

Copyright Undertaking

This work is protected by copyright, with all rights reserved.

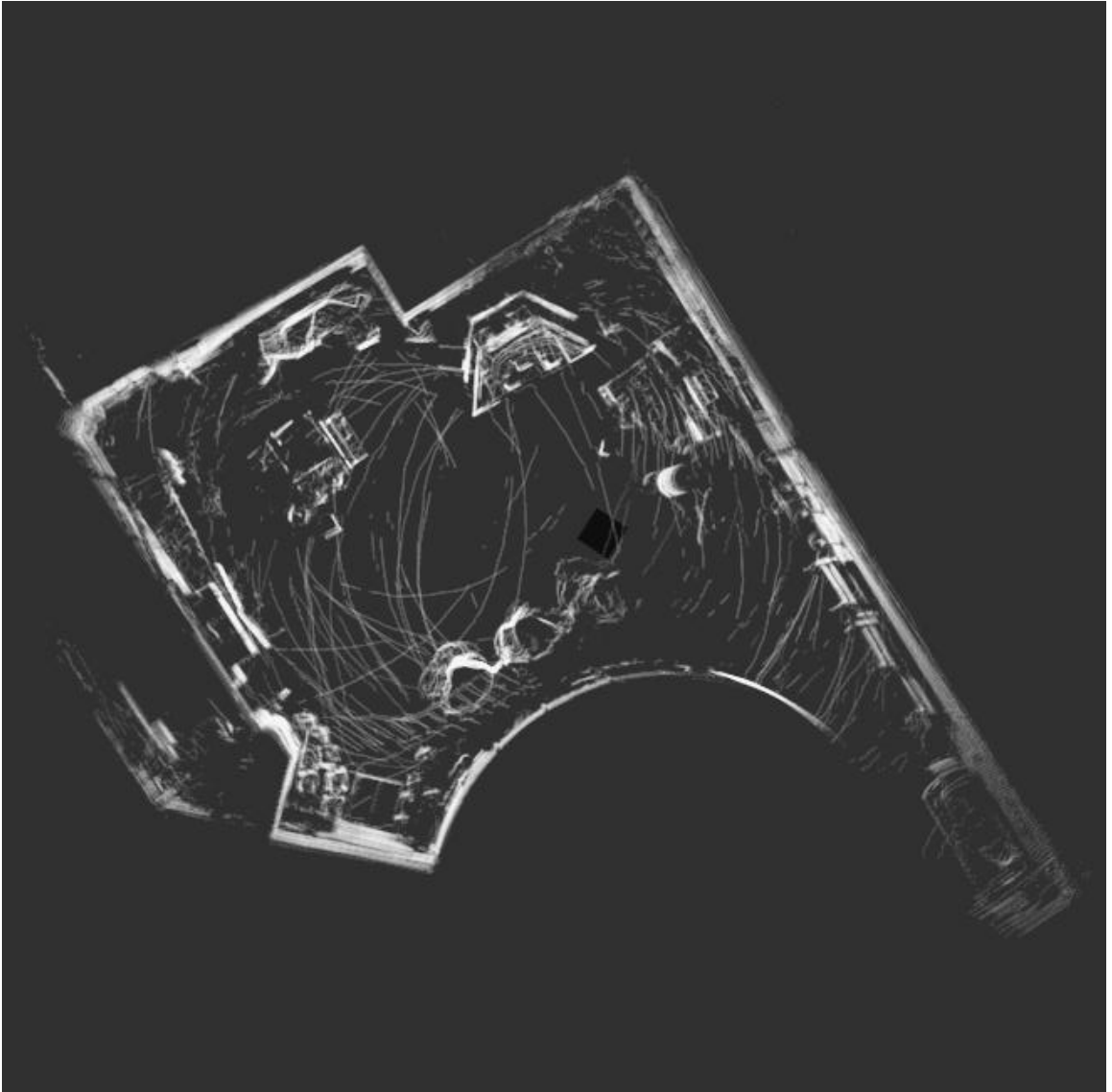
By reading and using the work, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the work.
2. The reader will use the work for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

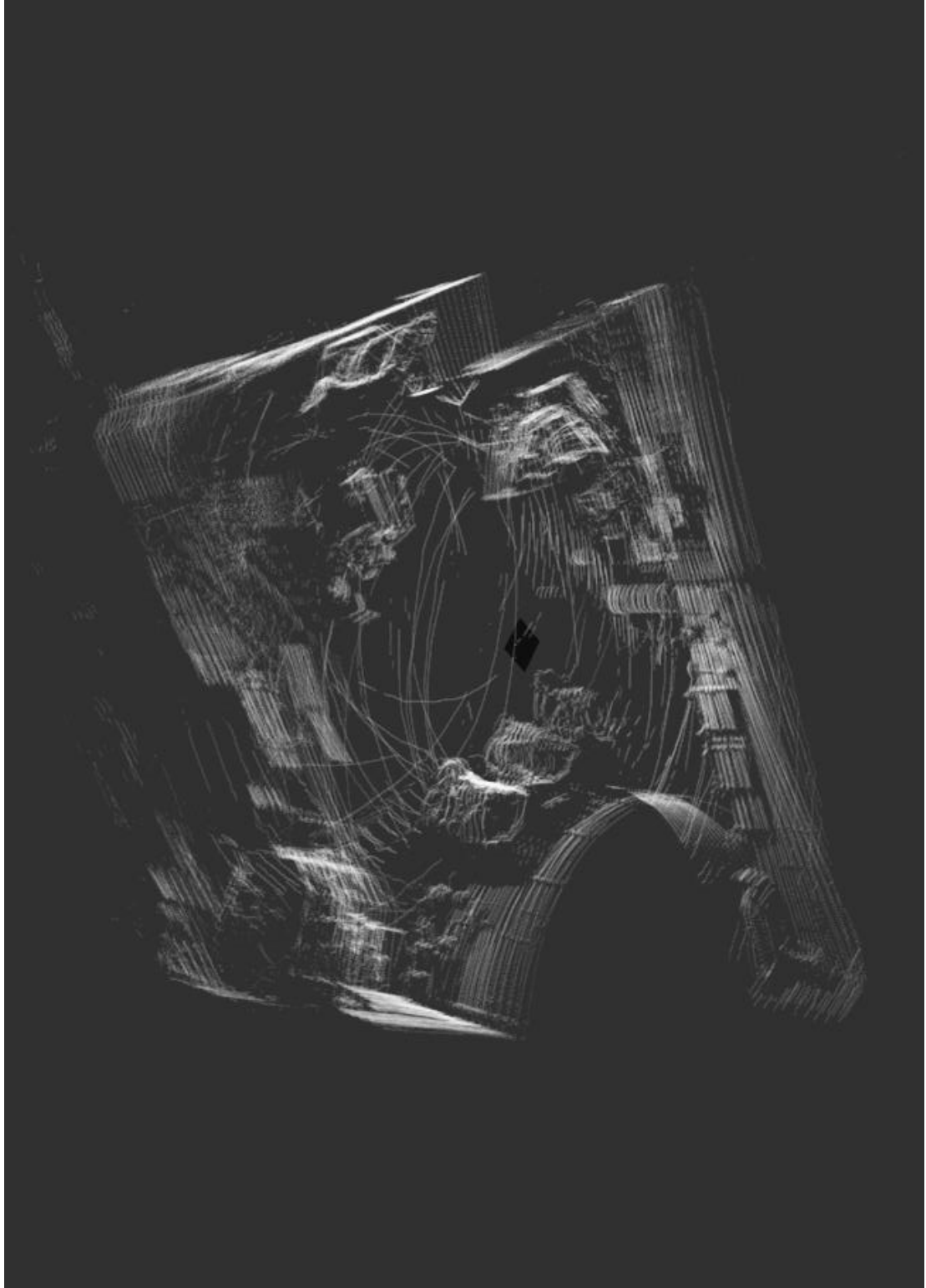
IMPORTANT

If you have reasons to believe that any materials in this work are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

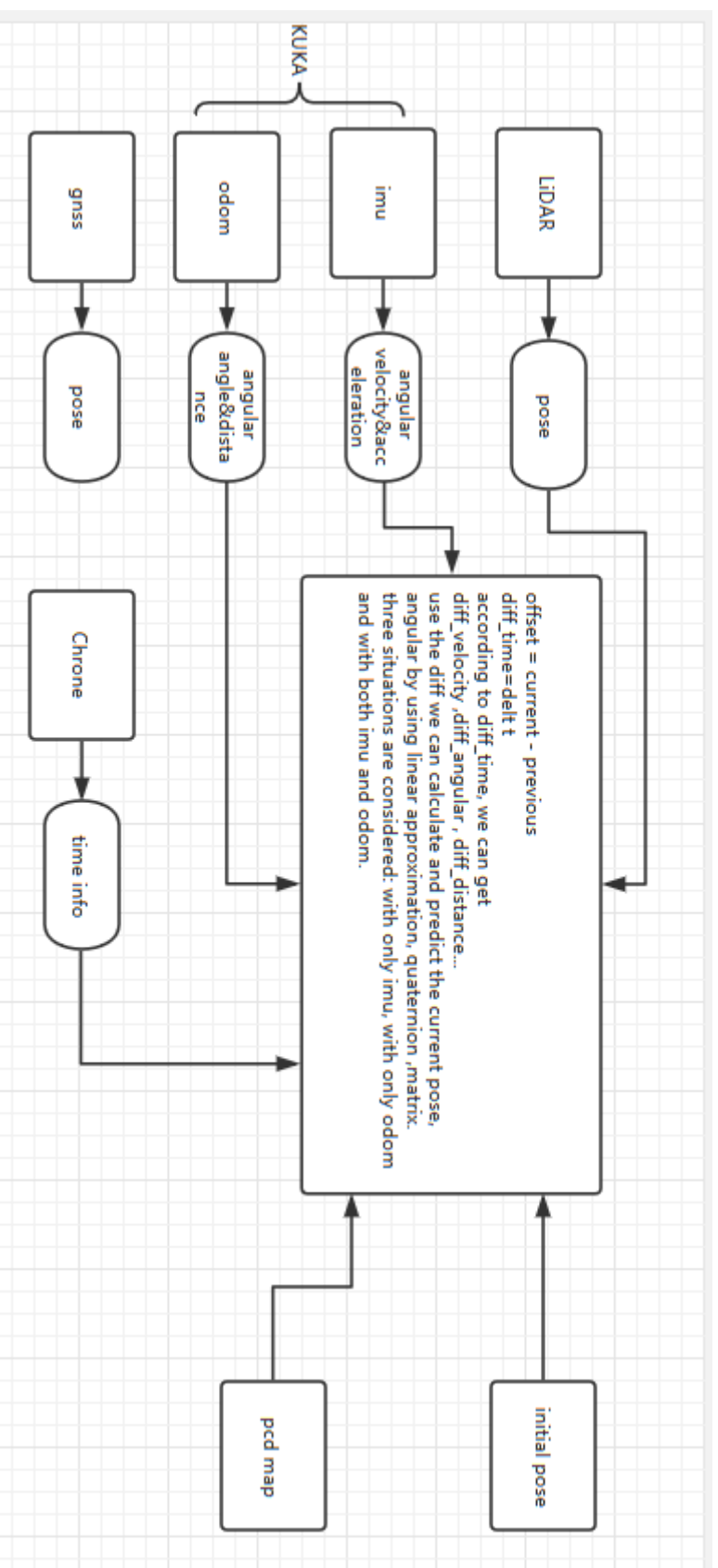
Appendix

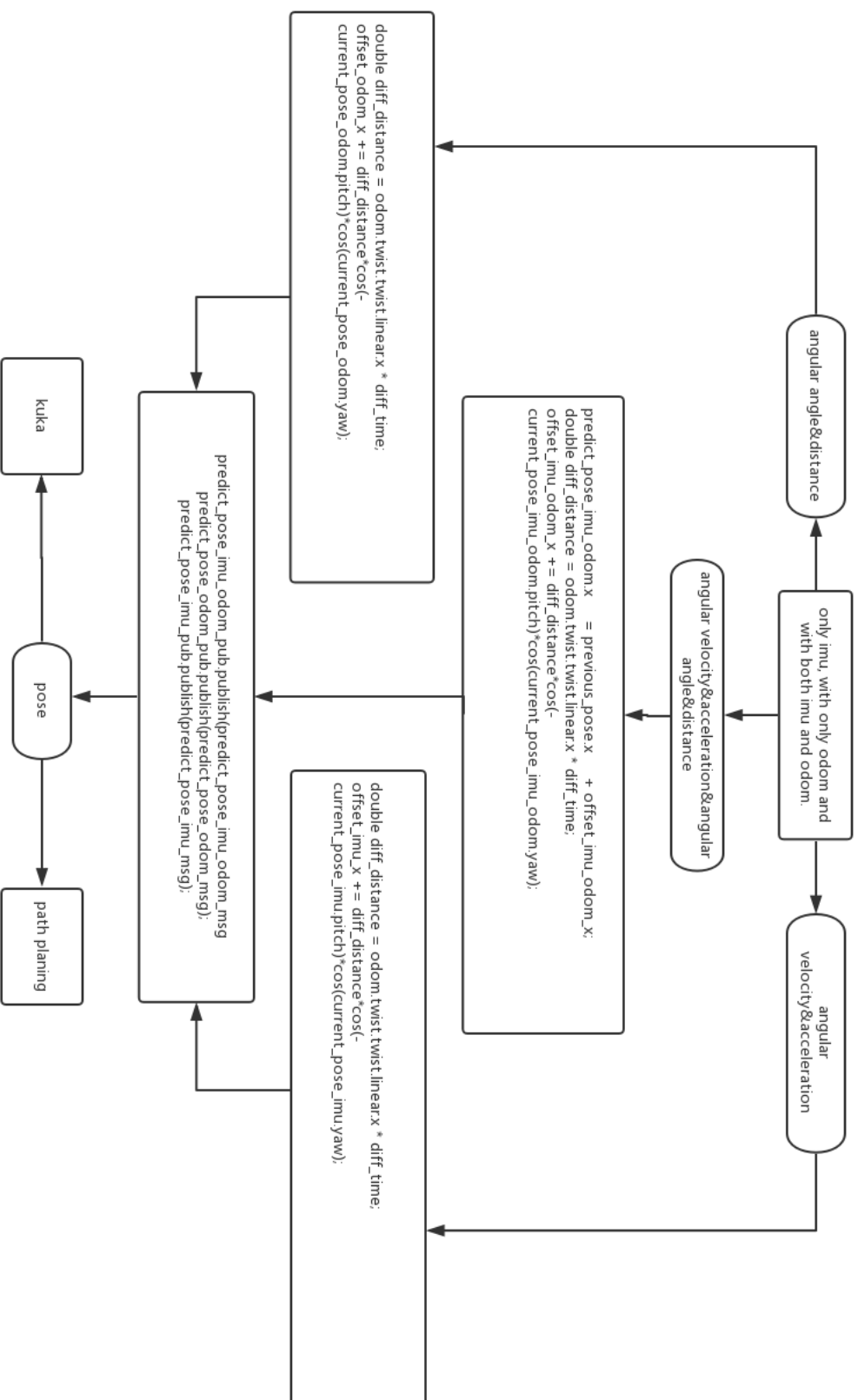


(FJ005 pcd map)

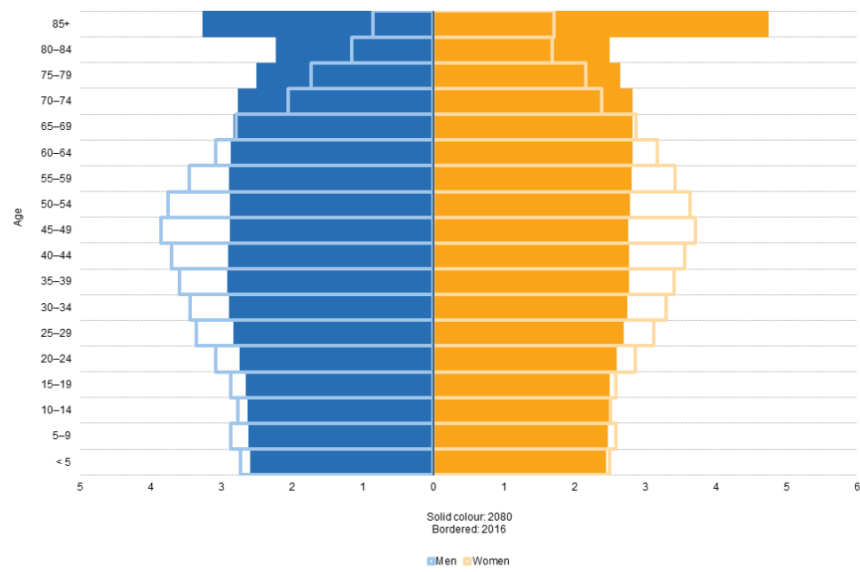


(FJ005 pcd map)





Population structure of European Union



(Population Structure and Ageing ,2017)

Ndt_mapping

```
1.  /*
2.  *   Copyright (c) 2015, Nagoya University
3.  *   All rights reserved.
4.  *
5.  *   Redistribution and use in source and binary forms, with or without
6.  *   modification, are permitted provided that the following conditions are met:
7.  *
8.  *   * Redistributions of source code must retain the above copyright notice, this
9.  *     list of conditions and the following disclaimer.
10. *
11. *   * Redistributions in binary form must reproduce the above copyright notice,
12. *     this list of conditions and the following disclaimer in the documentation
13. *     and/or other materials provided with the distribution.
14. *
15. *   * Neither the name of Autoware nor the names of its
16. *     contributors may be used to endorse or promote products derived from
17. *     this software without specific prior written permission.
18. *
19. *   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
20. *   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
21. *   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
22. *   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
23. *   FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
24. *   DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
25. *   SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
26. *   CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
27. *   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
28. *   OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
29. */
30.
31. /*
32. Localization and mapping program using Normal Distributions Transform
33.
34. Yuki KITSUKAWA
35. */
36.
37. #define OUTPUT // If you want to output "position_log.txt", "#define OUTPUT".
```

```
38.
39. #include <iostream>
40. #include <sstream>
41. #include <fstream>
42. #include <string>
43.
44. #include <ros/ros.h>
45. #include <std_msgs/Bool.h>
46. #include <std_msgs/Float32.h>
47. #include <nav_msgs/Odometry.h>
48. #include <sensor_msgs/Imu.h>
49. #include <sensor_msgs/PointCloud2.h>
50. #include <velodyne_pointcloud/point_types.h>
51. #include <velodyne_pointcloud/rawdata.h>
52.
53. #include <tf/transform_broadcaster.h>
54. #include <tf/transform_datatypes.h>
55.
56. #include <pcl/io/io.h>
57. #include <pcl/io/pcd_io.h>
58. #include <pcl/point_types.h>
59. #include <pcl_conversions/pcl_conversions.h>
60.
61. #ifdef USE_FAST_PCL
62. #include <fast_pcl/registration/ndt.h>
63. #include <fast_pcl/filters/voxel_grid.h>
64. #else
65. #include <pcl/registration/ndt.h>
66. #include <pcl/filters/voxel_grid.h>
67. #endif
68.
69. #include <runtime_manager/ConfigNdtMapping.h>
70. #include <runtime_manager/ConfigNdtMappingOutput.h>
71.
72. struct pose
73. {
74.     double x;
75.     double y;
76.     double z;
```

```

77. double roll;
78. double pitch;
79. double yaw;
80. };
81.
82. // global variables
83. static pose previous_pose, guess_pose, guess_pose_imu, guess_pose_odom, guess_pose_imu
    _odom, current_pose, current_pose_imu, current_pose_odom, current_pose_imu_odom, ndt_po
    se, added_pose, localizer_pose;
84.
85. static ros::Time current_scan_time;
86. static ros::Time previous_scan_time;
87. static ros::Duration scan_duration;
88.
89. static double diff = 0.0;
90. static double diff_x = 0.0, diff_y = 0.0, diff_z = 0.0, diff_yaw; // current_pose - pr
    evious_pose
91. static double offset_imu_x, offset_imu_y, offset_imu_z, offset_imu_roll, offset_imu_pi
    tch, offset_imu_yaw;
92. static double offset_odom_x, offset_odom_y, offset_odom_z, offset_odom_roll, offset_od
    om_pitch, offset_odom_yaw;
93. static double offset_imu_odom_x, offset_imu_odom_y, offset_imu_odom_z, offset_imu_odom
    _roll, offset_imu_odom_pitch, offset_imu_odom_yaw;
94.
95. static double current_velocity_x = 0.0;
96. static double current_velocity_y = 0.0;
97. static double current_velocity_z = 0.0;
98.
99. static double current_velocity_imu_x = 0.0;
100. static double current_velocity_imu_y = 0.0;
101. static double current_velocity_imu_z = 0.0;
102.
103. static pcl::PointCloud<pcl::PointXYZI> map;
104.
105. static pcl::NormalDistributionsTransform<pcl::PointXYZI, pcl::PointXYZI> ndt;
106. // Default values
107. static int max_iter = 30; // Maximum iterations
108. static float ndt_res = 1.0; // Resolution
109. static double step_size = 0.1; // Step size

```

```

110.static double trans_eps = 0.01; // Transformation epsilon
111.
112.// Leaf size of VoxelGrid filter.
113.static double voxel_leaf_size = 2.0;
114.
115.static ros::Time callback_start, callback_end, t1_start, t1_end, t2_start, t2_end, t3_
    start, t3_end, t4_start, t4_end,
116.t5_start, t5_end;
117.static ros::Duration d_callback, d1, d2, d3, d4, d5;
118.
119.static ros::Publisher ndt_map_pub;
120.static ros::Publisher current_pose_pub;
121.static ros::Publisher guess_pose_linaer_pub;
122.static geometry_msgs::PoseStamped current_pose_msg, guess_pose_msg;
123.
124.static ros::Publisher ndt_stat_pub;
125.static std_msgs::Bool ndt_stat_msg;
126.
127.static int initial_scan_loaded = 0;
128.
129.static Eigen::Matrix4f gnss_transform = Eigen::Matrix4f::Identity();
130.
131.static double min_scan_range = 5.0;
132.static double min_add_scan_shift = 1.0;
133.
134.static double _tf_x, _tf_y, _tf_z, _tf_roll, _tf_pitch, _tf_yaw;
135.static Eigen::Matrix4f tf_btol, tf_ltob;
136.
137.static bool isMapUpdate = true;
138.static bool _use_openmp = false;
139.static bool _use_imu = false;
140.static bool _use_odom = false;
141.static bool _imu_upside_down = false;
142.
143.static std::string _imu_topic = "/imu_raw";
144.
145.
146.static double fitness_score;
147.

```

```

148. static sensor_msgs::Imu imu;
149. static nav_msgs::Odometry odom;
150.
151. static void param_callback(const runtime_manager::ConfigNdtMapping::ConstPtr& input)
152. {
153.
154. ndt_res = input->resolution;
155. step_size = input->step_size;
156. trans_eps = input->trans_epsilon;
157. max_iter = input->max_iterations;
158. voxel_leaf_size = input->leaf_size;
159. min_scan_range = input->min_scan_range;
160. min_add_scan_shift = input->min_add_scan_shift;
161.
162. std::cout << "param_callback" << std::endl;
163. std::cout << "ndt_res: " << ndt_res << std::endl;
164. std::cout << "step_size: " << step_size << std::endl;
165. std::cout << "trans_epsilon: " << trans_eps << std::endl;
166. std::cout << "max_iter: " << max_iter << std::endl;
167. std::cout << "voxel_leaf_size: " << voxel_leaf_size << std::endl;
168. std::cout << "min_scan_range: " << min_scan_range << std::endl;
169. std::cout << "min_add_scan_shift: " << min_add_scan_shift << std::endl;
170. }
171.
172. static void output_callback(const runtime_manager::ConfigNdtMappingOutput::ConstPtr& input)
173. {
174. double filter_res = input->filter_res;
175. std::string filename = input->filename;
176. std::cout << "output_callback" << std::endl;
177. std::cout << "filter_res: " << filter_res << std::endl;
178. std::cout << "filename: " << filename << std::endl;
179.
180. pcl::PointCloud<pcl::PointXYZI>::Ptr map_ptr(new pcl::PointCloud<pcl::PointXYZI>(map))
    ;
181. pcl::PointCloud<pcl::PointXYZI>::Ptr map_filtered(new pcl::PointCloud<pcl::PointXYZI>(
    ));
182. map_ptr->header.frame_id = "map";
183. map_filtered->header.frame_id = "map";

```

```

184.sensor_msgs::PointCloud2::Ptr map_msg_ptr(new sensor_msgs::PointCloud2);
185.
186.// Apply voxelgrid filter
187.if (filter_res == 0.0)
188.{
189.std::cout << "Original: " << map_ptr->points.size() << " points." << std::endl;
190.pcl::toROSMsg(*map_ptr, *map_msg_ptr);
191.}
192.else
193.{
194.pcl::VoxelGrid<pcl::PointXYZI> voxel_grid_filter;
195.voxel_grid_filter.setLeafSize(filter_res, filter_res, filter_res);
196.voxel_grid_filter.setInputCloud(map_ptr);
197.voxel_grid_filter.filter(*map_filtered);
198.std::cout << "Original: " << map_ptr->points.size() << " points." << std::endl;
199.std::cout << "Filtered: " << map_filtered->points.size() << " points." << std::endl;
200.pcl::toROSMsg(*map_filtered, *map_msg_ptr);
201.}
202.
203.ndt_map_pub.publish(*map_msg_ptr);
204.
205.// Writing Point Cloud data to PCD file
206.if (filter_res == 0.0)
207.{
208.pcl::io::savePCDFileASCII(filename, *map_ptr);
209.std::cout << "Saved " << map_ptr->
    >points.size() << " data points to " << filename << "." << std::endl;
210.}
211.else
212.{
213.pcl::io::savePCDFileASCII(filename, *map_filtered);
214.std::cout << "Saved " << map_filtered->
    >points.size() << " data points to " << filename << "." << std::endl;
215.}
216.}
217.
218.static void imu_odom_calc(ros::Time current_time)
219.{
220.static ros::Time previous_time = current_time;

```

```

221.double diff_time = (current_time - previous_time).toSec();
222.
223.double diff_imu_roll = imu.angular_velocity.x * diff_time;
224.double diff_imu_pitch = imu.angular_velocity.y * diff_time;
225.double diff_imu_yaw = imu.angular_velocity.z * diff_time;
226.
227.current_pose_imu_odom.roll += diff_imu_roll;
228.current_pose_imu_odom.pitch += diff_imu_pitch;
229.current_pose_imu_odom.yaw += diff_imu_yaw;
230.
231.double diff_distance = odom.twist.twist.linear.x * diff_time;
232.offset_imu_odom_x += diff_distance*cos(-
    current_pose_imu_odom.pitch)*cos(current_pose_imu_odom.yaw);
233.offset_imu_odom_y += diff_distance*cos(-
    current_pose_imu_odom.pitch)*sin(current_pose_imu_odom.yaw);
234.offset_imu_odom_z += diff_distance*sin(-current_pose_imu_odom.pitch);
235.
236.offset_imu_odom_roll += diff_imu_roll;
237.offset_imu_odom_pitch += diff_imu_pitch;
238.offset_imu_odom_yaw += diff_imu_yaw;
239.
240.guess_pose_imu_odom.x = previous_pose.x + offset_imu_odom_x;
241.guess_pose_imu_odom.y = previous_pose.y + offset_imu_odom_y;
242.guess_pose_imu_odom.z = previous_pose.z + offset_imu_odom_z;
243.guess_pose_imu_odom.roll = previous_pose.roll + offset_imu_odom_roll;
244.guess_pose_imu_odom.pitch = previous_pose.pitch + offset_imu_odom_pitch;
245.guess_pose_imu_odom.yaw = previous_pose.yaw + offset_imu_odom_yaw;
246.
247.previous_time = current_time;
248.}
249.
250.
251.static void odom_calc(ros::Time current_time)
252.{
253.static ros::Time previous_time = current_time;
254.double diff_time = (current_time - previous_time).toSec();
255.
256.double diff_odom_roll = odom.twist.twist.angular.x * diff_time;
257.double diff_odom_pitch = odom.twist.twist.angular.y * diff_time;

```

```

258.double diff_odom_yaw    = odom.twist.twist.angular.z * diff_time;
259.
260.current_pose_odom.roll  += diff_odom_roll;
261.current_pose_odom.pitch += diff_odom_pitch;
262.current_pose_odom.yaw   += diff_odom_yaw;
263.
264.double diff_distance = odom.twist.twist.linear.x * diff_time;
265.offset_odom_x += diff_distance*cos(-
    current_pose_odom.pitch)*cos(current_pose_odom.yaw);
266.offset_odom_y += diff_distance*cos(-
    current_pose_odom.pitch)*sin(current_pose_odom.yaw);
267.offset_odom_z += diff_distance*sin(-current_pose_odom.pitch);
268.
269.offset_odom_roll  += diff_odom_roll;
270.offset_odom_pitch += diff_odom_pitch;
271.offset_odom_yaw   += diff_odom_yaw;
272.
273.guess_pose_odom.x    = previous_pose.x    + offset_odom_x;
274.guess_pose_odom.y    = previous_pose.y    + offset_odom_y;
275.guess_pose_odom.z    = previous_pose.z    + offset_odom_z;
276.guess_pose_odom.roll = previous_pose.roll + offset_odom_roll;
277.guess_pose_odom.pitch = previous_pose.pitch + offset_odom_pitch;
278.guess_pose_odom.yaw  = previous_pose.yaw  + offset_odom_yaw;
279.
280.previous_time = current_time;
281.}
282.
283.static void imu_calc(ros::Time current_time)
284.{
285.
286.static ros::Time previous_time = current_time;
287.double diff_time = (current_time - previous_time).toSec();
288.
289.double diff_imu_roll  = imu.angular_velocity.x * diff_time;
290.double diff_imu_pitch = imu.angular_velocity.y * diff_time;
291.double diff_imu_yaw   = imu.angular_velocity.z * diff_time;
292.
293.current_pose_imu.roll += diff_imu_roll;
294.current_pose_imu.pitch += diff_imu_pitch;

```

```

295.current_pose_imu.yaw += diff_imu_yaw;
296.
297.double accX1 = imu.linear_acceleration.x;
298.double accY1 = std::cos(current_pose_imu.roll) * imu.linear_acceleration.y
299.-std::sin(current_pose_imu.roll) * imu.linear_acceleration.z;
300.double accZ1 = std::sin(current_pose_imu.roll) * imu.linear_acceleration.y
301.+std::cos(current_pose_imu.roll) * imu.linear_acceleration.z;
302.
303.double accX2 = std::sin(current_pose_imu.pitch) * accZ1 + std::cos(current_pose_imu.pi
    tch) * accX1;
304.double accY2 = accY1;
305.double accZ2 = std::cos(current_pose_imu.pitch) * accZ1 - std::sin(current_pose_imu.pi
    tch) * accX1;
306.
307.double accX = std::cos(current_pose_imu.yaw) * accX2 - std::sin(current_pose_imu.yaw)
    * accY2;
308.double accY = std::sin(current_pose_imu.yaw) * accX2 + std::cos(current_pose_imu.yaw)
    * accY2;
309.double accZ = accZ2;
310.
311.offset_imu_x += current_velocity_imu_x * diff_time + accX * diff_time * diff_time / 2.
    0;
312.offset_imu_y += current_velocity_imu_y * diff_time + accY * diff_time * diff_time / 2.
    0;
313.offset_imu_z += current_velocity_imu_z * diff_time + accZ * diff_time * diff_time / 2.
    0;
314.
315.current_velocity_imu_x += accX * diff_time;
316.current_velocity_imu_y += accY * diff_time;
317.current_velocity_imu_z += accZ * diff_time;
318.
319.offset_imu_roll += diff_imu_roll;
320.offset_imu_pitch += diff_imu_pitch;
321.offset_imu_yaw += diff_imu_yaw;
322.
323.guess_pose_imu.x = previous_pose.x + offset_imu_x;
324.guess_pose_imu.y = previous_pose.y + offset_imu_y;
325.guess_pose_imu.z = previous_pose.z + offset_imu_z;
326.guess_pose_imu.roll = previous_pose.roll + offset_imu_roll;

```

```

327.guess_pose_imu.pitch = previous_pose.pitch + offset_imu_pitch;
328.guess_pose_imu.yaw   = previous_pose.yaw   + offset_imu_yaw;
329.
330.previous_time = current_time;
331.}
332.
333.
334.static double wrapToPm(double a_num, const double a_max)
335.{
336.if (a_num >= a_max)
337.{
338.a_num -= 2.0 * a_max;
339.}
340.return a_num;
341.}
342.
343.static double wrapToPmPi(double a_angle_rad)
344.{
345.return wrapToPm(a_angle_rad, M_PI);
346.}
347.
348.static void odom_callback(const nav_msgs::Odometry::ConstPtr& input)
349.{
350.//std::cout << __func__ << std::endl;
351.
352.odom = *input;
353.odom_calc(input->header.stamp);
354.}
355.
356.
357.static void imuUpsideDown(const sensor_msgs::Imu::Ptr input)
358.{
359.double input_roll, input_pitch, input_yaw;
360.
361.tf::Quaternion input_orientation;
362.tf::quaternionMsgToTF(input->orientation, input_orientation);
363.tf::Matrix3x3(input_orientation).getRPY(input_roll, input_pitch, input_yaw);
364.
365.input->angular_velocity.x *= -1;

```

```

366.input->angular_velocity.y *= -1;
367.input->angular_velocity.z *= -1;
368.
369.input->linear_acceleration.x *= -1;
370.input->linear_acceleration.y *= -1;
371.input->linear_acceleration.z *= -1;
372.
373.input_roll *= -1;
374.input_pitch *= -1;
375.input_yaw *= -1;
376.
377.input-
    >orientation = tf::createQuaternionMsgFromRollPitchYaw(input_roll, input_pitch, input_
        yaw);
378.}
379.
380.
381.static void imu_callback(const sensor_msgs::Imu::Ptr& input)
382.{
383.//std::cout << __func__ << std::endl;
384.
385.if(_imu_upside_down)
386.imuUpsideDown(input);
387.
388.const ros::Time current_time = input->header.stamp;
389.static ros::Time previous_time = current_time;
390.const double diff_time = (current_time - previous_time).toSec();
391.
392.double imu_roll, imu_pitch, imu_yaw;
393.tf::Quaternion imu_orientation;
394.tf::quaternionMsgToTF(input->orientation, imu_orientation);
395.tf::Matrix3x3(imu_orientation).getRPY(imu_roll, imu_pitch, imu_yaw);
396.
397.imu_roll = wrapToPmPi(imu_roll);
398.imu_pitch = wrapToPmPi(imu_pitch);
399.imu_yaw = wrapToPmPi(imu_yaw);
400.
401.static double previous_imu_roll = imu_roll, previous_imu_pitch = imu_pitch, previous_i
    mu_yaw = imu_yaw;

```

```

402.const double diff_imu_roll = imu_roll - previous_imu_roll;
403.
404.const double diff_imu_pitch = imu_pitch - previous_imu_pitch;
405.
406.double diff_imu_yaw;
407.if(fabs(imu_yaw - previous_imu_yaw) > M_PI)
408.{
409.if(imu_yaw > 0)
410.diff_imu_yaw = (imu_yaw - previous_imu_yaw) - M_PI*2;
411.else
412.diff_imu_yaw = -M_PI*2 - (imu_yaw - previous_imu_yaw);
413.}
414.else
415.diff_imu_yaw = imu_yaw - previous_imu_yaw;
416.
417.imu.header = input->header;
418.imu.linear_acceleration.x = input->linear_acceleration.x;
419.//imu.linear_acceleration.y = input->linear_acceleration.y;
420.//imu.linear_acceleration.z = input->linear_acceleration.z;
421.imu.linear_acceleration.y = 0;
422.imu.linear_acceleration.z = 0;
423.
424.if(diff_time != 0)
425.{
426.imu.angular_velocity.x = diff_imu_roll / diff_time;
427.imu.angular_velocity.y = diff_imu_pitch / diff_time;
428.imu.angular_velocity.z = diff_imu_yaw / diff_time;
429.}
430.else
431.{
432.imu.angular_velocity.x = 0;
433.imu.angular_velocity.y = 0;
434.imu.angular_velocity.z = 0;
435.}
436.
437.imu_calc(input->header.stamp);
438.
439.previous_time = current_time;
440.previous_imu_roll = imu_roll;

```

```

441.previous_imu_pitch = imu_pitch;
442.previous_imu_yaw   = imu_yaw;
443.}
444.
445.
446.static void points_callback(const sensor_msgs::PointCloud2::ConstPtr& input)
447.{
448.double r;
449.pcl::PointXYZI p;
450.pcl::PointCloud<pcl::PointXYZI> tmp, scan;
451.pcl::PointCloud<pcl::PointXYZI>::Ptr filtered_scan_ptr(new pcl::PointCloud<pcl::PointXYZI>());
452.pcl::PointCloud<pcl::PointXYZI>::Ptr transformed_scan_ptr(new pcl::PointCloud<pcl::PointXYZI>());
453.tf::Quaternion q;
454.
455.Eigen::Matrix4f t_localizer(Eigen::Matrix4f::Identity());
456.Eigen::Matrix4f t_base_link(Eigen::Matrix4f::Identity());
457.tf::TransformBroadcaster br;
458.tf::Transform transform;
459.
460.current_scan_time = input->header.stamp;
461.
462.pcl::fromROSMsg(*input, tmp);
463.
464.for (pcl::PointCloud<pcl::PointXYZI>::const_iterator item = tmp.begin(); item != tmp.end(); item++)
465.{
466.p.x = (double)item->x;
467.p.y = (double)item->y;
468.p.z = (double)item->z;
469.p.intensity = (double)item->intensity;
470.
471.r = sqrt(pow(p.x, 2.0) + pow(p.y, 2.0));
472.if (r > min_scan_range)
473.{
474.scan.push_back(p);
475.}
476.}

```

```
477.
478.pcl::PointCloud<pcl::PointXYZ>::Ptr scan_ptr(new pcl::PointCloud<pcl::PointXYZ>(scan
    ));
479.
480.// Add initial point cloud to velodyne_map
481.if (initial_scan_loaded == 0)
482.{
483.pcl::transformPointCloud(*scan_ptr, *transformed_scan_ptr, tf_bt01);
484.map += *transformed_scan_ptr;
485.initial_scan_loaded = 1;
486.}
487.
488.// Apply voxelgrid filter
489.pcl::VoxelGrid<pcl::PointXYZ> voxel_grid_filter;
490.voxel_grid_filter.setLeafSize(voxel_leaf_size, voxel_leaf_size, voxel_leaf_size);
491.voxel_grid_filter.setInputCloud(scan_ptr);
492.voxel_grid_filter.filter(*filtered_scan_ptr);
493.
494.pcl::PointCloud<pcl::PointXYZ>::Ptr map_ptr(new pcl::PointCloud<pcl::PointXYZ>(map))
    ;
495.
496.ndt.setTransformationEpsilon(trans_eps);
497.ndt.setStepSize(step_size);
498.ndt.setResolution(ndt_res);
499.ndt.setMaximumIterations(max_iter);
500.ndt.setInputSource(filtered_scan_ptr);
501.
502.if (isMapUpdate == true)
503.{
504.ndt.setInputTarget(map_ptr);
505.isMapUpdate = false;
506.}
507.
508.guess_pose.x = previous_pose.x + diff_x;
509.guess_pose.y = previous_pose.y + diff_y;
510.guess_pose.z = previous_pose.z + diff_z;
511.guess_pose.roll = previous_pose.roll;
512.guess_pose.pitch = previous_pose.pitch;
513.guess_pose.yaw = previous_pose.yaw + diff_yaw;
```

```

514.
515.
516.if (_use_imu == true && _use_odom == true)
517.imu_odom_calc(current_scan_time);
518.if(_use_imu == true && _use_odom == true)
519.imu_calc(current_scan_time);
520.if (_use_imu == false && _use_odom == true)
521.odom_calc(current_scan_time);
522.
523.pose_guess_pose_for_ndt;
524.if(_use_imu == true && _use_odom == true)
525.guess_pose_for_ndt = guess_pose_imu_odom;
526.else if(_use_imu == true && _use_odom == false)
527.guess_pose_for_ndt = guess_pose_imu;
528.else if(_use_imu == false && _use_odom == true)
529.guess_pose_for_ndt = guess_pose_odom;
530.else
531.guess_pose_for_ndt = guess_pose;
532.
533.
534.Eigen::AngleAxisf init_rotation_x(guess_pose_for_ndt.roll, Eigen::Vector3f::UnitX());
535.Eigen::AngleAxisf init_rotation_y(guess_pose_for_ndt.pitch, Eigen::Vector3f::UnitY());
536.Eigen::AngleAxisf init_rotation_z(guess_pose_for_ndt.yaw, Eigen::Vector3f::UnitZ());
537.
538.Eigen::Translation3f init_translation(guess_pose_for_ndt.x, guess_pose_for_ndt.y, guess_pose_for_ndt.z);
539.
540.Eigen::Matrix4f init_guess =
541.(init_translation * init_rotation_z * init_rotation_y * init_rotation_x).matrix() * tf
    _btol;
542.
543.t3_end = ros::Time::now();
544.d3 = t3_end - t3_start;
545.
546.t4_start = ros::Time::now();
547.
548.pcl::PointCloud<pcl::PointXYZI>::Ptr output_cloud(new pcl::PointCloud<pcl::PointXYZI>)
    ;
549.#ifdef USE_FAST_PCL

```

```

550.if (_use_omp == true)
551.{
552.ndt.omp_align(*output_cloud, init_guess);
553.fitness_score = ndt.omp_getFitnessScore();
554.}
555.else
556.{
557.#endif
558.ndt.align(*output_cloud, init_guess);
559.fitness_score = ndt.getFitnessScore();
560.#ifdef USE_FAST_PCL
561.}
562.#endif
563.
564.t_localizer = ndt.getFinalTransformation();
565.t_base_link = t_localizer * tf_lto_b;
566.
567.pcl::transformPointCloud(*scan_ptr, *transformed_scan_ptr, t_localizer);
568.
569.tf::Matrix3x3 mat_l, mat_b;
570.
571.mat_l.setValue(static_cast<double>(t_localizer(0, 0)), static_cast<double>(t_localizer
    (0, 1)),
572.static_cast<double>(t_localizer(0, 2)), static_cast<double>(t_localizer(1, 0)),
573.static_cast<double>(t_localizer(1, 1)), static_cast<double>(t_localizer(1, 2)),
574.static_cast<double>(t_localizer(2, 0)), static_cast<double>(t_localizer(2, 1)),
575.static_cast<double>(t_localizer(2, 2)));
576.
577.mat_b.setValue(static_cast<double>(t_base_link(0, 0)), static_cast<double>(t_base_link
    (0, 1)),
578.static_cast<double>(t_base_link(0, 2)), static_cast<double>(t_base_link(1, 0)),
579.static_cast<double>(t_base_link(1, 1)), static_cast<double>(t_base_link(1, 2)),
580.static_cast<double>(t_base_link(2, 0)), static_cast<double>(t_base_link(2, 1)),
581.static_cast<double>(t_base_link(2, 2)));
582.
583.// Update localizer_pose.
584.localizer_pose.x = t_localizer(0, 3);
585.localizer_pose.y = t_localizer(1, 3);
586.localizer_pose.z = t_localizer(2, 3);

```

```

587.mat_l.getRPY(localizer_pose.roll, localizer_pose.pitch, localizer_pose.yaw, 1);
588.
589.// Update ndt_pose.
590.ndt_pose.x = t_base_link(0, 3);
591.ndt_pose.y = t_base_link(1, 3);
592.ndt_pose.z = t_base_link(2, 3);
593.mat_b.getRPY(ndt_pose.roll, ndt_pose.pitch, ndt_pose.yaw, 1);
594.
595.current_pose.x = ndt_pose.x;
596.current_pose.y = ndt_pose.y;
597.current_pose.z = ndt_pose.z;
598.current_pose.roll = ndt_pose.roll;
599.current_pose.pitch = ndt_pose.pitch;
600.current_pose.yaw = ndt_pose.yaw;
601.
602.transform.setOrigin(tf::Vector3(current_pose.x, current_pose.y, current_pose.z));
603.q.setRPY(current_pose.roll, current_pose.pitch, current_pose.yaw);
604.transform.setRotation(q);
605.
606.br.sendTransform(tf::StampedTransform(transform, current_scan_time, "map", "base_link"
    ));
607.
608.scan_duration = current_scan_time - previous_scan_time;
609.double secs = scan_duration.toSec();
610.
611.// Calculate the offset (current_pos - previous_pos)
612.diff_x = current_pose.x - previous_pose.x;
613.diff_y = current_pose.y - previous_pose.y;
614.diff_z = current_pose.z - previous_pose.z;
615.diff_yaw = current_pose.yaw - previous_pose.yaw;
616.diff = sqrt(diff_x * diff_x + diff_y * diff_y + diff_z * diff_z);
617.
618.current_velocity_x = diff_x / secs;
619.current_velocity_y = diff_y / secs;
620.current_velocity_z = diff_z / secs;
621.
622.current_pose_imu.x = current_pose.x;
623.current_pose_imu.y = current_pose.y;
624.current_pose_imu.z = current_pose.z;

```

```
625.current_pose_imu.roll = current_pose.roll;
626.current_pose_imu.pitch = current_pose.pitch;
627.current_pose_imu.yaw = current_pose.yaw;
628.
629.current_pose_odom.x = current_pose.x;
630.current_pose_odom.y = current_pose.y;
631.current_pose_odom.z = current_pose.z;
632.current_pose_odom.roll = current_pose.roll;
633.current_pose_odom.pitch = current_pose.pitch;
634.current_pose_odom.yaw = current_pose.yaw;
635.
636.current_pose_imu_odom.x = current_pose.x;
637.current_pose_imu_odom.y = current_pose.y;
638.current_pose_imu_odom.z = current_pose.z;
639.current_pose_imu_odom.roll = current_pose.roll;
640.current_pose_imu_odom.pitch = current_pose.pitch;
641.current_pose_imu_odom.yaw = current_pose.yaw;
642.
643.current_velocity_imu_x = current_velocity_x;
644.current_velocity_imu_y = current_velocity_y;
645.current_velocity_imu_z = current_velocity_z;
646.
647.// Update position and posture. current_pos -> previous_pos
648.previous_pose.x = current_pose.x;
649.previous_pose.y = current_pose.y;
650.previous_pose.z = current_pose.z;
651.previous_pose.roll = current_pose.roll;
652.previous_pose.pitch = current_pose.pitch;
653.previous_pose.yaw = current_pose.yaw;
654.
655.previous_scan_time.sec = current_scan_time.sec;
656.previous_scan_time.nsec = current_scan_time.nsec;
657.
658.
659.offset_imu_x = 0.0;
660.offset_imu_y = 0.0;
661.offset_imu_z = 0.0;
662.offset_imu_roll = 0.0;
663.offset_imu_pitch = 0.0;
```

```

664.offset_imu_yaw = 0.0;

665.

666.offset_odom_x = 0.0;

667.offset_odom_y = 0.0;

668.offset_odom_z = 0.0;

669.offset_odom_roll = 0.0;

670.offset_odom_pitch = 0.0;

671.offset_odom_yaw = 0.0;

672.

673.offset_imu_odom_x = 0.0;

674.offset_imu_odom_y = 0.0;

675.offset_imu_odom_z = 0.0;

676.offset_imu_odom_roll = 0.0;

677.offset_imu_odom_pitch = 0.0;

678.offset_imu_odom_yaw = 0.0;

679.

680.// Calculate the shift between added_pos and current_pos

681.double shift = sqrt(pow(current_pose.x - added_pose.x, 2.0) + pow(current_pose.y - add
    ed_pose.y, 2.0));

682.if (shift >= min_add_scan_shift)

683.{

684.map += *transformed_scan_ptr;

685.added_pose.x = current_pose.x;

686.added_pose.y = current_pose.y;

687.added_pose.z = current_pose.z;

688.added_pose.roll = current_pose.roll;

689.added_pose.pitch = current_pose.pitch;

690.added_pose.yaw = current_pose.yaw;

691.isMapUpdate = true;

692.}

693.

694.sensor_msgs::PointCloud2::Ptr map_msg_ptr(new sensor_msgs::PointCloud2);

695.pcl::toROSMsg(*map_ptr, *map_msg_ptr);

696.ndt_map_pub.publish(*map_msg_ptr);

697.

698.q.setRPY(current_pose.roll, current_pose.pitch, current_pose.yaw);

699.current_pose_msg.header.frame_id = "map";

700.current_pose_msg.header.stamp = current_scan_time;

701.current_pose_msg.pose.position.x = current_pose.x;

```

```

702.current_pose_msg.pose.position.y = current_pose.y;
703.current_pose_msg.pose.position.z = current_pose.z;
704.current_pose_msg.pose.orientation.x = q.x();
705.current_pose_msg.pose.orientation.y = q.y();
706.current_pose_msg.pose.orientation.z = q.z();
707.current_pose_msg.pose.orientation.w = q.w();
708.
709.current_pose_pub.publish(current_pose_msg);
710.
711.
712.std::cout << "-----
    " << std::endl;
713.std::cout << "Sequence number: " << input->header.seq << std::endl;
714.std::cout << "Number of scan points: " << scan_ptr->size() << " points." << std::endl;
715.std::cout << "Number of filtered scan points: " << filtered_scan_ptr-
    >size() << " points." << std::endl;
716.std::cout << "transformed_scan_ptr: " << transformed_scan_ptr-
    >points.size() << " points." << std::endl;
717.std::cout << "map: " << map.points.size() << " points." << std::endl;
718.std::cout << "NDT has converged: " << ndt.hasConverged() << std::endl;
719.std::cout << "Fitness score: " << fitness_score << std::endl;
720.std::cout << "Number of iteration: " << ndt.getFinalNumIteration() << std::endl;
721.std::cout << "(x,y,z,roll,pitch,yaw):" << std::endl;
722.std::cout << "(" << current_pose.x << ", " << current_pose.y << ", " << current_pose.z
    << ", " << current_pose.roll
723.<< ", " << current_pose.pitch << ", " << current_pose.yaw << ")" << std::endl;
724.std::cout << "Transformation Matrix:" << std::endl;
725.std::cout << t_localizer << std::endl;
726.std::cout << "shift: " << shift << std::endl;
727.std::cout << "-----
    " << std::endl;
728.}
729.
730.int main(int argc, char** argv)
731.{
732.previous_pose.x = 0.0;
733.previous_pose.y = 0.0;
734.previous_pose.z = 0.0;
735.previous_pose.roll = 0.0;

```

```
736.previous_pose.pitch = 0.0;
737.previous_pose.yaw = 0.0;
738.
739.ndt_pose.x = 0.0;
740.ndt_pose.y = 0.0;
741.ndt_pose.z = 0.0;
742.ndt_pose.roll = 0.0;
743.ndt_pose.pitch = 0.0;
744.ndt_pose.yaw = 0.0;
745.
746.current_pose.x = 0.0;
747.current_pose.y = 0.0;
748.current_pose.z = 0.0;
749.current_pose.roll = 0.0;
750.current_pose.pitch = 0.0;
751.current_pose.yaw = 0.0;
752.
753.current_pose_imu.x = 0.0;
754.current_pose_imu.y = 0.0;
755.current_pose_imu.z = 0.0;
756.current_pose_imu.roll = 0.0;
757.current_pose_imu.pitch = 0.0;
758.current_pose_imu.yaw = 0.0;
759.
760.guess_pose.x = 0.0;
761.guess_pose.y = 0.0;
762.guess_pose.z = 0.0;
763.guess_pose.roll = 0.0;
764.guess_pose.pitch = 0.0;
765.guess_pose.yaw = 0.0;
766.
767.added_pose.x = 0.0;
768.added_pose.y = 0.0;
769.added_pose.z = 0.0;
770.added_pose.roll = 0.0;
771.added_pose.pitch = 0.0;
772.added_pose.yaw = 0.0;
773.
774.diff_x = 0.0;
```

```

775.diff_y = 0.0;
776.diff_z = 0.0;
777.diff_yaw = 0.0;
778.
779.offset_imu_x = 0.0;
780.offset_imu_y = 0.0;
781.offset_imu_z = 0.0;
782.offset_imu_roll = 0.0;
783.offset_imu_pitch = 0.0;
784.offset_imu_yaw = 0.0;
785.
786.offset_odom_x = 0.0;
787.offset_odom_y = 0.0;
788.offset_odom_z = 0.0;
789.offset_odom_roll = 0.0;
790.offset_odom_pitch = 0.0;
791.offset_odom_yaw = 0.0;
792.
793.offset_imu_odom_x = 0.0;
794.offset_imu_odom_y = 0.0;
795.offset_imu_odom_z = 0.0;
796.offset_imu_odom_roll = 0.0;
797.offset_imu_odom_pitch = 0.0;
798.offset_imu_odom_yaw = 0.0;
799.
800.ros::init(argc, argv, "ndt_mapping");
801.
802.ros::NodeHandle nh;
803.ros::NodeHandle private_nh("~");
804.
805.// setting parameters
806.private_nh.getParam("use_openmp", _use_openmp);
807.private_nh.getParam("use_imu", _use_imu);
808.private_nh.getParam("use_odom", _use_odom);
809.private_nh.getParam("imu_upside_down", _imu_upside_down);
810.private_nh.getParam("imu_topic", _imu_topic);
811.
812.std::cout << "use_openmp: " << _use_openmp << std::endl;
813.std::cout << "use_imu: " << _use_imu << std::endl;

```

```

814.std::cout << "imu_upside_down: " << _imu_upside_down << std::endl;
815.std::cout << "use_odom: " << _use_odom << std::endl;
816.
817.std::cout << "imu_topic: " << _imu_topic << std::endl;
818.
819.if (nh.getParam("tf_x", _tf_x) == false)
820.{
821.std::cout << "tf_x is not set." << std::endl;
822.return 1;
823.}
824.if (nh.getParam("tf_y", _tf_y) == false)
825.{
826.std::cout << "tf_y is not set." << std::endl;
827.return 1;
828.}
829.if (nh.getParam("tf_z", _tf_z) == false)
830.{
831.std::cout << "tf_z is not set." << std::endl;
832.return 1;
833.}
834.if (nh.getParam("tf_roll", _tf_roll) == false)
835.{
836.std::cout << "tf_roll is not set." << std::endl;
837.return 1;
838.}
839.if (nh.getParam("tf_pitch", _tf_pitch) == false)
840.{
841.std::cout << "tf_pitch is not set." << std::endl;
842.return 1;
843.}
844.if (nh.getParam("tf_yaw", _tf_yaw) == false)
845.{
846.std::cout << "tf_yaw is not set." << std::endl;
847.return 1;
848.}
849.
850.std::cout << "(tf_x,tf_y,tf_z,tf_roll,tf_pitch,tf_yaw): (" << _tf_x << ", " << _tf_y <
    < ", " << _tf_z << ", "
851.<< _tf_roll << ", " << _tf_pitch << ", " << _tf_yaw << ")" << std::endl;

```

```

852.
853.Eigen::Translation3f tl_btol(_tf_x, _tf_y, _tf_z);           // tl: translation
854.Eigen::AngleAxisf rot_x_btol(_tf_roll, Eigen::Vector3f::UnitX()); // rot: rotation
855.Eigen::AngleAxisf rot_y_btol(_tf_pitch, Eigen::Vector3f::UnitY());
856.Eigen::AngleAxisf rot_z_btol(_tf_yaw, Eigen::Vector3f::UnitZ());
857.tf_btol = (tl_btol * rot_z_btol * rot_y_btol * rot_x_btol).matrix();
858.
859.Eigen::Translation3f tl_ltob((-1.0) * _tf_x, (-1.0) * _tf_y, (-
    1.0) * _tf_z); // tl: translation
860.Eigen::AngleAxisf rot_x_ltob((-
    1.0) * _tf_roll, Eigen::Vector3f::UnitX()); // rot: rotation
861.Eigen::AngleAxisf rot_y_ltob((-1.0) * _tf_pitch, Eigen::Vector3f::UnitY());
862.Eigen::AngleAxisf rot_z_ltob((-1.0) * _tf_yaw, Eigen::Vector3f::UnitZ());
863.tf_ltob = (tl_ltob * rot_z_ltob * rot_y_ltob * rot_x_ltob).matrix();
864.
865.map.header.frame_id = "map";
866.
867.ndt_map_pub = nh.advertise<sensor_msgs::PointCloud2>("/ndt_map", 1000);
868.current_pose_pub = nh.advertise<geometry_msgs::PoseStamped>("/current_pose", 1000);
869.
870.ros::Subscriber param_sub = nh.subscribe("config/ndt_mapping", 10, param_callback);
871.ros::Subscriber output_sub = nh.subscribe("config/ndt_mapping_output", 10, output_call
    back);
872.ros::Subscriber points_sub = nh.subscribe("velodyne_points_0", 100000, points_callback
    );
873.ros::Subscriber odom_sub = nh.subscribe("/odom_pose", 100000, odom_callback);
874.ros::Subscriber imu_sub = nh.subscribe(_imu_topic, 100000, imu_callback);
875.
876.ros::spin();
877.
878.return 0;
879.}

```

Ndt_matching

```

1. * Copyright (c) 2015, Nagoya University

```

```
2. * All rights reserved.
3. *
4. * Redistribution and use in source and binary forms, with or without
5. * modification, are permitted provided that the following conditions are met:
6. *
7. * * Redistributions of source code must retain the above copyright notice, this
8. * list of conditions and the following disclaimer.
9. *
10. * * Redistributions in binary form must reproduce the above copyright notice,
11. * this list of conditions and the following disclaimer in the documentation
12. * and/or other materials provided with the distribution.
13. *
14. * * Neither the name of Autoware nor the names of its
15. * contributors may be used to endorse or promote products derived from
16. * this software without specific prior written permission.
17. *
18. * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
19. * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
20. * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
21. * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
22. * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
23. * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
24. * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
25. * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
26. * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
27. * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
28. */
29.
30. /*
31. Localization program using Normal Distributions Transform
32.
33. Yuki KITSUKAWA
34. */
35.
36. #include <iostream>
37. #include <sstream>
38. #include <fstream>
39. #include <string>
40. #include <chrono>
```

```
41.
42. #include <ros/ros.h>
43. #include <std_msgs/Float32.h>
44. #include <std_msgs/String.h>
45. #include <std_msgs/Bool.h>
46. #include <nav_msgs/Odometry.h>
47. #include <nav_msgs/Path.h>
48. #include <sensor_msgs/Imu.h>
49. #include <sensor_msgs/PointCloud2.h>
50. #include <velodyne_pointcloud/point_types.h>
51. #include <velodyne_pointcloud/rawdata.h>
52.
53. #include <geometry_msgs/TwistStamped.h>
54. #include <geometry_msgs/PoseWithCovarianceStamped.h>
55.
56. #include <tf/tf.h>
57. #include <tf/transform_broadcaster.h>
58. #include <tf/transform_datatypes.h>
59. #include <tf/transform_listener.h>
60.
61. #include <pcl/io/io.h>
62. #include <pcl/io/pcd_io.h>
63. #include <pcl/point_types.h>
64. #include <pcl_conversions/pcl_conversions.h>
65. #ifdef USE_FAST_PCL
66. #include <fast_pcl/registration/ndt.h>
67. #else
68. #include <pcl/registration/ndt.h>
69. #endif
70.
71. #include <pcl_ros/point_cloud.h>
72. #include <pcl_ros/transforms.h>
73.
74. #include <runtime_manager/ConfigNdt.h>
75.
76. #include <ndt_localizer/ndt_stat.h>
77.
78. #define PREDICT_POSE_THRESHOLD 0.5
79.
```

```

80. #define Wa 0.4
81. #define Wb 0.3
82. #define Wc 0.3
83.
84. struct pose
85. {
86.     double x;
87.     double y;
88.     double z;
89.     double roll;
90.     double pitch;
91.     double yaw;
92. };
93.
94. static pose initial_pose, predict_pose, predict_pose_imu, predict_pose_odom, predict_p
    ose_imu_odom, previous_pose, ndt_pose, current_pose, current_pose_imu, current_pose_od
    om, current_pose_imu_odom, localizer_pose, previous_gnss_pose,
95. current_gnss_pose;
96.
97. static double offset_x, offset_y, offset_z, offset_yaw; // current_pos - previous_pos
    e
98. static double offset_imu_x, offset_imu_y, offset_imu_z, offset_imu_roll, offset_imu_pi
    tch, offset_imu_yaw;
99. static double offset_odom_x, offset_odom_y, offset_odom_z, offset_odom_roll, offset_od
    om_pitch, offset_odom_yaw;
100. static double offset_imu_odom_x, offset_imu_odom_y, offset_imu_odom_z, offset_imu_odom
    _roll, offset_imu_odom_pitch, offset_imu_odom_yaw;
101.
102. // Can't load if typed "pcl::PointCloud<pcl::PointXYZRGB> map, add;"
103. static pcl::PointCloud<pcl::PointXYZ> map, add;
104.
105. // If the map is loaded, map_loaded will be 1.
106. static int map_loaded = 0;
107. static int _use_gnss = 1;
108. static int init_pos_set = 0;
109.
110. static pcl::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> ndt;
111. // Default values
112. static int max_iter = 30; // Maximum iterations

```

```

113. static float ndt_res = 1.0;      // Resolution
114. static double step_size = 0.1;   // Step size
115. static double trans_eps = 0.01;  // Transformation epsilon
116.
117. static ros::Publisher predict_pose_pub;
118. static geometry_msgs::PoseStamped predict_pose_msg;
119.
120. static ros::Publisher predict_pose_imu_pub;
121. static geometry_msgs::PoseStamped predict_pose_imu_msg;
122.
123. static ros::Publisher predict_pose_odom_pub;
124. static geometry_msgs::PoseStamped predict_pose_odom_msg;
125.
126. static ros::Publisher predict_pose_imu_odom_pub;
127. static geometry_msgs::PoseStamped predict_pose_imu_odom_msg;
128.
129. static ros::Publisher ndt_pose_pub;
130. static geometry_msgs::PoseStamped ndt_pose_msg;
131. //Add By WEN WS
132. static geometry_msgs::PoseStamped navsatOdomPose;
133. static nav_msgs::Odometry navsatOdomPoseOdom;
134.
135.
136. // current_pose is published by vel_pose_mux
137. /*
138. static ros::Publisher current_pose_pub;
139. static geometry_msgs::PoseStamped current_pose_msg;
140. */
141.
142. static ros::Publisher localizer_pose_pub;
143. static geometry_msgs::PoseStamped localizer_pose_msg;
144.
145. static ros::Publisher estimate_twist_pub;
146. static geometry_msgs::TwistStamped estimate_twist_msg;
147.
148. static ros::Time current_scan_time;
149. static ros::Time previous_scan_time;
150. static ros::Duration scan_duration;
151.

```

```

152. static double exe_time = 0.0;
153. static int iteration = 0;
154. static double fitness_score = 0.0;
155. static double trans_probability = 0.0;
156.
157. static double diff = 0.0;
158. static double diff_x = 0.0, diff_y = 0.0, diff_z = 0.0, diff_yaw;
159.
160. static double current_velocity = 0.0, previous_velocity = 0.0, previous_previous_velocity = 0.0; // [m/s]
161. static double current_velocity_x = 0.0, previous_velocity_x = 0.0;
162. static double current_velocity_y = 0.0, previous_velocity_y = 0.0;
163. static double current_velocity_z = 0.0, previous_velocity_z = 0.0;
164. // static double current_velocity_yaw = 0.0, previous_velocity_yaw = 0.0;
165. static double current_velocity_smooth = 0.0;
166.
167. static double current_velocity_imu_x = 0.0;
168. static double current_velocity_imu_y = 0.0;
169. static double current_velocity_imu_z = 0.0;
170.
171. static double current_accel = 0.0, previous_accel = 0.0; // [m/s^2]
172. static double current_accel_x = 0.0;
173. static double current_accel_y = 0.0;
174. static double current_accel_z = 0.0;
175. // static double current_accel_yaw = 0.0;
176.
177. static double angular_velocity = 0.0;
178.
179. static int use_predict_pose = 0;
180.
181. static ros::Publisher estimated_vel_mps_pub, estimated_vel_kmph_pub, estimated_vel_pub;
182. static std_msgs::Float32 estimated_vel_mps, estimated_vel_kmph, previous_estimated_vel_kmph;
183.
184. static std::chrono::time_point<std::chrono::system_clock> matching_start, matching_end;
185.
186. static ros::Publisher time_ndt_matching_pub;

```

```
187.static std_msgs::Float32 time_ndt_matching;
188.
189.static int _queue_size = 1000;
190.
191.static ros::Publisher ndt_stat_pub;
192.static ndt_localizer::ndt_stat ndt_stat_msg;
193.
194.static double predict_pose_error = 0.0;
195.
196.static double _tf_x, _tf_y, _tf_z, _tf_roll, _tf_pitch, _tf_yaw;
197.static Eigen::Matrix4f tf_btol, tf_lto;
198.
199.static std::string _localizer = "velodyne";
200.static std::string _offset = "linear"; // linear, zero, quadratic
201.
202.static ros::Publisher ndt_reliability_pub;
203.static std_msgs::Float32 ndt_reliability;
204.
205.static bool _use_openmp = false;
206.static bool _get_height = false;
207.static bool _use_local_transform = false;
208.static bool _use_imu = false;
209.static bool _use_odom = false;
210.static bool _imu_upside_down = false;
211.
212.static std::string _imu_topic = "/imu_raw";
213.
214.static std::ofstream ofs;
215.static std::string filename;
216.
217.static sensor_msgs::Imu imu;
218.static nav_msgs::Odometry odom;
219.
220.
221.// static tf::TransformListener local_transform_listener;
222.static tf::StampedTransform local_transform;
223.
224.
225.//Add by WEN WS
```

```

226. static nav_msgs::Path ndtPath;//create a path denote ndt pose
227. static ros::Publisher pubNdtPath;
228.
229. static nav_msgs::Path spancptPath;//create a path denote span-cpt pose
230. static ros::Publisher pubspancptPath;
231. static bool firstSpancptOdomMessage=0;// if the first topic got, this value will be 1
232. static double firstSpancptOdomMessageX=0;//X of the first message
233. static double firstSpancptOdomMessageY=0;//Y of the first message
234. static double firstSpancptOdomMessageZ=0;//Z of the first message
235.
236.
237. static void param_callback(const runtime_manager::ConfigNdt::ConstPtr& input)
238. {
239. if (_use_gnss != input->init_pos_gnss)
240. {
241. init_pos_set = 0;
242. }
243. else if (_use_gnss == 0 &&
244. (initial_pose.x != input->x || initial_pose.y != input->y || initial_pose.z != input-
    >z ||
245. initial_pose.roll != input->roll || initial_pose.pitch != input-
    >pitch || initial_pose.yaw != input->yaw))
246. {
247. init_pos_set = 0;
248. }
249.
250. _use_gnss = input->init_pos_gnss;
251.
252. // Setting parameters
253. if (input->resolution != ndt_res)
254. {
255. ndt_res = input->resolution;
256. ndt.setResolution(ndt_res);
257. }
258. if (input->step_size != step_size)
259. {
260. step_size = input->step_size;
261. ndt.setStepSize(step_size);
262. }

```

```

263. if (input->trans_epsilon != trans_eps)
264. {
265.     trans_eps = input->trans_epsilon;
266.     ndt.setTransformationEpsilon(trans_eps);
267. }
268. if (input->max_iterations != max_iter)
269. {
270.     max_iter = input->max_iterations;
271.     ndt.setMaximumIterations(max_iter);
272. }
273.
274. if (_use_gnss == 0 && init_pos_set == 0)
275. {
276.     initial_pose.x = input->x;
277.     initial_pose.y = input->y;
278.     initial_pose.z = input->z;
279.     initial_pose.roll = input->roll;
280.     initial_pose.pitch = input->pitch;
281.     initial_pose.yaw = input->yaw;
282.
283.     if (_use_local_transform == true)
284.     {
285.         tf::Vector3 v(input->x, input->y, input->z);
286.         tf::Quaternion q;
287.         q.setRPY(input->roll, input->pitch, input->yaw);
288.         tf::Transform transform(q, v);
289.         initial_pose.x = (local_transform.inverse() * transform).getOrigin().getX();
290.         initial_pose.y = (local_transform.inverse() * transform).getOrigin().getY();
291.         initial_pose.z = (local_transform.inverse() * transform).getOrigin().getZ();
292.
293.         tf::Matrix3x3 m(q);
294.         m.getRPY(initial_pose.roll, initial_pose.pitch, initial_pose.yaw);
295.
296.         std::cout << "initial_pose.x: " << initial_pose.x << std::endl;
297.         std::cout << "initial_pose.y: " << initial_pose.y << std::endl;
298.         std::cout << "initial_pose.z: " << initial_pose.z << std::endl;
299.         std::cout << "initial_pose.roll: " << initial_pose.roll << std::endl;
300.         std::cout << "initial_pose.pitch: " << initial_pose.pitch << std::endl;
301.         std::cout << "initial_pose.yaw: " << initial_pose.yaw << std::endl;

```

```
302.}
303.
304.// Setting position and posture for the first time.
305.localizer_pose.x = initial_pose.x;
306.localizer_pose.y = initial_pose.y;
307.localizer_pose.z = initial_pose.z;
308.localizer_pose.roll = initial_pose.roll;
309.localizer_pose.pitch = initial_pose.pitch;
310.localizer_pose.yaw = initial_pose.yaw;
311.
312.previous_pose.x = initial_pose.x;
313.previous_pose.y = initial_pose.y;
314.previous_pose.z = initial_pose.z;
315.previous_pose.roll = initial_pose.roll;
316.previous_pose.pitch = initial_pose.pitch;
317.previous_pose.yaw = initial_pose.yaw;
318.
319.current_pose.x = initial_pose.x;
320.current_pose.y = initial_pose.y;
321.current_pose.z = initial_pose.z;
322.current_pose.roll = initial_pose.roll;
323.current_pose.pitch = initial_pose.pitch;
324.current_pose.yaw = initial_pose.yaw;
325.
326.current_velocity = 0;
327.current_velocity_x = 0;
328.current_velocity_y = 0;
329.current_velocity_z = 0;
330.angular_velocity = 0;
331.
332.current_pose_imu.x = 0;
333.current_pose_imu.y = 0;
334.current_pose_imu.z = 0;
335.current_pose_imu.roll = 0;
336.current_pose_imu.pitch = 0;
337.current_pose_imu.yaw = 0;
338.
339.current_velocity_imu_x = current_velocity_x;
340.current_velocity_imu_y = current_velocity_y;
```

```

341.current_velocity_imu_z = current_velocity_z;
342.init_pos_set = 1;
343.}
344.}
345.
346.static void map_callback(const sensor_msgs::PointCloud2::ConstPtr& input)
347.{
348.if (map_loaded == 0)
349.{
350.// Convert the data type(from sensor_msgs to pcl).
351.pcl::fromROSMsg(*input, map);
352.
353.if (_use_local_transform == true)
354.{
355.tf::TransformListener local_transform_listener;
356.try
357.{
358.ros::Time now = ros::Time(0);
359.local_transform_listener.waitForTransform("/map", "/world", now, ros::Duration(10.0));
360.local_transform_listener.lookupTransform("/map", "world", now, local_transform);
361.}
362.catch (tf::TransformException& ex)
363.{
364.ROS_ERROR("%s", ex.what());
365.}
366.
367.pcl_ros::transformPointCloud(map, map, local_transform.inverse());
368.}
369.
370.pcl::PointCloud<pcl::PointXYZ>::Ptr map_ptr(new pcl::PointCloud<pcl::PointXYZ>(map));
371.// Setting point cloud to be aligned to.
372.ndt.setInputTarget(map_ptr);
373.
374.// Setting NDT parameters to default values
375.ndt.setMaximumIterations(max_iter);
376.ndt.setResolution(ndt_res);
377.ndt.setStepSize(step_size);
378.ndt.setTransformationEpsilon(trans_eps);
379.

```

```

380.map_loaded = 1;
381.}
382.}
383.
384.static void gnss_callback(const geometry_msgs::PoseStamped::ConstPtr& input)
385.{
386.tf::Quaternion gnss_q(input->pose.orientation.x, input->pose.orientation.y, input-
    >pose.orientation.z,
387.input->pose.orientation.w);
388.tf::Matrix3x3 gnss_m(gnss_q);
389.current_gnss_pose.x = input->pose.position.x;
390.current_gnss_pose.y = input->pose.position.y;
391.current_gnss_pose.z = input->pose.position.z;
392.gnss_m.getRPY(current_gnss_pose.roll, current_gnss_pose.pitch, current_gnss_pose.yaw);
393.
394.if ((_use_gnss == 1 && init_pos_set == 0) || fitness_score >= 500.0)
395.{
396.previous_pose.x = previous_gnss_pose.x;
397.previous_pose.y = previous_gnss_pose.y;
398.previous_pose.z = previous_gnss_pose.z;
399.previous_pose.roll = previous_gnss_pose.roll;
400.previous_pose.pitch = previous_gnss_pose.pitch;
401.previous_pose.yaw = previous_gnss_pose.yaw;
402.
403.current_pose.x = current_gnss_pose.x;
404.current_pose.y = current_gnss_pose.y;
405.current_pose.z = current_gnss_pose.z;
406.current_pose.roll = current_gnss_pose.roll;
407.current_pose.pitch = current_gnss_pose.pitch;
408.current_pose.yaw = current_gnss_pose.yaw;
409.
410.current_pose_imu = current_pose_odom = current_pose_imu_odom = current_pose;
411.
412.offset_x = current_pose.x - previous_pose.x;
413.offset_y = current_pose.y - previous_pose.y;
414.offset_z = current_pose.z - previous_pose.z;
415.offset_yaw = current_pose.yaw - previous_pose.yaw;
416.
417.init_pos_set = 1;

```

```

418.}
419.
420.previous_gnss_pose.x = current_gnss_pose.x;
421.previous_gnss_pose.y = current_gnss_pose.y;
422.previous_gnss_pose.z = current_gnss_pose.z;
423.previous_gnss_pose.roll = current_gnss_pose.roll;
424.previous_gnss_pose.pitch = current_gnss_pose.pitch;
425.previous_gnss_pose.yaw = current_gnss_pose.yaw;
426.}
427.
428.static void initialpose_callback(const geometry_msgs::PoseWithCovarianceStamped::Const
    Ptr& input)
429.{
430.tf::TransformListener listener;
431.tf::StampedTransform transform;
432.try
433.{
434.ros::Time now = ros::Time(0);
435.listener.waitForTransform("/map", "/world", now, ros::Duration(10.0));
436.listener.lookupTransform("/map", "world", now, transform);
437.}
438.catch (tf::TransformException& ex)
439.{
440.ROS_ERROR("%s", ex.what());
441.}
442.
443.tf::Quaternion q(input->pose.pose.orientation.x, input-
    >pose.pose.orientation.y, input->pose.pose.orientation.z,
444.input->pose.pose.orientation.w);
445.tf::Matrix3x3 m(q);
446.
447.if (_use_local_transform == true)
448.{
449.current_pose.x = input->pose.pose.position.x;
450.current_pose.y = input->pose.pose.position.y;
451.current_pose.z = input->pose.pose.position.z;
452.}
453.else
454.{

```

```
455.current_pose.x = input->pose.pose.position.x + transform.getOrigin().x();
456.current_pose.y = input->pose.pose.position.y + transform.getOrigin().y();
457.current_pose.z = input->pose.pose.position.z + transform.getOrigin().z();
458.}
459.m.getRPY(current_pose.roll, current_pose.pitch, current_pose.yaw);
460.
461.if (_get_height == true && map_loaded == 1)
462.{
463.double min_distance = DBL_MAX;
464.double nearest_z = current_pose.z;
465.for (const auto& p : map)
466.{
467.double distance = hypot(current_pose.x - p.x, current_pose.y - p.y);
468.if (distance < min_distance)
469.{
470.min_distance = distance;
471.nearest_z = p.z;
472.}
473.}
474.current_pose.z = nearest_z;
475.}
476.
477.current_pose_imu = current_pose_odom = current_pose_imu_odom = current_pose;
478.previous_pose.x = current_pose.x;
479.previous_pose.y = current_pose.y;
480.previous_pose.z = current_pose.z;
481.previous_pose.roll = current_pose.roll;
482.previous_pose.pitch = current_pose.pitch;
483.previous_pose.yaw = current_pose.yaw;
484.
485.offset_x = 0.0;
486.offset_y = 0.0;
487.offset_z = 0.0;
488.offset_yaw = 0.0;
489.
490.offset_imu_x = 0.0;
491.offset_imu_y = 0.0;
492.offset_imu_z = 0.0;
493.offset_imu_roll = 0.0;
```

```

494.offset_imu_pitch = 0.0;
495.offset_imu_yaw = 0.0;
496.
497.offset_odom_x = 0.0;
498.offset_odom_y = 0.0;
499.offset_odom_z = 0.0;
500.offset_odom_roll = 0.0;
501.offset_odom_pitch = 0.0;
502.offset_odom_yaw = 0.0;
503.
504.offset_imu_odom_x = 0.0;
505.offset_imu_odom_y = 0.0;
506.offset_imu_odom_z = 0.0;
507.offset_imu_odom_roll = 0.0;
508.offset_imu_odom_pitch = 0.0;
509.offset_imu_odom_yaw = 0.0;
510.
511.}
512.
513.static void imu_odom_calc(ros::Time current_time)
514.{
515.static ros::Time previous_time = current_time;
516.double diff_time = (current_time - previous_time).toSec();
517.
518.double diff_imu_roll = imu.angular_velocity.x * diff_time;
519.double diff_imu_pitch = imu.angular_velocity.y * diff_time;
520.double diff_imu_yaw = imu.angular_velocity.z * diff_time;
521.
522.current_pose_imu_odom.roll += diff_imu_roll;
523.current_pose_imu_odom.pitch += diff_imu_pitch;
524.current_pose_imu_odom.yaw += diff_imu_yaw;
525.
526.double diff_distance = odom.twist.twist.linear.x * diff_time;
527.offset_imu_odom_x += diff_distance*cos(-
    current_pose_imu_odom.pitch)*cos(current_pose_imu_odom.yaw);
528.offset_imu_odom_y += diff_distance*cos(-
    current_pose_imu_odom.pitch)*sin(current_pose_imu_odom.yaw);
529.offset_imu_odom_z += diff_distance*sin(-current_pose_imu_odom.pitch);
530.

```

```

531.offset_imu_odom_roll += diff_imu_roll;
532.offset_imu_odom_pitch += diff_imu_pitch;
533.offset_imu_odom_yaw += diff_imu_yaw;
534.
535.predict_pose_imu_odom.x = previous_pose.x + offset_imu_odom_x;
536.predict_pose_imu_odom.y = previous_pose.y + offset_imu_odom_y;
537.predict_pose_imu_odom.z = previous_pose.z + offset_imu_odom_z;
538.predict_pose_imu_odom.roll = previous_pose.roll + offset_imu_odom_roll;
539.predict_pose_imu_odom.pitch = previous_pose.pitch + offset_imu_odom_pitch;
540.predict_pose_imu_odom.yaw = previous_pose.yaw + offset_imu_odom_yaw;
541.
542.previous_time = current_time;
543.}
544.
545.
546.static void odom_calc(ros::Time current_time)
547.{
548.static ros::Time previous_time = current_time;
549.double diff_time = (current_time - previous_time).toSec();
550.
551.double diff_odom_roll = odom.twist.twist.angular.x * diff_time;
552.double diff_odom_pitch = odom.twist.twist.angular.y * diff_time;
553.double diff_odom_yaw = odom.twist.twist.angular.z * diff_time;
554.
555.current_pose_odom.roll += diff_odom_roll;
556.current_pose_odom.pitch += diff_odom_pitch;
557.current_pose_odom.yaw += diff_odom_yaw;
558.
559.double diff_distance = odom.twist.twist.linear.x * diff_time;
560.offset_odom_x += diff_distance*cos(-
    current_pose_odom.pitch)*cos(current_pose_odom.yaw);
561.offset_odom_y += diff_distance*cos(-
    current_pose_odom.pitch)*sin(current_pose_odom.yaw);
562.offset_odom_z += diff_distance*sin(-current_pose_odom.pitch);
563.
564.offset_odom_roll += diff_odom_roll;
565.offset_odom_pitch += diff_odom_pitch;
566.offset_odom_yaw += diff_odom_yaw;
567.

```

```

568.predict_pose_odom.x      = previous_pose.x      + offset_odom_x;
569.predict_pose_odom.y      = previous_pose.y      + offset_odom_y;
570.predict_pose_odom.z      = previous_pose.z      + offset_odom_z;
571.predict_pose_odom.roll   = previous_pose.roll   + offset_odom_roll;
572.predict_pose_odom.pitch  = previous_pose.pitch  + offset_odom_pitch;
573.predict_pose_odom.yaw    = previous_pose.yaw    + offset_odom_yaw;
574.
575.previous_time = current_time;
576.
577.}
578.
579.static void imu_calc(ros::Time current_time)
580.{
581.static ros::Time previous_time = current_time;
582.double diff_time = (current_time - previous_time).toSec();
583.
584.double diff_imu_roll  = imu.angular_velocity.x * diff_time;
585.double diff_imu_pitch = imu.angular_velocity.y * diff_time;
586.double diff_imu_yaw   = imu.angular_velocity.z * diff_time;
587.
588.current_pose_imu.roll += diff_imu_roll;
589.current_pose_imu.pitch += diff_imu_pitch;
590.current_pose_imu.yaw += diff_imu_yaw;
591.
592.double accX1 = imu.linear_acceleration.x;
593.double accY1 = std::cos(current_pose_imu.roll) * imu.linear_acceleration.y
594.-std::sin(current_pose_imu.roll) * imu.linear_acceleration.z;
595.double accZ1 = std::sin(current_pose_imu.roll) * imu.linear_acceleration.y
596.+std::cos(current_pose_imu.roll) * imu.linear_acceleration.z;
597.
598.double accX2 = std::sin(current_pose_imu.pitch) * accZ1 + std::cos(current_pose_imu.pi
    tch) * accX1;
599.double accY2 = accY1;
600.double accZ2 = std::cos(current_pose_imu.pitch) * accZ1 - std::sin(current_pose_imu.pi
    tch) * accX1;
601.
602.double accX = std::cos(current_pose_imu.yaw) * accX2 - std::sin(current_pose_imu.yaw)
    * accY2;

```

```

603.double accY = std::sin(current_pose_imu.yaw) * accX2 + std::cos(current_pose_imu.yaw)
    * accY2;

604.double accZ = accZ2;

605.

606.offset_imu_x += current_velocity_imu_x * diff_time + accX * diff_time * diff_time / 2.
    0;

607.offset_imu_y += current_velocity_imu_y * diff_time + accY * diff_time * diff_time / 2.
    0;

608.offset_imu_z += current_velocity_imu_z * diff_time + accZ * diff_time * diff_time / 2.
    0;

609.

610.current_velocity_imu_x += accX * diff_time;

611.current_velocity_imu_y += accY * diff_time;

612.current_velocity_imu_z += accZ * diff_time;

613.

614.offset_imu_roll += diff_imu_roll;

615.offset_imu_pitch += diff_imu_pitch;

616.offset_imu_yaw += diff_imu_yaw;

617.

618.predict_pose_imu.x = previous_pose.x + offset_imu_x;

619.predict_pose_imu.y = previous_pose.y + offset_imu_y;

620.predict_pose_imu.z = previous_pose.z + offset_imu_z;

621.predict_pose_imu.roll = previous_pose.roll + offset_imu_roll;

622.predict_pose_imu.pitch = previous_pose.pitch + offset_imu_pitch;

623.predict_pose_imu.yaw = previous_pose.yaw + offset_imu_yaw;

624.

625.previous_time = current_time;

626.}

627.

628.

629.static const double wrapToPm(double a_num, const double a_max)

630.{

631.if (a_num >= a_max)

632.{

633.a_num -= 2.0 * a_max;

634.}

635.return a_num;

636.}

637.

```

```

638.static const double wrapToPmPi(double a_angle_rad)
639.{
640.return wrapToPm(a_angle_rad, M_PI);
641.}
642.
643.
644.static void odom_callback(const nav_msgs::Odometry::ConstPtr& input)
645.{
646.//std::cout << __func__ << std::endl;
647.
648.odom = *input;
649.odom_calc(input->header.stamp);
650.}
651.
652.static void imuUpsideDown(const sensor_msgs::Imu::Ptr input)
653.{
654.double input_roll, input_pitch, input_yaw;
655.
656.tf::Quaternion input_orientation;
657.tf::quaternionMsgToTF(input->orientation, input_orientation);
658.tf::Matrix3x3(input_orientation).getRPY(input_roll, input_pitch, input_yaw);
659.
660.input->angular_velocity.x *= -1;
661.input->angular_velocity.y *= -1;
662.input->angular_velocity.z *= -1;
663.
664.input->linear_acceleration.x *= -1;
665.input->linear_acceleration.y *= -1;
666.input->linear_acceleration.z *= -1;
667.
668.input_roll *= -1;
669.input_pitch *= -1;
670.input_yaw *= -1;
671.
672.input->
    >orientation = tf::createQuaternionMsgFromRollPitchYaw(input_roll, input_pitch, input_
        yaw);
673.}
674.

```

```

675. static void imu_callback(const sensor_msgs::Imu::Ptr& input)
676. {
677. //std::cout << __func__ << std::endl;
678.
679. if(_imu_upside_down)
680. imuUpsideDown(input);
681.
682. const ros::Time current_time = input->header.stamp;
683. static ros::Time previous_time = current_time;
684. const double diff_time = (current_time - previous_time).toSec();
685.
686. double imu_roll, imu_pitch, imu_yaw;
687. tf::Quaternion imu_orientation;
688. tf::quaternionMsgToTF(input->orientation, imu_orientation);
689. tf::Matrix3x3(imu_orientation).getRPY(imu_roll, imu_pitch, imu_yaw);
690.
691. imu_roll = wrapToPmPi(imu_roll);
692. imu_pitch = wrapToPmPi(imu_pitch);
693. imu_yaw = wrapToPmPi(imu_yaw);
694.
695. static double previous_imu_roll = imu_roll, previous_imu_pitch = imu_pitch, previous_i
    mu_yaw = imu_yaw;
696. const double diff_imu_roll = imu_roll - previous_imu_roll;
697.
698. const double diff_imu_pitch = imu_pitch - previous_imu_pitch;
699.
700. double diff_imu_yaw;
701. if(fabs(imu_yaw - previous_imu_yaw) > M_PI)
702. {
703. if(imu_yaw > 0)
704. diff_imu_yaw = (imu_yaw - previous_imu_yaw) - M_PI*2;
705. else
706. diff_imu_yaw = -M_PI*2 - (imu_yaw - previous_imu_yaw);
707. }
708. else
709. diff_imu_yaw = imu_yaw - previous_imu_yaw;
710.
711. imu.header = input->header;
712. imu.linear_acceleration.x = input->linear_acceleration.x;

```

```

713.//imu.linear_acceleration.y = input->linear_acceleration.y;
714.//imu.linear_acceleration.z = input->linear_acceleration.z;
715.imu.linear_acceleration.y = 0;
716.imu.linear_acceleration.z = 0;
717.
718.if(diff_time != 0)
719.{
720.imu.angular_velocity.x = diff_imu_roll / diff_time;
721.imu.angular_velocity.y = diff_imu_pitch / diff_time;
722.imu.angular_velocity.z = diff_imu_yaw / diff_time;
723.}
724.else
725.{
726.imu.angular_velocity.x = 0;
727.imu.angular_velocity.y = 0;
728.imu.angular_velocity.z = 0;
729.}
730.
731.imu_calc(input->header.stamp);
732.
733.previous_time = current_time;
734.previous_imu_roll = imu_roll;
735.previous_imu_pitch = imu_pitch;
736.previous_imu_yaw = imu_yaw;
737.}
738.
739.static void points_callback(const sensor_msgs::PointCloud2::ConstPtr& input)
740.{
741.if (map_loaded == 1 && init_pos_set == 1)
742.{
743.matching_start = std::chrono::system_clock::now();
744.
745.static tf::TransformBroadcaster br;
746.tf::Transform transform;
747.tf::Quaternion predict_q, ndt_q, current_q, localizer_q;
748.
749.pcl::PointXYZ p;
750.pcl::PointCloud<pcl::PointXYZ> filtered_scan;
751.

```

```

752.current_scan_time = input->header.stamp;
753.
754.pcl::fromROSMsg(*input, filtered_scan);
755.pcl::PointCloud<pcl::PointXYZ>::Ptr filtered_scan_ptr(new pcl::PointCloud<pcl::PointXYZ>(filtered_scan));
756.int scan_points_num = filtered_scan_ptr->size();
757.
758.Eigen::Matrix4f t(Eigen::Matrix4f::Identity()); // base_link
759.Eigen::Matrix4f t2(Eigen::Matrix4f::Identity()); // localizer
760.
761.std::chrono::time_point<std::chrono::system_clock> align_start, align_end, getFitnessScore_start,
    core_start,
762.getFitnessScore_end;
763.static double align_time, getFitnessScore_time = 0.0;
764.
765.// Setting point cloud to be aligned.
766.ndt.setInputSource(filtered_scan_ptr);
767.
768.// Guess the initial gross estimation of the transformation
769.predict_pose.x = previous_pose.x + offset_x;
770.predict_pose.y = previous_pose.y + offset_y;
771.predict_pose.z = previous_pose.z + offset_z;
772.predict_pose.roll = previous_pose.roll;
773.predict_pose.pitch = previous_pose.pitch;
774.predict_pose.yaw = previous_pose.yaw + offset_yaw;
775.
776.
777.if (_use_imu == true && _use_odom == true)
778.imu_odom_calc(current_scan_time);
779.if(_use_imu == true && _use_odom == true)
780.imu_calc(current_scan_time);
781.if (_use_imu == false && _use_odom == true)
782.odom_calc(current_scan_time);
783.
784.pose predict_pose_for_ndt;
785.if (_use_imu == true && _use_odom == true)
786.predict_pose_for_ndt = predict_pose_imu_odom;
787.else if (_use_imu == true && _use_odom == false)
788.predict_pose_for_ndt = predict_pose_imu;

```

```

789.else if (_use_imu == false && _use_odom == true)
790.predict_pose_for_ndt = predict_pose_odom;
791.else
792.predict_pose_for_ndt = predict_pose;
793.
794.Eigen::Translation3f init_translation(predict_pose_for_ndt.x, predict_pose_for_ndt.y,
    predict_pose_for_ndt.z);
795.Eigen::AngleAxisf init_rotation_x(predict_pose_for_ndt.roll, Eigen::Vector3f::UnitX())
    ;
796.Eigen::AngleAxisf init_rotation_y(predict_pose_for_ndt.pitch, Eigen::Vector3f::UnitY()
    );
797.Eigen::AngleAxisf init_rotation_z(predict_pose_for_ndt.yaw, Eigen::Vector3f::UnitZ());
798.Eigen::Matrix4f init_guess = (init_translation * init_rotation_z * init_rotation_y * i
    nit_rotation_x) * tf_btol;
799.
800.
801.
802.
803.pcl::PointCloud<pcl::PointXYZ>::Ptr output_cloud(new pcl::PointCloud<pcl::PointXYZ>);
804.#ifdef USE_FAST_PCL
805.if (_use_openmp == true)
806.{
807.align_start = std::chrono::system_clock::now();
808.ndt.omp_align(*output_cloud, init_guess);
809.align_end = std::chrono::system_clock::now();
810.}
811.else
812.{
813.#endif
814.align_start = std::chrono::system_clock::now();
815.ndt.align(*output_cloud, init_guess);
816.align_end = std::chrono::system_clock::now();
817.#ifdef USE_FAST_PCL
818.}
819.#endif
820.
821.align_time = std::chrono::duration_cast<std::chrono::microseconds>(align_end - align_s
    tart).count() / 1000.0;
822.

```

```

823.
824.t = ndt.getFinalTransformation(); // localizer
825.t2 = t * tf_ltob;                // base_link
826.
827.iteration = ndt.getFinalNumIteration();
828.#ifdef USE_FAST_PCL
829.if (_use_omp == true)
830.{
831.getFitnessScore_start = std::chrono::system_clock::now();
832.fitness_score = ndt.omp_getFitnessScore();
833.getFitnessScore_end = std::chrono::system_clock::now();
834.}
835.else
836.{
837.#endif
838.getFitnessScore_start = std::chrono::system_clock::now();
839.fitness_score = ndt.getFitnessScore();
840.getFitnessScore_end = std::chrono::system_clock::now();
841.#ifdef USE_FAST_PCL
842.}
843.#endif
844.getFitnessScore_time =
845.std::chrono::duration_cast<std::chrono::microseconds>(getFitnessScore_end - getFitness
    Score_start).count() /
846.1000.0;
847.
848.
849.trans_probability = ndt.getTransformationProbability();
850.
851.tf::Matrix3x3 mat_l; // localizer
852.mat_l.setValue(static_cast<double>(t(0, 0)), static_cast<double>(t(0, 1)), static_cast
    <double>(t(0, 2)),
853.static_cast<double>(t(1, 0)), static_cast<double>(t(1, 1)), static_cast<double>(t(1, 2
    )),
854.static_cast<double>(t(2, 0)), static_cast<double>(t(2, 1)), static_cast<double>(t(2, 2
    )));
855.
856.// Update localizer_pose
857.localizer_pose.x = t(0, 3);

```

```

858.localizer_pose.y = t(1, 3);
859.localizer_pose.z = t(2, 3);
860.mat_l.getRPY(localizer_pose.roll, localizer_pose.pitch, localizer_pose.yaw, 1);
861.
862.tf::Matrix3x3 mat_b; // base_link
863.mat_b.setValue(static_cast<double>(t2(0, 0)), static_cast<double>(t2(0, 1)), static_cast<double>(t2(0, 2)),
static_cast<double>(t2(1, 0)), static_cast<double>(t2(1, 1)), static_cast<double>(t2(1, 2)),
static_cast<double>(t2(2, 0)), static_cast<double>(t2(2, 1)), static_cast<double>(t2(2, 2))));
866.
867.// Update ndt_pose
868.ndt_pose.x = t2(0, 3);
869.ndt_pose.y = t2(1, 3);
870.ndt_pose.z = t2(2, 3);
871.mat_b.getRPY(ndt_pose.roll, ndt_pose.pitch, ndt_pose.yaw, 1);
872.
873.// Calculate the difference between ndt_pose and predict_pose
874.predict_pose_error = sqrt((ndt_pose.x - predict_pose_for_ndt.x) * (ndt_pose.x - predict_pose_for_ndt.x) +
(ndt_pose.y - predict_pose_for_ndt.y) * (ndt_pose.y - predict_pose_for_ndt.y) +
(ndt_pose.z - predict_pose_for_ndt.z) * (ndt_pose.z - predict_pose_for_ndt.z));
877.
878.if (predict_pose_error <= PREDICT_POSE_THRESHOLD)
879.{
880.use_predict_pose = 0;
881.}
882.else
883.{
884.use_predict_pose = 1;
885.}
886.use_predict_pose = 0;
887.
888.
889.if (use_predict_pose == 0)
890.{
891.current_pose.x = ndt_pose.x;
892.current_pose.y = ndt_pose.y;

```

```
893.current_pose.z = ndt_pose.z;
894.current_pose.roll = ndt_pose.roll;
895.current_pose.pitch = ndt_pose.pitch;
896.current_pose.yaw = ndt_pose.yaw;
897.}
898.else
899.{
900.current_pose.x = predict_pose_for_ndt.x;
901.current_pose.y = predict_pose_for_ndt.y;
902.current_pose.z = predict_pose_for_ndt.z;
903.current_pose.roll = predict_pose_for_ndt.roll;
904.current_pose.pitch = predict_pose_for_ndt.pitch;
905.current_pose.yaw = predict_pose_for_ndt.yaw;
906.}
907.
908.
909.// Compute the velocity and acceleration
910.scan_duration = current_scan_time - previous_scan_time;
911.double secs = scan_duration.toSec();
912.diff_x = current_pose.x - previous_pose.x;
913.diff_y = current_pose.y - previous_pose.y;
914.diff_z = current_pose.z - previous_pose.z;
915.diff_yaw = current_pose.yaw - previous_pose.yaw;
916.diff = sqrt(diff_x * diff_x + diff_y * diff_y + diff_z * diff_z);
917.
918.current_velocity = diff / secs;
919.current_velocity_x = diff_x / secs;
920.current_velocity_y = diff_y / secs;
921.current_velocity_z = diff_z / secs;
922.angular_velocity = diff_yaw / secs;
923.
924.current_pose_imu.x = current_pose.x;
925.current_pose_imu.y = current_pose.y;
926.current_pose_imu.z = current_pose.z;
927.current_pose_imu.roll = current_pose.roll;
928.current_pose_imu.pitch = current_pose.pitch;
929.current_pose_imu.yaw = current_pose.yaw;
930.
931.current_velocity_imu_x = current_velocity_x;
```

```

932.current_velocity_imu_y = current_velocity_y;
933.current_velocity_imu_z = current_velocity_z;
934.
935.
936.current_pose_odom.x = current_pose.x;
937.current_pose_odom.y = current_pose.y;
938.current_pose_odom.z = current_pose.z;
939.current_pose_odom.roll = current_pose.roll;
940.current_pose_odom.pitch = current_pose.pitch;
941.current_pose_odom.yaw = current_pose.yaw;
942.
943.current_pose_imu_odom.x = current_pose.x;
944.current_pose_imu_odom.y = current_pose.y;
945.current_pose_imu_odom.z = current_pose.z;
946.current_pose_imu_odom.roll = current_pose.roll;
947.current_pose_imu_odom.pitch = current_pose.pitch;
948.current_pose_imu_odom.yaw = current_pose.yaw;
949.
950.current_velocity_smooth = (current_velocity + previous_velocity + previous_previous_velocity) / 3.0;
951.if (current_velocity_smooth < 0.2)
952.{
953.current_velocity_smooth = 0.0;
954.}
955.
956.current_accel = (current_velocity - previous_velocity) / secs;
957.current_accel_x = (current_velocity_x - previous_velocity_x) / secs;
958.current_accel_y = (current_velocity_y - previous_velocity_y) / secs;
959.current_accel_z = (current_velocity_z - previous_velocity_z) / secs;
960.
961.estimated_vel_mps.data = current_velocity;
962.estimated_vel_kmph.data = current_velocity * 3.6;
963.
964.estimated_vel_mps_pub.publish(estimated_vel_mps);
965.estimated_vel_kmph_pub.publish(estimated_vel_kmph);
966.
967.// Set values for publishing pose
968.predict_q.setRPY(predict_pose.roll, predict_pose.pitch, predict_pose.yaw);
969.if (_use_local_transform == true)

```

```

970.{
971.tf::Vector3 v(predict_pose.x, predict_pose.y, predict_pose.z);
972.tf::Transform transform(predict_q, v);
973.predict_pose_msg.header.frame_id = "/map";
974.predict_pose_msg.header.stamp = current_scan_time;
975.predict_pose_msg.pose.position.x = (local_transform * transform).getOrigin().getX();
976.predict_pose_msg.pose.position.y = (local_transform * transform).getOrigin().getY();
977.predict_pose_msg.pose.position.z = (local_transform * transform).getOrigin().getZ();
978.predict_pose_msg.pose.orientation.x = (local_transform * transform).getRotation().x();
979.predict_pose_msg.pose.orientation.y = (local_transform * transform).getRotation().y();
980.predict_pose_msg.pose.orientation.z = (local_transform * transform).getRotation().z();
981.predict_pose_msg.pose.orientation.w = (local_transform * transform).getRotation().w();
982.}
983.else
984.{
985.predict_pose_msg.header.frame_id = "/map";
986.predict_pose_msg.header.stamp = current_scan_time;
987.predict_pose_msg.pose.position.x = predict_pose.x;
988.predict_pose_msg.pose.position.y = predict_pose.y;
989.predict_pose_msg.pose.position.z = predict_pose.z;
990.predict_pose_msg.pose.orientation.x = predict_q.x();
991.predict_pose_msg.pose.orientation.y = predict_q.y();
992.predict_pose_msg.pose.orientation.z = predict_q.z();
993.predict_pose_msg.pose.orientation.w = predict_q.w();
994.}
995.
996.tf::Quaternion predict_q_imu;
997.predict_q_imu.setRPY(predict_pose_imu.roll, predict_pose_imu.pitch, predict_pose_imu.yaw);
998.predict_pose_imu_msg.header.frame_id = "map";
999.predict_pose_imu_msg.header.stamp = input->header.stamp;
1000.    predict_pose_imu_msg.pose.position.x = predict_pose_imu.x;
1001.    predict_pose_imu_msg.pose.position.y = predict_pose_imu.y;
1002.    predict_pose_imu_msg.pose.position.z = predict_pose_imu.z;
1003.    predict_pose_imu_msg.pose.orientation.x = predict_q_imu.x();
1004.    predict_pose_imu_msg.pose.orientation.y = predict_q_imu.y();
1005.    predict_pose_imu_msg.pose.orientation.z = predict_q_imu.z();
1006.    predict_pose_imu_msg.pose.orientation.w = predict_q_imu.w();
1007.    predict_pose_imu_pub.publish(predict_pose_imu_msg);

```

```

1008.
1009.     tf::Quaternion predict_q_odom;
1010.     predict_q_odom.setRPY(predict_pose_odom.roll, predict_pose_odom.pitch, predict
        _pose_odom.yaw);
1011.     predict_pose_odom_msg.header.frame_id = "map";
1012.     predict_pose_odom_msg.header.stamp = input->header.stamp;
1013.     predict_pose_odom_msg.pose.position.x = predict_pose_odom.x;
1014.     predict_pose_odom_msg.pose.position.y = predict_pose_odom.y;
1015.     predict_pose_odom_msg.pose.position.z = predict_pose_odom.z;
1016.     predict_pose_odom_msg.pose.orientation.x = predict_q_odom.x();
1017.     predict_pose_odom_msg.pose.orientation.y = predict_q_odom.y();
1018.     predict_pose_odom_msg.pose.orientation.z = predict_q_odom.z();
1019.     predict_pose_odom_msg.pose.orientation.w = predict_q_odom.w();
1020.     predict_pose_odom_pub.publish(predict_pose_odom_msg);
1021.
1022.
1023.     tf::Quaternion predict_q_imu_odom;
1024.     predict_q_imu_odom.setRPY(predict_pose_imu_odom.roll, predict_pose_imu_odom.pi
        tch, predict_pose_imu_odom.yaw);
1025.     predict_pose_imu_odom_msg.header.frame_id = "map";
1026.     predict_pose_imu_odom_msg.header.stamp = input->header.stamp;
1027.     predict_pose_imu_odom_msg.pose.position.x = predict_pose_imu_odom.x;
1028.     predict_pose_imu_odom_msg.pose.position.y = predict_pose_imu_odom.y;
1029.     predict_pose_imu_odom_msg.pose.position.z = predict_pose_imu_odom.z;
1030.     predict_pose_imu_odom_msg.pose.orientation.x = predict_q_imu_odom.x();
1031.     predict_pose_imu_odom_msg.pose.orientation.y = predict_q_imu_odom.y();
1032.     predict_pose_imu_odom_msg.pose.orientation.z = predict_q_imu_odom.z();
1033.     predict_pose_imu_odom_msg.pose.orientation.w = predict_q_imu_odom.w();
1034.     predict_pose_imu_odom_pub.publish(predict_pose_imu_odom_msg);
1035.
1036.     ndt_q.setRPY(ndt_pose.roll, ndt_pose.pitch, ndt_pose.yaw);
1037.     if (_use_local_transform == true)
1038.     {
1039.         tf::Vector3 v(ndt_pose.x, ndt_pose.y, ndt_pose.z);
1040.         tf::Transform transform(ndt_q, v);
1041.         ndt_pose_msg.header.frame_id = "/map";
1042.         ndt_pose_msg.header.stamp = current_scan_time;
1043.         ndt_pose_msg.pose.position.x = (local_transform * transform).getOrigin().getX(
        );

```

```

1044.     ndt_pose_msg.pose.position.y = (local_transform * transform).getOrigin().getY(
        );
1045.     ndt_pose_msg.pose.position.z = (local_transform * transform).getOrigin().getZ(
        );
1046.     ndt_pose_msg.pose.orientation.x = (local_transform * transform).getRotation().
        x();
1047.     ndt_pose_msg.pose.orientation.y = (local_transform * transform).getRotation().
        y();
1048.     ndt_pose_msg.pose.orientation.z = (local_transform * transform).getRotation().
        z();
1049.     ndt_pose_msg.pose.orientation.w = (local_transform * transform).getRotation().
        w();
1050. }
1051. else
1052. {
1053.     ndt_pose_msg.header.frame_id = "/map";
1054.     ndt_pose_msg.header.stamp = current_scan_time;
1055.     ndt_pose_msg.pose.position.x = ndt_pose.x;
1056.     ndt_pose_msg.pose.position.y = ndt_pose.y;
1057.     ndt_pose_msg.pose.position.z = ndt_pose.z;
1058.     ndt_pose_msg.pose.orientation.x = ndt_q.x();
1059.     ndt_pose_msg.pose.orientation.y = ndt_q.y();
1060.     ndt_pose_msg.pose.orientation.z = ndt_q.z();
1061.     ndt_pose_msg.pose.orientation.w = ndt_q.w();
1062. }
1063.
1064.     current_q.setRPY(current_pose.roll, current_pose.pitch, current_pose.yaw);
1065.     // current_pose is published by vel_pose_mux
1066.     /*
1067.     current_pose_msg.header.frame_id = "/map";
1068.     current_pose_msg.header.stamp = current_scan_time;
1069.     current_pose_msg.pose.position.x = current_pose.x;
1070.     current_pose_msg.pose.position.y = current_pose.y;
1071.     current_pose_msg.pose.position.z = current_pose.z;
1072.     current_pose_msg.pose.orientation.x = current_q.x();
1073.     current_pose_msg.pose.orientation.y = current_q.y();
1074.     current_pose_msg.pose.orientation.z = current_q.z();
1075.     current_pose_msg.pose.orientation.w = current_q.w();
1076.     */

```

```
1077.
1078.     localizer_q.setRPY(localizer_pose.roll, localizer_pose.pitch, localizer_pose.y
    aw);
1079.     if (_use_local_transform == true)
1080.     {
1081.         tf::Vector3 v(localizer_pose.x, localizer_pose.y, localizer_pose.z);
1082.         tf::Transform transform(localizer_q, v);
1083.         localizer_pose_msg.header.frame_id = "/map";
1084.         localizer_pose_msg.header.stamp = current_scan_time;
1085.         localizer_pose_msg.pose.position.x = (local_transform * transform).getOrigin()
    .getX();
1086.         localizer_pose_msg.pose.position.y = (local_transform * transform).getOrigin()
    .getY();
1087.         localizer_pose_msg.pose.position.z = (local_transform * transform).getOrigin()
    .getZ();
1088.         localizer_pose_msg.pose.orientation.x = (local_transform * transform).getRotat
    ion().x();
1089.         localizer_pose_msg.pose.orientation.y = (local_transform * transform).getRotat
    ion().y();
1090.         localizer_pose_msg.pose.orientation.z = (local_transform * transform).getRotat
    ion().z();
1091.         localizer_pose_msg.pose.orientation.w = (local_transform * transform).getRotat
    ion().w();
1092.     }
1093.     else
1094.     {
1095.         localizer_pose_msg.header.frame_id = "/map";
1096.         localizer_pose_msg.header.stamp = current_scan_time;
1097.         localizer_pose_msg.pose.position.x = localizer_pose.x;
1098.         localizer_pose_msg.pose.position.y = localizer_pose.y;
1099.         localizer_pose_msg.pose.position.z = localizer_pose.z;
1100.         localizer_pose_msg.pose.orientation.x = localizer_q.x();
1101.         localizer_pose_msg.pose.orientation.y = localizer_q.y();
1102.         localizer_pose_msg.pose.orientation.z = localizer_q.z();
1103.         localizer_pose_msg.pose.orientation.w = localizer_q.w();
1104.     }
1105.
1106.     predict_pose_pub.publish(predict_pose_msg);
1107.     ndt_pose_pub.publish(ndt_pose_msg);
```

```

1108.     // current_pose is published by vel_pose_mux
1109.     //     current_pose_pub.publish(current_pose_msg);
1110.     localizer_pose_pub.publish(localizer_pose_msg);
1111.
1112.     //Add by WEN WS
1113.     ndtPath.header.frame_id="map";
1114.     ndtPath.header.stamp=ndt_pose_msg.header.stamp;
1115.     ndtPath.poses.push_back(ndt_pose_msg);
1116.     pubNdtPath.publish(ndtPath);
1117.
1118.     // Send TF "/base_link" to "/map"
1119.     transform.setOrigin(tf::Vector3(current_pose.x, current_pose.y, current_pose.z
    ));
1120.     transform.setRotation(current_q);
1121.     //     br.sendTransform(tf::StampedTransform(transform, current_scan_time, "/ma
    p", "/base_link"));
1122.     if (_use_local_transform == true)
1123.     {
1124.         br.sendTransform(tf::StampedTransform(local_transform * transform, current_sca
            n_time, "/map", "/base_link"));
1125.     }
1126.     else
1127.     {
1128.         br.sendTransform(tf::StampedTransform(transform, current_scan_time, "/map", "/"
            base_link"));
1129.     }
1130.
1131.     matching_end = std::chrono::system_clock::now();
1132.     exe_time = std::chrono::duration_cast<std::chrono::microseconds>(matching_end
        - matching_start).count() / 1000.0;
1133.     time_ndt_matching.data = exe_time;
1134.     time_ndt_matching_pub.publish(time_ndt_matching);
1135.
1136.     // Set values for /estimate_twist
1137.     estimate_twist_msg.header.stamp = current_scan_time;
1138.     estimate_twist_msg.header.frame_id = "/base_link";
1139.     estimate_twist_msg.twist.linear.x = current_velocity;
1140.     estimate_twist_msg.twist.linear.y = 0.0;
1141.     estimate_twist_msg.twist.linear.z = 0.0;

```

```

1142.     estimate_twist_msg.twist.angular.x = 0.0;
1143.     estimate_twist_msg.twist.angular.y = 0.0;
1144.     estimate_twist_msg.twist.angular.z = angular_velocity;
1145.
1146.     estimate_twist_pub.publish(estimate_twist_msg);
1147.
1148.     geometry_msgs::Vector3Stamped estimate_vel_msg;
1149.     estimate_vel_msg.header.stamp = current_scan_time;
1150.     estimate_vel_msg.vector.x = current_velocity;
1151.     estimated_vel_pub.publish(estimate_vel_msg);
1152.
1153.     // Set values for /ndt_stat
1154.     ndt_stat_msg.header.stamp = current_scan_time;
1155.     ndt_stat_msg.exe_time = time_ndt_matching.data;
1156.     ndt_stat_msg.iteration = iteration;
1157.     ndt_stat_msg.score = fitness_score;
1158.     ndt_stat_msg.velocity = current_velocity;
1159.     ndt_stat_msg.acceleration = current_accel;
1160.     ndt_stat_msg.use_predict_pose = 0;
1161.
1162.     ndt_stat_pub.publish(ndt_stat_msg);
1163.
1164.     /* Compute NDT_Reliability */
1165.     ndt_reliability.data = Wa * (exe_time / 100.0) * 100.0 + Wb * (iteration / 10.
        0) * 100.0 +
1166.     Wc * ((2.0 - trans_probability) / 2.0) * 100.0;
1167.     ndt_reliability_pub.publish(ndt_reliability);
1168.
1169.
1170.
1171.
1172.     // Write log
1173.     if (!ofs)
1174.     {
1175.         std::cerr << "Could not open " << filename << "." << std::endl;
1176.         exit(1);
1177.     }
1178.     static ros::Time start_time = input->header.stamp;
1179.

```

```

1180.         ofs << input-
            >header.seq << "," << scan_points_num << "," << step_size << "," << trans_eps << "," <
            < std::fixed
1181.         << std::setprecision(5) << current_pose.x << "," << std::fixed << std::setprec
            ision(5) << current_pose.y << ","
1182.         << std::fixed << std::setprecision(5) << current_pose.z << "," << current_pose
            .roll << "," << current_pose.pitch
1183.         << "," << current_pose.yaw << "," << predict_pose.x << "," << predict_pose.y <
            < "," << predict_pose.z << ","
1184.         << predict_pose.roll << "," << predict_pose.pitch << "," << predict_pose.yaw <
            < ","
1185.         << current_pose.x - predict_pose.x << "," << current_pose.y - predict_pose.y <
            < ","
1186.         << current_pose.z - predict_pose.z << "," << current_pose.roll - predict_pose.
            roll << ","
1187.         << current_pose.pitch - predict_pose.pitch << "," << current_pose.yaw - predic
            t_pose.yaw << ","
1188.         << predict_pose_error << "," << iteration << "," << fitness_score << "," << tr
            ans_probability << ","
1189.         << ndt_reliability.data << "," << current_velocity << "," << current_velocity_
            smooth << "," << current_accel
1190.         << "," << angular_velocity << "," << time_ndt_matching.data << "," << align_ti
            me << "," << getFitnessScore_time
1191.         << std::endl;
1192.
1193.         std::cout << "-----
            -" << std::endl;
1194.         std::cout << "Sequence: " << input->header.seq << std::endl;
1195.         std::cout << "Timestamp: " << input->header.stamp << std::endl;
1196.         std::cout << "Frame ID: " << input->header.frame_id << std::endl;
1197.         //         std::cout << "Number of Scan Points: " << scan_ptr-
            >size() << " points." << std::endl;
1198.         std::cout << "Number of Filtered Scan Points: " << scan_points_num << " points
            ." << std::endl;
1199.         std::cout << "NDT has converged: " << ndt.hasConverged() << std::endl;
1200.         std::cout << "Fitness Score: " << fitness_score << std::endl;
1201.         std::cout << "Transformation Probability: " << ndt.getTransformationProbabilit
            y() << std::endl;
1202.         std::cout << "Execution Time: " << exe_time << " ms." << std::endl;

```

```

1203.         std::cout << "Number of Iterations: " << ndt.getFinalNumIteration() << std::endl;
1204.         std::cout << "NDT Reliability: " << ndt_reliability.data << std::endl;
1205.         std::cout << "(x,y,z,roll,pitch,yaw): " << std::endl;
1206.         std::cout << "(" << current_pose.x << ", " << current_pose.y << ", " << current_pose.z << ", " << current_pose.roll
1207.         << ", " << current_pose.pitch << ", " << current_pose.yaw << ")" << std::endl;
1208.         std::cout << "Transformation Matrix: " << std::endl;
1209.         std::cout << t << std::endl;
1210.         std::cout << "-----
-" << std::endl;
1211.
1212.         // Update offset
1213.         if (_offset == "linear")
1214.         {
1215.             offset_x = diff_x;
1216.             offset_y = diff_y;
1217.             offset_z = diff_z;
1218.             offset_yaw = diff_yaw;
1219.         }
1220.         else if (_offset == "quadratic")
1221.         {
1222.             offset_x = (current_velocity_x + current_accel_x * secs) * secs;
1223.             offset_y = (current_velocity_y + current_accel_y * secs) * secs;
1224.             offset_z = diff_z;
1225.             offset_yaw = diff_yaw;
1226.         }
1227.         else if (_offset == "zero")
1228.         {
1229.             offset_x = 0.0;
1230.             offset_y = 0.0;
1231.             offset_z = 0.0;
1232.             offset_yaw = 0.0;
1233.         }
1234.
1235.         offset_imu_x = 0.0;
1236.         offset_imu_y = 0.0;
1237.         offset_imu_z = 0.0;
1238.         offset_imu_roll = 0.0;

```

```
1239.     offset_imu_pitch = 0.0;
1240.     offset_imu_yaw = 0.0;
1241.
1242.     offset_odom_x = 0.0;
1243.     offset_odom_y = 0.0;
1244.     offset_odom_z = 0.0;
1245.     offset_odom_roll = 0.0;
1246.     offset_odom_pitch = 0.0;
1247.     offset_odom_yaw = 0.0;
1248.
1249.     offset_imu_odom_x = 0.0;
1250.     offset_imu_odom_y = 0.0;
1251.     offset_imu_odom_z = 0.0;
1252.     offset_imu_odom_roll = 0.0;
1253.     offset_imu_odom_pitch = 0.0;
1254.     offset_imu_odom_yaw = 0.0;
1255.
1256.     // Update previous***
1257.     previous_pose.x = current_pose.x;
1258.     previous_pose.y = current_pose.y;
1259.     previous_pose.z = current_pose.z;
1260.     previous_pose.roll = current_pose.roll;
1261.     previous_pose.pitch = current_pose.pitch;
1262.     previous_pose.yaw = current_pose.yaw;
1263.
1264.     previous_scan_time.sec = current_scan_time.sec;
1265.     previous_scan_time.nsec = current_scan_time.nsec;
1266.
1267.     previous_previous_velocity = previous_velocity;
1268.     previous_velocity = current_velocity;
1269.     previous_velocity_x = current_velocity_x;
1270.     previous_velocity_y = current_velocity_y;
1271.     previous_velocity_z = current_velocity_z;
1272.     previous_accel = current_accel;
1273.
1274.     previous_estimated_vel_kmph.data = estimated_vel_kmph.data;
1275. }
1276. }
1277.
```

```

1278.     static void navsat_odom_sub_callback(const nav_msgs::Odometry::ConstPtr& msg)
1279.     {
1280.         navsatOdomPoseOdom=*msg;
1281.         navsatOdomPose.pose.position.x=navsatOdomPoseOdom.pose.pose.position.x;
1282.         navsatOdomPose.pose.position.y=navsatOdomPoseOdom.pose.pose.position.y;
1283.         navsatOdomPose.pose.position.z=navsatOdomPoseOdom.pose.pose.position.z;
1284.         navsatOdomPose.pose.orientation.x=navsatOdomPoseOdom.pose.pose.orientation.x;
1285.         navsatOdomPose.pose.orientation.y=navsatOdomPoseOdom.pose.pose.orientation.y;
1286.         navsatOdomPose.pose.orientation.z=navsatOdomPoseOdom.pose.pose.orientation.z;
1287.         navsatOdomPose.pose.orientation.w=navsatOdomPoseOdom.pose.pose.orientation.w;
1288.
1289.         if(firstSpancptOdomMessage==0)
1290.         {
1291.             firstSpancptOdomMessageX=navsatOdomPoseOdom.pose.pose.position.x;
1292.             firstSpancptOdomMessageY=navsatOdomPoseOdom.pose.pose.position.y;
1293.             firstSpancptOdomMessageZ=navsatOdomPoseOdom.pose.pose.position.z;
1294.             firstSpancptOdomMessage=1;
1295.         }
1296.         if(firstSpancptOdomMessage==1)
1297.         {
1298.             navsatOdomPose.pose.position.x=navsatOdomPose.pose.position.x-
                firstSpancptOdomMessageX;
1299.             navsatOdomPose.pose.position.y=navsatOdomPose.pose.position.y-
                firstSpancptOdomMessageY;
1300.             navsatOdomPose.pose.position.z=navsatOdomPose.pose.position.z-
                firstSpancptOdomMessageZ;
1301.             spancptPath.header.frame_id="map";
1302.             //spancptPath.header.stamp=navsatOdomPoseOdom.header.stamp;
1303.             spancptPath.poses.push_back(navsatOdomPose);
1304.             pubspancptPath.publish(spancptPath);
1305.         }
1306.         std::cout << "firstSpancptOdomMessage: " <<firstSpancptOdomMessage<< std::endl
            ;
1307.
1308.     }
1309.
1310.     int main(int argc, char** argv)
1311.     {
1312.         ros::init(argc, argv, "ndt_matching");

```

```

1313.
1314.     ros::NodeHandle nh;
1315.     ros::NodeHandle private_nh("~");
1316.
1317.     // Set log file name.
1318.     char buffer[80];
1319.     std::time_t now = std::time(NULL);
1320.     std::tm* pnow = std::localtime(&now);
1321.     std::strftime(buffer, 80, "%Y%m%d_%H%M%S", pnow);
1322.     filename = "ndt_matching_" + std::string(buffer) + ".csv";
1323.     ofs.open(filename.c_str(), std::ios::app);
1324.
1325.     // Getting parameters
1326.     private_nh.getParam("use_gnss", _use_gnss);
1327.     private_nh.getParam("queue_size", _queue_size);
1328.     private_nh.getParam("offset", _offset);
1329.     private_nh.getParam("use_openmp", _use_openmp);
1330.     private_nh.getParam("get_height", _get_height);
1331.     private_nh.getParam("use_local_transform", _use_local_transform);
1332.     private_nh.getParam("use_imu", _use_imu);
1333.     private_nh.getParam("use_odom", _use_odom);
1334.     private_nh.getParam("imu_upside_down", _imu_upside_down);
1335.     private_nh.getParam("imu_topic", _imu_topic);
1336.
1337.     if (nh.getParam("localizer", _localizer) == false)
1338.     {
1339.         std::cout << "localizer is not set." << std::endl;
1340.         return 1;
1341.     }
1342.     if (nh.getParam("tf_x", _tf_x) == false)
1343.     {
1344.         std::cout << "tf_x is not set." << std::endl;
1345.         return 1;
1346.     }
1347.     if (nh.getParam("tf_y", _tf_y) == false)
1348.     {
1349.         std::cout << "tf_y is not set." << std::endl;
1350.         return 1;
1351.     }

```

```

1352.     if (nh.getParam("tf_z", _tf_z) == false)
1353.     {
1354.         std::cout << "tf_z is not set." << std::endl;
1355.         return 1;
1356.     }
1357.     if (nh.getParam("tf_roll", _tf_roll) == false)
1358.     {
1359.         std::cout << "tf_roll is not set." << std::endl;
1360.         return 1;
1361.     }
1362.     if (nh.getParam("tf_pitch", _tf_pitch) == false)
1363.     {
1364.         std::cout << "tf_pitch is not set." << std::endl;
1365.         return 1;
1366.     }
1367.     if (nh.getParam("tf_yaw", _tf_yaw) == false)
1368.     {
1369.         std::cout << "tf_yaw is not set." << std::endl;
1370.         return 1;
1371.     }
1372.
1373.     std::cout << "-----
    -" << std::endl;
1374.     std::cout << "Log file: " << filename << std::endl;
1375.     std::cout << "use_gnss: " << _use_gnss << std::endl;
1376.     std::cout << "queue_size: " << _queue_size << std::endl;
1377.     std::cout << "offset: " << _offset << std::endl;
1378.     std::cout << "use_openmp: " << _use_openmp << std::endl;
1379.     std::cout << "get_height: " << _get_height << std::endl;
1380.     std::cout << "use_local_transform: " << _use_local_transform << std::endl;
1381.     std::cout << "use_imu: " << _use_imu << std::endl;
1382.     std::cout << "use_odom: " << _use_odom << std::endl;
1383.     std::cout << "imu_upside_down: " << _imu_upside_down << std::endl;
1384.     std::cout << "localizer: " << _localizer << std::endl;
1385.     std::cout << "imu_topic: " << _imu_topic << std::endl;
1386.     std::cout << "(tf_x,tf_y,tf_z,tf_roll,tf_pitch,tf_yaw): (" << _tf_x << ", " <<
        _tf_y << ", " << _tf_z << ", "
1387.         << _tf_roll << ", " << _tf_pitch << ", " << _tf_yaw << ")" << std::endl;

```

```

1388.         std::cout << "-----
        -" << std::endl;
1389.
1390.         Eigen::Translation3f tl_btol(_tf_x, _tf_y, _tf_z);           // tl: tran
        slation
1391.         Eigen::AngleAxisf rot_x_btol(_tf_roll, Eigen::Vector3f::UnitX()); // rot: rot
        ation
1392.         Eigen::AngleAxisf rot_y_btol(_tf_pitch, Eigen::Vector3f::UnitY());
1393.         Eigen::AngleAxisf rot_z_btol(_tf_yaw, Eigen::Vector3f::UnitZ());
1394.         tf_btol = (tl_btol * rot_z_btol * rot_y_btol * rot_x_btol).matrix();
1395.
1396.         Eigen::Translation3f tl_ltob((-1.0) * _tf_x, (-1.0) * _tf_y, (-
        1.0) * _tf_z); // tl: translation
1397.         Eigen::AngleAxisf rot_x_ltob((-
        1.0) * _tf_roll, Eigen::Vector3f::UnitX()); // rot: rotation
1398.         Eigen::AngleAxisf rot_y_ltob((-1.0) * _tf_pitch, Eigen::Vector3f::UnitY());
1399.         Eigen::AngleAxisf rot_z_ltob((-1.0) * _tf_yaw, Eigen::Vector3f::UnitZ());
1400.         tf_ltob = (tl_ltob * rot_z_ltob * rot_y_ltob * rot_x_ltob).matrix();
1401.
1402.         // Updated in initialpose_callback or gnss_callback
1403.         initial_pose.x = 0.0;
1404.         initial_pose.y = 0.0;
1405.         initial_pose.z = 0.0;
1406.         initial_pose.roll = 0.0;
1407.         initial_pose.pitch = 0.0;
1408.         initial_pose.yaw = 0.0;
1409.
1410.         // Publishers
1411.         predict_pose_pub = nh.advertise<geometry_msgs::PoseStamped>("/predict_pose", 1
        000);
1412.         predict_pose_imu_pub = nh.advertise<geometry_msgs::PoseStamped>("/predict_pose
        _imu", 1000);
1413.         predict_pose_odom_pub = nh.advertise<geometry_msgs::PoseStamped>("/predict_pos
        e_odom", 1000);
1414.         predict_pose_imu_odom_pub = nh.advertise<geometry_msgs::PoseStamped>("/predict
        _pose_imu_odom", 1000);
1415.         ndt_pose_pub = nh.advertise<geometry_msgs::PoseStamped>("/ndt_pose", 1000);
1416.         // current_pose_pub = nh.advertise<geometry_msgs::PoseStamped>("/current_pose"
        , 1000);

```

```

1417.     localizer_pose_pub = nh.advertise<geometry_msgs::PoseStamped>("/localizer_pose
    ", 1000);
1418.     estimate_twist_pub = nh.advertise<geometry_msgs::TwistStamped>("/estimate_twis
    t", 1000);
1419.     estimated_vel_mps_pub = nh.advertise<std_msgs::Float32>("/estimated_vel_mps",
    1000);
1420.     estimated_vel_kmph_pub = nh.advertise<std_msgs::Float32>("/estimated_vel_kmph"
    , 1000);
1421.     estimated_vel_pub = nh.advertise<geometry_msgs::Vector3Stamped>("/estimated_ve
    l", 1000);
1422.     time_ndt_matching_pub = nh.advertise<std_msgs::Float32>("/time_ndt_matching",
    1000);
1423.     ndt_stat_pub = nh.advertise<ndt_localizer::ndt_stat>("/ndt_stat", 1000);
1424.     ndt_reliability_pub = nh.advertise<std_msgs::Float32>("/ndt_reliability", 1000
    );
1425.
1426.     // Subscribers
1427.     ros::Subscriber param_sub = nh.subscribe("config/ndt", 10, param_callback);
1428.     ros::Subscriber gnss_sub = nh.subscribe("gnss_pose", 10, gnss_callback);
1429.     ros::Subscriber map_sub = nh.subscribe("points_map", 10, map_callback);
1430.     ros::Subscriber initialpose_sub = nh.subscribe("initialpose", 1000, initialpos
    e_callback);
1431.     ros::Subscriber points_sub = nh.subscribe("filtered_points", _queue_size, poin
    ts_callback);
1432.     ros::Subscriber odom_sub = nh.subscribe("/odom_pose", _queue_size*10, odom_cal
    lback);
1433.     ros::Subscriber imu_sub = nh.subscribe(_imu_topic.c_str(), _queue_size*10, imu
    _callback);
1434.
1435.     //Add by WEN WS
1436.     pubNdtPath=nh.advertise<nav_msgs::Path>("ndtPath",20);
1437.     pubspancptPath=nh.advertise<nav_msgs::Path>("spancptPath",20);
1438.     ros::Subscriber navsat_odom_sub=nh.subscribe("navsat/odom",_queue_size*10,navs
    at_odom_sub_callback);
1439.     // static nav_msgs::Path spancptPath;//create a path denote span-cpt pose
1440.     // static ros::Publisher pubspancptPath;
1441.
1442.     ros::spin();
1443.

```

```
1444.     return 0;
1445. }
```

Euclidean Cluster

```
1. #include <ros/ros.h>
2.
3. #include <pcl_conversions/pcl_conversions.h>
4. #include <pcl/PCLPointCloud2.h>
5. #include <pcl/conversions.h>
6. #include <pcl_ros/transforms.h>
7. #include <pcl_ros/point_cloud.h>
8.
9. #include <pcl/ModelCoefficients.h>
10. #include <pcl/point_types.h>
11.
12. #include <pcl/filters/extract_indices.h>
13. #include <pcl/filters/voxel_grid.h>
14. #include <pcl/filters/conditional_removal.h>
15.
16. #include <pcl/features/normal_3d.h>
17. #include <pcl/features/normal_3d_omp.h>
18. #include <pcl/features/don.h>
19. #include <pcl/features/fpfh_omp.h>
20.
21. #include <pcl/kdtree/kdtree.h>
22.
23. #include <pcl/sample_consensus/method_types.h>
24. #include <pcl/sample_consensus/model_types.h>
25.
26. #include <pcl/segmentation/sac_segmentation.h>
```

```
27. #include <pcl/segmentation/extract_clusters.h>
28. #include <pcl/segmentation/conditional_euclidean_clustering.h>
29.
30. #include <pcl/common/common.h>
31.
32. #include <pcl/search/organized.h>
33. #include <pcl/search/kdtree.h>
34.
35. #include <pcl/segmentation/extract_clusters.h>
36.
37. #include <visualization_msgs/MarkerArray.h>
38. #include <visualization_msgs/Marker.h>
39.
40. #include <std_msgs/Float32MultiArray.h>
41. #include <std_msgs/MultiArrayLayout.h>
42. #include <std_msgs/MultiArrayDimension.h>
43.
44. #include <lidar_tracker/centroids.h>
45. #include <lidar_tracker/CloudCluster.h>
46. #include <lidar_tracker/CloudClusterArray.h>
47.
48. #include <vector_map_server/PositionState.h>
49.
50. #include <jsk_recognition_msgs/BoundingBox.h>
51. #include <jsk_recognition_msgs/BoundingBoxArray.h>
52. #include <jsk_rviz_plugins/Pictogram.h>
53. #include <jsk_rviz_plugins/PictogramArray.h>
54.
55. #include <tf/tf.h>
56.
57. #include <limits>
58. #include <cmath>
59.
60. #include <opencv/cv.h>
61. #include <opencv/highgui.h>
62. #include <opencv2/core/version.hpp>
63. #if (CV_MAJOR_VERSION == 3)
64. #include "gencolors.cpp"
65. #else
```

```
66. #include <opencv2/contrib/contrib.hpp>
67. #endif
68.
69. #include <chrono>
70. #include <iostream>
71. #include <vector>
72. #include <string>
73. #include <sstream>
74.
75. #include "Cluster.h"
76.
77. // #include <vector_map/vector_map.h>
78. // #include <vector_map_server/GetSignal.h>
79.
80. #ifdef GPU_CLUSTERING
81.     #include "gpu_euclidean_clustering.h"
82. #endif
83.
84. using namespace cv;
85.
86. std::vector<cv::Scalar> _colors;
87. ros::Publisher _pub_cluster_cloud;
88. ros::Publisher _pub_ground_cloud;
89. ros::Publisher _centroid_pub;
90. ros::Publisher _marker_pub;
91. ros::Publisher _pub_clusters_message;
92. ros::Publisher _pub_text_pictogram;
93. visualization_msgs::Marker _visualization_marker;
94.
95. ros::Publisher _pub_points_lanes_cloud;
96. ros::Publisher _pub_jsk_boundingboxes;
97. ros::Publisher _pub_jsk_hulls;
98. ros::Publisher chat;
99. ros::Publisher chat1;
100.
101. ros::ServiceClient _vectormap_server;
102.
103. std_msgs::Header _velodyne_header;
104.
```

```

105.pcl::PointCloud<pcl::PointXYZ> _sensor_cloud;
106.
107.std::vector<double> _clustering_thresholds;
108.std::vector<double> _clustering_distances;
109.
110.tf::StampedTransform* _transform;
111.tf::StampedTransform* _velodyne_output_transform;
112.tf::TransformListener* _transform_listener;
113.
114.std::string _output_frame;
115.std::string _vectormap_frame;
116.static bool _velodyne_transform_available;
117.static bool _downsample_cloud;
118.static bool _pose_estimation;
119.static double _leaf_size;
120.static int _cluster_size_min;
121.static int _cluster_size_max;
122.
123.static bool _remove_ground; //only ground
124.
125.static bool _using_sensor_cloud;
126.static bool _use_diffnormals;
127.static bool _use_vector_map;
128.
129.static double _clip_min_height;
130.static double _clip_max_height;
131.
132.static bool _keep_lanes;
133.static double _keep_lane_left_distance;
134.static double _keep_lane_right_distance;
135.
136.static double _max_boundingbox_side;
137.static double _remove_points_upto;
138.static double _cluster_merge_threshold;
139.
140.void transformBoundingBox(const jsk_recognition_msgs::BoundingBox& in_boundingbox, jsk
    _recognition_msgs::BoundingBox& out_boundingbox, const std::string& in_target_frame, c
    onst std_msgs::Header& in_header)
141.{

```

```

142.     geometry_msgs::PoseStamped pose_in, pose_out;
143.     pose_in.header = in_header;
144.     pose_in.pose = in_boundingBox.pose;
145.     try
146.     {
147.         _transform_listener-
            >transformPose(in_target_frame, ros::Time(), pose_in, in_header.frame_id, pose_out);

148.     }
149.     catch (tf::TransformException &ex)
150.     {
151.         ROS_ERROR("transformBoundingBox: %s", ex.what());
152.     }
153.     out_boundingBox.pose = pose_out.pose;
154.     out_boundingBox.header = in_header;
155.     out_boundingBox.header.frame_id = in_target_frame;
156.     out_boundingBox.dimensions = in_boundingBox.dimensions;
157.     out_boundingBox.value = in_boundingBox.value;
158.     out_boundingBox.label = in_boundingBox.label;
159. }
160.
161. void publishCloudClusters(const ros::Publisher* in_publisher, const lidar_tracker::Clo
    udClusterArray& in_clusters, const std::string& in_target_frame, const std_msgs::Heade
    r& in_header)
162. {
163.     if (in_target_frame != in_header.frame_id)
164.     {
165.         lidar_tracker::CloudClusterArray clusters_transformed;
166.         clusters_transformed.header = in_header;
167.         clusters_transformed.header.frame_id = in_target_frame;
168.         for (auto i=in_clusters.clusters.begin(); i!= in_clusters.clusters.end(); i++)
169.         {
170.             lidar_tracker::CloudCluster cluster_transformed;
171.             cluster_transformed.header = in_header;
172.             try
173.             {
174.                 _transform_listener-
                    >lookupTransform(in_target_frame, _velodyne_header.frame_id,

```

```

175.                                     ros::Time(), *_transform);
176.             pcl_ros::transformPointCloud(in_target_frame, *_transform, i-
               >cloud, cluster_transformed.cloud);
177.             _transform_listener->transformPoint(in_target_frame, ros::Time(), i-
               >min_point, in_header.frame_id, cluster_transformed.min_point);
178.             _transform_listener->transformPoint(in_target_frame, ros::Time(), i-
               >max_point, in_header.frame_id, cluster_transformed.max_point);
179.             _transform_listener->transformPoint(in_target_frame, ros::Time(), i-
               >avg_point, in_header.frame_id, cluster_transformed.avg_point);
180.             _transform_listener->transformPoint(in_target_frame, ros::Time(), i-
               >centroid_point, in_header.frame_id, cluster_transformed.centroid_point);
181.
182.             cluster_transformed.dimensions = i->dimensions;
183.             cluster_transformed.eigen_values = i->eigen_values;
184.             cluster_transformed.eigen_vectors = i->eigen_vectors;
185.
186.             transformBoundingBox(i-
               >bounding_box, cluster_transformed.bounding_box, in_target_frame, in_header);
187.
188.             clusters_transformed.clusters.push_back(cluster_transformed);
189.         }
190.         catch (tf::TransformException &ex)
191.         {
192.             ROS_ERROR("publishCloudClusters: %s", ex.what());
193.         }
194.     }
195.     in_publisher->publish(clusters_transformed);
196. }
197. else
198. {
199.     in_publisher->publish(in_clusters);
200. }
201.}
202.
203.void publishCentroids(const ros::Publisher* in_publisher, const lidar_tracker::centroi
    ds& in_centroids, const std::string& in_target_frame, const std_msgs::Header& in_heade
    r)
204.{
205.    if (in_target_frame!=in_header.frame_id)

```

```

206.     {
207.         lidar_tracker::centroids centroids_transformed;
208.         centroids_transformed.header = in_header;
209.         centroids_transformed.header.frame_id = in_target_frame;
210.         for (auto i=centroids_transformed.points.begin(); i!= centroids_transformed.po
            ints.end(); i++)
211.             {
212.                 geometry_msgs::PointStamped centroid_in, centroid_out;
213.                 centroid_in.header = in_header;
214.                 centroid_in.point = *i;
215.                 try
216.                 {
217.                     _transform_listener-
                        >transformPoint(in_target_frame, ros::Time(), centroid_in, in_header.frame_id, centroi
                            d_out);
218.
219.                     centroids_transformed.points.push_back(centroid_out.point);
220.                 }
221.                 catch (tf::TransformException &ex)
222.                 {
223.                     ROS_ERROR("publishCentroids: %s", ex.what());
224.                 }
225.             }
226.         in_publisher->publish(centroids_transformed);
227.     }
228.     else
229.     {
230.         in_publisher->publish(in_centroids);
231.     }
232. }
233.
234. void publishBoundingBoxArray(const ros::Publisher* in_publisher, const jsk_recognition
    _msgs::BoundingBoxArray& in_boundingBox_array, const std::string& in_target_frame, con
    st std_msgs::Header& in_header)
235. {
236.     if (in_target_frame!=in_header.frame_id)
237.     {
238.         jsk_recognition_msgs::BoundingBoxArray boundingboxes_transformed;
239.         boundingboxes_transformed.header = in_header;

```

```

240.         boundingboxes_transformed.header.frame_id = in_target_frame;
241.         for (auto i=in_boundingbox_array.bboxes.begin(); i!= in_boundingbox_array.bboxes
242.             .end(); i++)
243.         {
244.             jsk_recognition_msgs::BoundingBox boundingbox_transformed;
245.             transformBoundingBox(*i, boundingbox_transformed, in_target_frame, in_head
246.                 er);
247.             boundingboxes_transformed.bboxes.push_back(boundingbox_transformed);
248.         }
249.         in_publisher->publish(boundingboxes_transformed);
250.     }
251. else
252. {
253.     in_publisher->publish(in_boundingbox_array);
254. }
255. void publishCloud(const ros::Publisher* in_publisher, const pcl::PointCloud<pcl::Point
256.     XYZ>::Ptr in_cloud_to_publish_ptr)
257. {
258.     sensor_msgs::PointCloud2 cloud_msg;
259.     pcl::toROSMsg(*in_cloud_to_publish_ptr, cloud_msg);
260.     cloud_msg.header=_velodyne_header;
261.     in_publisher->publish(cloud_msg);
262. }
263. void publishColorCloud(const ros::Publisher* in_publisher, const pcl::PointCloud<pcl::
264.     PointXYZRGB>::Ptr in_cloud_to_publish_ptr)
265. {
266.     sensor_msgs::PointCloud2 cloud_msg;
267.     pcl::toROSMsg(*in_cloud_to_publish_ptr, cloud_msg);
268.     cloud_msg.header=_velodyne_header;
269.     in_publisher->publish(cloud_msg);
270. }
271. void keepLanePoints(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_cloud_ptr,
272.     pcl::PointCloud<pcl::PointXYZ>::Ptr out_cloud_ptr,
273.     float in_left_lane_threshold = 1.5,
274.     float in_right_lane_threshold = 1.5)

```

```

275.{
276.    pcl::PointIndices::Ptr far_indices (new pcl::PointIndices);
277.    for(unsigned int i=0; i< in_cloud_ptr->points.size(); i++)
278.    {
279.        pcl::PointXYZ current_point;
280.        current_point.x=in_cloud_ptr->points[i].x;
281.        current_point.y=in_cloud_ptr->points[i].y;
282.        current_point.z=in_cloud_ptr->points[i].z;
283.
284.        if (
285.            current_point.y > (in_left_lane_threshold) || current_point.y < -
                1.0*in_right_lane_threshold
286.        )
287.        {
288.            far_indices->indices.push_back(i);
289.        }
290.    }
291.    out_cloud_ptr->points.clear();
292.    pcl::ExtractIndices<pcl::PointXYZ> extract;
293.    extract.setInputCloud (in_cloud_ptr);
294.    extract.setIndices(far_indices);
295.    extract.setNegative(true); //true removes the indices, false leaves only the indices
296.    extract.filter(*out_cloud_ptr);
297.}
298.
299.#ifdef GPU_CLUSTERING
300.std::vector<ClusterPtr> clusterAndColorGpu(const pcl::PointCloud<pcl::PointXYZ>::Ptr in
    n_cloud_ptr,
301.                                           pcl::PointCloud<pcl::PointXYZRGB>::Ptr out
    _cloud_ptr,
302.                                           jsk_recognition_msgs::BoundingBoxArray& in
    _out_boundingbox_array,
303.                                           lidar_tracker::centroids& in_out_centroids
    ,
304.                                           double in_max_cluster_distance=0.5)
305.{
306.    std::vector<ClusterPtr> clusters;
307.

```

```

308.     //Convert input point cloud to vectors of x, y, and z
309.
310.     int size = in_cloud_ptr->points.size();
311.
312.     if (size == 0)
313.         return clusters;
314.
315.     float *tmp_x, *tmp_y, *tmp_z;
316.
317.     tmp_x = (float *)malloc(sizeof(float) * size);
318.     tmp_y = (float *)malloc(sizeof(float) * size);
319.     tmp_z = (float *)malloc(sizeof(float) * size);
320.
321.     for (int i = 0; i < size; i++) {
322.         pcl::PointXYZ tmp_point = in_cloud_ptr->at(i);
323.
324.         tmp_x[i] = tmp_point.x;
325.         tmp_y[i] = tmp_point.y;
326.         tmp_z[i] = tmp_point.z;
327.     }
328.
329.     GpuEuclideanCluster gec1_cluster;
330.
331.     gec1_cluster.setInputPoints(tmp_x, tmp_y, tmp_z, size);
332.     gec1_cluster.setThreshold(in_max_cluster_distance);
333.     gec1_cluster.setMinClusterPts (_cluster_size_min);
334.     gec1_cluster.setMaxClusterPts (_cluster_size_max);
335.     gec1_cluster.extractClusters();
336.     std::vector<GpuEuclideanCluster::GClusterIndex> cluster_indices = gec1_cluster.get
        Output();
337.
338.     unsigned int k = 0;
339.
340.     for (auto it = cluster_indices.begin(); it != cluster_indices.end(); it++)
341.     {
342.         ClusterPtr cluster(new Cluster());
343.         cluster->SetCloud(in_cloud_ptr, it-
            >points_in_cluster, _velodyne_header, k, (int)_colors[k].val[0], (int)_colors[k].val[1]
            ], (int)_colors[k].val[2], "", _pose_estimation);

```

```

344.         clusters.push_back(cluster);
345.
346.         k++;
347.     }
348.
349.     free(tmp_x);
350.     free(tmp_y);
351.     free(tmp_z);
352.
353.     return clusters;
354.}
355.#endif
356.
357.std::vector<ClusterPtr> clusterAndColor(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_c
    loud_ptr,
358.    pcl::PointCloud<pcl::PointXYZRGB>::Ptr out_cloud_ptr,
359.    jsk_recognition_msgs::BoundingBoxArray& in_out_boundingBox_array,
360.    lidar_tracker::centroids& in_out_centroids,
361.    double in_max_cluster_distance=0.5)
362.{
363.    pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXY
    Z>);
364.
365.    //create 2d pc
366.    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_2d(new pcl::PointCloud<pcl::PointXYZ>);
367.
368.    pcl::copyPointCloud(*in_cloud_ptr, *cloud_2d);
369.    //make it flat
370.    for (size_t i=0; i<cloud_2d->points.size(); i++)
371.    {
372.        cloud_2d->points[i].z = 0;
373.    }
374.
375.    if (cloud_2d->points.size() > 0)
376.        tree->setInputCloud (cloud_2d);
377.
378.    std::vector<pcl::PointIndices> cluster_indices;
379.
380.    //perform clustering on 2d cloud

```

```

380.     pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
381.     ec.setClusterTolerance (in_max_cluster_distance); //
382.     ec.setMinClusterSize (_cluster_size_min);
383.     ec.setMaxClusterSize (_cluster_size_max);
384.     ec.setSearchMethod(tree);
385.     ec.setInputCloud (cloud_2d);
386.     ec.extract (cluster_indices);
387.     //use indices on 3d cloud
388.
389.     /*pcl::ConditionalEuclideanClustering<pcl::PointXYZ> cec (true);
390.     cec.setInputCloud (in_cloud_ptr);
391.     cec.setConditionFunction (&independentDistance);
392.     cec.setMinClusterSize (cluster_size_min);
393.     cec.setMaxClusterSize (cluster_size_max);
394.     cec.setClusterTolerance (_distance*2.0f);
395.     cec.segment (cluster_indices);*/
396.
397.     //////////////////////////////////////
398.     //--- 3. Color clustered points
399.     //////////////////////////////////////
400.     unsigned int k = 0;
401.     //pcl::PointCloud<pcl::PointXYZRGB>::Ptr final_cluster (new pcl::PointCloud<pcl::P
ointXYZRGB>);
402.
403.     std::vector<ClusterPtr> clusters;
404.     //pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_cluster (new pcl::PointCloud<pcl::P
ointXYZRGB>); //coord + color cluster
405.     for (auto it = cluster_indices.begin(); it != cluster_indices.end(); ++it)
406.     {
407.         ClusterPtr cluster(new Cluster());
408.         cluster->SetCloud(in_cloud_ptr, it-
>indices, _velodyne_header, k, (int)_colors[k].val[0], (int)_colors[k].val[1], (int)_c
olors[k].val[2], "", _pose_estimation);
409.         clusters.push_back(cluster);
410.
411.         k++;
412.     }
413.     //std::cout << "Clusters: " << k << std::endl;
414.     return clusters;

```

```

415.
416.}
417.
418.void checkClusterMerge(size_t in_cluster_id, std::vector<ClusterPtr>& in_clusters, std
    ::vector<bool>& in_out_visited_clusters, std::vector<size_t>& out_merge_indices, doubl
    e in_merge_threshold)
419.{
420.    //std::cout << "checkClusterMerge" << std::endl;
421.    pcl::PointXYZ point_a = in_clusters[in_cluster_id]->GetCentroid();
422.    for(size_t i=0; i< in_clusters.size(); i++)
423.    {
424.        if (i != in_cluster_id && !in_out_visited_clusters[i])
425.        {
426.            pcl::PointXYZ point_b = in_clusters[i]->GetCentroid();
427.            double distance = sqrt( pow(point_b.x - point_a.x,2) + pow(point_b.y - poi
                nt_a.y,2) );
428.            if (distance <= in_merge_threshold)
429.            {
430.                in_out_visited_clusters[i] = true;
431.                out_merge_indices.push_back(i);
432.                //std::cout << "Merging " << in_cluster_id << " with " << i << " dist:
                    " << distance << std::endl;
433.                checkClusterMerge(i, in_clusters, in_out_visited_clusters, out_merge_i
                    ndices, in_merge_threshold);
434.            }
435.        }
436.    }
437.}
438.
439.void mergeClusters(const std::vector<ClusterPtr>& in_clusters, std::vector<ClusterPtr>
    & out_clusters, std::vector<size_t> in_merge_indices, const size_t& current_index, std
    ::vector<bool>& in_out_merged_clusters)
440.{
441.    //std::cout << "mergeClusters:" << in_merge_indices.size() << std::endl;
442.    pcl::PointCloud<pcl::PointXYZRGB> sum_cloud;
443.    pcl::PointCloud<pcl::PointXYZ> mono_cloud;
444.    ClusterPtr merged_cluster(new Cluster());
445.    for (size_t i=0; i<in_merge_indices.size(); i++)
446.    {

```

```

447.         sum_cloud += *(in_clusters[in_merge_indices[i]]->GetCloud());
448.         in_out_merged_clusters[in_merge_indices[i]] = true;
449.     }
450.     std::vector<int> indices(sum_cloud.points.size(), 0);
451.     for (size_t i=0; i<sum_cloud.points.size(); i++)
452.     {
453.         indices[i]=i;
454.     }
455.
456.     if (sum_cloud.points.size() > 0)
457.     {
458.         pcl::copyPointCloud(sum_cloud, mono_cloud);
459.         //std::cout << "mergedClusters " << sum_cloud.points.size() << " mono:" << mon
460.         o_cloud.points.size() << std::endl;
461.         //cluster->SetCloud(in_cloud_ptr, it-
462.         >indices, _velodyne_header, k, (int)_colors[k].val[0], (int)_colors[k].val[1], (int)_c
463.         olors[k].val[2], "", _pose_estimation);
464.         merged_cluster-
465.         >SetCloud(mono_cloud.makeShared(), indices, _velodyne_header, current_index,(int)_colo
466.         rs[current_index].val[0], (int)_colors[current_index].val[1], (int)_colors[current_ind
467.         ex].val[2], "", _pose_estimation);
468.         out_clusters.push_back(merged_cluster);
469.     }
470. }
471.
472. void checkAllForMerge(std::vector<ClusterPtr>& in_clusters, std::vector<ClusterPtr>& o
473. ut_clusters, float in_merge_threshold)
474. {
475.     //std::cout << "checkAllForMerge" << std::endl;
476.     std::vector<bool> visited_clusters(in_clusters.size(), false);
477.     std::vector<bool> merged_clusters(in_clusters.size(), false);
478.     size_t current_index=0;
479.     for (size_t i = 0; i< in_clusters.size(); i++)
480.     {
481.         if (!visited_clusters[i])
482.         {
483.             visited_clusters[i] = true;
484.             std::vector<size_t> merge_indices;

```

```

478.         checkClusterMerge(i, in_clusters, visited_clusters, merge_indices, in_merg
           e_threshold);
479.         mergeClusters(in_clusters, out_clusters, merge_indices, current_index++, m
           erged_clusters);
480.     }
481. }
482. for(size_t i =0; i< in_clusters.size(); i++)
483. {
484.     //check for clusters not merged, add them to the output
485.     if (!merged_clusters[i])
486.     {
487.         out_clusters.push_back(in_clusters[i]);
488.     }
489. }
490.
491. //ClusterPtr cluster(new Cluster());
492.}
493.
494.void segmentByDistance(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_cloud_ptr,
495.    pcl::PointCloud<pcl::PointXYZRGB>::Ptr out_cloud_ptr,
496.    jsk_recognition_msgs::BoundingBoxArray& in_out_boundingbox_array,
497.    lidar_tracker::centroids& in_out_centroids,
498.    lidar_tracker::CloudClusterArray& in_out_clusters,
499.    jsk_recognition_msgs::PolygonArray& in_out_polygon_array,
500.    jsk_rviz_plugins::PictogramArray& in_out_pictogram_array)
501.{
502.    //cluster the pointcloud according to the distance of the points using different t
        hresholds (not only one for the entire pc)
503.    //in this way, the points farther in the pc will also be clustered
504.
505.    //0 => 0-15m d=0.5
506.    //1 => 15-30 d=1
507.    //2 => 30-45 d=1.6
508.    //3 => 45-60 d=2.1
509.    //4 => >60 d=2.6
510.
511.    std::vector<pcl::PointCloud<pcl::PointXYZ>::Ptr> cloud_segments_array(5);
512.
513.    for(unsigned int i=0; i<cloud_segments_array.size(); i++)

```

```

514.     {
515.         pcl::PointCloud<pcl::PointXYZ>::Ptr tmp_cloud(new pcl::PointCloud<pcl::PointXYZ>);
516.         cloud_segments_array[i] = tmp_cloud;
517.     }
518.
519.     for (unsigned int i=0; i<in_cloud_ptr->points.size(); i++)
520.     {
521.         pcl::PointXYZ current_point;
522.         current_point.x = in_cloud_ptr->points[i].x;
523.         current_point.y = in_cloud_ptr->points[i].y;
524.         current_point.z = in_cloud_ptr->points[i].z;
525.
526.         float origin_distance = sqrt( pow(current_point.x,2) + pow(current_point.y,2)
527.         );
528.         if (origin_distance < _clustering_distances[0] ) {cloud_segments_array
529.         [0]->points.push_back (current_point);}
530.         else if(origin_distance < _clustering_distances[1]) {cloud_segments_array
531.         [1]->points.push_back (current_point);}
532.         else if(origin_distance < _clustering_distances[2]) {cloud_segments_array
533.         [2]->points.push_back (current_point);}
534.         else if(origin_distance < _clustering_distances[3]) {cloud_segments_array
535.         [3]->points.push_back (current_point);}
536.         else {cloud_segments_array[
537.         4]->points.push_back (current_point);}
538.     }
539.
540.     std::vector <ClusterPtr> all_clusters;
541.     for(unsigned int i=0; i<cloud_segments_array.size(); i++)
542.     {
543. #ifdef GPU_CLUSTERING
544.         std::vector<ClusterPtr> local_clusters = clusterAndColorGpu(cloud_segments_array[i], out_cloud_ptr, in_out_boundingbox_array, in_out_centroids, _clustering_thresholds[i]);
545. #else
546.         std::vector<ClusterPtr> local_clusters = clusterAndColor(cloud_segments_array[i], out_cloud_ptr, in_out_boundingbox_array, in_out_centroids, _clustering_thresholds[i]);
547. #endif
548.     }

```

```

542. #endif
543.     all_clusters.insert(all_clusters.end(), local_clusters.begin(), local_clusters
        .end());
544. }
545.
546.     //Clusters can be merged or checked in here
547.     //....
548.     //check for mergable clusters
549.     std::vector<ClusterPtr> mid_clusters;
550.     std::vector<ClusterPtr> final_clusters;
551.
552.     if (all_clusters.size() > 0)
553.         checkAllForMerge(all_clusters, mid_clusters, _cluster_merge_threshold);
554.     else
555.         mid_clusters = all_clusters;
556.
557.     if (mid_clusters.size() > 0)
558.         checkAllForMerge(mid_clusters, final_clusters, _cluster_merge_threshold);
559.     else
560.         final_clusters = mid_clusters;
561.
562.     tf::StampedTransform vectormap_transform;
563.     if (_use_vector_map)
564.     {
565.         cv::TickMeter timer;
566.
567.         try
568.         {
569.             //if the frame of the vectormap is different than the input, obtain transf
orm
570.             if (_vectormap_frame != _velodyne_header.frame_id)
571.             {
572.                 _transform_listener-
>lookupTransform(_vectormap_frame, _velodyne_header.frame_id, ros::Time(), vectormap_t
ransform);
573.             }
574.
575.             timer.reset();timer.start();

```

```

576.
577.         //check if centroids are inside the drivable area
578.         for(unsigned int i=0; i<final_clusters.size(); i++)
579.         {
580.             //transform centroid points to vectormap frame
581.             pcl::PointXYZ pcl_centroid = final_clusters[i]->GetCentroid();
582.
583.             tf::Vector3 vector_centroid (pcl_centroid.x, pcl_centroid.y, pcl_centroid.z);
584.             tf::Vector3 transformed_centroid;
585.
586.             if (_vectormap_frame != _velodyne_header.frame_id)
587.                 transformed_centroid = vectormap_transform*vector_centroid;
588.             else
589.                 transformed_centroid = vector_centroid;
590.
591.             vector_map_server::PositionState position_state;
592.             position_state.request.position.x = transformed_centroid.getX();
593.             position_state.request.position.y = transformed_centroid.getY();
594.             position_state.request.position.z = transformed_centroid.getZ();
595.
596.
597.             if (_vectormap_server.call(position_state))
598.             {
599.                 final_clusters[i]->SetValidity(position_state.response.state);
600.                 /*std::cout << "Original:" << pcl_centroid.x << "," << pcl_centroid.y << "," << pcl_centroid.z <<
601.                     " Transformed:" << transformed_centroid.x() << "," << transformed_centroid.y() << "," << transformed_centroid.z() <<
602.                     " Validity:" << position_state.response.state << std::endl
603.                 ;*/
603.             }
604.             else
605.             {
606.                 ROS_INFO("vectormap_filtering: VectorMap Server Call failed. Make sure vectormap_server is running. No filtering performed.");
607.                 final_clusters[i]->SetValidity(true);
608.             }
609.         }

```

```

610.         timer.stop();
611.         //std::cout << "vm server took " << timer.getTimeMilli() << " ms to check
        " << final_clusters.size() << std::endl;
612.     }
613.     catch(tf::TransformException &ex)
614.     {
615.         ROS_INFO("vectormap_filtering: %s", ex.what());
616.     }
617. }
618. //Get final PointCloud to be published
619. in_out_polygon_array.header = _velodyne_header;
620. in_out_pictogram_array.header = _velodyne_header;
621. for(unsigned int i=0; i<final_clusters.size(); i++)
622. {
623.     *out_cloud_ptr = *out_cloud_ptr + *(final_clusters[i]->GetCloud());
624.
625.     jsk_recognition_msgs::BoundingBox bounding_box = final_clusters[i]-
        >GetBoundingBox();
626.
627.     geometry_msgs::PolygonStamped polygon = final_clusters[i]->GetPolygon();
628.     jsk_rviz_plugins::Pictogram pictogram_cluster;
629.     pictogram_cluster.header = _velodyne_header;
630.
631.     //PICTO
632.     pictogram_cluster.mode = pictogram_cluster.STRING_MODE;
633.     pictogram_cluster.pose.position.x = final_clusters[i]->GetMaxPoint().x;
634.     pictogram_cluster.pose.position.y = final_clusters[i]->GetMaxPoint().y;
635.     pictogram_cluster.pose.position.z = final_clusters[i]->GetMaxPoint().z;
636.     tf::Quaternion quat(0.0, -0.7, 0.0, 0.7);
637.     tf::quaternionTFToMsg(quat, pictogram_cluster.pose.orientation);
638.     pictogram_cluster.size = 4;
639.     std_msgs::ColorRGBA color;
640.     color.a = 1; color.r = 1; color.g = 1; color.b = 1;
641.     pictogram_cluster.color = color;
642.     pictogram_cluster.character = std::to_string( i );
643.     //PICTO
644.
645.     //pcl::PointXYZ min_point = final_clusters[i]->GetMinPoint();
646.     //pcl::PointXYZ max_point = final_clusters[i]->GetMaxPoint();

```

```

647.         pcl::PointXYZ center_point = final_clusters[i]->GetCentroid();
648.         geometry_msgs::Point centroid;
649.         centroid.x = center_point.x; centroid.y = center_point.y; centroid.z = center_
        point.z;
650.         bounding_box.header = _velodyne_header;
651.         polygon.header = _velodyne_header;
652.         geometry_msgs::Point centroid2;
653.         centroid2.x=bounding_box.pose.position.x;
654.         centroid2.y=bounding_box.pose.position.y;
655.         centroid2.z=bounding_box.pose.position.z;
656.         chat.publish(centroid);
657.         chat1.publish(centroid2);
658.
659.         if (    final_clusters[i]->IsValid()
660.             //&& bounding_box.dimensions.x >0 && bounding_box.dimensions.y >0 && b
        ounding_box.dimensions.z > 0
661.             //&&    bounding_box.dimensions.x < _max_boundingbox_side && bounding_
        box.dimensions.y < _max_boundingbox_side
662.             )
663.         {
664.             in_out_boundingbox_array.bboxes.push_back(bounding_box);
665.             in_out_centroids.points.push_back(centroid);
666.             _visualization_marker.points.push_back(centroid);
667.
668.             in_out_polygon_array.polygons.push_back(polygon);
669.             in_out_pictogram_array.pictograms.push_back(pictogram_cluster);
670.
671.             lidar_tracker::CloudCluster cloud_cluster;
672.             final_clusters[i]->ToRosMessage(_velodyne_header, cloud_cluster);
673.             in_out_clusters.clusters.push_back(cloud_cluster);
674.         }
675.     }
676.
677.     for(size_t i=0; i< in_out_polygon_array.polygons.size();i++)
678.     {
679.         in_out_polygon_array.labels.push_back(i);
680.     }
681.
682. }

```

```

683.
684. void removeFloor(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_cloud_ptr, pcl::PointClo
    ud<pcl::PointXYZ>::Ptr out_nofloor_cloud_ptr, pcl::PointCloud<pcl::PointXYZ>::Ptr out_
    onlyfloor_cloud_ptr, float in_max_height=0.2, float in_floor_max_angle=0.1)
685. {
686.     /*pcl::PointIndicesPtr ground (new pcl::PointIndices);
687.     // Create the filtering object
688.     pcl::ProgressiveMorphologicalFilter<pcl::PointXYZ> pmf;
689.     pmf.setInputCloud (in_cloud_ptr);
690.     pmf.setMaxWindowSize (20);
691.     pmf.setSlope (1.0f);
692.     pmf.setInitialDistance (0.5f);
693.     pmf.setMaxDistance (3.0f);
694.     pmf.extract (ground->indices);
695.
696.     // Create the filtering object
697.     pcl::ExtractIndices<pcl::PointXYZ> extract;
698.     extract.setInputCloud (in_cloud_ptr);
699.     extract.setIndices (ground);
700.     extract.setNegative(true); //true removes the indices, false leaves only the indice
        s
701.     extract.filter(*out_nofloor_cloud_ptr);
702.
703.     //EXTRACT THE FLOOR FROM THE CLOUD
704.     extract.setNegative(false); //true removes the indices, false leaves only the indic
        es
705.     extract.filter(*out_onlyfloor_cloud_ptr); */
706.
707.     pcl::SACSegmentation<pcl::PointXYZ> seg;
708.     pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
709.     pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
710.
711.     seg.setOptimizeCoefficients (true);
712.     seg.setModelType(pcl::SACMODEL_PERPENDICULAR_PLANE);
713.     seg.setMethodType(pcl::SAC_RANSAC);
714.     seg.setMaxIterations(100);
715.     seg.setAxis(Eigen::Vector3f(0,0,1));
716.     seg.setEpsAngle(in_floor_max_angle);
717.

```

```

718.     seg.setDistanceThreshold (in_max_height);//floor distance
719.     seg.setOptimizeCoefficients(true);
720.     seg.setInputCloud(in_cloud_ptr);
721.     seg.segment(*inliers, *coefficients);
722.     if (inliers->indices.size () == 0)
723.     {
724.         std::cout << "Could not estimate a planar model for the given dataset." << std
            ::endl;
725.     }
726.
727.     //REMOVE THE FLOOR FROM THE CLOUD
728.     pcl::ExtractIndices<pcl::PointXYZ> extract;
729.     extract.setInputCloud (in_cloud_ptr);
730.     extract.setIndices(inliers);
731.     extract.setNegative(true);//true removes the indices, false leaves only the indice
        s
732.     extract.filter(*out_nofloor_cloud_ptr);
733.
734.     //EXTRACT THE FLOOR FROM THE CLOUD
735.     extract.setNegative(false);//true removes the indices, false leaves only the indic
        es
736.     extract.filter(*out_onlyfloor_cloud_ptr);
737.}
738.
739.void downsampleCloud(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_cloud_ptr, pcl::Poin
    tCloud<pcl::PointXYZ>::Ptr out_cloud_ptr, float in_leaf_size=0.2)
740.{
741.    pcl::VoxelGrid<pcl::PointXYZ> sor;
742.    sor.setInputCloud(in_cloud_ptr);
743.    sor.setLeafSize((float)in_leaf_size, (float)in_leaf_size, (float)in_leaf_size);
744.    sor.filter(*out_cloud_ptr);
745.}
746.
747.void clipCloud(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_cloud_ptr, pcl::PointCloud
    <pcl::PointXYZ>::Ptr out_cloud_ptr, float in_min_height=-
        1.3, float in_max_height=0.5)
748.{
749.    out_cloud_ptr->points.clear();
750.    for (unsigned int i=0; i<in_cloud_ptr->points.size(); i++)

```

```

751.     {
752.         if (in_cloud_ptr->points[i].z >= in_min_height &&
753.             in_cloud_ptr->points[i].z <= in_max_height)
754.         {
755.             out_cloud_ptr->points.push_back(in_cloud_ptr->points[i]);
756.         }
757.     }
758.}
759.
760.void differenceNormalsSegmentation(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_cloud_
    ptr, pcl::PointCloud<pcl::PointXYZ>::Ptr out_cloud_ptr)
761.{
762.    float small_scale=0.5;
763.    float large_scale=2.0;
764.    float angle_threshold=0.5;
765.    pcl::search::Search<pcl::PointXYZ>::Ptr tree;
766.    if (in_cloud_ptr->isOrganized ())
767.    {
768.        tree.reset (new pcl::search::OrganizedNeighbor<pcl::PointXYZ> ());
769.    }
770.    else
771.    {
772.        tree.reset (new pcl::search::KdTree<pcl::PointXYZ> (false));
773.    }
774.
775.    // Set the input pointcloud for the search tree
776.    tree->setInputCloud (in_cloud_ptr);
777.
778.    pcl::NormalEstimationOMP<pcl::PointXYZ, pcl::PointNormal> normal_estimation;
779.    //pcl::gpu::NormalEstimation<pcl::PointXYZ, pcl::PointNormal> normal_estimation;
780.    normal_estimation.setInputCloud (in_cloud_ptr);
781.    normal_estimation.setSearchMethod (tree);
782.
783.    normal_estimation.setViewPoint (std::numeric_limits<float>::max (), std::numeric_l
        imits<float>::max (), std::numeric_limits<float>::max ());
784.
785.    pcl::PointCloud<pcl::PointNormal>::Ptr normals_small_scale (new pcl::PointCloud<pc
        l::PointNormal>);

```

```

786.     pcl::PointCloud<pcl::PointNormal>::Ptr normals_large_scale (new pcl::PointCloud<pcl::PointNormal>);
787.
788.     normal_estimation.setRadiusSearch (small_scale);
789.     normal_estimation.compute (*normals_small_scale);
790.
791.     normal_estimation.setRadiusSearch (large_scale);
792.     normal_estimation.compute (*normals_large_scale);
793.
794.     pcl::PointCloud<pcl::PointNormal>::Ptr diffnormals_cloud (new pcl::PointCloud<pcl::PointNormal>);
795.     pcl::copyPointCloud<pcl::PointXYZ, pcl::PointNormal>(*in_cloud_ptr, *diffnormals_cloud);
796.
797.     // Create DoN operator
798.     pcl::DifferenceOfNormalsEstimation<pcl::PointXYZ, pcl::PointNormal, pcl::PointNormal> diffnormals_estimator;
799.     diffnormals_estimator.setInputCloud (in_cloud_ptr);
800.     diffnormals_estimator.setNormalScaleLarge (normals_large_scale);
801.     diffnormals_estimator.setNormalScaleSmall (normals_small_scale);
802.
803.     diffnormals_estimator.initCompute();
804.
805.     diffnormals_estimator.computeFeature(*diffnormals_cloud);
806.
807.     pcl::ConditionOr<pcl::PointNormal>::Ptr range_cond (new pcl::ConditionOr<pcl::PointNormal>());
808.     range_cond->addComparison (pcl::FieldComparison<pcl::PointNormal>::ConstPtr (
809.         new pcl::FieldComparison<pcl::PointNormal> ("curvature", pcl::ComparisonOps::GT, angle_threshold) )
810.     );
811.     // Build the filter
812.     pcl::ConditionalRemoval<pcl::PointNormal> cond_removal;
813.     cond_removal.setCondition(range_cond);
814.     cond_removal.setInputCloud (diffnormals_cloud);
815.
816.     pcl::PointCloud<pcl::PointNormal>::Ptr diffnormals_cloud_filtered (new pcl::PointCloud<pcl::PointNormal>);
817.

```

```

818.    // Apply filter
819.    cond_removal.filter (*diffnormals_cloud_filtered);
820.
821.    pcl::copyPointCloud<pcl::PointNormal, pcl::PointXYZ>(*diffnormals_cloud, *out_clo
        d_ptr);
822.}
823.
824.void removePointsUpTo(const pcl::PointCloud<pcl::PointXYZ>::Ptr in_cloud_ptr, pcl::Poi
        ntCloud<pcl::PointXYZ>::Ptr out_cloud_ptr, const double in_distance)
825.{
826.    out_cloud_ptr->points.clear();
827.    for (unsigned int i=0; i<in_cloud_ptr->points.size(); i++)
828.    {
829.        float origin_distance = sqrt( pow(in_cloud_ptr->
            points[i].x,2) + pow(in_cloud_ptr->points[i].y,2) );
830.        if (origin_distance > in_distance)
831.        {
832.            out_cloud_ptr->points.push_back(in_cloud_ptr->points[i]);
833.        }
834.    }
835.}
836.
837.void velodyne_callback(const sensor_msgs::PointCloud2ConstPtr& in_sensor_cloud)
838.{
839.    if (!_using_sensor_cloud)
840.    {
841.        _using_sensor_cloud = true;
842.
843.        pcl::PointCloud<pcl::PointXYZ>::Ptr current_sensor_cloud_ptr (new pcl::PointCl
            oud<pcl::PointXYZ>);
844.        pcl::PointCloud<pcl::PointXYZ>::Ptr removed_points_cloud_ptr (new pcl::PointCl
            oud<pcl::PointXYZ>);
845.        pcl::PointCloud<pcl::PointXYZ>::Ptr downsampled_cloud_ptr (new pcl::PointCloud
            <pcl::PointXYZ>);
846.        pcl::PointCloud<pcl::PointXYZ>::Ptr inlanes_cloud_ptr (new pcl::PointCloud<pcl
            ::PointXYZ>);
847.        pcl::PointCloud<pcl::PointXYZ>::Ptr nofloor_cloud_ptr (new pcl::PointCloud<pcl
            ::PointXYZ>);

```

```

848.         pcl::PointCloud<pcl::PointXYZ>::Ptr onlyfloor_cloud_ptr (new pcl::PointCloud<p
            cl::PointXYZ>);
849.         pcl::PointCloud<pcl::PointXYZ>::Ptr diffnormals_cloud_ptr (new pcl::PointCloud
            <pcl::PointXYZ>);
850.         pcl::PointCloud<pcl::PointXYZ>::Ptr clipped_cloud_ptr (new pcl::PointCloud<pcl
            ::PointXYZ>);
851.         pcl::PointCloud<pcl::PointXYZRGB>::Ptr colored_clustered_cloud_ptr (new pcl::P
            ointCloud<pcl::PointXYZRGB>);
852.
853.         lidar_tracker::centroids centroids;
854.         lidar_tracker::CloudClusterArray cloud_clusters;
855.         jsk_recognition_msgs::BoundingBoxArray boundingbox_array;
856.         jsk_recognition_msgs::PolygonArray polygon_array;
857.         jsk_rviz_plugins::PictogramArray pictograms_array;
858.
859.         pcl::fromROSMsg(*in_sensor_cloud, *current_sensor_cloud_ptr);
860.
861.         _velodyne_header = in_sensor_cloud->header;
862.
863.         cv::TickMeter timer;
864.
865.         timer.reset();timer.start();
866.
867.         if (_remove_points_upto > 0.0)
868.         {
869.             removePointsUpTo(current_sensor_cloud_ptr, removed_points_cloud_ptr, _remo
            ve_points_upto);
870.         }
871.         else
872.             removed_points_cloud_ptr = current_sensor_cloud_ptr;
873.
874.         //std::cout << "Downsample before: " <<removed_points_cloud_ptr->
            points.size();
875.         if (_downsample_cloud)
876.             downsampleCloud(removed_points_cloud_ptr, downsampled_cloud_ptr, _leaf_siz
            e);
877.         else
878.             downsampled_cloud_ptr =removed_points_cloud_ptr;
879.

```

```

880.         //std::cout << " after: " << downsampled_cloud_ptr->points.size();
881.         timer.stop(); //std::cout << "downsampleCloud:" << timer.getTimeMilli() << "ms
            " << std::endl;
882.
883.         timer.reset();timer.start();
884.         clipCloud(downsampled_cloud_ptr, clipped_cloud_ptr, _clip_min_height, _clip_max_height);
885.         timer.stop(); //std::cout << "clipCloud:" << clipped_cloud_ptr->points.size() << "time " << timer.getTimeMilli() << "ms" << std::endl;
886.
887.         timer.reset();timer.start();
888.         if(_keep_lanes)
889.             keepLanePoints(clipped_cloud_ptr, inlanes_cloud_ptr, _keep_lane_left_distance, _keep_lane_right_distance);
890.         else
891.             inlanes_cloud_ptr = clipped_cloud_ptr;
892.         timer.stop(); //std::cout << "keepLanePoints:" << timer.getTimeMilli() << "ms" << std::endl;
893.
894.         timer.reset();timer.start();
895.         if(_remove_ground)
896.         {
897.             removeFloor(inlanes_cloud_ptr, nofloor_cloud_ptr, onlyfloor_cloud_ptr);
898.             publishCloud(&_pub_ground_cloud, onlyfloor_cloud_ptr);
899.         }
900.         else
901.             nofloor_cloud_ptr = inlanes_cloud_ptr;
902.         timer.stop(); //std::cout << "removeFloor:" << timer.getTimeMilli() << "ms" << std::endl;
903.
904.         publishCloud(&_pub_points_lanes_cloud, nofloor_cloud_ptr);
905.
906.         timer.reset();timer.start();
907.         if (_use_diffnormals)
908.             differenceNormalsSegmentation(nofloor_cloud_ptr, diffnormals_cloud_ptr);
909.         else
910.             diffnormals_cloud_ptr = nofloor_cloud_ptr;
911.         timer.stop(); //std::cout << "differenceNormalsSegmentation:" << timer.getTimeMilli() << "ms" << std::endl;

```

```

912.
913.     timer.reset();timer.start();
914.     segmentByDistance(diffnormals_cloud_ptr, colored_clustered_cloud_ptr, bounding
        box_array, centroids, cloud_clusters, polygon_array, pictograms_array);
915.     //timer.stop(); std::cout << "segmentByDistance:" << timer.getTimeMilli() << "
        ms" << std::endl;
916.
917.     timer.reset();timer.start();
918.     publishColorCloud(&_pub_cluster_cloud, colored_clustered_cloud_ptr);
919.     timer.stop(); //std::cout << "publishColorCloud:" << timer.getTimeMilli() << "
        ms" << std::endl;
920.     // Publish BB
921.     boundingbox_array.header = _velodyne_header;
922.
923.     _pub_jsk_hulls.publish(polygon_array);//publish convex hulls
924.     _pub_text_pictogram.publish(pictograms_array);//publish_ids
925.
926.     timer.reset();timer.start();
927.     publishBoundingBoxArray(&_pub_jsk_boundingboxes, boundingbox_array, _output_fr
        ame, _velodyne_header);
928.     centroids.header = _velodyne_header;
929.     timer.stop(); //std::cout << "publishBoundingBoxArray:" << timer.getTimeMilli(
        ) << "ms" << std::endl;
930.
931.     timer.reset();timer.start();
932.     publishCentroids(&_centroid_pub, centroids, _output_frame, _velodyne_header);
933.     timer.stop(); //std::cout << "publishCentroids:" << timer.getTimeMilli() << "m
        s" << std::endl;
934.
935.     _marker_pub.publish(_visualization_marker);
936.     _visualization_marker.points.clear();//transform? is it used?
937.     cloud_clusters.header = _velodyne_header;
938.
939.     timer.reset();timer.start();
940.     publishCloudClusters(&_pub_clusters_message, cloud_clusters, _output_frame, _v
        elodyne_header);
941.     timer.stop(); //std::cout << "publishCloudClusters:" << timer.getTimeMilli() <
        < "ms" << std::endl << std::endl;

```

```

942.
943.     _using_sensor_cloud = false;
944. }
945.}
946.
947./*
948.void vectormap_callback(const visualization_msgs::MarkerArray::Ptr in_vectormap_markers)
949.{
950.    float min_x=std::numeric_limits<float>::max();float max_x=-
        std::numeric_limits<float>::max();
951.    float min_y=std::numeric_limits<float>::max();float max_y=-
        std::numeric_limits<float>::max();
952.    pcl::PointXYZ min_point;
953.    pcl::PointXYZ max_point;
954.    std::vector<geometry_msgs::Point> vectormap_points;
955.    std::string marker_frame;
956.    double map_scale = -10.0;
957.    for(auto i=in_vectormap_markers->markers.begin(); i!= in_vectormap_markers-
        >markers.end(); i++)
958.    {
959.        visualization_msgs::Marker current_marker = *i;
960.        marker_frame = current_marker.header.frame_id;
961.        if (current_marker.ns == "road_edge")
962.        {
963.            for (unsigned int j=0; j< current_marker.points.size(); j++)
964.            {
965.                geometry_msgs::Point p = current_marker.points[j];
966.                p.x*=map_scale;
967.                p.y*=map_scale;
968.                if(p.x<min_x)    min_x = p.x;
969.                if(p.y<min_y)    min_y = p.y;
970.                if(p.x>max_x)    max_x = p.x;
971.                if(p.y>max_y)    max_y = p.y;
972.                vectormap_points.push_back(p);
973.            }
974.        }
975.    }
976.    min_point.x = min_x;    min_point.y = min_y;

```

```

977.     max_point.x = max_x;     max_point.y = max_y;
978.
979.     min_point.x*=-1.0;
980.     min_point.y*=-1.0;
981.     //translate the points to the minimum point
982.     for (auto i=vectormap_points.begin(); i!=vectormap_points.end(); i++)
983.     {
984.         (*i).x+=min_point.x;
985.         (*i).y+=min_point.y;
986.     }
987.     max_point.x+=min_point.x;
988.     max_point.y+=min_point.y;
989.     //get world tf
990.     std::string error_transform_msg;
991.     tf::Vector3 map_origin_point;
992.     if(_transform_listener-
        >waitForTransform("/map", marker_frame, ros::Time(0), ros::Duration(5), ros::Duration(
            0.1), &error_transform_msg))
993.     {
994.         _transform_listener-
            >lookupTransform("/map", marker_frame, ros::Time(0), *_transform);
995.         map_origin_point = _transform->getOrigin();
996.         map_origin_point.setX( map_origin_point.x() - min_point.x);
997.         map_origin_point.setY( map_origin_point.y() - min_point.y);
998.     }
999.     else
1000.    {
1001.        ROS_INFO("Euclidean Cluster (vectormap_callback): %s", error_transform
            _msg.c_str());
1002.    }
1003.
1004.    cv::Mat map_image = cv::Mat::zeros(max_point.y, max_point.x, CV_8UC3);
1005.
1006.    std::cout << "W,H:" << max_point << std::endl;
1007.
1008.    cv::Point image_start_point (vectormap_points[0].x, vectormap_points[0].y)
        ;
1009.    cv::Point prev_point = image_start_point;
1010.    for (auto i=vectormap_points.begin(); i!=vectormap_points.end(); i++)

```

```

1011.         {
1012.             cv::line(map_image, prev_point, cv::Point((int)(i->x), (int)(i-
                >y)), cv::Scalar::all(255));
1013.
1014.             prev_point.x = (int)(i->x);
1015.             prev_point.y = (int)(i->y);
1016.         }
1017.         cv::circle(map_image, image_start_point, 3, cv::Scalar(255,0,0));
1018.         cv::imshow("vectormap", map_image);
1019.         cv::waitKey(0);
1020.     }*/
1021.
1022.     int main (int argc, char** argv)
1023.     {
1024.         // Initialize ROS
1025.         ros::init (argc, argv, "euclidean_cluster");
1026.
1027.         ros::NodeHandle h;
1028.         ros::NodeHandle private_nh("~");
1029.
1030.         tf::StampedTransform transform;
1031.         tf::TransformListener listener;
1032.
1033.         _transform = &transform;
1034.         _transform_listener = &listener;
1035.
1036.         #if (CV_MAJOR_VERSION == 3)
1037.             generateColors(_colors, 100);
1038.         #else
1039.             cv::generateColors(_colors, 100);
1040.         #endif
1041.
1042.         _pub_cluster_cloud = h.advertise<sensor_msgs::PointCloud2>("/points_cluste
            r",1);
1043.         _pub_ground_cloud = h.advertise<sensor_msgs::PointCloud2>("/points_ground"
            ,1);
1044.         _centroid_pub = h.advertise<lidar_tracker::centroids>("/cluster_centroids"
            ,1);

```

```

1045.         _marker_pub = h.advertise<visualization_msgs::Marker>("centroid_marker",1)
        ;
1046.
1047.         _pub_points_lanes_cloud = h.advertise<sensor_msgs::PointCloud2>("/points_l
        anes",1);
1048.         _pub_jsk_boundingboxes = h.advertise<jsk_recognition_msgs::BoundingBoxArra
        y>("/bounding_boxes",1);
1049.         _pub_jsk_hulls = h.advertise<jsk_recognition_msgs::PolygonArray>("/cluster
        _hulls",1);
1050.         _pub_clusters_message = h.advertise<lidar_tracker::CloudClusterArray>("/cl
        oud_clusters",1);
1051.         _pub_text_pictogram = h.advertise<jsk_rviz_plugins::PictogramArray>("clust
        er_ids", 10); ROS_INFO("output pictograms topic: %s", "cluster_id");
1052.         chat=h.advertise<geometry_msgs::Point>("chatter",10);
1053.         chat1=h.advertise<geometry_msgs::Point>("chatter1",10);
1054.         std::string points_topic;
1055.
1056.         _using_sensor_cloud = false;
1057.
1058.         if (private_nh.getParam("points_node", points_topic))
1059.         {
1060.             ROS_INFO("euclidean_cluster > Setting points node to %s", points_topic
        .c_str());
1061.         }
1062.         else
1063.         {
1064.             ROS_INFO("euclidean_cluster > No points node received, defaulting to p
        oints_raw, you can use _points_node:=YOUR_TOPIC");
1065.             points_topic = "/points_raw";
1066.         }
1067.
1068.         _use_diffnormals = false;
1069.         if (private_nh.getParam("use_diffnormals", _use_diffnormals))
1070.         {
1071.             if (_use_diffnormals)
1072.                 ROS_INFO("Euclidean Clustering: Applying difference of normals on
        clustering pipeline");
1073.             else

```

```

1074.             ROS_INFO("Euclidean Clustering: Difference of Normals will not be
                used.");
1075.         }
1076.
1077.         /* Initialize tuning parameter */
1078.         private_nh.param("downsample_cloud", _downsample_cloud, false); ROS_INFO("
                downsample_cloud: %d", _downsample_cloud);
1079.         private_nh.param("remove_ground", _remove_ground, true);      ROS_INFO("
                remove_ground: %d", _remove_ground);
1080.         private_nh.param("leaf_size", _leaf_size, 0.1);      ROS_INFO("
                leaf_size: %f", _leaf_size);
1081.         private_nh.param("cluster_size_min", _cluster_size_min, 20);  ROS_INFO("
                cluster_size_min %d", _cluster_size_min);
1082.         private_nh.param("cluster_size_max", _cluster_size_max, 100000);ROS_INFO("
                cluster_size_max: %d", _cluster_size_max);
1083.         private_nh.param("pose_estimation", _pose_estimation, false);  ROS_INFO("
                pose_estimation: %d", _pose_estimation);
1084.         private_nh.param("clip_min_height", _clip_min_height, -
                1.3);  ROS_INFO("clip_min_height: %f", _clip_min_height);
1085.         private_nh.param("clip_max_height", _clip_max_height, 0.5);  ROS_INFO("
                clip_max_height: %f", _clip_max_height);
1086.         private_nh.param("keep_lanes", _keep_lanes, false);      ROS_INFO("
                keep_lanes: %d", _keep_lanes);
1087.         private_nh.param("keep_lane_left_distance", _keep_lane_left_distance, 5.0)
                ;  ROS_INFO("keep_lane_left_distance: %f", _keep_lane_left_distance);
1088.         private_nh.param("keep_lane_right_distance", _keep_lane_right_distance, 5.
                0);  ROS_INFO("keep_lane_right_distance: %f", _keep_lane_right_distance);
1089.         private_nh.param("clustering_thresholds", _clustering_thresholds);
1090.         private_nh.param("clustering_distances", _clustering_distances);
1091.         private_nh.param("max_boundingbox_side", _max_boundingbox_side, 10.0);
                ROS_INFO("max_boundingbox_side: %f", _max_boundingbox_side);
1092.         private_nh.param("cluster_merge_threshold", _cluster_merge_threshold, 1.5)
                ;  ROS_INFO("cluster_merge_threshold: %f", _cluster_merge_threshold);
1093.         private_nh.param<std::string>("output_frame", _output_frame, "velodyne");
                ROS_INFO("output_frame: %s", _output_frame.c_str());
1094.
1095.         private_nh.param("use_vector_map", _use_vector_map, false);
                ROS_INFO("use_vector_map: %d", _use_vector_map);

```

```

1096.         private_nh.param<std::string>("vectormap_frame", _vectormap_frame, "map");
           ROS_INFO("vectormap_frame: %s", _output_frame.c_str());

1097.

1098.         private_nh.param("remove_points_upto", _remove_points_upto, 0.0);           RO
           S_INFO("remove_points_upto: %f", _remove_points_upto);

1099.

1100.

1101.         _velodyne_transform_available = false;

1102.

1103.         if (_clustering_distances.size()!=4)
1104.         {
1105.             _clustering_distances = {15, 30, 45, 60}; //maximum distance from senso
r origin to separate segments
1106.         }
1107.         if (_clustering_thresholds.size()!=5)
1108.         {
1109.             _clustering_thresholds = {0.5, 1.1, 1.6, 2.1, 2.6}; //Nearest neighbor
distance threshold for each segment
1110.         }
1111.

1112.         std::cout << "_clustering_thresholds: "; for (auto i = _clustering_thresho
lds.begin(); i != _clustering_thresholds.end(); ++i) std::cout << *i << ' '; std::cou
t << std::endl;

1113.         std::cout << "_clustering_distances: "; for (auto i = _clustering_distances
.begin(); i != _clustering_distances.end(); ++i) std::cout << *i << ' '; std::cout <<
std::endl;

1114.

1115.         // Create a ROS subscriber for the input point cloud
1116.         ros::Subscriber sub = h.subscribe (points_topic, 1, velodyne_callback);
1117.         //ros::Subscriber sub_vectormap = h.subscribe ("vector_map", 1, vectormap_
callback);

1118.         _vectormap_server = h.serviceClient<vector_map_server::PositionState>("vec
tor_map_server/is_way_area");

1119.

1120.         _visualization_marker.header.frame_id = "velodyne";
1121.         _visualization_marker.header.stamp = ros::Time();
1122.         _visualization_marker.ns = "my_namespace";
1123.         _visualization_marker.id = 0;
1124.         _visualization_marker.type = visualization_msgs::Marker::SPHERE_LIST;

```

```

1125.         _visualization_marker.action = visualization_msgs::Marker::ADD;
1126.         _visualization_marker.scale.x = 1.0;
1127.         _visualization_marker.scale.y = 1.0;
1128.         _visualization_marker.scale.z = 1.0;
1129.         _visualization_marker.color.a = 1.0;
1130.         _visualization_marker.color.r = 0.0;
1131.         _visualization_marker.color.g = 0.0;
1132.         _visualization_marker.color.b = 1.0;
1133.         // marker.lifetime = ros::Duration(0.1);
1134.         _visualization_marker.frame_locked = true;
1135.
1136.         // Spin
1137.         ros::spin ();
1138.     }

```

Laserscan

```

1.  #include <pointcloud_to_laserscan/pointcloud_to_laserscan_nodelet.h>
2.  #include <sensor_msgs/LaserScan.h>
3.  #include <pluginlib/class_list_macros.h>
4.  #include <sensor_msgs/point_cloud2_iterator.h>
5.  #include <tf2_sensor_msgs/tf2_sensor_msgs.h>
6.  #include <ros/ros.h>
7.  #include <ros/console.h>
8.  #include "pcl/visualization/pcl_plotter.h"
9.  #include <vector>
10. #include <iostream>
11. #include <utility>
12. #include <pcl/visualization/interactor.h>
13. #include <pcl/visualization/pcl_plotter.h>
14. #include <pcl/common/common_headers.h>
15. #include "std_msgs/String.h"
16. #include <pcl/point_types.h>
17. #include <pcl/filters/passthrough.h>
18. #include <pcl_conversions/pcl_conversions.h>
19. #include <sensor_msgs/PointCloud2.h>
20.

```

```

21. using namespace std;
22. using namespace pcl::visualization;
23. namespace pointcloud_to_laserscan
24. {
25.
26.     PointCloudToLaserScanNodelet::PointCloudToLaserScanNodelet() {}
27.
28.     void PointCloudToLaserScanNodelet::onInit()
29.     {
30.         //std::cout<<"onInit...";
31.         NODELET_INFO("onInit...");
32.         ROS_INFO(" ROS INFO my info ");
33.         maxinum_elevation=0.0;
34.         maxinum_zhight=0.0;
35.         boost::mutex::scoped_lock lock(connect_mutex_);
36.         private_nh_ = getPrivateNodeHandle();
37.
38.         private_nh_.param<std::string>("target_frame", target_frame_, "");
39.         private_nh_.param<double>("transform_tolerance", tolerance_, 0.01);
40.         private_nh_.param<double>("min_height", min_height_, 0.0);
41.         private_nh_.param<double>("max_height", max_height_, 1.0);
42.
43.         private_nh_.param<double>("angle_min", angle_min_, -M_PI / 2.0);
44.         private_nh_.param<double>("angle_max", angle_max_, M_PI / 2.0);
45.         private_nh_.param<double>("angle_increment", angle_increment_, M_PI / 360.0);
46.         private_nh_.param<double>("scan_time", scan_time_, 1.0 / 30.0);
47.         private_nh_.param<double>("range_min", range_min_, 0.45);
48.         private_nh_.param<double>("range_max", range_max_, 4.0);
49.
50.         int concurrency_level;
51.         private_nh_.param<int>("concurrency_level", concurrency_level, 1);
52.         private_nh_.param<bool>("use_inf", use_inf_, true);
53.
54.         //Check if explicitly single threaded, otherwise, let nodelet manager dictate thre
         ad pool size
55.         if (concurrency_level == 1)
56.         {
57.             nh_ = getNodeHandle();
58.         }

```

```

59.     else
60.     {
61.         nh_ = getMTNodeHandle();
62.     }
63.
64.     // Only queue one pointcloud per running thread
65.     if (concurrency_level > 0)
66.     {
67.         input_queue_size_ = concurrency_level;
68.     }
69.     else
70.     {
71.         input_queue_size_ = boost::thread::hardware_concurrency();
72.     }
73.
74.     // if pointcloud target frame specified, we need to filter by transform availability
75.     if (!target_frame_.empty())
76.     {
77.         tf2_.reset(new tf2_ros::Buffer());
78.         tf2_listener_.reset(new tf2_ros::TransformListener(*tf2_));
79.         message_filter_.reset(new MessageFilter(sub_, *tf2_, target_frame_, input_queue_size_, nh_));
80.         message_filter_-
            >registerCallback(boost::bind(&PointCloudToLaserScanNodelet::cloudCb, this, _1));
81.         message_filter_-
            >registerFailureCallback(boost::bind(&PointCloudToLaserScanNodelet::failureCb, this, _1, _2));
82.     }
83.     else // otherwise setup direct subscription
84.     {
85.         sub_.registerCallback(boost::bind(&PointCloudToLaserScanNodelet::cloudCb, this, _1));
86.     }
87.
88.     pub_ = nh_.advertise<sensor_msgs::LaserScan>("scan", 10,
89.
                                                    boost::bind(&PointCloudToLaserScanNodelet::connectCb, this),

```

```

90.                                     boost::bind(&PointCloudToLaserScanNodelet::disconnectCb, this));
91.     azimuth_elevation_message_pub=nh_.advertise<geometry_msgs::PoseArray>("azi_eleva",
    1000);
92.     filtered_points_pub=nh_.advertise<sensor_msgs::PointCloud2>("pass_through_filtered
    _points",1000);
93.     sub_.subscribe(nh_, "cloud_in", input_queue_size_);//subscribe cloud_in manually

94. }
95.
96. void PointCloudToLaserScanNodelet::connectCb()//subscribe point cloud
97. {
98.     boost::mutex::scoped_lock lock(connect_mutex_);
99.     // ROS_INFO("pub_.getNumSubscribers() %d", pub_.getNumSubscribers());
100.    // std::cout<<"pub_.getNumSubscribers()="<<pub_.getNumSubscribers();
101.    if (pub_.getNumSubscribers() > 0 && sub_.getSubscriber().getNumPublishers() == 0)

102.    {
103.        NODELET_INFO("Got a subscriber to scan, starting subscriber to pointcloud");
104.        sub_.subscribe(nh_, "cloud_in", input_queue_size_);
105.    }
106. }
107.
108. void PointCloudToLaserScanNodelet::disconnectCb()
109. {
110.     boost::mutex::scoped_lock lock(connect_mutex_);
111.     std::cout<<"pub_.getNumSubscribers()="<<pub_.getNumSubscribers();
112.     if (pub_.getNumSubscribers() == 0)
113.     {
114.         NODELET_INFO("No subscribers to scan, shutting down subscriber to pointcloud");
115.         sub_.unsubscribe();
116.     }
117. }
118.
119. void PointCloudToLaserScanNodelet::failureCb(const sensor_msgs::PointCloud2ConstPtr
    &cloud_msg,
120.                                     tf2_ros::filter_failure_reasons::Filter
    FailureReason reason)
121. {

```

```

122.     NODELET_WARN_STREAM_THROTTLE(1.0, "Can't transform pointcloud from frame " << clou
        d_msg->header.frame_id << " to "
123.         << message_filter_->getTargetFramesString());
124.     }
125.
126.     void PointCloudToLaserScanNodelet::cloudCb(const sensor_msgs::PointCloud2ConstPtr &c
        loud_msg)
127.     {
128.
129.         //build laserscan output
130.         sensor_msgs::LaserScan output;
131.         output.header = cloud_msg->header;
132.         if (!target_frame_.empty())
133.         {
134.             output.header.frame_id = target_frame_;
135.         }
136.
137.         output.angle_min = angle_min_;//~angle_min (double, default: -蟻
            /2) The minimum scan angle in radians.
138.         output.angle_max = angle_max_;//~angle_max (double, default: 蟻
            /2) The maximum scan angle in radians.
139.         output.angle_increment = angle_increment_;//~angle_increment (double, default: 蟻
            /360) Resolution of laser scan in radians per ray.
140.         output.time_increment = 0.0;//
141.         output.scan_time = scan_time_;//~scan_time (double, default: 1.0/30.0) The scan ra
            te in seconds.
142.         output.range_min = range_min_; //The minimum ranges to return in meters. ~range_mi
            n (double, default: 0.45)
143.         output.range_max = range_max_;//~range_max (double, default: 4.0) The maximum rang
            es to return in meters.
144.
145.         //determine amount of rays to create
146.         uint32_t ranges_size = std::ceil((output.angle_max - output.angle_min) / output.an
            gle_increment);
147.
148.         //determine if laserscan rays with no obstacle data will evaluate to infinity or m
            ax_range
149.         if (use_inf_)// infinite or not
150.         {

```

```

151.     output.ranges.assign(ranges_size, std::numeric_limits<double>::infinity());
152. }
153. else
154. {
155.     output.ranges.assign(ranges_size, output.range_max + 1.0);
156. }
157. //notice:input is cloud_msg
158. sensor_msgs::PointCloud2ConstPtr cloud_out;
159. sensor_msgs::PointCloud2Ptr cloud;
160.
161. // Transform cloud if necessary
162. if (!(output.header.frame_id == cloud_msg->header.frame_id))
163. {
164.     try
165.     {
166.         cloud.reset(new sensor_msgs::PointCloud2);
167.         tf2_-
            >transform(*cloud_msg, *cloud, target_frame_, ros::Duration(tolerance_));
168.         cloud_out = cloud;
169.         cloud_out_overall=cloud;//save overall variates
170.     }
171.     catch (tf2::TransformException ex)
172.     {
173.         NODELET_ERROR_STREAM("Transform failure: " << ex.what());
174.         return;
175.     }
176. }
177. else
178. {
179.     cloud_out = cloud_msg;
180.     cloud_out_overall=cloud_msg;//save overall variates
181. }
182.
183. /*****Do some Filter before *****/
184.// pass-through filter to get rid of near and far ranges
185. pcl::PassThrough<pcl::PointXYZ> pass;
186. pcl::PointCloud<pcl::PointXYZ>::Ptr pcd_tmp(new pcl::PointCloud<pcl::PointXYZ>());

```

```

187.     pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl::Point
        XYZ>);
188.     // point cloud in ros topic
189. #if PCL_VERSION_COMPARE(>=,1,7,0)
190.     pcl::PCLPointCloud2 pcd2_tmp;
191.     pcl_conversions::toPCL(*cloud_msg, pcd2_tmp);
192.     pcl::fromPCLPointCloud2(pcd2_tmp, *pcd_tmp);
193. #else
194.     pcl::fromROSMsg(*msg, *pcd_tmp);
195. #endif
196.     pass.setInputCloud (pcd_tmp);
197.     pass.setFilterFieldName ("z");
198.     pass.setFilterLimits (1, 10);
199.     pass.filter (*cloud_filtered); // point cloud in pcl::PointCloud<pcl::PointXYZ> mes
        sage
200.     pcl::toROSMsg(*cloud_filtered, pass_filtered_pointcloud);
201.     filtered_points_pub.publish(pass_filtered_pointcloud);
202. // voxel-grid filter to get rid of near and far ranges
203.
204.
205.
206.     // Iterate through pointcloud
207.     for (sensor_msgs::PointCloud2ConstIterator<float>
208.         iter_x(*cloud_out, "x"), iter_y(*cloud_out, "y"), iter_z(*cloud_out, "z"
        );
209.         iter_x != iter_x.end();
210.         ++iter_x, ++iter_y, ++iter_z)
211.     {
212.         if (std::isnan(*iter_x) || std::isnan(*iter_y) || std::isnan(*iter_z))
213.         {
214.             NODELET_DEBUG("rejected for nan in point(%f, %f, %f)\n", *iter_x, *iter_y, *it
                er_z);
215.             ROS_INFO("rejected for nan in point(%f, %f, %f)\n", *iter_x, *iter_y, *iter_z)
                ;
216.             continue;
217.         }
218.
219.         if (*iter_z > max_height_ || *iter_z < min_height_)
220.         {

```

```

221.         NODELET_DEBUG("rejected for height %f not in range (%f, %f)\n", *iter_z, min_h
            eight_, max_height_);
222.         // ROS_INFO("rejected for height %f not in range (%f, %f)\n", *iter_z, min_hei
            ght_, max_height_);
223.         continue;
224.     }
225.     ROS_INFO("*iter_z= %f",*iter_z);
226.
227.
228.     //calculate azimuth and elevation
229.     double angle = atan2(*iter_y, *iter_x);//azimuth
230.     double elevation=atan2(*iter_z,sqrt(*iter_x *(*iter_x) + *iter_y *(*iter_y)));//
        elevation
231.     if(fabs(elevation)>maxinum_elevation) maxinum_elevation=elevation;
232.     // ROS_INFO("maxinum_elevation=%f",maxinum_elevation*180/3.14159);
233.     if(fabs(*iter_z)>maxinum_zhight && *iter_z>0) maxinum_zhight=*iter_z;
234.     // ROS_INFO("maxinum_zhight=%f",maxinum_zhight);
235.     poseArrayElement.position.x=angle*180/3.14159;//azimuth
236.     poseArrayElement.position.y=elevation*180/3.14159;//elevation
237.     poseArrayElement.position.z=3.0;
238.     poseArrayElement.orientation.x=0.1;
239.     poseArrayElement.orientation.y=0.2;
240.     poseArrayElement.orientation.z=0.21;
241.     poseArrayElement.orientation.w=0.22;
242.     pose_array_skyplot.poses.push_back(poseArrayElement);
243.
244.     double range = hypot(*iter_x, *iter_y);
245.     if (range < range_min_)
246.     {
247.         NODELET_DEBUG("rejected for range %f below minimum value %f. Point: (%f, %f, %
            f)", range, range_min_, *iter_x, *iter_y,
248.             *iter_z);
249.         ROS_INFO("rejected for range %f below minimum value %f. Point: (%f, %f, %f)",
            range, range_min_, *iter_x, *iter_y,
250.             *iter_z);
251.         continue;
252.     }
253.
254.     // double angle = atan2(*iter_y, *iter_x);

```

```

255.     if (angle < output.angle_min || angle > output.angle_max)
256.     {
257.         NODELET_DEBUG("rejected for angle %f not in range (%f, %f)\n", angle, output.a
angle_min, output.angle_max);
258.         ROS_INFO("rejected for angle %f not in range (%f, %f)\n", angle, output.angle_
min, output.angle_max);
259.         continue;
260.     }
261.
262.     //overwrite range at laserscan ray if new range is smaller
263.     int index = (angle - output.angle_min) / output.angle_increment;
264.     if (range < output.ranges[index])
265.     {
266.         output.ranges[index] = range;
267.     }
268.
269. }//iterate stop here
270. /*****print size *****/
271. int a=pose_array_skyplot.poses.size();
272. // ROS_INFO("pose_array_skyplot.poses.size()= %d",a);
273.
274. azimuth_elevation_message_pub.publish(pose_array_skyplot);//publish topic
275. pose_array_skyplot.poses.clear();
276.
277. pub_.publish(output);
278. }
279.
280.}
281.
282.PLUGINLIB_DECLARE_CLASS(pointcloud_to_laserscan, PointCloudToLaserScanNodelet, pointcl
oud_to_laserscan::PointCloudToLaserScanNodelet, nodelet::Nodelet);

```