DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Técnicas de los Sistemas Inteligentes. Curso 2015-16. Práctica 1: Robótica

Entrega 2: Guía para la implementación de costmaps

Objetivo

En este documento se encuentra una guía para poder llevar a cabo las tareas 2.b y 2.c de la segunda entrega. La idea es que se dedique una persona a documentarse sobre cómo manejar costmaps en ROS y cómo implementar un proceso de búsqueda basado en un costmap para determinar una trayectoria segura (dentro del plano representado por el costmap) para ir de una posición inicial a una posición destino. El documento tiene dos partes, una guía para implementar costmaps, donde se usan conceptos generales de costmaps, y una guía para la implementación de un planificador local con costmaps, basada en la arquitectura cliente servidor que estamos implementando en esta entrega.

Guía para implementar costmaps.

1. ¿Qué es un costmap?

Un costmap es una representación icónica, generalmente una matriz bidimensional (aunque también puede ser tridimensional), de un mapa o de una porción de un mapa. Puede entenderse como una matriz bidimensional de enteros positivos (tipo unsigned char, intervalo [0,255]) cuyas celdas contienen no solo información sobre posiciones libres y ocupadas, sino también información sobre "cómo de segura" es una posición considerando su proximidad a un obstáculo.

2. ¿Qué fuentes de información necesita un costmap?

Un costmap trabaja a partir de información de: **sensores**, **matriz de ocupación (occupancy grid)**, **footprint del robot (la forma geométrica del robot en el plano)** e **información propia de configuración (proporcionada en un fichero de configuración)**. Cuando se crea costmap (mediante la declaración de un objeto de tipo costmap_2d::costmap2DROS) se subscribe automáticamente (*esto no hay que programarlo, y es la ventaja de los costmaps*) a topics de sensores publicados en ROS (tienen que ser sensores de rango porque el costmap guarda información sobre distancias a obstáculos) y necesita conocer la información sobre la matriz de ocupación del mapa en el que está situado. Los topics a los que se subscribe se definen en un fichero de configuración del costmap, la matriz de ocupación la obtiene de algún nodo que publique tal información (normalmente del paquete "mapserver" de ROS) y la información del footprint se representa en el propio fichero de configuración del costmap.

3. ¿Cómo funciona un costmap?

Un costmap accede a información sobre una matriz de ocupación, publicada por algún otro nodo en ROS (normalmente "mapserver") y, a partir de unos parámetros de configuración proporcionados por el usuario, lleva a cabo un procesamiento de la información sensorial del robot (por ejemplo, escaneo láser) que resulta en un poblamiento de las celdas de la matriz del costmap. La información adquirida permite que un robot pueda tomar decisiones sobre movimiento local.

El procesamiento de la información sensorial en un costmap se lleva a cabo automáticamente (**no hay que programarlo**) y en ciclos de actualización de la matriz a una velocidad determinada por un parámetro llamado "update_frequency" (ver más abajo cómo configurar los parámetros de un fichero de configuración de un costmap). En cada ciclo de actualización, el costmap recibe información de sensores y lleva a cabo operaciones de marcado o *marking* (insertar información de un obstáculo en la matriz) y limpieza o *clearing* (eliminar información de obstáculos del costmap).

1. Inflación (Inflation)

Una vez realizado el marcado y limpieza de acuerdo a la nueva información sensorial recibida en cada ciclo, se lleva a cabo un proceso de propagación (denominado *inflación*) de valores desde las celdas ocupadas que van decreciendo de acuerdo a la distancia del obstáculo. El proceso de inflación está bien documentado en http://wiki.ros.org/costmap_2d#Inflation.

4. ¿Qué necesitamos saber para manejar un costmap?

Para manejar un costmap en primer lugar tenemos que saber cómo se pueden manejar mapas en ROS y cómo publicar una matriz de ocupación (esto se hace mediante el paquete mapserver). Esto es así porque un costmap recibe información de la matriz de ocupación de un mapa previamente publicado por map_server.

Es necesario conocer qué topics publican información sensorial del robot. En el caso de un robot simulado por Stage nos interesa el topic "base_scan". También necesitamos saber cómo configurar un costmap, esto aparece más adelante en esta guía. Necesitamos también saber cómo visualizar un costmap, mediante la herramienta rviz, que se ha explicado en la sesión de prácticas.

1. ¿Qué es necesario para que pueda funcionar un costmap?

Teniendo en cuenta los requisitos de información del costmap, para que podamos implementar adecuadamente un nodo ROS que incluya manejo de costmap, al menos deben existir en el entorno de ejecución los siguientes nodos ROS:

1. un nodo que publique la matriz de ocupación (occupancy matrix). Esto lo realiza el paquete map_server.
2. un nodo que publique información sensorial y de odometría. Esto lo hace el simulador del paquete stage_ros.
3. el propio nodo que implementa el costmap, al que se deben pasar los parámetros de configuración en un fichero (más adelante se describen estos parámetros)
4. Usar un nodo para la visualización (rviz), esto nos permitirá visualizar adecuadamente toda la información publicada, incluida la información del costmap.
5. Usar un nodo para publicar información sobre la localización del robot y poder visualizar correctamente al robot en rviz (paquete fake_localization).

El fichero miscotmaps.zip (en PRADO) contiene un paquete con ficheros para manejar costmaps. Descargar el paquete, hacer catkin_make y seguir con esta guía.

¿Cómo se maneja un costmap?

Primero vamos a ver un ejemplo en el que usaremos dos tipos de costmap en un nodo implementado específicamente para ello, que hará la función de publicar información de los costmaps (matriz de costes y footprint del robot). Como vamos a usar costmaps necesitamos también el nodo que publique la matriz de ocupación (del paquete mapserver), como vamos a

visualizar los nodos tenemos que lanzar el visualizador rviz. Como rviz necesita conocer la posición inicial del robot, usaremos un nodo del paquete fake_localization que simplemente se suscribe a la posición del robot publicada por Stage y la publica en el topic "initial_pose" que es el topic al que se suscribe rviz para situar al robot en la posición real. En un entorno real el paquete fake_localization se sustituye por el paquete amcl que estima la posición verdadera del robot mediante ténicas de localización y mapeo simultáneos (ver más información en fake_localization y amcl).

Implementación de objetos costmap en un nodo

La forma más adecuada de manejar un costmap es mediante la declaración de objeto costmap y usar las funciones proporcionadas en ROS mara manejar costmaps. Una vez creado el costmap podremos acceder a la información si cuando ejecutamos el nodo le pasamos adecuadamente los parámetros en los ficheros de configuración.

ROS suministra dos clases fundamentales para manejar costmaps:

1. Costmap_2d::Costmap2DROS que es una interfaz "wrapper" para poder gestionar costmaps en ROS.
2. costmap_2d::Costmap2D : la clase que "realmente" implementa un costmap.

En general, hay que crear un objeto Costmap_2d::Costmap2DROS, pero el acceso a la información del costmap se hacer a partir de objetos de tipo costmap_2d::Costmap2D. Ver las Apis:

- Para costmap_2d::Costmap2DROS http://docs.ros.org/jade/api/costmap_2d/html/classcostmap__2d_1_1Costmap2DROS.html
- Para costmap_2d::Costmap2D http://docs.ros.org/jade/api/costmap_2d/html/classcostmap__2d_1_1Costmap2D.html

Los ejemplos que vienen a continuación están en el paquete miscostmaps en PRADO.

```
#include "ros/ros.h"
#include "tf/transform_listener.h"
#include "costmap_2d/costmap_2d_ros.h"

int main(int argc, char** argv){
 ros::init(argc, argv, "miscostmaps_node");
 ros::NodeHandle nh;
 tf::TransformListener tf(ros::Duration(10));
 costmap_2d::Costmap2DROS localcostmap("local_costmap", tf);
 costmap_2d::Costmap2DROS globalcostmap("global_costmap", tf);

double res;
   nh.getParam("local_costmap/resolution",res);
   ROS_INFO("La resolución del costmap local es %f", res);
   ros::spin();

  return(0);
 }
```

Observar que en el código fuente aparecen dos objetos costmap con nombres distintos y además aparece la definición de un objeto "tf". Tf es un paquete de ROS para manejar transformaciones geométricas entre distintos marcos de coordenadas. Es un paquete fundamental para trabajar en ROS adecuadamente, para más información consultar el paquete tf. Ahora mismo nos basta con saber que para crear un costmap tenemos que crear un transform_listener, que básicamente es un objeto que comprueba si hay transformación correcta entre el marco de coordenadas del sistema de odometría y el del mapa. Si el resto de nodos de nuestro ecosistema ROS publican adecuadamente las transformaciones entre marcos de coordenadas, no habrá problema.

Respecto a los dos objetos costmap, es importante tener en cuenta que hay dos formas distintas de inicializar un costmap y nos lleva distinguir dos tipos fundamentales de costmaps, que se declaran con la misma clase, pero que tienen comportamientos distintos:

1. **Global costmap.** Un global costmap guarda información sobre obstáculos a partir del mapa global del entorno donde está situado el robot. Se inicializa alimentándolo a partir de un mapa estático generado por el usuario y que se convierte en una matriz de ocupación mediante el paquete map_server. En este caso el costmap se inicializa automáticamente para ajustarse a las dimensiones del mapa estático y recibe la información sobre ocupación de la matriz de ocupación del mapa (ver map_server).
2. **Local costmap.** Un mapa de coste local guarda información de los obstáculos en el entorno local del robot. El entorno local se define a partir de una altura y anchura (razonablemente pequeñas) y poniendo a true el parámetro "rolling window" del fichero de configuración del costmap. El parámetro "rolling window" hace que el robot

se mantenga en el centro del costmap conforme se mueve en el entorno, además la información sobre obstáculos y espacio libre se va actualizando acordemente. En este sentido, el robot tiene un mapa completamente actualizado de su entorno más cercano y, por tanto, un costmap local es una herramienta muy adecuada para implementar comportamientos de navegación local.

Ficheros de configuración de costmaps.

Ahora es el momento de conocer qué contienen los ficheros de configuración de costmaps.

Para ejecutar correctamente el nodo debemos pasarle los parámetros de configuración de los costmaps. Los parámetros de un costmap están definidos y explicados en http://wiki.ros.org/costmap_2d#Component_API. Podemos capturar sus valores si nos interesara con nh getparam() pero la API de ROS para costmaps suministra funciones para manejar estos elementos. A continuación un ejemplo del fichero launch "miscostmaps_fake_5cm.launch" donde se lanza el nodo con el código fuente anterior y, además, nodos map_server, stage_ros, fake_localization y rviz.

```
<launch>
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>
  <!-- include file="$(find navigation_stage)/move_base_config/move_base.xml"/ -->
   <node name="miscostmaps_node" pkg="miscostmaps" type="miscostmaps_node" output="screen">
      <rosparam file="$(find miscostmaps)/configuration/costmap_common_params.yaml" command="load"
ns="global_costmap" />
          <rosparam file="$(find miscostmaps)/configuration/costmap_common_params.yaml" command="load"
ns="local_costmap" />
          <rosparam file="$(find miscostmaps)/configuration/local_costmap_params.yaml"  command="load" />
          <rosparam file="$(find miscostmaps)/configuration/global_costmap_params.yaml" command="load" />
        </node>
   <node name="map_server" pkg="map_server" type="map_server" args="$(find
miscostmaps)/maps/simple_rooms.yaml" respawn="false"/>


  <node pkg="stage_ros" type="stageros" name="stageros" args="$(find
mistage1516)/configuracion/mundos/mi-simplerooms.world" respawn="false">
             <param name="base_watchdog_timeout" value="0.2"/>
        </node>
 <node name="fake_localization" pkg="fake_localization" type="fake_localization" respawn="false" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find navigation_stage)/single_robot.rviz" />
</launch>
```

En el código ejemplo para llamar al nodo utilizamos 3 ficheros de parámetros (ver su contenido en el paquete miscostmaps directorio "configuration" subido a la plataforma PRADO):

1. Costmap_common_params.yaml: se configuran parámetros comunes a ambos costmaps como, por ejemplo, qué **topics** relativos a fuentes sensoriales

(observation_sources) se van a usar (especificando el topic y tipo de mensaje entre otras cosas), qué **valor de coste del costmap usamos como umbral** para, a partir de él, considerar que la celda que contiene ese valor es un obstáculo, qué **radio consideramos para hacer la inflación** (en este ejemplo 55 cm, es decir, los valores de las casillas a menos de 55 cm de un obstáculo tendrán un valor distinto al de FREE_SPACE), qué **dimensiones tiene el footprint del robot** (especificado en este caso como los puntos de un pentágono "con pico"), etc.

2. Local_costmap_params.yaml: parámetros exclusivos del costmap local:
    a. a qué frecuencia se modifica el costmap (**update frequency**). Este valor es **importante** porque hay que asignarlo considerando la frecuencia a la que se actualiza el escaneo láser y la frecuencia a la que se envían órdenes al robot.
    b. **Frecuencia de publicación**: este valor es **fundamental** porque por defecto está a 0 (significa no publicar el costmap) puede dar dolores de cabeza si no se pone a un valor adecuado, por ejemplo 2 o 5 Hz.
    c. Configuración del **rolling window** (ventana que avanza junto al robot):
        i. rolling_window:**tiene que estar a true**,
        ii. las **dimensiones de la ventana** (width y height, en metros),
        iii. el **origen de coordenadas** (dejarlo a origin_x = origin_y = 0)
        iv. **la resolución** (resolution en metros/celda) indica "**cómo de grandes son las celdas del costmap"**, a mayor resolución, las celdas serán más pequeñas, pero esto afectará al tamaño del costmap y por tanto a la eficiencia de cualquier algoritmo que lo recorra.
3. Global_costmap_params.yaml: son los mismos parámetros que los del local costmap pero con valores distintos, especialmente **rolling_window tiene que ser false** (el global costmap no se desplza con el robot), por tanto es estático (**static_map = true**) y es importante especificar en **qué topic se publica el mapa** del que se nutre el costmap local que en nuestro caso es "/map".

Consultar esos ficheros en el código suministrado para observar la sintaxis sencilla (el fichero contiene comentarios) para asignar valores a los parámetros. Hacer a continuación lo siguiente:

1. Ejecutar el fichero launch "miscostmaps_fake_5cm.launch".
2. Añadir en rviz un nuevo display de tipo Map y asociar el topic "miscotmaps_node/local_costmap/costmap" observar que aparece el costmap local.
3. Hacer lo mismo con el topic "miscotmaps_node/global_costmap/costmap

Configuración dinámica de parámetros: paquete rqt_reconfigure.

Los parámetros configurados pueden cambiarse dinámicamente en tiempo de ejecución con la herramienta "rqt_reconfigure" del paquete rqt_reconfigure http://wiki.ros.org/rqt_reconfigure .

Ejecutar rosrun rqt_reconfigure rqt_reconfigure, observar la ventana que aparece y cambiar los parámetros. Por ejemplo, observar cómo al cambiar la resolución del costmap podemos tener un costmap más o menos pixelado, dependiendo de la menor o mayor resolución, respectivamente.

Guía para implementar un planificador local con costmaps, basado en una arquitectura cliente-servidor

Nuestro objetivo es implementar el comportamiento de un planificador local de movimientos con costmaps, es decir, que pueda acceder a la información de un mapa local para poder tomar decisión, y visualizarlo en rviz. Disponemos de una arquitectura cliente-servidor en la que el cliente envía goals hacia un servidor para que éste controle al robot en la consecución del goal (según la técnica de campos de potencial). En concreto, aunque hay varias alternativas y esta no tiene que ser la mejor, vamos a tomar como una decisión de diseño implementar el costmap en el lado del cliente porque nuestra intención última es desarrollar una técnica para sacar automáticamente de un atasco al robot, mediante la búsqueda de un punto dentro del costmap (y una trayectoria desde la posición actual al punto destino), al que se tendrá que dirigir el robot para salir del atasco.
A continuación vamos a conocer cómo usar un costmap en un cliente dentro de la arquitectura cliente-servidor, y veremos ejemplos de cómo imprimir un costmap, cómo calcular las celdas vecinas a una dada en un costmap, y qué cosas hay que tener en cuenta para hacer una función de búsqueda de una trayectoria dentro de un costmap.
El código fuente de ayuda para esta guía está paquete "action_lib_costmaps" en PRADO.

1. *Hacer que el nodo ros haga spin en una hebra aparte (Threaded Cliente).*

Un aspecto fundamental para que funcionen los costmaps es llamar al método ros::spin(), en caso contrario los costmaps no comenzarían su funcionamento en "background" para actualizar su información.

La cuestión es que si utilizamos una llamada a ros::spin(), la función main() desplaza el control al método ros::spin() y se queda en espera hasta que ros::spin() finaliza (e.d., hasta que el nodo ros muere por alguna razón). Si hacemos esto en el cliente, todo el procesamiento del cliente (envío de goals y monitorización) no se puede ejecutar. Por tanto necesitamos lanzar el método ros::spin() como una hebra y permitir que la función main() del cliente siga su curso.

El código fuente suministrado para un cliente con costmaps hace esto, para más información consultar el tutorial http://wiki.ros.org/actionlib_tutorials/Tutorials/SimpleActionClient(Threaded)

Observar cómo se lanza ros::spin como una hebra. Observar también que se hace una llamada a pause() y start() para cada uno de los costmaps. Observar también cómo se finaliza el nodo, haciendo un join de la hebra.

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
//gestion de costmaps
#include "tf/transform_listener.h"
#include "costmap_2d/costmap_2d_ros.h"
#include <costmap_2d/costmap_2d.h>
//gestion de hebras
#include <boost/thread.hpp>
//consultado en http://wiki.ros.org/actionlib_tutorials/Tutorials/SimpleActionClient%28Threaded%29

void spinThread()
{
  ros::spin();
}

int main(int argc, char** argv) {

  ros::init(argc,argv, "send_goals_node");
  //el cliente usarn costmap local para poder tener una representaci□el entorno
  //en su vecindad pr□a.
  tf::TransformListener tf(ros::Duration(10));
  costmap_2d::Costmap2DROS* localcostmap = new costmap_2d::Costmap2DROS("local_costmap", tf);
  costmap_2d::Costmap2DROS* globalcostmap = new costmap_2d::Costmap2DROS("global_costmap", tf);
  //detener el funcionamiento de los costmaps hasta que estemos conectados con el servidor.
  localcostmap->pause();
  globalcostmap->pause();
  //crear el "action client"
  MoveBaseClient ac("mi_move_base", true);  //<- poner "mi_move_base" para hacer mi propio action server.

  //creamos una hebra para que ros::spin se haga en paralelo y no interrumpa el procesamiento del cliente.
  //ros::spin es necesario para que los costmaps se actualicen adecuadamente.
  boost::thread spin_thread(&spinThread);

  //Esperar 60 sg. a que el action server est頻evantado.
  ROS_INFO("Esperando a que el action server move_base se levante");
  //si no se conecta, a los 60 sg. se mata el nodo.
  ac.waitForServer(ros::Duration(60));

  ROS_INFO("Conectado al servidor mi_move_base");

  //Arrancar los costmaps una vez que estamos conectados al servidor
  localcostmap->start();
  globalcostmap->start();
  ...
  /Esperar al retorno de la acci□span>
  ROS_INFO("Esperando al resultado  de la acci□/span>");
  ac.waitForResult();

  if (ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    ROS_INFO(" Objetivo alcanzado !!");
  else ROS_INFO("Se ha fallado por alguna raz□);

  // shutdown the node and join the thread back before exiting
  ros::shutdown();
  spin_thread.join();


  return 0;

}
```

2. Imprimir un costmap

```
//imprime en pantalla el contenido de un costmap wrapper de ROS
void printCostmap (costmap_2d::Costmap2DROS * costmap_ros) {
     //typedef basic_string<unsigned char> ustring;


     costmap_2d::Costmap2D *costmap;
     costmap = costmap_ros-> getCostmap();

     for (int x=0 ;x < costmap->getSizeInCellsX() ;x++)
          for (int y= 0; y < costmap->getSizeInCellsY();y++){
              std::cout << (int) costmap->getCost(x,y) << " ";
            }
        }
```

Como ejemplo para ver cómo acceder a la información de un costmap consultar el código fuente de la función void printCostmap(…). Observar que recibe como parámetro un puntero a un costmap_2d::Costmap2DROS y en el interior accede a la información del costmap mediante el método costmap_ros->getCostmap(), asignándole el resultado a un puntero a objeto costmap2d::Costmap2D. Observar también el uso de los métodos getSizeInCellsX() y getSizeInCellsY() para acceder a las dimensiones del costmap y el uso de getCost(x,y) para acceder al valor del costmap en la posición (x,y). Tener en cuenta que aunque el costmap se represente como una matriz de unsigned chars, las coordenadas de acceso a los elementos a través de las función getCost(x,y) se interpretan como coordenadas cartesianas (x horizontal y vertical) no como filas y columnas.

3. Celdas vecinas a una dada.

```
std::vector <unsigned int> findFreeNeighborCell
        (unsigned int CellID, costmap_2d::Costmap2D *costmap){

unsigned int mx, my;
        //convertir el índice de celda en coordenadas del costmap
        costmap->indexToCells(CellID,mx,my);
        ROS_INFO("CellID = %d, mx = %d, my= %d", CellID, mx, my);
        std::vector <unsigned int>  freeNeighborCells;

        for (int x=-1;x<=1;x++)
          for (int y=-1; y<=1;y++){
            //comprobar si el índice es válido
           if ((mx+x>=0)&&(mx+x < costmap->getSizeInCellsX())
                        &&(my+y >=0 )&&(my+y < costmap->getSizeInCellsY())){

              ROS_WARN("Index valid!!!!! costmap[%d,%d] = %d",
                               mx+x,my+y,costmap->getCost(mx,my));
              if(costmap->getCost(mx+x,my+y) == costmap_2d::FREE_SPACE    && (!(x==0 && y==0))){
                  unsigned int index = costmap->getIndex(mx+x,my+y);
                  ROS_INFO("FREECELL: %d con valor %d", index, costmap_2d::FREE_SPACE);
                  freeNeighborCells.push_back(index);
              }
              else {
                  unsigned int index = costmap->getIndex(mx+x,my+y);
                  ROS_INFO("OCCUPIEDCELL: %d con valor %d", index, costmap->getCost(mx+x,my+y));
                 }
           }
        }
          return  freeNeighborCells;
```

La función vector<unsigned int> findFreeNeighborCell(unsigned int CellID, costmap_2d::Costmap2D *costmap) devuelve un conjunto de celdas vecinas a una dada (representada como un índice representando la posición del costmap considerado como un vector lineal) , a partir de un puntero al contenido de un costmap. Observar el uso de la función indexToCells(CellID, mx,my) que devuelve en mx y my las coordenadas del costmap asociadas al índice CellID. Observar también el uso de la función inversa getIndex(mx,my) que devuelve el índice de una posición dada del costmap.

4. *Buscar una trayectoria entre dos puntos dados.*

```
bool localPlanner(const geometry_msgs::PoseStamped& start, const geometry_msgs::PoseStamped& goal,
std::vector<geometry_msgs::PoseStamped>& plan, costmap_2d::Costmap2DROS *costmap_ros){


    ROS_INFO("localPlanner: Got a start: %.2f, %.2f, and a goal: %.2f, %.2f", start.pose.position.x,
start.pose.position.y, goal.pose.position.x, goal.pose.position.y);

    plan.clear();
    costmap_2d::Costmap2D *costmap = costmap_ros->getCostmap();

    //pasamos el goal y start a coordenadas del costmap
    double goal_x = goal.pose.position.x;
    double goal_y = goal.pose.position.y;
    unsigned int mgoal_x, mgoal_y;
    //transformamos las coordenadas del mundo a coordenadas del costmap
    costmap->worldToMap(goal_x,goal_y,mgoal_x, mgoal_y);
    if ((mgoal_x>=0)&&(mgoal_x < costmap->getSizeInCellsX())&&(mgoal_y>=0 )&&(mgoal_y < costmap-
>getSizeInCellsY()))
      unsigned int indice_goal = costmap->getIndex(mgoal_x, mgoal_y);
    else ROS_WARN("He recibido unas coordenadas del mundo para el objetivo que está fuera de las
dimensiones del costmap");

    //transformamos las coordenadas de la posición inicial a coordenadas del costmap.
    double start_x = start.pose.position.x;
    double start_y = start.pose.position.y;
    unsigned int mstart_x, mstart_y;
    costmap->worldToMap(start_x,start_y, mstart_x, mstart_y);
    unsigned int start_index = costmap->getIndex(mstart_x, mstart_y);


    //*****************************************************************************
    // ya tenemos transformadas las coordenadas del mundo a las del costmap,
    // ahora hay que implementar la búsqueda de la trayectoria, a partir de aquí.
    //*****************************************************************************
    }
```

En el código fuente hay una cabecera de función para implementar un proceso de búsqueda y así aproximarnos a la idea de la planificación local con mapa. La idea es devolver una trayectoria segura cercana a la óptima entre una posición de inicio y una posición objetivo, dentro del costmap local.

Observar que antes de empezar con el proceso de búsqueda hay que transformar las coordenadas del mundo de los parámetros de entrada a coordenadas del costmap. Las primeras líneas implementadas en la función son para ello.

La trayectoria obtenida (plan) puede representarse internamente en la función como se quiera, por ejemplo como una lista de índices de celdas. Pero hay que tener en cuenta que una vez conseguida una lista de celdas, cada elemento lista[i] tiene que convertirse a coordenadas del costmap y éstas a coordenadas del mundo mediante las funciones indexToCells(lista[i], mx,my) y mapToWorld(mx,my,wx,wy), respectivamente. Además, para devolver

adecuadamente el plan como un objeto de tipo std::vector<geometry_msgs::PoseStamped> hay que convertir cada coordenada del mundo (wx,wy) en un elemento de tipo <geometry_msgs::PoseStamped>. Esto se puede hacer con la función rellenaPoseStamped(double x, double y, geometry_msgs::PoseStamped &pose, costmap_2d::Costmap2DROS *costmap) que está en el código fuente.