Práctica 1.

Carrera de robots

ISMAEL PIÑEIRO RAMOS
48560399-Q
ipr25@alu.ua.es

Introducción

En esta primera aventura en lo que ha robótica se refiere, se ha propuesto el desarrollo de un sistema de navegación para un robot conocido como Turtlebot con algunos añadidos, como cámaras traseras.

Las tareas que han sido llevadas a cabo son:

- Realización de una panorámica en tiempo real
- Guiado del robot por el circuito

Contenido de la práctica

El contenido de la práctica se trata del workspace de ROS con el que se ha ido trabajando. La raíz del mismo es “p1_ws” y a destacar tenemos:

p1_ws --|
src --|
listener → paquete que contiene lo referente a la panorámica
p1_pkg → paquete que contiene lo referente a la navegación

Además de lo nombrado anteriormente, se tienen otros archivos, como los necesarios para los paquetes/espacio de trabajo, otras pruebas realizadas como las realizadas con GMapping y AMCL sin ningún resultado concreto y lo visto en el seminario de prácticas.

Dentro de ambos paquetes se contienen las clases que se verán a continuación junto a un “main” para que puedan ser utilizadas de manera sencilla.

Navegación

La navegación del robot se ha realizado utilizando el Imu y el Laserscan del robot Turtlebot. Principalmente, en la navegación, es importante conocer dos cosas: Dónde estamos y qué es lo que nos rodea. Para ello utilizaremos el Imu y la odometría para poder tener una idea certera de dónde nos encontramos y utilizamos el láser para poder detectar los obstáculos que se antepongan al robot.

Con la información obtenida de donde estamos y que nos rodea podemos proceder a publicar mensajes de velocidad del tipo geometry_msgs::Twist para mover el robot.

Implementación

Para poder realizar la navegación se ha implementado la clase RobotDriver, la cual se describe a continuación:

Método	**Descripción**
Públicos	
RobotDriver(*ros::NodeHandle&*)	**Constructor.** El constructor de la clase se encarga de inicializar los suscriptores del nodo a los diferentes elementos con los que vamos a comunicarnos. Por lo tanto se suscribe en el mismo a los tópicos: odom, imu_data y scan. Además se inicializa un publicador para poder enviar los mensajes de velocidad.
void **Init**()	Este método tiene el bucle principal, el cual debe ser llamado después del constructor para que el nodo empieza a funcionar y el robot a moverse.
void **IMUCallback**(*const sensor_msgs::Imu::ConstPtr&*)	Callback para los datos del IMU, únicamente obtiene el mensaje del Imu y lo almacena para su tratamiento posterior.
void **LaserCallback**(*const sensor_msgs::LaserScan::ConstPtr&*)	Callback para los datos del Láser, únicamente obtiene el mensaje procedente del láser y lo almacena.
void **OdomCallback**(*const nav_msgs::Odometery::ConstPtr&*)	Callback para los datos de la odometría, únicamente almacenará el mensaje recibido para que sea tratado posteriormente.
Privados	
void **Navigate**()	Este método que será llamado constantemente con los datos obtenidos de los diferentes callbacks es el que realiza los cálculos geométricos con los datos que disponemos para darle vida al robot.

Atributo	**Tipo**	**Descripción**
Privados		
nh_	ros::NodeHandle	Interfaz manejadora del nodo que nos permitirá crear los suscriptores/publicadores.
cmd_vel_pub_	ros::Publisher	Publicador que nos permitirá mandar los comandos de velocidad al tópico velocity.
laser_sub_	ros::Subscriber	Suscriptor para poder obtener los mensajes procedentes del láser
imu_sub_	ros::Subscriber	Suscriptor para obtener los mensajes del Imu.

odom_sub_	ros::Subscriber	Suscriptor para obtener los mensajes de la odometría.
twist_msg_	geometry_msgs::Twist	Mensajes que será enviado mediante el publicador que contiene los comandos de velocidad a ejecutar, lo cual nos permite mover el robot.
odom_msg_	nav_msgs::Odometry	Mensajes que contendrá los datos recibidos a través de la odometría.
imu_msg_rcvd_	sensor_msgs::Imu	Mensaje con los datos del IMU para ser tratados posteriormente.
laser_msg_rcvd_	sensor_msgs::LaserScan	Mensaje que contendrá los datos del láser para su posterior tratamiento.

Además se dispone en el fichero robotdriver.cpp de las siguientes macros del preprocesador:

- DEBUG (default: 0) → Nos permite mostrar una serie de mensajes útiles para DEBUG cuando su valor sea 1.
- VELOCITY (default: 0.5) → Velocidad máxima del robot, por defecto 0.5m/s que es lo máximo permitido.
- TURBO_VELOCITY (default: 0.9) → Velocidad cuando se active el turbo al visualizar un robot solo durante 5 segundos.
- DISTANCE_OBS_AVOID (default: 2) → Distancia a partir de la cual se tendrán en cuenta los objetos para esquivarlos.

Algoritmo

El algoritmo implementado consiste principalmente en la utilización del láser para detectar obstáculos y movernos observando nuestro alrededor. Las principales fases que componen el algoritmo son:

1. Obtener roll, pitch y yaw con los datos de donde nos encontramos.
2. Para cada rango detectado por el láser los cuáles están en una posición menor que un umbral (es decir, están cercanos al robot) comprobar en qué cuadrante se encuentran izquierdo o derecho para realizar la rotación adecuada.
3. Si el láser encuentra algo al frente con un umbral menor se reducirá la velocidad actual a la mitad para facilitar el giro.
4. Con los cálculos de la rotación y velocidad mandamos los mensajes adecuados a través del publicador de los comandos de velocidad.

Más detalladamente el pseudocódigo del algoritmo de navegación sería algo como así:

```
girar_izquierda = false, girar_derecha = false;
obtener roll, pitch y yaw de la posición actual;
laser_theta = laser_msg.angle_min;
para cada rango en laser_msg
  si rango < UMBRAL_DISTANCIA // quiere decir que algo esta cerca
    si laser_theta >= -PI/2.0 Y laser_theta <= 0.0 // detectado en cuadrante derecho
      si girar_izquierda == false
        rotación += 1.0 // rotamos a la izquierda
        girar_izquierda = true;
      finsi
    sino si laser_theta >= 0.0 Y laser_theta <= PI/2.0 // detectado en cuadrante
izq.
      si girar_derecha == false
        rotacion += -1.0; // rotamos a la derecha
        girar_derecha = true;
      finsi
    finsi
    si laser_theta <= -PI/2.0 Y laser_theta < PI/2.0 Y rango < 1.5 // en frente
      velocity = velocity/2
    finsi
  finsi
  laser_theta += laser_msg.angle_increment
finpara
twist_msg.linear.x = velocity
twist_msg.angular.z = rotation - yaw;
velocity_pub.publish(twist_msg)
```

Problemas encontrados

El principal problema que me he encontrado con esta parte es la falta de documentación para alguien que es novato en el tema de robótica por parte de ROS. Creo que la documentación de ROS es demasiado básica y no he conseguido encontrar gran cosa para la correcta realización de la práctica.

Si bien es cierto que ROS tiene la que se llama “Navigation stack” para realizar tareas de navegación, la documentación que encontré respecto a esto únicamente cubre el modo de navegación respecto a un previo reconocimiento del mapa (que generalmente se realiza con teclado). Como este no era el objetivo de la práctica, tras probarlo, lo tuve que descarta.

Otro problema es que mis conocimientos de geometría no son muy elevados y tareas básicas se convierten en una tarea más compleja ya que he tenido que buscar información sobre temas de geometría para ponerme al día.

Respecto al algoritmo implementado, si bien es cierto que no funciona correctamente, ya que dependiendo la ejecución y dependiendo la situación de partida consigue avanzar más o menos.

Panorámica trasera

Se ha realizado una vista panorámica trasera a través de las dos cámaras extras añadidas al robot turtlebot. Esta tarea ha sido llevada a cabo con OpenCV de dos maneras diferentes. La primera de ellas ha sido utilizando OpenCV para detectar las diferentes características de las imágenes en tiempo real, obtener los descriptores de estas características para posteriormente aplicar RANSAC y así obtener las similitudes entre ambas imágenes.

Por otro lado, se ha utilizado la clase “Stitcher” proporcionada por OpenCV para realizar una vista panorámica de una manera menos rudimentaria, ya que nos abstrae bastante del proceso de creación de la vista panorámica.

Implementación

Para esto se ha construido la clase Panorama, a continuación se describe la clase de la misma:

Método	**Descripción**
Públicos	
Panorama()	**Constructor.** Se encarga de inicializar el ImageTransport y con ello los suscriptores de las dos cámaras traseras del robot. Además, registrará el Callback para recoger los datos de las cámaras de manera sincronizada mediante la clase Synchronizer. Por último creará un thread para abrir una ventana donde se visualizará el resultado.
void **Init**()	**Inicializador del nodo.** Lanza el nodo y aplicará el método StitchingImage() que se describe posteriormente, siempre que se hayan recogido dos imágenes como información en el Callback.
void **ImageCallback**(*const sensor_msgs::ImageConstPtr&, const sensor_msgs::ImageConstPtr&*)	Método que recibe los mensajes de los suscriptores de manera sincronizada y almacena la imagen una matriz de la clase para su posterior tratamiento.
Privados	
void **StitchingImage**()	Este el método que implementa el algoritmo para realizar la panorámica a partir de la información almacenada en las matrices de las dos imágenes. Existe una macro del preprocesador llamada “USE_STITCHER_CLASS” que cuando se encuentra a 1 utilizará el algoritmo que proporciona OpenCV el cual nos abstrae bastante del proceso. (Por defecto USE_STITCHER_CLASS = 0, por lo que se utiliza el algoritmo que se ha implementado y se detalla posteriormente.

Atributo	Tipo	Descripción
Privados		
m_nh	ros::NodeHandle	Interfaz que nos permitirá crear los suscriptores.
m_it	image_transport:: ImageTransport	Nos abstrae de la comunicación de imágenes en los suscriptores de la red.
m_leftCamSub	image_transport:: SubscriberFilter	Suscriptor para la cámara izquierda que nos permitirá a su vez sincronizar los callbacks.
m_rightCamSub	image_transport:: SubscriberFilter	Suscriptor para la cámara derecha que nos permitirá a su vez sincronizar los callbacks.
m_sync	message_filters:: Synchronizer<>	Nos ayudará a registrar el callback para que ambos suscriptores actúen de manera sincronizada.
m_leftImgMat	cv::Mat	Imagen de la cámara izquierda almacenada como matriz
m_rightImgMat	cv::Mat	Imagen de la cámara derecha almacenada como matriz

Esta clase dispone de dos macros del preprocesador para variar su funcionamiento, las cuales son:

- USE_STITCHER_CLASS → Determina si se utilizará la clase Stitcher proporcionada por OpenCV, generalmente se obtienen mejores resultados.
- SHOW_CAPTURE_IMAGES → Determina si se mostrarán dos ventanas adicionales con las imágenes capturadas por las cámaras, útil para debug.

Algoritmo

El algoritmo que se ha implementado consta principalmente de seis fases que se presentan a continuación:

1. Obtener las dos imágenes y convertirlas a escala de grises.
2. Calcular los keypoints de ambas imágenes.
3. Encontrar los descriptores de ambas imágenes en dos arrays utilizando el algoritmo SURF.
4. Emparejar los descriptores encontrados entre las dos imágenes obteniendo sus correspondencias.
5. Obtener la matriz de homografía utilizando RANSAC sobre los descriptores emparejados anteriormente.
6. Aplicar trigonometría y las deformaciones necesarias para unir las dos imágenes basándonos en la matriz de homografía.

Más detalladamente el algoritmo actúa de la siguiente manera.

En primer lugar se obtienen las dos imágenes convertidas en escala de grises para una mejor obtención de los keypoints, a continuación se comenzará a calcular los keypoints de

ambas imágenes mediante el algoritmo de Surf (cv::SurfFeatureDetector) con un umbral de 300 y los almacenamos en dos vectores de tipo cv::KeyPoint.

Una vez hemos obtenido los keypoints de ambas imágenes en escala de grises, obtenemos los descriptores de las mismas, basándonos en estos keypoints con el algoritmo de Surf (cv::SurfDescriptorExtractor) y guardandolos en dos matrices.

Cuando ya se han extraído los descriptores de las dos imágenes se intentan emparejar entre ellos de manera que encuentra similitudes entre ambas imágenes y almacenamos los emparejamientos obtenidos en un vector de tipo cv::DMatch para posteriormente obtener la distancia entre todos estos descriptores y solo quedarnos con los mejores.

Para obtener los mejores descriptores es tan sencillo como obtener la distancia mínima y máxima de todos los descriptores obtenidos y posteriormente solo quedarnos con los que tengan una distancia menor de un umbral, el cual es un múltiplo de la distancia mínima, (4 * distancia mínima, por defecto).

Con los buenos emparejamientos obtenidos se obtiene la matriz de homografía utilizando los keypoints de estos emparejamientos y utilizando CV_RANSAC con la ayuda del método “findHomography(...)”. Por último se deforma la imagen izquierda añadiendo el tamaño de la imágen derecha mediante warpPerspective(..) y se unen ambas imágenes en una nueva matriz resultado que será la imagen que finalmente aparezca por pantalla.

cv::Stitcher

La clase de Stitcher de OpenCV proporciona una clase con la que realizar vistas panorámicas y unir dos imágenes. Para ello, en primer lugar, se debe de crear un stitcher mediante cv::Stitcher::createDefault(), podemos pasarle como argumento un booleano que indique si se utilizará la GPU para la realización de cálculos o no.

Para poder continuar y realizar el stitcher se debe de crear una matriz donde se almacenará el resultado, y un vector con las imágenes de entrada. Una vez realizado esto, basta con utilizar el método “stitch” de la clase stitcher que recibe como argumento las imágenes de entrada y la matriz de salida. Este método devolverá cv::Stitcher::Status y si todo ha ido bien devolverá OK. Es posible que con dos imágenes falle y devuelva un error -1 de imágenes insuficientes, para ello basta con ajustar un parámetro de la clase Stitcher, más concretamente sticher_obj.setPanoConfidenceThresh(0)

Resultados obtenidos

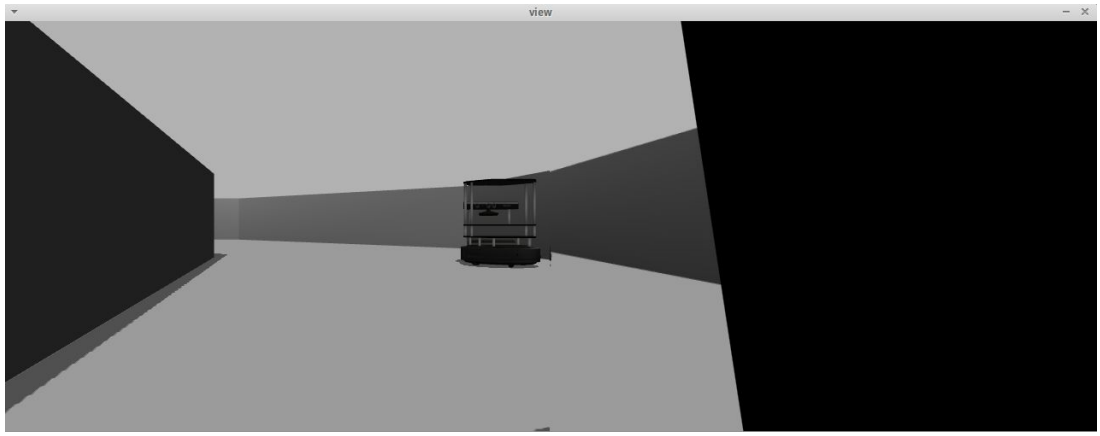

Imagen 1. Resultado obtenido con algoritmo desarrollado

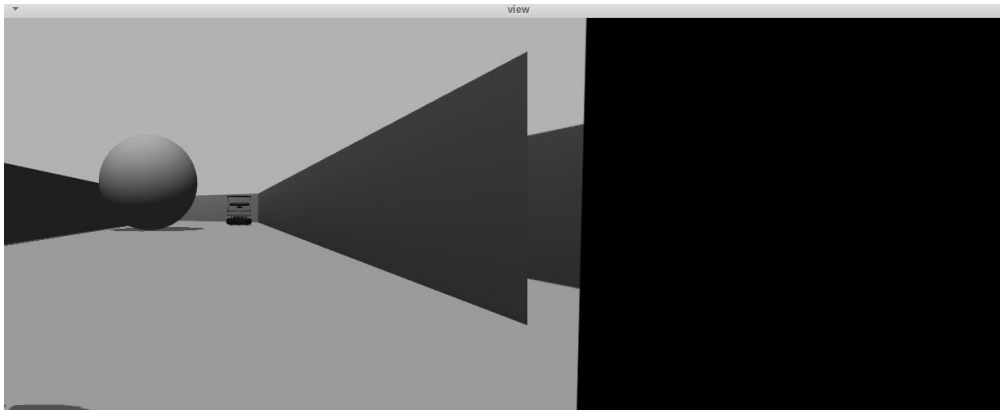

Imagen 2. Resultado obtenido con algoritmo desarrollado

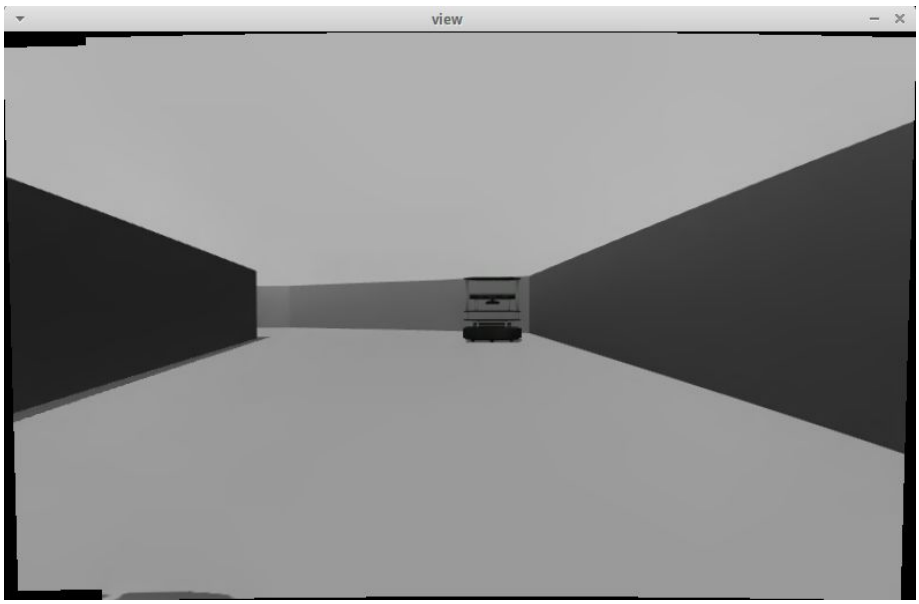

Imagen 3. Resultado obtenido con Stitcher de OpenCV

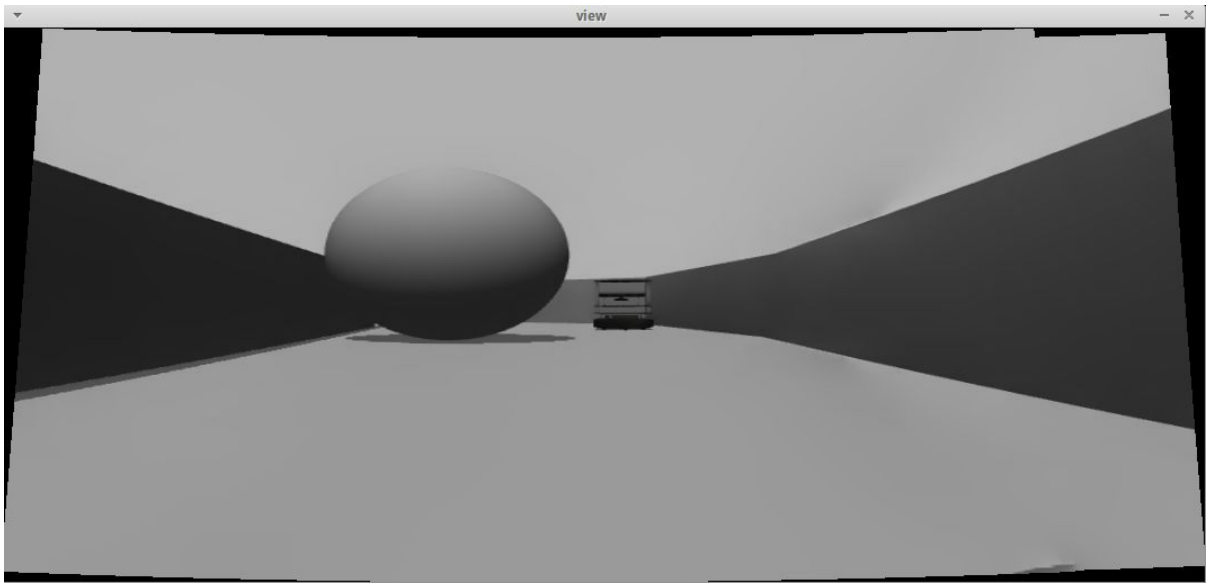

Imagen 4. Resultado obtenido con Stitcher de OpenCV

Problemas encontrados

Aunque esta parte resulta algo más sencilla que la navegación, debido a que existe bastante documentación de OpenCV, resulta complicado comprender al máximo los algoritmos que se utilizan y requieren de estudio individual para sacarle el máximo partido.

Por otro lado, en cuanto al desarrollo del algoritmo propio, se observa en los resultados obtenidos que no siempre obtenemos un resultado que parezca correcto y mucho menos un resultado perfecto. En primer lugar se tuvo la problemática que los suscriptores no estaban sincronizados, para ello se utilizó los métodos que proporciona ROS para sincronizar suscriptores e image_transport, ya que las imágenes no llegaban al mismo tiempo y era prácticamente imposible construir una panorámica.

Una vez solucionado los problemas con la sincronización, seguía sin funcionar correctamente y no he conseguido hacerlo funcionar bien. No obstante, investigue y encontre una clase de OpenCV que realiza una vista panorámica y también la utilice, esta tampoco siempre consigue buenos resultados.

El principal problema es que las imágenes que obtenemos son muy parecidas, apenas tienen puntos que destaquen y puedan ser obtenidos como características de cada imágen. O lo que es lo mismo, no se están consiguiendo los suficientes keypoints.

Conclusión

Si bien la práctica ha sido interesante, pero también ha sido en ocasiones muy frustrante, sobre todo la parte de navegación. También los problemas derivados para poder hacer funcionar correctamente Gazebo en una máquina virtual, lo cual no pude realizarlo porque al parecer el hardware del que dispongo no es compatible con la virtualización de la aceleración 3D por lo que lo hacía imposible de usar Gazebo. Pero todo se soluciono cuando conseguí un disco duro extraible e instale la versión recomendada de ubuntu y ROS.

Por lo visto, ROS es ampliamente utilizado en el mundo de la robótica teniendo soporte para diversos robots de manera oficial. Desde mi punto de vista es complicado comenzar en el mundo de la robótica con ROS debido a la totalidad de componentes que lo forman, pero tiene bastante potencial.

El sistema formado por nodos, tópicos y mensajes al principio resulta algo inquietante, pero enseguida se comprende y es muy fácil de utilizar.

Bibliografía

- http://ramsrigoutham.com/2012/11/22/panorama-image-stitching-in-opencv/
  [última rev. 28/03/2016]
- http://answers.ros.org/question/9705/synchronizer-and-image_transportsubscriber/
  [última rev. 28/03/2016]
- http://study.marearts.com/2013/11/opencv-stitching-example-stitcher-class.html
  [última rev. 28/03/2016]
- https://github.com/antimodular/ofxCv_stitch/tree/master/using%20loaded%20images
  [última rev. 28/03/2016]
- http://wiki.ros.org/message_filters/ApproximateTime
  [última rev. 29/03/2016]
- http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom
  [última rev 1/04/2016]
- http://docs.ros.org/hydro/api/tf/html/c++/classtf_1_1Quaternion.html
  [última rev. 1/04/2016]
- http://web.engr.oregonstate.edu/~kraftko/code/me456_hw2/lab2_JJC.py
  [última rev. 4/04/2016]
- http://answers.ros.org/question/50113/transform-quaternion/
  [última rev. 4/04/2016]