Contenido

1. Introducción a PCLpy

1.1. ¿Qué es?

PCLpy es un wrapper que permite la utilización de PCL en python, para su integración con ROS, e incluye soporte para características básicas de los módulos: visualization, pcd_io, filters y registration, así como otras operaciones sobre nubes de puntos: concatenación, iteración.

1.2. ¿Cómo funciona?

PCLpy utiliza como base nubes de puntos del tipo *PointXYZRGB*, y funciona mediante la serialización de mensajes de ROS. Por convención, se utilizan los mismos nombres de clases y funciones que la versión original de PCL en c++; aunque el orden de los parámetros y/o valores de retorno pueden no ser los mismo.

1.3. ¿Cómo lo instalo?

Dado que *PCLpy* utiliza la serialización de los mensajes de ROS, es necesario añadirlo como un paquete al espacio de trabajo, a fin de construirlo y utilizarlo.

Para ello, se listan los siguientes pasos, requeridos para su instalación:

1. Añade el espacio de trabajo a $ROS_PACKAGE_PATH

```
user@hostname:~/catkin_ws$ source devel/setup.bash
```

2. Mueve la carpeta pclpy al espacio de trabajo

```
user@hostname:~$ mv pclpy ~/catkin_ws/src
```

3. Construye el paquete

```
user@hostname:~/catkin_ws$ catkin_make --pkg pclpy -DCMAKE_BUILD_TYPE=Release
```

1.4. ¿Cómo lo utilizo?

Cada vez que desee ejecutar un script que utilice *PCLpy* en una terminal nueva, es necesario añadir el espacio de trabajo a $ROS_PACKAGE_PATH (para más información consulte el paso 1 en la sección de instalación).

Para su utilización, es necesario importar el módulo (o los submódulos requeridos) dentro el script. Por ejemplo, si desea utilizar el filtro *VoxelGrid*, utilize:

```
from pclpy.filters import VoxelGrid
```

2. PCLpy common

2.1. ¿Qué es?

Son una serie de operaciones propias de PCL para la manipulación de nubes de puntos, que el mensaje de ROS (PointCloud2) no soporta nativamente.

2.2. ¿Cómo funciona?

PCLpy common, incluye los siguientes métodos/atributos a la clase PointCLoud2:

- *points*. Regresa un generator de los puntos contenidos en la nube, permitiendo iterar sobre ellos o almacenarlos en alguna estructura de datos externa.
- *__add__*. Concatena dos nubes de puntos.

2.3. ¿Cómo lo utilizo?

Para utilizar PCLpy common, simplemente importe pclpy en su script:

```
import pclpy
```

2.3.1. El atributo points.

Aquí un ejemplo de la utilización de points:

```
1 import pclpy
2
3 cloud = loadPCDFile('inputCloud.pcd')
4
5 for point in cloud.points:
6         print( point )
```

2.3.2. Concatenando nubes de puntos

Para concatenar dos nubes de puntos, utilice el operador de adición (+)
Aquí un ejemplo de su utilización:

```
1 from __future__ import print_function
2 import pclpy
3
4 cloud1 = loadPCDFile('inputCloud1.pcd')
5 cloud2 = loadPCDFile('inputCloud2.pcd')
6
7 clouds = cloud1 + cloud2
8
9 print(cloud1, cloud2, clouds, sep='\n')
```

3. Módulo pcdIO

3.1. ¿Qué es?

Es un módulo de *PCLpy* que corresponde con la biblioteca pcd_io de PCL, y contiene las funciones para la lectura y escritura de archivos pcd (Point Cloud Data).

3.2. ¿Cómo funciona?

PcdIO contiene sólo dos funciones: *loadPCDFile* y *savePCDFile*, para la lectura y escritura de archivos pcd, respectivamente.

3.2.1. loadPCDFile

La función *loadPCDFile* es el equivalente a *pcl::io::loadPCDFile*, perteneciente a *<pcl/io/pcd_io.h>*. Su modo de operación es el siguiente:
→ Recibe como parámetro el nombre del archivos
→ Regresa un *PointCloud2* con la nube de puntos obtenida

A diferencia de *pcl::io::loadPCDFile*, no acepta parámetros extra como: *origin*, *orientation*. Para más información sobre la utilización de estos parámetros, consulta la documentación oficial de PCL.

3.2.2. savePCDFile

La función *savePCDFile* es el equivalente a *pcl::io::savePCDFile*, perteneciente a *<pcl/io/pcd_io.h>*. Su modo de operación es el siguiente:
→ Recibe como parámetros el nombre del archivo y la nube de puntos (de tipo *PointCloud2*)
→ Regresa un *bool* indicando el éxito de dicha operación

A diferencia de *pcl::io::savePCDFile*, *pcdio.savePCDFile* no acepta como parámetro *binary_mode*, por defecto se utiliza en *false*; o cualquier otro parámetro extra indicado en la documentación de *pcl::io::savePCDFile*.

3.3. ¿Cómo lo utilizo?

Para utilizar *pcdIO* en su programa, sólo basta con importar el módulo *pcdio* de *PCLpy*.

```
import pclpy.pcdio
```

3.3.1. Leyendo una nube de puntos desde un archivo.

Para leer una nube de puntos, a partir de un archivo PCD, utilice la función *loadPCDFile* contenida en el modulo *pcdIO* de *PCLpy*. Para más información sobre su funcionamiento, consulte la subsección *¿Cómo funciona?* De la sección *Módulo pcdIO*.

Aquí un ejemplo para la lectura de un archivo pcd:

```
1 from pclpy.pcdio import loadPCDFile
2
3 cloud = loadPCDFile('inputCloud.pcd')
```

Explicación línea por línea:

- Línea 1, importamos únicamente la función *loadPCDFile* del módulo *pcdIO* de *PCLpy*
- Línea 3, cargamos en *cloud* el archivo *'inputCloud.pcd'*.

3.3.2. Escribiendo una nube de puntos en un archivo.

Para guardar una determinada nube de puntos en un archivo PCD, utilice la función *savePCDFile*, contenida en el modulo *pcdIO* de *PCLpy*. Para más información sobre su funcionamiento, consulte la subsección *¿Cómo funciona?* De la sección *Módulo pcdIO*.

Aquí un ejemplo para la escritura de una nube de puntos en un archivo pcd:

```
1 from pclpy.pcdio import loadPCDFile, savePCDFile
2
3 cloud = loadPCDFile('inputCloud.pcd')
4
5 ans = savePCDFile( cloud, 'outputCloud.pcd' )
6
7 print( 'Exito al guardar' if ans else 'Ocurrió un error al guardar' )
```

Explicación línea por línea:

- Línea 1, importamos las funciones *loadPCDFile* y *savePCDFile* del módulo *pcdIO* de *PCLpy*.
- Línea 3, cargamos en *cloud* la nube de puntos del archivo *'inputCloud.pcd'*
- Línea 5, guardamos la nube de puntos (*cloud*) en un archivo con el nombre *'outputCloud.pcd'*; el resultado de tal operación se lo asignamos a *ans*.
- Línea 7, imprimimos el resultado de la operación: *'Éxito al guardar'* si fue satisfactorio, '*Ocurrió un error al guardar'* en caso contrario.

4. Módulo visualization

4.1. ¿Qué es?

Es un módulo de *PCLpy* que corresponde con la biblioteca visualization de PCL, con ella es posible visualizar una (o más) nube(s) de puntos en un plano 3D.

4.2. ¿Cómo funciona?

Visualization contiene sólo una clase para desplegar nubes de puntos en un plano 3D: *PCLVisualizer*.

4.2.1. PCLVisualizer

La clase *PCLVisualizer*, contiene los siguientes métodos:

→ Para la utilización de viewports
- *createViewPort*. Recibe como parámetros la posición relativa del nuevo viewport ($[xmin, ymin] \rightarrow [xmax, xmax]$). Regresa el id del nuevo viewport creado.

→ Para personalizar la ventana sobre la que se muestra(n) la(s) nube(s) de puntos
- *setBackgroundColor*. Recibe como parámetros la combinación de colores como canales separados *(R,G,B)* y el viewport sobre el cual se aplicará el color. Por defecto se aplica a todos los viewports (viewport=0).
- *setWindowName*. Recibe como parámetro el nombre de la ventana, también puede especificarse en el constructor del objeto. Por defecto, su valor es la cadena vacía.

→ Para añadir o quitar nube(s) de puntos en el plano 3D.
- *addPointCloud*. Recibe como parámetros una nube de puntos y el viewport al cual se añadirá dicha nube, se asigna automáticamente un nombre (id); y regresa el id asignado a la nube de puntos. Si no se pudo agregar, este id es la cadena vacía. Por defecto, se añade la nube de puntos a todos los viewports (viewport=0).
- *removePointCloud*. Recibe como parámetros el id y viewport de la nube de puntos a remover de la pantalla; y regresa si dicha operación fue posible (*bool*).

→ Para iniciar o detener el visualizador interactivo:
- *spin*. Inicia el visualizador interactivo
- *spinOnce*. Recibe como parámetros el tiempo que durará la interacción con el visualizador y si debe o no forzar la actualización de la pantalla. Por defecto se usan *time=1* y *force_redraw=False*.
- *close*. Detiene el visualizador

→ Para verificar el estado del visualizador:
- *wasStopped*. Regresa verdadero si el usuario intentó cerrar la ventana.

4.3. ¿Cómo lo utilizo?

Para utilizar *PCLVisualizer* en su programa, sólo basta con importar la clase *PCLVisualizer* del módulo *visualization* de *PCLpy*.

```
from pclpy.visualization import PCLVisualizer
```

4.3.1. Visualizando una nube de puntos

Para visualizar una nube de puntos en un plano 3D, utilice una instancia de la clase *PCLVisualizer*, contenida en el modulo *visualization* de *PCLpy*. Para más información sobre su funcionamiento, consulte la subsección *¿Cómo funciona?* De la sección *Módulo visualization*.

Aquí un ejemplo para visualizar una nube de puntos:

```
1 from pclpy.visualization import PCLVisualizer
2 from pclpy.pcdio import loadPCDFile
3
4 vis = PCLVisualizer()
5
6 cloud = loadPCDFile('inputCloud.pcd')
7 vis.addPointCloud(cloud)
8
9 vis.spin()
```

Explicación línea por línea:

- Línea 1~2, importamos la clase *PCLVisualizer* del módulo *visualization*, y la función *loadPCDFile* del módulo *pcdIO*, de *PCLpy*
- Línea 4, creamos una instancia de *PCLVisualizer* con el nombre *vis*.
- Línea 6, cargamos en *cloud* la nube de puntos contenida en *'inputCloud.pcd'*
- Línea 7, añadimos la nube de puntos al visualizador.
- Línea 9, iniciamos el interactor del visualizador.

4.3.2. Quitando una nube de puntos

Para quitar una nube de puntos del plano 3D, utilice el método *removePointCloud* del visualizador, utilizando como argumento el id de dicha nube.

Aquí un ejemplo para quitar una nube de puntos desplegada:

```
1 from pclpy.visualization import PCLVisualizer
2 from pclpy.pcdio import loadPCDFile
3
4 vis = PCLVisualizer()
5
6 cloud1 = loadPCDFile('inputCloud1.pcd')
7 cloud2 = loadPCDFile('inputCloud2.pcd')
8
9 cid1 = vis.addPointCloud(cloud1)
```

```
10 cid2 = vis.addPointCloud(cloud2)
11
12 vis.spin()
13
14 vis.removePointCloud(cid2)
15 vis.spin()
```

Explicación línea por línea:

- Línea 1~2, importamos la clase *PCLVisualizer* del módulo *visualization*, y la función *loadPCDFile* del módulo *pcdIO*, de *PCLpy*
- Línea 4, creamos una instancia de *PCLVisualizer* con el nombre *vis*.
- Línea 6~7, cargamos dos nubes de puntos, contenidas en *'inputCloud1.pcd' e 'inputCloud2pcd'*, respectivamente.
- Línea 9~10, añadimos las nubes de puntos al visualizador, guardando su *cloud_id*, en *cid1* y *cid2*.
- Línea 12, iniciamos el interactor del visualizador, para comprobar que ambas nubes de puntos fueron añadidas.
- Línea 14, removemos la segunda nube de puntos, utilizando *cid2* como parámetro.
- Línea 15, iniciamos el interactor del visualizador, para comprobar que ha sido removida.

4.3.3. Personalizando el visualizador

Para personalizar la ventana del visualizador, utilice los métodos *setBackGroundColor* y *setWindowName*, pasando como argumentos el color y el nombre de la ventana, respectivamente.

Aquí un ejemplo del visualizador personalizado:

```
 1 from pclpy.visualization import PCLVisualizer
 2 from pclpy.pcdio import loadPCDFile
 3
 4 vis = PCLVisualizer()
 5 vis.setBackgroundColor(59, 156, 150)
 6 vis.setWindowName('My custom title')
 7
 8 cloud = loadPCDFile('inputCloud1.pcd')
 9
10 vis.addPointCloud(cloud)
11
12 vis.spin()
```

Explicación línea por línea:

- Línea 1~2, importamos la clase *PCLVisualizer* del módulo *visualization*, y la función *loadPCDFile* del módulo *pcdIO*, de *PCLpy*
- Línea 4, creamos una instancia de *PCLVisualizer* con el nombre *vis*.
- Línea 5, establecemos el fondo de color en *(r=59, g=156, b=150)*
- Línea 6, establecemos el nombre de la ventana en *'My custom title'*
- Línea 8, cargamos en *cloud* la nube de puntos contenida en *'inputCloud.pcd'*

- Línea 10, añadimos la nube de puntos al visualizador
- Línea 12, iniciamos el interactor del visualizador

4.3.4 Utilizando viewports

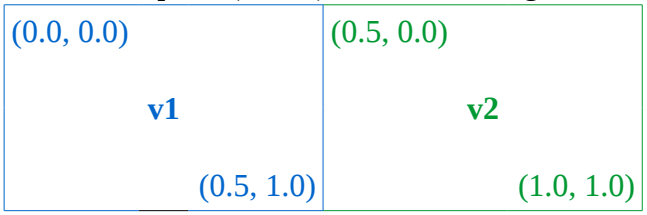
Para utilizar viewports en su programa, cree uno utilizando el método *createViewPort* pasándole como argumentos las posiciones relativas del nuevo viewport. Estas posiciones relativas se refieren al porcentaje de la ventana, por lo que se expresan en el rango $[0,1]$.

Aquí un ejemplo utilizando viewports:

```
 1 from pclpy.visualization import PCLVisualizer
 2 from pclpy.filters import VoxelGrid
 3 from pclpy.pcdio import loadPCDFile
 4
 5 i_cloud = loadPCDFile('cloud0.pcd')
 6
 7 vg = VoxelGrid()
 8 vg.setInputCloud(i_cloud)
 9 vg.setLeafSize(.01, .01, .01)
10 r_cloud = vg.filter()
11
12 vis = PCLVisualizer()
13
14 v1 = vis.createViewPort( 0., 0., .5, 1. )
15 v2 = vis.createViewPort( .5, 0., 1., 1. )
16
17 vis.setBackgroundColor( 255, 0, 0, v1 )
18 vis.setBackgroundColor( 0, 255, 0, v2 )
19
20 vis.addPointCloud( i_cloud, v1 )
21 vis.addPointCloud( r_cloud, v2 )
22
23 vis.spin()
```

Explicación línea a línea:

- Líneas 1~3, importamos *PCLVisualizer*, *VoxelGrid*, y *loadPCDFile* de los módulos *visualization*, *filters* y *pcdio* de PCLpy, respectivamente.
- Líneas 5~10, cargamos en *i_cloud* la nube contenida en *'cloud0.pcd'* y guardamos en *r_cloud* la nube resultante de aplicar un filtro *voxel_grid* a *i_cloud*.
- Línea 12, creamos en *vis* una instancia de *PCLVisualizer*.
- Líneas 14~15, creamos dos viewports (*v1*, *v2*) utilizando la siguiente referencia:



- Líneas 17~23, asignamos un color de fondo diferente para cada viewport, añadimos *i_cloud* al viewport *v1*, *r_cloud* al viewport *v2* e iniciamos el interactor del visualizador.

5. Módulo filters

5.1. ¿Qué es?

Es un módulo de *PCLpy* que contiene diferentes mecanismos para aplicar filtros a nubes de puntos.

5.2. ¿Cómo funciona?

Los filtros contenidos en este módulo son: VoxelGrid, Passthrough, StatisticalOutlierRemoval.

Cada uno de ellos, posee dos métodos en común:

- *setInputCloud*. Recibe como parámetro una nube de puntos, sobre la cual se aplicará el filtro.
- *filter*. Regresa una nube de puntos, con el filtro aplicado.

5.2.1. Filtro Passthrough

Mantiene un conjunto de puntos, de una nube de entrada dada, utilizando para ello restricciones de posición, aplicadas a un eje en particular. Los puntos que no se apegan a esta restricción, son eliminados del conjunto.

Contiene dos métodos de configuración del filtro:

- *setFilterFieldName*. Recibe como parámetro el nombre del eje sobre el cual se aplicarán las restricciones. No utiliza ningún eje por defecto, por lo que debe configurarlo apropiadamente antes de aplicarlo.
- *setFilterLimits*. Recibe como parámetros los limites, inferior y superior, permitidos para los puntos en la nube, en el eje indicado con *setFilterFieldName*. Por defecto se utilizan los valores *FLT_MIN* y *FLT_MAX*, para más información sobre estos valores consulte la documentación oficial de PCL.

5.2.2. Filtro VoxelGrid

El filtro VoxelGrid crea una rejilla de vóxeles (en 3D) sobre una nube de puntos dada. En cada vóxel, todos los puntos presentes estarán aproximados (*downsampled*) con su centroide. Representando entonces la misma superficie, con un menor número de puntos.

Contiene un método de configuración del filtro:

- *setLeafSize*. Recibe como parámetros las dimensiones del vóxel (para x, y, z). Este valor está expresado en metros, por lo que 0.01 será 1 centímetro. Por defecto, las dimensiones son 0.

5.2.3. Filtro StatisticalOutlierRemoval

El filtro StatisticalOutlierRemoval utiliza las estadísticas de los puntos vecinos para filtrar aquellos puntos aislados (*outlier*). Internamente itera dos veces sobre la nube de puntos:

1. Calcula la distancia promedio de cada punto con sus *k* vecinos más cercanos. Enseguida se

determina la distancia límite (*threshold*) utilizando la media y la desviación estándar de las distancias calculadas.
2. Los puntos son clasificados como aislados (*outlier*) y no aislados (*inlier*), si su distancia promedio es mayor o menor a su distancia límite (*threshold*), respectivamente.

La distancia limite se calcula como sigue:
$mean + stddev\_mult * stddev$
Dónde Mean es la distancia media, stddev_mult es un multiplicador para la desviación estándar y stddev es la desviación estándar.

Este filtro, contiene los siguientes métodos de configuración:

- *setMeanK*. Recibe como parámetro el número de vecinos más cercanos, para estimar la distancia media. Por defecto usa el valor de 1
- *setStddevMulThresh*. Establece el multiplicador para la desviación estándar, para el cálculo de la distancia límite (*threshold*). Por defecto usa el valor de 0.0
- *setNegative*. Establece si se eliminan los puntos aislados (*outlier*) o los no aislados (*inlier*). Por defecto tiene el valor de *false*, resultando en la eliminación de los outlier.

5.3. ¿Cómo lo utilizo?

Para utilizar filtros en su programa, sólo basta con importar el filtro deseado, desde el módulo *filters* de *PCLpy*.

5.3.1. Filtrando con Passthrough

Para filtrar utilizando *Passthrough*, impórtelo a su programa utilizando:

```
from pclpy.filters import Passthrough
```

Aquí un ejemplo aplicando el filtro passthrough:

```
from pclpy.pcdio import loadPCDFile
from pclpy.visualization import PCLVisualizer
from pclpy.filters import PassThrough

cloud = loadPCDFile('inputCloud.pcd')

_pass = PassThrough()
_pass.setInputCloud(cloud)
_pass.setFilterFieldName('x')
_pass.setFilterLimits( 0.0, 1.5 )

filtered_cloud = _pass.filter()

vis = PCLVisualizer()

vis.addPointCloud(filtered_cloud)
vis.spin()
```

Explicación línea por línea:

- Líneas 1~2, importamos *loadPCDFile* y *PCLVisualizer*. Para mas información consulte la seccines de los módulos *pcdIO* y *visualization*.
- Línea 3, importamos el filtro *Passthrough*, del módulo *filters* de *PCLpy*.
- Línea 5, cargamos en *cloud*, la nube de puntos contenida en *'inputCloud.pcd'*
- Líneas 7~10, creamos una instancia del filtro (en *_pass*) y establecemos la restricción en el eje *x* entre 0.0 y 1.5
- Línea 12, guardamos en *filtered_cloud*, el resultado de aplicar el filtro a la nube de puntos.
- Líneas 14~17, visualizamos la nube de puntos filtrada (*filtered_cloud*) en un plano 3D.

5.3.2. Filtrando con VoxelGrid

Para filtrar utilizando *VoxelGrid*, impórtelo a su programa utilizando:

```
from pclpy.filters import VoxelGrid
```

Aquí un ejemplo aplicando el filtro voxelgrid:

```
 1 from __future__ import print_function
 2 from pclpy.pcdio import loadPCDFile
 3 from pclpy.filters import VoxelGrid
 4
 5 cloud = loadPCDFile('inputCloud.pcd')
 6
 7 print( 'Points before filtering: {} data points {}'.format(\
 8         cloud.width * cloud.height, cloud.field_names ) )
 9
10 vg = VoxelGrid()
11 vg.setInputCloud(cloud)
12 vg.setLeafSize(0.01, 0.01, 0.01)
13
14 filtered_cloud = vg.filter()
15
16 print( 'Points after filtering: {} data points {}'.format(\
17         cloud.width * cloud.height, cloud.field_names ) )
```

Explicación línea por línea:

- Línea 2, importamos *loadPCDFile* del módulo *pcdIO* de *PCLpy*
- Línea 3, importamos el filtro *VoxelGrid*, del módulo *filters* de *PCLpy*.
- Líneas 5~7, cargamos en *cloud* la nube de puntos contenida en *'inputCloud.pcd'* e imprimimos el número de puntos antes de ser filtrada.
- Líneas 10~12, creamos una instancia del filtro *VoxelGrid* y la configuramos con la nube de puntos (*cloud*), además de las dimensiones del vóxel en 1cm (0.01)
- Línea 14, guardamos en *filtered_cloud* la nube resultante de aplicar el filtro
- Línea 16, imprimimos el número de puntos que contiene la nube filtrada

5.3.3. Filtrando con StatisticalOutlierRemoval

Para filtrar utilizando *StatisticalOutlierRemoval*, impórtelo a su programa utilizando:

```
from pclpy.filters import StatisticalOutlierRemoval
```

Aquí un ejemplo utilizando el filtro outlier:

```
 1 from pclpy.pcdio import loadPCDFile
 2 from pclpy.visualization import PCLVisualizer
 3 from pclpy.filters import StatisticalOutlierRemoval
 4
 5 cloud = loadPCDFile('inputCloud.pcd')
 6
 7 sor = StatisticalOutlierRemoval()
 8 sor.setInputCloud(cloud)
 9 sor.setMeanK(100)
10 sor.setStddevMulThresh(1.0)
11
12 filtered_cloud = _sor.filter()
13
14 vis = PCLVisualizer()
15
16 vis.addPointCloud(filtered_cloud)
17 vis.spin()
```

Explicación línea por línea:

- Líneas 1~2, importamos *loadPCDFile* y *PCLVisualizer*. Para mas información consulte la seccines de los módulos *pcdIO* y *visualization*.
- Línea 3, importamos el filtro *StatisticalOutlierRemoval*, del módulo *filters* de *PCLpy*.
- Línea 5, cargamos en *cloud*, la nube de puntos contenida en *'inputCloud.pcd'*
- Líneas 7~10, creamos una instancia del filtro (en *sor*), establecemos el número de vecinos en 100 y el multiplicador de la desviación estándar en 1.0
- Línea 12, guardamos en *filtered_cloud*, el resultado de aplicar el filtro a la nube de puntos.
- Líneas 14~17, visualizamos la nube de puntos filtrada (*filtered_cloud*) en un plano 3D.

6. Módulo registration

6.1. ¿Qué es?

Es un módulo de PCLpy, con el que es posible la combinación de varios conjuntos de datos (nubes de puntos), en un modelo global consistente. Estos conjuntos de datos pueden corresponder a la misma escena, pero vista desde una perspectiva diferente.

6.2. ¿Cómo funciona?

La idea clave es identificar puntos correspondientes entre conjuntos de datos y encontrar una transformación que minimice la distancia (error de alineación) entre dichos puntos.

Para hacer coincidir dos conjuntos de puntos (*datasets*):

1. De un conjunto de puntos, identificar los puntos de interés (*keypoints*) que mejor representen la escena en ambos datasets.
2. Para cada punto clave, calcular una característica descriptiva.
3. De un conjunto de características descriptivas, junto con sus posiciones en *XYZ* de ambas datasets, estimar un conjunto de correspondencias (partes coincidentes) basado en las similitudes entre características y posiciones.
4. Dado que la información suele contener ruido, no todas las correspondencias son válidas, así que se rechaza aquellas correspondencias erróneas que puedan contribuir negativamente al proceso. Esto puede ser hecho usando RANSAC o reduciendo su cantidad y en su lugar usar solo un porcentaje de las correspondencias encontradas.
5. Del conjunto restante (buenas correspondencias), se estima una transformación (rotación/traslación).

PCLpy incluye una clase para dicha operación: *IterativeClosestPoint*.

6.2.1. IterativeClosestPoint (ICP)

El funcionamiento de esta clase, puede resumirse en el siguiente enunciado: dado una nube de puntos (*cloud_in*) encuentra una transformación tal que *cloud_in* sea aproximadamente igual a una nube de puntos objetivo (*cloud_out*).

El proceso general puede ser descrito con los siguientes pasos:

1. Búsqueda de correspondencias
2. Rechazo de las malas correspondencias
3. Estimación de una transformación usando las buenas correspondencias
4. Iterar

El algoritmo ICP puede tener diferentes criterios de terminación:

1. Número máximo de iteraciones alcanzado
2. La diferencia entre la transformación previa y la estimación actual es más pequeña que épsilon (un valor determinado por el usuario)

3. El error es más pequeño que el límite definido por el usuario (euclidean fitness)

La clase ICP contiene los siguientes métodos de configuración:

- *setInputSource*. Recibe como parámetro la nube de puntos de origen, sobre la que se aplicará el alineamiento para hacerla coincidir con la nube de puntos objetivo.
- *setInputTarget*. Recibe como parámetro la nube de puntos objetivo, utilizada como referencia para el alineamiento de la nube de puntos de origen.
- *setMaximumIterations*. Recibe como parámetro el número máximo de iteraciones. Por defecto itera un máximo de 10 veces.
- *setMaxCorrespondenceDistance*. Recibe como parámetro la distancia máxima de correspondencia entre puntos (*source* ↔ *target*). Por defecto utiliza el valor de $\sqrt{\text{FLT\_MAX}}$
- *setTransformationEpsilon*. Recibe como parámetro la transformación épsilon, para establecer otro criterio de terminación. Por defecto, utiliza el valor de 0.0
- *setEuclideanFitnessEpsilon*. Recibe como parámetro el máximo error permitido, para establecer otro criterio de terminación. Por defecto, utiliza el valor *FLT_MIN*
- *align*. Estima la transformación y regresa la nube de puntos de origen, con dicha transformación aplicada.

6.3. ¿Cómo lo utilizo?

Para utilizar registration en su programa, sólo basta con importar el algoritmo deseado, desde el módulo *registration* de *PCLpy*.

6.3.1. Utilizando IterativeClosestPoint (ICP)

Para registrar varias nubes de puntos utilizando ICP, impórtelo en su programa utilizando:

```
from pclpy.registration import IterativeClosestPoint
```

Aquí un ejemplo utilizando ICP:

```
from pclpy.pcdio import loadPCDFile
from pclpy.visualization import PCLVisualizer
from pclpy.filters import PassThrough
from pclpy.registration import IterativeClosestPoint

pcd_files = [ 'inputCloud0', 'inputCloud1' ]

clouds = [ loadPCDFile(x+'.pcd') for x in pcd_files ]

_pass = PassThrough()
_pass.setFilterFieldName('z')
_pass.setFilterLimits( 0.0, 5.0 )

# filtrar todas
for i in range(len(clouds)):
        _pass.setInputCloud(clouds[i])
```

```
17         clouds[i] = _pass.filter()
18
19 icp = IterativeClosestPoint()
20
21 vis = PCLVisualizer()
22
23 t_cloud = clouds[0]
24 vis.addPointCloud(t_cloud)
25
26 for i in range( 1, len(clouds) ):
27         icp.setInputSource( clouds[i] )
28         icp.setInputTarget( t_cloud )
29         t_cloud = icp.align()
30         vis.addPointCloud( t_cloud )
31 vis.spin()
```

Explicación línea por línea:

- Líneas 1~3, importamos *loadPCDFile*, *PCLVisualizer* y *Passthrough* desde los módulos *pcdIO*, *visualization* y *filters*, respectivamente, de *PCLpy*.
- Línea 4, importamos *IterativeClosestPoint* desde el módulo *registration* de *PCLpy*.
- Líneas 6~8, cargamos dos archivos pcd y los almacenamos en una lista (*clouds*).
- Líneas 10~17, filtramos todas las nubes de puntos, utilizando passthrough.
- Línea 19, creamos una instancia de *IterativeClosestPoint*
- Línea 21, creamos una instancia del *PCLVisualizer*
- Líneas 23~30, aplicamos ICP sobre cada nube de puntos (a partir del índice 1), alineándola con la 1ra nube de puntos cargada (con índice 0). Enseguida, las nubes transformadas obtenidas son agregadas al visualizador
- Línea 30, inicia el interactor del visualizador.

Anexo A. Ampliando la funcionalidad de PCLpy

A.1 ¿Cómo agrego métodos a una clase ya existente?

Para agregar métodos a una clase ya existente, siga los siguientes pasos:

1. Edite el archivo de cabecera de la clase (ej: pcdio.h), y agregue los métodos que desee.

2. Edite el archivo de implementación (c++) de la clase (ej: pcdio_wrapper.cpp), y añada la implementación de los métodos del paso 1.

3. Edite el archivo de implementación (py) de dicha clase (ej: _pcl_pcdio_wrapper_py.py), añadiendo los métodos definidos en el paso 1.

A continuación se explica, de forma más detallada, cada uno de los pasos previamente descritos.

A.1.1 Editando el archivo de cabecera

Los archivos de cabecera, se encuentran en el directorio *include*, del paquete pclpy. Todos estos se encuentran organizados de acuerdo al módulo al que pertenecen; por ejemplo, las cabeceras para los filtros se encuentran en el subdirectorio *filters*, dentro de *include*.

Una vez localizado el archivo de cabecera de su interés, añada los métodos deseados, utilizando el tipo *std::string* para cada uno de los parámetros y valores de retorno requeridos.

Por ejemplo: el método *loadPCDFile*, que recibe un parámetro y regresa otro.

```
std::string loadPCDFile( std::string fileName );
```

A.1.2 Editando el archivo de implementación (c++)

Los archivos de implementación, se encuentran en el directorio *src*, del paquete pclpy. Todos estos se encuentran organizados de acuerdo al módulo al que pertenecen, su nombre es similar al del archivo de cabecera, sólo que al final lleva la palabra wrapper; por ejemplo, el archivo de implementación para *pcdio.h* se llama *pcdio_wrapper.cpp*.

Una vez localizado el archivo de implementación, añada la implementación de los métodos definidos previamente en el archivo de cabecera. Para ello, es necesario la utilización de los mensajes de ROS para la serialización o deserialización de los parámetros (de entrada y salida), y las funciones *to_python* y *from_python*. El archivo de cabecera *definitions.h* ya incluye las bibliotecas de algunos mensajes de ROS, así como del archivo con las funciones *to_python* y *from_python*; puede revisar si dicho archivo ya contiene el tipo de dato que desee utilizar.

Por ejemplo: para el método *loadPCDFile*.

```
1 std::string PcdIOWrapper::loadPCDFile( std::string fileName ) {
2         sensor_msgs::PointCloud2 cloud2; // pointcloud2
3         std_msgs::String t_filename =
```

```
 4                 from_python<std_msgs::String>( fileName );
 5         int ret =
 6                 pcl::io::loadPCDFile( t_filename.data, cloud2 );
 7         if ( ret == -1 )
 8         {
 9                 PCL_ERROR( "Could not load file: '%s'\n", t_filename.data.c_str() );
10         }
11
12         return to_python( cloud2 );
13 }
```

Se utilizan los mensajes *std_msgs::String* y *sensor_msgs::PointCloud2* de ROS, para deserializar el nombre del archivo pcd y para almacenar la nube contenida en dicho archivo, respectivamente. En la línea 4, se utiliza la función *from_python<T>* para deserializar el parámetro *fileName*Posteriormente, en la línea 12, se utiliza la función *to_python* para serializar la nube cargada en *cloud2* y regresar una cadena con ella como contenido.
Nótese que para acceder al valor de un mensaje de ROS de utiliza el atributo *data* (línea 6).

Una vez definido el método, agréguelo a BOOST_PYTHON_MODULE (al final del mismo archivo) para que al crear la *lib* se incluya como método de interfaz pública. Para ello, utilice la siguiente sintaxis:

```
   .def( "nombre_metodo", &clase::nombre_metodo )
```

Justo antes del *;* del último método registrado. Por ejemplo:

```
1 BOOST_PYTHON_MODULE( _pcl_pcdio_wrapper_cpp ) {
2         boost::python::class_<PcdIOWrapper> ( "PcdIOWrapper", boost::python::init<>())
3         .def( "savePCDFile", &PcdIOWrapper::savePCDFile )
4         .def( "loadPCDFile", &PcdIOWrapper::loadPCDFile )
5         ;
6 }
```

Nótese que por conveniencia, el *;* se coloca en una línea separada.

A.1.3 Editando el archivo de implementación (py)

Los archivos de implementación, se encuentran en el directorio *src/pclpy*, del paquete pclpy. Todos estos se encuentran organizados de acuerdo al módulo al que pertenecen, su nombre es similar al del archivo de implementación de c++, sólo que está envuelto entre *_pcl* y *py*; por ejemplo, el archivo de implementación (py) para *pcdio_wrapper.cpp* se llama *_pcl_pcdio_wrapper_py.py*

Una vez localizado el archivo de implementación (py), añada la implementación de los métodos definidos en el archivo de cabecera. Para ello, y al igual que con el archivo de implementación (cpp), es necesario la utilización de los mensajes de ROS y las funciones *to_cpp* y *from_cpp*; para la serialización y deserialización de los parámetros (de entrada y salida) hacia la *lib* de c++.

Por ejemplo, para *loadPCDFile* se realizó lo siguiente:

```
1 def loadPCDFile(self, filename):
```

```
 2          '''Load PCD file
 3          Returns the resultant PointCloud2
 4          Parameters
 5          ----------
 6          - filename: the name of the file to load
 7          '''
 8          if not isinstance( filename, str ):
 9                  rospy.ROSException( 'filename is not str' )
10
11          str_filename = self._to_cpp( String(filename) )
12          str_cloud = self._pcd_io.loadPCDFile( str_filename )
13          return self._from_cpp( str_cloud, PointCloud2 )
```

Nótese que se validan los parámetros, para evitar errores inesperados con otros tipos de datos (líneas 8~9). En la línea 11 se serializa el nombre del archivo, utilizando el método *_to_cpp* y pasándole como parámetro un mensaje de ROS (String). En la línea 13, se deserializa la cadena obtenida al ejecutar el método *loadPCDFile* de la *lib* y se regresa como valor definitivo.

Nótese que al serializar/deserializar se utilizan los mismos tipos de mensajes de ROS, tanto en el archivo de implementación c++ como en el archivo de python.

Para finalizar, construya el paquete pclpy y pruebe que el método añadido se comporta según lo estimado.

A.2 ¿Cómo agrego una nueva clase?

Para agregar una nueva clase, siga los siguientes pasos:

1. Agregue el archivo de cabecera de su clase (ej: pcdio.h), en el directorio que corresponda.

2. Agregue el archivo de implementación (c++) de su clase (ej: pcdio_wrapper.cpp), en el directorio que corresponda.

3. Agregue el archivo de implementación (py) de su clase (ej: _pcl_pcdio_wrapper_py.py), a su módulo correspondiente.

4. Edite el archivo *CMakeList.txt* para añadir su clase como una nueva biblioteca (*lib*).

A continuación se explica, de forma más detallada, cada uno de los pasos previamente descritos.

A.2.1 Agregando el archivo de cabecera

Los archivos de cabecera, se encuentran en el directorio *include*, del paquete pclpy. Todos estos se encuentran organizados de acuerdo al módulo al que pertenecen; por ejemplo, las cabeceras para los filtros se encuentran en el subdirectorio *filters*, dentro de *include*.

El nuevo archivo de clase, puede heredar de la clase de PCL o simplemente contener un atributo con

una instancia a la misma. Redefina los métodos que desee, utilizando para ello parámetros de tipo *std::string*. Si requiere definir el tipo de punto (ej: *PointXYZ*) a utilizar, puede incluir la biblioteca <*pclpy/definitions.h*> y utilizar *PointT* para ello.

Por ejemplo, para el módulo pcdio se definió lo siguiente:

```
1 class PcdIOWrapper {
2 public:
3         PcdIOWrapper();
4         std::string loadPCDFile( std::string fileName );
5         std::string savePCDFile( std::string fileName, std::string cloud );
6 };
```

Nótese que se utiliza *std::string* tanto para parámetros como para valores de retorno.

A.2.2 Agregando el archivo de implementación (c++)

Los archivos de implementación, se encuentran en el directorio *src*, del paquete pclpy. Todos estos se encuentran organizados de acuerdo al módulo al que pertenecen, su nombre es similar al del archivo de cabecera, sólo que al final lleva la palabra wrapper; por ejemplo, el archivo de implementación para *pcdio.h* se llama *pcdio_wrapper.cpp*.

El nuevo archivo de implementación de su clase, puede seguir el formato de nombre: mi_clase_wrapper.cpp, o elegir el nombre que desee. Dentro de la implementación, utilice las funciones ya definidas para la serialización/deserialización de mensajes de ROS: *to_python* y *from_python*. Contenidas en la biblioteca <*pclpy/wrapper.h*>. No olvide incluir el archivo de cabecera de su clase, utilizando para ello: <*pclpy/mi_clase.h*>.

El archivo de cabecera <*pclpy/definitions.h*> ya tiene incluidos el wrapper.h y algunas otras bibliotecas de mensajes de ROS más comunes, utilizados en pclpy. Por lo que no estaría de más revisarlo y/o inluirlo en su archivo de implementación (c++).

No olvide que para construir las bibliotecas compartidas (para su utilización en python), es necesario incluir <*boost/python.hpp*> (pclpy/definitions.h ya lo incluye).

Al final del archivo de implementación, incluya lo siguiente:

```
1 BOOST_PYTHON_MODULE( _pcl_miClase_wrapper_cpp ) {
2         boost::python::class_<miClase> ( "miClase", boost::python::init<>())
3         .def( "metodo1", &miClase::metodo1 )
4         .def( "metodo2", &miClase::metodo2 )
5         ;
6 }
```

Siendo *miClase* el nombre de su clase, método1 y método2 los nombres de los métodos públicos definidos en su clase. Para este ejemplo sólo se utilizan dos métodos, pero puede incluir más o menos según lo requiera. No olvide que al final de los métodos definidos en BOOST_PYTHON_MODULE debe existir un *;* (línea 5). El nombre contenido entre paréntesis (línea 1) es el nombre de la

biblioteca compartida generada con boost.

A.2.3 Agregando el archivo de implementación (py)

Los archivos de implementación, se encuentran en el directorio *src/pclpy*, del paquete pclpy. Todos estos se encuentran organizados de acuerdo al módulo al que pertenecen, su nombre es similar al del archivo de implementación de c++, sólo que está envuelto entre *_pcl* y *py*; por ejemplo, el archivo de implementación (py) para *pcdio_wrapper.cpp* se llama *_pcl_pcdio_wrapper_py.py*

El nuevo archivo de implementación (py) de su clase, puede seguir el formato de nombre: *_pcl_miClase_wrapper_py.py*, o elegir el nombre que desee. Dentro de la implementación, utilice las funciones ya definidas para la serialización/deserialización de mensajes de ROS: *to_cpp* y *from_cpp*. Contenidas en el módulo *pclpy._wrapper.py*.

Incluya los mensajes de ROS que requiera, además de *pclpy._wrapper* y *pclpy._pcl_miClase_wrapper_cpp*, o bien, el nombre que eligió para la biblioteca compartida (shared lib) en el paso anterior. Por ejemplo, para el módulo pcdio se utilizó lo siguiente:

```
1 import rospy
2 from sensor_msgs.msg import PointCloud2
3 from std_msgs.msg import String, Bool
4 from pclpy._wrapper import pyWrapper
5
6 from pclpy._pcl_pcdio_wrapper_cpp import PcdIOWrapper
```

PyWrapper contiene los métodos estáticos *_to_cpp* y *_from_cpp*.

A continuación, proceda a escribir la implementación de una clase en python, que incluya como atributo una instancia de su clase. Para cada parámetro, valide que sea del tipo de dato requerido (int, float, str, PointCloud2, etc.), indicando errores de tipo de ser necesario. Para una referencia rápida, puede ver la implementación de los módulos ya existentes.

Una vez añadido el archivo de implementación (py), agregue su clase al __init__.py del módulo. Por ejemplo, para el módulo filters se utilizó:

```
1 from pclpy.filters._pcl_voxel_grid_wrapper_py import VoxelGrid
2 from pclpy.filters._pcl_passthrough_wrapper_py import PassThrough
3 from pclpy.filters._pcl_outliers_wrapper_py import StatisticalOutlierRemoval
```

De esta forma, puede importar los filtros en su script, utilizando:

```
1 from pclpy.filters import VoxelGrid
2 from pclpy.filters import PassThrough
3 from pclpy.filters import StatisticalOutlierRemoval
```

A.2.4 Editando CMakeList.txt

Como último paso, en el archivo CMakeList.txt localizado en el directorio raíz (/) del paquete pclpy, añada lo siguiente:

1. El nombre de la biblioteca compartida (shared lib) de su clase, definida en el archivo de implementación de c++ (por ejemplo, *_pcl_miClase_wrapper_cpp)* a la sección catkin specific configuration:

```
##################################
## catkin specific configuration ##
##################################
catkin_package(
                INCLUDE_DIRS include
                LIBRARIES
                _pcl_miClase_wrapper_cpp
                CATKIN_DEPENDS roscpp
                #  DEPENDS system_lib
)
```

2. En la sección Build:

- Declare la biblioteca compartida, utilizando el nombre del paso anterior y el path de su archivo de implementación c++.

```
add_library(_pcl_miClase_wrapper_cpp src/ruta_a_miClase/miClase_wrapper.cpp)
```

- Especifique las biblioteca para vincular con su clase (shared lib).

```
target_link_libraries(
        _pcl_miClase_wrapper_cpp
        ${catkin_LIBRARIES}
        ${PCL_LIBRARIES}
        ${Boost_LIBRARIES}
)
```

- Especifique el nombre de las bibliotecas a añadir como *python libs*.

```
set_target_properties(
        _pcl_miClase_wrapper_cpp
        PROPERTIES
        PREFIX ""
        LIBRARY_OUTPUT_DIRECTORY
           ${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE_PYTHON_DESTINATION}
)
```

Construya el paquete pclpy y pruebe que su clase funciona según lo estimado. Para más información, puede consultar el tutorial: Using a C++ class in Python de ROS.