5.3. Manual de usuario de la odometría.

El nodo dedicado al cálculo de la odometría visual, si bien es prácticamente automático una vez puesto en marcha, también requiere de cierta explicación previa a su utilización. Su responsabilidad principal consiste en recibir pares de nubes de puntos que, como precondición, se consideran consecutivas al inicio y final de un movimiento; y determinar el movimiento para publicarlo inmediatamente en términos de desplazamiento y cambio de orientación. Intervienen en este caso los topics de lectura de nubes de puntos y un grupo de servicios de control adicionales. Toda la configuración se realiza mediante parámetros básicos estáticos de ROS que son leídos al ejecutar el nodo. Además de la lista de requisitos, se enumeran a continuación los pasos de instalación y los que corresponden al manejo completo de la aplicación en el caso más complejo.

5.3.1. Requisitos necesarios

- Requisitos hardware (equipo de las pruebas)
  - Procesador Intel Pentium T4500 (2.30 GHz) o superior
  - 2 GiB de Memoria RAM
  - 20 GiB de espacio en el disco duro
    (incluyendo un pequeño margen para actualizaciones y ficheros temporales)
  - Teclado para el control del entorno
  - Microsoft Kinect
  - Puerto de conexión USB 2.0 para el Kinect
- Requisitos software
  - Ubuntu Linux 10.04 (Lucid Lynx)
  - Robot Operating System (ROS) versión *Fuerte*
  - Paquetes de Linux:
    - python2.6
    - python2.6-dev
  - Paquetes (de Linux) específicos para ROS:
    - ros-fuerte-openni-kinect (controlador openni para el Kinect)
    - ros-fuerte-perception-pcl (librería de nubes de puntos PCL)

El paquete *python2.6-dev* es utilizado por ROS cuando se modifica el fichero de configuración del servidor dinámico de parámetros, que se introducirá más adelante. Pueden existir requisitos "implícitos" que no son especificados en la documentación de ROS porque se instalan como dependencias de los nombrados.

5.3.2. Instalación del entorno

Antes de poder utilizar el nodo de odometría visual es necesario instalar el entorno ROS que ofrece los canales de comunicación para el mismo, el propio nodo y también los distintos paquetes adicionales requeridos para el código por el código para los cálculos y las comunicaciones. No será necesario repetir los pasos que se hayan realizado ya durante la instalación de la interfaz en caso de haberse realizado primero.

Se utiliza Ubuntu Linux 10.04 para mostrar la instalación básica y como versión de ROS se utilizará "ROS fuerte" debido a ciertas incompatibilidades que ofrece la versión anterior de ICP con algunos parámetros que antes no eran modificables. En cualquier caso la versión de Python es la 2.6 por ser, también, la más antigua con la que se puede garantizar el funcionamiento correcto.

1. El primer paso será la instalación del propio entorno ROS siguiendo las instrucciones de la página oficial[5], pero que se enumeran a continuación como parte imprescindible del presente documento.

   a) Si el sistema operativo no está configurado para permitir repositorios "restricted", "universe" y "multiverse", estos pueden activarse desde la ventana de "Software Resources" o "Recursos Software", en el menú de "Sistema / Administración" que se muestra en la Figura 5.5.
   La explicación original y actualizada, en caso de que hubiera cambios en las futuras versiones, puede encontrarse en la siguiente dirección:
   " `https://help.ubuntu.com/community/Repositories/Ubuntu` "

   b) El segundo paso es incluir las direcciones de los repositorios. A continuación se muestra el comando exacto para las versiones especificadas anteriormente que son utilizadas en este manual.

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu lucid main" > /etc/apt/sources.
       list.d/ros-latest.list'
   ```

   c) Es necesario configurar las claves para el acceso al repositorio de la siguiente manera:

   ```
   wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
   ```

   d) Con los repositorios correctamente preparados para la instalación se utiliza la herramienta "apt-get" según los pasos habituales con el entorno ROS como objetivo.

   ```
   sudo apt-get update
   sudo apt-get install ros-fuerte-desktop
   ```

   Sólo para la parte de odometría: la versión "-full" será utilizada debido a que se requieren los paquetes de navegación y de percepción que no se incluyen en la versión "desktop" básica.

   ```
   # Si se actualizo recientemente con "sudo apt-get update"
   sudo apt-get install ros-fuerte-desktop-full
   ```

   e) Una vez acabada la instalación, es necesario incluir el fichero "setup.bash" de ROS en el fichero ".bashrc" para que el primero sea ejecutado al inicio definiendo así las variables de entorno necesarias para el correcto funcionamiento de ROS.

   ```
   echo "source /opt/ros/fuerte/setup.bash" >> \~/.bashrc
   . \~/.bashrc
   ```

[5] Instrucciones de instalación de ROS en su web: " `www.ros.org/wiki/fuerte/Installation/Ubuntu` "

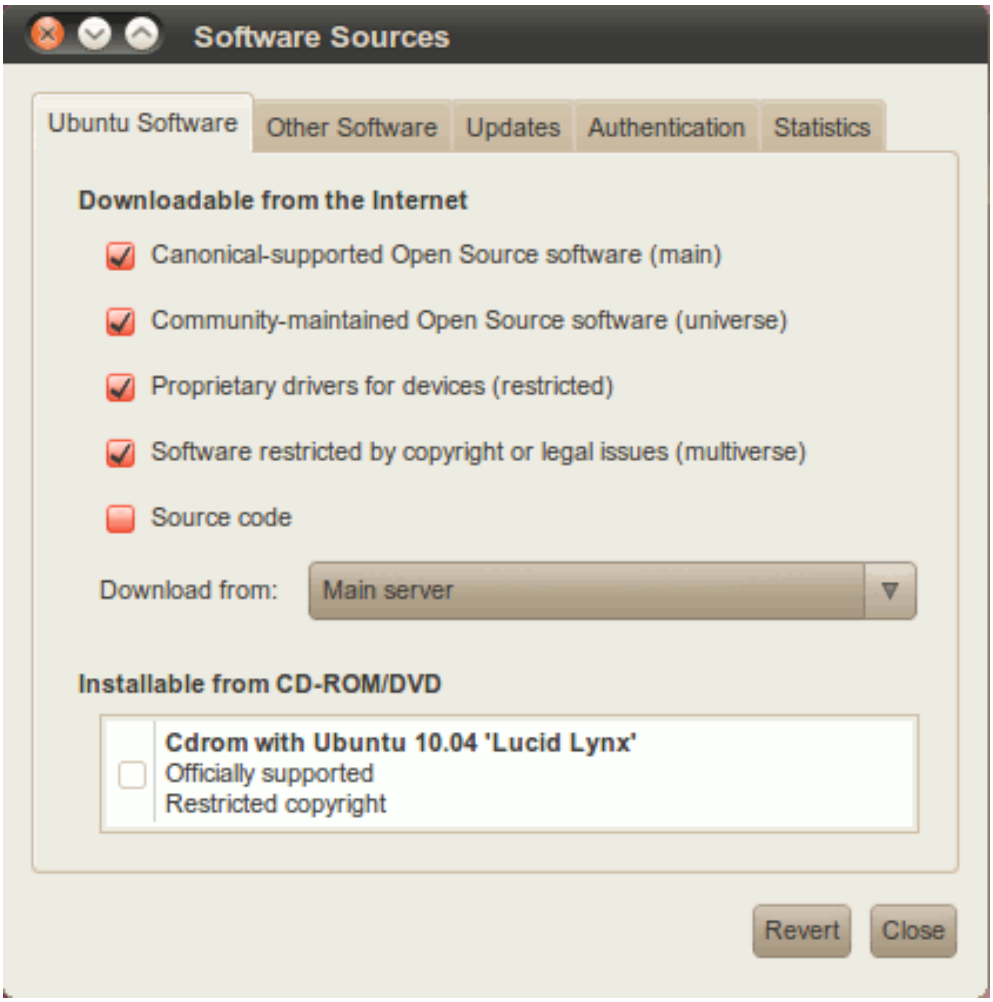

Figura 5.5: Ubuntu - Software Sources Panel

(En su lugar se puede ejecutar directamente « `source /opt/ros/fuerte/setup.bash` » para que los cambios no sean permanentes y desaparezcan al cargar de nuevo el entorno Linux.

Hecho esto, el entorno ROS, que es la base sobre la que se trabaja, deberá haber quedado completamente instalado y funcional.

2. En caso de no encontrarse instalado Python 2.6 o superior (y del mismo modo en caso de duda), es necesario obtener los paquetes adecuados que son necesarios para el funcionamiento del paquete, en este caso a través de apt-get:

```
# Si se actualizo recientemente con "sudo apt-get update" basta escribir:
sudo apt-get install python2.6 python2.6-dev python-yaml
```

3. El driver del periférico de entrada deberá estar instalado de forma independiente de la odometría. Si no lo estuviera, a continuación se sugiere una posible instalación para el caso particular del Kinect con los drivers de *openni_kinect*. Eso se traduce en lo siguiente al utilizar apt-get para la instalación:

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install ros-fuerte-openni-kinect ros-fuerte-camera-drivers
```

4. Si, a pesar de la nueva tendencia a incluirlos, los paquetes relacionados con PCL no son instalados junto con la instalación inicial de ROS (o simplemente se ha realizado la instalación mínima), será necesario instalar concretamente los paquetes: *ros-fuerte-perception-pcl* y *ros-fuerte-pcl*.

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install ros-fuerte-perception-pcl ros-fuerte-pcl
```

5. Será necesario para el siguiente paso instalar un cliente git que está disponible mediante apt-get y que permitirá descargar y actualizar de forma fácil el código fuente de los paquetes.

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install git
```

6. Tras completar la instalación de todos los paquetes de dependencias gracias a los pasos anteriores, llega el paso central del proceso que es instalar el propio software de odometría visual desde su repositorio[6]. Se instalarán los archivos en una carpeta nueva de proyectos en vez de utilizar las carpetas ROS existentes, de manera que se muestre así un ejemplo más completo.

```
cd /home/usr_name # Nombre de usuario "usr_name" como ejemplo
sudo git clone https://github.com/michi05/my_odometry
cd my_odometry
rosmake
```

5.3.3. Puesta en marcha y detención de la odometría

Siguiendo los mismos principios que en el desarrollo de la interfaz, se trató de realizar un componente muy simple que, sobre todo, realizara funciones básicas y sirviera como base funcional para otros desarrollos, en vez de desarrollar un software excesivamente completo y complicado. Por ese motivo, y tal como ocurre con la interfaz, toda la configuración se realiza antes del arranque (se explicará en la subsección de configuración) mediante parámetros ROS y/o ficheros de lanzamiento. Sin embargo, en este caso, no es imprescindible ningún fichero de lanzamiento que invoque nodos complementarios ni que realice la configuración, sino que se puede lanzar por sí solo tras cargar la configuración adecuada.

Dicho lo anterior sólo queda especificar que la combinación de teclas para la terminación es *Ctrl+C* y el comando de arranque « `rosrun my_odometry pcl_to_tf` » siempre y cuando se haya lanzado ya el núcleo de ros con « `roscore &` » que al utilizar *roslaunch* se ejecuta automáticamente.

5.3.4. Configuración de al odometría

Tal como se adelantaba, es necesaria una configuración adecuada para el funcionamiento correcto de la odometría, pero esta debe ser introducida de antemano en el servidor de parámetros. Para ello existen tres métodos igual de válidos y que son ofrecidos por el propio entorno ROS, a saber:

[6]El código se guarda bajo la siguiente dirección: “ `https://github.com/michi05/my_odometry` ”

El uso de ficheros de lanzamiento. Tal como se explica en la sección correspondiente (5.4), es posible utilizar ficheros de lanzamiento. En este caso bastaría con un fichero mínimo que lance un solo nodo y sobre el cual se pueda añadir la configuración a gusto del usuario:

```
<launch>
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" >
    <!-- Aqui los distinos parametros y sus valores -->
  </node>
</launch>
```

Cada parámetro de configuración se establecería con una nueva línea « `<param name="nombre_parametro"value="nuevo_valor_cadena"/>` » o, si se desea, con argumentos que pueden ser especificados durante la llamada:

```
<arg name="nombre_argumento" default="nuevo_valor_cadena" />
<arg name="velocidad_arg" default="5" />
<launch>
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" >

  <param name="nombre_parametro" value="$(arg nombre_argumento)" />
  <param name="velocidad" value="$(arg velocidad)" />
  <param name="modo" value="auto" />

  </node>
</launch>
```

La etiqueta “default” es opcional. Si se elimina, el argumento pasa a ser obligatorio para poder realizar el lanzamiento. Dicho argumento se especifica en la llamada inicial usando el operador “dos puntos-igual” (:=) de la siguiente manera: « *roslaunch* `pckg file velocidad:= 3 nombre_argumento :=valor_argumento` ».

La introducción manual de cada valor mediante el comando rosparam. Es un método más simple pero menos práctico dado que introduce los valores directamente en el servidor de parámetros que va a ser borrado al reiniciar el sistema. Consiste en introducir manual e individualmente cada parámetro y su valor con el comando « *rosparam* `set /ruta/nombre_nodo/nombre_parametro valor` », en el que la ruta puede variar, pero acabando con el nombre del nodo debido a que los parámetros se encuentran en la zona privada del mismo.

Cargar los valores desde un fichero de parámetros guardado en el disco. Se trata de una opción intermedia que ofrece el propio entorno ROS mediante el comando « *rosparam* `load` ». Para ello es necesario tener un archivo con los parámetros; ya sea escrito por el usuario (en forma de diccionario con lenguaje YAML) o bien generado mediante « *rosparam* `dump nombre_fichero_params` ». Una vez creado, bastará con ejecutar « *rosparam* `load nombre_fichero_params` » cada vez que se cargue el núcleo de ROS (*roscore*) y antes de ejecutar el nodo que hará uso de los parámetros.

5.3.5. Utilización de las funciones básicas.

A partir de este punto se procederá a ilustrar las acciones que debe efectuar el lector para hacer uso de las distintas funcionalidades ofrecidas por el nodo. Se comienza por la configuración previa al arranque, que define principalmente el modo de funcionamiento (automático o manual) y se seguirá con las acciones básicas que se pueden realizar.

Configurar comportamiento

El nodo de odometría visual sigue una estructura fija específica en cuanto a su configuración y funcionamiento de tal manera que su comportamiento puede adaptarse levemente mediante parámetros. Los dos parámetros más esenciales son son “odometry_manual_mode” y “odometry_ignore_time”.

El primero es un valor booleano para indicar que las nubes de puntos serán recibidas sólo cuando se soliciten; mientras que, en caso contrario, se leerán y procesarán todas las que sea posible adquiriendo una nueva tras cada estimación. El otro parámetro, “odometry_ignore_time” necesita ser activado en el caso de que se utilicen nubes de puntos grabadas con anterioridad o una fuente cuyas marcas de tiempo no sean fiables o coherentes; de lo contrario, los mensajes pueden ser desestimados al ser considerados “viejos”.

Relación de topics a disposición del usuario

A modo de canal de comunicación continua, en el nodo de odometría se utiliza un pequeño grupo de topics para ciertas comunicaciones esenciales como son enviar los resultados de la odometría y recibir las nubes de puntos. Dichos topics pueden cambiar de nombre o reubicarse en nuevos espacios de nombres pero para tener una idea básica de su funcionamiento, en la tabla 5.1 se expone un listado con las direcciones por defecto de los topics; los parámetros configurables que permiten modificarlas y su utilidad.

Procesar nubes de puntos

Una vez establecidos los parámetros según los puntos anteriores, existen dos formas de empezar a generar nuevas medidas:

- **Solicitando cada nueva medida desde el modo manual.**
  Mediante el uso del servicio “updateOdometry”, se obtendrá con cada llamada del usuario, el cálculo del movimiento que define la relación entre la siguiente nube que el nodo se capaz de obtener, y la última conocida. En el caso de la primera llamada se espera una captura y se devuelve un mensaje indicando la disposición para la siguiente llamada.

  El servicio responde con un solo mensaje que incluye el último movimiento realizado, los movimientos acumulados (que denotan la posición) y un código que define el estado actual del algoritmo, todo ello repartido en los campos correspondientes de una estructura de datos de ROS. A continuación un ejemplo de llamada sin movimiento:

Parámetro	Topic	Utilidad
topic_input_str	inputStr	Recibe la dirección de un topic entendido como fuente de nubes de puntos. Adquiere una nube de puntos y olvida de inmediato la dirección recibida.
topic_input_pcl	inputPCL	Si está a la espera, recibe mensajes en forma de nubes de puntos que introduce inmediatamente al algoritmo.
topic_camera_tf	input_camera_tf	Permite enviar al nodo la posición y orientación de la cámara con respecto al robot. Los datos deben llegar en forma de TF "tf/tfmessage" de ROS.
global_odometry_ out_topic	globalOdometry	Topic de salida en el que se publican la posición y orientación acumuladas del robot.
relative_odometry_ out_topic	lastRelativeOdometry	Topic de salida en el que se publica la TF del último movimiento del robot.
topic_aligned_cloud	aligned_cloud	Publicación de la nube de puntos alineada. Si la estimación es correcta, la nube alineada quedará superpuesta a la tomada desde la posición final.
topic_initial_cloud	initial_cloud	Publicación de la nube de puntos obtenida desde la posición inicial.
topic_filtered_cloud	final_filtered_cloud	Publicación de la nube de puntos obtenida desde la posición final.
topic_odometry_ answer	odometry_answer	Publicación del mensaje de odometría con la estimación definitiva del último movimiento, de la posición y orientación actuales y el estado actual del algoritmo.

Tabla 5.1: Listado de topics de comunicación con el nodo de odometría.

```
$ rosservice call /pcl_to_tf/updateOdometry
>
answer:
  globalTF:
    transforms:
      -
        header:
          seq: 0
          stamp:
            secs: 0
            nsecs: 0
          frame_id: my_odom_tf
        child_frame_id: base_link
        transform:
          translation:
            x: 0.0
            y: 0.0
            z: 0.0
          rotation:
            x: 0.0
            y: 0.0
            z: 0.0
            w: 0.0
...
  statusCode: 3
```

- **Enviando nubes de puntos al topic correspondiente en modo automático.**
  En el modo automático, el nodo espera hasta recibir dos nubes de puntos válidas y tratará de calcular la transformación que las relaciona en función del movimiento realizado entre las dos capturas. Una vez obtenida una medida, se desprecia la primera nube y se espera una nueva para volver a repetir sucesivamente la operación para cada par consecutivo. De este modo se procesa el máximo posible de capturas según las limitaciones de procesamiento o recepción, obteniendo.

  En este caso es posible observar los resultados monitorizados, mediante el comando "rostopic", a través del topic "odometry_main/odometry_answer" que publica en cada mensaje la odometría del último movimiento y el acumulado globalmente. Debajo se muestra un ejemplo de la llamada realizada, si bien la respuesta se omite pues se mostraría con aspecto idéntico al ejemplo anterior.

```
# Es importante poner el espacio de nombres, pero en este caso es la raiz: /
$ rostopic pub /odometry\_main/odometry\_answer
```

Resetear estimaciones. Puesta a cero

En caso de que se desee comenzar de nuevo con las mediciones de odometría, o se quiera borrar la posición estimada porque acumula ya demasiado error, se debe utilizar el servicio "resetGlobals". El reseteo no requiere de ningún argumento para funcionar y retorna el mensaje de odometría estándar del nodo, es decir, incluyendo la odometría global y local y el estado del algoritmo para comprobar que todo ello ha quedado como se espera.

```
$ rosservice call /odometry\_main/resetGlobals
```

Asignar una nueva TF a la cámara

En caso de que la cámara de profundidad no sea el centro del sistema de referencia cuyo movimiento desea estimarse, como en el el ejemplo de la figura 5.6, existe la opción de proporcionar su posición relativa a este para que el nodo calcule el movimiento del sistema completo.

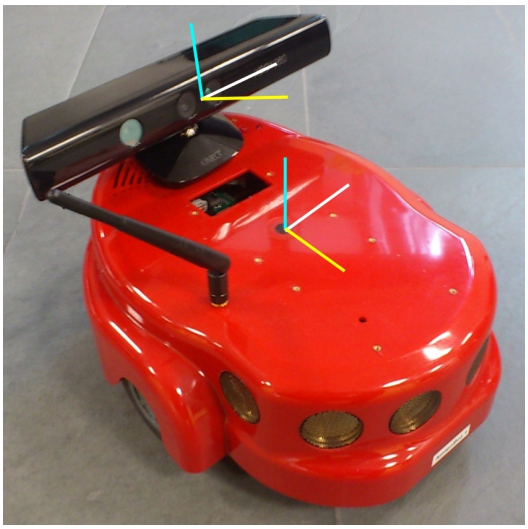

Figura 5.6: Robot amigobot con el MS Kinect y sus dos sistemas de referencia

Se trata de publicar un mensaje con los valores de posición y orientación de la cámara tomando como referencia la posición cero y la orientación consideradas para el robot. Para ello es necesario escribir un fichero con los valores adecuados asignados a los distintos campos del tipo ROS "tf::transform" usando lenguaje YAML. Se propone utilizar el fichero "camera_fixed_tf.yaml" como modelo, que se encuentra disponible en el directorio raíz del código y se trata de un fichero idéntico al que se desea construir pero con valores nulos:

```
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 0
        nsecs: 0
      frame_id: my_odom_tf
    child_frame_id: base_link
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 0.0
```

En el ejemplo siguiente se observa cómo publicar el fichero "camera_fixed_tf.yaml", una vez prepara-

do, a través del topic correspondiente “input_camera_tf”. Se utiliza el comando “rostopic pub”[7] especificando, necesariamente, el tipo de mensaje publicado “tf/tfMessage”, mientras que el modificador “–once” determina que sólo se realizará una publicación, frente a la opción por defecto de repetir el mensaje indefinidamente hasta cerrar la herramienta mediante la combinación de teclas “Ctrl + C”.

```
rostopic pub --once -f camera_fixed_tf.yaml /odometry_main/input_camera_tf tf/tfMessage
```

[7]La documentación completa sobre “rostopic pub” se puede consultar en “ *http://mediabox.grasp.upenn.edu/roswiki/rostopic.html#rostopic_pub* ”

5.4. Creación de ficheros de lanzamiento personalizados

Si bien este apartado no se refiere a una parte del proyecto sino a una herramienta que pertenece al entorno ROS, todo el desarrollo está orientado a ser compatible con los ficheros de lanzamiento por su extrema utilidad e importancia. Son una parte irrenunciable del sistema y una pieza más de la solución que simplifica al máximo el lanzamiento y arranque de nodos y grupos de nodos incluyendo configuraciones preestablecidas y también parametrizadas según las necesidades del usuario.

Los tutoriales completos y oficiales de cómo sacarle el máximo partido a dichos ficheros se encuentran en la página web de ROS[8], por lo que sólo corresponde al presente texto una pequeña introducción que ilustre cómo incluir los nodos del proyecto en un fichero de lanzamiento junto con sus parámetros de configuración. De los susodichos se extrae el siguiente ejemplo mínimo de lanzamiento de un nodo:

```
<launch>
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" />
</launch>
```

Para completarlo es posible agregar otros nodos que se quieran lanzar (como controladores de dispositivos) o incluso una jerarquía de ficheros de lanzamiento según las necesidades del usuario. Se utiliza la etiqueta “node” en el primer caso e “include” en el segundo. En el siguiente ejemplo (no funcional) se lanza primero el driver secundario (“kinect_aux”) del kinect y después se incluye el lanzador de *my_adaptor*, dejando el nodo de *my_odometry* para el final.

```
<launch>
  <node name="kinect_aux_node" pkg="kinect_aux" type="kinect_aux_node" />
  <include file="robotDrivers.launch" />
  <node name="pcl_to_tf" pkg="my_odometry" type="pcl_to_tf" />
</launch>
```

En el ejemplo anterior falta una parte importante de los ficheros de lanzamiento sobre configurar los parámetros de los nodos. Se trata de la etiqueta “arg” que realiza una carga de valores en el servidor de parámetros, que a su vez conformarán la configuración de los nodos. Su uso es tan simple como añadir una línea por cada parámetro junto con con su nombre tras el modificador “name” y su valor entre comillas usando “value”.

[8] El uso de los ficheros de lanzamiento se puede encontrar en: “ *http://ros.org/wiki/roslaunch* ”

```
<launch>
  <include file="included.launch">
    <!-- las variables que necesita included.launch deben estar definidas -->
    <arg name="hoge" value="fuga" />
  </include>
</launch>
```

En todos los casos se han omitido los atributos opcionales y las etiquetas innecesarias para limitar esta sección a lo básico. Para más información al respecto se puede consultar la documentación oficial mencionada al principio de este apartado.

6.6. Estructura de clases de la odometría

El segundo paquete de la solución, encargado de estimar la odometría visual tras cada movimiento, comparte el objetivo de favorecer la mantenibilidad, y está dividido en dos clases que son suficientes para dividirse entre la funcionalidad principal (clase *odometryComm*) y los métodos de tratamiento de TFs (clase *myTransf*). La primera clase se encarga de reunir toda la configuración necesaria para preparar el algoritmo ICP y ponerlo en marcha cada vez que obtiene una nueva nube de puntos; y la segunda hereda de la clase " *tf::Transform* " de ROS para agregarle nuevos métodos sin perjuicio de los ya existentes.

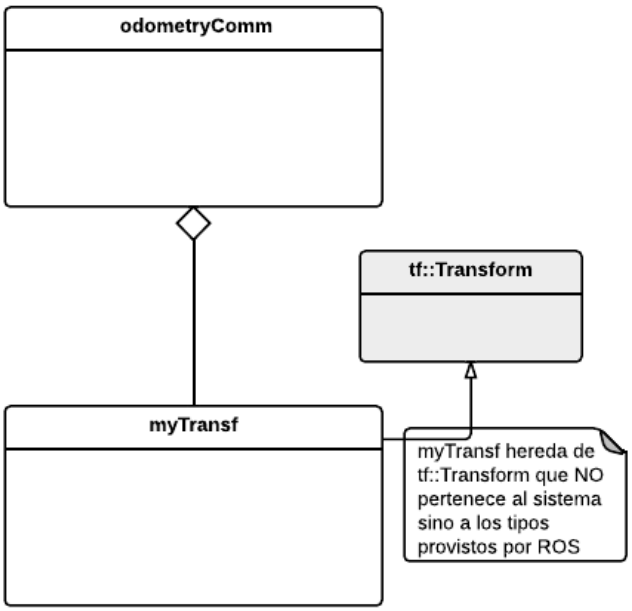

Figura 6.2: Diagrama de clases que incluye las del subsistema de odometría

A continuación se detallan las distintas clases de la estructura representada en el diagrama de clases de la figura 6.2. En cada una se describen los datos miembro y los métodos definiendo cada uno de ellos y siguiendo aún el convenio de "Python Dosctring" para documentar especificando los parámetros de entrada mediante "@param" y los valores de retorno con "@return".

El modo adecuado de lanzar el nodo de odometría, cargar las configuraciones y poner en marcha la clase principal se exponen como parte del documento de *Diseño* bajo el apartado *Código fuente del producto* (4.6).

6.6.1. Clases del paquete odometría

Clase *odometryComm*

Datos Miembro

Variables para el funcionamiento del algoritmo ICP

- **PointCloudT::Ptr previous** conserva la última nube de puntos recibida con la que se realizará la comparación en el algoritmo de ICP para estimar el movimiento realizado.
- **myTransf globalTF, lastRelativeTF, fixed_camera_transform** son las tres TFs básicas durante los cálculos: la posición y orientación actuales, el movimiento relativo a la posición global anterior y la posición fija de la cámara respecto al robot.
- **int odomStatus** guarda en cada instante el último estado de actividad del algoritmo para conocer si está en ocupado, a la espera o produjo algún error. Los estados pueden ser modificados y ampliados, pero actuamlente se identifican con una estructura *enum* con las siguientes opciones: ERROR_STATUS =-1, UNLOADED=0, INITIALIZED=1, RUNNING=2, WAITING=3 y UNRELIABLE_RESULT=4.

Adicionalmente, cabría incluir en esta lista todas las variables del apartado "Parámetros del algoritmo ICP" enumeradas en la sección 6.7 sobre los parámetros de configuración.

Variables relacionadas con el comportamiento del nodo

- **bool mIsInitialised; boost::recursive_mutex mIsInitialisedMutex; ros::AsyncSpinner *mSpinner;** son variables auxiliares para ayudar a los métodos "init" y "shutdown" a controlar el estado y sincronización del nodo.
- **boost::mutex callback_mutex** evita que dos métodos manejadores de eventos entren en conflicto por ejecutarse al mismo tiempo.
- **ros::ServiceServer *server_updateOdometry, *server_resetGlobals, *server_getLastStatus** son punteros a los servicios publicados por el nodo, que deben existir durante toda la ejecución para que otros nodos puedan acceder a las funcionalidades correspondientes.
- **ros::Publisher pcl_pub_aligned, pcl_pub_final, pcl_pub_initial, odom_answer_publisher** guardan instancias de los topics publicados por el nodo y que deben mantenerse mientras estén en funcionamiento.
- **ros::Subscriber pcl_sub, camera_tf_sub** contienen los suscriptores a topics externos para las nubes de puntos y en caso de que llegue una nueva TF con la posición de la cámara.

Funciones Miembro

Control de ejecución del nodo

- **bool init(int argc, char** argv, std::string nodeName, uint32_t rosInitOptions=0)**
  Método encargado de inicializar el objeto tras su creación. Se distingue del constructor por diversos motivos prácticos como la capacidad para reinicializar un objeto desde la sección principal del código. De ocurrir así el método detiene toda acción del objeto y lo reinicializa desde cero.

  @param argc - número de argumentos de línea de comando esperados.

  @param argv - argumentos provenientes de la línea de comando.

  @param nodeName - nombre con el que se publicará el nodo ROS en el entorno.

  @param initOptions - opciones de inicialización.

  @return Verdadero en caso de éxito y Falso en caso contrario.

- **bool shutdown()**
  Al llamar a este método, el objeto es "apagado", es decir, se detienen las acciones que llevaba a cabo y se trata de limpiar en la medida de lo posible el rastro de recursos utilizados. Este método siempre es llamado durante la "destrucción" del objeto, pero no tiene efecto si se llama con el objeto ya parado (pero no destruído).

  @return Verdadero en caso de éxito y Falso en caso contrario.

Habilitar filtrados de nubes

- **void setDownsampleFiltering(float newLeafSize)**
  Realiza los cambios necesarios para activar el filtrado de resolución (submuestreo) de las nubes de puntos. En la implementación actual se encarga de activar la marca (o *flag*) correspondiente y asignar el tamaño de "hoja" (*leaf*).

  @param newLeafSize - distancia, en metros, entre los puntos de la nube para que sean considerados de interés (lo bastante alejados).

  NOTA: La documentación no especifica el modo de medir la distancia, por lo que se asume que se trata de la distancia tridimensional al tener en cuenta todas las coordenadas (X, Y, Z).

- **void setDistanceFiltering(float minimum, float maximum)**
  Realiza los cambios necesarios para activar el filtrado por distancia en profundidad (sólo en función de X respecto a la cámara). En la implementación actual se encarga de activar la marca (o *flag*) correspondiente y asignar los valores mínimo y máximo.

  @param minimum - valores mínimo de distancia para los puntos a conservar en la nube.

  @param maximum - valores máximo de distancia para los puntos a conservar en la nube.

- **void setNoiseFiltering(float radDist, int noOfNeighbours)**
  Realiza los cambios necesarios para activar el filtrado de ruido (según la cantidad de vecinos y la distancia a la que estos se consideren tales). En la implementación actual se encarga de activar la marca (o *flag*) correspondiente y asignar los valores de radio (distancia de los considerados vecinos) y número de vecinos.

  @param radDist - máxima distancia, en metros, entre cada punto y aquellos que se considerarán "vecinos".

  @param noOfNeighbours - mínimo número de vecinos para que un punto se considere "acompañado" y conservarlo.

Métodos privados auxiliares

- **void printConfiguration()**
  Muestra, mediante texto en la consola de ejecución, la lista de parámetros disponibles y sus valores de configuración.

- **PointCloudT::ConstPtr fetchPointCloud (std::string topicName, int timeout = 20)**
  Obtiene la última nube de puntos publicada en el topic especificado (la más reciente). No comprueba la marca de tiempo puesto que se deja tal responsabilidad al método que lo invoca.

  @param topicName - nombre del topic del que obtener la nube de puntos.

  @param timeout - tiempo límite a esperar por la nube de puntos.

  @return un puntero constante a la nube de puntos obtenida o nulo si no fue posible obtenerla.

- **void apply_filters(PointCloudT::ConstPtr pointCloud1_in, PointCloudT::Ptr &pointCloud1_out)**

  Aplica, sobre la nube de puntos especificada, los distintos filtros activos gurdando el resultado en la otra nube de puntos pasada como parámetro.

  @param pointCloud1_in - nube de puntos entrante para ser filtrada.

  @param pointCloud1_out - dirección en la que almacenar la nube de puntos resultante.

- **void trimPreviousCloud(PointCloudT::Ptr &pointCloud1_out, double x = 0.75, double y = 0.9, double z = 0.9)**
  Descarta los puntos más alejados del centro de la nube según los valores "x,y,z" entendidos como el "tanto por uno" de puntos a conservar en cada coordenada.

  @param pointCloud1_out - nube de puntos entrante para ser recortada

  @param x - proporción (sobre uno) a mantener respecto al valor de 'x'

  @param y - proporción (sobre uno) a mantener respecto al valor de 'y'

@param z - proporción (sobre uno) a mantener respecto al valor de 'z'

- **double round(double value, int noDecimals)**
  Redondea un número real hasta el número de decimales especificado.

  @param value - número a redondear.

  @param noDecimals - cantidad de decimales a conservar.

- **void printMatrix4f(Eigen::Matrix4f &tMatrix)**
  Imprime la TF especificada almacenada en forma matriz de tipo "Matrix4f".

  @param tMatrix - matriz "Matrix4f" con los valores de la TF.

- **bool generate_tf(Eigen::Matrix4f &mat, double roll, double pitch, double yaw)**
  Genera una TF orientada según las rotaciones "roll,pitch,yaw" especificadas y centrada en la posición espacial (0, 0, 0).

  @param mat - variable de salida en la que almacenar la TF generada.

  @param roll - rotación a aplicar sobre el eje X.

  @param pitch - rotación a aplicar sobre el eje Y.

  @param yaw - rotación a aplicar sobre el eje Z.

- **void setOdomStatus(int status)**
  Actualiza el código de estado del algoritmo en función del parámetro especificado. Actualmente se trata como un valor entero si bien podrían utilizarse otros tipos o estructuras de datos.

  @param status - código que se almacenará en la variable "odomStatus".

Tratamiento de matrices

- **double matToDist(Eigen::Matrix4f t)**
  Calcula la distancia, en metros, recorrida según el movimiento descrito como una TF

  @param t - TF sobre la que realizar el cálculo.

- **void matToXYZ(Eigen::Matrix4f t, double& x, double& y, double& z)**
  Extrae los valores individuales de "X,Y,Z" de una TF representada como Matrix4f.

  @param t - TF de la cual obtener los valores.

@param x - variable de salida para almacenar el valor de X.

@param y - variable de salida para almacenar el valor de Y.

@param z - variable de salida para almacenar el valor de Z.

- **void matToRPY(Eigen::Matrix4f t, double& roll, double& pitch, double& yaw)**
  Extrae los valores individuales de "R,P,Y" (rotaciones) de una TF representada como Matrix4f.

  @param t - TF de la cual obtener los valores.

  @param roll - variables de salida para almacenar el valor de giro roll (alabeo).

  @param pitch - variables de salida para almacenar el valor de giro pitch (cabeceo).

  @param yaw - variables de salida para almacenar el valor de giro yaw (guiñada).

- **void filter_resulting_TF(myTransf *targetTF)**
  Realiza el filtrado de la TF especificada mediante diferentes pasos y criterios (según la implementación). Actualmente se anulan los valores cercanos a cero (considerados ruido), se cambia el sistema de coordenadas desde la cámara al robot (cuyo movimiento es el que realmente se busca) y se finaliza con un último redondeo orientado de valores cercanos a cero para eliminar valores residuales tras los cálculos.

  @param targetTF - TF sobre la cual realizar el filtrado (modificando la variable original).

Métodos conversores entre estructuras

- **myTransf eigenToTransform(Eigen::Matrix4f tMatrix)**
  Convierte la matriz de tipo "Matrix4f" en la TF de tipo "myTransf" correspondiente.

  @param tMatrix - matriz original de la TF.

  @return la TF resultante, almacenada como "myTransf" y representando el mismo movimiento o posición y orientación.

- **bool fill_in_answer(my_odometry::odom_answer &res)**
  Método para cargar, según la información actual, la estructura de datos correspondiente al mensaje que el nodo publicará como actualiación. Incluye el código de estado del algoritmo y las posiciones y orientaciones global y relativa.

  @param res - estructura en la cual almacenar la información a enviar.

  @return Verdadero siempre que se llegue al final del método sin que interrumpa una excepción.

Métodos para lanzar el algoritmo

- **Eigen::Matrix4f process2CloudsICP(PointCloudT::Ptr &cloud_initial, PointCloudT::Ptr &cloud_final, double *final_score_out=0)**
  Configura y lanza el algoritmo ICP aplicado sobre las dos nubes de puntos especificadas para obtener la TF que describe la posición relativa entre ellas.

  @param cloud_initial - nube de puntos inicial tomada en el instante considerado "de partida".

  @param cloud_final - segunda nube de puntos final, tomada tras realizar el presunto movimiento.

  @return la TF que representa la posición relativa entre las dos escenas, almacenada como matriz "Matrix4f".

- **Eigen::Matrix4f process2CloudsICP(PointCloudT::Ptr &cloud_initial, PointCloudT::Ptr &cloud_final, Eigen::Matrix4f &hint, double *final_score_out=0)**
  Sobrecarga del método del mismo nombre con el añadido de incluir una "pista" para que el algoritmo comience su búsqueda desde ella.

  @param cloud_initial - nube de puntos inicial tomada en el instante considerado "de partida".

  @param cloud_final - segunda nube de puntos final, tomada tras realizar el presunto movimiento.

  @param hint - TF que sugiere una solución "probable" de la cual parte el algoritmo para afinar la búsqueda.

  @return la TF que representa la posición relativa entre las dos escenas, almacenada como matriz "Matrix4f".

Manejadores de servicios y lanzador

- **bool update_odometry(my_odometry::odom_update_srv::Request &req, my_odometry::odom_update_srv::Response &res)**
  Manejador del servicio que ordena una nueva estimación de movimiento al algoritmo de odometría utilizando la última nube almacenada y la nueva que sea capturada.

  @param req - estructura vacía para mantener el formato de los manejadores de servicios.

  @param res - variable de respuesta en la que almacenar los datos. Almacena las TFs globa y relativa, tanto en estructuras "myTransf" como "Odometry", así como el código de estado del algoritmo.

  @return Verdadero si no se produjo ningún error.

- **bool get_last_status(my_odometry::statusMsg::Request &req, my_odometry::statusMsg::Response &res)**
  Manejador del servicio que permite solicitar el código de estado del algoritmo.

  @param req - estructura vacía para mantener el formato de los manejadores de servicios.

@param res - variable de respuesta en la que almacenar el valor solicitado. Almacena el código de estado del algoritmo.

@return Verdadero si no se produjo ningún error.

- **bool reset_globals(my_odometry::emptyRequest::Request &req, my_odometry::emptyRequest::Response &res)**
  Manejador del servicio que permite reiniciar los valores de odometría para recobrar la posición y orientación iniciales (matriz identidad: posición en el origen de coordenadas y sin rotaciones). Se ocupa de borrar también la última nube de puntos conocida, pues es anterior al reinicio desde la nueva posición inicial mencionada.

  @param req - estructura vacía para mantener el formato de los manejadores de servicios.

  @param res - variable de respuesta en la que almacenar la misma. Almacena un valor booleano correspondiente al valor de retorno del presente método.

  @return Verdadero si no se produjo ningún error.

Métodos de callback

- **void cameraTF_callback(const tf::tfMessageConstPtr& newTF)**
  Método desencadenado al recibir una TF (vía topic) para actualizar la posición de la cámara, es decir, la TF "fixed_camera_transform". El mensaje debe contener la TF que describe la posición y orientación relativas de la cámara respecto al centro de coordenadas del robot. Es necesaria para poder calcular el movimiento con respecto al robot en vez de hacerlo respecto a la cámara.

  @param newTF- TF que describe la nueva situación relativa de la cámara.

- **void PCL_callback(const PointCloudT::ConstPtr & cloud_msg)**
  Método desencadenado al recibir una nueva nube de puntos (vía topic). Si el modo "manual" está desactivado (en favor del automático), se lanza un nuevo cálculo de odometría mediante la invocación del método "common_callback_routine".

  @param cloud_msg - nueva nube de puntos recibida.

- **void STR_callback(const std_msgs::String::ConstPtr & nextTopic)**
  Método desencadenado al recibir una cadena de texto correspondiente a un topic donde leer una nueva nube de puntos. Como respuesta obtiene trata de obtener una nube de puntos del topic especificado e invoca al método "common_callback_routine" para estimar el último movimiento y la nueva posición.

  @param nextTopic - nuevo nombre de topic recibido.

- **void common_callback_routine(PointCloudT::ConstPtr & pointCloud1_aux, std::string nextTopic)**
  Este método es invocado por otros callbacks para realizar acciones comunes para el cálculo de nuevas estimaciones de odometría a partir de las dos últimas nubes de puntos obtenidas.

  @param pointCloud1_aux - nueva nube de puntos para filtrar y sobre la cual aplicar el algoritmo.

  @param nextTopic - nombre del topic del cual obtener nuevas nubes de puntos si la recibida es demasiado antigua.

6.6.2. Clase *myTransf*

Datos Miembro

Los datos miembro de esta clase están imprementados en la clase padre "tf::transform" y no pertenecen, por tanto, al alcance del proyecto actual. Aún así es necesario decir que se trata de una TF formada por dos miembros, a saber: una matriz de dimensión 3 para definir la orientación (***Matrix3x3 tf::Transform::m_basis***) y un vector de 3 posiciones para determinar la posición (u origen) de la TF (***Vector3 tf::Transform::m_origin***).

Funciones Miembro

No se incluyen aquí los métodos pertenecientes a la clase padre "tf::transform" cuya implementación no fue parte de este desarrollo y pueden ser consultados en su propia documentación de referencia[4].

Métodos conversores de TFs

- **tf::Transform getTransform()**
  Método para convertir un objeto "myTransf" (sobre el que se invoca el método) en su versión equivalente "tf::Transform"

  @return la TF original almacenada como "tf::Transform"

- **nav_msgs::Odometry transformToOdometry(ros::Time stamp = ros::Time(0), std::string frameID = "my_odom_tf", std::string childID = "base_link")**
  Convierte la TF sobre la que se invoca en una estructura de tipo "Odometry" describiendo la misma TF. Se trata de una estructura más orientado a la publicación de movimientos y estimaciones odométricas generalizada en el entorno ROS.

  @param stamp - marca de tiempo para incluir en la estructura. Se puede usar "0" como tiempo desconocido o irrelevante.

[4]La referencia de tf::transform puede encontrarse en " *http://mediabox.grasp.upenn.edu/roswiki/doc/unstable/api/tf/html/c++/classtf_1_1Transform.html* "

@param frameID - nombre para identificar la TF de referencia en el entorno ROS global.

@param childID - nombre para identificar la propia TF en el entorno ROS global.

@return la estructura resultante que expresa la TF inicial y los datos adicionales.

- **tfMessage transformToTFMsg(ros::Time stamp = ros::Time(0), std::string frameID = "my_odom_tf", std::string childID = "base_link")**
  Convierte la TF sobre la que se invoca en una estructura de tipo "tfMessage" describiendo la misma TF. Se trata de una estructura más orientado a la publicación de TFs como tales en el entorno ROS.

  @param stamp - marca de tiempo para incluir en la estructura. Se puede usar "0" como tiempo desconocido o irrelevante.

  @param frameID - nombre para identificar la TF de referencia en el entorno ROS global.

  @param childID - nombre para identificar la propia TF en el entorno ROS global.

  @return la estructura resultante que expresa la TF inicial y los datos adicionales.

Métodos para extraer, publicar o mostrar por pantalla las TFs

- **void publishOdom(ros::Publisher &odom_publisher)**
  Publica un mensaje ROS de tipo "Odometry" al entorno global con la información de la TF sobre la que se invoca y usando el canal "odom_publisher" especificado.

  @param odom_publisher - objeto en el cual depositar el mensaje para su envío a través de su topic asociado.

- **void printTransform()**
  Muestra por pantalla, en la consola de ejecución, la información de la TF "myTransf" sobre la que se invoca

- **void broadcastTransform(std::string tfChannel, std::string tfParent="map")**
  Publica la TF sobre la que se invoca al entorno ROS a través de un objeto "TransformBroadcaster" para que sea visible por el resto de los nodos.

  @param tfChannel - nombre para identificar la propia TF en el entorno ROS global.

  @param tfParent - nombre para identificar la TF de referencia en el entorno ROS global.

- **double transformToDistance()**
  Calcula la distancia, en metros, recorrida según el movimiento descrito por la TF sobre la que se invoca

  @param t - TF sobre la que realizar el cálculo.

- **double transformToRotation(double accuracy)**
  Calcula la cantidad de rotación correspondiente al movimiento descrito por la TF sobre la que se invoca. Se calcula este en base a la componente 'w' de la rotación (descrita como cuaternión) y su diferencia con '1'.

  @param t - TF sobre la que realizar el cálculo.

Métodos para modificar TFs

- **void changeCoordinates()**
  Corrige los valores originales de la TF para adaptarlos al convenio ROS que dice:

  "Los sistemas de coordenadas 3D en ROS siempre siguen la regla de la mano derecha, con X hacia delante, Y hacia la izquierda y Z hacia arriba."[5].

  Dado que las nubes de puntos siempre utilizan Z para la profundidad e Y para la altura, el cambio es de (X, Y, Z) a (Z, X, Y)

- **void round_near_zero_values(double accuracy, double rotationAccuracy)**
  Realiza el redondeo de los valores cercanos a '0' (aquellos cuya diferencia en valor absoluto sea menor que "accuracy" o, en el caso de las rotaciones, "rotationAccuracy") que son considerados "ruido".

  @param accuracy - la diferencia mínima entre '0' y cada valor para que éste no sea considerado ruido

  @param rotationAccuracy - la diferencia mínima entre '0' y cada rotación para que ésta no sea considerada ruidos

- **void get_robot_relative_tf(myTransf &fixedTF)**
  Realiza una rotación descrita por "fixedTF" sobre la TF desde la que se invoca. Se interpreta que "fixedTF" describe la relación en el espacio de dos sistemas de coordenadas fijos entre sí de tal manera que el movimiento de uno permite calcular el movimiento del otro desde su propio sistema de referencia.

  @param fixedTF - la TF fija que describe la relación entre los dos sistemas de coordenadas (orientada del original al que se pretende obtener).

- **void rotate_tf(double roll, double pitch, double yaw)**
  Realiza una rotación de la TF sobre la que es invocado y de forma acorde a los giros R,P,Y especificados

  @param roll - rotación a aplicar respecto al eje X

  @param pitch - rotación a aplicar respecto al eje Y

  @param yaw - rotación a aplicar respecto al eje Z

[5]Tomado de " `http://ros.org/wiki/tf/Overview/Transformations` ", apartado 2.

Métodos para crear TFs

- **bool generate_tf(double roll, double pitch, double yaw)**
  Genera una TF en 6 dimensiones que describe la posición en el origen (0, 0, 0) y una orientación dada por los valores de giro "R,P,Y" especificados.

  @param roll - rotación a aplicar respecto al eje X

  @param pitch - rotación a aplicar respecto al eje Y

  @param yaw - rotación a aplicar respecto al eje Z

6.7. Configuraciones y parámetros

Con la intención de dar el máximo control al usuario final, la mayoría de los parámetros del código han sido externalizados como parámetros de la aplicación. A continuación se muestra una lista de las variables que se cargan durante la inicialización de cada uno de los dos nodos.

6.7.1. Parámetros del nodo interfaz configurables durante el lanzamiento

Se muestra, debajo, la lista de parámetros configurables en *my_adaptor* que se encuentran implementados en forma de variables globales declaradas en el fichero "ctrl_interface.py" y se incluyen sus parámetros por defecto.

Nombre del fichero de traducción y valores de espera máximos para las conexiones

- **propertyConfigFile = "propertyConfigFile.yaml"**
- **topicTimeOut = 3**
- **serviceTimeOut = 3**

Direcciones de los servicios para leer del controlador mediante la interfaz

- **getStrSrv = "get/StringProperty"**
- **getIntSrv = "get/IntProperty"**
- **getFloatSrv = "get/FloatProperty"**
- **getBoolSrv = "get/BoolProperty"**
- **getDispImgSrv = "get/TopicLocation"**

Direcciones de los servicios para modificar el controlador mediante la interfaz

- **setStrSrv = "set/StrProperty"**
- **setIntSrv = "set/IntProperty"**
- **setFloatSrv = "set/FloatProperty"**
- **setBoolSrv = "set/BoolProperty"**

- **setLocationSrv = “set/TopicLocation”**

Los timeouts indican el tiempo exacto, en milisegundos, que esperarán servicios o topics por un mensaje o conexión nuevos desde que comienza la escucha hasta que asume que ha fallado y se cancela la operación. Si el tiempo especificado es demasiado escaso, el nodo renunciará a mensajes que, considera, llegan tarde aunque en realidad no haya ningún problema. Si es demasiado alto (o infinito, si se establecer a cero), el nodo se bloqueará durante ese tiempo cada vez que un mensaje o señal falle sin llegar.

Las direcciones de los servicios establecen las llamadas que deberán hacer otros nodos que deseen comunicarse con la interfaz con el objetivo de leer o modificar parámetros del controlador.

Todo ello se encuentra al principio del fichero “ctrl_interface.py”, comenzando por una lista de nombres clave del tipo “KEY_NOMBRE_PARAMETRO” que almacenan el nombre **externo** del parámetro (dentro del servidor de parámetros). Inmediatamente debajo de la lista de claves se encuentran los valores por omisión bajo el nombre “DEFAULT_NOMBRE_PARAMETRO”. Las dos listas anteriores sólo pueden ser modificadas por el programador en tiempo de desarrollo, pero marcan el procedimiento para implementar configuraciones adicionales para la inicialización del nodo.

6.7.2. Parámetros para configurar el comportamiento de la odometría

Los parámetros que se listan ahora están incluidos en el fichero *my_odometry.hpp* para integrarlos con la clase *odometryComm* , pero podrían haberse implementado en un fichero adicional exclusivo de parámetros. En cualquier caso completan el listado de datos miembro de dicha clase y se presentan a continuación con su valor por defecto y su significado.

Parámetros del algoritmo ICP

- **int maxIterations** es el máximo número de iteraciones que realizará el algoritmo ICP en búsqueda de un estimación.
- **double maxDistance** define la distancia máxima esperable entre los puntos correspondientes de las dos nubes analizadas.
- **double epsilon** es un criterio de parada basado en la diferencia máxima entre el resultado de la última iteración y el de la anterior.
- **double euclideanDistance** es un criterio de parada que depende de la suma de distancias euclidianas entre puntos correspondientes elevadas al cuadrado.
- **int maxRansacIterations** determina el máximo número de iteraciones del algoritmo RANSAC que se ejecuta en cada iteración de ICP.
- **double ransacInlierThreshold** define la distancia máxima entre puntos para que sean considerados “útiles” por el algoritmo RANSAC.
- **double ICPMinScore** dicta la puntuación mínima de una estimación de ICP para que ésta pueda considerarse válida..

- **bool doDownsampleFiltering** activa el filtrado por submuestreo
- **double leafSize** determina la precisión de la nube de puntos. Su valor es inversamente proporcional a la cantidad de puntos.
- **bool doDepthFiltering** activa el filtrado por distancia.
- **double minDepth, maxDepth** son la distancia mínima y máxima de los puntos respecto a la cámara para el filtrado por distancia.
- **bool doNoiseFiltering** activa el filtrado de ruido en las nubes de puntos.
- **int neighbours** es el mínimo número de vecinos para que un punto se considere "acompañado" y conservarlo.
- **double radius** define la máxima distancia, en metros, entre cada punto y aquellos que se considerarán "vecinos"

Nombres de publicación de topics

- **std::string inputStrRequest_topic -** recibe un texto con topic del que deberá leer la siguiente nube de puntos
- **std::string inputPCL_topic -** lee las nubes de puntos publicadas en su nombre para realizar estimaciones
- **std::string cameraTF_topic -** permite recibir una TF nueva para la cámara respecto al robot
- **std::string outputGlobalOdometry_topic -** publica la TF global con la posición y orientación estimada del robot
- **std::string outputRelativeOdometry_topic -** publica la TF relativa del último movimeinto
- **std::string outputAlignedCloud_topic -** publica la nube alineada
- **std::string outputInitialCloud_topic -** publica la nube de partida
- **std::string outputFilteredCloud_topic -** publica la nube inicial filtrada
- **std::string outputOdometryAnswer_topic -** publica el estado completo del algoritmo

Parámetros para tratamiento de nubes y TFs

- **bool manualMode** activa el modo manual si el valor es verdadero, es decir, que el algoritmo sólo captura y estima un nuevo movimiento si así se lo solicita otro nodo o usuario.
- **bool ignoreTime** permite ignorar las marcas de tiempo de las nubes de puntos cuando es verdadero.
- **double measureAccuracy, rotationAccuracy** definen respectivamente la precisión lineal, en metros, y la rotación esperada de la cámara. Si un valor es menor que la precisión de la cámara para medirlo se asume que debe ser ruido.

- **double cloud_trim_x, cloud_trim_y, cloud_trim_z** son las porciones (sobre ’1’) de cada escena que deberán seguir en el rango de visión tras un movimiento. Dicho de otro modo, “1-cloud_trim” es la parte que se considera “margen” de la nube anterior y se elimina al compararla con la siguiente.
- **double kinectRoll,kinectPitch, kinectYaw** son los valores de giro “alabeo, cabeceo y guiñada” que determinan la orientación de la cámara respecto al robot.