Guía confeccionada en el marco de la beca de servicio

Índice de contenidos

1 - Paquetes

El software en ROS se organiza en paquetes. Los paquetes pueden contener nodos, librerias, configuraciones, software externo a ROS, cualquier cosa que resulte de utilidad en el paquete.

1.1 - Estructura

`include/package_name`: C++ include headers

`msg/`: Directorio que contiene tipos de mensaje

`src/package_name/`: Source files

`srv/`: Directorio que contiene tipos de servicios

`scripts/`: Scripts ejecutables

`CMakeLists.txt`: CMake build file

`package.xml`: Archivo que define las propiedades del paquete

`CHANGELOG.rst`: —

1.2 - Creación

Dentro de `src` en el workspace creado

```
catkin_create_pkg <nombre_paquete> [dependencia1] [dependencia2]
```

Si luego de creado el paquete se quieren agregar más dependencias, editar el archivo package.xml y agregar:

```
<build_depend>dependencia</build_depend>
<build_export_depend>dependencia</build_export_depend>
<exec_depend>dependencia</exec_depend>
```

Para usar ROS con C++ hay que agregar `roscpp`, y `rospy` en el caso de Python.

Dentro del workspace ejecutar

```
catkin_make
```

Este comando construye cualquier proyecto que se encuentre en la carpeta `src`

2 - Nodos

Un nodo es un archivo ejecutable dentro de un paquete de ROS. Los nodos pueden publicar o suscribir a un `topic` para comunicarse con otros nodos. También pueden proveer o usar un servicio.

`rosnode` muestra información sobre los nodos que se están ejecutando. Si se lo ejecuta sin argumentos muestra los distintos comandos posibles.

2.1 - Publisher

Se trata de nodos que publican información en la red de ROS.

2.1.1 - Estructura básica

Incluir el header de ROS para hacer uso de sus funcionalidades

```
#include "ros/ros.h"
```

Incluir el header correspondiente al tipo de mensaje que necesitamos. En este caso `String.h`, que es parte de `std_msgs`

```
#include "std_msgs/String.h"
```

Primero es necesario inicializar el nodo, que puede recibir los argumentos de la función `main` y un nombre, en este caso "`talker`"

```
ros::init(argc, argv, "talker");
```

Luego de inicializado el nodo, creamos un `NodeHandle` para realizar funciones como suscribir a un `topic` o anunciarlo para publicar.

```
ros::NodeHandle nh;
```

En este caso se trata de un nodo que publica un string en un topic que llamaremos "`chatter`". Haciendo uso del `NodeHandle` definido, lo anunciamos

```
ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("chatter",1000);
```

Esta llamada se conecta con el master para anunciar que el nodo estará publicando mensajes en el topic dado. Devuelve un `Publisher`, en este caso `chatter_pub`, que permitirá publicar un mensaje en este topic. El segundo parámetro es el máximo de mensajes salientes que se pondrán en la cola para enviar a los subscriptores.

Podemos hacer uso de la clase `Rate` para determinar la frecuencia del bucle. En este caso 10[Hz] pasados en el constructor.

```
 ros::Rate loop_rate(10);
```

Declaramos un mensaje del tipo String en el que alojaremos la información para publicarla

```
std_msgs::String msg;
```

La condición `ros::ok()` devolverá false cuando el nodo se detenga. La usamos como condición del bucle

```
while (ros::ok())
{
      ...
}
```

Dentro del bucle guardamos el string “hello world “ junto al valor del contador dentro del miembro `data` del mensaje.

ROS_INFO es una de las formas de registrar información de los nodos y presentarla de manera legible. Además de INFO existen: DEBUG, WARN, ERROR y FATAL.

```
ROS_INFO("%s", msg.data.c_str());
```

Para publicar el mensaje al topic ya determinado, usamos la función `publish()` del `Publisher`, en nuestro `caso chatter_pub`, pasando como argumento el mensaje `msg`.

```
chatter_pub.publish(msg);
```

ros::spinOnce();

Finalmente llamamos a la función `sleep()` para corroborar que se cumpla la condición de frecuencia determinada anteriormente. Esperará si falta tiempo para completar el ciclo, y una vez cumplido devolverá `true`

```
loop_rate.sleep();
```

Codigo completo

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle nh;
    ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("chatter",1000);
    ros::Rate loop_rate(10);
    std_msgs::String msg;
    int count = 0;
    while (ros::ok())
    {
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```

2.2 - Subscriber

Se trata de nodos que toman información en la red de ROS suscribiéndose a topics existentes.

2.2.1 - Estructura básica

Incluimos los headers necesarios siguiendo la misma lógica que en el ejemplo del publisher.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
```

La inicialización del nodo es similar al publisher, pero esta vez llamaremos al nodo "`listener`"

```
ros::init(argc, argv, "listener");
```

Declaramos el NodeHandle

```
ros::NodeHandle nh;
```

Esta vez utilizamos la función `subscribe()` para que el nodo reciba mensajes a través del topic "`chatter`". Cuando un mensaje es recibido, se pasan una función callback, en este caso chatterCallback.

```
ros::Subscriber sub = nh.subscribe("chatter", 1000, chatterCallback);
```

La función `chatterCallback()` toma el mensaje recibido y lo muestra por consola haciendo uso de `ROS_INFO()`. Su definición es la siguiente:

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
      ROS_INFO("Escuché: [%s]", msg->data.c_str());
}
```

Finalmente, `ros::spin()` se ocupa de que el programa entre en un bucle que siga recibiendo los mensajes y ejecutando la función callback mientras el nodo se esté ejecutando.

Codigo completo

```
#include "ros/ros.h"
#include "std_msgs/String.h"
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("Escuché: [%s]", msg->data.c_str());
}
int main(int argc, char **argv)
{
    ros::init(argc, argv, "listener");
    ros::NodeHandle nh;
    ros::Subscriber sub = nh.subscribe("chatter", 1000, chatterCallback);
    ros::spin();
    return 0;
}
```

2.3 - Building

Dentro del paquete donde se encuentra el código de nuestros nodos, editar el archivo CMakeLists.txt y agregar para cada nodo:

```
add_executable(nombre_nodo src/codigo_nodo.cpp)
target_link_libraries(nombre_nodo
      ${catkin_LIBRARIES}
)
```

En el caso de existir varios archivos que componen al nodo, como en el caso de clases definidas en archivos separados, se agregan dichos archivos seguidos del primer .cpp:

```
add_executable(nombre_nodo src/codigo_nodo.cpp src/clase1.cpp src/clase2.cpp)
```

Seguido de esto, estando en el directorio del `workspace`, ejecutamos:

```
catkin_make
```

2.4 - Ejecución

Para ejecutar un nodo, primero debe estar corriendo el núcleo de ROS. Para esto usamos el comando:

```
roscore
```

2.4.1 - Ejecución directa

Ahora podemos correr el nodo en otra terminal utilizando el comando `rosrun`, que nos permite ejecutar un nodo directamente usando el nombre del paquete que lo contiene

```
rosrun [package_name] [node_name]
```

Para correr el nodo con otro nombre, agregar el siguiente argumento de reasignación al final:

```
__name:=nuevo_nombre
```

Si al listar los nodos aparece uno que no debería, probar:

2.4.2 - Launch files

`roslaunch` es un paquete que contiene herramientas para leer y ejecutar archivos de formato `.launch/XML`. Toma como argumentos uno o más archivos `.launch`.

```
roslaunch [package_name] [launch-filename] [args]
```

Los archivos launch usualmente se usan para correr varios nodos simultaneamente.

Ejemplo de launch file para los nodos vistos en la sección 2.1.1 y 2.2.1:

```
<launch>
      <node name="listener_node" pkg="test_package"
            type="listener" output="screen"/>
      <node name="talker_node" pkg="test_package"
            type="talker" output="screen"/>
</launch>
```

2.x - Otros

```
rosnode cleanup
```

Una forma de chequear que un nodo está corriendo es hacerle ping

```
rosnode ping [nombre del nodo]
```

4 - Transformaciones

`tf` es un paquete que permite mantener un seguimiento de múltiples marcos de coordenadas en el tiempo. Relaciona los distintos marcos organizándolos en una estructura de árbol almacenada en el tiempo, y permite transformar puntos, vectores, etc entre dos marcos de coordenadas cualesquiera en cualquier momento del tiempo.

(Usar `tf2`, `tf` está obsoleto desde ROS Hydro)

Esencialmente tf2 se usa para escuchar y transmitir transformadas, aplicaciones que veremos a continuación. Para esto crearemos un paquete siguiendo los pasos indicados en la sección 1, en este caso lo llamaremos `tf2_test`. Podemos hacerlo mediante la extensión de ROS para VSCode o el siguiente comando:

```
catkin_create_pkg tf2_test tf2 tf2_ros roscpp turtlesim
```

Como podemos ver, este paquete dependerá de `tf2`, `tf2_ros`, `roscpp` y `turtlesim`.

4.1 - Broadcaster

Un `broadcaster` (transmisor) es quien se ocupa de transmitir transformadas, es decir, enviar la pose relativa de marcos de coordenadas al resto del sistema. Un sistema puede tener varios transmisores que proveen información sobre diferentes partes de un robot.

4.1.1 - Estructura básica

Vamos a necesitar los siguientes headers:

```
#include <ros/ros.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_ros/transform_broadcaster.h>
#include <geometry_msgs/TransformStamped.h>
#include <turtlesim/Pose.h>
```

`tf2/LinearMath/Quaternion.h`: Para implementar quaterniones que nos servirán para realizar rotaciones de álgebra lineal.
`tf2_ros/transform_broadcaster.h`: Esta clase provee una forma fácil de publicar información sobre las transformadas de los marcos de coordenadas. Maneja los mensajes y su contenido.

```
ros::init(argc, argv, "test_tf2_broadcaster");
ros::NodeHandle private_node("~");
if (! private_node.hasParam("turtle"))
{
      if (argc != 2){ROS_ERROR("need turtle name as argument"); return -1;};
      turtle_name = argv[1];
}else private_node.getParam("turtle", turtle_name);
```

```
static tf2_ros::TransformBroadcaster br;
```

Dentro de la función `poseCallback()` crearemos un objeto `TransformBroadcaster` que más tarde usaremos para enviar las transformaciones.

```
geometry_msgs::TransformStamped transformStamped;
transformStamped.header.stamp = ros::Time::now();
transformStamped.header.frame_id = "world";
transformStamped.child_frame_id = turtle_name;
```

Dentro de la misma función creamos un mensaje del tipo `TransformStamped` y le asignamos la información correspondiente. El miembro `header` se utiliza generalmente para comunicar información con marcas de tiempo en un marco de coordenadas particular, por eso en `header.stamp` cargamos el momento actual y en `header.frame_id` el marco con el que se asocia esta información.

En el miembro `child_frame_id` asignaremos el identificador del marco secundario (child frame), que en este caso será el nombre de la tortuga.

```
transformStamped.transform.translation.x = msg->x;
transformStamped.transform.translation.y = msg->y;
transformStamped.transform.translation.z = 0.0;
tf2::Quaternion q;
q.setRPY(0, 0, msg->theta);
transformStamped.transform.rotation.x = q.x();
transformStamped.transform.rotation.y = q.y();
transformStamped.transform.rotation.z = q.z();
transformStamped.transform.rotation.w = q.w();
```

Además guardamos la información de posición de la tortuga en el plano y su ángulo theta en `transformStamped.transform`. La función `setRPY()` setea los valores del quaternion de acuerdo a los parámetros de yaw, pitch y roll.

```
br.sendTransform(transformStamped);
```

Finalmente enviamos la información mediante el objeto `br` definido al comienzo de la función.

Código completo

```cpp
#include <ros/ros.h>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_ros/transform_broadcaster.h>
#include <geometry_msgs/TransformStamped.h>
#include <turtlesim/Pose.h>

std::string turtle_name;

void poseCallback(const turtlesim::PoseConstPtr& msg){
  static tf2_ros::TransformBroadcaster br;
  geometry_msgs::TransformStamped transformStamped;

  transformStamped.header.stamp = ros::Time::now();
  transformStamped.header.frame_id = "world";
  transformStamped.child_frame_id = turtle_name;
  transformStamped.transform.translation.x = msg->x;
  transformStamped.transform.translation.y = msg->y;
  transformStamped.transform.translation.z = 0.0;
  tf2::Quaternion q;
  q.setRPY(0, 0, msg->theta);
  transformStamped.transform.rotation.x = q.x();
  transformStamped.transform.rotation.y = q.y();
  transformStamped.transform.rotation.z = q.z();
  transformStamped.transform.rotation.w = q.w();

  br.sendTransform(transformStamped);
}

int main(int argc, char** argv){
  ros::init(argc, argv, "test_tf2_broadcaster");

  ros::NodeHandle private_node("~");
  if(!private_node.hasParam("turtle"))
  {
    if(argc!=2){ROS_ERROR("necesita un nombre de tortuga como argumento");return -1;};
    turtle_name = argv[1];
  }else private_node.getParam("turtle", turtle_name);

  ros::NodeHandle node;
  ros::Subscriber sub = node.subscribe(turtle_name+"/pose", 10, &poseCallback);

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

4.2 - Listener

Es el programa que se ocupa de recibir y almacenar todos los marcos de coordenadas que son transmitidos al sistema, y solicitar transformadas específicas entre distintos marcos.

4.2.1 - Estructura básica

```
#include <ros/ros.h>
#include <tf2_ros/transform_listener.h>
#include <geometry_msgs/TransformStamped.h>
#include <geometry_msgs/Twist.h>
#include <turtlesim/Spawn.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "test_tf2_listener");

  ros::NodeHandle node;

  ros::service::waitForService("spawn");
  ros::ServiceClient spawner =
    node.serviceClient<turtlesim::Spawn>("spawn");
  turtlesim::Spawn turtle;
  turtle.request.x = 4;
  turtle.request.y = 2;
  turtle.request.theta = 0;
  turtle.request.name = "turtle2";
  spawner.call(turtle);

  ros::Publisher turtle_vel =
    node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);

  tf2_ros::Buffer tfBuffer;
  tf2_ros::TransformListener tfListener(tfBuffer);

  ros::Rate rate(10.0);
  while (node.ok()){
    geometry_msgs::TransformStamped transformStamped;
    try{
      transformStamped = tfBuffer.lookupTransform("turtle2", "turtle1",
                                ros::Time(0));
    }
    catch (tf2::TransformException &ex) {
      ROS_WARN("%s",ex.what());
      ros::Duration(1.0).sleep();
      continue;
    }

    geometry_msgs::Twist vel_msg;

    vel_msg.angular.z = 4.0 * atan2(transformStamped.transform.translation.y,
                                    transformStamped.transform.translation.x);
    vel_msg.linear.x = 0.5 * sqrt(pow(transformStamped.transform.translation.x, 2) +
                                  pow(transformStamped.transform.translation.y, 2));
    turtle_vel.publish(vel_msg);

    rate.sleep();
  }
  return 0;
};
```

5 - Utilidades

rqt_plot

`rqt_plot` muestra un gráfico de la información publicada en los topics a través del tiempo.

Para ejecutar `rqt_plot`:

```
rosrun rqt_plot rqt_plot
```

En la ventana nueva que debería aparecer, un cuadro de texto en la esquina superior izquierda nos permite añadir cualquier topic al gráfico.
Presionar el botón - (menos) muestra un menú que permite sacar del gráfico cualquer topic especificado.

Rosed

`rosed` es parte de la suite de `rosbash`. Permite editar un archivo directamente dentro de un paquete usando el nombre del paquete en lugar de tipear la dirección completa del paquete

```
rosed [package_name] [filename]
```

El editor por defecto es vim.

6 - Aplicaciones

6.1 - Arduino

Podemos usar Arduino para correr en éste nodos que suscriban y publiquen mientras maneja sensores y actuadores. Para esto necesitamos agregar la librería de ROS al IDE de Arduino.

Primero instalamos `rosserial` para comunicarnos con la placa mediante comunicación por serie

```
sudo apt-get install ros-noetic-rosserial-arduino

sudo apt-get install ros-noetic-rosserial
```

Agregar la librería de ROS a la carpeta de librerías de Arduino:

```
rosrun rosserial_arduino make_libraries.py /home/user/Arduino/libraries/
```

Después de esto debería aparecer `ros_lib` en la lista de ejemplos del IDE de arduino bajo “Ejemplos de Librerías Personalizadas”.

6.1.1 - Detector de obstáculos

Este paquete se puede usar en conjunto con cualquier simulación del TurtleBot. Usa el lidar integrado para encontrar obstáculos en 3 direcciones, al frente y a los lados. En función de lo que detecta, se detiene y cambia la dirección del robot para esquivar cualquier obstáculo.

También hay un nodo implementado en Arduino que enciende 3 leds indicando la posición del obstáculo en la simulación.
Este nodo a su vez detecta la presencia de una persona através de un sensor PIR conenctado al Arduino con el cual detiene el robot dentro de la simulación.

laser_test.cpp

```cpp
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"
#include "geometry_msgs/Vector3.h"
#include <iostream>

#define ADEL 0
#define IZQ 1
#define DER 2

using namespace std;

class Scanner
{
    private:
        double check_dist = 0.6;
        double info_scan[3] = {0.0,0.0,0.0};
        geometry_msgs::Vector3 obst_dir;
        ros::NodeHandle nh;
        ros::Publisher info_obst_pub;
        ros::Subscriber laser_scan_sub;
    public:
        Scanner();
        void laserScanMsgCallBack(const sensor_msgs::LaserScan::ConstPtr &msg);
        void detectarObstaculos();
        void publicar();
};
```

```
Scanner::Scanner()
{
    info_obst_pub = nh.advertise<geometry_msgs::Vector3>("obst_pos", 1);
    laser_scan_sub = nh.subscribe("scan", 10, &Scanner::laserScanMsgCallBack, this);
}

void Scanner::laserScanMsgCallBack(const sensor_msgs::LaserScan::ConstPtr &msg)
{
    int i, angulo[3] = {0,30,330};                      //0° adelante; 30° izquierda;
330° derecha
    for(i = 0; i < 3; i++)
        if (std::isinf(msg->ranges.at(angulo[i])))
            info_scan[i] = msg->range_max;              //Si no hay obstaculos
(distancia infinita) toma el rango máximo
        else
            info_scan[i] = msg->ranges.at(angulo[i]);   //Toma la información
escaneada a en la dirección correspondiente
}

void Scanner::detectarObstaculos()
{
    if(info_scan[ADEL] <= check_dist) obst_dir.x = 1;
    else obst_dir.x = 0;

    if(info_scan[IZQ] <= check_dist) obst_dir.y = 1;
    else obst_dir.y = 0;

    if(info_scan[DER] <= check_dist) obst_dir.z = 1;
    else obst_dir.z = 0;
}

void Scanner::publicar()
{
    info_obst_pub.publish(obst_dir);
}


int main(int argc, char **argv)
{
    ros::init(argc, argv, "laser_test");
    Scanner scanner;

    ros::Rate loop_rate(125);

    while (ros::ok())
    {

        scanner.detectarObstaculos();
        scanner.publicar();

        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

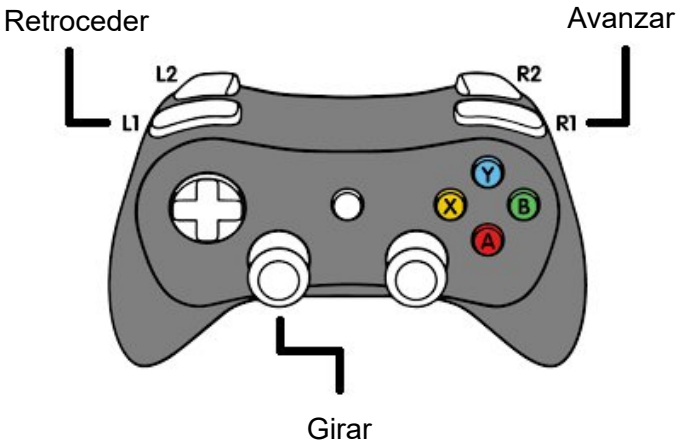
laser_sub_test.ino

```
#include <ros.h>
#include <geometry_msgs/Vector3.h>
#include <std_msgs/Bool.h>

#define LED_ADEL 3
#define LED_IZQ 4
#define LED_DER 2
#define PIN_PIR 8

ros::NodeHandle  nh;
std_msgs::Bool pir_data;

void msgCallBack( const geometry_msgs::Vector3 &vector_msg){
  digitalWrite(LED_ADEL, (int)vector_msg.x);
  digitalWrite(LED_IZQ, (int)vector_msg.y);
  digitalWrite(LED_DER, (int)vector_msg.z);
}

ros::Subscriber<geometry_msgs::Vector3> sub("obst_pos", &msgCallBack );
ros::Publisher pub_pir("det_mov", &pir_data);

void setup()
{
  pinMode(LED_ADEL, OUTPUT);
  pinMode(LED_IZQ, OUTPUT);
  pinMode(LED_DER, OUTPUT);
  nh.initNode();
  nh.advertise(pub_pir);
  nh.subscribe(sub);
}

void loop()
{
  pir_data.data = digitalRead(PIN_PIR);

  pub_pir.publish(&pir_data);
  nh.spinOnce();
  delay(1);
}
```

6.2 - Joystick

Este nodo lee los eventos de un joystick, tomando los valores de un eje y dos botones para controlar la velocidad angular y lineal del robot, que toma esta información a través del topic `cmd_vel`.

Elegí los siguientes controles:



Consideraciones sobre valores numéricos:

La velocidad máxima de avance y retroceso lineal es de 0.22, de otra manera al robot pierde agarre con la superficie y afecta la dirección.
Los valores que pueden leerse de la posición del eje alcanzan el orden de 30000, por lo que ajusté estos valores para que puedan ser usados como rotación angular dividiéndolos por 10000 como se verá más adelante.

En mi caso usé la API de Linux para leer eventos del joystick, y cree una clase que se encarga de la configuración:

joystick.h

```
#ifndef JOYSTICK_H
#define JOYSTICK_H

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/joystick.h>

struct axis_state {
    short x, y;
};

class joystick{
    private:
        const char *device;
    public:
        const char* getDevice();
        void setDevice(const char*);
```

```
        int read_event(int fd, struct js_event *event);
        size_t get_axis_count(int fd);
        size_t get_button_count(int fd);
        size_t get_axis_state(struct js_event *event, struct axis_state axes[3]);
};

#endif // JOYSTICK_H
```

joystick.cpp

```
#include "joystick.h"

const char* joystick::getDevice()
{
    return this->device;
}

void joystick::setDevice(const char* device)
{
    this->device = device;
}

/**
 * Lee un evento del joystick
 * Devuelve 0 si resulta exitoso, si no devuelve -1
 */
int joystick::read_event(int fd, struct js_event *event)
{
    ssize_t bytes;

    bytes = read(fd, event, sizeof(*event));

    if (bytes == sizeof(*event))
        return 0;

    /* Error, could not read full event. */
    return -1;
}

/**
 * Devuelve el número de ejes del control o 0 si ocurre un error.
 */
size_t joystick::get_axis_count(int fd)
{
    __u8 axes;

    if (ioctl(fd, JSIOCGAXES, &axes) == -1)
        return 0;

    return axes;
}

/**
 * Devuelve el número de botones del control o 0 si ocurre un error.
 */
size_t joystick::get_button_count(int fd)
{
    __u8 buttons;
    if (ioctl(fd, JSIOCGBUTTONS, &buttons) == -1)
        return 0;

    return buttons;
}

/**
 * Seguimiento del estado actual del eje.
 *
 * NOTA: Asume que los ejes se numeran empezando del 0, que el eje X es
 * un número par y que el eje Y es un número impar.
```

```
 *
 * Devuelve el eje que indica el evento.
 */
size_t joystick::get_axis_state(struct js_event *event, struct axis_state axes[3])
{
    size_t axis = event->number / 2;

    if (axis < 3)
    {
        if (event->number % 2 == 0)
            axes[axis].x = event->value;
        else
            axes[axis].y = event->value;
    }

    return axis;
}
```

Documentación del API
https://www.kernel.org/doc/Documentation/input/joystick-api.txt

Explicación del programa

Incluimos los headers que necesita ROS

```
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>
```

Los siguientes headers para mostrar la información en consola y para hacer uso del comando `system()` para limpiar la consola:

```
#include <iostream>
#include <cstdlib>
```

Y el header de la clase que definimos anteriormente

```
#include "joystick.h"
```

Creamos un objeto para controlar los eventos del joystick usando nuestra clase y definimos otras variables necesarias

```
joystick control;
int js;
truct js_event event;
struct axis_state axes[3] = {0};
size_t axis;
```

Abrimos el dispositivo que se haya especificado como argumento, o en su defecto `js0`. Si no se puede abrir ningún dispositivo, devuelve un error.

```
if (argc > 1)
      control.setDevice(argv[1]);
else
      control.setDevice("/dev/input/js0");

js = open(control.getDevice(), O_RDONLY);

if (js == -1)
      perror("No se pudo abrir el joystick");
```

Inicializamos el nodo como siempre, esta vez lo llamaremos teleop_joystick

```
ros::init(argc, argv, "teleop_joystick");
```

Creamos un `NodeHandle` y lo usamos para publicar el topic `cmd_vel` que se encarga de enviar comandos de velocidad al robot y es de tipo `geometry_msgs::Twist`

```
ros::NodeHandle nh;
ros::Publisher pub = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1);
```

Creamos un mensaje de ese tipo que llamaremos `twist` y lo vamos a usar para guardar la información de velocidad angular y lineal

```
geometry_msgs::Twist twist;
```

En la condición del bucle chequearemos que no haya errores en ROS ni en la lectura de eventos del joystick

```
while ((control.read_event(js, &event) == 0) && (ros::ok()))
```

Usamos el siguiente comando para limpiar la consola cada vez que vamos a mostrar información nuevamente

```
system("clear");
```

Cada vez que se presione un botón o mueva un eje del control, se leerá un evento y acorde al botón presionado modificamos los valores de velocidad. En nuestro caso nos importan los botones 5 y 4, que modifican la velocidad lineal para avanzar o retroceder respectivamente, y el eje 0, que modifica la velocidad angular para girar hacia a la izquierda o derecha en torno al eje z del robot.

```
switch (event.type)
{
      case JS_EVENT_BUTTON:
            if((event.number == 5)&&(event.value))
                  twist.linear.x = 0.22;
            else if((event.number == 4)&&(event.value)
                  twist.linear.x = -0.22;
            else  twist.linear.x = 0;
      break;
      case JS_EVENT_AXIS:
            axis = control.get_axis_state(&event, axes);
            if (axis == 0) twist.angular.z = -axes[0].x/10000.0f;
      break;
      default:
      break;
}
```

Seteamos las componentes lineales y angulares que no modificamos en 0

```
twist.linear.y = 0;
twist.linear.z = 0;
```

```
twist.angular.x = 0;
twist.angular.y = 0;
```

Publicamos twist en el topic cmd_vel para que estos sean los nuevos valores de velocidad del robot

```
pub.publish(twist);
```

Hacemos que ROS procese el callback

```
ros::spinOnce();
```

Finalmente, dentro del bucle, mostramos la información con formato

```
std::cout << "Lineal" << std::endl
         << "x: " << twist.linear.x << std::endl
         << "y: " << twist.linear.y << std::endl
         << "z: " << twist.linear.z << std::endl
         << "Angular" << std::endl
         << "x: " << twist.angular.x << std::endl
         << "y: " << twist.angular.y << std::endl
         << "z: " << twist.angular.z << std::endl;
```

Ya fuera del bucle, cerramos el dispositivo antes de finalizar el programa

```
close(js);
```

Código completo

```
#include "ros/ros.h"
#include <geometry_msgs/Twist.h>

#include <iostream>
#include <cstdlib>

#include "joystick.h"

int main(int argc, char **argv)
{
    joystick control;
    int js;
    struct js_event event;
    struct axis_state axes[3] = {0};
    size_t axis;

    if (argc > 1)
        control.setDevice(argv[1]);
    else
        control.setDevice("/dev/input/js0");

    js = open(control.getDevice(), O_RDONLY);

    if (js == -1)
        perror("No se pudo abrir el joystick");

    ros::init(argc, argv, "teleop_joystick");

    ros::NodeHandle nh;
    ros::Publisher pub = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1);

    geometry_msgs::Twist twist;
```

```
    while ((control.read_event(js, &event) == 0) && (ros::ok()))
    {
        system("clear");

        switch (event.type)
        {
            case JS_EVENT_BUTTON:
                if((event.number == 5)&&(event.value)) twist.linear.x = 0.22;
                else if((event.number == 4)&&(event.value)) twist.linear.x = -0.22;
                else twist.linear.x = 0;
                break;
            case JS_EVENT_AXIS:
                axis = control.get_axis_state(&event, axes);
                if (axis == 0) twist.angular.z = -axes[0].x/10000.0f;
                break;
            default:
                break;
        }

        twist.linear.y = 0;
        twist.linear.z = 0;
        twist.angular.x = 0;
        twist.angular.y = 0;

        pub.publish(twist);
        ros::spinOnce();

        std::cout << "Lineal" << std::endl
            << "x: " << twist.linear.x << std::endl
            << "y: " << twist.linear.y << std::endl
            << "z: " << twist.linear.z << std::endl
            << "Angular" << std::endl
            << "x: " << twist.angular.x << std::endl
            << "y: " << twist.angular.y << std::endl
            << "z: " << twist.angular.z << std::endl;
    }

    close(js);
    return 0;
}
```

6.3 - gmapping

6.3.1 - Slam

Todavía no pude hacer funcionar el tema de la transformada, así que por ahora estoy usando un nodo que se ocupa del mapeo en tiempo real y publica los datos en el topic `map`.

```
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

La información que se publica en el topic map es de tipo `nav_msgs::OccupancyGrid`. La definición de este mensaje es:

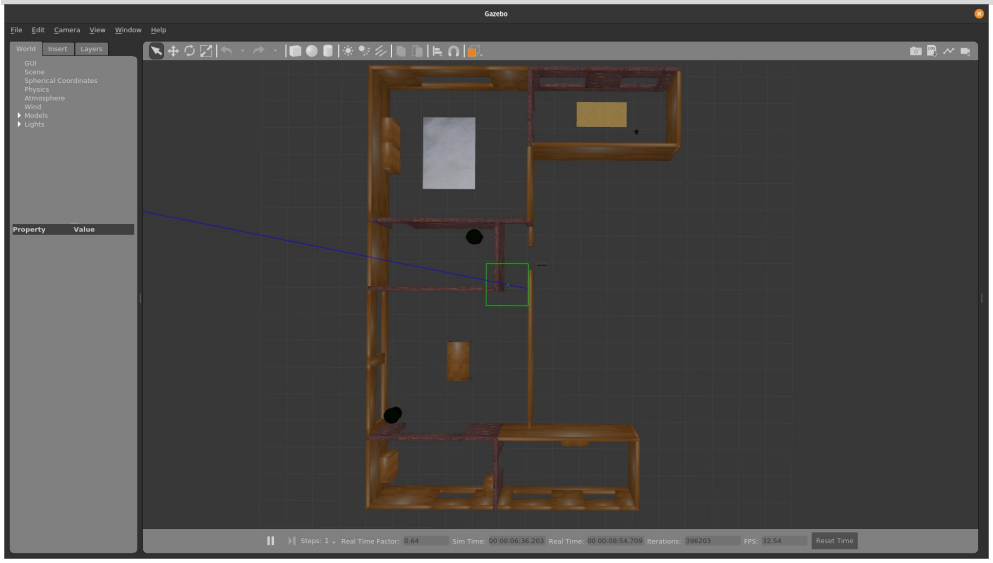
std_msgs/Header header
nav_msgs/MapMetaData info
int8[] data

En el miembro `info` guarda el tiempo en el que se cargó el mapa, la resolución, el ancho, el alto y la posición real del origen.

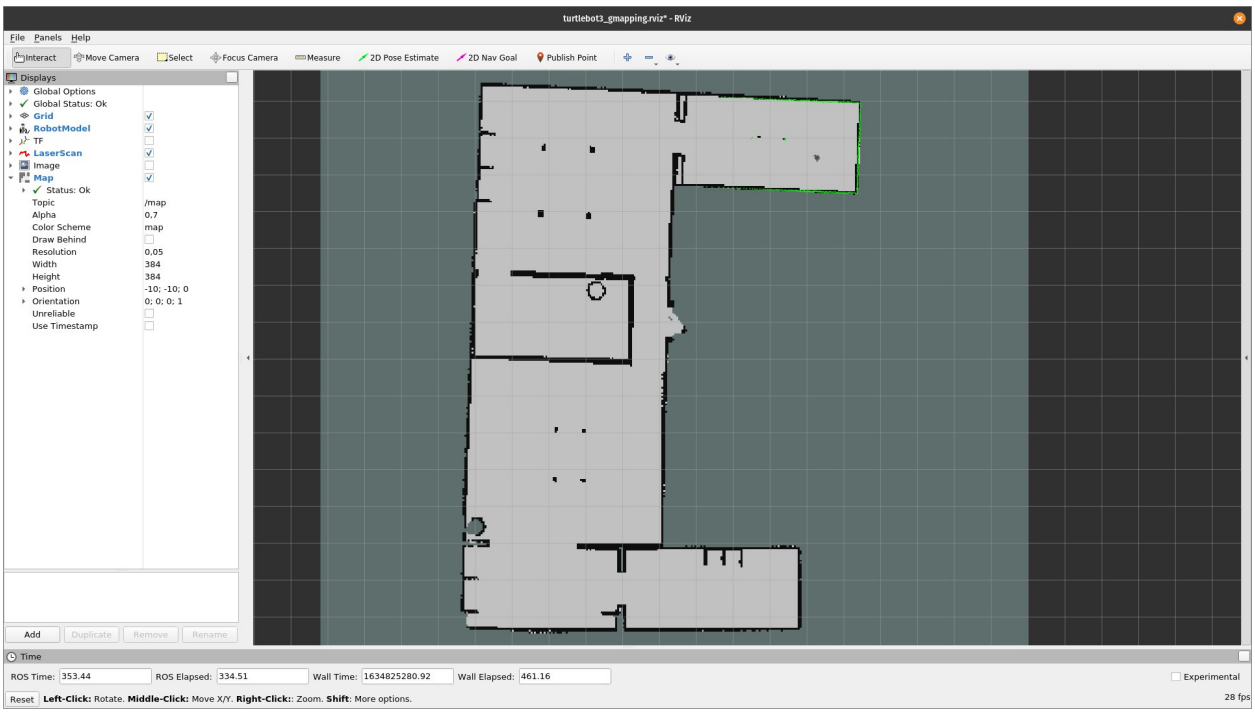
El vector `data` guarda la probabilidad de que una celda esté ocupada en un rango de 0 a 100. Si la ocupación es desconocida guarda -1.

Para generar la información de las celdas uso uno de los mapas de gazebo, la simulación de la casa:

```
roslaunch turtlebot3_gazebo turtlebot3_house.launch
```



Y haciendo uso del nodo `teleop_joystick` creado anteriormente muevo el robot para escanear todo el ambiente. Podemos ver los datos que se publican en el topic `map` en RViz:



Una vez que tengo suficiente información, guardo los datos en un archivo con un nodo que llamé provisoriamente `gmapping_test`. Lo hago de esta manera porque quiero visualizar el mapa en un programa gráfico propio y implementar OpenGL me va a tomar mucho tiempo. Uso un programa hecho en Qt para el control del robot y el algoritmo de pathfinding. Como no pude comunicarlo perfectamente con los nodos, uso estos archivos con el mapa para el calculo del camino.

```
void PathPlanner::MapCallBack(const nav_msgs::OccupancyGrid::ConstPtr
&grid)
{
    int i, j;
    std::ofstream f("mapa", std::ios::binary);
    if(f)
    {
        f.write((char*)&grid->info.width, sizeof(uint32_t));
        f.write((char*)&grid->info.height, sizeof(uint32_t));
        for(i = 0; i < grid->data.size(); i++)
        {
            f.write((char*)&grid->data.at(i), sizeof(int8_t));
        }
        f.close();
        std::cout << 1 << std::endl;
        ros::shutdown();
    }else std::cout << "no se pudo abrir el archivo" << std::endl;
}
```

Por ahora la función callback se ve así, guarda la información del grid en un archivo llamado `mapa`.

6.4 - Pathfinding

Como dije en el punto anterior, temporalmente estoy ejecutando el algoritmo de pathfinding desde un programa gráfico hecho en Qt que toma el archivo donde guardé el mapa.

El algoritmo de pathfinding que implementé es el algoritmo de búsqueda A*. El pseudocódigo que seguí fue:

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path// A* finds a path from start to goal.// h is the
heuristic function. h(n) estimates the cost to reach goal from node
n.function A_Star(start, goal, h)
    // The set of discovered nodes that may need to be (re-)expanded.
    // Initially, only the start node is known.
    // This is usually implemented as a min-heap or priority queue rather
than a hash-set.
    openSet := {start}

    // For node n, cameFrom[n] is the node immediately preceding it on the
cheapest path from start
    // to n currently known.
    cameFrom := an empty map

    // For node n, gScore[n] is the cost of the cheapest path from start to n
currently known.
    gScore := map with default value of Infinity
    gScore[start] := 0

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our
current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a min-heap or
a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)
        for each neighbor of current
            // d(current,neighbor) is the weight of the edge from current to
neighbor
            // tentative_gScore is the distance from start to the neighbor
through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one.
Record it!
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := gScore[neighbor] + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)

    // Open set is empty but goal was never reached
    return failure
```

Mi implementación quedó de la siguiente manera:

```
std::list<Vector2i> getCamino(Vector2i inicio, Vector2i fin, nav_msgs::OccupancyGrid grid)
{
    std::list<Nodo*> abiertos, cerrados, vecinos;   //abiertos: lista de nodos candidatos para evaluación;
cerrados: lista de nodos evaluados; vecinos: lista de nodos vecinos del punto a evaluar
    std::list<Nodo*>::iterator it, it2, it3, mejor;
    std::list<Vector2i> path;
    Nodo *puntoInicial, *puntoFinal, *puntoActual, *puntoAux;
    bool flagCerrado, flagAbierto, pathNuevo;
    int i, j, progreso = 0, progresoAnterior = 0;
    float tempG, distInicial;

    //Aloja la misma cantidad de memoria que ocupa el mapa para los nodos
    Nodo **nodos = new Nodo*[grid.info.width*grid.info.height];
    for(i = 0; i < grid.info.width; i++)
        for(j = 0; j < grid.info.height; j++)
        {
            nodos[i+grid.info.width*j] = new Nodo;
            nodos[i+grid.info.width*j]->x = i;
            nodos[i+grid.info.width*j]->y = j;
        }

    puntoInicial = nodos[inicio.x+grid.info.width*inicio.y];
    puntoFinal = nodos[fin.x+grid.info.width*fin.y];
    puntoActual = puntoInicial;
    distInicial = distPuntos(puntoInicial->x,puntoInicial->y,0.0f,puntoFinal->x,puntoFinal->y,0.0f);

    abiertos.push_back(puntoInicial);
    if(grid.data[puntoFinal->x+grid.info.width*puntoFinal->y] == 0) //Si el punto objetivo no está bloqueado
    {
        while(!abiertos.empty())
        {
            //Elije la mejor opción de los nodos abiertos buscando el que tenga el menor valor de f
            mejor = abiertos.begin();
            for (it = abiertos.begin(); it != abiertos.end(); it++)
                if((*it)->f < (*mejor)->f) mejor = it;
            puntoActual = (*mejor);

            //Si llegó al punto final, encontró el camino y sale del bucle
            if(puntoActual == puntoFinal)
            {
                std::cout << "Camino listorti" << std::endl;
                break;
            }

            //Saca el punto que esta evaluando de la lista de nodos abiertos y la pasa a la de nodos cerrados
            abiertos.remove(puntoActual);
            cerrados.push_back(puntoActual);

            //Mete los nodos vecinos del punto a evaluar en una lista para recorrerlos más adelante
            vecinos.clear();
            for(i = -1; i < 2; i++)
                for(j = -1; j < 2; j++)
                    if(!((i==0)&&(j==0)))
                    {
                        puntoAux = nodos[(puntoActual->x+i)+grid.info.width*(puntoActual->y+j)];
                        vecinos.push_back(puntoAux);
                    }

            //Si los vecinos son candidatos a evaluación y se pueden recorrer, se calculan los factores
            //y si se trata de un camino mejor que el que se venía considerando, se pone como sucesor del punto
actual
            for (it = vecinos.begin(); it != vecinos.end(); it++)
            {
                flagCerrado = false;
                flagAbierto = false;
                for (it2 = cerrados.begin(); it2 != cerrados.end(); it2++)
                    if(*it == *it2) flagCerrado = true;
                for (it3 = abiertos.begin(); it3 != abiertos.end(); it3++)
                    if(*it == *it3) flagAbierto = true;
                if((!flagCerrado) && (grid.data[(*it)->x+grid.info.width*(*it)->y] == 0))
                {
                    tempG = puntoActual->g +
distPuntos((*it)->x,(*it)->y,0.0f,puntoActual->x,puntoActual->y,0.0f);
                    pathNuevo = false;
                    if(flagAbierto)
                    {
                        if(tempG < (*it)->g)
                        {
                            (*it)->g = tempG;
                            pathNuevo = true;
                        }
                    }else{
                        (*it)->g = tempG;
                        pathNuevo = true;
                        abiertos.push_back(*it);
```

```
                    }

                    if(pathNuevo)
                    {
                        (*it)->h = distPuntos((*it)->x,(*it)->y,0.0f,puntoFinal->x,puntoFinal->y,0.0f);
                        (*it)->f = (*it)->g + (*it)->h;
                        (*it)->anterior = puntoActual;
                    }
                }
            }
        }
    }else //Si el objetivo está bloqueado función termina devolviendo el punto final
    {
        path.push_back((Vector2i){puntoFinal->x,puntoFinal->y});
        return path;
    }

    //Reconstruye el camino y lo guarda en una lista de vectores de dos dimensiones
    puntoAux = puntoActual;
    path.push_front((Vector2i){puntoAux->x, puntoAux->y});
    while(puntoAux->anterior != NULL)
    {
        path.push_front((Vector2i){puntoAux->anterior->x, puntoAux->anterior->y});
        puntoAux = puntoAux->anterior;
    }

    //Borra la matriz de punteros a nodos
    for(i = 0; i < grid.info.width*grid.info.height; ++i)
        delete [] nodos[i];

    return path;
}
```

El siguiente paso es adaptar la función a la utilización de mensajes nativos de ROS de tipo `nav_msgs::Path` para dejar de usar las estructura `Vector2i` que sirvió puramente de prueba:

```
typedef struct Vector2i
{
    int x;
    int y
}Vector2i;
```

La estructura que uso para ayudarme a calcular los valores de f, g y h es:

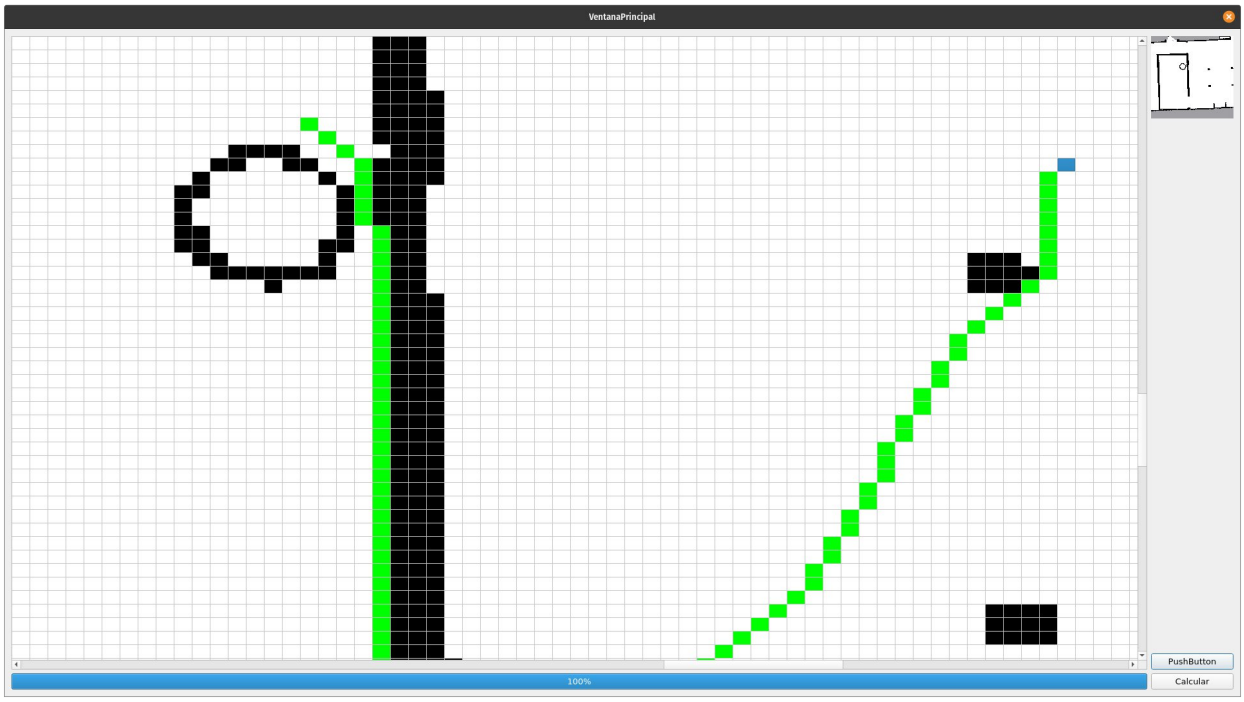
```
typedef struct Nodo
{
    int x;
    int y;
    float f = 0;
    float g = 0;
    float h = 0;
    Nodo *anterior = NULL;
}Nodo;
```

La definición de las funciones `mapear` y `distPuntos` utilizadas es:

```
float mapear(float val, float valMin, float valMax, float outMin, float outMax)
{
    return (val - valMin)*(outMax-outMin)/(valMax-valMin) + outMin;
}
float distPuntos(float x1, float y1, float z1, float x2, float y2, float z2)
{
    return sqrtf((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2) + (z1 - z2)*(z1 -
z2));
}
```

navegador_qt

Usé este programa para calcular el camino y verlo gráficamente con finalidades de testeo.



La grilla permite seleccionar el punto de destino hasta el cual se calculará el camino desde la posición del robot. La barra de carga inferior sirve de indicador del procesamiento, porque en trayectos largos puede tomar varios segundos donde no se observan avances.

El “mini mapa” de la esquina superior derecha ayuda a ubicarse en el mapa, porque de tan cerca puede ser confuso.

Este código tiene unicamente finalidades de prueba y eventualmente pasará a ser un tipo de dato compatible con RViz para visualizarlo.

El programa lee el archivo con los datos de las celdas que fueron escaneadas del mapa, y el algoritmo se encarga de calcular el camino más corto entre dos puntos esquivando los obstáculos.

El primer problema que se hace evidente es que el algoritmo no tiene en cuenta las dimensiones del robot.

En conclusión falta terminar la migración del algoritmo al uso de mensajes nativos de ROS, lo que me va a permitir hacer los cálculos directamente entre nodos en tiempo real, además de posibilitar la visualización del camino en RViz.