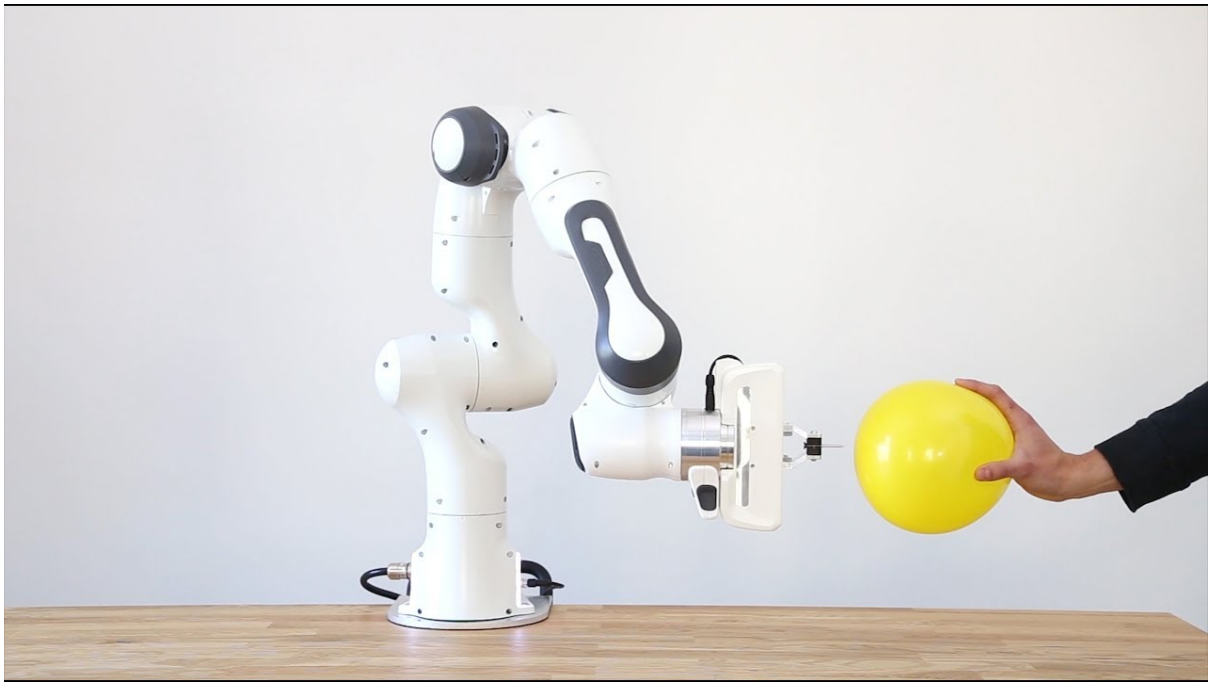
COMP0129: Robotic Sensing, Manipulation, and Interaction

Labs

Instructors: *Eddie Edwards, Francisco Vasconcelos*
TAs: *Kefeng Huang, Bowie (Heiyin) Wong*



Lab 3

Date: Week 22
Goal: Point Cloud Library (PCL) Tutorial

We will give a tutorial on the Point Cloud Library (PCL), for visual-based tasks (segmentation, object detection, filtering, etc.).

PCL and Octomap Installations

Install the required libraries for the lab, if you have not installed them during lab 1:

```
> sudo apt-get install ros-noetic-moveit-resources
> sudo apt-get install ros-noetic-octomap*
> sudo apt-get install libpcl-dev
> sudo apt-get install ros-noetic-pcl-ros
```

Update your comp0129_s24 Workspaces

Update the workspaces we have worked on (labs):

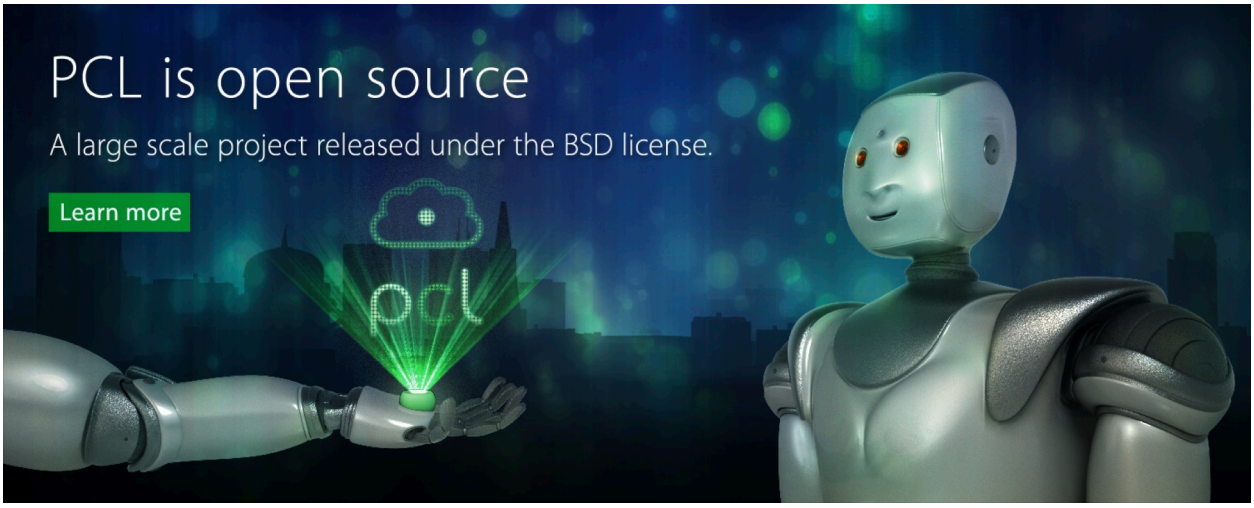
```
> cd comp0129_s24_labs
> git pull
> rosdep install -y --from-paths . --ignore-src --rosdistro noetic
> cd ..
> catkin config --extend /opt/ros/${ROS_DISTRO} --cmake-args -
DCMAKE_BUILD_TYPE=Release
> catkin build
> source devel/setup.bash
```

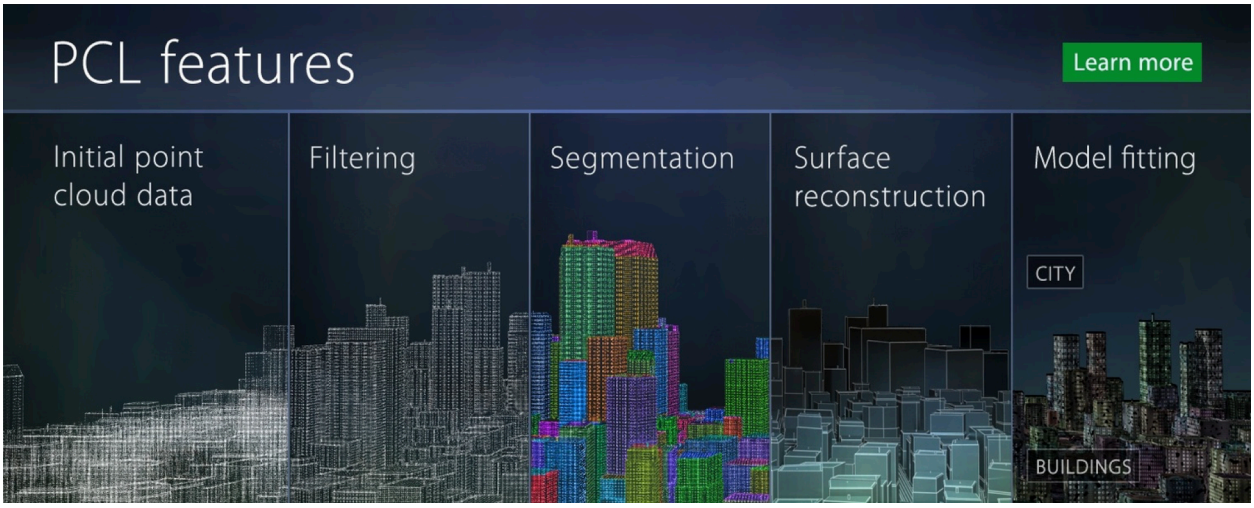
PCL Introduction

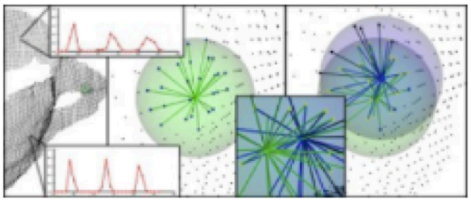
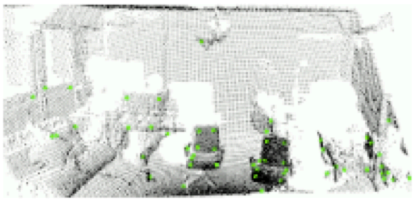
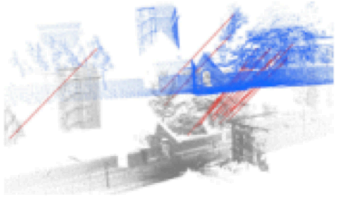
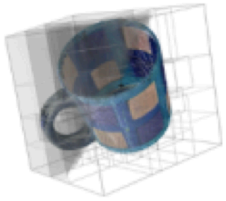
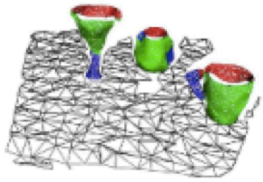
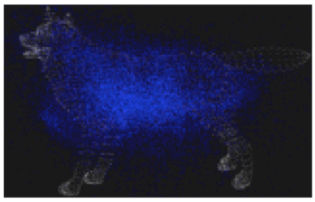
The **Point Cloud Library** is an open-source library of algorithms for point cloud processing tasks and 3D geometry processing, such as occur in three-dimensional computer vision.

The library contains algorithms for feature estimation, surface reconstruction, 3D registration, model fitting, and segmentation.

Check PCL: http://pointclouds.org



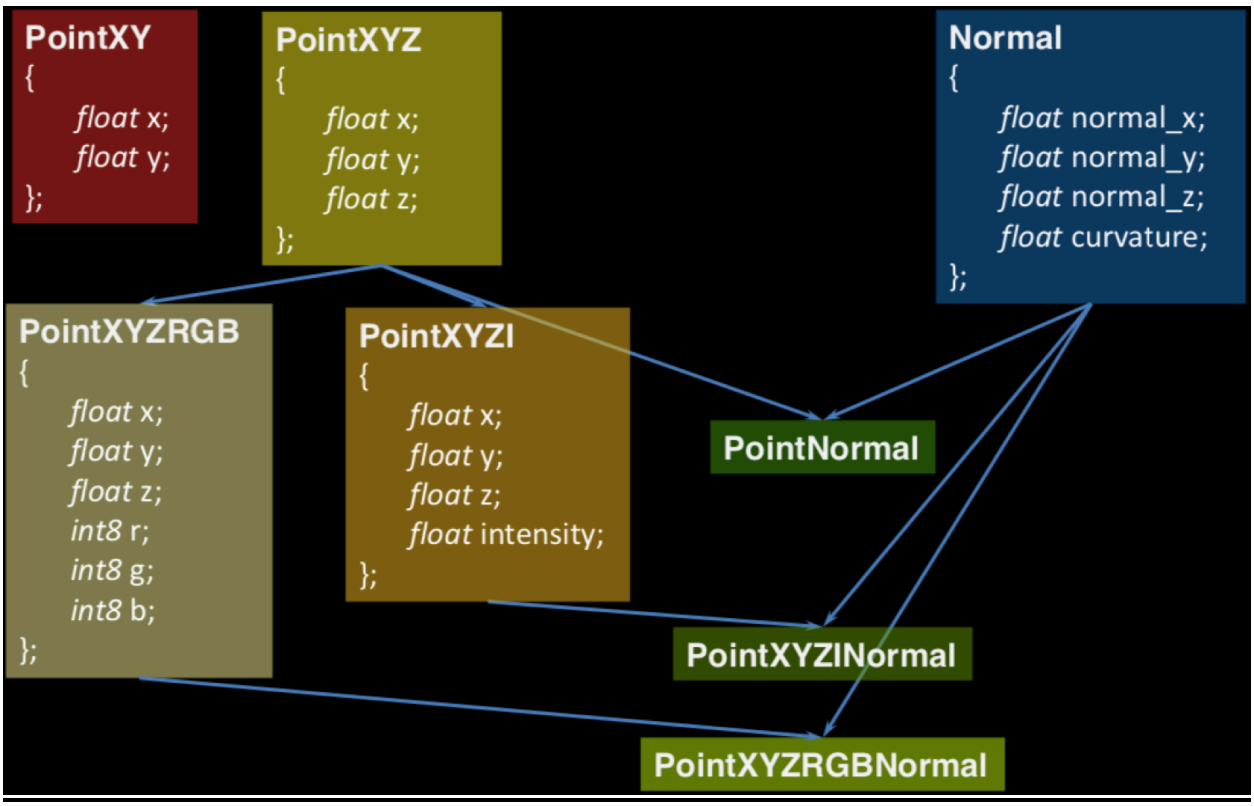


filters	features	keypoints
		
registration	**kdtree**	**octree**
		
segmentation	**sample_consensus**	**surface**
		
recognition	**io**	**visualization**
		

Point Clouds

The class **pcl::PointCloud (or pcl:PCL)** is a container of elements of **PointT** class.

To indicate which type of **PointT** you want to store in your cloud you have to use this syntax: **pcl::PointCloud<PointT>()**



In our example we use: **pcl::PointXYZRGBA** (a type of PointT) and **sensor_msgs::PointCloud2** (messages from the visual sensor that are either input to functions or can be transformed to pcl via pcl_conversions, i.e., pcl::fromROSMsg and pcl::toROSMsg).

Of course, based on the functions we could use other types, such as: pcl:PointXYZ, etc.

Important: clouds are often handled using smart pointers, e.g.:
PointCloud<PointType>::Ptr cloud_ptr;

PointCloud Recording in rosbags

ROS allows to record any type of message or topic via *bag* files. We have generated a rosbag file from the robot (so that you do not need to open gazebo and load the Panda robot with an object in it) that you download and play it back in a loop: link

Exercise 1:

Playback and visualize the point cloud (simulated from the Panda in Gazebo)

Terminal 1:
> roscore

Terminal 2:
> rosbag play -l Recorded-Can.bag

Terminal 3:
> rviz
Add the PointCloud2 and select the right topic (/r200/camera/depth_registered/points)

Notice that **sensor_msgs::PointCloud2** are coming **structured** (in a grid). Sometime this structure (which includes NaNs when there is no point in a particular pixel) makes algorithms faster, so it worth keeping it when you filter the point cloud. Check the type:

> rostopic type /r200/camera/depth_registered/points

From (http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/PointCloud2.html):

sensor_msgs/PointCloud2 Message

File: sensor_msgs/PointCloud2.msg

Raw Message Definition

```
# This message holds a collection of N-dimensional points, which may
# contain additional information such as normals, intensity, etc. The
# point data is stored as a binary blob, its layout described by the
# contents of the "fields" array.

# The point cloud data may be organized 2d (image-like) or 1d
# (unordered). Point clouds organized as 2d images may be produced by
# camera depth sensors such as stereo or time-of-flight.

# Time of sensor data acquisition, and the coordinate frame ID (for 3d
# points).
Header header

# 2D structure of the point cloud. If the cloud is unordered, height is
# 1 and width is the length of the point cloud.
uint32 height
uint32 width

# Describes the channels and their layout in the binary data blob.
PointField[] fields

bool    is_bigendian # Is this data bigendian?
uint32  point_step   # Length of a point in bytes
uint32  row_step     # Length of a row in bytes
uint8[] data         # Actual point data, size is (row_step*height)

bool is_dense        # True if there are no invalid points
```

Compact Message Definition

```
std_msgs/Header header
uint32 height
uint32 width
sensor_msgs/PointField[] fields
bool is_bigendian
uint32 point_step
uint32 row_step
uint8[] data
bool is_dense
```

PointCloud Callback

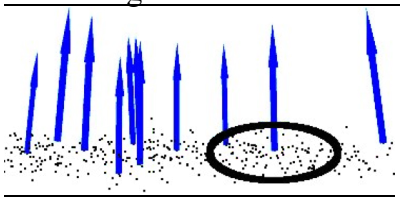
Go into your *labs* workspace:
> cd comp0129_s24_labs/src/pcl_tutorial

And investigate the **callback functions** (cloudCallBackOne). This is where we listen to messages from the visual RGB-D sensor on the robot.

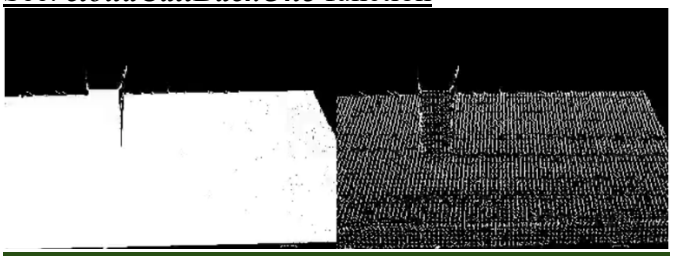
PointCloud Filtering

There are various filters that can be used.

- Estimating Surface Normals in a PointCloud: **pcl::NormalEstimation**

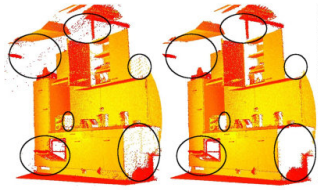


- Downsampling a PointCloud using a VoxelGrid filter: **pcl::VoxelGrid**
  See: ***cloudCallBackOne*** function



- Filtering a PointCloud using a PassThrough filter: **pcl::PassThrough**
  Simple filtering along a specified dimension, i.e. cut off values that are either inside or outside a given user range.
  See:

- Removing outliers using a StatisticalOutlierRemoval filter: **pcl::StatisticalOutlierRemoval**



Exercise 2:

Check the Pass Through and Voxel Grid filtering. Try to change the leaf size, the threshold sizes, etc.

Terminal 1:
> roscore

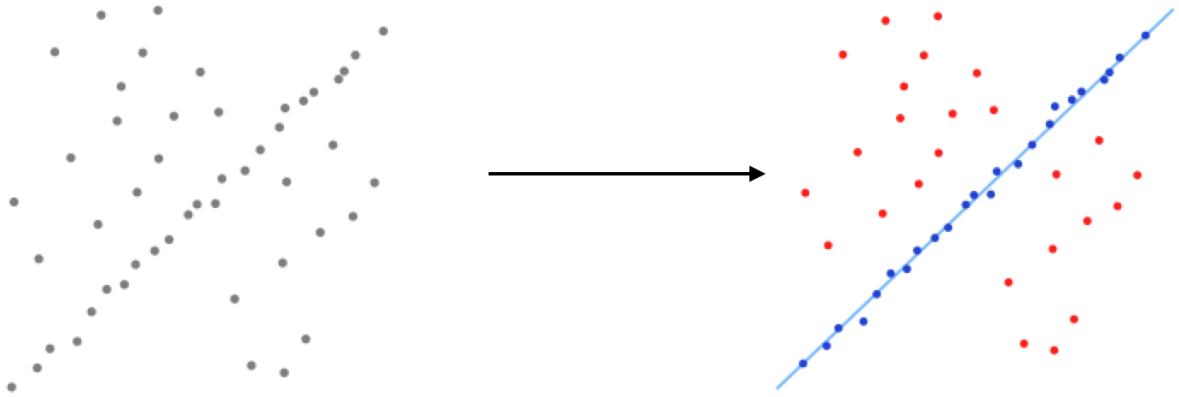
Terminal 2:
> rosbag play -l Recorded-Can.bag

Terminal 3:
> rviz
Add the PointCloud2 and select the right topic (/r200/camera/depth_registered/points)

Terminal 4:
> roslaunch pcl_tutorial pcl_tutorial.launch

PointCloud Model Fitting

Random Sample Consensus (RANSAC) model: pcl::RandomSampleConsensus.



1. **A model is fitted** to the hypothetical **inliers**, i.e., all free parameters of the model are reconstructed from the inliers.
2. All **other data** are then tested **against the fitted model** and, if a point fits well to the estimated model, also considered as a hypothetical inlier.
3. The estimated model is reasonably **good** if **sufficiently many points** have been classified as hypothetical inliers.
4. The model is **re-estimated** from all hypothetical inliers, because it has only been estimated from the initial set of hypothetical inliers.
5. Finally, the model is **evaluated** by estimating the error of the inliers relative to the model.

Exercise 3:

Check the plane and cylinder extraction (uncomment the right code inside the callback function).
Check how the published centroid cylinder point is transformed to the right frame.