Implementation of Point Cloud Library on DTU Mobotware

by

Kristian Villien – s020320

(Jan 2012)

Introduction

PCL (Point Cloud Library) is a standard functions library of 3D image analysis algorithms. The PCL has been developed by the same people who have developed the openCV library, which is mainly for 2D image analysis.

In this document, the steps needed in order to use PCL in a mobotware plugin has been described, aswell as the known problems between PCL and Mobotware. At the end of this document, a simple plugin have been described.

All 3D data have been gathered with the Kinect, and therefore some filtering that might have been necessary with a stereo camera has not been done. To get the point cloud data from the Kinect, the plugin auKinect has been updated to work with PCL. How to get the data from the kinect has been described in this document, however all other changes to the auKinect has been described in a separate document.

Using PCL in a plugin

To use PCL in a plugin the correct include files must be added and the make file must be updated *(until this has been set to be done automatically).*

As help to writing code using the PCL, an example plugin can be found at the end of this document and below links to tutorials and PCL forums can be found.

Include files

As a minimum to be able to create point clouds and use the basic PCL classes, the following files must be included:

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

For down sampling additionally `<pcl/filters/voxel_grid.h>` must be included. *(This does not currently work, see know problems).*

For basic position filtering `<pcl/filters/passthrough.h>` must be included.

For plane location and basic rotation the following libraries must be included:

```
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/common/transforms.h>
```

There are many more include files, all needed for one or more point cloud operations, it would be foolish to include them all in every project. Every included header file increase the size of the resulting object file.

When finding code examples the include files needed for the example is normally listed, remember to add them to the code.

Make files

(The following section is usable until the PCL has automatically been included in the make files by Christian Andersen)

To be able to compile a plugin the following should be added to the make file:

The line that says:

```
CPPFLAGS = -I../../include
```

Should be changed to:

```
CPPFLAGS = -I../../include -I/usr/include/pcl-1.4 -I/usr/include/eigen3
```

This will allow the compiler to find the header files needed for the basic PCL function. The link to the PCL version of the Flann library should be added here too when the problem of the conflicting flann libraries have been solved *(see known problems with PCL and Mobotware).*

Additionally `-I/usr/include/vtk-5.2` could be added in case the visualization is needed *(which does not work with the plugins)*.

The line:

```
LDFLAGS = -g3 -shared -Wl,-soname,$(libname)
```

Should be changed to

```
LDFLAGS = -g3 -shared -Wl,-soname,$(libname) -lpcl_segmentation -
lpcl_filters
```

Where `-lpcl_segmentation -lpcl_filters` refers to the precompiled libraries needed. The two libraries shown here are what is needed to compile the example in this document, in total there are 16 different libraries. The libraries needed in the make file depends on the functions used, the below list shows the command needed to add each of the 16 library files.

```
-lpcl_apps
-lpcl_common
-lpcl_features
-lpcl_filters
-lpcl_io
-lpcl_kdtree
-lpcl_keypoints
-lpcl_octree
-lpcl_range_image_border_extractor
-lpcl_range_image
-lpcl_registration
-lpcl_sample_consensus
-lpcl_search
-lpcl_segmentation
-lpcl_surface
-lpcl_visualization
```

PCL tutorial and other material

Besides inspiration from the plugin examples in this document, there are many tutorials available from the PCL homepage which can be found at http://pointclouds.org/documentation/tutorials/
Code documentation for the PCL can be found at http://docs.pointclouds.org/1.4.0/
Active PCL development and bug reports can be found at http://dev.pointclouds.org/
A general forum for PCL users can be found at http://www.pcl-users.org/
For specific problems and code help I recommend http://www.google.com/

Known problems with PCL and Mobotware

Servers

Currently it is not possible to install PCL on Jensen, Nyquist or any of the SMR. It is currently only possible to work with PCL on Bode, using “rt12” as the camera server.

Warning spam

When compiling the PCL, many, many warnings are generated. These warning does not appear to result in any problems, however it does make it more difficult to spot any error or serious warnings.

There are currently no easy way to remove these warnings.

Down sampling

It is possible to down sample the pointcloud data, this achieves a reduced computation time for the rest of the program. Data down sampling can be done with `<pcl/filters/voxel_grid.h>` part of the library, however there has been problems with implementing this in the mobotware.

The problem manifests itself by making a memory dump when returning from the function doing the calling the down sampling. It is possible to continue working on the down sampled data, it is just not possible to return from the function, this could be an indication a stack override.

A possible cause for this problem could be the Flann 3rd party library. This library is included in both the openCV and the PCL libraries, however with different versions located at different places. The library first found is the one used in openCV.

There is currently no fix for this problem.

Viewer

The view library cannot be used with the plugin, which means that viewing the 3D point clouds is not possible. Viewing 3D pointcloud is an important requirement doing plugin development, however it is not required for the actual plugin.

To allow for viewing the point clouds doing development, it is possible to save the cloud to a .pcd file, which is the point cloud library’s own file format, and view the file in a separate program. Some sample code for making such a program can be found later in this document.

For later use it is recommended to implement the point cloud viewer in the auClient software, to allow for easy development.

Simple viewer example

Since the plugin cannot use the PCL viewer library, the point clouds that one wish to view can be saved, and a simple viewer program can be build separately *(remember to put the .pcd file in the right folder).*

To save a point cloud to a file, the header file <pcl/io/pcd_io.h> must be included, and then the file can be saved with the code:

```
pcl::io::savePCDFileASCII ("test.pcd", cloud);
```
http://pointclouds.org/documentation/tutorials/writing_pcd.php#writing-pcd

Making a viewer program can be done by making the two files “viewer.cpp” and “CMakeLists.txt” and placing them in a folder. The program can then be compiled and run on ‘Bode’ by the terminal command lines below:

```
mkdir build
cd build
cmake ..
make
./viewer
```

When updateing the code (file name for example), only the “*make*” and “*./viewer*” commands need to be run.

The files content can be seen below

viewer.cpp

```
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/visualization/cloud_viewer.h>
int main (int argc, char** argv)
{
  char FileName[] = "test.pcd";
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
  if (pcl::io::loadPCDFile<pcl::PointXYZ> (FileName, *cloud) == -1) //* load the file
    { PCL_ERROR ("Couldn't read file %s \n",FileName); return (-1); }
  pcl::visualization::CloudViewer viewer ("Simple Cloud Viewer");
  viewer.showCloud (cloud);
  while (!viewer.wasStopped ());
  return (0);
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
project(viewer)
find_package(PCL 1.2 REQUIRED)
include_directories(${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
add_definitions(${PCL_DEFINITIONS})
add_executable (viewer viewer.cpp)
target_link_libraries (viewer ${PCL_LIBRARIES})
```

Plugin examples

Accessing Point Cloud Data from the Kinect

There are two ways to access the point cloud data from the auKinect plugin, either through .pcd files or through a method call. To limit the relevant kinect data, the "var kinect.MaxUsedDepth" can be used.

A .pcd file is the PCL's own file format. The kinect module can save the next point cloud as a .pcd, however since the .pcd file is in ascii format, a VGA sized cloud file can be more than 8 MB, which is very slow to read and write. Using .pcd files is suggested for development and debugging purposes only. It should be noted that the .pcd files cannot be generated if the "var kinect.useCorrectedDepth" is not set.

The point cloud data can also be directly inserted into a point cloud data object. The aukinect plugin has a method that can be called with an empty point cloud as an argument, and all the valid data points will be inserted into the cloud. An example code that calls this argument is shown below.

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>);
double pars[2], v;
bool isOK;
int n = 1;
pars[0] = 0;
pars[1] = -1;
isOK = callGlobal("kinect.GetPointCloud", "dd", NULL, pars, &v, (UDataBase**) &cloud, &n);
```

Plugin example: Kinect map

Description

This plugin take a point cloud from the kinect, finds the ground plane, rotates the point cloud, and segments the cloud into floor plane, obstacles and roof. After the three segments have been found they are mapped onto a 2D map so that the future navigation can be calculated.

Use

The module is called auPCLTest, and it is loaded into the ucamserver on rt12 by writing:
module load="aukinect.so.0"
kinect open
module load="<path>/aupcltest.so.0"

When a new map is required the ucamserver is given the command:
kinectmap

The resulting map is then placed in the imagepool number indicated in "var kinectmap.KinectMapPool", the default is 30.

Viewing the resulting map in the auclient can be done with the following commands

cam connect=rt12:24920
cam poolget img=30 all

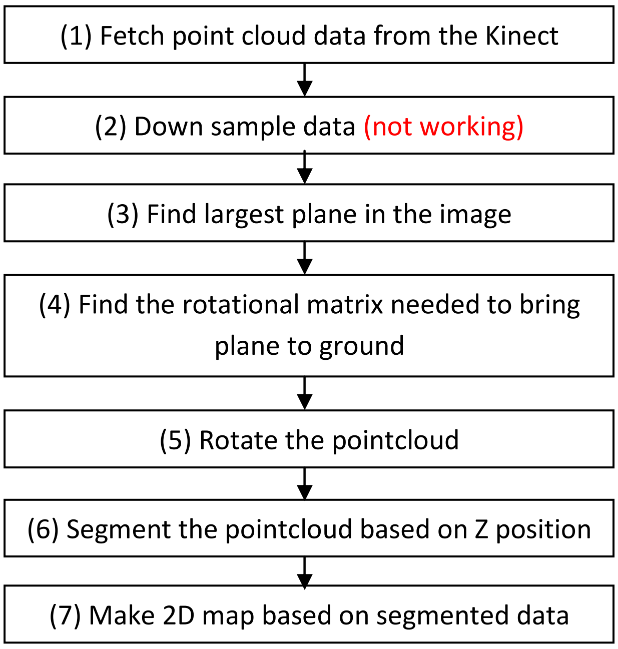
Each pixel in the resulting map image is a square 1cm x 1cm, or whatever the resolution has been set to in "var kinectmap.SquareSize" the size and relative position of the map can be set in "var kinectmap.MinMapX", "var kinectmap.MaxMapX", "var kinectmap.MinMapY" and "var kinectmap.MaxMapY".

The variable "var kinectmap.PlaneHeight" indicates how far above the estimated plane something has to be to be considered an obstacle, and "var kinectmap.RobotHeight" indicates how high above the estimated plane points have to be before the robot can pass under them.

If only obstacles are of interest, the variables "var kinectmap.ShowFloorPlane" and "var kinectmap.ShowRoof" can be cleared, this significantly increases the prcessing speed.

Flow chart

Below is the basic calculation steps after the "kinectmap" command has been given.


(1) Fetch point cloud data from the Kinect
(2) Down sample data (not working)
(3) Find largest plane in the image
(4) Find the rotational matrix needed to bring plane to ground
(5) Rotate the pointcloud
(6) Segment the pointcloud based on Z position
(7) Make 2D map based on segmented data

Results

The resulting pictures can be seen below:

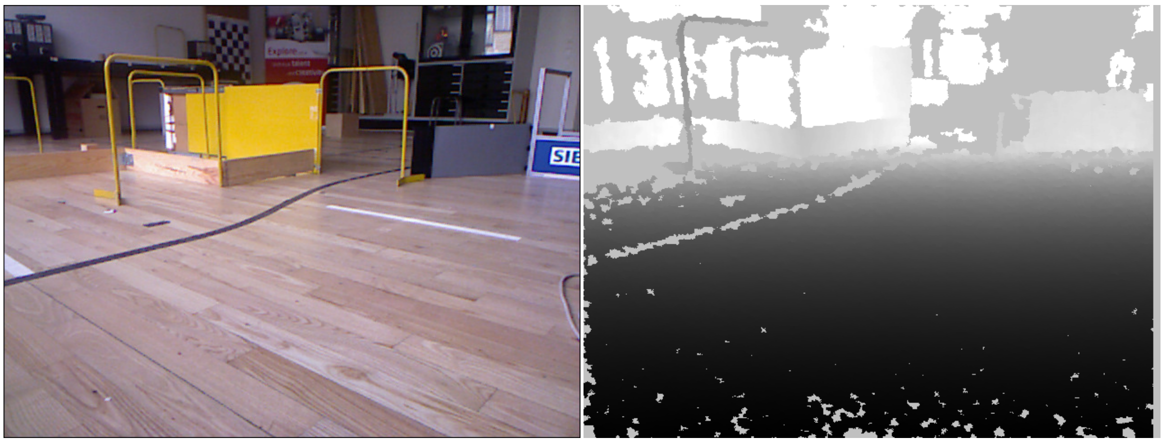


Figure 1 : RGB and corrected depth image

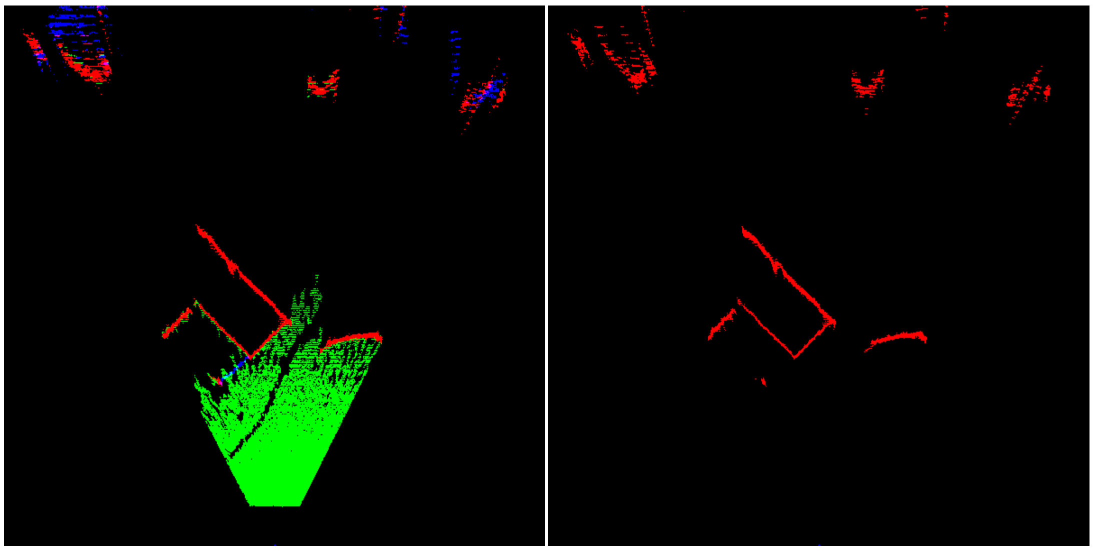


Figure 2 : Resulting maps. (left - all planes, right - only obstacles)

Timing:

	Full map	Obstacles only
Step (1)	35.2 [ms]	34.9 [ms]
Step (2)	Not working	Not working
Step (3) & Step (4)	88.7 [ms]	87.4 [ms]
Step (5)	1019.4 [ms]	228.9 [ms]
Step (6)	40.0 [ms]	4.4 [ms]
Step (7)	38.2 [ms]	7.0 [ms]
total	1221.6 [ms]	362.6 [ms]

As can be seen from the timing results, significant time can be saved by ignoring the floorplane, mainly on the point rotation which is the most time consuming. A similar time save might be possible by downsampling the 3D data, however currently this does not work correctly *(see Known problems with PCL and mobotware).*

Code for steps

Explanation for the PCL code needed for each step will be given in this section. The actual source code contains a great deal more code, since several conditions must be checked and interpreted, however this section contains the basic code for using the PCL objects.

Step 0: Creating point clouds

The first step is to create a pointcloud object. There are several different kinds of point cloud objects, not to be confused with point objects contained within the cloud, which there can also be several kinds of.

In most tutorials the point cloud objects used are `pcl::PointCloud<pcl::PointXYZ>` and `sensor_msgs::PointCloud2::Ptr`, In the example in this document only the first one is used. The object in the <> is the point object, indicating that the work is being done with XYZ only, it is however also possible to define it as `pcl::PointCloud<pcl::PointXYZRGB>` if color is also needed. If needed, it is also possible to create custom point objects and use them with the point cloud library, for details see http://pointclouds.org/documentation/tutorials/adding_custom_ptype.php#adding-custom-ptype

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

Step 1: Fetch point cloud data from the Kinect

A function has been made for this, the function calls a method in the “kinect” plugin, supplying the pointer (to a pointer) to an empty pointcloud object. The returning object is filled with pointcloud data from the last kinect image.

```
void UFuncPCLTest::GetKinectPointCloudData(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud )
{
 const int MPC = 2; // parameter count
 double pars[MPC], v;
 bool isOK;
 int n = 1;
 pars[0] = 0;
 pars[1] = -1;
 isOK = callGlobal("kinect.GetPointCloud", "dd", NULL, pars, &v, (UDataBase**) &cloud, &n);
}
```

Step 2: Down sample data (not working in plugin, but OK in stand-alone)

Down sampling works by creating a grid in the 3D image and allowing only one point to be within each grid area. To do this, a VoxelGrid object is made and set up, and the input cloud is then filtered, and the result is outputted in the cloud supplied with the filter function. In the example below, the plugin plane height is used as base for the grid height, since the grid will be transferred to a 2D map at the end.

```
 pcl::VoxelGrid<pcl::PointXYZ> sor;
 sor.setInputCloud (cloud);
 sor.setLeafSize (varPlaneHeight->getDouble()/2, varPlaneHeight->getDouble()/2,
varPlaneHeight->getDouble()/2);
 sor.filter (*cloud_filtered);
```

Step 3: Find largest plane in the image

Finding a plane is done using a SACSegmentation object, which is set up to find a plane using the Ransac method. The resulting plane parameters are placed in a ModelCoefficients object and the points included in the cloud is placed in a PointIndices object.

```
 pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
 pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
 pcl::SACSegmentation<pcl::PointXYZ> seg;
 seg.setOptimizeCoefficients (true);
 seg.setModelType (pcl::SACMODEL_PLANE);
 seg.setMethodType (pcl::SAC_RANSAC);
 seg.setDistanceThreshold (varPlaneHeight->getDouble());
 seg.setInputCloud (cloud->makeShared ());
 seg.segment (*inliers, *coefficients);
```

If there is a wish to isolate the points found within the plane, it can be done with the ExtractIndices object. The setNegative object function controls if the resulting cloud is only the plane points or if the resulting cloud are all the points not found to be in the plane.

```
 pcl::ExtractIndices<pcl::PointXYZ> extract;
 extract.setInputCloud (cloud);
 extract.setIndices (inliers);
 extract.setNegative (true);
 extract.filter (*cloud_filtered);
```

Step 4: Find the rotational matrix needed to bring plane to ground

Finding the needed rotational matrix is not done with the PCL, and will therefore not be covered here. The basics is that it finds the transform needed to bring the Y part of the plane to zero and then the X part of the resulting plane to zero.

Step 5: Rotate the point cloud

Rotating an entire point cloud is done with a global function, no object is needed. The function takes three parameters, the input cloud, the output cloud and the requested rotation.

```
 Eigen::Matrix4f RR;
 RR << 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1;
 pcl::transformPointCloud(*cloud, *cloud_Rot, RR);
```

Step 6: Segment the point cloud based on Z position

When the cloud needs to be filtered using only one axis, the PassThrough filter object is be the best choice. The object is set up with the requested axis and the limits and then the input cloud is filtered into a new cloud. Once again the object has a function to indicate if it is a band-pass filter or band-stop filter, called setFilterLimitsNegative.

The same object can be used for several filtering, so only one object is needed to split the cloud into the three parts.

```
pcl::PassThrough<pcl::PointXYZ> pass;
pass.setInputCloud (cloud_Rot);
pass.setFilterFieldName ("z");
pass.setFilterLimitsNegative (false);

pass.setFilterLimits (-1.0, varPlaneHeight->getDouble());
pass.filter (*cloud_floor);

pass.setFilterLimits (varPlaneHeight->getDouble(), varRobotHeight->getDouble());
pass.filter (*cloud_obst);

pass.setFilterLimits (varRobotHeight->getDouble(), 5);
pass.filter (*cloud_roof);
```

Step 7: Make 2D map based on segmented data

To plot the pixels into a new image, it is nesesary to manually run through all the cloud point and The loop

```
for (int i = 0; i < cloud_obst ->points.size(); i++)
{
  Mx = (int) round(cloud_obst ->points[i].x/resolution-minX/resolution);
  My = (int) round(cloud_obst ->points[i].y/resolution-minY/resolution);
  if(Mx >= 0 && Mx < w && My >= 0 && My < h)
  {
    pix2 =pix+((h-1-My)*w+Mx);
    pix2->p1 = 0xFF; // red
    pix2->p2 = pix2->p2/2; //green
  }
}
```

Source code

The source code has been supplied with the project.