1. Introduction

2. Model Based Segmentation

3. First Example

4. SACSegmentation

5. Polygonal Prism

6. Euclidean Clustering
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**
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. $\epsilon$-band
4. repeat until **sufficiently confident** — e.g. 95%
Introduction  Model Based Segmentation  First Example  SACSegmentation  Polygonal Prism  Euclidean Clustering

RANSAC
RANSAC
Introduction

Model Based Segmentation

First Example

SACSegmentation

Polygonal Prism

Euclidean Clustering

RANSAC

Nico Blodow / PCL :: Segmentation
Introduction

Model Based Segmentation

First Example

SACSegmentation

Polygonal Prism

Euclidean Clustering

RANSAC

Nico Blodow / PCL :: Segmentation
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
- ...
So let's look at some code:

```cpp
// 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 segmentation 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!)
We then

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

```cpp
// 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] << ",";
cout << coeff_refined[1] << ",";
cout << 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

Nico Blodow / PCL :: Segmentation
PCL also provides a more convenient wrapper in `SACSegmentation`:

```cpp
pcl::SACSegmentation<PointT> seg;
seg.setModelType (pcl::SACMODEL_PLANE);
seg.setMethodType (pcl::RANSAC);
seg.setDistanceThreshold (threshold);
seg.setInputCloud (input);
seg.segment (*inliers, *coefficients);
```

We will use this class in this session.
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
**ExtractPolygonalPrismData** is a class in PCL intended for just this purpose. Let’s look at the front drawer handles of a kitchen:
Starting from the segmented plane for the furniture fronts,

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

// 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);
Finally, we want to segment the remaining point cloud into separate clusters. For a table plane, this gives us **table top object segmentation**.
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.
// 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.
When we combine these segmentation algorithms consequently, we can use them to effectively and efficiently process whole rooms: