pointcloudlibrary

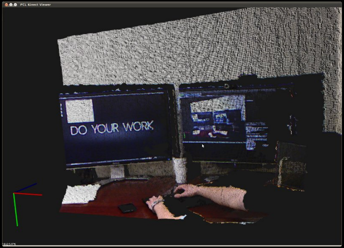
KARLSRUHE ICRA 2013

PCL Basics

May 10th, 2013

pointcloudlibrary

I/O



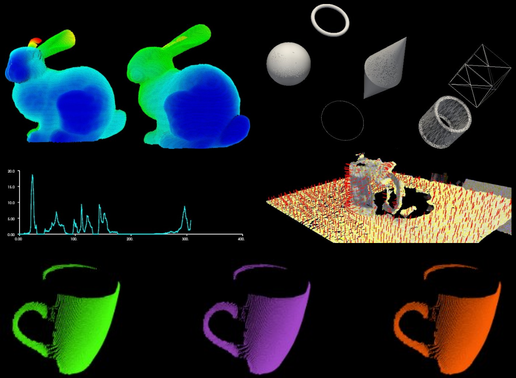
pcl::OpenNIGrabber

Documentation: `http://docs.pointclouds.org/trunk/group__io.html`

Tutorials: `http://pointclouds.org/documentation/tutorials/#i-o`

pointcloudlibrary

Visualization



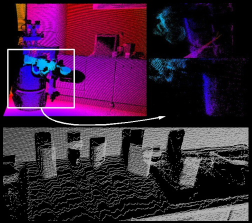
Documentation: `http://docs.pointclouds.org/trunk/group__visualization.html`

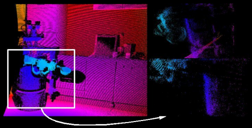
Tutorials: `http://pointclouds.org/documentation/tutorials/#visualization-tutorial`

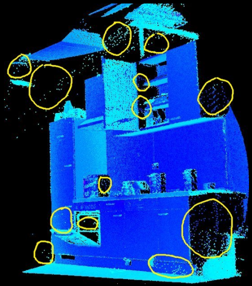
pointcloudlibrary

Filtering 1/6

- irregular density (2.5D)
- occlusions
- massive amount of data
- noise







pointcloudlibrary

Filtering 2/6

Modifying the point cloud or point attributes.

- Removing Points:
  - Conditional Removal
  - Radius/Statistical Outlier Removal
  - Color Filtering
  - Passthrough
- Downsampling:
  - Voxelgrid Filter
  - approximate Voxelgrid filtering
- Modifying Other Point Attributes:
  - Contrast
  - Bilateral Filtering

Filtering 3/6

All filters are derived from the **Filter** base class with following interface:

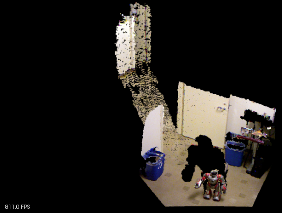
```
template<typename PointT> class Filter : public PCLBase<PointT>
{
  public:
    Filter (bool extract_removed_indices = false);
    inline IndicesConstPtr const getRemovedIndices ();
    inline void setFilterFieldName (const std::string &field_name);
    inline std::string const getFilterFieldName ();
    inline void setFilterLimits (const double &limit_min, const double &limit_max);
    inline void getFilterLimits (double &limit_min, double &limit_max);
    inline void setFilterLimitsNegative (const bool limit_negative);
    inline bool getFilterLimitsNegative ();
    inline void filter (PointCloud &output);
};
```

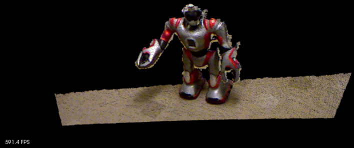
pointcloudlibrary

Filtering 4/6

Example: Passthrough Filter

```
// point cloud instance for the result
PointCloudPtr thresholded (new PointCloud);

// create passthrough filter instance
pcl::PassThrough<PointT> pass_through;

// set input cloud
pass_through.setInputCloud (input);

// set fieldname we want to filter over
pass_through.setFilterFieldName ("z");

// set range for selected field to 1.0 - 1.5 meters
pass_through.setFilterLimits (1.0, 1.5);

// do filtering
pass_through.filter (*thresholded);
```

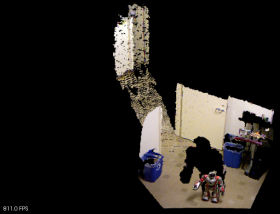




Filtering 5/6

Example: VoxelGrid Filter

```
// point cloud instance for the result
PointCloudPtr downsampled (new PointCloud);

// create passthrough filter instance
pcl::VoxelGrid<PointT> voxel_grid;

// set input cloud
voxel_grid.setInputCloud (input);

// set cell/voxel size to 0.1 meters in each dimension
voxel_grid.setLeafSize (0.1, 0.1, 0.1);

// do filtering
voxel_grid.filter (*downsampled);
```

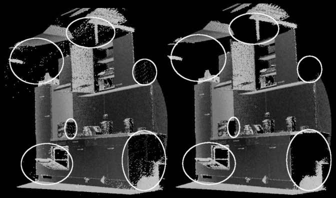


pointcloudlibrary

Filtering 6/6

Example: Radius Outlier Removal

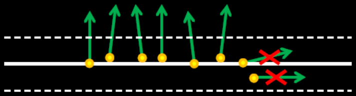
```
// point cloud instance for the result
PointCloudPtr cleaned (new PointCloud);

// create the radius outlier removal filter
pcl::RadiusOutlierRemoval<pcl::PointXYZRGB> radius_outlier_removal;

// set input cloud
radius_outlier_removal.setInputCloud (input);

// set radius for neighbor search
radius_outlier_removal.setRadiusSearch (0.05);

// set threshold for minimum required neighbors neighbors
radius_outlier_removal.setMinNeighborsInRadius (800);

// do filtering
radius_outlier_removal.filter (*cleaned);
```



pointcloudlibrary

Normal Estimation 1/9

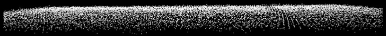
- Plane fitting can be supported by surface normals.



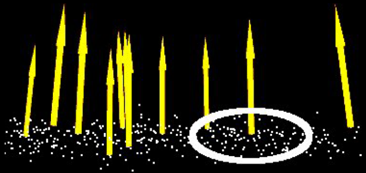
Normal Estimation 2/9

How do we compute normals in practice?

- Input: point cloud $\mathcal{P}$ of 3D points $p = (x, y, z)^T$



- Surface Normal Estimation:
  1. Select a set of points $\mathcal{Q} \subseteq \mathcal{P}$ from the neighborhood of p.
  2. Fit a local plane through $\mathcal{Q}$.
  3. Compute the normal $\vec{n}$ of the plane.



pointcloudlibrary

Normal Estimation 3/9

Available Methods

- Arbitrary Point Clouds:
  - we can not make any assumptions about structure of the point cloud
  - use FLANN-based KdTree to find approx. nearest neighbors (pcl::NormalEstimation)
- Organized Point Clouds:
  - regular grid of points (width $w \times$ height h)
  - but, not all points in the regular grid have to be valid
  - we can use:
    - FLANN-based KdTree to find approx. nearest neighbors (pcl::NormalEstimation)
    - or faster: an Integral Image based approach (pcl::IntegralImageNormalEstimation)

Normal Estimation 4/9

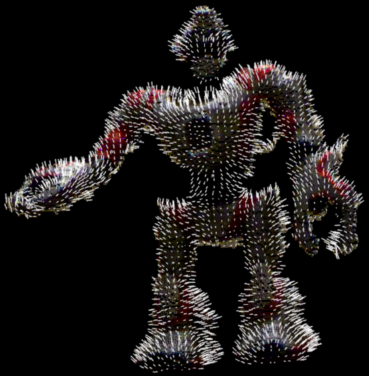
Normal Estimation using pcl::NormalEstimation

```
pcl::PointCloud<pcl::Normal>::Ptr normals_out
  (new pcl::PointCloud<pcl::Normal>);

pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> norm_est;

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

// Specify the size of the local neighborhood to use when
// computing the surface normals
norm_est.setRadiusSearch (normal_radius);

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

// Set the search surface (i.e., the points that will be used
// when search for the input points' neighbors)
norm_est.setSearchSurface (points);

// Estimate the surface normals and
// store the result in "normals_out"
norm_est.compute (*normals_out);
```

pointcloudlibrary

Normal Estimation 5/9

Normal Estimation using pcl::NormalEstimation



Normal Estimation 6/9

Normal Estimation using pcl::IntegralImageNormalEstimation

```
  pcl::PointCloud<pcl::Normal>::Ptr normals_out
    (new pcl::PointCloud<pcl::Normal>);

  pcl::IntegralImageNormalEstimation<pcl::PointXYZRGB, pcl::Normal> norm_est;

  // Specify method for normal estimation
  norm_est.setNormalEstimationMethod (ne.AVERAGE_3D_GRADIENT);

  // Specify max depth change factor
  norm_est.setMaxDepthChangeFactor(0.02f);

  // Specify smoothing area size
  norm_est.setNormalSmoothingSize(10.0f);

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

  // Estimate the surface normals and
  // store the result in "normals_out"
  norm_est.compute (*normals_out);
}
```

pointcloudlibrary

Normal Estimation 7/9

Normal Estimation using pcl::IntegralImageNormalEstimation

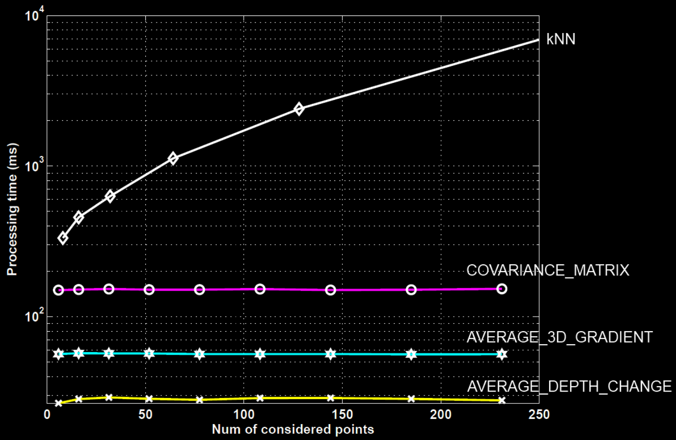
There are three ways of computing surface normals using integral images in PCL:

1. COVARIANCE_ MATRIX
   - Compute surface normal as eigenvector corresp. to smallest eigenvalue of covariance matrix
   - Needs 9 integral images
2. AVERAGE_3D_GRADIENT
   - Compute average horizontal and vertical 3D difference vectors between neighbors
   - Needs 6 integral images
3. AVERAGE_DEPTH_CHANGE
   - Compute horizontal and vertical 3D difference vectors from averaged neighbors
   - Needs 1 integral images

pointcloudlibrary

Normal Estimation 8/9

Comparison



pointcloudlibrary

Normal Estimation 9/9

So let's look how we use the normals for plane fitting:

```
// necessary includes
#include <pcl/sample_consensus/ransac.h>
#include <pcl/sample_consensus/sac_model_normal_plane.h>

// ...

// Create a shared plane model pointer directly
SampleConsensusModelNormalPlane<PointXYZ, pcl::Normal>::Ptr model
  (new SampleConsensusModelNormalPlane<PointXYZ, pcl::Normal> (input));

// Set normals
model->setInputNormals(normals);
// Set the normal angular distance weight.
model->setNormalDistanceWeight(0.5f);

// Create the RANSAC object
RandomSampleConsensus<PointXYZ> sac (model, 0.03);

// perform the segmenation step
bool result = sac.computeModel ();
```

pointcloudlibrary

Segmentation 1/6

If we know what to expect, we can (usually) efficiently segment our data:

RANSAC (Random Sample Consensus) is a randomized algorithm for robust model fitting.

Its basic operation:

1. select sample set
2. compute model
3. compute and count inliers
4. repeat until sufficiently confident

pointcloudlibrary

Segmentation 1/6

If we know what to expect, we can (usually) efficiently segment our data:

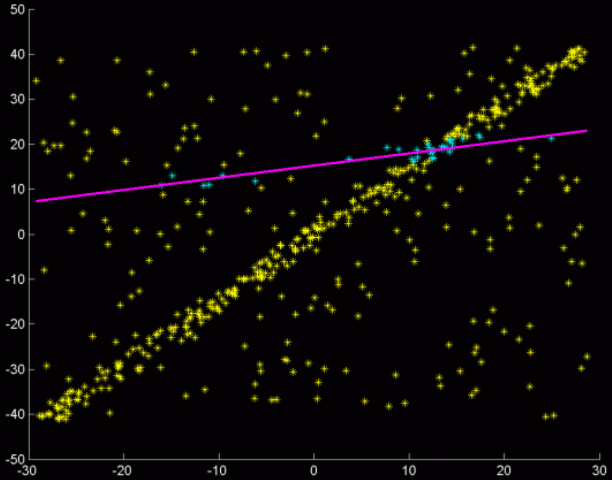
RANSAC (Random Sample Consensus) is a randomized algorithm for robust model fitting.

Its basic operation: line example

1. select sample set — 2 points
2. compute model — line equation
3. compute and count inliers — e.g. ϵ-band
4. repeat until sufficiently confident — e.g. 95%

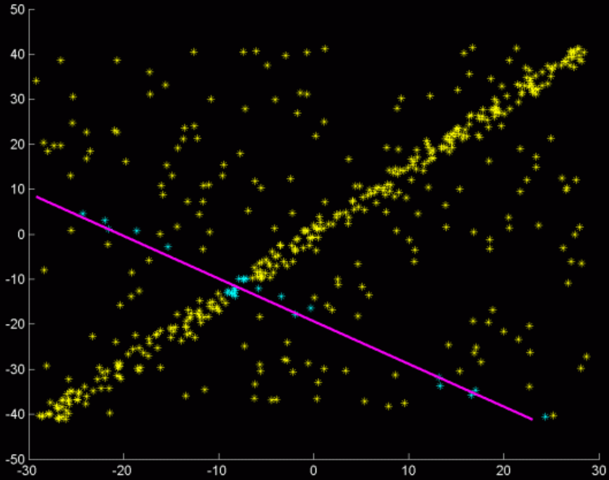
pointcloudlibrary

Segmentation 2/6



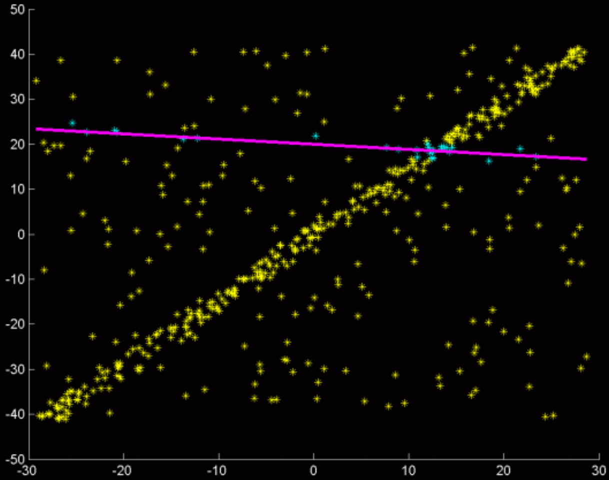
pointcloudlibrary

Segmentation 2/6



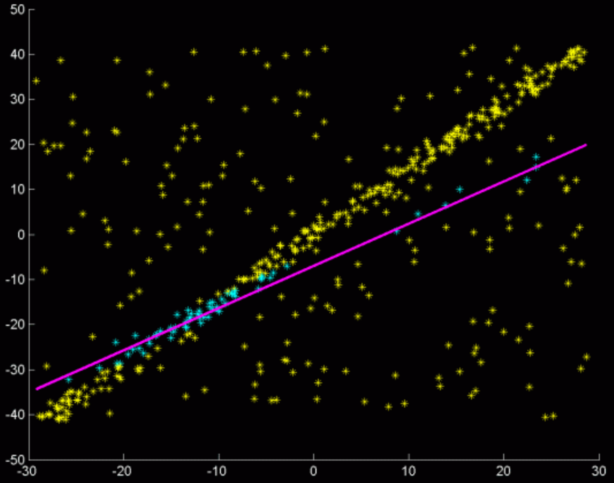
pointcloudlibrary

Segmentation 2/6



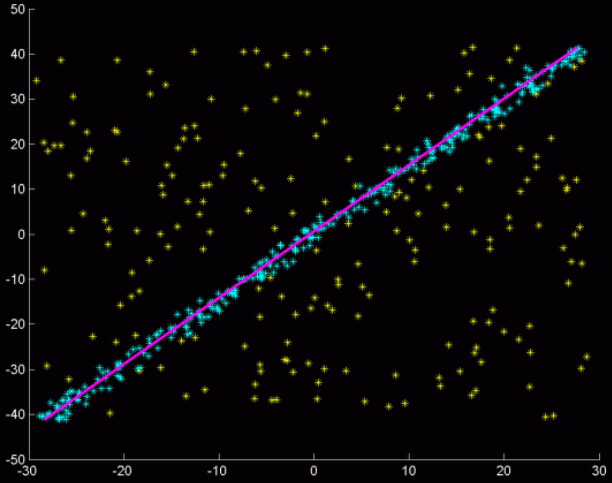
pointcloudlibrary

Segmentation 2/6



pointcloudlibrary

Segmentation 2/6



pointcloudlibrary

Segmentation 3/6

several extensions exist in PCL:

- MSAC (weighted distances instead of hard thresholds)
- MLESAC (Maximum Likelihood Estimator)
- PROSAC (Progressive Sample Consensus)

also, several model types are provided in PCL:

- Plane models (with constraints such as orientation)
- Cone
- Cylinder
- Sphere
- Line
- Circle
- ...

pointcloudlibrary

Segmentation 4/6

So let's look at some code:

```
// necessary includes
#include <pcl/sample_consensus/ransac.h>
#include <pcl/sample_consensus/sac_model_plane.h>

// ...

// Create a shared plane model pointer directly
SampleConsensusModelPlane<PointXYZ>::Ptr model
  (new SampleConsensusModelPlane<PointXYZ> (input));

// Create the RANSAC object
RandomSampleConsensus<PointXYZ> sac (model, 0.03);

// perform the segmenation step
bool result = sac.computeModel ();
```

Here, we

- create a SAC model for detecting planes,
- create a RANSAC algorithm, parameterized on $\epsilon = 3cm$,
- and compute the best model (one complete RANSAC run, not just a single iteration!)

pointcloudlibrary

Segmentation 5/6

```
// get inlier indices
boost::shared_ptr<vector<int> > inliers (new vector<int>);
sac.getInliers (*inliers);
cout << "Found␣model␣with␣" << inliers->size () << "␣inliers";

// get model coefficients
Eigen::VectorXf coeff;
sac.getModelCoefficients (coeff);
cout << ",␣plane␣normal␣is:␣" << coeff[0] << ",␣" << coeff[1] << ",␣" << coeff[2] << "." << e
```

We then

- retrieve the best set of inliers
- and the corr. plane model coefficients

pointcloudlibrary

Segmentation 6/6

Optional:

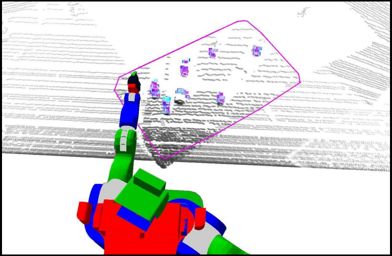
```
// perform a refitting step
Eigen::VectorXf coeff_refined;
model->optimizeModelCoefficients
  (*inliers, coeff, coeff_refined);
model->selectWithinDistance
  (coeff_refined, 0.03, *inliers);
cout << "After refitting, model contains "
                              << inliers->size () << " inliers";
cout << ", plane normal is: " << coeff_refined[0] << ", "
                              << coeff_refined[1] << ", "
                              << coeff_refined[2] << "." << endl;

// Projection
PointCloud<PointXYZ> proj_points;
model->projectPoints (*inliers, coeff_refined, proj_points);
```

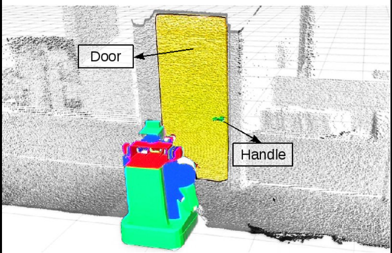
If desired, models can be refined by:

- refitting a model to the inliers (in a least squares sense)
- or projecting the inliers onto the found model

pointcloudlibrary

Clustering 1/5





Once we have a plane model, we can find

- objects standing on tables or shelves
- protruding objects such as door handles

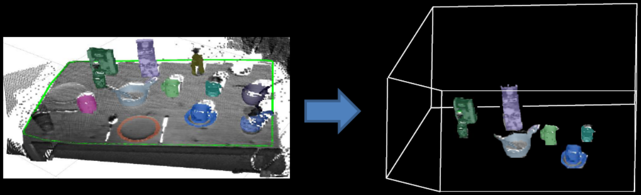
by

- computing the convex hull of the planar points
- and extruding this outline along the plane normal

pointcloudlibrary

Clustering 2/5

ExtractPolygonalPrismData is a class in PCL intended fur just this purpose.



pointcloudlibrary

Clustering 3/5

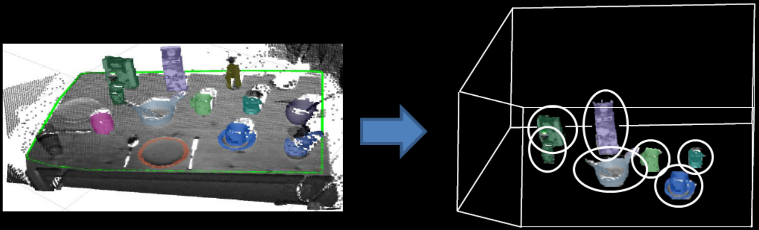
```
// Create a Convex Hull representation of the projected inliers
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_hull
  (new pcl::PointCloud<pcl::PointXYZ>);
pcl::ConvexHull<pcl::PointXYZ> chull;
chull.setInputCloud (inliers_cloud);
chull.reconstruct (*cloud_hull);

// segment those points that are in the polygonal prism
ExtractPolygonalPrismData<PointXYZ> ex;
ex.setInputCloud (outliers);
ex.setInputPlanarHull (cloud_hull);

PointIndices::Ptr output (new PointIndices);
ex.segment (*output);
```

Starting from the segmented plane,

- we compute its convex hull,
- and pass it to a ExtractPolygonalPrismData object.

pointcloudlibrary

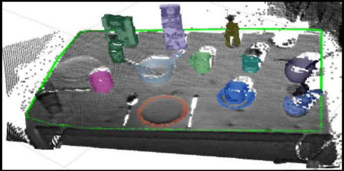
Clustering 4/5

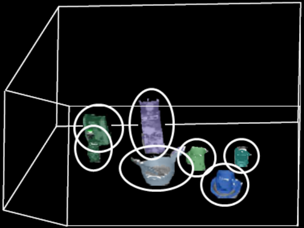


Finally, we want to segment the remaining point cloud into separate clusters. For a table plane, this gives us table top object segmentation.

pointcloudlibrary

Clustering 4/5





The basic idea is to use a region growing approach that cannot "grow" / connect two points with a high distance, therefore merging locally dense areas and splitting separate clusters.

pointcloudlibrary

Clustering 5/5

```
// Create EuclideanClusterExtraction and set parameters
pcl::EuclideanClusterExtraction<PointT> ec;
ec.setClusterTolerance (cluster_tolerance);
ec.setMinClusterSize (min_cluster_size);
ec.setMaxClusterSize (max_cluster_size);

// set input cloud and let it run
ec.setInputCloud (input);
ec.extract (cluster_indices_out);
```

Very straightforward.