PCL Tutorial:
The Point Cloud Library By Example

Jeff Delmerico

Vision and Perceptual Machines Lab
106 Davis Hall
UB North Campus

jad12@buffalo.edu

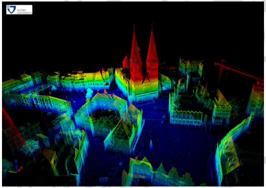
February 11, 2013

Point Clouds

Definition

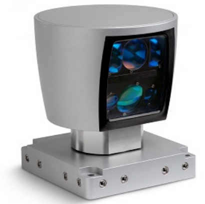
A point cloud is a data structure used to represent a collection of multi-dimensional points and is commonly used to represent three-dimensional data.

In a 3D point cloud, the points usually represent the X, Y, and Z geometric coordinates of an underlying sampled surface. When color information is present, the point cloud becomes 4D.



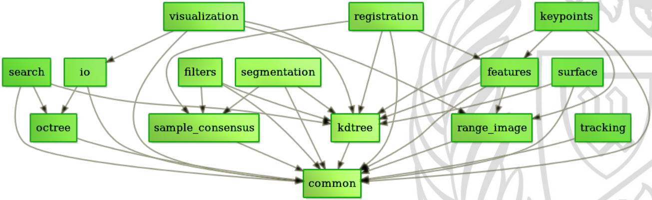
Where do point clouds come from?

- RGB-D cameras
- Stereo cameras
- 3D laser scanners
- Time-of-flight cameras
- Sythetically from software (e.g. Blender)



Point Cloud Library

- PCL is a large scale, open project for 2D/3D image and point cloud processing (in C++, w/ new python bindings).
- The PCL framework contains numerous state-of-the art algorithms including filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation.
- PCL is cross-platform, and has been successfully compiled and deployed on Linux, MacOS, Windows, and Android/iOS.
- Website: `pointclouds.org`



Getting PCL

- First, download PCL for your system from: `http://pointclouds.org/downloads/`
- If you want to try the python bindings (currently for only a subset of the full PCL functionality), go here: `http://strawlab.github.com/python-pcl/`
- PCL provides the 3D processing pipeline for ROS, so you can also get the perception_pcl stack and still use PCL standalone.
- PCL depends on Boost, Eigen, FLANN, and VTK.

Basic Structures

The basic data type in PCL is a PointCloud. A PointCloud is a templated C++ class which contains the following data fields:

- **width (int)** - secifies the width of the point cloud dataset in the number of points.
  - the total number of points in the cloud (equal with the number of elements in points) for unorganized datasets
  - the width (total number of points in a row) of an organized point cloud dataset
- **height (int)** - Specifies the height of the point cloud dataset in the number of points.
  - set to **1** for unorganized point clouds
  - the height (total number of rows) of an organized point cloud dataset
- **points (std::vector⟨PointT⟩)** - Contains the data array where all the points of type PointT are stored.

Basic Structures

- **is_dense (bool)** - Specifies if all the data in **points** is finite (true), or whether the XYZ values of certain points might contain Inf/NaN values (false).
- **sensor_origin_ (Eigen::Vector4f)** - Specifies the sensor acquisition pose (origin/translation). This member is usually optional, and not used by the majority of the algorithms in PCL.
- **sensor_orientation_ (Eigen::Quaternionf)** - Specifies the sensor acquisition pose (orientation). This member is usually optional, and not used by the majority of the algorithms in PCL.

Point Types

- **PointXYZ** - float x, y, z
- **PointXYZI** - float x, y, z, intensity
- **PointXYZRGB** - float x, y, z, rgb
- **PointXYZRGBA** - float x, y, z, uint32_t rgba
- **Normal** - float normal[3], curvature
- **PointNormal** - float x, y, z, normal[3], curvature
- **Histogram** - float histogram[N]
- And many, many, more. Plus you can define new types to suit your needs.

Building PCL Projects

PCL relies on **CMake** as a build tool. CMake just requires that you place a file called **CMakeLists.txt** somewhere on your project path.

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
project(MY_GRAND_PROJECT)
find_package(PCL 1.3 REQUIRED COMPONENTS common io)
include_directories($PCL_INCLUDE_DIRS)
link_directories($PCL_LIBRARY_DIRS)
add_definitions($PCL_DEFINITIONS)
add_executable(pcd_write_test pcd_write.cpp)
target_link_libraries(pcd_write_test $PCL_COMMON_LIBRARIES
$PCL_IO_LIBRARIES)
```

Building PCL Projects

Generating the Makefile & Building the Project

```
$ cd /PATH/TO/MY/GRAND/PROJECT
$ mkdir build
$ cd build
$ cmake ..
$ make
```

PCD File Format

A simple file format for storing multi-dimensional point data. It consists of a text header (with the fields below), followed by the data in ASCII (w/ points on separate lines) or binary (a memory copy of the *points* vector of the PC).

- VERSION - the PCD file version (usually .7)
- FIELDS - the name of each dimension/field that a point can have (e.g. FIELDS x y z)
- SIZE - the size of each dimension in bytes (e.g. a float is 4)
- TYPE - the type of each dimension as a char (**I** = signed, **U** = unsigned, **F** = float)
- COUNT - the number of elements in each dimension (e.g. x, y, or z would only have 1, but a histogram would have N)
- WIDTH - the width of the point cloud
- HEIGHT - the height of the point cloud
- VIEWPOINT - an acquisition viewpoint for the points: translation (tx ty tz) + quaternion (qw qx qy qz)
- POINTS - the total number of points in the cloud
- DATA - the data type that the point cloud data is stored in (`ascii` or `binary`)

PCD Example

```
# .PCD v.7 - Point Cloud Data file format
VERSION .7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 213
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 213
DATA ascii
0.93773 0.33763 0 4.2108e+06
0.90805 0.35641 0 4.2108e+06
0.81915 0.32 0 4.2108e+06
0.97192 0.278 0 4.2108e+06
0.944 0.29474 0 4.2108e+06
0.98111 0.24247 0 4.2108e+06
0.93655 0.26143 0 4.2108e+06
0.91631 0.27442 0 4.2108e+06
0.81921 0.29315 0 4.2108e+06
0.90701 0.24109 0 4.2108e+06
0.83239 0.23398 0 4.2108e+06
0.99185 0.2116 0 4.2108e+06
0.89264 0.21174 0 4.2108e+06
.
.
.
```

Writing PCD Files

write_pcd.cpp

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>

int
main (int argc, char** argv)
{
  pcl::PointCloud<pcl::PointXYZ> cloud;

  // Fill in the cloud data
  cloud.width    = 50;
  cloud.height   = 1;
  cloud.is_dense = false;
  cloud.points.resize (cloud.width * cloud.height);
  for (size_t i = 0; i < cloud.points.size (); ++i)
  {
    cloud.points[i].x = 1024 * rand () / (RAND_MAX + 1.0f);
    cloud.points[i].y = 1024 * rand () / (RAND_MAX + 1.0f);
    cloud.points[i].z = 1024 * rand () / (RAND_MAX + 1.0f);
  }

  pcl::io::savePCDFileASCII ("test_pcd.pcd", cloud);
  return (0);
}
```

Reading PCD Files

read_pcd.cpp

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>

int
main (int argc, char** argv)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);

  // Load the file
  if (pcl::io::loadPCDFile<pcl::PointXYZ> ("test_pcd.pcd", *cloud) == -1)
  {
    PCL_ERROR ("Couldn't read file test_pcd.pcd \n");
    return (-1);
  }

  // Do some processing on the cloud here

  return (0);
}
```

Getting Point Clouds from OpenNI

openni_grabber.cpp

```
#include <pcl/io/openni_grabber.h>
#include <pcl/visualization/cloud_viewer.h>

class SimpleOpenNIViewer
{
  public:
    SimpleOpenNIViewer () : viewer ("PCL OpenNI Viewer") {}
    void cloud_cb_ (const pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr &cloud)
    {
      if (!viewer.wasStopped())
        viewer.showCloud (cloud);
    }

    pcl::visualization::CloudViewer viewer;
```

Getting Point Clouds from OpenNI

openni_grabber.cpp

```
    void run ()
    {
      pcl::Grabber* interface = new pcl::OpenNIGrabber();
      boost::function<void (const pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr&)> f =
        boost::bind (&SimpleOpenNIViewer::cloud_cb_, this, _1);
      interface->registerCallback (f);
      interface->start ();
      while (!viewer.wasStopped())
      {
        boost::this_thread::sleep (boost::posix_time::seconds (1));
      }
      interface->stop ();
    }
};

int main ()
{
  SimpleOpenNIViewer v;
  v.run ();
  return 0;
}
```

Normal Estimation

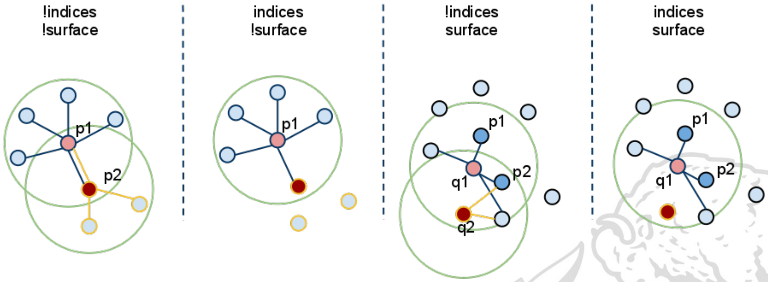
compute_normals.cpp

```
void
downsample (pcl::PointCloud<pcl::PointXYZRGB>::Ptr &points, float leaf_size,
            pcl::PointCloud<pcl::PointXYZRGB>::Ptr &downsampled_out)
{
  pcl::VoxelGrid<pcl::PointXYZRGB> vox_grid;
  vox_grid.setLeafSize (leaf_size, leaf_size, leaf_size);
  vox_grid.setInputCloud (points);
  vox_grid.filter (*downsampled_out);
}

void compute_surface_normals (pcl::PointCloud<pcl::PointXYZRGB>::Ptr &points,
              float normal_radius, pcl::PointCloud<pcl::Normal>::Ptr &normals_out)
{
  pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> norm_est;
  // Use a FLANN-based KdTree to perform neighborhood searches
  norm_est.setSearchMethod (pcl::search::KdTree<pcl::PointXYZRGB>::Ptr
                           (new pcl::search::KdTree<pcl::PointXYZRGB>));
  // Specify the local neighborhood size for computing the surface normals
  norm_est.setRadiusSearch (normal_radius);
  // Set the input points
  norm_est.setInputCloud (points);
  // Estimate the surface normals and store the result in "normals_out"
  norm_est.compute (*normals_out);
}
```

compute_normals.cpp

```
void visualize_normals (const pcl::PointCloud<pcl::PointXYZRGB>::Ptr points,
                        const pcl::PointCloud<pcl::PointXYZRGB>::Ptr normal_points,
                        const pcl::PointCloud<pcl::Normal>::Ptr normals)
{
  pcl::visualization::PCLVisualizer viz;
  viz.addPointCloud (points, "points");
  viz.addPointCloud (normal_points, "normal_points");
  viz.addPointCloudNormals<pcl::PointXYZRGB, pcl::Normal> (normal_points, normals, 1, 0.
  viz.spin ();
}

int main (int argc, char** argv)
{
  // Load data from pcd ...

  pcl::PointCloud<pcl::PointXYZRGB>::Ptr ds (new pcl::PointCloud<pcl::PointXYZRGB>);
  pcl::PointCloud<pcl::Normal>::Ptr normals (new pcl::PointCloud<pcl::Normal>);
  // Downsample the cloud
  const float voxel_grid_leaf_size = 0.01;
  downsample (cloud, voxel_grid_leaf_size, ds);
  // Compute surface normals
  const float normal_radius = 0.03;
  compute_surface_normals (ds normal_radius, normals);
  // Visualize the normals
  visualize_normals(cloud, ds, normals);
  return(0);
}
```

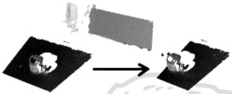
Computing 3D Features



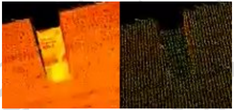
	setInputCloud = False	setInputCloud = True
setSearchSurface = False	compute on all points, using all points	compute on a subset, using all points
setSearchSurface = True	compute on all points, using a subset	compute on a subset, using a subset

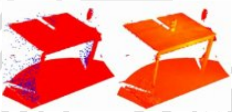
Filtering

When working with 3D data, there are many reasons for filtering your data:



- Restricting range (PassThrough)
- Downsampling (VoxelGrid)
- Outlier removal (StatisticalOutlierRemoval / RadiusOutlierRemoval)
- Selecting indices





PassThrough Filter

Filter out points outside a specified range in one dimension. (Or filter them in with setFilterLimitsNegative)

filtering.cpp

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud
           (new pcl::PointCloud<pcl::PointXYZ>);
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered
           (new pcl::PointCloud<pcl::PointXYZ>);

// PassThrough filter
pcl::PassThrough<pcl::PointXYZ> pass;
pass.setInputCloud (cloud);
pass.setFilterFieldName ("x");
pass.setFilterLimits (-0.75, 0.5);
//pass.setFilterLimitsNegative (true);
pass.filter (*cloud_filtered);
```

Downsampling to a Voxel Grid

Voxelize the cloud to a 3D grid. Each occupied voxel is approximated by the centroid of the points inside of it.

filtering.cpp

```
// Downsample to voxel grid
pcl::VoxelGrid<pcl::PointXYZ> vg;
vg.setInputCloud (cloud);
vg.setLeafSize (0.01f, 0.01f, 0.01f);
vg.filter (*cloud_filtered);
```

Statistical Outlier Removal

Filter points based on their local point densities. Remove points that are sparse relative to the mean point density of the whole cloud.

filtering.cpp

```
// Statistical Outlier Removal
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
sor.setInputCloud (cloud);
sor.setMeanK (50);
sor.setStddevMulThresh (1.0);
sor.filter (*cloud_filtered);
```

What is a keypoint?

A keypoint (also known as an "interest point") is simply a point that has been identied as a relevant in some way. A good keypoint detector will find points with the following properties:

- **Sparseness:** Typically, only a small subset of the points in the scene are keypoints.
- **Repeatiblity:** If a point was determined to be a keypoint in one point cloud, a keypoint should also be found at the corresponding location in a similar point cloud. (Such points are often called "stable".)
- **Distinctiveness:** The area surrounding each keypoint should have a unique shape or appearance that can be captured by some feature descriptor.

Why compute keypoints?

- Some features are expensive to compute, and it would be prohibitive to compute them at every point. Keypoints identify **a small number of locations** where computing feature descriptors is likely to be most effective.
- When searching for corresponding points, features computed at non-descriptive points will lead to ambiguous feature corespondences. By ignoring non-keypoints, one can **reduce error when matching points.**

Detecting 3D SIFT Keypoints

keypoints.cpp

```
void
detect_keypoints (pcl::PointCloud<pcl::PointXYZRGB>::Ptr &points, float min_scale,
                  int nr_octaves, int nr_scales_per_octave, float min_contrast,
                  pcl::PointCloud<pcl::PointWithScale>::Ptr &keypoints_out)
{
  pcl::SIFTKeypoint<pcl::PointXYZRGB, pcl::PointWithScale> sift_detect;

  // Use a FLANN-based KdTree to perform neighborhood searches
  sift_detect.setSearchMethod (pcl::search::KdTree<pcl::PointXYZRGB>::Ptr
                               (new pcl::search::KdTree<pcl::PointXYZRGB>));

  // Set the detection parameters
  sift_detect.setScales (min_scale, nr_octaves, nr_scales_per_octave);
  sift_detect.setMinimumContrast (min_contrast);

  // Set the input
  sift_detect.setInputCloud (points);

  // Detect the keypoints and store them in "keypoints_out"
  sift_detect.compute (*keypoints_out);
}
```

Computing PFH Features at Keypoints

keypoints.cpp

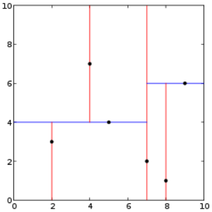
```
void
PFH_features_at_keypoints(pcl::PointCloud<pcl::PointXYZRGB>::Ptr &points,
                          pcl::PointCloud<pcl::Normal>::Ptr &normals,
                          pcl::PointCloud<pcl::PointWithScale>::Ptr &keypoints,
                          float feature_radius,
                          pcl::PointCloud<pcl::PFHSignature125>::Ptr &descriptors_out)
{
  // Create a PFHEstimation object
  pcl::PFHEstimation<pcl::PointXYZRGB, pcl::Normal, pcl::PFHSignature125> pfh_est;
  pfh_est.setSearchMethod (pcl::search::KdTree<pcl::PointXYZRGB>::Ptr
                          (new pcl::search::KdTree<pcl::PointXYZRGB>));
  // Specify the radius of the PFH feature
  pfh_est.setRadiusSearch (feature_radius);
  // Copy XYZ data for use in estimating features
  pcl::PointCloud<pcl::PointXYZRGB>::Ptr keypoints_xyzrgb
                          (new pcl::PointCloud<pcl::PointXYZRGB>);
  pcl::copyPointCloud (*keypoints, *keypoints_xyzrgb);
  // Use all of the points for analyzing the local structure of the cloud
  pfh_est.setSearchSurface (points);
  pfh_est.setInputNormals (normals);
  // But only compute features at the keypoints
  pfh_est.setInputCloud (keypoints_xyzrgb);
  // Compute the features
  pfh_est.compute (*descriptors_out);
}
```

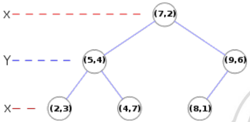
Finding Correspondences

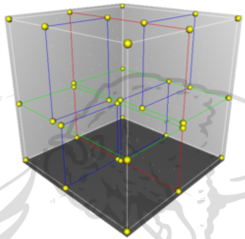
keypoints.cpp

```
void
feature_correspondences (pcl::PointCloud<pcl::PFHSignature125>::Ptr &source_descriptors,
                         pcl::PointCloud<pcl::PFHSignature125>::Ptr &target_descriptors,
                         std::vector<int> &correspondences_out,
                         std::vector<float> &correspondence_scores_out)
{
  // Resize the output vector
  correspondences_out.resize (source_descriptors->size ());
  correspondence_scores_out.resize (source_descriptors->size ());

  // Use a KdTree to search for the nearest matches in feature space
  pcl::search::KdTree<pcl::PFHSignature125> descriptor_kdtree;
  descriptor_kdtree.setInputCloud (target_descriptors);

  // Find the index of the best match for each keypoint
  const int k = 1;
  std::vector<int> k_indices (k);
  std::vector<float> k_squared_distances (k);
  for (size_t i = 0; i < source_descriptors->size (); ++i)
  {
    descriptor_kdtree.nearestKSearch (*source_descriptors, i, k,
                                      k_indices, k_squared_distances);
    correspondences_out[i] = k_indices[0];
    correspondence_scores_out[i] = k_squared_distances[0];
  }
}
```

K-d Trees





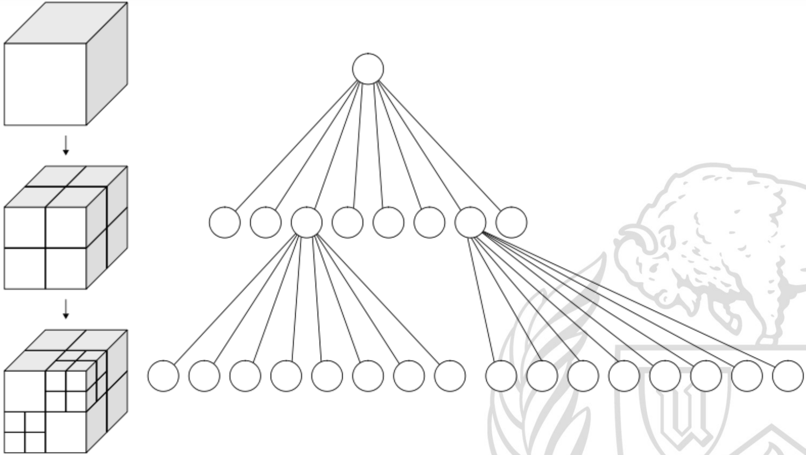


KdTree Neighbor Search

kdtree.cpp

```
#include <pcl/kdtree/kdtree_flann.h>
...
pcl::KdTreeFLANN<pcl::PointXYZ> kdtree;
kdtree.setInputCloud (cloud);
...
// K nearest neighbor search
int K = 10;
pcl::PointXYZ searchPoint;
std::vector<int> pointIdxNKNSearch(K);
std::vector<float> pointNKNSquaredDistance(K);
if ( kdtree.nearestKSearch (searchPoint, K, pointIdxNKNSearch,
                            pointNKNSquaredDistance) > 0 )
{
    ...
}

// Neighbors within radius search
std::vector<int> pointIdxRadiusSearch;
std::vector<float> pointRadiusSquaredDistance;
float radius = 256.0f * rand () / (RAND_MAX + 1.0f);
if ( kdtree.radiusSearch (searchPoint, radius, pointIdxRadiusSearch,
                          pointRadiusSquaredDistance) > 0 )
{
    ...
}
```

Octrees



octree.cpp

```
#include <pcl/octree/octree.h>
...
float resolution = 128.0f;
pcl::octree::OctreePointCloudSearch<pcl::PointXYZ> octree (resolution);
octree.setInputCloud (cloud);
octree.addPointsFromInputCloud ();
...
// Neighbors within voxel search
if (octree.voxelSearch (searchPoint, pointIdxVec))
{
    ...
}

// K nearest neighbor search
int K = 10;
if (octree.nearestKSearch (searchPoint, K,
                    pointIdxNKNSearch, pointNKNSquaredDistance) > 0)
{
    ...
}

// Neighbors within radius search
if (octree.radiusSearch (searchPoint, radius,
                    pointIdxRadiusSearch, pointRadiusSquaredDistance) > 0)
{
    ...
}
```

Sample Consensus

The Random Sample Consensus (RANSAC) algorithm assumes the data is comprised of both inliers and outliers. The distribution of inliers can be explained by a set of parameters and a model. The outlying data does not fit the model.

Plane Fitting with RANSAC

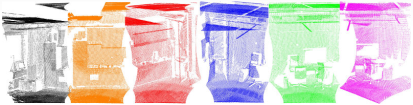
sample_consensus.cpp

```
#include <pcl/sample_consensus/ransac.h>
#include <pcl/sample_consensus/sac_model_plane.h>
#include <pcl/sample_consensus/sac_model_sphere.h>
...
std::vector<int> inliers;

// created RandomSampleConsensus object and compute the model
pcl::SampleConsensusModelPlane<pcl::PointXYZ>::Ptr
  model_p (new pcl::SampleConsensusModelPlane<pcl::PointXYZ> (cloud));
pcl::RandomSampleConsensus<pcl::PointXYZ> ransac (model_p);
ransac.setDistanceThreshold (.01);
ransac.computeModel();
ransac.getInliers(inliers);

// copies all inliers of the model computed to another PointCloud
pcl::copyPointCloud<pcl::PointXYZ>(*cloud, inliers, *final);
```

euclidean_cluster_extraction.cpp

```
#include <pcl/segmentation/extract_clusters.h>

pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>);
tree->setInputCloud (cloud_filtered);

std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance (0.02); // 2cm
ec.setMinClusterSize (100);
ec.setMaxClusterSize (25000);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices);

for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin ();
     it != cluster_indices.end (); ++it)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster
                        (new pcl::PointCloud<pcl::PointXYZ>);
  for (std::vector<int>::const_iterator pit = it->indices.begin ();
       pit != it->indices.end (); pit++)
    cloud_cluster->points.push_back (cloud_filtered->points[*pit]);

  cloud_cluster->width = cloud_cluster->points.size ();
  cloud_cluster->height = 1;
  cloud_cluster->is_dense = true;
}
```

Iterative Closest Point



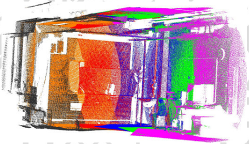
ICP iteratively revises the transformation (translation, rotation) needed to minimize the distance between the points of two raw scans.

Inputs: points from two raw scans, initial estimation of the transformation, criteria for stopping the iteration.

Output: refined transformation.

The algorithm steps are :

1. Associate points by the nearest neighbor criteria.
2. Estimate transformation parameters using a mean square cost function.
3. Transform the points using the estimated parameters.
4. Iterate (re-associate the points and so on).



Iterative Closest Point

icp.cpp

```
#include <pcl/registration/icp.h>
...
pcl::IterativeClosestPoint<pcl::PointXYZRGB, pcl::PointXYZRGB> icp;
icp.setInputCloud (cloud2);
icp.setInputTarget (cloud1);
icp.setMaximumIterations (20);
icp.setMaxCorrespondenceDistance (0.1);
Eigen::Matrix4f trafo;
icp.align (*cloud2);
(*cloud2) += *(cloud1);
...
```

Conclusion

PCL has *many* more tutorials and lots sample code here: `http://pointclouds.org/documentation/tutorials/`. And the tutorials only cover a small portion of its overall functionality.

I hope you find a use for PCL in your own projects, and you should feel free to ask me any PCL-related questions in the future (jad12@buffalo.edu).