Building a Python Service Node

In this exercise, we will fill in the appropriate pieces of code to build a perception pipeline. The end goal will be to broadcast a transform with the pose information of the object of interest.

Prepare New Workspace:

We will create a new catkin workspace, since this exercise does not overlap with the previous PlanNScan exercises.

1. Disable automatic sourcing of your previous catkin workspace:
    1. `gedit ~/.bashrc`
    2. comment out # the last line, sourcing your `~/catkin_ws/devel/setup.bash`

source `/opt/ros/kinetic/setup.bash`

Intro (Review Existing Code)

Most of the infrastructure for a ros node has already been completed for you; the focus of this exercise is the perception algorithms/pipleline. The `CMakelists.txt` and `package.xml` are complete and an executable has been provided. You could run the executable as is, but you would get errors. At this time we will explore the source code that has been provided - browse the provided `perception_node.cpp` file. This tutorial is a follow on to training Exercise 5.1 Building a Perception Pipeline and as such the C++ code has already been set up. Open up the preception_node.cpp file and look over the filtering functions.

Create a Python node

Now that we have converted several filters to C++ functions, we are ready to call it from a Python node. If you have not done so already, install PyCharm, community edition.

1. Create a new project inside your perception_ws. In the terminal source to your src folder:

```
$ cd ~/perception_ws/src/
$ catkin_create_pkg filter_call rospy roscpp perception_msgs
```

2. Check that your package was created:

```
$ ls
```

You can open the file in Pycharm or QT (or you can use nano, emacs, vim, or sublime)

1. Edit line 2 to define the name of the project:

```
$ project(filter_call)
```

2. Edit line 7-10:

```
find_package(catkin REQUIRED COMPONENTS
perception_msgs
roscpp
rospy
)
```

Specify the packages your package depends on. We will not be using 'perception_msgs' as we will not be creating custom messages in this course. It is included for further student knowledge. If you wish for a more in depth explanation including how to implement customer messages, here is a good MIT resource on the steps taken

1. Line 20, uncomment and save.

```
catkin_python_setup()
```

2. Next, we need to update the package.xml file to allow Python communication to ROS. Open your package.xml file in QT and begin on line 3 by updating the package name

```
The lesson_perception package
```

3. Update your and to reflect roscpp, rospy, and perception_msgs.
4. Save and close the file.

Creating setup.py

The `setup.py` file makes your python module available to the entire workspace and subsequent packaged. By default, this isn't created by the `catkin_create_pkg` script.

1. In your terminal type

```
$ nano filter_call/setup.py
```

2. Copy and paste the following to the setup.py file (to paste into a terminal, Ctrl+Shift+V)

```
## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD
from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup
# fetch values from package.xml
setup_args = generate_distutils_setup(
packages=[''],
package_dir={'': 'include'},
)
setup(**setup_args)
```

change `packages = [ . . . ]`, to your list of strings of the name of the folders inside your 'include' folder. By convention, these should be the same name. The configures 'filter_call'/include/filter_call as a python module available to the whole workspace.

1. Save and close the file.

In order for this folder to be treated as a python module, the **init**.py file must exist.

1. Create one in the terminal by typing:

```
$ touch filter_call/include/filter_call/init.py
```

Where **init**.py is two underscores on either side.

1. Now we are ready to start developing the client node to call our C++ service. Create a service called 'FilterCloud.srv' as outlined in Section 2.0, updating the `CMakeLists.txt` and the `package.xml` file respectively. Copy and paste the following into the file.

```
#request
sensor_msgs/PointCloud2 input_cloud
string topic
string pcdfilename

# Removes objects outside a defined grid pattern x,y,z
byte VOXELGRID=0

# Removes the objects based on volume of space
byte PASSTHROUGH=1

# Issolate objects located along the largest flat surface (floor)
byte PLANESEGMENTATION=2

# Determine clusters based on pcd density to identify multiple objects
byte CLUSTEREXTRACTION=3
```

```
# Operation to be performed
byte operation

---
#response
sensor_msgs/PointCloud2 output_cloud
bool success
```

Be sure to include the header file associated with this service.

Publishing the Point Cloud

As iterated before, we are creating a ROS C++ node to publish the point cloud and then have a Python node listen and call filtering operations, resulting in a new, aggregated point cloud. Let's start with converting our C++ code to publish in a manner supportive to python. Remember, the C++ code is already done so all you need to do is write your python script and view the results in rviz.

Implement a Voxel Filter

Create a Boolean called `filterCallBack()` the brings in the service. This will be used by the python client to run subsequent filtering operations.

1. Above your `main:`, after your filter declarations, copy and paste the code below.

```
bool filterCallback(lesson_perception::FilterCloud::Request& request,
                    lesson_perception::FilterCloud::Response& response)
{
  pcl::PointCloud::Ptr cloud (new pcl::PointCloud);
  pcl::PointCloud::Ptr filtered_cloud (new pcl::PointCloud);
  if (request.pcdfilename.empty())
  {
    pcl::fromROSMsg(request.input_cloud, *cloud);
    ROS_INFO_STREAM("cloud size: " size());
    if (cloud->empty())
    {
      ROS_ERROR("input cloud empty");
      response.success = false;
      return false;
    }
    //response.output_cloud.header=request.input_cloud.header;
  }
  else
  {
    pcl::io::loadPCDFile(request.pcdfilename, *cloud);
    //response.output_cloud.header.frame_id="kinect_link";
  }
```

```
switch (request.operation)
{

  case lesson_perception::FilterCloud::Request::VOXELGRID :
  {
    filtered_cloud = voxelGrid(cloud, 0.01);
    break;
  }
  default :
  {
    ROS_ERROR("no point cloud found");
    return false;
  }

 }
```

Now that we have the framework for the filtering, save the file and open your filter_call.py and find:

```
# =======================
# FILL CODE: VOXEL FILTER
# =======================
```

Copy and paste the following inside the `try:` function:

```
srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = rospy.get_param('~pcdfilename', '')
req.operation = 0
# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = PointCloud2()

# ERROR HANDLING
if '.pcdfilename' == '':
    print('no file found')
    raise Exception('no file found')

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_voxel = srvp(req)
print('response received')
if res_voxel.success == False:
    res_voxel.success = True
    raise Exception('execution not valid')

# PUBLISH VOXEL FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_voxelGrid', PointCloud2, queu
pub.publish(res_voxel.output_cloud)
```

```
        print("published: voxel grid filter response")
        rate = rospy.Rate(10)
```

Uncomment and save.

```
        rospy.init_node('filter_cloud', anonymous=True)
        rospy.wait_for_service('filter_cloud')
```

Viewing Results

1. run

   ```
   roscore
   ```

2. Source a new terminal

   ```
   rosrun lesson_perception perception_node
   ```

3. Source a new terminal

   ```
   rosrun filter_call listener _pcdfilename:="/me/ros-industrial/catkin
   ```

4. Source a new terminal

   ```
   rosrun rviz rviz
   ```

5. Add a new PointCloud2

6. Change the fixed frame to kinect_link

   You may need to uncheck and recheck the PointCloud2.

Implement Pass-Through Filters

1. In the `main()`: uncomment

```
priv_nh_.param("passThrough_max", passThrough_max_, 1.0f);
priv_nh_.param("passThrough_min", passThrough_min_, -1.0f);
```

1. Update the switch to look as shown below:

```
switch (request.operation)
{

  case lesson_perception::FilterCloud::Request::VOXELGRID :
  {
```

```
      filtered_cloud = voxelGrid(cloud, 0.01);
      break;
    }

    case lesson_perception::FilterCloud::Request::PASSTHROUGH :
    {
      filtered_cloud = passThrough(cloud);
      break;
    }
    default :
    {
      ROS_ERROR("no point cloud found");
      return false;
    }

   }
```

However, because we can pass the file as two types, we need to check that there is even data so above the switch inside the `filterCallback()` function, add the following code:

```
if (request.pcdfilename.empty())
{
  pcl::fromROSMsg(request.input_cloud, *cloud);
  ROS_INFO_STREAM("cloud size: " size());
  if (cloud->empty())
  {
    ROS_ERROR("input cloud empty");
    response.success = false;
    return false;
  }
}
else
{
  pcl::io::loadPCDFile(request.pcdfilename, *cloud);
}
case lesson_perception::FilterCloud::Request::VOXELGRID :
{
  filtered_cloud = voxelGrid(cloud, 0.01);
  break;
}
```

This checks for data based on both service types.

1. Add variable declarations at the top of the program and uncomment the pcl header similar to the voxel filter.
2. Save and build

Edit the Python Code

Open the python node and find

```
# =======================
# FILL CODE: PASSTHROUGH FILTER
# =======================
```

1. Copy paste the following code. Keep care to maintain indents:

```
srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = ''
req.operation = 1
# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = res_voxel.output_cloud

# ERROR HANDLING
if '.pcdfilename' == '':
    print('no file found')
    raise Exception('no file found')

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_pass = srvp(req)
print('response received')
if res_pass.success == False:
    res_pass.success = True
    raise Exception('execution not valid')

# PUBLISH PASSTHROUGH FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_passThrough', PointCloud2, qu
pub.publish(res_pass.output_cloud)
print("published: pass through filter response")
rate = rospy.Rate(10)
```

1. Save and run from the terminal

Within Rviz, compare PointCloud2 displays based on the /kinect/depth_registered/points (original camera data) and object_cluster (latest processing step) topics. Part of the original point cloud has been “clipped” out of the latest processing result.

When you are satisfied with the pass-through filter results, press Ctrl+C to kill the node. There is no need to close or kill the other terminals/nodes.

Plane Segmentation

This method is one of the most useful for any application where the object is on

a flat surface. In order to isolate the objects on a table, you perform a plane fit to the points, which finds the points which comprise the table, and then subtract those points so that you are left with only points corresponding to the object(s) above the table. This is the most complicated PCL method we will be using and it is actually a combination of two: the RANSAC segmentation model, and the extract indices tool. An in depth example can be found on the PCL Plane Model Segmentation Tutorial; otherwise you can copy the below code snippet.

1. In the `main()`: uncomment

```
priv_nh_.param("maxIterations", maxIterations_, 200.0f);
priv_nh_.param("distThreshold", distThreshold_, 0.01f);
```

1. Update the switch to look as shown below:

```
switch (request.operation)
{

  case lesson_perception::FilterCloud::Request::VOXELGRID :
  {
    filtered_cloud = voxelGrid(cloud, 0.01);
    break;
  }

  case lesson_perception::FilterCloud::Request::PASSTHROUGH :
  {
    filtered_cloud = passThrough(cloud);
    break;
  }
  case lesson_perception::FilterCloud::Request::PLANESEGMENTATION :
  {
    filtered_cloud = planeSegmentation(cloud);
    break;
  }
  default :
  {
    ROS_ERROR("no point cloud found");
    return false;
  }

 }
```

1. Add variable declarations at the top up the program and uncomment the pcl header.
2. Save and build

Edit the Python Code

Open the python node and find

```
# ======================
# FILL CODE: PLANE SEGMENTATION
# ======================
```

1. Copy paste the following code. Keep care to maintain indents:

```
srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = ''
req.operation = 2
# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = res_pass.output_cloud

# ERROR HANDLING
if '.pcdfilename' == '':
    print('no file found')
    raise Exception('no file found')

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_seg = srvp(req)
print('response received')
if res_seg.success == False:
    res_seg.success = True
    raise Exception('execution not valid')

# PUBLISH PLANESEGMENTATION FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_planeSegmentation', PointClou
pub.publish(res_seg.output_cloud)
print("published: plane segmentation filter response")
rate = rospy.Rate(10)
```

1. Save and run from the terminal

Within Rviz, compare PointCloud2 displays based on the /kinect/depth_registered/ points (original camera data) and object_cluster (latest processing step) topics. Only points lying above the table plane remain in the latest processing result.

1. When you are done viewing the results you can go back and change the

2. When you are satisfied with the plane segmentation results, use Ctrl+

Euclidian Cluster Extraction

This method is useful for any application where there are multiple objects. This is also a complicated PCL method. An in depth example can be found on the PCL Euclidean Cluster Extration Tutorial.

1. In the `main()`: uncomment

```
priv_nh_.param("clustTol", clustTol_, 0.01f);
priv_nh_.param("clustMax", clustMax_, 10000.0);
priv_nh_.param("clustMin", clustMin_, 300.0f);
```

1. Update the switch to look as shown below:

```
switch (request.operation)
{

  case lesson_perception::FilterCloud::Request::VOXELGRID :
  {
    filtered_cloud = voxelGrid(cloud, 0.01);
    break;
  }

  case lesson_perception::FilterCloud::Request::PASSTHROUGH :
  {
    filtered_cloud = passThrough(cloud);
    break;
  }
  case lesson_perception::FilterCloud::Request::PLANESEGMENTATION :
  {
    filtered_cloud = planeSegmentation(cloud);
    break;
  }
  case lesson_perception::FilterCloud::Request::CLUSTEREXTRACTION :
  {
    std::vector::Ptr> temp =clusterExtraction(cloud);
    if (temp.size()>0)
    {
      filtered_cloud = temp[0];
    }
    //filtered_cloud = clusterExtraction(cloud)[0];
    break;
  }
  default :
  {
    ROS_ERROR("no point cloud found");
    return false;
  }

 }
```

1. Add variable declarations at the top up the program and uncomment the pcl header.
2. Save and build

Edit the Python Code

Open the python node and find

```
# =======================
# FILL CODE: CLUSTER EXTRACTION
# =======================
```

1. Copy paste the following code. Keep care to maintain indents:

```
srvp = rospy.ServiceProxy('filter_cloud', lesson_perception.srv.
req = lesson_perception.srv.FilterCloudRequest()
req.pcdfilename = ''
req.operation = 3
# FROM THE SERVICE, ASSIGN POINTS
req.input_cloud = res_seg.output_cloud

# ERROR HANDLING
if '.pcdfilename' == '':
    print('no file found')
    raise Exception('no file found')

# PACKAGE THE FILTERED POINTCLOUD2 TO BE PUBLISHED
res_cluster = srvp(req)
print('response received')
if res_cluster.success == False:
    res_cluster.success = True
    raise Exception('execution not valid')

# PUBLISH CLUSTEREXTRACTION FILTERED POINTCLOUD2
pub = rospy.Publisher('/perception_clusterExtraction', PointClou
pub.publish(res_cluster.output_cloud)
print("published: cluster extraction filter response")
rate = rospy.Rate(10)
```

1. Save and run from the terminal
    1. When you are satisfied with the cluster extraction results, use Ctrl+C to kill the node. There is no need to close or kill the other terminals/ nodes.

Future Study

The student is encouraged to convert Exercise 5.1 into callable functions and further refine the filtering operations.

Futher, for simplicity the python code was repeated for each filtering instance, the student is encouraged to create a function to handle the publishing.