PCL REGISTRATION

Alexandru E. Ichim

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

RGB-D mapping in 1h

1. Motivation
2. General pipeline
3. ICP steps
4. Graph optimizers
5. Putting it all together
6. Meshing and texturing
7. Results

SUMMARY

- auto-exposure is bad for RGB features
- want a system that is independent of the lighting conditions and repeatable at any time of day
- indoor environments have repetitive textures - bad for localization
- want a coherent depth-only registration pipeline that can only improve by adding RGB information

Introduction - Why depth-only registration?

1. Incremental
   - Kinect Fusion
   - Kintinuous
2. perform loop closure
   - RGB-D Mapping variants
   - various commercial systems
3. with manual intervention

IZADI S., KIM D., HILLIGES O., MOLYNEAUX D., NEWCOMBE R., KOHLI P., SHOT- TON J., HODGES S., FREEMAN D., DAVISON A., FITZGIBBON A.: Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In Proceedings of the 24th annual ACM symposium on User interface software and technology (New York, NY, USA, 2011), UIST '11, ACM, pp. 559–568.

WHELAN T., JOHANNSSON H., KAESS M., LEONARD J., MCDONALD J.: Robust Tracking for Real-Time Dense RGB-D Mapping with Kintinuous. Tech. Rep. MIT- CSAIL-TR-2012-031, Computer Science and Artificial Intelligence Laboratory, MIT, Sep 2012.

HENRY P., KRAININ M., HERBST E., REN X., FOX D.: Rgbd mapping: Using depth cameras for dense 3d modeling of indoor environments. In In RGB-D: Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS (2010).

STURM J., ENGELHARD N., ENDRES F., BURGARD W., CREMERS D.: A benchmark for the evaluation of rgb-d slam systems. In Proc. of the International Conference on Intelligent Robot Systems (IROS) (Oct. 2012).

PITZER B., KAMMEL S., DUHADWAY C., BECKER J.: Automatic reconstruction of textured 3d models. In IEEE International Conference on Robotics and Automa- tion, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010 (2010), IEEE, pp. 3486– 3493.

DU H., HENRY P., REN X., CHENG M., GOLDMAN D. B., SEITZ S. M., FOX D.: Interactive 3d modeling of indoor environments with a consumer depth camera. In Proceedings of the 13th international conference on Ubiquitous computing (New York, NY, USA, 2011), UbiComp '11, ACM, pp. 75–84.

Can now quickly build any of them with PCL!

Systems Out There

Room modeling, outside modeling, object modeling - various sizes

Blensor

TUM Datasets

with ground truth transformations

Our own

http://svn.pointclouds.org/data/

http://www.pointclouds.org/media/

Datasets

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

```
int
main (int argc,
      char **argv)
{
  PCL_INFO ("Syntax: %s -input_dir <dir_with_pcds> -output_dir <dir_with_resulting_transforms>\n", argv[0]);

  /// Get the pcd file list from a folder
  std::string input_dir = "";
  console::parse_argument (argc, argv, "-input_dir", input_dir);

  std::string output_dir = "";
  console::parse_argument (argc, argv, "-output_dir", output_dir);

  if (input_dir == "" || output_dir == "")
  {
    PCL_ERROR ("Syntax error!\n");
    return (-1);
  }

  std::vector<std::string> input_pcd_files;
  getFilesFromDirectory (input_dir, ".PCD", input_pcd_files);
  PCL_INFO ("Processing %zu files ...\n", input_pcd_files.size ());

  CompleteGraphGeometricRegistration<PointXYZ> complete_registration;
  /// Load files one by one and add them to the CompleteGraphGeometricRegistration instance
  for (size_t cloud_i = 0; cloud_i < input_pcd_files.size (); ++cloud_i)
  {
    PCL_INFO (">   Reading in cloud %s\n", input_pcd_files[cloud_i].c_str ());
    PointCloud<PointXYZ>::Ptr cloud_in (new PointCloud<PointXYZ> ());
    io::loadPCDFile (input_pcd_files[cloud_i], *cloud_in);
    complete_registration.addNewPointCloud (cloud_in);
  }

(...)
```

The entire code! (1/2)

```
(...)
  /// Get the transforms and write them to the corresponding files
  /// Strip the base names and add the output folder name and the .transform extension
  std::map<size_t, Eigen::Matrix4d> transforms;
  complete_registration.getTransforms (transforms);

  /// Create output dir first
  boost::filesystem::create_directory (output_dir);
  for (std::map<size_t, Eigen::Matrix4d>::const_iterator t_it = transforms.begin ();
       t_it != transforms.end (); ++t_it)
  {
    std::string transform_name = output_dir + "/" + af::getBaseName (input_pcd_files[t_it->first]) + ".transform";
    saveTransform (transform_name, t_it->second);
  }

  /// Save keyframe transformations separately
  boost::filesystem::create_directory (output_dir + "/keyframes");
  std::map<size_t, Eigen::Matrix4d> keyframe_transforms;
  complete_registration.getKeyframeTransforms (keyframe_transforms);
  for (std::map<size_t, Eigen::Matrix4d>::const_iterator t_it = keyframe_transforms.begin ();
       t_it != keyframe_transforms.end (); ++t_it)
  {
    std::string transform_name = output_dir + "/keyframes/" + af::getBaseName (input_pcd_files[t_it->first]) + ".transform";
    saveTransform (transform_name, t_it->second);
  }

  return (0);
}
```

The entire code! (2/2)

```
CompleteGraphGeometricRegistration<PointXYZ> complete_registration;
```

This is where the magic happens.

+ some helper functions

The entire code! Well, not really :-)

```
std::string
getBaseName (std::string &filename)
{
  size_t slash_pos = filename.find_last_of ('/');
  size_t dot_pos = filename.find_last_of (".");
  std::string base_name = filename.substr (slash_pos + 1, dot_pos - slash_pos - 1);

  return (base_name);
}
```

```
void
getFilesFromDirectory (const std::string path_dir,
                       const std::string extension,
                       std::vector<std::string> &files)
{
  if (path_dir != "" && boost::filesystem::exists (path_dir))
  {
    boost::filesystem::directory_iterator end_itr;
    for (boost::filesystem::directory_iterator itr (path_dir); itr != end_itr; ++itr)
      if (!is_directory (itr->status ()) &&
          boost::algorithm::to_upper_copy (boost::filesystem::extension (itr->path ())) ==
boost::algorithm::to_upper_copy (extension))
      {
        files.push_back (itr->path ().string ());
      }
  }
  sort (files.begin (), files.end ());
}
```

The helper functions (1/2)

```
void
saveTransform (const std::string &file, const Eigen::Matrix4d &transform)
{
  std::ofstream ofs;
  ofs.open (file.c_str (), std::ios::trunc | std::ios::binary);
  for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 4; ++j)
      ofs.write (reinterpret_cast<const char*>(&transform (i, j)), sizeof (double));
  ofs.close ();
}
```

```
void
loadTransform (const std::string &file, Eigen::Matrix4d &transform)
{
  std::ifstream ifs;
  ifs.open (file.c_str (), std::ios::binary);
  for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 4; ++j)
      ifs.read (reinterpret_cast<char*>(&transform (i, j)), sizeof (double));
  ifs.close ();
}
```

The helper functions (2/2)

- subsample clouds
- associate points from the source and target clouds
- filter correspondences
- estimate the transformation using the correspondences
- re-iterate until convergence

Iterative Closest Point (ICP)

Experiments on combinations of algorithms for the steps of the ICP algorithm:

- random sampling
- closest point correspondence
- filter correspondences based on normals
- constant weighting of point pairs
- point-to-plane error metric

Baseline

- random sampling
- projection-based pairing
- constant weighting of point pairs
- point-to-plane error metric

Best

RUSINKIEWICZ S., LEVOY M.: Efficient variants of the ICP algorithm. In Third International Conference on 3D Digital Imaging and Modeling (3DIM) (June 2001).

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

RGB-D
camera noise

$$\sigma_L(\theta)[px] = 0.8 + 0.035 * \frac{\theta}{\pi/2-\theta}$$
$$\sigma_L(\theta)[m] = \sigma_L(\theta)[px] * z * p_x / f_x$$

$$\sigma_z(z,\theta) = 0.0012 + 0.0019 * (z-0.4)^2, \text{ when } 10^\circ \leq \theta \leq 60^\circ$$

Noise
model

Filtering 1/5

Median filter

- simple image processing filter
- good for impulse noise
- fixed size window - take median

```
#include <pcl/filters/median_filter.h>

  pcl::MedianFilter<pcl::PointXYZ> median_filter;
  median_filter.setInputCloud (cloud_in);
  median_filter.setWindowSize (7);
  median_filter.setMaxAllowedMovement (0.02);
  median_filter.filter (*cloud_out);
```

Bilateral filter smooths the signal and preserves strong edges

one image:

$$\tilde{I}_p = \frac{\sum_{s \in \mathcal{N}} f(p-s) * g(I_p - I_s) * I_s}{\sum_{p \in \mathcal{N}} f(p-s) * g(I_p - I_s)}$$

```
pcl::FastBilateralFilter<pcl::PointXYZ> bilateral_filter;
bilateral_filter.setInputCloud (cloud_in);
bilateral_filter.setSigmaS (5);
bilateral_filter.setSigmaR (0.005f);
bilateral_filter.filter (*cloud_out);
```

* fast implementation by interpreting the image as a 3D function
the filter becomes a convolution in 3D

PARIS S., DURAND F.: A fast approximation of the bilateral filter using a signal processing approach. Int. J. Comput. Vision 81, 1 (Jan. 2009), 24–52.

Bilateral filter

two images
(depth + RGB):

$$\tilde{S}_p = \frac{\sum_{q \in \mathcal{N}} S_q * f(||p-q||) * g(||I_p - I_q||)}{\sum_{q \in \mathcal{N}} f(||p-q||) * g(||I_p - I_q||)}$$

```
#include <pcl/surface/bilateral_upsampling.h>

  pcl::BilateralUpsampling<PointXYZRGB, PointXYZRGB> bilateral_upsampling;
  bilateral_upsampling.setInputCloud (cloud_rgb_in);
  bilateral_upsampling.setSigmaColor (15.f);
  bilateral_upsampling.setSigmaDepth (0.5f);
  bilateral_upsampling.process (*cloud_rgb_out);
```

Kopf, Johannes and Cohen, Michael F. and Lischinski, Dani and Uyttendaele, Matt - Joint Bilateral Upsampling, ACM Transactions in Graphics, July 2007

Moving Least Squares

- fit a plane locally with PCA
- fit a 2D function on the plane
- project the point onto the function

$$\sum_{i=1}^{N} [g(x_i, y_i) - n \circ (p_i - q)]^2 * \theta(||p_i - q||)$$

```
#include <pcl/surface/mls.h>

  pcl::MovingLeastSquares<pcl::PointXYZ, pcl::PointNormal> mls;
  mls.setInputCloud (cloud_in);
  mls.setPolynomialFit (true);
  mls.setPolynomialOrder (3);
  mls.setSearchRadius (0.03);
  mls.process (*cloud_mls);
```

- Do not do it for all the point in the cloud, but for a voxel grid-ed version
- allows for up-sampling and uniform point distribution

ALEXA M., BEHR J., COHEN-OR D., FLEISHMAN S., LEVIN D., SILVA C. T.: Com- puting and rendering point set surfaces. IEEE Transactions on Visualization and Computer Graphics 9 (2003), 3–15.

Cloud	RMSE original Simulated Noise [cm]	RMSE Bilateral Filter [cm]	RMSE MLS [cm] Filter [cm]	RMSE Median Filter [cm]
1	18.219	17.465	17.672	17.457
2	19.730	19.092	19.123	19.147
3	17.066	16.782	16.665	16.436
4	17.259	17.173	16.756	16.475
5	11.542	11.533	11.197	10.938

Filtering 5/5

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

- use all the points
- uniform sampling
- random sampling
- normal space sampling
- sample points on color edges

Rusinkiewicz et al. - best

- covariance sampling

new

RUSINKIEWICZ S., LEVOY M.: Efficient variants of the ICP algorithm. In Third International Conference on 3D Digital Imaging and Modeling (3DIM) (June 2001).

- use all the points
- uniform sampling
- random sampling

very similar results

advantage of random - force to choose different pairs at each iteration

```
#include <pcl/filters/random_sample.h>

  pcl::RandomSample<pcl::PointXYZRGB> random_sampling;
  random_sampling.setSample (cloud_rgb_in->size () / 4);
  random_sampling.setInputCloud (cloud_rgb_in);
  random_sampling.filter (*cloud_rgb_out);
```

- **normal space sampling**

place normals into m3 bins
randomly (uniform) select samples from each bin
=> uniformly cover the half-sphere of normals for each scan

```
#include <pcl/filters/normal_space.h>

  pcl::NormalSpaceSampling<pcl::PointXYZ, pcl::Normal> normal_sampling;
  normal_sampling.setBins (16, 16, 16);
  normal_sampling.setSample (cloud_in->size () / 8);
  normal_sampling.setInputCloud (cloud_in);
  normal_sampling.setNormals (normals);
  normal_sampling.filter (*cloud_out);
```

sample points so that the transformation is "locked" better the sampled cloud needs to have all the eigenvalues of the covariance matrix equal

- **covariance sampling**

iterative algorithm - sort points by contribution to each eigenvalue, and select the smallest to balance the eigenvalues

$$\Delta d_i = [\Delta r^T \ \Delta t^T] \begin{bmatrix} p_i \times n_i \\ n_i \end{bmatrix}$$

each point has a translational force and a rotation torque

$$C = FF^T = \begin{bmatrix} p_1 \times n_1 & \dots & p_k \times n_k \\ n_1 & \dots & n_k \end{bmatrix} \begin{bmatrix} (p_1 \times n_1)^T & n_1^T \\ \dots & \dots \\ (p_k \times n_k)^T & n_k^T \end{bmatrix}$$

the covariance matrix tells us about the stability of the registration

not full rank => transform is not unique

want a condition number close to 1

condition number !!!

single wall 2946

two walls 862.45

room corner 10.27

desk clutter 12.83

boxes 4.58

Sampling 5/8

condition number !!!

```
#include <pcl/filters/covariance_sampling.h>

  pcl::CovarianceSampling<pcl::PointNormal, pcl::PointNormal> covariance_sampling;
  covariance_sampling.setNumberOfSamples (cloud_with_normals->size () / 8);
  covariance_sampling.setKeepOrganized (true);
  covariance_sampling.setInputCloud (cloud_with_normals);
  covariance_sampling.setNormals (cloud_with_normals);
  covariance_sampling.filter (*cloud_with_normals_sampled);
```

organized output

can use the same cloud

uniform

random

normals

covariance

Scan	Condition number				
	input	uniform	random	normal	covariance
single wall	2946.26	2910.05	2959.42	498.65	635.89
two walls	862.45	1099.66	824.77	98.47	101.21
room corner	10.27	10.27	10.31	7.7	6.97
desk clutter	12.83	13.10	12.91	6.89	21.36
boxes	4.58	4.59	4.73	3.32	9.33

Sampling 7/8

Amount of subsampling

HENRY P., KRAININ M., HERBST E., REN X., FOX D.: Rgbd mapping: Using depth cameras for dense 3d modeling of indoor environments. In In RGB-D: Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS (2010).

uniform sampling - 25% - 10% of the initial size => 75k-30k points

POMERLEAU F., MAGNENAT S., COLAS F., LIU M., SIEGWART R.: Tracking a depth camera: Parameter exploration for fast icp. In Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2011).

complex sampling techniques are too slow compared to the performance improvement

uniformly subsample the depth grid to 6.25% of the initial size (19.2k points) and then random subsample to 1% (3.7k points)

The number of points is the factor that influences performance the most!

Tuning the sampling rate is essential for real-time systems!

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

3D sensors create sample locations, and lose information about the surfaces

- averaging methods
- optimization-based methods

weighting:

- angles
- area
- centroid
- gravitational

- PlaneSVD — fit a plane in the local neighborhood, minimize the error with SVD
- PlanePCA — minimize the variance of the points by computing a local coordinate system

superior

```
#include <pcl/features/normal_3d.h>

  pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normal_estimation;
  normal_estimation.setRadiusSearch (0.03);
  normal_estimation.setInputCloud (cloud_in);
  normal_estimation.compute (*normals);
```

Computation time: seconds

KLASING K., ALTHOFF D., WOLLHERR D., BUSS M.: Comparison of surface normal estimation methods for range sensing applications. In Proceedings of the 2009 IEEE international conference on Robotics and Automation (Piscataway, NJ, USA, 2009), ICRA'09, IEEE Press, pp. 1977–1982.

Normal Estimation 1/2

Normal Estimation using integral images

- smooth the data with a variable window size depending on the depth
- mask for depth changes
- compute the normals as cross products of grid neighbors - adaptive window

Authors claim better results than PlanePCA

Downside: cannot compute normals
near the image borders

```
#include <pcl/features/integral_image_normal.h>

  pcl::IntegralImageNormalEstimation<pcl::PointXYZ, pcl::Normal>
normal_estimation_ii;
  normal_estimation_ii.setNormalEstimationMethod
(normal_estimation_ii.AVERAGE_DEPTH_CHANGE);
  normal_estimation_ii.setMaxDepthChangeFactor (0.01f);
  normal_estimation_ii.setDepthDependentSmoothing (true);
  normal_estimation_ii.setNormalSmoothingSize (50.0f);
  normal_estimation_ii.setInputCloud (cloud_in);
  normal_estimation_ii.compute (*normals);
```

Computation time: ~50 ms

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

- projection-based

$$\begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix} = \begin{bmatrix} d \cdot u \\ d \cdot v \\ d \end{bmatrix}$$

bad when the transformation is large

- search-based
  - exhaustive O(N)
  - kd-tree Long init times O(N logN), but fast search times O(logN)

Cloud size	Naive search [ms]	Kd-tree		Projection-based [ms]
		init [ms]	search [ms]	
307 200	35045	106	660	3
76800	2188	23	130	1
19200	139	5	25	1
4800	8	1	6	1

Timing

Correspondence Estimation 1/2

```
#include <pcl/registration/correspondence_estimation.h>

  pcl::registration::CorrespondenceEstimation<pcl::PointXYZ, pcl::PointXYZ> corresp_kdtree;
  corresp_kdtree.setInputSource (cloud_src);
  corresp_kdtree.setInputTarget (cloud_tgt);
  pcl::Correspondences correspondences;
  corresp_kdtree.determineCorrespondences (correspondences, 0.05f);
```

```
#include <pcl/registration/correspondence_estimation_organized_projection.h>

  pcl::registration::CorrespondenceEstimationOrganizedProjection<pcl::PointXYZ, pcl::PointXYZ> organized_projection;
  organized_projection.setInputSource (cloud_src);
  organized_projection.setInputTarget (cloud_tgt);
  organized_projection.setFocalLengths (525.f, 525.f);
  organized_projection.setDepthThreshold (0.03f);
  organized_projection.determineCorrespondences (correspondences, 0.05f);
```

Correspondence Estimation 2/2

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

Options:

- distance (CorrespondenceRejectorDistance)
- median distance (CorrespondenceRejectorMedianDistance)
- one to one (CorrespondenceRejectorOneToOne)
- sample consensus (CorrespondenceRejectorSampleConsensus)
- sample consensus 2D (CorrespondenceRejectorSampleConsensus2D)
- surface normal (CorrespondenceRejectorSurfaceNormal)
- boundary points (CorrespondenceRejectorOrganizedBoundary)

- trimmed ICP (CorrespondenceRejectorTrimmed)
- another trimmed ICP (CorrespondenceRejectorVarTrimmed)
- geometric constraints (CorrespondenceRejectorPoly)

```
pcl::registration::CorrespondenceRejectorPoly<pcl::PointXYZ> rejector_poly;

rejector_poly.setCardinality (100);
rejector_poly.setIterations (1000);
rejector_poly.setSimilarityThreshold (0.75f);
rejector_poly.setInputSource (cloud_src);
rejector_poly.setInputTarget (cloud_tgt);
rejector_poly.setInputCorrespondences (correspondences);
rejector_poly.getCorrespondences (*correspondences_filtered);
```

Benchmarks

ONE	no filtering
TWO	median distance rejection (threshold = 2x median)
THREE	median distance + surface normal (threshold of 30 degrees)
FOUR	median distance + surface normal + reject correspondences that contain boundary points
FIVE	median distance + surface normal + boundary points + one to one
SIX	median distance + surface normal + boundary points + one to one + sample consensus (1000 iterations with an inlier threshold of 5 cm)

Correspondence Rejection 3/6

Benchmarks

one source scan and three target scans:

- rotation of 1.3 degrees and 1 cm translation
- rotation of 4.3 degrees and 4.5 cm translation
- rotation of 8.3 degrees and 23 cm translation

Cloud pairs

1 - 2

1 - 3

1 - 4

iteration vs percentage that passed filtering

iteration vs MSE

Incremental registration

cloud number vs angle RMSE

cloud number vs translation RMSE

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

Error metrics:

- point to point: $$\sum_{i=1}^{N} ||R\,p_i + t - q_i||^2$$
- point to plane: $$\sum_{i=1}^{N} ((R\,p_i + t) - q_i) \cdot n_{q_i})^2$$

```
#include <pcl/registration/
transformation_estimation_point_to_plane_weighted.h>


pcl::registration::TransformationEstimationPointToPlaneWeighted<pcl
::PointXYZ, pcl::PointNormal, double> te;
  te.setWeights (correspondence_weights);
  te.estimateRigidTransformation (*cloud_src, *cloud_tgt_normals,
*correspondences_filtered, transform);
```

Methods:

- Point to plane Linear Least Squares (TransformationEstimationPointToPlaneLLS)
- Point to plane LLS weighted (TransformationEstimationPointToPlaneLLSWeighted)
- SVD (TransformationEstimationSVD)
- Point to point LM (TransformationEstimationLM)
- Point to plane LM (TransformationEstimationPointToPlane)
- Point to plane LM weighted (TransformationEstimationPointToPlaneWeighted)

Transformation Estimation 1/3

Benchmarks

Same 3 pairs of clouds

3 types of correspondences:

- noisy
- clean (filtering based on normals)
- clean + randomization

iterations vs MSE

weighting based on the camera noise model:

$$w(p_i, q_i) = 0.0012 + 0.0019 * (max(p_{i_z}, q_{i_z}) - 0.4)^2$$

iterations vs MSE

Transformation Estimation 3/3

maximum number of iterations

maximum number of similar iterations

absolute transformation threshold

relative MSE threshold

relative transformation threshold

absolute MSE threshold

Stopping Criteria 1/2

The ICP loop:

```
pcl::registration::TransformationEstimationPointToPlane<pcl::PointXYZ, pcl::PointXYZ, double> te;
pcl::CorrespondencesPtr correspondences (new pcl::Correspondences ()),
    correspondences_filtered (new pcl::Correspondences ());
int iteration = 0;

Eigen::Matrix4d final_transform (Eigen::Matrix4d::Identity ()), transform;
pcl::registration::DefaultConvergenceCriteria<double> convergence_criteria (iteration, transform, *correspondences);

do
{
  pcl::transformPointCloudWithNormals (*cloud_src, *cloud_src_aligned, final_transform);

  getCorrespondencesClosestPoint (cloud_src_aligned,
                                  cloud_tgt,
                                  *correspondences);

  filterCorrespondences (correspondences,
                         cloud_src,
                         cloud_tgt,
                         *correspondences_filtered);

  te.estimateRigidTransformation (*cloud_src_aligned, *cloud_tgt, *correspondences_filtered, transform);

  /// Update variables
  final_transform = transform * final_transform;
  iteration ++;
} while (!convergence_criteria.hasConverged ());
```

Stopping Criteria 2/2

Additional criteria for validating a transformation:

- percentage of valid dexels
- percentage of inliers
- motion limit
- stability of the overlapping regions (condition number of the inliers)

Transformation Validation

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

Incremental error accumulation

incremental map

ground truth map

need additional links
in the graph

Graph Optimization 1/7

Graph representation

$$
\begin{aligned}
F(x) &= e_{12}^T \, \Omega_{12} \, e_{12} \\
&+ e_{23}^T \, \Omega_{23} \, e_{23} \\
&+ e_{25}^T \, \Omega_{25} \, e_{25} \\
&+ e_{43}^T \, \Omega_{43} \, e_{43}
\end{aligned}
$$

$$
\begin{aligned}
x^* &= argmin_x[F(x)] \\
F(x) &= \sum e(x_i, x_j, z_{ij})^T \Omega_{ij} e(x_i, x_j, z_{ij}) \\
e(x_i, x_j, z_{ij}) &= T(z_{ij}) T(x_i) T(x_j)^{-1}
\end{aligned}
$$

By linearization:

$$e_{ij}(\check{x}_i + \Delta x_i, \check{x}_j + \Delta x_j) = e_{ij}(\check{x} + \Delta x) \simeq e_{ij} + J_{ij} \, \Delta x$$

it reduces to:

$$H \Delta x^* = -b$$

and can be solved by Gauss-Newton or Levenberg-Marquardt

Ideally: N(N-1) edges in the graph

N edges from incremental registration

need more, how?

keyframes

thresholding in angle, xyz and time space

Loop Closure

Graph Optimization 4/7

Edge Weights

- uniform weighting

- isotropic weighting based on the pairwise registration
  - percentage of overlap between the frames
  - condition number of the overlapping points

- true covariance matrix of ICP
  - by perturbation

$$\left[\frac{\delta RMSE}{\delta x}, \frac{\delta RMSE}{\delta y}, \frac{\delta RMSE}{\delta z}, \frac{\delta RMSE}{\delta \alpha}, \frac{\delta RMSE}{\delta \beta}, \frac{\delta RMSE}{\delta \gamma}\right] \qquad C = J\,J^T$$

  - covariance as the cloud stability (Gelfand et al.)
  - ICP covariance in closed form (Censi et al.)

Split the graph when edges are not good enough

culprit frames

Graph Optimization 6/7

```
class GraphOptimizer
{
  public:
    GraphOptimizer ();

    void setAbsoluteErrorThreshold (const double thresh);
    double getAbsoluteErrorThreshold ();

    void setRelativeErrorThreshold (const double thresh);
    double getRelativeErrorThreshold ();

    void addCamera (const uint64_t node_id,
                    const Eigen::Matrix4d &node_to_world_transform = Eigen::Matrix4d::Identity (),
                    const bool is_fixed = false);

    void addCameraCameraConstraint (const uint64_t source_node_id, const uint64_t target_node_id,
                                    const Eigen::Matrix4d &source_to_target_transform,
                                    const Eigen::Matrix<double, 6, 6> &covariance_matrix);
    int optimize ();

    void getCameraTransforms (std::map<size_t, Eigen::Matrix4d> &poses);

    void getEdgeDeltas (std::vector<EdgeDelta> &edge_deltas);

    void removeBadEdges (const double translation_threshold, const double rotation_threshold);
};
```

Graph Optimization 7/7

```
template<typename PointT> void
CompleteGraphGeometricRegistration<PointT>::addNewPointCloud (typename CompleteGraphGeometricRegistration<PointT>::CloudConstPtr
cloud_new)
  {
    pcl::PointCloud<pcl::PointNormal>::Ptr cloud_processed (new pcl::PointCloud<pcl::PointNormal> ());
    pcl::IndicesPtr indices_non_nan (new std::vector<int> ());
    cloud_preprocessing_->preprocess (cloud_new, *cloud_processed, indices_non_nan);

    (...)

    /// Constant velocity assumption
    Eigen::Matrix4d initial_transform = Eigen::Matrix4d::Identity ();
    if (current_cloud_id_ >= 2)
      initial_transform = transforms_[current_cloud_id_ - 2].inverse () * transforms_[current_cloud_id_ - 1];

    Eigen::Matrix4d pair_transform;
    pcl::PointCloud<pcl::PointNormal>::Ptr cloud_aligned (new pcl::PointCloud<pcl::PointNormal> ());
    Eigen::Matrix<double, 6, 6> covariance_matrix;
    double alignment_score = 0.;
    bool align_ok = registration_->align (cloud_processed, cloud_previous_,
                                          indices_non_nan, indices_non_nan_previous_,
                                          cloud_aligned, pair_transform, alignment_score, covariance_matrix);

    if (align_ok)
    {
      /// Add the current frame to the graph
      size_t cloud_previous_id = transforms_.rbegin ()->first;
      Eigen::Matrix4d transf_abs = transforms_[cloud_previous_id] * pair_transform;
      graph_optimizer_->addCamera (current_cloud_id_, transf_abs);
      graph_optimizer_->addCameraCameraConstraint (current_cloud_id_, cloud_previous_id,
                                                   pair_transform, covariance_matrix);

      (...)
```

Everything put together

```
(...)

/// Check if it is a keyframe
 size_t last_keyframe_id = keyframes_.rbegin ()->first;
 Eigen::Matrix4d last_keyframe_trans = transforms_[last_keyframe_id];
 Eigen::Matrix3d rotation_diff = last_keyframe_trans.block<3, 3> (0, 0).transpose () * transf_abs.block<3, 3> (0, 0);
 Eigen::Vector3d translation_diff = last_keyframe_trans.block<3, 1> (0, 3) - transf_abs.block<3, 1> (0, 3);
 double angle_diff = Eigen::AngleAxisd (rotation_diff).angle ();
 double dist_diff = translation_diff.norm ();

 if (angle_diff > keyframe_rotation_threshold_ ||
     dist_diff > keyframe_translation_threshold_)
 {
   /// Try to register it with all the other keyframes
   for (std::map<size_t, pcl::PointCloud<pcl::PointNormal>::Ptr>::const_iterator m_it = keyframes_.begin ();
        m_it != keyframes_.end (); ++m_it)
   {
     /// Check for viewpoint directions
     Eigen::Vector3d cloud_src_viewpoint (0.0, 0.0, 1.0),
         cloud_tgt_viewpoint (0.0, 0.0, 1.0);
     cloud_src_viewpoint = transf_abs.block<3, 3> (0, 0) * cloud_src_viewpoint;
     cloud_tgt_viewpoint = transforms_[m_it->first].block (0, 0, 3, 3) * cloud_tgt_viewpoint;
     if (cloud_src_viewpoint.dot (cloud_tgt_viewpoint) < kf_viewpoint_angle_threshold_)
     {
       PCL_DEBUG ("Keyframe pair %zu - %zu rejected due to viewpoint angles being too wide.\n",
                  m_it->first, current_cloud_id_);
       continue;
     }

(...)
```

Everything put together

```
(...)
  /// Check for projection overlap
  pcl::PointCloud<pcl::PointNormal>::Ptr current_cloud_tr (new pcl::PointCloud<pcl::PointNormal> ());
  transformPointCloudWithNormals (*cloud_processed, *current_cloud_tr,
                                  (transforms_[m_it->first].inverse () * transf_abs).template cast<float> ());
  pcl::Correspondences corresps;
  registration_keyframes_->getCorrespondencesProjection  (current_cloud_tr,
                                                           m_it->second,
                                                           corresps);
  float overlap = static_cast<float> (corresps.size ()) / static_cast<float> (indices_non_nan->size ());
  if (overlap > keyframe_overlap_threshold_)
  {
    // Register the two clouds
    Eigen::Matrix<double, 6, 6> covariance_matrix;
    align_ok = registration_keyframes_->align (current_cloud_tr,
                                               m_it->second,
                                               indices_non_nan,
                                               keyframes_indices_[m_it->first],
                                               cloud_aligned,
                                               pair_transform,
                                               alignment_score,
                                               covariance_matrix);

    if (align_ok)
    {
      bool pass_validation = transformation_validation_->isValid (current_cloud_tr,
                                                                  m_it->second,
                                                                  pair_transform);
      if (!pass_validation)
      {
        PCL_ERROR ("Keyframes did not pass the transformation validation process\n");
        continue;
      }
  (...)
```

Everything put together

```
            (...)

             /// Add to graph
             graph_optimizer_->addCameraCameraConstraint (current_cloud_id_,
                                                          m_it->first,
                                                          pair_transform *
                                                          transforms_[m_it->first].inverse () * transf_abs,
                                                          covariance_matrix);
             PCL_DEBUG ("Added edge between keyframes %zu and %zu.\n", m_it->first, current_cloud_id_);
           }
         }

         graph_optimizer_->optimize ();
       }


       /// Add the current frame to the keyframes list
       keyframes_.insert (std::make_pair (current_cloud_id_, cloud_processed));
       keyframes_indices_.insert (std::make_pair (current_cloud_id_, indices_non_nan));
     }
   }

   *cloud_previous_ = *cloud_processed;
   *indices_non_nan_previous_ = *indices_non_nan;
   current_cloud_id_ ++;
   graph_optimizer_->optimize ();
   getTransforms (transforms_);
 }
```

Everything put together

$$argmin_{R_i,t_i}\Big(\sum_{clouds\ i}\sum_{clouds\ j}\sum_{points\ k}\sum_{points\ l}[[(R_i\,p_{i,k}+t_i)-(R_j\,p_{j,l}+t_j)]\cdot(R_j\,n_{j,l})\,w_{i,j,k,l}]^2\Big)$$

Dataset	Pose graph		Global ICP	
	Angular RMSE [degrees]	Translation RMSE [m]	Angular RMSE [degrees]	Translation RMSE [m]
freiburg1_360	12.93	0.337	9.76	0.201
freiburg1_desk	3.08	0.071	2.77	0.052
freiburg1_desk2	4.25	0.141	3.46	0.161
freiburg1_rpy	3.41	0.081	3.86	0.123
freiburg1_xyz	1.62	0.033	1.52	0.032
freiburg3_long_office_household	4.53	0.098	4.25	0.068

Global ICP

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

Poisson meshing with hallucinations

cleaned up

```
#include <pcl/surface/poisson.h>

  pcl::Poisson<pcl::PointNormal> poisson;
  poisson.setDepth (depth);
  poisson.setSolverDivide (solver_divide);
  poisson.setIsoDivide (iso_divide);
  poisson.setPointWeight (point_weight);
  poisson.setInputCloud (cloud_with_normals);
  pcl::PolygonMesh mesh;
  poisson.reconstruct (mesh);
```

Meshing and Texturing

transfer color

Vertex coloring

bad with compressed meshes

Meshing and Texturing

UV Mapping

trivial per-triangle

space-optimizing
trivial per-triangle

Maya automatic
UV mapping

UV Mapping

Color bleeding between triangles from different parts of the mesh

Need the texture to preserve the triangle neighborhoods!

affine transformation

perspective transformation

Meshing and Texturing

Weighting texel contributions:

- angle between the surface normal and the viewing direction of the camera
- distance of the 3D point on the mesh triangle to the camera center
- linear combination of the previous two
- pixel area of the mesh triangle in the camera image

no averaging

with averaging

Input Data
Filtering
Noise removal
Sampling
Normal Estimation
Point Cloud Registration
Correspondence Estimation
Correspondence Rejection
Transformation Estimation
Iterative Closest Point
Convergence and Validation
Planes
Graph Optimization
Meshing
Texturing

Geometric Features

- PFH, SHOT, FPFH, NARF, spin images etc.
- semantic mapping concepts:
  - cylinders
  - spheres
  - planes

very frequent in indoor mapping

Plane Extraction

1. Region growing

seed points + use normal, curvature, Euclidean distance to grow planar regions

```
  std::vector<pcl::ModelCoefficients> model_coefficients;
  pcl::PointCloud<pcl::Label>::Ptr labels (new pcl::PointCloud<pcl::Label>);
  std::vector<pcl::PointIndices> label_indices;
  std::vector<pcl::PointIndices> boundary_indices;
  typedef std::vector<pcl::PlanarRegion<pcl::PointXYZRGBNormal>,
Eigen::aligned_allocator<pcl::PlanarRegion<pcl::PointXYZRGBNormal> > > PlanarRegionsVector;
  PlanarRegionsVector regions;

  pcl::OrganizedMultiPlaneSegmentation<PointXYZRGBNormal, PointXYZRGBNormal, pcl::Label>
multiplane_segmentation;
  multiplane_segmentation.setMinInliers (1000);
  multiplane_segmentation.setAngularThreshold (pcl::deg2rad (1.0));
  multiplane_segmentation.setDistanceThreshold (0.02); //2cm
  multiplane_segmentation.setMaximumCurvature (0.1);
  multiplane_segmentation.setInputCloud (cloud);
  multiplane_segmentation.setInputNormals (cloud);
  multiplane_segmentation.segmentAndRefine (regions,
                                            model_coefficients,
                                            inlier_indices,
                                            labels,
                                            label_indices,
                                            boundary_indices);
```

2. Based on RANSAC

RANSAC to find plane inliers

clustering

refine plane parameters and inliers iteratively using PCA

concave hull of the points

simplify the contour

```
pcl::SampleConsensusModelPlane<pcl::PointXYZ>::Ptr model (new pcl::SampleConsensusModelPlane<pcl::PointXYZ> (cloud));
pcl::RandomSampleConsensus<pcl::PointXYZ> sac (model, 0.025);
sac.setMaxIterations (1500);

bool res = sac.computeModel ();
if (!res) {
  PCL_ERROR ("No planar model found. Relax thresholds or select object again and continue.\n");
  return (false);
}
res = sac.refineModel (3., 500);
if (!res) {
  PCL_ERROR ("No planar model found. Relax thresholds or select object again and continue.\n");
  return (false);
}

sac.getInliers (inlier_indices);
Eigen::VectorXf aux;
sac.getModelCoefficients (aux);
plane_params = aux.head<4> ();

/// Cluster the inlier indices to take only the contiguous plane
std::vector<pcl::PointIndices> clusters;
pcl::search::KdTree<PointXYZ>::Ptr tree (new pcl::search::KdTree<PointXYZ> ());
tree->setInputCloud (cloud, pcl::IndicesPtr (new std::vector<int> (inlier_indices)));
pcl::extractEuclideanClusters<PointT> (*cloud, inlier_indices, tree, 0.03, clusters);

/// Keep only the largest cluster
size_t max_cluster_size = 0, max_cluster_i = 0;
for (size_t i = 0; i < clusters.size (); ++i)
  if (clusters[i].indices.size () > max_cluster_size)
  {
    max_cluster_size = clusters[i].indices.size ();
    max_cluster_i = i;
  }
inlier_indices = clusters[max_cluster_i].indices;

/// Recompute the plane equation for this cluster only
float curvature_aux;
pcl::computePointNormal<PointT> (*cloud, inlier_indices, plane_params, curvature_aux);

/// Orient the normal towards the viewpoint
if (plane_params.head<3> ().dot (Eigen::Vector3f (0.f, 0.f, 1.f)) < 0.f)
  plane_params *= -1.0;
```

Plane Tracking

1. frame to frame
   - project inliers from one image to the other
   - threshold
2. frame to plane map
   - get candidates via equations
   - project source plane contour into target plane
   - compute intersection area
   - threshold

Plane Fusion

- weighted average of the plane parameters
- project source plane contour to target plane
- compute union of the contour polygons
- simplify the polygon

Planes as landmarks in the graph

$$e = \begin{bmatrix} R^T * \vec{n} \\ \vec{n} \circ t + d \end{bmatrix} - \begin{bmatrix} \vec{n}_m \\ d_m \end{bmatrix}$$

Observation edge - camera node to plane node

$$e = \vec{n_{src}}^T \, \vec{n_{tgt}} - \alpha$$

Manhattan edge - plane node to plane node

Before

After

Frame to frame alignment using planes

1. RANSAC and rendered depth maps

   1. choose first & second pairs at random
   2. check if the angle difference between the normals of the first pair is close to that of the second pair - if not, goto 1.
   3. choose third pair at random
   4. check if all the angles are consistent
   5. compute the transformation T using the 3 plane pairs
   6. render the source planes transformed by T
   7. compute the number of inliers between the rendered source and target maps
   8. if inliers > max, store the current transformation

Planes 7/8

Frame to frame alignment using planes

2. Joint Point and Plane Optimization

$$E = \alpha \sum_{i=0}^{N} [(p_i - q_i) \circ n_i]^2 + (1-\alpha) \sum_{i=0}^{M} w_i [(1.0 - n_{src_i} \circ n_{tgt_i}) + \beta(d_{src_i} - d_{tgt_j})]^2$$

3. Plane-Constrained ICP

reduce the dimensionality of the solution space from 6D to 3D (TransformationEstimation2D)

Results - SLAM

Results - SLAM

Results - SLAM

Results - SLAM

Results - SLAM

Results - SLAM

no planes

with planes

Results - SLAM

Results - Modeling

Results - Modeling

Results - Modeling

Thank you!

(for now)