PCL :: Segmentation

Nico Blodow

July 1, 2011

pointcloudlibrary

Outline

pointcloudlibrary

Session files

The C++ source files and data sets related to this session can be obtained from:

`http://ias.in.tum.de/people/blodow/rss_pcl_04.bz2`

pointcloudlibrary

RANSAC

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

RANSAC

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

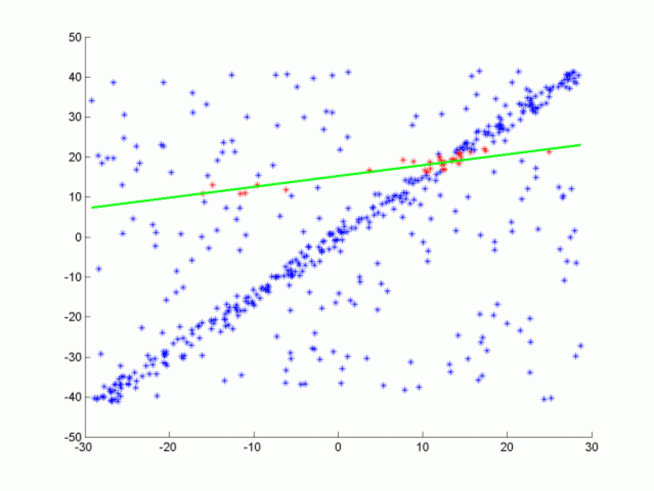
RANSAC (Random Sample Consensus) is a randomized algorithm for robust model fitting.

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%

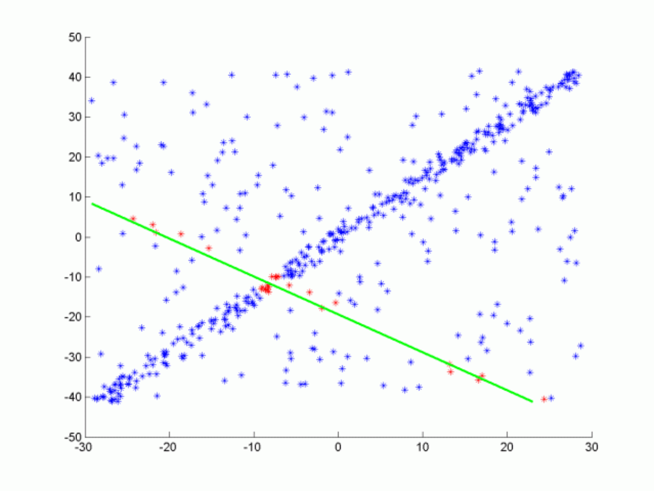
pointcloudlibrary

RANSAC



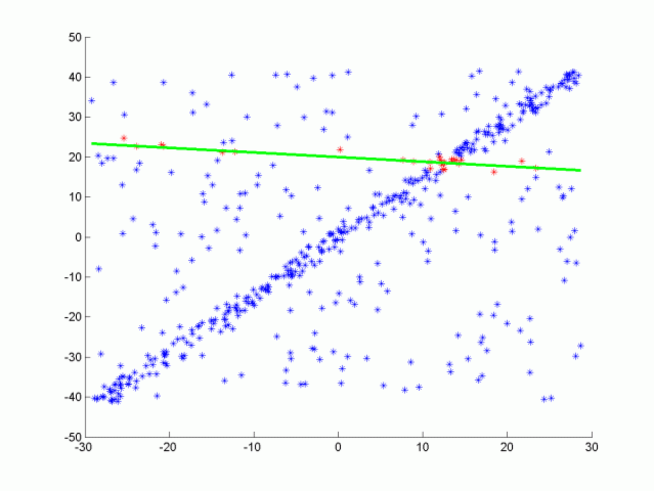
pointcloudlibrary

RANSAC



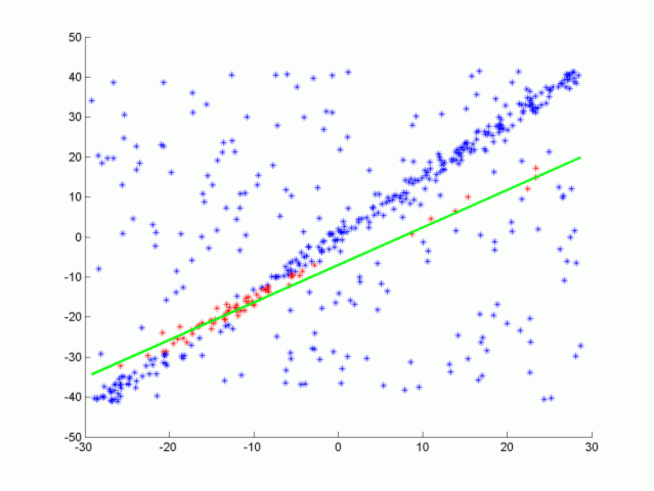
pointcloudlibrary

RANSAC



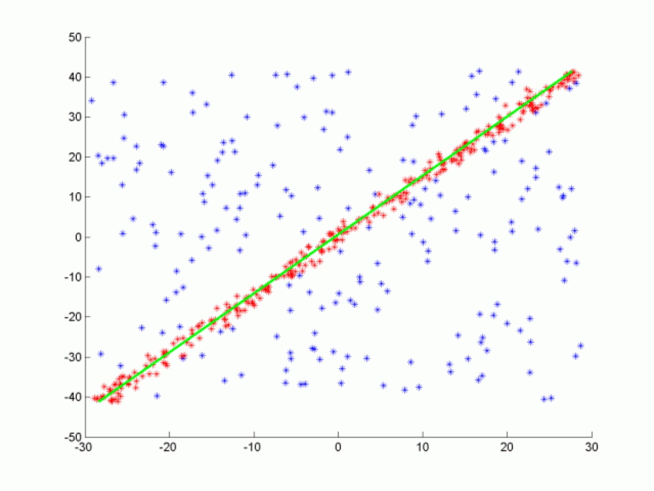
pointcloudlibrary

RANSAC



pointcloudlibrary

RANSAC



pointcloudlibrary

*SAC

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

Example 1

So let's look at some code: 04_sample_1.cpp

```
// some basic includes
#include <pcl/point_cloud.h>
#include <pcl/io/pcd_io.h>

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

// ...

// compute a planar model
void sample1 (const PointCloud<PointXYZ>::ConstPtr & input)
{
  // 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 ();

// ...
```

pointcloudlibrary

Example 1

```
// 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

Example 1

```
// 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] << ",␣"
```

We then

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

pointcloudlibrary

Example 1

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

Sample 2

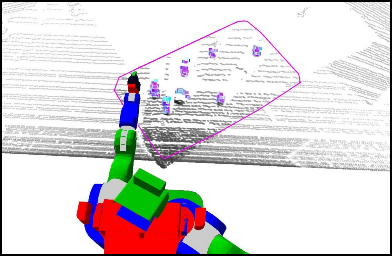
Sample 2 is just an “advanced” version of sample 1, including 3D visualization of the detected plane model:

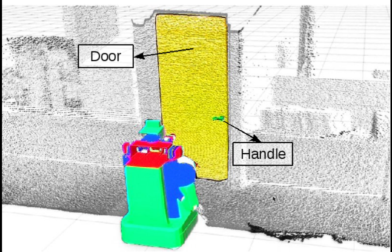
```
// Create a visualizer
PCLVisualizer vis ("RSS␣2011␣PCL␣Tutorial␣-␣04␣Segmentation");

// Create the filtering object
ExtractIndices<PointXYZ> extract;

// Extract the inliers
extract.setInputCloud (input);
extract.setIndices (inliers);
extract.setNegative (false);
PointCloud<PointXYZ>::Ptr subcloud (new PointCloud<PointXYZ>);
extract.filter (*subcloud);

// finally, add both clouds to screen
vis.addPointCloud<PointXYZ>(input, WhiteCloudHandler (input), "cloud")
vis.addPointCloud<PointXYZ>
  (subcloud, RedCloudHandler (input), "inliers");
```

As you can see in the code, this whole segmentation process can become quite tedious, so PCL provides a more convenient wrapper in SACSegmentation.

pointcloudlibrary

Polygonal Prism





Once we have a plane model, we can find

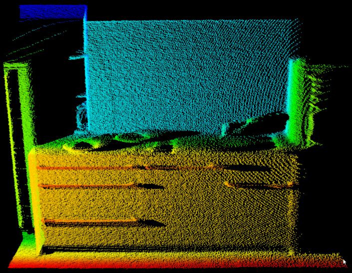
- objects standing on tables or shelves
- protruding objects such as door handles

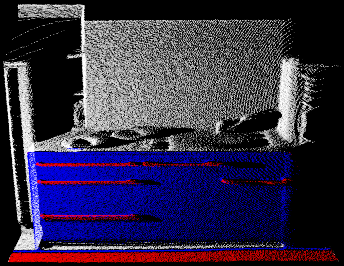
by

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

ExtractPolygonalPrismData

ExtractPolygonalPrismData is a class in PCL intended fur just this purpose.
In Sample 3, we will look at the front drawer handles of a kitchen:





pointcloudlibrary

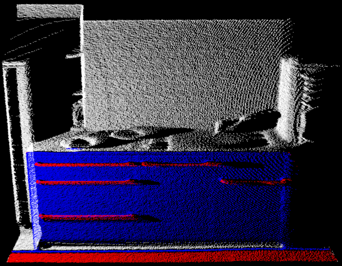
Sample 3

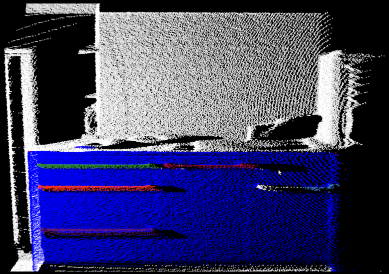
```
// Create a Convex Hull representation of the projected inliers
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_hull
  (new pcl::PointCloud<pcl::PointXYZ>);
pcl::ConvexHull<pcl::PointXYZ> chull;
chull.setInputCloud (inliers_cloud);
chull.reconstruct (*cloud_hull);

// 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 for the furniture fronts,

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

For our final example, we want to segment the point cloud containing all handles into separate handle clusters





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

Sample 4

```
// Creating the KdTree object for the search method of the extraction
KdTree<PointXYZ>::Ptr tree (new KdTreeFLANN<PointXYZ>);
tree->setInputCloud (subcloud);

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

We need

- to create a search structure for the points
- and a EuclideanClusterExtraction parameterized to the task.

pointcloudlibrary

Outlook

When we combine these segmentation algorithms consequently, we can use them to effectively and efficiently process whole rooms:

`http://www.youtube.com/watch?v=U8zhJMsao34`