1. Model Based Segmentation

2. Plane Fitting Example

3. Normal Estimation

4. Polygonal Prism

5. 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%
RANSAC
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
- ...

Stefan Holzer and Nico Blodow / PCL :: Segmentation
So let's look at some code:

```cpp
#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] << "
    << coeff_refined[1] << "
    << coeff_refined[2] << ".
```

// 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)
- **projecting** the inliers onto the found model
Plane fitting can be supported by surface normals.
Normal Estimation

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 $Q \subseteq \mathcal{P}$ from the neighborhood of $p$.
  2. Fit a local plane through $Q$.
  3. Compute the normal $\vec{n}$ of the plane.
Normal Estimation

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

Normal Estimation using pcl::NormalEstimation

```cpp
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);
```

Stefan Holzer and Nico Blodow / PCL :: Segmentation
Normal Estimation using pcl::NormalEstimation
Normal Estimation using `pcl::IntegralImageNormalEstimation`

```cpp
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);
```
There are three ways of computing surface normals using integral images in PCL:

1. **COVARIANCE_MATRIX**
   - Compute surface normal as eigenvector corresponding 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
So let's look how we use the normals for plane fitting:

```cpp
#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 segmentation step
bool result = sac.computeModel ();
```
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.
Starting from the segmented plane,

- we compute its **convex hull**,
- and pass it to a **ExtractPolygonalPrismData** object.
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.
➤ **See RANSAC tutorial at:**
http://www.pointclouds.org/documentation/tutorials/random_sample_consensus.php

➤ **See plane segmentation tutorial at:**
http://www.pointclouds.org/documentation/tutorials/planar_segmentation.php

➤ **See normal estimation tutorials! at:**
See projecting points using parametric model tutorial at:
http://www.pointclouds.org/documentation/tutorials/project_inliers.php

See convex/concave hull tutorial at:
http://www.pointclouds.org/documentation/tutorials/hull_2d.php

See euclidean clustering tutorial at:
http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php