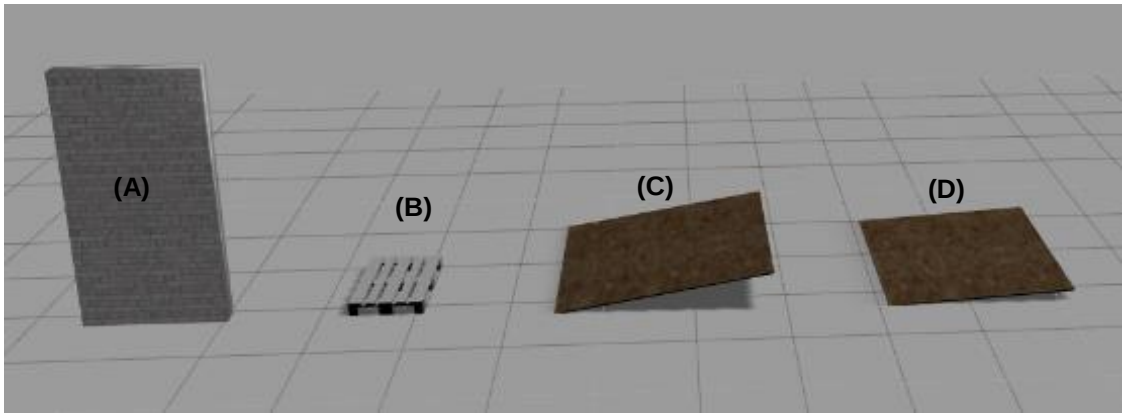


Figura 11. Diferentes obstáculos

Para realizar este procesamiento se compara la altura de las celdas vecinas de cada una de ellas (arriba, abajo, derecha e izquierda), y a partir de esta diferencia se establece si es obstáculo o no, dependiendo también del máximo ángulo de inclinación que pueda subir el robot. El pseudocódigo siguiente implementa el método:

```
for(fila=0; fila<max_numero_filas;fila++){
  for(columna = 0; columna < max_numero_columnas; columna++){
    if(celda(fila,columna) == not_value){
      celda(fila,columna) = desconocida;
    }else{
       if (celda(fila-1,columna) == value && abs(celda(fila-1,columna)-celda(fila,columna))>max_dif) {
          celda(fila,columna) = obstaculo;
      }elseif(celda(fila+1,columna) == value && abs(celda(fila+1,columna)-celda(fila,columna))>max_dig){
            celda(fila,columna) = obstaculo;
      }elseif (celda(fila,columna-1) == value && abs(celda(fila,columna-1)-celda(fila,columna))>max_dif)
            celda(fila,columna) = obstaculo;
      }elseif(celda(fila,columna+1) == value && abs(celda(fila,columna+1)-celda(fila,columna))> max_dif){
            celda(fila,columna) = obstaculo;
      }else
          celda(fila,columna) = transitable;
    }
    }
    map(fila,columna) = celda(fila,columna);
  }
 }
```

El valor umbral para decidir si es obstáculo o no dependerá de cada robot. En la Figura 12 se ve la dependencia entre la resolución del mapa, que es la distancia entre cada vecino, y el ángulo máximo que puede afrontar el robot según sus características.

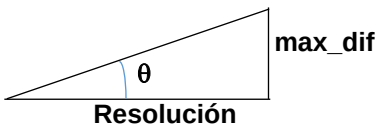


Figura 12. Máxima altura

Resolución: es la distancia entre los vecinos.

max_dif: Diferencia máxima de altura entre vecinos.

θ: Ángulo de la pendiente máxima que puede subir el robot.

Entonces se obtiene el valor que se utilizara como “max_dif” con la fórmula:

$$\max\_dif = Resolucion * \tan\theta$$

Para construir el mapa se guarda en una matriz llamada “mapa” el valor según la celda sea transitable, obstáculo, o desconocida. Esta matriz es el mapa de obstáculos creado.

Otro punto a tener en cuenta en el procesamiento del mapa, es que el robot no puede ver a una distancia menor de 0.5m de la cámara. Por tanto, como se ve en la Figura 13. El robot puede ver a su alrededor (lo marcado en rosa) a una distancia superior a ese límite, la zona de radio aproximado de 0.5m alrededor de él es desconocida. En movimiento este problema no aparece, ya que esa área alrededor del robot habrá sido vista anteriormente, por lo que tendrá un valor de elevación y etiqueta de transitabilidad. Sin embargo antes de comenzar el movimiento, cuando empieza a crear el mapa esa zona es desconocida. Una condición es que inicialmente no haya obstáculos en esa área. El mapa de obstáculos deberá representar inicialmente esa zona como transitables, para poder comenzar la navegación.

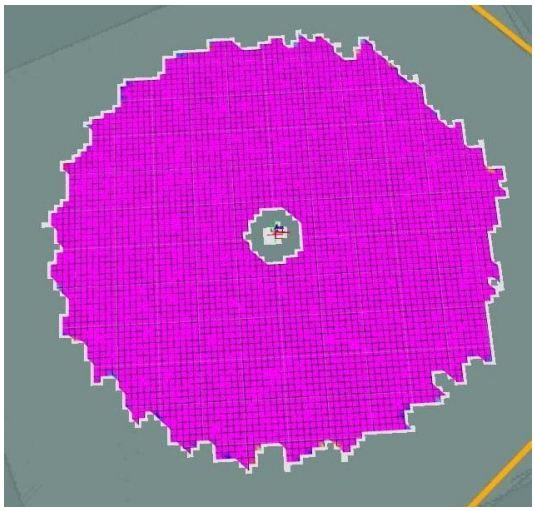


Figura 13. Vista alrededor del robot

La explicación relacionada con la implementación del procesado del mapa se encuentra en el capítulo 4.2 Procesamiento de mapa.

3.3 Navegación

Para la planificación y navegación, se utiliza el método de ventana dinámica (DWA). Esta técnica se encarga de calcular unos comandos de velocidad lineal y angular (v, ω), para planificar la mejor trayectoria del robot hasta alcanzar su destino. Primero crea un abanico de posibles velocidades. Para ello, descarta todo ese conjunto de velocidades que no sean seguras. Es decir, aquellas con las que el robot no sea capaz de detenerse antes de llegar a un futuro obstáculo que pueda haber en esa trayectoria. Después descarta también las que el robot no es capaz de alcanzar en un pequeño intervalo teniendo en cuenta su aceleración. Una vez se dispone de las velocidades posibles que no conducen a colisión, elige la óptima minimizando una función de coste. Esta función da prioridad a trayectorias que vayan más directas hasta el destino, las que estén más alejadas de posibles obstáculos y las que tengan velocidades lineales (v) más altas.

4. Implementación

En este apartado se tratará de la implementación de los distintos métodos necesarios para el correcto funcionamiento. La implementación se ha realizado mediante el sistema operativo ROS [ver Anexo A (ROS)]. Se centrará sobre todo en esos métodos que se consideren más importantes para este trabajo:

4.1 Obtención de la posición en 3D

Para la obtención de la posición del robot en 3D, se ha tenido que modificar el árbol de transformaciones de éste. Se ha explicado de forma conceptual en el capítulo 2.2 Sistemas de referencia y transformaciones y aquí, se verá cómo se ha llevado a cabo en ROS.

Primeramente, todos los nodos que publicaban transformadas del robot se han remapeado (redireccionado, para que no publiquen sobre las transformadas que se quieren crear). Una vez que no se publica ninguna transformada, se tendrá que crear las transformadas que se necesiten para este trabajo. Se tendrá en cuenta la Figura 14. donde se representan las transformadas que se necesitan para el correcto funcionamiento del robot.

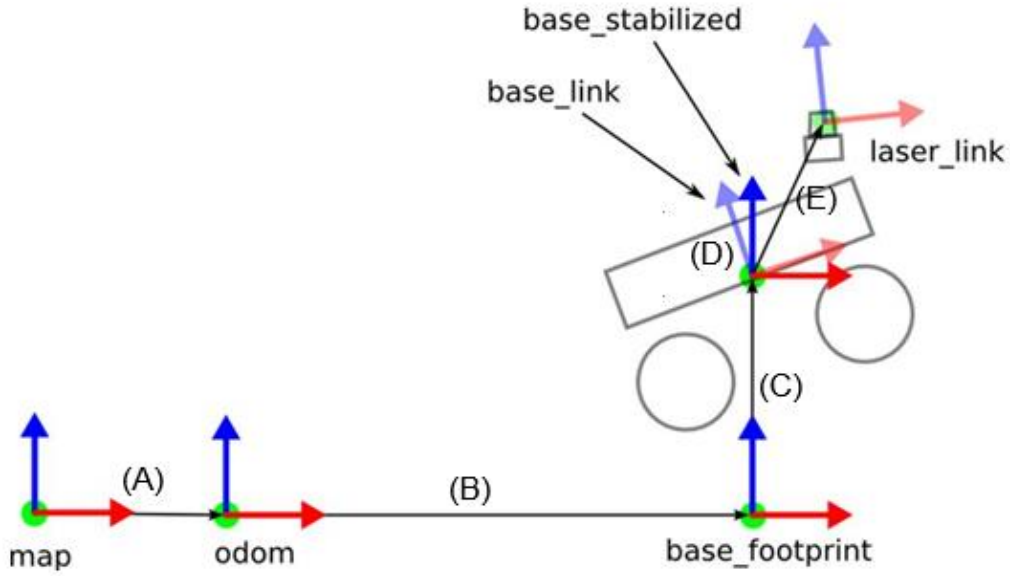


Figura 14. Referencias y transformaciones básicas del robot en ROS para navegación en 3D

(A) La transformada de “map” a “odom”, la realiza el propio sistema de localización utilizado (“hector_mapping”) y representa la corrección que hace este sistema a la información que llega de la odometría del robot.

(B) La transformada de “odom” a “base_footprint”, es la información de odometría del propio robot. Para sacar esta transformada se requiere crear un nodo. Éste se encarga de transformar la información de odometría a una transformada entre estos dos “frames” o referencias (odom y base_footprint). Ésta se actualiza a la vez que lo hace la odometría del robot.

(C) La transformada de “base_footprint” a “base_stabilizer”, es la que da la información de la altura del robot. Esta transformada se ha tenido que crear por medio de un nodo que estime la altura del robot en cada momento. Este nodo se

suscribe a la posición 2D del robot, que da la odometría con la corrección del "hector_mapping". También se suscribe a la información de inclinación del robot que proporciona una unidad de medida inercial (IMU). Con la posición del robot en 2D (x,y,ψ) se estima la distancia que avanza el robot (da), aproximada a una recta. calculada *da* se pasa a hallar la altura. Para esto se hacen varias comparaciones. Si Pitch es 0, el robot no tiene inclinación, y por tanto no aumenta la diferencia de altura (dh). En cambio si no es 0 se haya dh con la fórmula:

dh= abs(da)*sin(-θ)/cos(-θ)

Los signos menos del Pitch (θ), es debido a que una inclinación del (subiendo) corresponde a una inclinación negativa. El valor absoluto, es porque lo que interesa es el valor que recorre el robot y no la dirección en la que lo recorre.

Tanto para el Yaw (ψ) como para el Pitch (θ) se utiliza la media del tramo para tener una estimación más precisa. Una vez calculada dh, se suma a la altura anterior. Y se publica esta altura como la transformada entre "base_footprint" y "base_stabilizer". El código de este nodo se encuentra en el Anexo I (Código utilizado en ROS).

(D) La transformada de "base_stabilizer" a "base_link", es la que le da la información de la inclinación del robot. Esta información la crea un nodo del paquete "hector_mapping", que se encarga de leer la información de la unidad de medida inercial (IMU) y publicarla como transformada entre estos 2 frames ("base_stabilizer" y "base_link").

(E) La transformada de "base_link" a "laser_link" o a cualquier sensor que se necesite, es la que da la información de la localización de los sensores del robot con respecto a la base. Estas son transformadas fijas, ya que en este caso todos los sensores están fijos al robot. Para crear todas estas transformadas se ha creado un nodo que establece las uniones entre estos diferentes *frames* que se necesiten. En este caso se han creado las necesarias hasta llegar a los diferentes sensores:

(E1) La trasformada que relaciona "base_link" a "camera_RGB_frame" y todas sus ramas posteriores, son las que dan la información de la posición de la cámara Asus Xtion Pro Live respecto a la base. Esta referencia es necesaria para crear el mapa de elevación.

(E2) La transformada que relaciona "base_link" a "base_laser_link" con las transformadas intermedias, dan la información de la posición del láser respecto a la base. Esta referencia se utiliza para el "hector_mapping" y la localización.

(E3) La transformada que relaciona "base_link" a "gyro_link", da la información de la posición de la unidad de medición inercial. Esta referencia es muy importante porque luego se utilizará este dispositivo para obtener la inclinación del robot.

Tras la creación de estos nodos, las transformadas del robot quedan como se ve en las figuras: 1.**Figura** 15, que es la parte superior del árbol de transformadas. 2. Figura 16, que corresponde a la parte inferior izquierda. 3.Figura 17 que es la parte inferior derecha. Aquí se ve que además de las mencionadas anteriormente hay una transformada más:

(F) La transformada que relaciona “map” a “scanmatcher_frame”, que es una transformada que crea el “hector_mapping” y dá la información de posición del robot simplemente con la información que detecta el sensor láser.

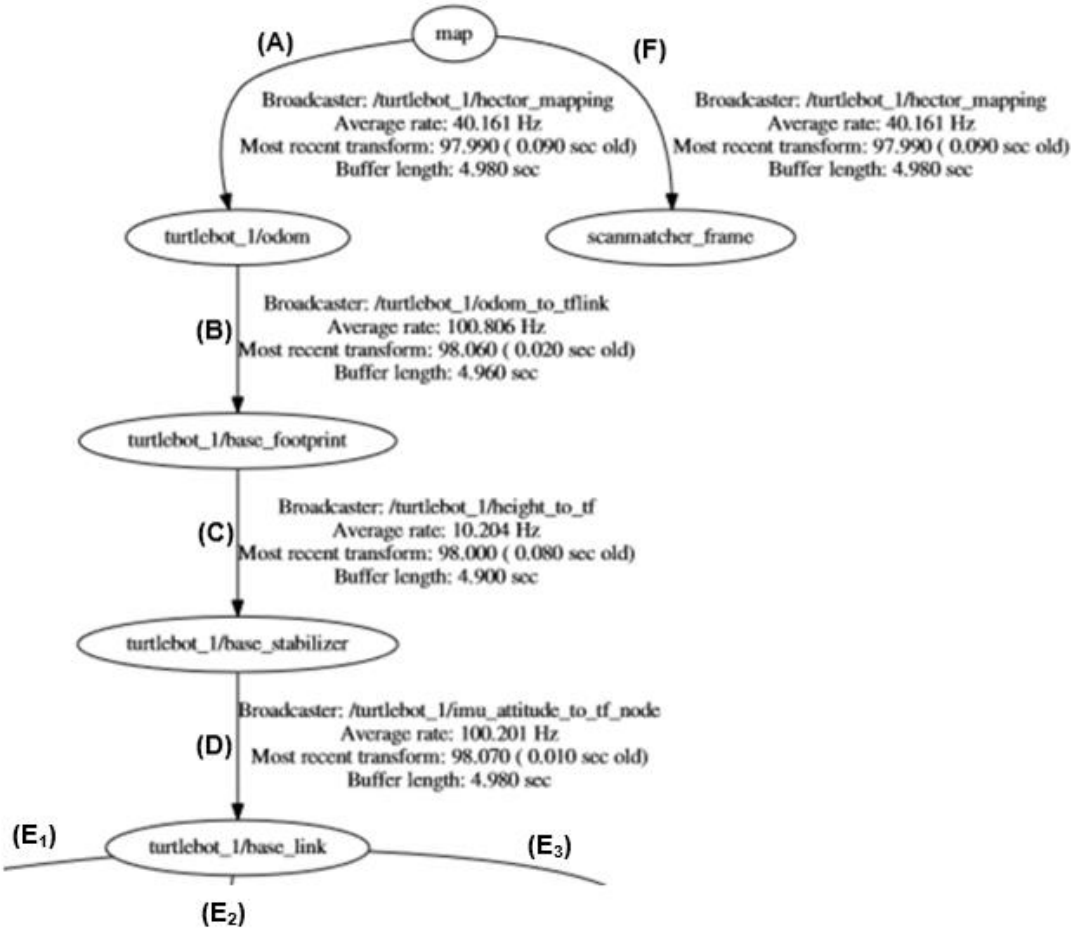


Figura 15. Transformadas del robot a

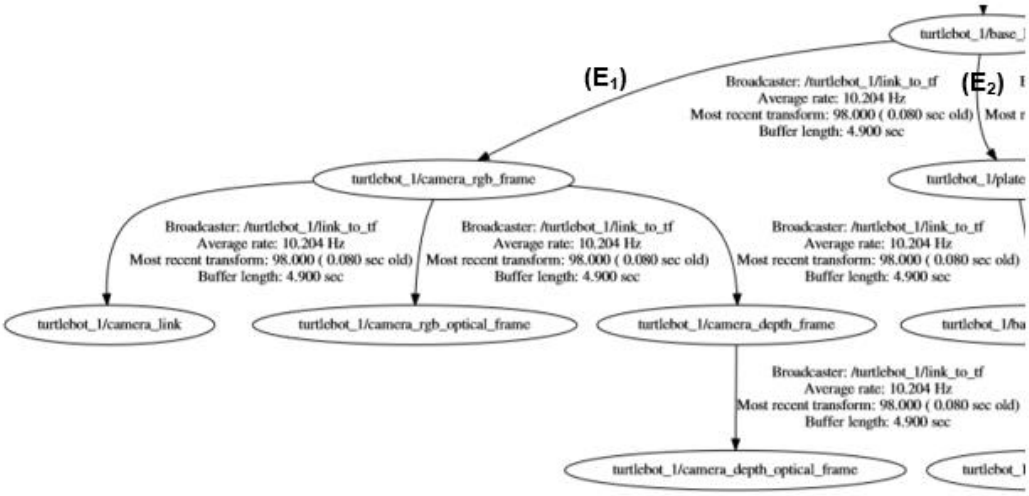


Figura 16. Transformadas del robot b

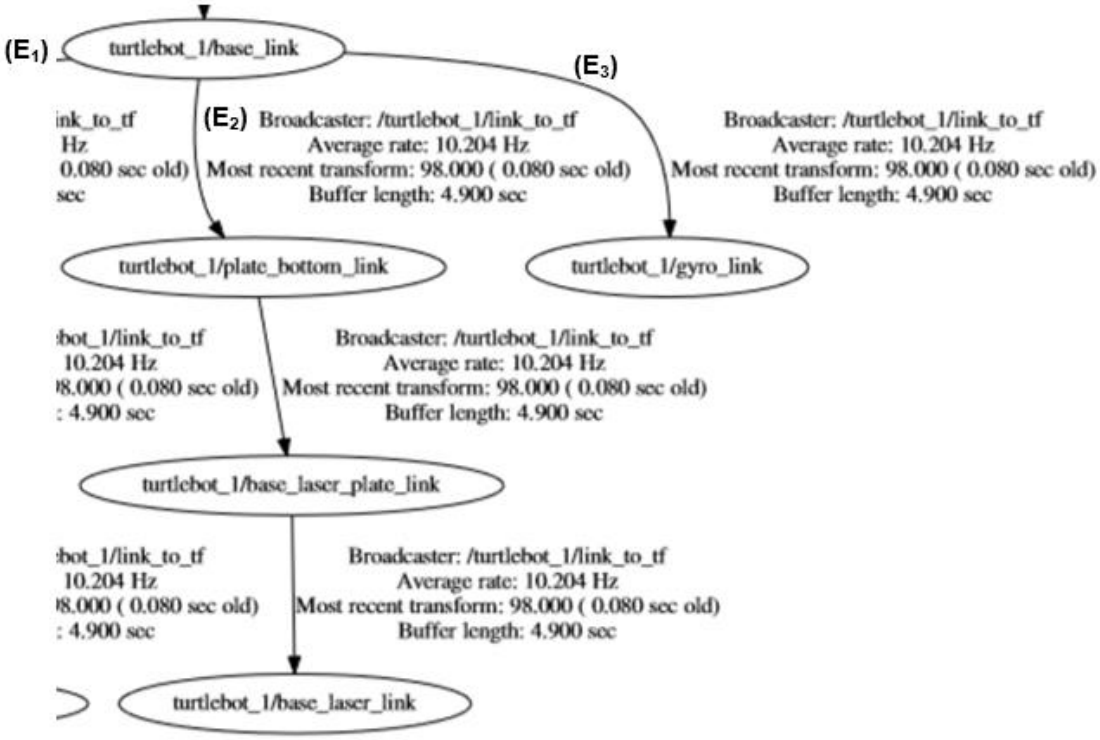


Figura 17. Transformadas del robot c

Una vez definidas todas las transformaciones es necesario obtener la posición en 3D para el “elevation_mapping”. Para esto se crea un nodo que se suscribe a la transformada entre “map” y “base_link” y la transforma a un mensaje del tipo “PoseWithCovariance” (con la posición(x, y, z, ϕ, θ, ψ) y una matriz de covarianza, que representa el error de localización). Después de realizar esta transformada, publica la posición para que el “elevation_mapping” se pueda suscribir. El código utilizado para implementar este nodo se encuentra también en el Anexo I (Código utilizado en ROS).

4.2 Procesamiento de mapa

En este apartado se explica la implementación que se ha llevado a cabo en ROS para el procesamiento del mapa, que se ha descrito de forma conceptual en el capítulo 2.2 Sistemas de referencia y transformaciones

Se ha definido un nodo que se suscribe al mapa de elevación, que obtiene toda la información de ese mapa. Este mismo nodo se encarga de definir un mapa de obstáculos, con las mismas características (resolución, tamaño de mapa, celda actual...) que el de elevación. Después se establecen algunos parámetros como el valor máximo y mínimo de altura o el rango del mapa de ocupación que va de 0 (espacio transitable) a 100 (obstáculo) y utiliza el -1 como espacio desconocido. También se establece el valor máximo que se quiere tener como diferencia entre vecinos para que se considere obstáculo o no. Una vez ya se tienen los parámetros se crea un bucle con un *iterador* que recorre cada punto del mapa hasta llegar al límite del mapa.

Primero se extrae a partir del *iterador* la posición de la celda en el mapa. Una vez se tiene la posición de la celda se extrae el valor de altura que tiene. Si el valor es desconocido y la celda está alrededor del robot (0,5m), al principio se etiqueta como valor transitable para que pueda comenzar a moverse el robot en la zona sin visibilidad. Un ejemplo de como se ve el cuadrado blanco de zona inicial transitable, con lo que ve el robot en rosa se puede ver en la Figura 18.

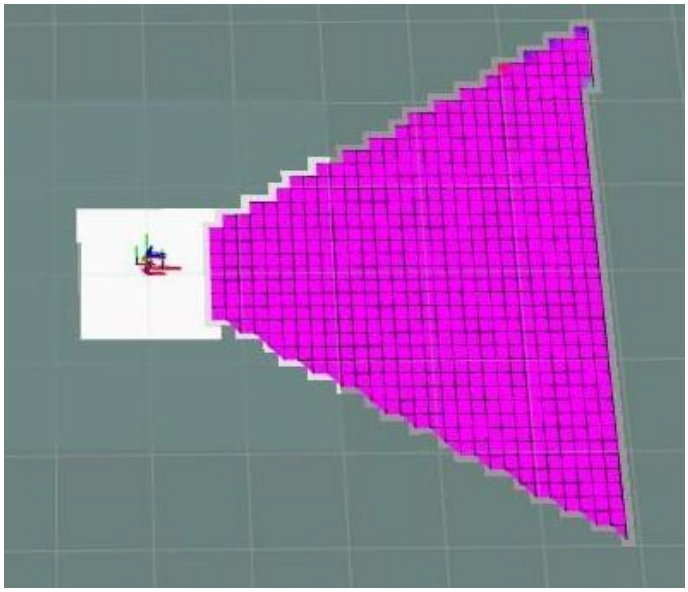


Figura 18. Vista mapa de elevación con cuadrado inicial

Si el valor de la celda es desconocido, pero no es esa zona inicial, se da valor desconocido (-1). Si la celda tiene valor, ese valor se escala a un rango de 0 a 100 que es cómo funciona el mapa de ocupación, siendo 0 el valor más bajo en los límites que se utilizan y 100 el más alto. Una vez ya se tiene el valor en el rango deseado, se compara si existen los vecinos de esa celda (arriba, abajo, derecha e izquierda). Si alguno no existe es porque está en los límites del mapa y hay que comprobarlo, para no intentar acceder a celdas que no existen, porque daría error. Si los vecinos existen, se comprueba si tienen un valor o son desconocidos. En caso de tener un valor, se normaliza en el rango deseado,

como el valor anterior. Se calcula la diferencia en valor absoluto entre el valor de la celda y los diferentes vecinos y si alguno supera el valor máximo que se ha puesto se considera obstáculo. Si no sobrepasa ese valor con ningún vecino se considera espacio transitable. Con esto se sabe si esa celda es obstáculo, transitable o desconocida. Este valor se traslada al mapa definido antes, en la posición que corresponde a su celda. Este proceso se repite con todas las celdas que se van recorriendo. Al final se obtiene el mapa completo de obstáculos, y este mapa se publica un *topic* para que otros nodos se puedan suscribir a él. Por ejemplo en la Figura 19 se representa un escenario donde hay una rampa de subida, unas plataformas elevadas y una rampa de bajada.

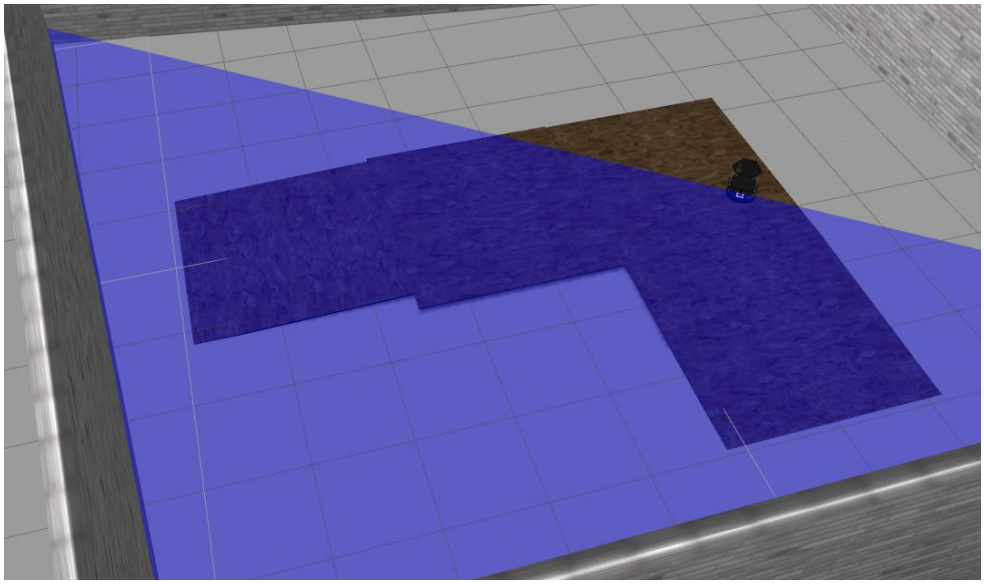


Figura 19. Escenario1

En la Figura 20 se ve el mapa de elevación de este escenario, remarcando con diferente color las diferencias de alturas.

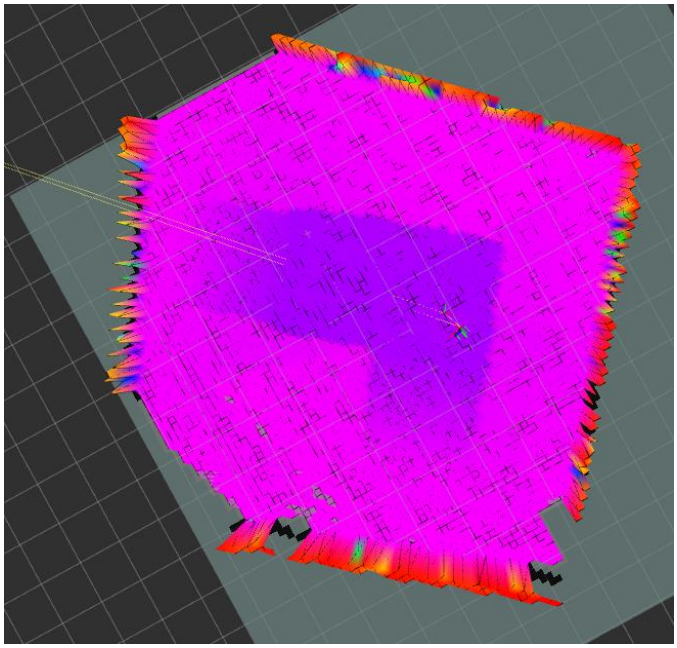


Figura 20. Mapa de elevación del escenario1

Tras el procesamiento de este mapa de elevación, con el nodo descrito anteriormente, se obtiene el mapa de obstáculos que muestra la Figura 21. Las paredes laterales que ha visto, se etiquetan como obstáculo (negro), las caídas que hay por las plataformas, que es un escalón que el robot no puede pasar, también se etiquetan como obstáculo. Sin embargo, las rampas se etiquetan como zona transitable ya que la diferencia de altura es pequeña y por tanto si puede pasar. También se ve que la zona que no ve detrás de las paredes queda como zona desconocida (verde).

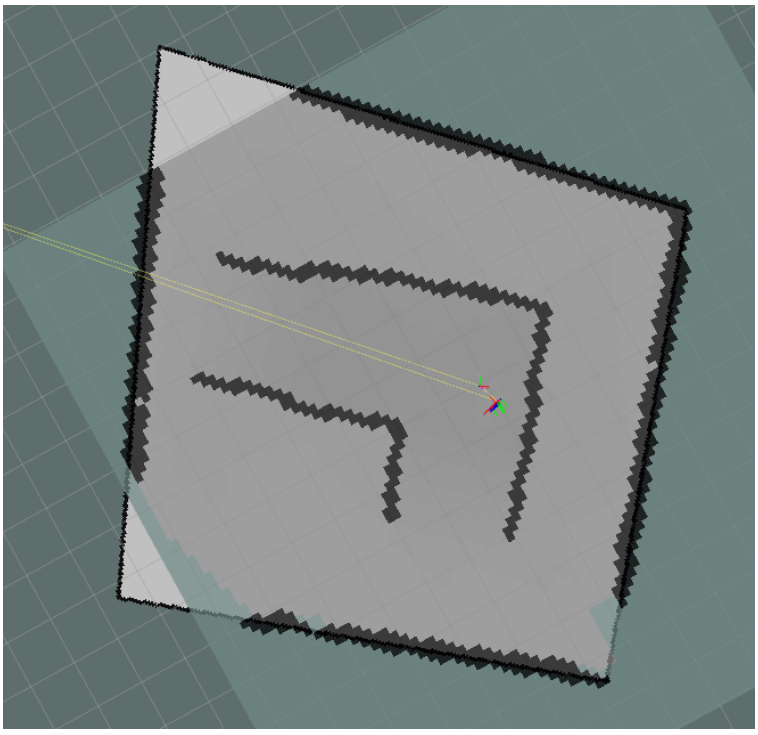


Figura 21. Mapa de obstáculos escenario 1

5. Nodos ROS

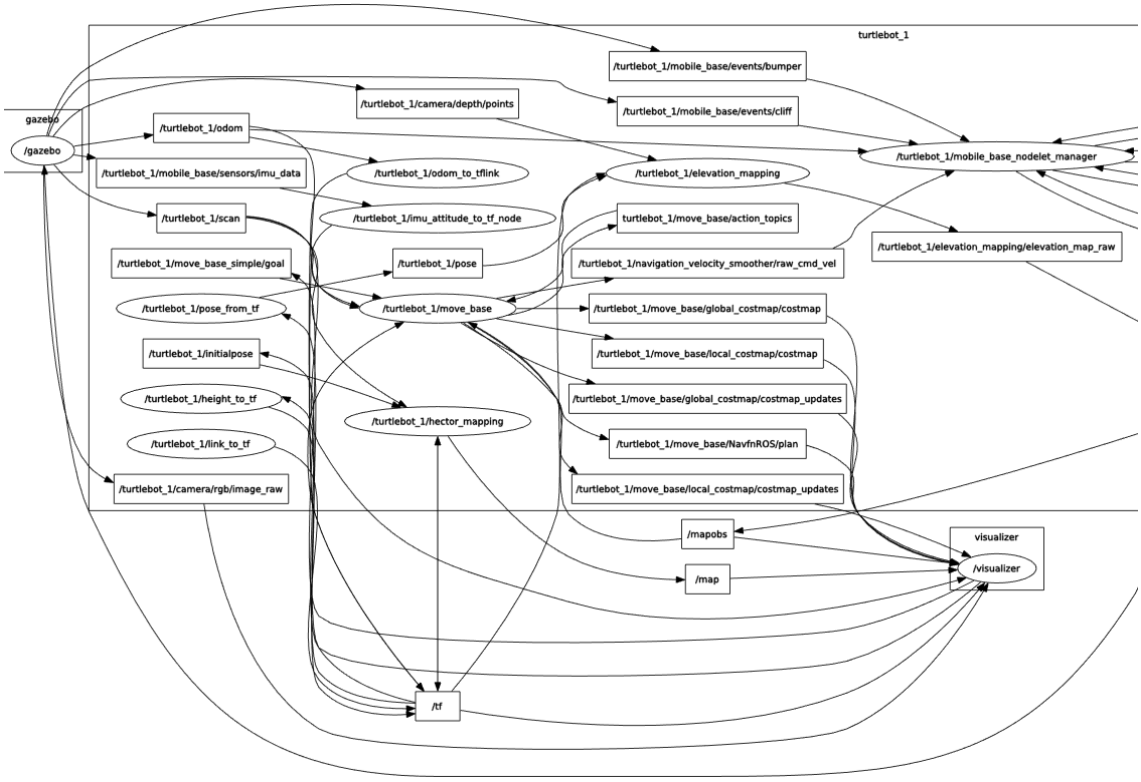


Figura 22. Nodos ROS izquierda

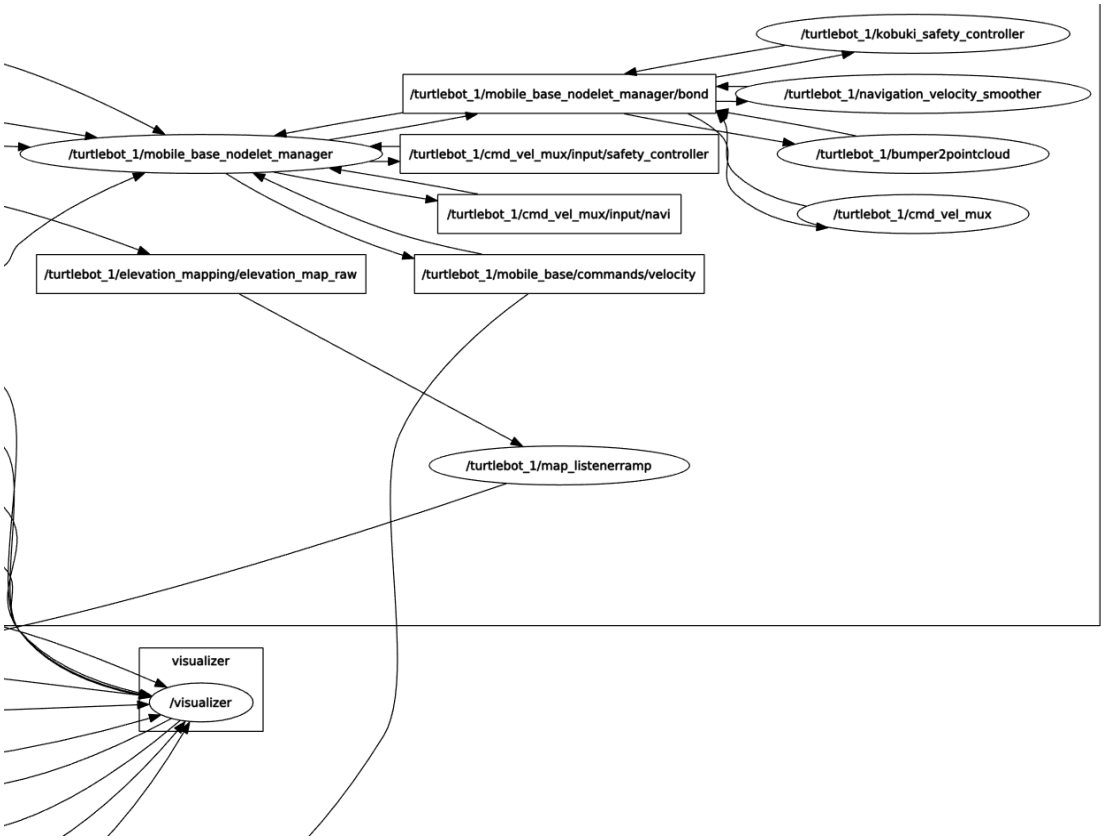


Figura 23. Nodos ROS derecha

En la Figura 22 y Figura 23 se ven los nodos que están en ejecución, representados dentro de una elipse. Y los topics; representados dentro de un rectángulo. En este capítulo se habla de los nodos, para un mayor detalle sobre los topics ver Anexo H (Topics ROS).

Los nodos son los que se encargan de realizar los procesos y estarían divididos en:

-**Nodos de simulación y visualización:**

A. Gazebo: es el simulador que se está utilizando.

B. Visualizer: es el visualizador para poder ir viendo todo lo que se requiera.

-**Nodos encargados de crear las transformadas entre las distintas partes del robot:**

C. link_to_tf crea todos los links fijos de base_link a los diferentes sensores que se necesiten.

D. odom_to_tf_link: transforma la odometría en una transformada entre odom y base_footprint.

E. imu_attitude_to_tf_node: a partir de la información del IMU es el que crea la transformada entre base_link y base_stabilizer.

F. height_to_tf: es el nodo que se encarga de estimar la transformada de altura que se representa entre base_footprint y base_stabilizer.

G. pose_from_tf: este nodo se encarga de a partir de las transformadas del robot pasar la posición de éste en cada momento.

-**Nodos de navegación:**

H. move_base: es el encargado de la planificación. A este nodo se le puede pasar un destino y por medio de los mapas de ocupación que tiene, crea la trayectoria óptima y manda una consigna de velocidad.

I. mobile_base_nodelet_manager: este nodo se encarga de recibir la consigna de velocidad deseada y hacer que el robot navegue a esa velocidad.

-**Nodos de seguridad:**

J. kobuki_safety_controller: se encarga de que la navegación sea segura, teniendo en cuenta sobre todo posibles fallos en las ruedas.

K. navigation_velocity_smoother: este nodo se encarga de limitar las velocidades y aceleraciones del robot.

L. bumper2pointcloud: pasa la información del bumper a una nube de puntos, para crear un obstáculo cuando el bumper choca y que no siga intentando ir contra él.

M. cmd_vel_mux: este nodo se encarga de que si llegan 2 consignas de velocidades elige la adecuada para el robot.

-**Nodos de creación de mapas:**

N. hector_mapping: es el nodo que se encarga de crear el mapa con el láser y localizarse en él.

O. elevation_mapping: es el nodo que se encarga de crear el mapa de elevación a partir de la información de la cámara Asus Xtion Pro Live.

P. map_listenerramp: es el nodo que transforma ese mapa de elevación en un mapa de obstáculos, para poder planificar sobre él la trayectoria del robot.

6. Evaluación experimental

En este capítulo se presentan y evalúa el método en base a diversas pruebas para analizar el comportamiento. Se explica brevemente el motivo de cada prueba y se valoran los resultados obtenidos. Las pruebas que se han considerado más importantes se dividen en:

6.1 Localización

En esta prueba se valora la calidad de localización que tiene la técnica utilizada en el robot. En este caso el "hector_mapping". Éste se detecta mientras va creando el mapa de obstáculos por medio de barridos del láser. Por tanto, para comprobar su funcionamiento se crea un escenario. Se ira recorriendo con el robot para que cree su mapa. Una vez tenga el mapa creado se comprobara si lo que detecta el sensor láser coincide con los obstáculos que se habían creado del mapa. Si el robot tiene una mala localización es posible que cree el mismo obstáculo en sitios cercanos pero diferentes. En cambio, si la localización funciona correctamente el láser tendría que detectar el obstáculo siempre en el mismo sitio.

En la Figura 24 se ve el escenario inicial en el simulador Gazebo. Tras recorrer el mapa con el visualizador, se ve en la Figura 25 que en el mapa láser los obstáculos coinciden exactamente donde se han colocado inicialmente. Se ve la posición del robot remarcada por un círculo azul alrededor. Se puede considerar que la localización es bastante precisa, al menos en lugares cerrados donde tenga obstáculos para localizarse. Está claro que si no hay ningún obstáculo para localizarse, es imposible la localización. En este caso la localización se basará exclusivamente en la odometría.

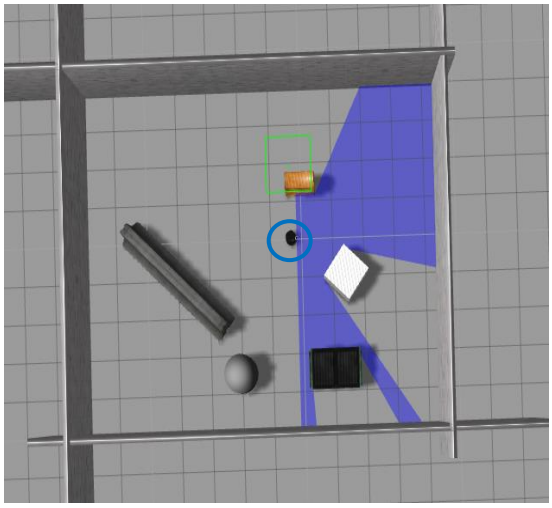


Figura 24. Escenario 2

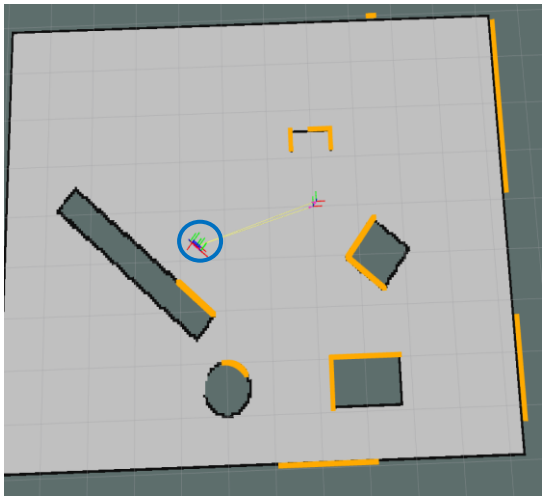


Figura 25. Prueba de localización

6.2 Detección de obstáculos

En este punto se harán diversas pruebas para comprobar la detección de los diferentes tipos de obstáculos. Estas pruebas valorarán la calidad de detección de la cámara Asus Xtion Pro Live para crear el mapa de elevación y también el procesamiento llevado a cabo para crear el mapa de obstáculos:

1. Primera prueba: Detección de obstáculos simples. Se tomara para esta prueba el mapa por defecto que tiene el simulador, el cual se puede ver en la Figura 24 Se recorrerá con el robot y se verá el mapa tanto de elevación como de obstáculos que ha ido creando. Posteriormente se comparan estos para evaluar la corrección de la detección de los obstáculos.

Tras realizar esta prueba, en la Figura 26 se representa el mapa de elevación que ha creado y se ven los obstáculos representados en la Figura 24, que era el mapa original. En este mapa de elevación, se representa en rosa el suelo liso, sin alturas. Los obstáculos se representan en colores, dependiendo su altura (siguiendo la escala de alturas adjunta). En la Figura 27, se muestra el mapa de obstáculos creado a partir del mapa de elevación. En este mapa se ven los obstáculos en negro oscuro, en gris claro las zonas que son transitables y luego en gris más verdoso las zonas desconocidas.

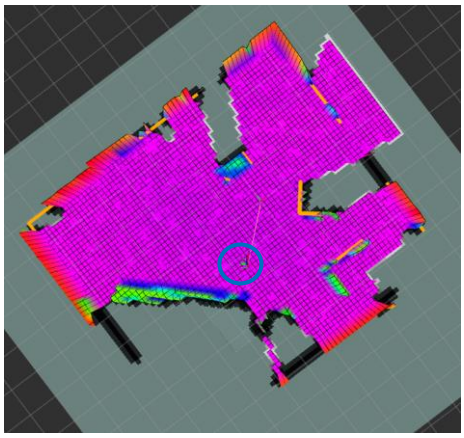


Figura 26. Mapa de elevación escenario 2

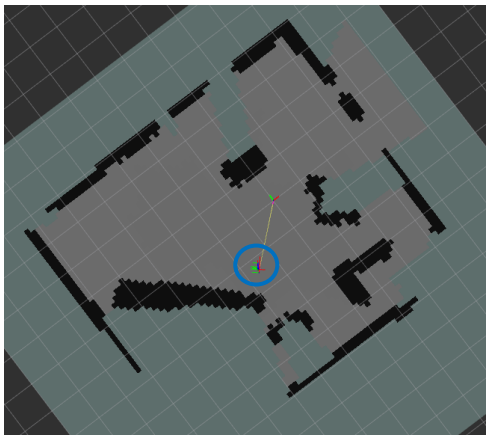


Figura 27. Mapa de obstáculos escenario 2

2. Segunda prueba: Detección de obstáculos bajos. En esta prueba se incluyen obstáculos bajos, como un pequeño escalón. En este caso se utilizará el mapa anterior añadiéndole un pallet como se ve en la Figura 28. El robot recorrerá el mapa construyendo el mapa de obstáculos. En esta prueba se compara también el mapa visto con el “elevation_mapping”, con el mapa que crearía un sensor láser.

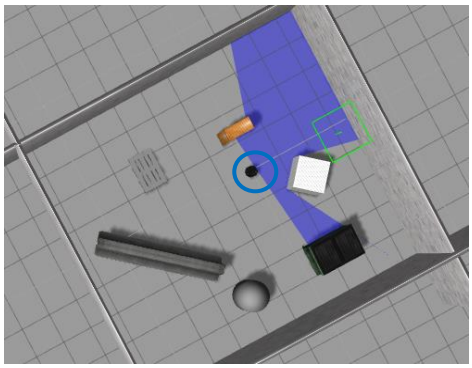


Figura 28. Escenario 3

En la Figura 29 se ve el mapa de elevación creado después de que el robot recorriese el mapa. Como se ve, detecta el pallet como obstáculo. En la Figura 30 se verifica que detecta bien los obstáculos, incluso el pallet. Sin embargo, en la Figura 31 en el mapa de obstáculos que crea el sensor láser no aparece el pallet. Esto se debe a que el funcionamiento del sensor láser es más simple, ya que solo detecta los obstáculos que estén a cierta altura (la que se encuentre el láser colocado). Por tanto no es capaz de detectar obstáculos que sean bajos, como este caso, ni obstáculos que estén por encima del láser. Así se puede concluir este experimento de forma satisfactoria. Se ve que con el mapa de elevación es capaz de detectar correctamente incluso pequeños obstáculos, que no detectaría navegando con el sensor láser.

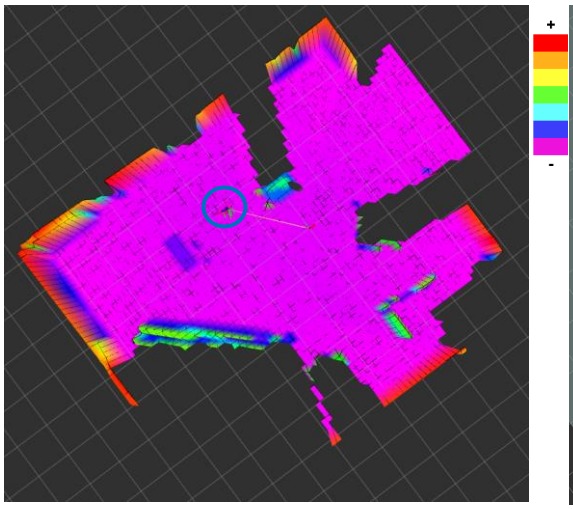


Figura 29. Mapa de elevación escenario3

Figura 30. Mapa de obstáculos escenario 3 por medio del "elevation_mapping"

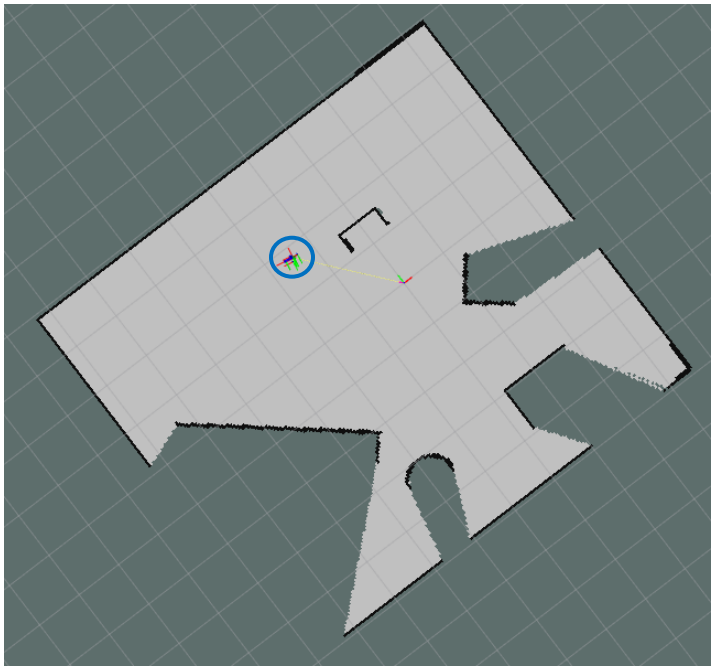


Figura 31. Mapa de obstáculos escenario 3 por medio del láser

3. Tercera prueba: Detección de rampas según la pendiente. Uno de los obstáculos más complicado serán las rampas. Ya que hay diferentes tipos de rampas y unas serán transitables y otras no. Además esto dependerá no solo de la inclinación de la rampa sino también de las características de cada robot. Para ello, tomando los parámetros del turtlebot (que no puede subir rampas muy inclinadas) se creara un escenario con 2 rampas diferentes como se ve en la Figura 32. La rampa de la derecha es de poca pendiente y debería considerarla transitable, mientras que la de la izquierda tiene más pendiente, y para este robot sería un obstáculo. En la Figura 33 se ve como se ha creado el mapa de elevación, donde sí se ve una clara diferencia entre ambas rampas. La de la derecha casi no tiene inclinación y la de la izquierda tiene una pendiente más fuerte. Después del procesamiento se visualiza en la Figura 34 como reconocería cada rampa. La de la izquierda la considera toda ella como un obstáculo. En cambio la de la derecha la considera como zona transitable en la dirección de la rampa. Sin embargo, los lados y el final, que son escalones, los etiqueta como obstáculo. Esto se verá mejor en la próxima prueba experimental. Por tanto, tras realizar estas pruebas se ve que las rampas no son un problema para la detección de obstáculos.

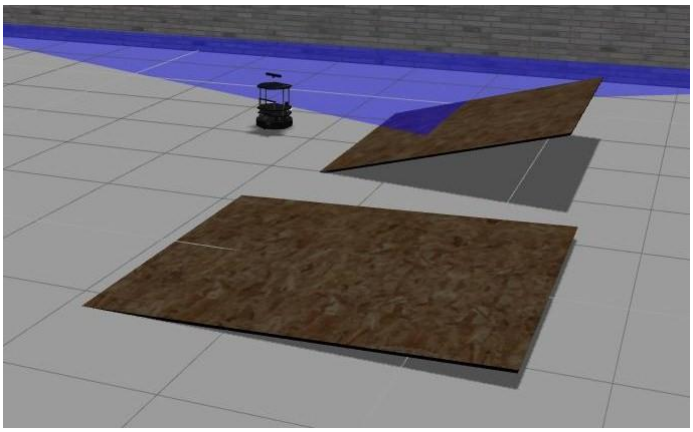


Figura 32. Escenario 4

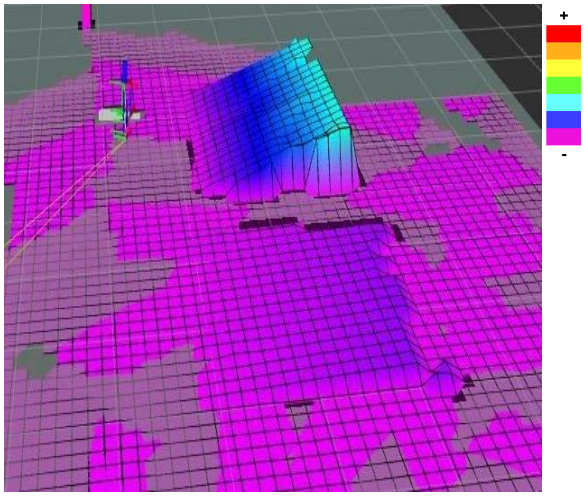


Figura 33. Mapa de elevación escenario 4

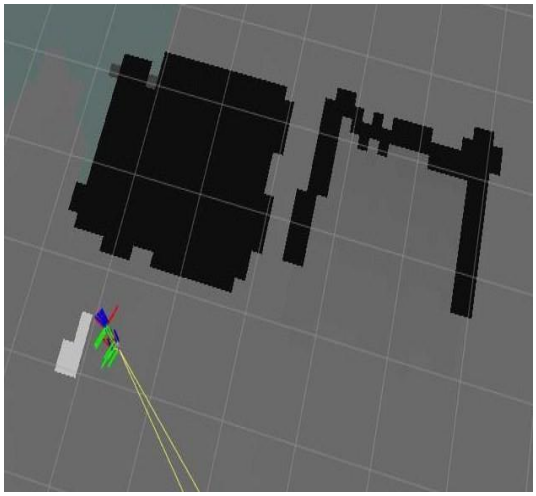


Figura 34.Mapa de obstáculos escenario 4

4. Cuarta prueba: Detección de caídas libres en rampas. Se pretende hacer otra prueba experimental para comprobar su funcionamiento frente a las caídas libres en las rampas. Sería el caso cuando el robot esté subiendo una rampa, o cuando esté encima de un obstáculo elevado. Se comprueba si se detectan bien las posibles bajadas, y evita lo que podría ser precipicios o caídas libres. Para ello se crea un escenario como el de la Figura 35 donde el robot se encuentra en una plataforma elevada. Existe solo una posible rampa para bajar, y el resto alrededor de la plataforma será un desnivel que el robot no es capaz de bajar.

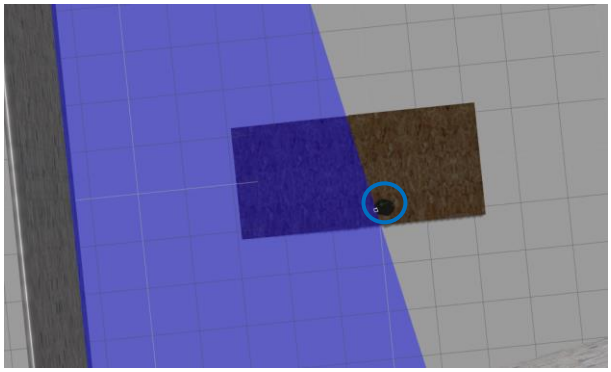


Figura 35. Escenario 5

Tras estas pruebas se ve en la Figura 36 el mapa de elevación que ha creado el robot. Se ve que las zonas de caída libre no las detecta bien, y se etiquetan como desconocidas en algunos tramos. Por tanto, al procesar esta información queda un mapa de obstáculos como el de la Figura 37, En el que se ve bien la rampa, pero no es capaz de distinguir bien las caídas libres. De quedarse así, cuando trate de plantear la navegación, no podrá pasar por las zonas desconocidas que son caídas. Al ser desconocidas, el robot se queda detenido frente a ellas intentando reconocerlas, y nunca llegará a conocerlas. El robot se detendría definitivamente en esa situación.

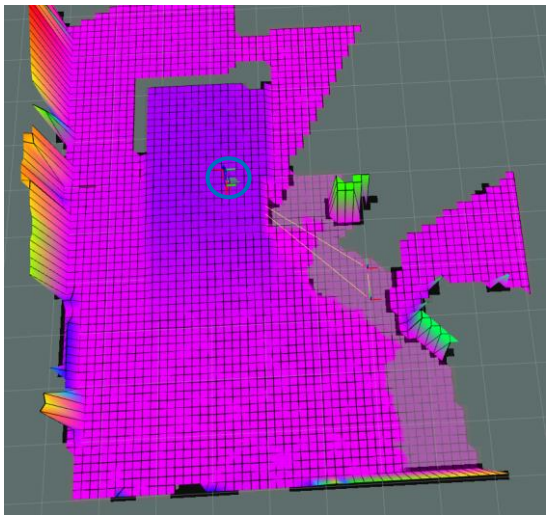


Figura 36. Mapa de elevación en escenario 5

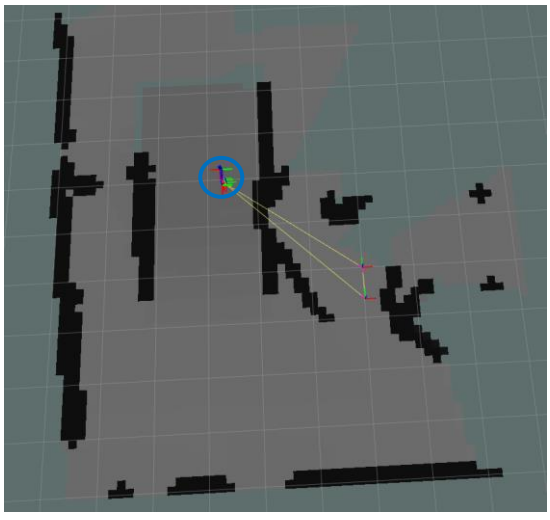


Figura 37. Mapa de obstáculos en escenario 5

Tras esta prueba, se decidió reubicar la cámara Asus Xtion Pro, en una posición más elevada en el robot, y con un pequeño ángulo Pitch hacia abajo, como se ve en la Figura 39. En la Figura 38 se representa la colocación anterior con la que se había realizado las pruebas hasta este momento.

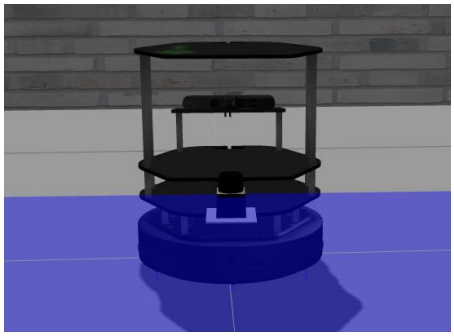


Figura 38. Posición de la cámara robot inicial

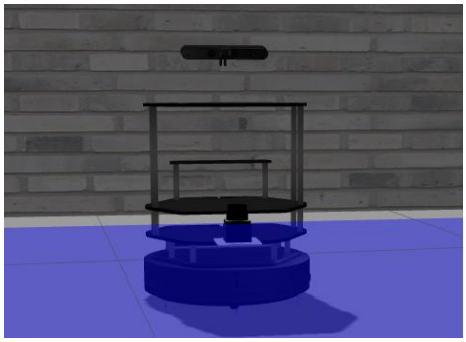


Figura 39. Posición de la cámara robot final

Con la nueva posición de la cámara se vuelve a repetir la última prueba, en el mismo escenario de la Figura 35. Se ve en la Figura 40 como el robot sí que es capaz esta vez de crear correctamente el mapa de elevación. En la Figura 41 se muestra el resultado del mapa de obstáculos. En este caso sí que distingue correctamente las caídas libres y deja como único camino transitable la rampa.

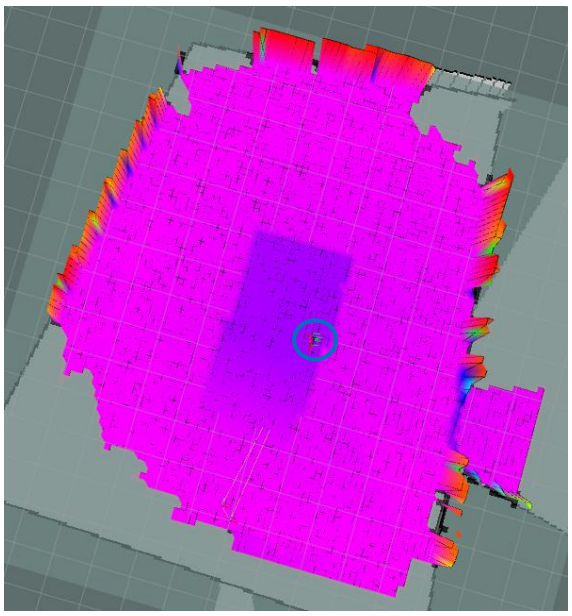


Figura 40. Mapa de elevación b escenario 5

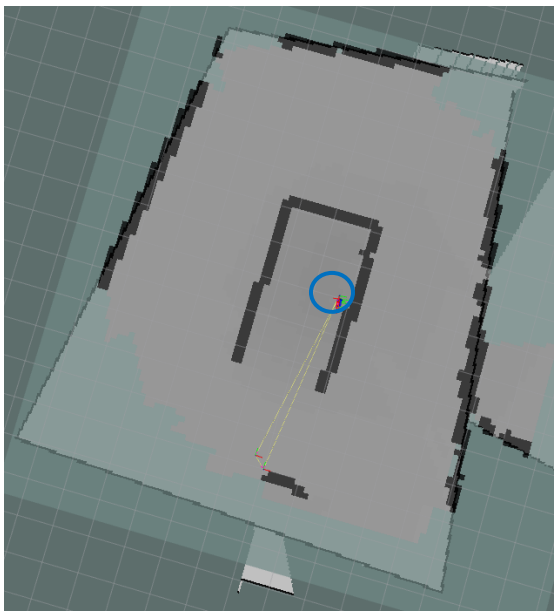


Figura 41. Mapa de obstáculos b escenario 5

5. **Quinta prueba: Asegurar detección caídas libres en rampas.** Se va a realizar otra prueba con el escenario de la Figura 42 que es algo más complejo. Se tiene en este caso un mapa que consta de unas plataformas elevadas y dos rampas. Las rampas deberían ser las únicas zonas de bajada o subida a la plataforma.

El mapa de elevación creado se representa en la Figura 43 donde se distingue, las dos rampas y la información de las caídas libres alrededor de las plataformas y en los lados de las rampas. Este procesamiento se puede ver en la Figura 44, creando un mapa de obstáculos con toda la información necesaria para poder navegar correctamente por él.

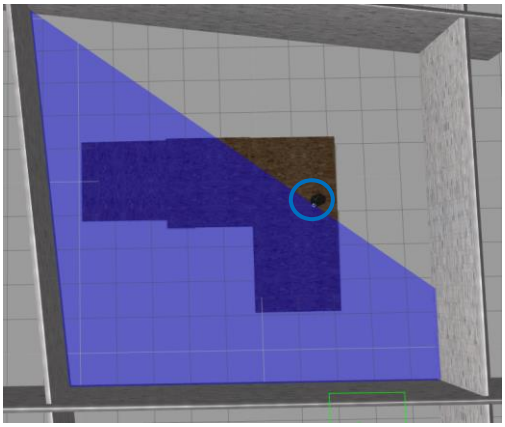


Figura 42. Escenario 6

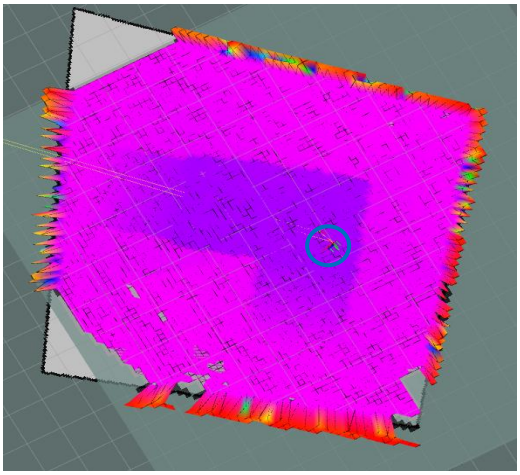


Figura 43. Mapa de elevación escenario6

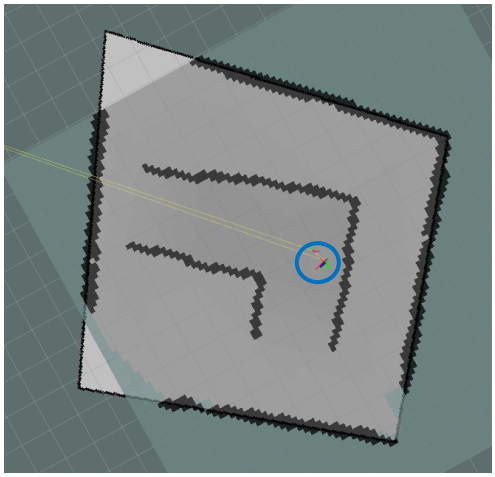


Figura 44. Mapa de obstáculos escenario6

6.3 Planificación de trayectoria y navegación

En este punto se pondrá a prueba la planificación del robot y la navegación en el mapa de obstáculos. La navegación del robot como se explica en el capítulo 3.3 Navegación, funciona con la ventana dinámica. Esta prueba se realizará en dos escenarios diferentes:

1. Primera prueba: Navegación en 2D. Para esta primera prueba se utiliza un escenario plano con diversos obstáculos como se ve en la Figura 45.

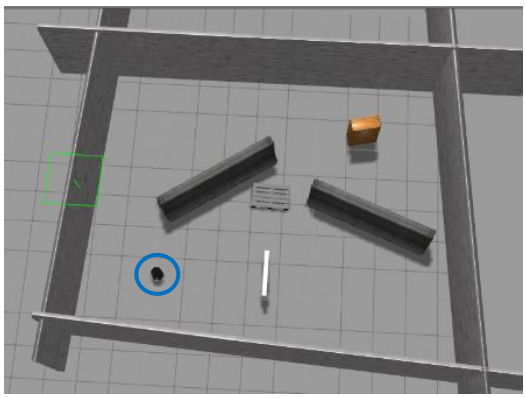


Figura 45. Escenario7

Se define un destino, como se ve en la Figura 46 y el robot planifica la trayectoria hasta el destino con los obstáculos detectados. Luego, el robot avanza siguiendo la trayectoria deseada, detectando los diferentes obstáculos. Cuando detecta alguno que interfiere su trayectoria, replanifica, como se ve en la Figura 47 al detectar el pallet (remarcado con una elipse roja alrededor). Éste impide trazar una trayectoria recta hasta el objetivo. Ha trazado otra trayectoria con los nuevos obstáculos. En la Figura 48, detecta los obstáculos que le impiden el paso por el lado derecho del escenario y por tanto, ve que no hay camino por ahí y vuelve a planificar por el lado contrario. En la Figura 49 planifica otra trayectoria realizable, para pasar más alejado del obstáculo, y poder así desplazarse a más velocidad. En la Figura 50 se detiene porque el destino que se le había dado al principio es un obstáculo, y por tanto inalcanzable. En cambio en la Figura 51 , que es el mismo mapa pero sin el obstáculo final el robot llega hasta el destino final correctamente porque es zona transitable.

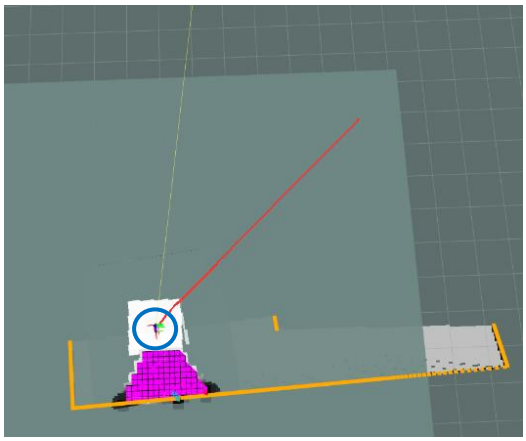


Figura 46. Mapa de elevación 1, escenario7 Trayectoria planificada a destino (línea roja)

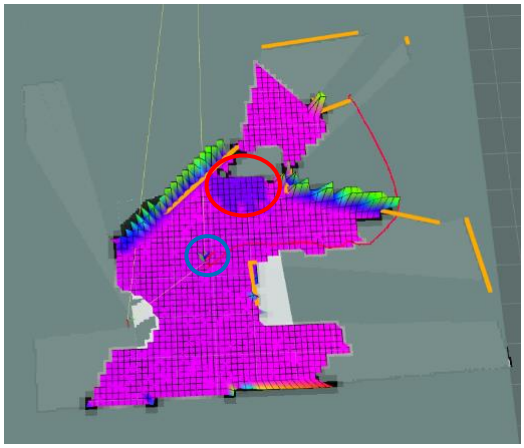


Figura 47. Mapa de elevación 2, escenario7

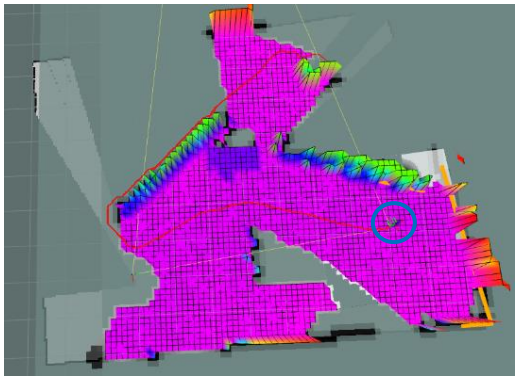


Figura 48. Mapa de elevación 3, escenario 7

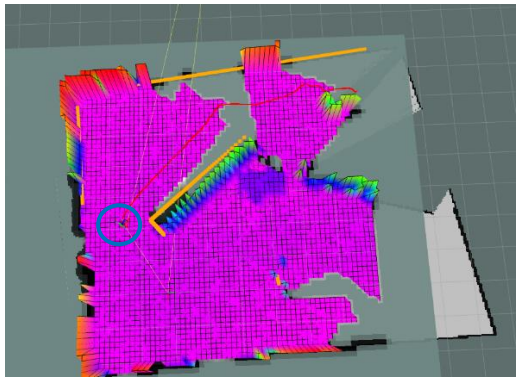


Figura 49. Mapa de elevación 4, escenario 7

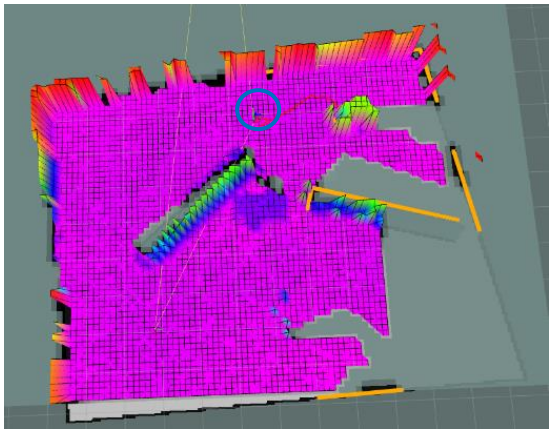


Figura 50. Mapa de elevación 5, escenario7

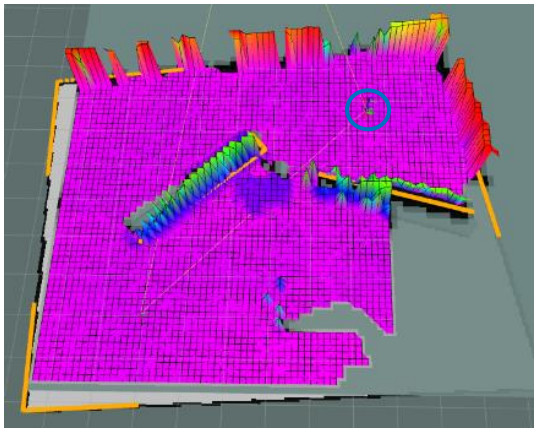


Figura 51. Mapa de elevación 6, escenario7

2. Segunda prueba: Navegación en 3D.En esta prueba, se utiliza un escenario con rampas, para ver cómo se comporta el robot frente a la diferencia de alturas (Figura 52). En la Figura 53 se ve una primera planificación de la trayectoria antes de detectar ningún obstáculo.

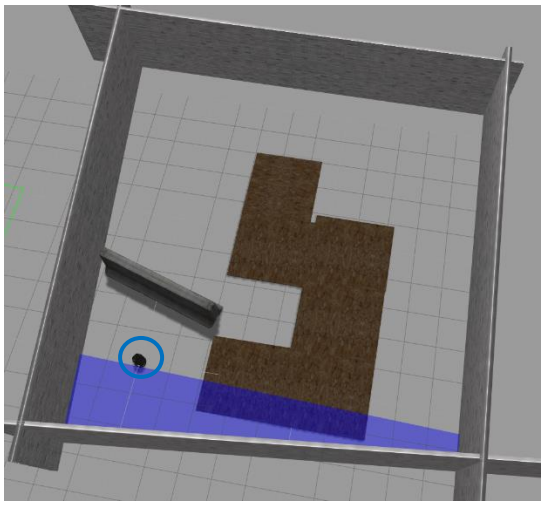


Figura 52. Escenario 8

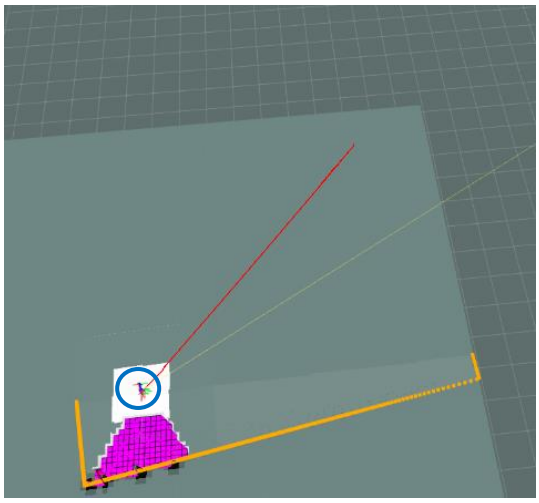


Figura 53. Mapa de elevación 1, escenario 8

Una vez detectada la rampa y los obstáculos, en la Figura 54 hace una primera replanificación. Manda al robot subir la rampa, y posteriormente continuar directo hasta el objetivo. En la Figura 55 se ve cómo el robot detecta correctamente las caídas del contorno desde la plataforma y replanifica un camino alternativo. Este camino pasa por la rampa que ha empezado a ver y que luego ve completamente, etiquetándola como transitable. Al final, en la Figura 56 llega a su destino, habiendo detectado correctamente las rampas y las caídas. En la Figura 57 se puede ver el mapa de obstáculos que ha ido creando el robot y que ha utilizado para desplazarse por las zonas que considera transitables. Además, en esta prueba de navegación en rampas se ve que la estimación de la altura es bastante precisa, ya que mantiene bien la referencia de localización al subir o bajar y sigue creando el mapa correctamente.

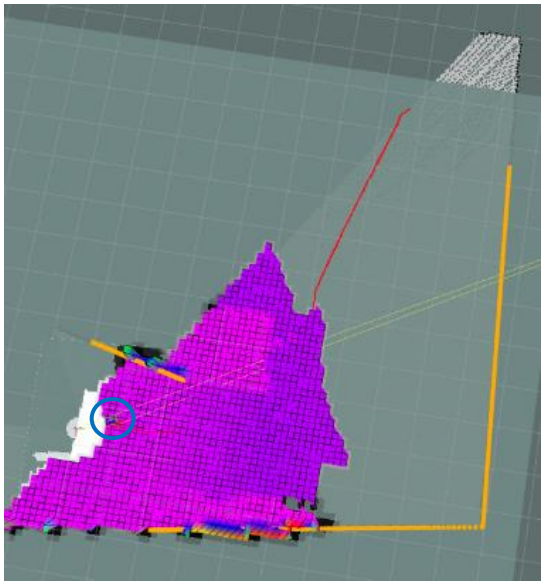


Figura 54. Mapa de elevación 2, escenario 8

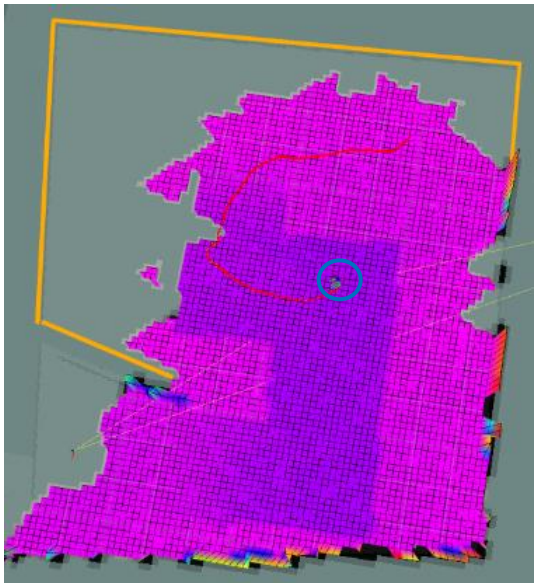


Figura 55. Mapa de elevación 3, escenario 8

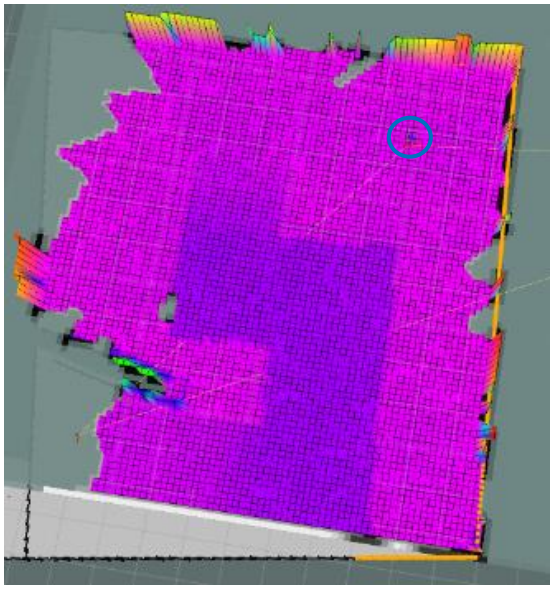


Figura 56. Mapa de elevación 4, escenario 8

Figura 57. Mapa de obstáculos, escenario 8

6.4 Pruebas con el robot

En este punto se pondrá a prueba el comportamiento del robot. Se comprobará el funcionamiento del robot después de su puesta a punto y si hay muchas diferencias con respecto a las simulaciones realizadas anteriormente. Se realizaran 2 tipos de pruebas diferentes en el laboratorio de robótica del edificio I3A (Instituto de Investigación de Ingeniería de Aragón) de la Universidad de Zaragoza.

1. Primera prueba con el robot: Detección de diferentes obstáculos. En esta prueba se va a comprobar si el robot detecta y etiqueta los obstáculos más problemáticos correctamente. Para ello primero se coloca un pequeño obstáculo delante del robot. En la Figura 58 se ve lo que está viendo el robot por medio de la cámara RGB. Se ve que hay un pequeño estuche (remarcado con una elipse roja alrededor), el cual es muy pequeño y podría no detectarlo, pero para este

robot se debería considerar obstáculo, porque no puede sobrepasar pequeños escalones, ni obstáculos de poca altura. En la Figura 59 se ve el mapa de obstáculos que ha creado, a partir del mapa de elevación, y se puede comprobar que detecta correctamente el estuche como un pequeño obstáculo (remarcado con una elipse roja alrededor). Se ve la posición del robot rodeado por un círculo azul.

Figura 58. Cámara RGB del robot, obstáculo bajo

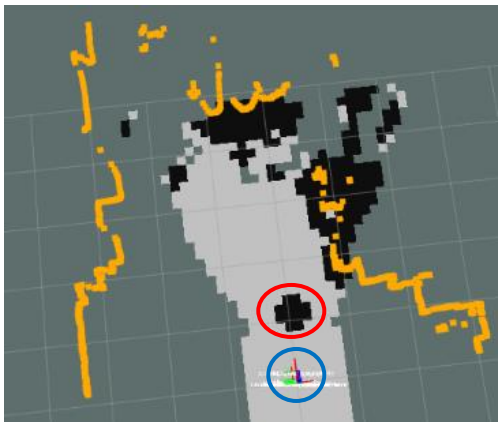


Figura 59. Mapa de obstáculos prueba de detección obstáculo bajo

Continuando con esta prueba se va a comprobar cómo detecta y etiqueta las rampas. Para esta prueba se coloca 2 tipos de pendientes delante del robot, una más fuerte y otra más suave. Este robot, por sus características, sólo puede subir las pendientes suaves, así que en un principio debería detectar la fuerte como obstáculo y la suave como transitable. Como se ve en la Figura 60, que muestra lo que está observando el robot por medio de la cámara RGB, el robot se encuentra frente a una pendiente. Después en el mapa de obstáculos de la Figura 61 se ve que ha etiquetado toda la pendiente como obstáculo porque es demasiado fuerte para este robot. Sin embargo en la Figura 62 se ve, por medio de la cámara RGB del robot, que éste se encuentra frente a otra pendiente, pero en este caso más suave. En la Figura 63 se ve el mapa de obstáculos, que ha creado a partir del mapa de elevación, y se puede comprobar que en este caso, etiqueta la pendiente como transitable y los bordes como obstáculo. Por tanto, la detección de obstáculos parece funcionar correctamente. Se ven las pendientes marcadas en el mapa con una elipse roja alrededor.

Figura 60.Cámara RGB del robot, rampa con pendiente fuerte

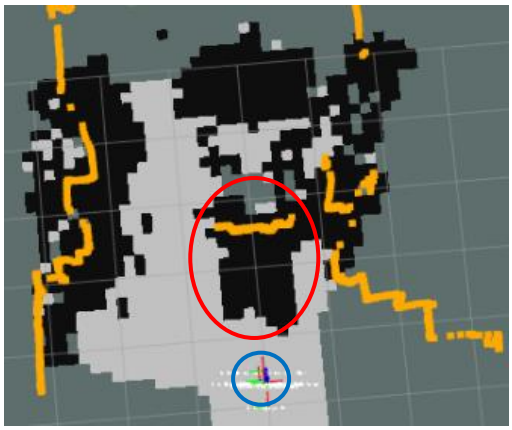


Figura 61. Mapa de obstáculos, rampa con pendiente fuerte

Figura 62. Cámara RGB del robot, rampa con pendiente suave

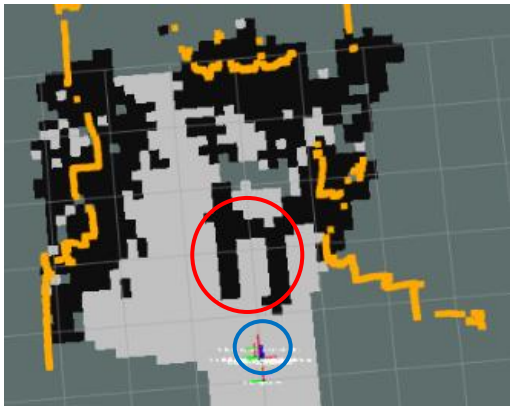


Figura 63. Mapa de obstáculos, rampa con pendiente suave

2. Segunda prueba con el robot: Navegación por el laboratorio. En esta prueba se le envía un objetivo al robot y se comprueba cómo navega por el laboratorio, esquivando los obstáculos que va detectando, hasta llegar al objetivo. Como se ve en la Figura 64 el robot comienza en este punto del laboratorio. En la Figura 65 se ve el mapa de obstáculos que ha creado, a partir del mapa de elevación, también el destino que se le ha proporcionado marcado, con una flecha roja, y la trayectoria que en principio ha planeado, con una línea verde. Después tras recorrer parte del camino, en la Figura 66 se ve el robot avanzando por el laboratorio y esquivando los diferentes obstáculos para tratar de llegar hasta el objetivo. En la Figura 67 se ve el mapa que ha ido creando en el visualizador y como ha replanteado su trayectoria inicial, para evitar obstáculos que ha ido detectando. En la Figura 68 se ve una imagen de lo que ve la cámara RGB del robot cuando se encuentra en el punto intermedio que se veía en la figura anterior. Después una vez el robot llega a su objetivo y queda con la orientación que se le había proporcionado, en la Figura 69 se puede ver lo que observa la cámara RGB del robot. En la Figura 70 se puede ver el mapa de obstáculos que ha creado el robot en su trayectoria hasta llegar al objetivo marcado. Por tanto se puede concluir que esta prueba ha funcionado de forma satisfactoria, porque aunque no es capaz de crear un mapa de obstáculos tan preciso como en simulación, es capaz de crear un mapa de obstáculos con el que navegar sin colisión de forma autónoma por el laboratorio hasta llegar a su destino.

Figura 64. Escenario inicial prueba navegación robot

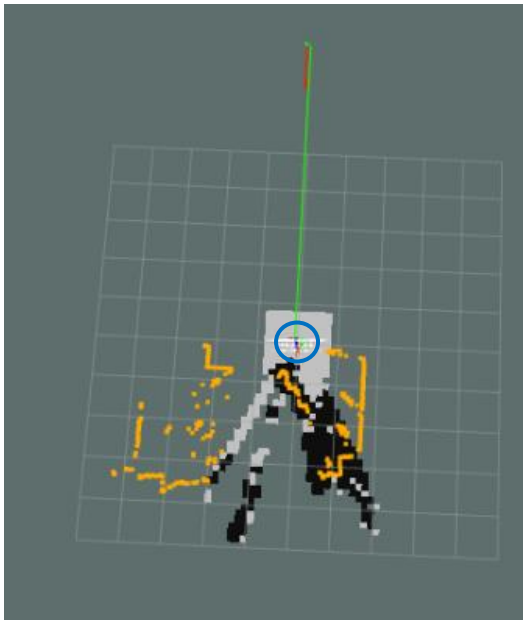


Figura 65. Mapa de obstáculos inicial prueba navegación robot

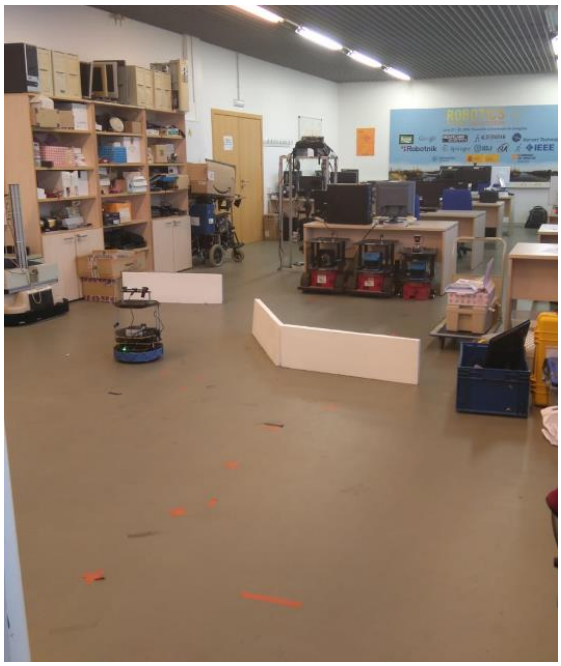


Figura 66. Escenario intermedio prueba navegación robot

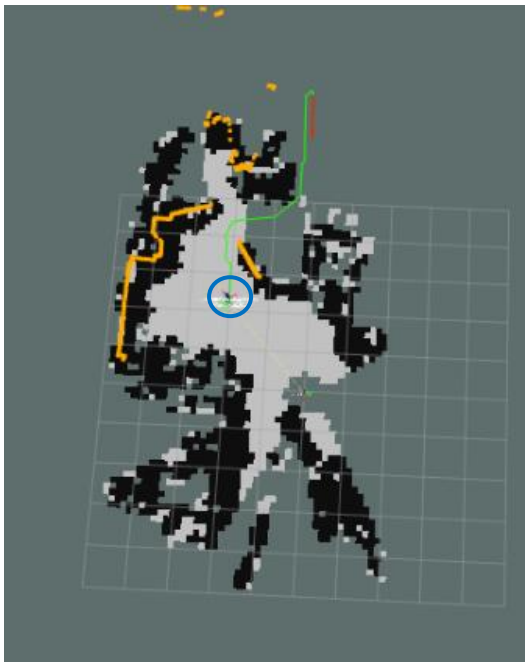


Figura 67. Mapa de obstáculos intermedio prueba navegación robot

Figura 68. Imagen de cámara RGB robot 1

Figura 69. Imagen de cámara RGB robot 2

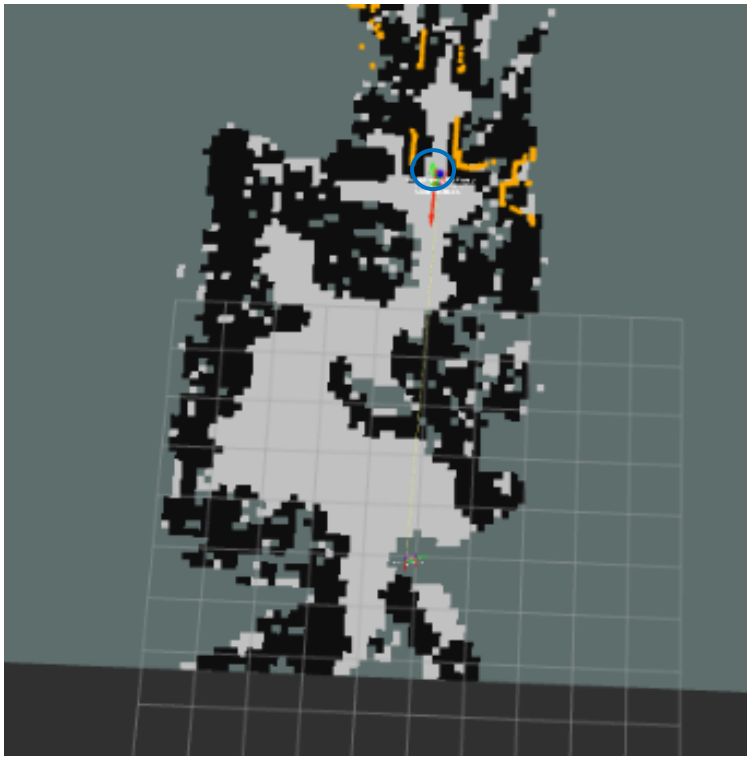


Figura 70. Mapa de obstáculos final en la prueba de navegación del robot

7. Conclusiones

7.1 Valoraciones del proyecto

El objetivo de este trabajo era hacer que el robot Turtlebot pudiera localizarse y navegar en un mapa desconocido, en donde haya zonas transitables y no transitables para las características de navegación de un robot. Como se ha visto en la memoria esto se ha logrado, y el robot es capaz de localizarse con el escáner láser Hokuyo dentro de un mapa, además es capaz de crear mapas de elevación mediante la cámara RGB-D Asus Xtion Pro Live. Con un procesamiento posterior de la información se convierten en mapa de obstáculos que diferencia los lugares transitables de los que no lo son (obstáculos y zonas de elevación no transitables). Además es capaz de navegar por estos mapas evitando estos obstáculos y recalculando la ruta hasta el destino marcado si alguno de estos obstáculos le impidiera pasar por el camino original. Es capaz de diferenciar las diferentes rampas para poder o no navegar por ellas, siempre teniendo en cuenta las características del robot.

En este proyecto se ha dado un paso más en el etiquetado de mapas de obstáculo y su conexión a través de *costmaps* con las técnicas de navegación implementadas en ROS, que no existían previamente en el software disponible, preparado fundamentalmente para navegación en terreno plano. Aún quedan varios problemas por solucionar, que se trataran en el próximo capítulo

Por tanto, a pesar de algunos inconvenientes encontrados en el desarrollo y que se han ido solucionando, el resultado final del trabajo es muy satisfactorio, ya que se ha conseguido lograr los objetivos marcados inicialmente.

7.2 Problemas encontrados

El problema más importante que ha ido apareciendo durante el trabajo, es la cantidad de recursos computacionales que consume, ya que el "elevation_mapping" para crear el mapa de alturas es muy costoso, necesita procesar todas las imágenes que va recibiendo de la cámara Asus Xtion Pro Live. Además también el procesar este mapa de elevación para convertirlo en mapa de obstáculos es costoso. En mi caso tuve que cambiar de ordenador porque el mío no era suficiente para procesar las imágenes de la cámara Asus Xtion Pro Live y crear el mapa de elevación. Con este ordenador siendo ya más potente se crea el mapa de elevación con mayor rapidez. Aunque se han tenido que reducir las velocidades iniciales del robot y aumentar la distancia entre vecinos en el mapa de elevación para hacer un procesamiento y generación de mapas correctos e tiempo real, hay veces que el sistema se queda esperando información nueva del mapa porque ha avanzado hasta el límite del espacio detectado y etiquetado. Una vez que se actualiza el mapa ya puede continuar con su trayectoria. Un problema que puede conllevar estos cambios para reducir el coste de procesamiento del "elevation_mapping" es que no se detecten bien los escalones pequeños, debido a que al aumentar, la distancia entre vecinos, se tiene que aumentar también la máxima diferencia de alturas entre ellos para etiquetar como transitable una misma pendiente. Por tanto, los escalones que sean más pequeños que ese umbral de máxima diferencia de altura entre vecinos se considerarán como transitables. La solución correcta para este caso

sería contar con un ordenador que tenga más potencia de procesamiento y así poder poner los parámetros de velocidad y de resolución adecuados.

Otro de los problemas está relacionado con, las restricciones de la cámara Asus Xtion Pro Live, que no permite la visualización alrededor del robot a menos de medio metro. La solución que se ha tomado en este proyecto es etiquetar un cuadrado de medio metro alrededor del robot como espacio (debido a que no puede navegar por zonas desconocidas y se quedaría detenido antes de empezar el movimiento). Una precaución obvia es colocar inicialmente siempre el robot a una distancia de más de medio metro de un obstáculo. En la Figura 71 se ve como queda ese cuadrado blanco alrededor del robot como zona transitable y en rosa lo que el robot sí que es capaz de detectar. Lo ideal para solucionar este problema sería utilizar otra cámara o sensor al principio que si pudiera detectar obstáculos en esa zona. Además la cámara, incluso con la inclinación que se le ha puesto para detectar mejor las caídas libres, no es capaz de detectar éstas si son con mucha profundidad. La solución para estas caídas seria añadirle a la cámara una plataforma móvil que permita controlar la inclinación (Pitch) que tiene en cada instante la cámara.

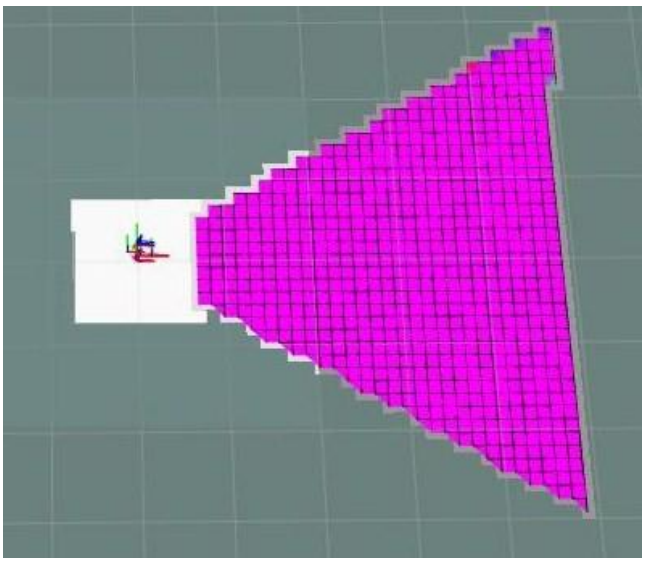


Figura 71. Vista del robot con cuadrado transitable

7.3 Líneas futuras de investigación

Existen varios caminos de investigación futura que podrían aportar mucho al proyecto. Por ejemplo en cuanto a la técnica de SLAM, la adición de alguna otra cámara que permitiera detectar más rango de obstáculos o una base móvil en la cámara para permitirle una rotación pan tilt(podría ser respecto al eje Z, para detectar más campo de visión lateral o incluso respecto al eje Y, para poder detectar mejor las caídas más prolongadas). También se podría empezar a navegar en un principio con el escáner láser o añadir una cámara que pueda ver en el rango cercano al robot, para asegurarse en esa zona inicial que no hay obstáculos a su alrededor.

Respecto a la detección de obstáculos, una línea futura de investigación, podría ser la detección de personas y de otros obstáculos en movimiento. Para ello habría que añadir alguna de las técnicas de reconocimiento de personas,

para diferenciarlas del resto de los obstáculos de la escena y etiquetarlas como obstáculos de diferente tipo de los considerados en este trabajo.

7.4 Conclusiones personales

Este trabajo me ha servido para darme cuenta de que en estos años de carrera, me he preparado para aprender cualquier cosa que me proponga. A ser capaz de realizar un proyecto completo. Documentándolo, de principio a fin, y tener claro los pasos a seguir. Además, nunca había utilizado la mayoría de las herramientas necesarias para realizarlo. Así, he sido capaz de aprender a utilizar el "framework" para el desarrollo de software para robots ROS, el sistema operativo Linux, los lenguajes de programación C++ y Phyton, la herramienta de visualización Rviz, el simulador Gazebo, el robot Turtlebot, el sensor láser y la cámara RGB-D. También he adquirido experiencia en el ámbito de la robótica y de la programación. Y, sobre todo, he aprendido a afrontar todos los problemas que van surgiendo en un proyecto, con metodología, y a no rendirme nunca, por muy complicado que parezca todo en algún momento.

Anexo A (ROS)

ROS (Robot Operating System) es un Framework para el desarrollo de sistemas robóticos. A pesar de que no es un sistema operativo, proporciona los servicios que cabría esperar de uno, como la abstracción de hardware, control de dispositivos a bajo nivel, comunicación entre procesos y el robot... Además, también proporciona librerías y herramientas para obtener, escribir, compilar y ejecutar código. Es un software libre, lo que da libertad para uso de investigación o uso comercial. Funciona, como se ve en la Figura 72, por medio de un Master que se encarga de manejar las conexiones de comunicación, además de permitir a los nodos localizarse unos a otros. También registra todos los nodos, servicios y *topics* que están en ejecución. Después están los nodos, que son los procesos que realizan los cálculos y se pueden combinar entre ellos para crear funciones más complejas. Se comunican entre ellos mediante mensajes y *topics* a los que pueden suscribirse o publicar. Los *topics* son entonces los canales de comunicación entre nodos, y cada uno de ellos envía un tipo de mensaje definido por ROS. Un mensaje no es más que una estructura de datos. Además de los ya disponibles, el usuario puede definir nuevos tipos combinando los existentes. En la Figura 73 se ve un esquema de como intercambian información los nodos, en el que varios pueden publicar un *topic* y también pueden estar suscritos a éste. Ésta es la forma de intercambiar información entre ellos. También se ve como un nodo manda un servicio que contesta otro nodo. Ésta es otra forma de intercambiar información entre nodos. Pero, en este caso, se requiere pedir una solicitud de información y esperar a la respuesta. Es una comunicación unidireccional.

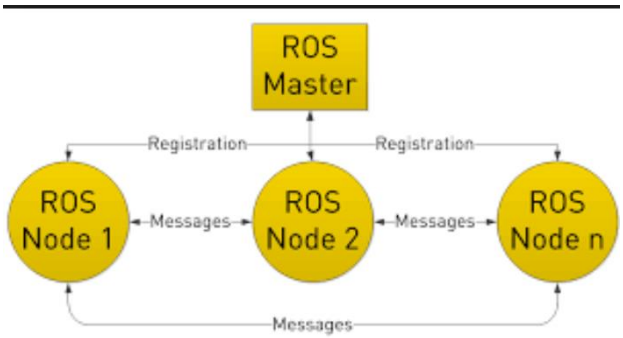


Figura 72. ROS master

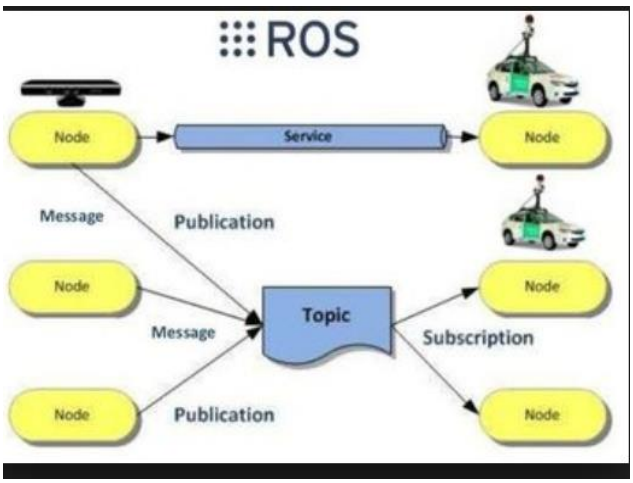


Figura 73. Nodos ROS

Anexo B (Turtlebot)

Es un robot de bajo coste con software libre que se utiliza para desarrollar aplicaciones robóticas. Está formado, como se ve en la Figura 74, por: (A) Una base móvil, en este caso la Kobuki. (B) Un sensor láser, que en este caso es el Hokuyo UST-20LX [ver Anexo C (Sensor Hokuyo UST-20LX)]. (C) Una cámara Asus Xtion Pro Live [ver Anexo D (Cámara Asus Xtion Pro Live)]. (D) Una batería externa. (E) El ordenador que lleva a bordo con el programa que se desea ejecutar.

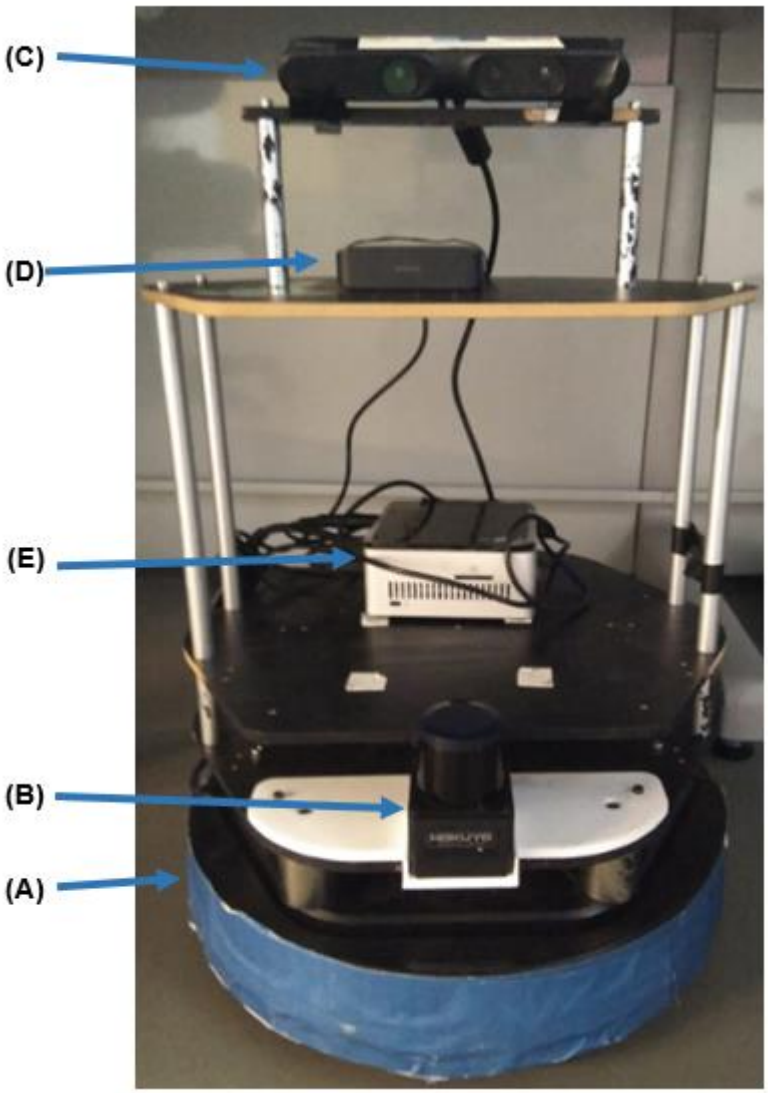


Figura 74. Robot Turtlebot y sensores embarcados

Anexo C (Sensor Hokuyo UST-20LX)

Es un sensor láser de 2 dimensiones para la medición de la distancia de objetos. Permite la detección de objetos, determinar su tamaño y la dirección de movimiento. Tiene un amplio rango, de unos 20 metros de distancia y 270º. Una alta precisión de ± 40mm de la distancia y una angular de 0.25º. En la Figura 75 se ve su gran rango de visión, para hacerse una idea gráficamente. Después, en la tabla de la Figura 76, se muestra las especificaciones de este sensor.

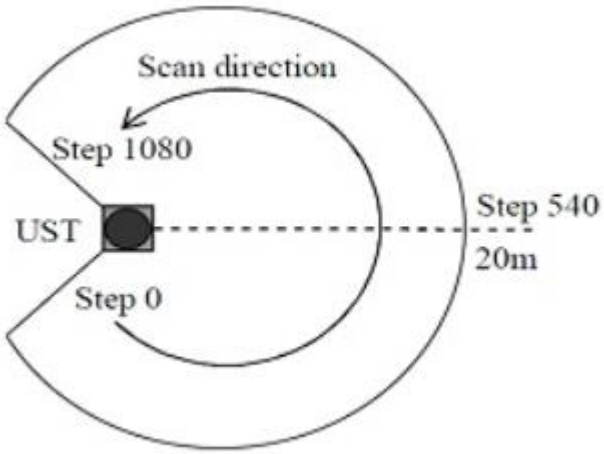


Figura 75. Rango de detección sensor Hokuyo UST-20LX

Supply voltaje	DC 12V/DC 24V (operation range10 to 30 V ripple within 10%
Supply current	150mA or less (during start up 450mA is necessary.)
Light source	Laser semiconductor (905nm), Laser class 1
Detection range	0.06m to 20m (white Kent sheet)
	0.06m to 8m (diffuse reflectance 10%)
	Max. detection distance : 60m
Accuracy	±40mm
Scan angle	270º
Scan speed	25ms (motor speed 2400rpm)
Angular resolution	0.25º
Dimensions(WxDxH)	50x50x70 (sensor only)

Figura 76. Características del sensor Hokuyo UST-20LX

Anexo D (Cámara Asus Xtion Pro Live)

Esta Cámara emplea un sensor de infrarrojos para la detección adaptativa de la profundidad. Además de utilizar el color y el sonido para captar movimientos en tiempo real. La cámara es como se ve en la Figura 77. Tiene además de los receptores de audio a los lados y la cámara RGB un emisor de infrarrojos y un receptor con los que se puede detectar la distancia a cada punto. En la Figura 78 se observa las especificaciones más importantes de esta cámara.

Figura 77. Cámara Asus Xtion Pro Live

Sensor	RGB y profundidad
Profundidad del tamaño de la imagen	VGA (640x480) :30 fps
	QVGA (320x240): 60 fps
Resolución	SXGA (1280*1024)
Campo de visión	58° H, 45° V, 70° D (Horizontal, Vertical, Diagonal)
Distancia de uso	Entre 0.8 m y 3.5 m
Consumo de energía	Inferior a 2.5 W
Dimensiones	18 x 3.5 x 5 cm

Figura 78. Características cámara Asus Xtion Pro Live

Anexo E (Gmapping y AMCL)

Tanto “gmapping” como “AMCL” son 2 paquetes de ROS [http://wiki.ros.org/gmapping, http://wiki.ros.org/amcl] que por medio de barridos de un sensor láser, permiten la localización en un mapa. La gran diferencia entre ambos es que el “AMCL” requiere un mapa como base para localizarse, y en cambio, el “gmapping” permite la localización sin necesidad de darle ningún mapa previo, de hecho, es el propio “gmapping” el que va creando a la vez este mapa para luego localizarse en él. Lo que se conoce como SLAM, del inglés “Simultaneous Localization And Mapping” que significa localización y creación de mapa simultáneo. Pese a esta gran diferencia, ambos utilizan un filtro de partículas, teniendo en cuenta no solo el movimiento del robot que proporciona la odometría a partir de los *encoders*, sino también la última medida que le da el sensor láser para localizarse en el mapa. Así que, el funcionamiento de ambos es parecido. Ambos tienen diferentes parámetros que se pueden variar para calcular esta posición de una forma más precisa o más rápida, aunque lo mejor suele ser buscar un equilibrio entre ambos. En la Figura 79 se observa un ejemplo de un mapa creado por el “gmapping”. Conforme ha ido avanzando el robot ha ido creando las paredes que ha visto hasta conseguir llegar al punto final. En la Figura 80 se ve como se obtiene la estimación de la localización con el “AMCL” (Adaptive Monte Carlo Localization). Donde, en este caso, el robot al principio ya tenía el mapa y con el “AMCL” obtiene la posición estimada del robot. Ésta se ve con la concentración de flechas rojas, cuanto más juntas estén las flechas, mayor es la certeza de una correcta localización del robot. Se ve en verde el camino que éste ha ido siguiendo.

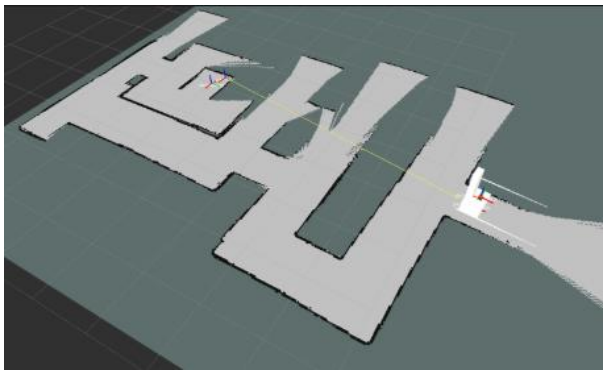


Figura 79. Ejemplo de gmapping

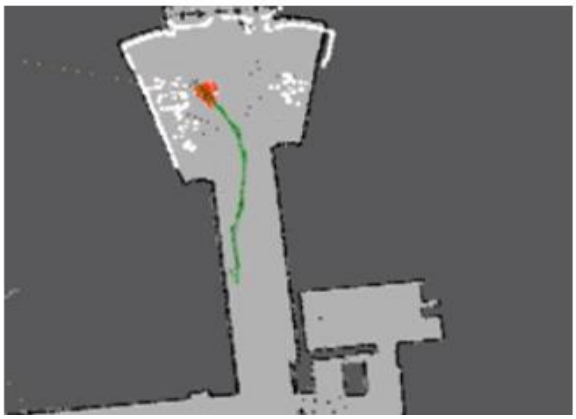


Figura 80. Ejemplo AMCL

Anexo F (Hector-SLAM)

Es un paquete de ROS creado por la universidad de Darmstadt que está compuesto a su vez por otros paquetes, el más importante es el "hector_mapping" y luego el "hector-geotiff", además de otros paquetes que sobre todo aportan más opciones o son necesarios para uno de estos dos paquetes .[http://wiki.ros.org/hector_slam]

El "hector-geotiff" es principalmente para guardar diferentes tipos de mapas como imágenes geotiff, pero en este proyecto no se va a utilizar, así que se desarrollara más en el siguiente. El "hector_mapping" es un paquete como el "gmapping" que permite la localización y creación de mapas simultáneos por medio de barridos de un sensor láser. En este caso no requiere la información de odometría, pero como nuestro robot si la tiene, se le puede dar también que siempre ayuda más para una mejor localización. Además, este paquete permite la localización en mapas en 3 dimensiones, con altura en el terreno como el que se va a utilizar. Esto se puede hacer con un paquete secundario que tiene, se llama Hector_imu_attitude_to_tf. Éste permite por medio de la información del IMU obtener datos de inclinación de Roll y Pitch, que en un mapa de 2 dimensiones no se tendrían, pero que se necesita para averiguar la altura posteriormente. A este paquete le he añadido un poco de código para compensar el offset que posee el IMU al principio.

Aunque para hacer uso de éste en 3D hace falta crear unas transformadas diferentes a las que tenía el robot, como se explica en el capítulo 2.2 Sistemas de referencia y transformaciones. En la Figura 81 se ve un ejemplo de como funciona el "hector_mapping". Se ve que ya ha creado el mapa con todos los obstáculos que ha ido viendo el robot con el láser y se marca en naranja lo que está viendo ahora el láser y lo que le está dando la posición de este instante.

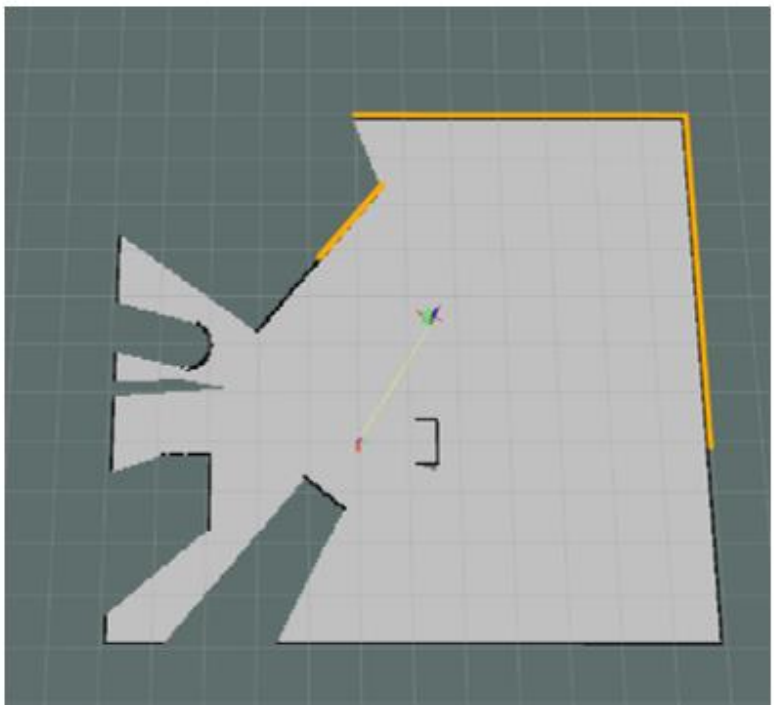


Figura 81. Ejemplo de hector_mapping

A continuación se muestran los parámetros utilizados en el "hector_mapping":

```
<param name="map_frame" value="map" />
<param name="base_frame" value="$(arg base_frame)" />
<param name="odom_frame" value="$(arg odom_frame)" />
<param name="use_tf_scan_transformation" value="true"/>
<param name="use_tf_pose_start_estimate" value="false"/>
<param name="pub_map_odom_transform" value="$(arg
pub_map_odom_transform)"/>
<param name="map_resolution" value="0.050"/>
<param name="map_size" value="$(arg map_size)"/>
<param name="map_start_x" value="0.5"/>
<param name="map_start_y" value="0.5" />
<param name="map_multi_res_levels" value="2" />
<param name="update_factor_free" value="0.4"/>
<param name="update_factor_occupied" value="0.9" />
<param name="map_update_distance_thresh" value="0.4"/>
<param name="map_update_angle_thresh" value="0.06" />
<param name="laser_z_min_value" value = "-1.0" />
<param name="laser_z_max_value" value = "1.0" />
<param name="advertise_map_service" value="true"/>
<param name="scan_subscriber_queue_size" value="$(arg
scan_subscriber_queue_size)"/>
<param name="scan_topic" value="$(arg scan_topic)"/>
<remap from="map" to="/map"/>
<param name="tf_map_scanmatch_transform_frame_name" value="$(arg
tf_map_scanmatch_transform_frame_name)" />
```

Anexo G (Elevation_mapping)

"Elevation_mapping" es un paquete de ROS [https://github.com/ethz-asl/elevation_mapping] diseñado para navegar con un robot y crear un mapa en 3 dimensiones, a partir de la pose del robot y de la información que recibe a de una cámara o sensor. Utiliza varios parámetros que se pueden ajustar por medio de 3 ficheros:

-Uno con la configuración del robot, en la que se ajustan diversos parámetros que tienen que ver con el robot y los topics donde se le proporciona la pose y la información del sensor. A continuación están los parámetros de este fichero que se han utilizado en el proyecto:

```
point_cloud_topic: "/turtlebot_1/camera/depth/points"
map_frame_id: "/map"
robot_base_frame_id: "/turtlebot_1/base_link"
robot_pose_with_covariance_topic: "/turtlebot_1/pose"
robot_pose_cache_size: 200
track_point_frame_id: "/turtlebot_1/base_link"
track_point_x: 0.0
track_point_y: 0.0
track_point_z: 0.0
fused_map_publishing_rate: 2
```

-Otro con la configuración del mapa, es decir, el tamaño de mapa que se crear, cada cuanto tiempo publicarlo y otros parámetros relacionados con el mapa que quieres crear. A continuación están los parámetros de este fichero que se han utilizado en el proyecto:

```
length_in_x: 15.0
length_in_y: 15.0
position_x: 0.0
position_y: 0.0
resolution: 0.15
min_variance: 0.000009
max_variance: 0.01
mahalanobis_distance_threshold: 2.5
multi_height_noise: 0.0000009
surface_normal_estimation_radius: 0.015
surface_normal_positive_axis: z
```

-Un tercero con el sensor que se va a utilizar y los parámetros que tiene esa cámara. A continuación están los parámetros de este fichero que se han utilizado en el proyecto:

```
sensor_processor/type: Kinect
sensor_processor/cutoff_min_depth: 0.35
sensor_processor/cutoff_max_depth: 6.0
sensor_processor/normal_factor_a: 0.0012
sensor_processor/normal_factor_b: 0.0019
sensor_processor/normal_factor_c: 0.4
sensor_processor/lateral_factor: 0.001376915
```

Con este paquete se obtiene un mapa que viene dado como un *gridMap,* el cual es una matriz. Esta matriz representa un mapa donde cada celda es un punto del mapa. La separación entre cada punto del mapa corresponde a la resolución que se le haya dado. Estas celdas tienen la información de la altura, de cada punto, en el mapa.

En la Figura 82 se ve una vista superior de un ejemplo de mapa de elevación que crea el robot. La zona gris es zona que aún no ha descubierto (desconocida). La zona rosa es el suelo que ve liso, y después, los diferentes colores corresponden con las alturas que va detectando en el mapa siguiendo la escala adjunta. En la Figura 83 se ve una vista lateral del mismo mapa de elevación, donde se puede ver mejor la altura de cada punto y como crearía ese mapa en 3D.

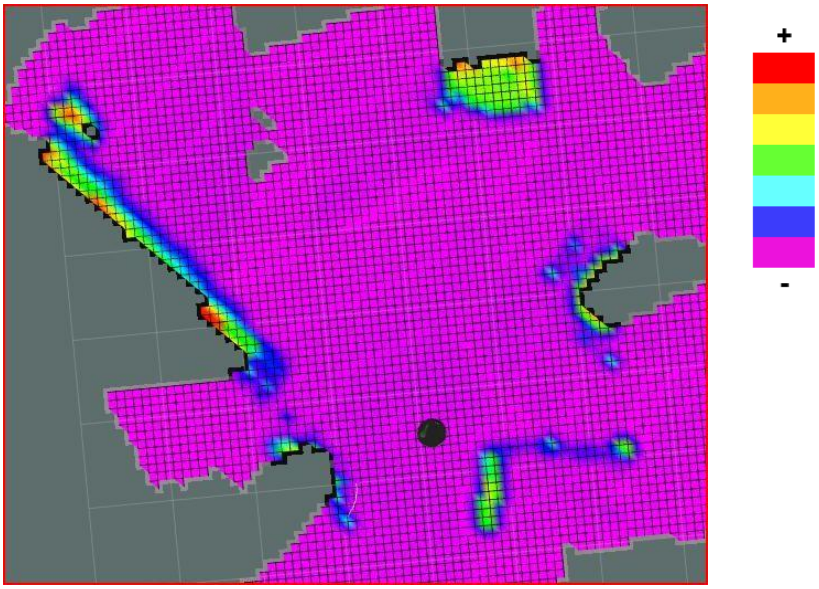


Figura 82. Vista superior del mapa de elevación

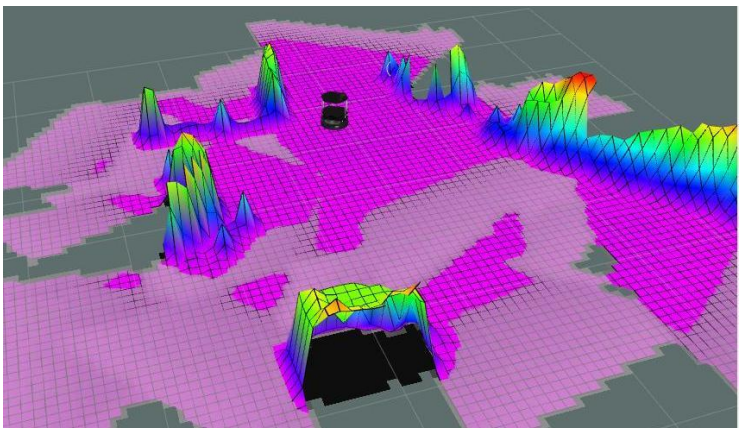


Figura 83. Vista lateral del mapa de elevación

Anexo H (Topics ROS)

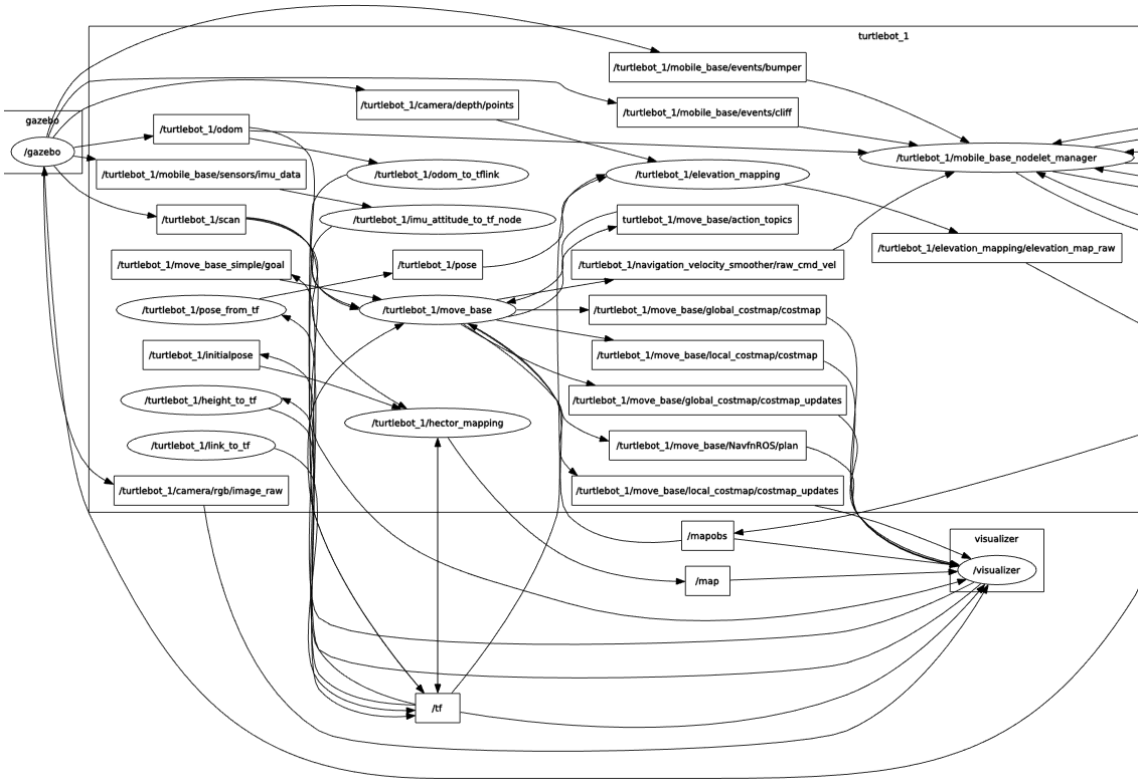


Figura 84. Topics ROS izquierda

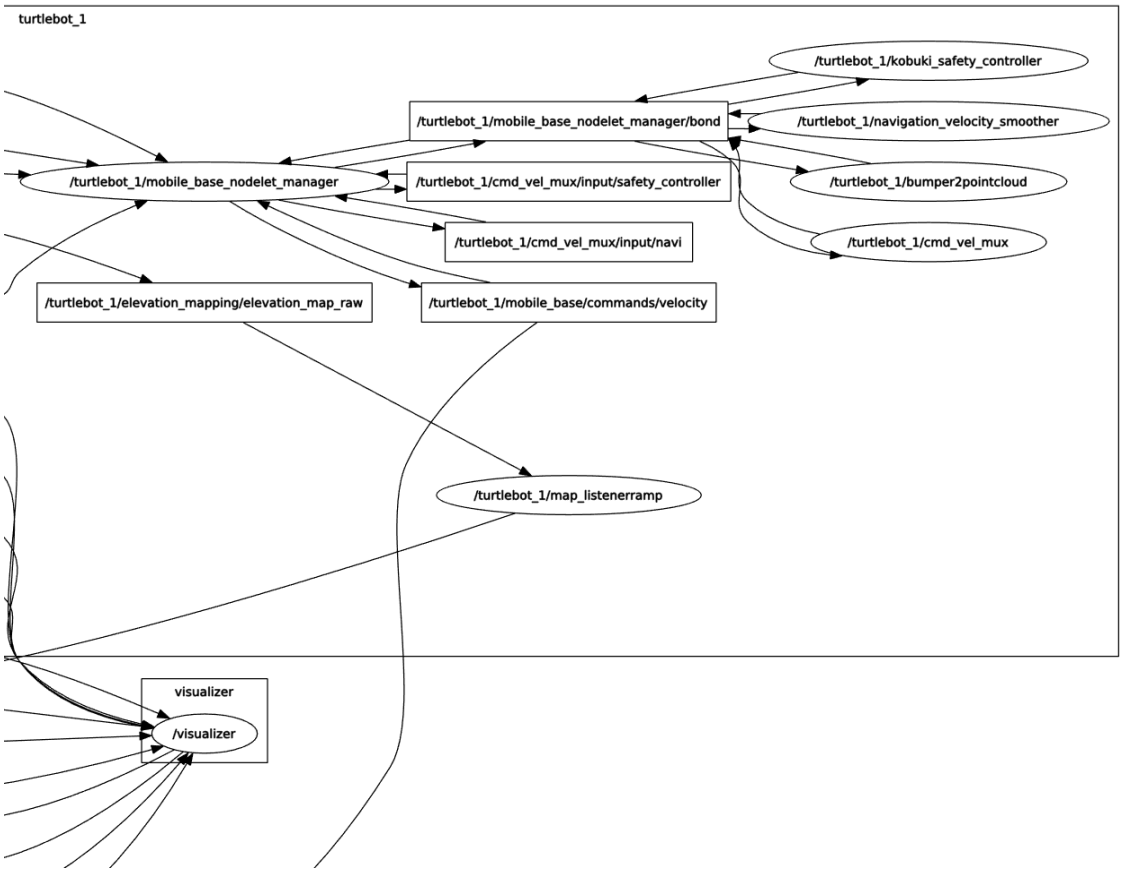


Figura 85. Topics ROS derecha

En la Figura 84 y Figura 85 se ven los nodos que están en ejecución, representados dentro de una elipse. Y los topics, representados dentro de un rectángulo. En el capítulo 5. Nodos ROS se ha hablado de los nodos y ahora se va a hablar de los topics que son los siguientes:

Odom: este topic es publicado por el simulador y representa la odometría del robot.

Scan: este topic es publicado por el simulador y es la información que detecta el sensor láser.

Camera/depth/points: este topic es publicado por el simulador y es la información que da la cámara Kinect.

Camera/rgb/image_raw: este topic es publicado por el simulador y es la información que toma la cámara Kinect pasado a una imagen.

move_base/simple_goal: este topic lo publica el visualizador y da la informacion del destino que se le ha mandado al robot.

Pose: este topic es publicado por el nodo pose_from_tf y es la posición 3D del robot.

mobile_base/events/bumper: este topic es publicado por el simulador y representa si el bumper del robot está pulsado o no.

mobile_base/sensors/Cliff: este topic lo publica el simulador y representa si el robot se encuentra en un acantilado o no.

navigation_velocity_smoother/raw_cmd_vel: este topic es publicado por el move_base y es la consigna de velocidad que ha calculado como óptima.

move_base/global_costmap/costmap: este topic es publicado por el move_base y el costmap completo que ha ido creando a partir del mapa de obstáculos.

move_base/local_costmap/costmap: este topic es publicado por el move_base y el costmap más próximo al robot que ha ido creando a partir del mapa de obstáculos.

move_base/global_costmap/costmap_updates: este topic es publicado por move_base y representa el área de datos que se ha actualizado del costmap global

move_base/local_costmap/costmap_updates: este topic es publicado por move_base y representa el área de datos que se ha actualizado del costmap local.

Map: este topic es publicado por el hector-mapping y representa el mapa que ha creado con los barridos del sensor láser.

Elevation_mapping/elevation_map_raw: este topic es publicado por el elevation_mapping y representa el mapa de elevación que ha creado a partir de la cámara Kinect.

Mapobs: este topic es publicado por el map_listenerramp y representa el mapa de obstáculos que se ha creado a partir del mapa de elevación.

mobile_base/command/velocity: este topic lo publica el mobile_base y es la consigna de velocidad que se pasa a las ruedas para que el robot se mueva a esa velocidad.

Cmd_vel_mux/input/navi: este topic lo publica el mobile_base y representa, la velocidad que tiene que tener el robot y que nodo maneja el robot en cada instante. Es principalmente cuando varios nodos quieren manejar el robot con diferentes velocidades.

Cmd_vel_mux/input/safety_controller: este topic lo publica el mobile_base y representa la velocidad que la llega pero ya procesada para que cumpla los distintos requisitos de seguridad que tiene.

mobile_base/nodelet_manager/bond: este topic está publicado por el mobile_base y los nodos de seguridad para que si cualquiera de ellos falla que no se mande velocidad y el robot se quede parado.

Anexo I (Código utilizado en ROS)

-Código odom_to_tf:

```
#include <ros/ros.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/Pose.h>
#include <tf/transform_broadcaster.h>

  void odomCallback(const nav_msgs::Odometry& input)
  {
    static tf::TransformBroadcaster output;
    tf::TransformBroadcaster odom_broadcaster;
    tf::Transform transform;
    tf::Vector3 translacion;
    translacion.setX(input.pose.pose.position.x);
    translacion.setY(input.pose.pose.position.y);
    translacion.setZ(input.pose.pose.position.z);
    transform.setOrigin(translacion);
    tf::Quaternion q;
    q.setX(input.pose.pose.orientation.x);
    q.setY(input.pose.pose.orientation.y);
    q.setZ(input.pose.pose.orientation.z);
    q.setW(input.pose.pose.orientation.w);
    transform.setRotation(q);

odom_broadcaster.sendTransform(tf::StampedTransform(transform,ros::Time::now(),"t
urtlebot_1/odom","turtlebot_1/base_footprint"));
  }

int main(int argc, char **argv)
{
  ros::init(argc, argv, "odomtf");
  ros::NodeHandle n_;
  ros::Subscriber sub_= n_.subscribe("/turtlebot_1/odom", 200, &odomCallback);
  ros::spin();
  return 0;
}
```

Este código se encarga de transformar la odometría del robot en la transformación entre “odom” y “base_footprint”.

-Código height_to_tf:

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <tf/transform_broadcaster.h>
#include <angles/angles.h>
#include <math.h>
int main(int argc, char** argv){
  ros::init(argc, argv, "pose_tf_listener");
  ros::NodeHandle node;
  tf::StampedTransform input;
  tf::StampedTransform input2;
  tf::TransformListener listener;
```

```
  tf::TransformListener listener2;
  tf::TransformBroadcaster height_broadcaster;
  tf::Transform output;
  tf::Vector3 translacionout;
  translacionout.setX(0);
  translacionout.setY(0);
  tf::Quaternion qout;
  qout.setX(0);
  qout.setY(0);
  qout.setZ(0);
  qout.setW(1);
  output.setRotation(qout);
  float
antigua_px=0.0,antigua_py=0.0,antigua_pz=0.0,dpx=0.0,dpy=0.0,dpz=0.0,dpa=0.0;
  double
roll=0.0,pitch=0.0,yaw=0.0,antigua_yaw=0.0,antigua_pitch=0.0,myaw=0.0,mpitch=0.0;
  ros::Rate rate(10);
  while (node.ok()){
    bool funciona=true;
    try{
      listener.lookupTransform("/map", "/turtlebot_1/base_footprint", ros::Time(0), input);
    }
    catch (tf::TransformException &ex) {
      funciona=false;
    }
    try{
      listener2.lookupTransform("/turtlebot_1/base_stabilizer", "/turtlebot_1/base_link",
ros::Time(0), input2);
    }
    catch (tf::TransformException &ex) {
      funciona=false;
    }
    if (funciona==true) {
        dpx=input.getOrigin()[0]-antigua_px;
        dpy=input.getOrigin()[1]-antigua_py;
        antigua_px=input.getOrigin()[0];
        antigua_py=input.getOrigin()[1];
        tf::Quaternion qin1;
           qin1=input.getRotation();
           tf::Matrix3x3(qin1).getRPY(roll,pitch,yaw);
           yaw=angles::normalize_angle(yaw);
           if(yaw+antigua_yaw==0.0){
                    myaw=0.0;
           }else {
                    myaw=(antigua_yaw+yaw)/2;
           }
           antigua_yaw=yaw;
           tf::Quaternion qin2;
           qin2=input2.getRotation();
           tf::Matrix3x3(qin2).getRPY(roll,pitch,yaw);
           pitch=angles::normalize_angle(pitch);
           if(antigua_pitch+pitch==0.0){
                    mpitch=0.0;
                    antigua_pitch=pitch;
                    dpz=0.0;
```

```
        }else{
                mpitch=(-(antigua_pitch+pitch)/2);
                antigua_pitch=pitch;
                if(cos(myaw)==0.0 ){
                        dpa=dpy;
                }else if (dpx==0.0){
                        dpa=0.0;
                }else{
                        dpa=dpx/cos(myaw);
                }
                if(sin(mpitch)==0.0 || dpa==0.0 || cos(mpitch)==0.0){
                        dpz=0.0;
                }else{
                        dpz=fabs(dpa)*sin(mpitch)/cos(mpitch)       ;
                }
        }
      antigua_pz=antigua_pz+dpz;
        translacionout.setZ(antigua_pz+0.0102);
      output.setOrigin(translacionout);

height_broadcaster.sendTransform(tf::StampedTransform(output,ros::Time::now(),"turtl
ebot_1/base_footprint","turtlebot_1/base_stabilizer"));
  }else if(funciona==false){
        translacionout.setZ(0+0.0102);
      output.setOrigin(translacionout);

height_broadcaster.sendTransform(tf::StampedTransform(output,ros::Time::now(),"turtl
ebot_1/base_footprint","turtlebot_1/base_stabilizer"));
  }
   rate.sleep();
 }
 return 0;
};
```

Este código se encarga de crear un nodo que estima la altura y da esa infromacion como la transformada entre “base_footprint” y “base_stabilizer”.

-Código tf_links:

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
 int main(int argc, char **argv)
{
  ros::init(argc, argv, "heighttf");
  ros::NodeHandle n_;
  tf::TransformBroadcaster br;
  tf::Transform transform1;
  tf::Transform transform2;
  tf::Transform transform3;
  tf::Transform transform4;
  tf::Transform transform5;
  tf::Transform transform6;
  tf::Transform transform7;
  tf::Transform transform8;
  tf::Transform transform9;
  tf::Transform transform10;
```

```
  ros::Rate rate(10);
  while (n_.ok()){
   transform1.setOrigin( tf::Vector3(0.0, 0.021, 0.5) );
   transform1.setRotation( tf::Quaternion(0, 0.1, 0, 0.995) );
   transform2.setOrigin( tf::Vector3(0.024, 0.0, 0.131) );
   transform2.setRotation( tf::Quaternion(0, 0, 0, 1.0) );
   transform3.setOrigin( tf::Vector3(-0.102, 0.0, 0.272) );
   transform3.setRotation( tf::Quaternion(0, 0.0, 0, 1.0) );
   transform4.setOrigin( tf::Vector3(0.0, -0.022, 0.0) );
   transform4.setRotation( tf::Quaternion(0, 0.0, 0, 1.0) );
   transform5.setOrigin( tf::Vector3(0.0, 0.0, 0.0) );
   transform5.setRotation( tf::Quaternion(-0.5, 0.5, -0.5, 0.5) );
   transform6.setOrigin( tf::Vector3(-0.0, 0.027, 0.0) );
   transform6.setRotation( tf::Quaternion(0, 0.0, 0, 1.0) );
   transform7.setOrigin( tf::Vector3(0.0, 0.0, 0.0) );
   transform7.setRotation( tf::Quaternion(-0.5, 0.5, -0.5, 0.5) );
   transform8.setOrigin( tf::Vector3(0.152, 0.0, 0.005) );
   transform8.setRotation( tf::Quaternion(0, 0.0, 0, 1.0) );
   transform9.setOrigin( tf::Vector3(0.0, 0.0, 0.037) );
   transform9.setRotation( tf::Quaternion(0, 0.0, 0, 1.0) );
   transform10.setOrigin( tf::Vector3(0.056, 0.062, 0.020) );
   transform10.setRotation( tf::Quaternion(0, 0, 0, 1.0) );
   br.sendTransform(tf::StampedTransform(transform1, ros::Time::now(),
"turtlebot_1/base_link", "turtlebot_1/camera_rgb_frame"));
   br.sendTransform(tf::StampedTransform(transform2, ros::Time::now(),
"turtlebot_1/base_link", "turtlebot_1/plate_bottom_link"));
   br.sendTransform(tf::StampedTransform(transform3, ros::Time::now(),
"turtlebot_1/base_link", "turtlebot_1/mount_asus_xtion_pro_link"));
   br.sendTransform(tf::StampedTransform(transform4, ros::Time::now(),
"turtlebot_1/camera_rgb_frame", "turtlebot_1/camera_link"));
   br.sendTransform(tf::StampedTransform(transform5, ros::Time::now(),
"turtlebot_1/camera_rgb_frame", "turtlebot_1/camera_rgb_optical_frame"));
   br.sendTransform(tf::StampedTransform(transform6, ros::Time::now(),
"turtlebot_1/camera_rgb_frame", "turtlebot_1/camera_depth_frame"));
   br.sendTransform(tf::StampedTransform(transform7, ros::Time::now(),
"turtlebot_1/camera_depth_frame", "turtlebot_1/camera_depth_optical_frame"));
   br.sendTransform(tf::StampedTransform(transform8, ros::Time::now(),
"turtlebot_1/plate_bottom_link", "turtlebot_1/base_laser_plate_link"));
   br.sendTransform(tf::StampedTransform(transform9, ros::Time::now(),
"turtlebot_1/base_laser_plate_link", "turtlebot_1/base_laser_link"));
   br.sendTransform(tf::StampedTransform(transform10, ros::Time::now(),
"turtlebot_1/base_link", "turtlebot_1/gyro_link"));
   rate.sleep();
  }
  return 0;
}
```

Este código se encarga de crear un nodo que cree las transformadas fijas que van desde “base_link” hasta los sensores que se necesitan.

-Código pose_from_tf:

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <tf/transform_broadcaster.h>
```

```
#include <angles/angles.h>
int main(int argc, char** argv){
  ros::init(argc, argv, "pose_tf_listener");
  ros::NodeHandle node;
  ros::Publisher pose_publisher =
node.advertise<geometry_msgs::PoseWithCovarianceStamped>("/turtlebot_1/pose",
200);
  tf::TransformListener listener;
 // float antigua_px=0,antigua_py=0,antigua_pz=0;
 // float dpx,dpy,dpz,dpa;
  //double roll,pitch,yaw,antigua_roll=0,antigua_pitch=0,antigua_yaw=0;
  ros::Rate rate(10);
  while (node.ok()){
    tf::StampedTransform transform;
    bool funciona=true;
    try{
      listener.lookupTransform("/map", "/turtlebot_1/base_link", ros::Time(0), transform);
    }
    catch (tf::TransformException &ex) {
      funciona=false;
    }
    if (funciona==true) {
          geometry_msgs::PoseWithCovarianceStamped pose_msg;
          pose_msg.header.stamp = ros::Time::now();
          pose_msg.header.frame_id = "odom";
        /*dpx=transform.getOrigin()[0]-antigua_px;
        dpy=transform.getOrigin()[1]-antigua_py;
        dpz=0;
        tf::Quaternion q;
          q=transform.getRotation();
          tf::Matrix3x3(q).getRPY(roll,pitch,yaw);
          roll=angles::normalize_angle_positive(roll);
          pitch=angles::normalize_angle_positive(pitch);
          yaw=angles::normalize_angle_positive(yaw);
        antigua_px=transform.getOrigin()[0];
        antigua_py=transform.getOrigin()[1];
          antigua_roll=roll;
          antigua_pitch=pitch;
          antigua_yaw=yaw;*/
        pose_msg.pose.pose.position.x = transform.getOrigin()[0];
          pose_msg.pose.pose.position.y = transform.getOrigin()[1];
          pose_msg.pose.pose.position.z = transform.getOrigin()[2];
          pose_msg.pose.pose.orientation.x = transform.getRotation()[0];
          pose_msg.pose.pose.orientation.y = transform.getRotation()[1];
          pose_msg.pose.pose.orientation.z = transform.getRotation()[2];
          pose_msg.pose.pose.orientation.w = transform.getRotation()[3];
          pose_publisher.publish(pose_msg);
   }
    rate.sleep();
  }
  return 0;
};
```

Este código crea un nodo que, a partir de la transformada entre “map” y “base_link”, extrae la posición del robot y la publica en un topic.

-Código map_listenerramp:

```
#include "grid_map_msgs/GridMap.h"
#include "nav_msgs/OccupancyGrid.h"
#include <ros/ros.h>
#include <grid_map_ros/grid_map_ros.hpp>
#include "grid_map_ros/GridMapMsgHelpers.hpp"
#include <grid_map_cv/grid_map_cv.hpp>
#include <vector>
#include <string>
#include <cmath>
#include <limits>
// ROS
#include <sensor_msgs/point_cloud2_iterator.h>
// STL
#include <algorithm>
using namespace Eigen;
using namespace grid_map;
using namespace std;
class SubscribeAndPublish
{
public:
  SubscribeAndPublish()
  {
    pub_ = n_.advertise<nav_msgs::OccupancyGrid>("/mapobs", 200);
    sub_ = n_.subscribe("/turtlebot_1/elevation_mapping/elevation_map_raw", 200,
&SubscribeAndPublish::callback, this);
  }
  void callback(const grid_map_msgs::GridMap& input)
  {
    ROS_INFO("entrando callback");
    grid_map::GridMap gridMap;
    GridMapRosConverter::fromMessage( input,gridMap);
    nav_msgs::OccupancyGrid output;
    std::string layer;
    layer=input.layers[0];
    output.header.frame_id = gridMap.getFrameId();
    output.header.stamp.fromNSec(gridMap.getTimestamp());
    output.info.map_load_time = output.header.stamp;
    output.info.resolution = gridMap.getResolution(); //resolución
    output.info.width = gridMap.getSize()(0);         //tamaño x
    output.info.height = gridMap.getSize()(1);        //tamaño y
    Position position = gridMap.getPosition() - 0.5 * gridMap.getLength().matrix();
    output.info.origin.position.x = position[0]; //posicion.x()
    output.info.origin.position.y = position[1];//posicion.y()
    output.info.origin.position.z = 0.0;
    output.info.origin.orientation.x = 0.0;
    output.info.origin.orientation.y = 0.0;
    output.info.origin.orientation.z = 0.0;
    output.info.origin.orientation.w = 1.0;
    size_t nCells = gridMap.getSize().prod();
    output.data.resize(nCells);
    const float resolution =gridMap.getResolution();
    const float cellMin = 0.0;
    const float cellMax = 100.0;// probabilidad de ocupación va de 0 a 100. Siendo -1
```

```
desconocida.
   const float cellRange = cellMax - cellMin;
   const float dataMin=-1.0;
   const float dataMax=1.0;
   //const float limite=3;
   const float maxangle=1.0;
   const int obs=100;//obstaculos
   const int desc=-1.0;//mapa desconocido
   const int blanco=0.0;//zonas libres
   Position
positionnow,positionup,positiondown,positionright,positionleft,positionblanco;
   //recorrido de toda la matriz para ir dando valores al mapa
   for (GridMapIterator iterator(gridMap); !iterator.isPastEnd(); ++iterator) {
    if(gridMap.getPosition(*iterator, positionnow)==true){
         float value=(gridMap.at(layer, *iterator) - dataMin) / (dataMax - dataMin);
      //bool desconocido=false;
         if (isnan(value) && (((positionnow[0]>position[0]+9.2) &&
(positionnow[0]<position[0]+10.8)) && ((positionnow[1]>position[1]+9.2) &&
(positionnow[1]<position[1]+10.8)))){    //da valor a las zonas de alrededor al principio
                  value=blanco;
         }else if(isnan(value)){
                  value=desc;
         }else if(obs==100){ //para que entre siempre
                  value = cellMin + min(max(0.0f, value), 1.0f) * cellRange; //valor en %
                  positionup[0]=positionnow[0];
                  positionup[1]=positionnow[1]+resolution;
                  if(gridMap.isInside(positionup)==true){       //comprobar si hay valor a la
arriba
                           float valueup=(gridMap.atPosition(layer, positionup) - dataMin) /
(dataMax - dataMin);
                           if (isnan(valueup)){
                           //desconocido=true;//nada
                           }else {
                                    valueup = cellMin + min(max(0.0f, valueup), 1.0f) *
cellRange;
                                    if(abs(value-valueup)>maxangle ){
                                             value=obs;
                                    }
                           }
                  }
                  positiondown[0]=positionnow[0];
                  positiondown[1]=positionnow[1]-resolution;
                  if(gridMap.isInside(positiondown)==true){//comprobar si hay valor a la
abajo
                           float valuedown=(gridMap.atPosition(layer, positiondown) -
dataMin) / (dataMax - dataMin);
                           if (isnan(valuedown)){
                           //desconocido=true;//nada
                           }else {
                                    valuedown = cellMin + min(max(0.0f, valuedown), 1.0f) *
cellRange;
                                    if(abs(value-valuedown)>maxangle ){
                                             value=obs;
                                    }
                           }
```

```
                }
                positionleft[0]=positionnow[0]-resolution;
                positionleft[1]=positionnow[1];
                if(gridMap.isInside(positionleft)==true){//comprobar si hay valor a la
izquierda
                        float valueleft=(gridMap.atPosition(layer, positionleft) - dataMin) /
(dataMax - dataMin);
                        if (isnan(valueleft)){
                        //desconocido=true;//nada
                        }else {
                                valueleft = cellMin + min(max(0.0f, valueleft), 1.0f) *
cellRange;
                                if(abs(value-valueleft)>maxangle ){
                                        value=obs;
                                }
                        }
                }
                positionright[0]=positionnow[0]+resolution;
                positionright[1]=positionnow[1];
                if(gridMap.isInside(positionright)==true){//comprobar si hay valor a la
derecha
                        float valueright=((gridMap.atPosition(layer, positionright)) -
dataMin) / (dataMax - dataMin);
                        if (isnan(valueright)){
                        //desconocido=true;//nada
                        }else {
                                valueright = cellMin + min(max(0.0f, valueright), 1.0f) *
cellRange;
                                if(abs(value-valueright)>maxangle ){
                                        value=obs;
                                }
                        }
                }
        }
    if (value!=obs && value!=desc)
         value=blanco;
    size_t index = getLinearIndexFromIndex(iterator.getUnwrappedIndex(),
gridMap.getSize(), false);
    //El orden entre los dos mapas de las celdas no es lo mismo por eso cambia el
index.
    output.data[nCells - index - 1] = value;
     }

    }
   pub_.publish(output);
ROS_INFO("saliendo callback");
  }
private:
  ros::NodeHandle n_;
  ros::Publisher pub_;
  ros::Subscriber sub_;
};
int main(int argc, char **argv)
{
  ros::init(argc, argv, "subscribe_and_publish");
```

```
  SubscribeAndPublish Leermaparamp;
  ros::spin();
  return 0;
}
```

Este código se encarga de crear un nodo que se suscribe al mapa de elevación que publica el “elevation_mapping”, después lo procesa y crea un mapa de obstáculos, que publica en un topic, para tener un mapa con el que navegar y planificar.

Bibliografía

[1] wiki.ros.org. ROS-Documentation [online] Avaible from: http://wiki.ros.org.

[2] turtlebot.com. Turtlebot [online] Avaible from: http://www.turtlebot.com.

[3] gazebosim.org. Gazebo-Documentation [online] Avaible from: http://gazebosim.org.

[4] github.com. ROS-Packages [online] Availbe from: http://github.com.

[5] D. Fox, W. Burgard, S. Thrun. The dynamic window approach to collision avoidance. IEEE Robot. Automat. Mag. year 1997 Volume 4 pages 23-33.

[6] P. Fankhauser, M. Bloesch, C. Gehring, M. Hutter. Robot-centric Elevation Mapping with Uncertainty Estimates. International Conference on Climbing and Walking Robots (CLAWAR), 2014.

[7] Q. Lin, Z. Ke, S. Bi, and S. Xu, Y. Liang, F. Hong, L. Feng. Indoor mapping using gmapping on embedded system.2017 IEEE International Conference on Robotics and Biomimetics (ROBIO) pages 2444-2449.

[8] T. Braun, H. Bitsch, K. Berns. Visual Terrain Traversability Estimation Using a Combined Slope/Elevation Model. Year 2008.

[9] S. Kohlbrecher. Hector SLAM for robust mapping in USAR environments. ROS Workshop Graz 22/08/2012.

[10] X. Zhu, C. Qiu and M. A. Minor. Terrain-inclination-based Three-dimensional Localization for Mobile Robots in Outdoor Environments. J. Field Robotics year 2014, volume 31. Pages 477-492.

[11] P. Fankhauser, M.Hutter. A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation. Robot Operating System (ROS). Year 2016. Volume 1, Chapter 5. ISBN 978-3-319-26052-5.

[12] Asus. Asus Xtion Pro Live Specifications [online] Avaible from: https://www.asus.com/es/3D-Sensor/Xtion_PRO_LIVE/ .

[13] Hokuyo. Hokuyo UST-20LX Specifications [online] Avaible from: https://www.hokuyo-aut.jp/search/single.php?serial=167.

[14] D.F. Wolf, G.S. Sukhatme, D. Fox. Autonomous Terrain Mapping and Classification Using Hidden Markov Models. Proceedings of the 2005 IEEE International Conference on Robotics and Automation Barcelona, Spain, April 2005.

[15] A. S. Souza, L. M. Gonçalves. 2.5-Dimensional Grid Mapping from Stereo Vision for Robotic Navigation. 2012 Brazilian Robotics Symposium and Latin American Robotics Symposium. Pages 39-44.

[16] H. Ahmed, M. Tahir. Accurate Attitude Estimation of a Moving Land Vehicle Using Low-Cost MEMS IMU Sensors. IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS, VOL. 18, NO. 7, JULY 2017. Pages 1723-1739.