CERN

Summer Student Programme 2021

รายงานการเข้าร่วมโครงการนักศึกษาภาคฤดูร้อน

ระหว่างวันที่ 5 กรกฎาคม - 27 สิงหาคม 2564

นายฐณพงศ์ ช่วงยรรยง

ภาควิชาวิศวกรรมหุ่นยนต์และระบบอัตโนมัติ

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าธนบุรี

Design and Implementation of a Point Cloud Registration Algorithm for CERN Robotics Framework (CRF)

Thanapong Chuangyanyong
cyy.thanapong@gmail.com

5th July, 2021 - 27th August, 2021

Contents

1 Acknowledgement

I cannot express enough gratitude to HRH Princess Maha Chakri Sirindhorn for selecting me as one of the Thai student representatives in the CERN Summer Student Programme 2021. Thank you, Thai-CERN Cooperation Project, and its supporting agency, e.g. Synchrotron Light Research Institute, and the National Science and Technology for continuous support, and for making this cooperation happens.

This research would not be possible without the supervision from Eloise Matheson and tremendous support from the Human-robot Interface (HRI) team members - Krzysztof Szczurek, Raúl Prades, Lucas Comte, José Nogueira, and Hugo Perier. Thank you, Alejandro Díaz Rosales, for advising me on the CERN Robotics Framework integration. Additionally, thank you, Norraphat Srimanobhas, for guidance and, with his team, managing the annual particle physics seminar in Thailand.

2 Introduction

This paper will explore the design and implementation of the new point cloud registration algorithm for CERN Robotics Framework (CRF) in C++. This project is initiated for the reason that, the old point cloud registration algorithm has a large codebase, and has numerous parameters to be tuned. These lead to an integration problem with the whole framework. Point cloud registration is the process of finding a transformation that aligns two point clouds. Here, the main use case is mapping and reconstructing a 3D environment. Therefore, the transformation comprises the 3D spatial rotation and the 3D spatial translation. The aligned point cloud should resemble the object or environment that sensors are trying to capture. In robotics, the transformation can be obtained from something like a sensor mounted on an end-effector of a manipulator. In fact, a Train Inspection Monorail (TIM) robot is an excellent candidate for mapping a tunnel at CERN. However, most of the time, like every mobile robotics application, the positioning sensor reading can be inaccurate or prone to integration drifting, and sensors noise. That is where the point cloud registration can enhance the accuracy of the robot's odometry. Moreover, the by-product of this action is a 3D map of the environment, which can benefit several Virtual Reality (VR), Augmented Reality (AR), or Mixed Reality (MR) researches. Nowadays, there are various researches about point cloud registration owing to a recent leap in the processing capability of microprocessors and affordability. Therefore, it is fitting to explore more point cloud registration algorithms; so that the new algorithm can improve upon existing ones.

2.1 Previous Works

2.1.1 Iterative Closest Point

ICP uses the least square optimisation to find the transformation that matches multiple captures of point cloud. After algebraic manipulation, e.g. relative translation, the problem can be solved with Singular Value Decomposition (SVD) seen in (2).

$$\mathbf{R}^* = \underset{\mathbf{R}\in\mathbf{SO}(3)}{\arg\min} \sum_i \mathbf{R}(\mathbf{y}_i - \bar{\mathbf{y}}) - (\mathbf{x}_i - \bar{\mathbf{x}}) \tag{1}$$

$$\begin{aligned} \mathbf{H} &= \sum_i (\mathbf{y}_i - \bar{\mathbf{y}})(\mathbf{x}_i - \bar{\mathbf{x}})^\top \\ SVD(\mathbf{H}) &= \mathbf{U}\mathbf{\Theta}\mathbf{V}^\top \\ \mathbf{R}^* &= \mathbf{V}\mathbf{U}^\top \end{aligned} \tag{2}$$

Where $\mathbf{x}_i$ and $\mathbf{y}_i$ are two sets of the point cloud that we are trying to match. $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are "centroids" of corresponding clouds. Nevertheless, with no way to know the actual correspondence, the algorithm needs to be solved iteratively, hence the "Iterative" in ICP. With the optimisation problem in (1), outliers can hugely impact the result. Moreover, Intel Realsense has an irregularly high magnitude noise floor. These contribute to the lower performance of the point cloud registration. The algorithm also does not perform well when matching a scene with a low overlapping area since the algorithm massively relies on the closest point as a correspondence.

To mitigate those shortcomings of the ICP algorithm. Several measures can further generalise the cost function to not be based on just the distances of (esitimated) correspondent points, such as the Coherent Point Drift (CPD) algorithm. For more information about Iterative Closest Point Algorithm, I recommend this lecture by Prof. Russ Tedrake from Massachusetts Institute of Technology.

2.1.2 Google's Cartographer

Google's Cartographer, based on [2], is a popular library among the open-source mobile robotics community. The library comes with great performance, demonstrated in various datasets. However, the problem is its dependencies and ease of integration. If I want to use it in the CRF many

dependencies are redundant e.g. Ceres Solver, the ROS PCL (not the same as the standalone PCL), and Protobuf (we only use a little portion of the library in `*.pbstream` file.) Google Cartographer seem promising in the video showcase that shows mapping in a huge environment like Deutsche Museum. Unfortunately, Cartographer is not well tested on the Intel Realsense Camera, and a 3D mapping application, generally. Besides, the fact that Google's Cartographer is deeply integrated with the Robot Operating System (ROS) make the integration with the CERN Robotics Framework (CRF) more intricate.

2.1.3 SSL-SLAM

The algorithm in SSL-SLAM [5] uses edges and planes to register point clouds. In this paper, the more aggressive surface can be separated by calculating a difference in-depth for each point and its neighbours. This approach is implemented with a 3D solid-state LiDAR, specifically, Intel Realsense L515, which supplies an organized point cloud ($M \times N \times 3$) as an output, where M and N is the resolution of the recorded point cloud in the y-axis and the x-axis, respectively. However, with some adjustment, this approach can be used with an unorganized point cloud ($M \times 3$) which occupy less memory space (since it is dense) and supports more sensor types, where, now, M is the number of points in the point cloud. After separating edge points, what is left will be called "plane points." For every edge and plane point, the residual value can be calculated from neighbouring points of the same type in the target point cloud. This limit the correspondence of each point to be the same type of point, which in turn reduce the chance of being stuck in local minima.

2.1.4 More Interesting Algorithms

Git Repo	Video
SSL_SLAM	`https://www.youtube.com/watch?v=iiKew-wplAo`
A-LOAM	`https://www.youtube.com/watch?v=N4QF5VNuoxw`
Cartographer	`https://www.youtube.com/watch?v=uNlPf8SBlsk`
BLAM	`https://www.youtube.com/watch?v=08GTGfNneCI`
LIO-SLAM	`https://www.youtube.com/watch?v=BtQHSkydiS0`
RTAB_ROS	`https://www.youtube.com/watch?v=qpTS7kg9J3A`

3 Proposed Algorithm

The proposed algorithm is based on the algorithm in [5] and another research on edge extraction algorithm for unorganized point cloud [1].

3.1 Edge Extraction

A covariance matrix is a symmetric, positive-semidefinite, square matrix that is used to explain variance in multiple dimensions. These research papers, [3], [4], [1], use eigenanalysis of the covariance matrix of neighbouring points to calculate a surface variation value of the surface.

First, for each point ($\mathbf{p}_k$), the covariance matrix is constructed from a set of its neighbouring point ($\mathcal{P}_k^n$), where $\mathcal{P}_k^n = \{\mathbf{p}_1^n, \mathbf{p}_2^n, \mathbf{p}_3^n, \ldots, \mathbf{p}_N^n\} \forall k \in \mathcal{K}$, N is the number of neighbouring points, and $\mathcal{K}$ is the index set of each point in the target point cloud. The covariance matrix of $\mathcal{P}_k^n$ can be computed by

$$\mathbf{C}(\mathcal{P}_k^n) = \begin{bmatrix} cov(x,x) & cov(y,x) & cov(z,x) \\ cov(x,y) & cov(y,y) & cov(z,y) \\ cov(x,z) & cov(y,z) & cov(z,z) \end{bmatrix} \tag{3}$$

$$cov(a,b) = \frac{1}{N}\sum_{i=1}^{N}((p_i^{n,a} - \bar{p}_i^{n,a})(p_i^{n,b} - \bar{p}_i^{n,b})) \tag{4}$$

where $p_{(\cdot)}^{n,a}$ is an a element of $\mathbf{p}_{(\cdot)}^{n}$ and in this case, $a, b \in \{x, y, z\}$. The eigenvalues of this 3-by-3 matrix can be computed. Now, three eigenvalues are obtained from this matrix denoted by $\lambda_0(\mathbf{C}(\cdot))$, $\lambda_1(\mathbf{C}(\cdot))$, $\lambda_2(\mathbf{C}(\cdot))$ where $\lambda_0(\mathbf{C}(\cdot)) \leq \lambda_1(\mathbf{C}(\cdot)) \leq \lambda_2(\mathbf{C}(\cdot))$. Lastly, the surface variation (σ_i) of the point $\mathbf{p}_k$ can be calculated from

$$\sigma_i(\mathcal{P}_k^n) = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2} \tag{5}$$

Then, this value, σ_i can be used to classify between edge points and plane points, in this application. A point with a high σ_i value indicates a sharper or rougher surface.

3.2 Computing Residual

After computing all the surface variation in the input point cloud $(\mathcal{P}_i)$, the input point cloud can be separated to the edge cloud $(\mathcal{P}_i^{\varepsilon})$ and the plane cloud $(\mathcal{P}_i^{\rho})$. The residual function is taken from [5] as seen below.

$$f_{\varepsilon}(\mathbf{p}) = \frac{\|(\mathbf{p} - \mathbf{p}_2^{\varepsilon}) \times (\mathbf{p} - \mathbf{p}_1^{\varepsilon})\|}{\|\mathbf{p}_1^{\varepsilon} - \mathbf{p}_2^{\varepsilon}\|} \tag{6}$$

$$f_{\rho}(\mathbf{p}) = \left\| (\mathbf{p} - \mathbf{p}_1^{\rho})^{\top} \cdot \frac{(\mathbf{p}_1^{\rho} - \mathbf{p}_2^{\rho}) \times (\mathbf{p}_1^{\rho} - \mathbf{p}_3^{\rho})}{\|(\mathbf{p}_1^{\rho} - \mathbf{p}_2^{\rho}) \times (\mathbf{p}_1^{\rho} - \mathbf{p}_3^{\rho})\|} \right\| \tag{7}$$

$f_{\varepsilon}(\cdot)$ and $f_{\rho}(\cdot)$ is the residual function of the edge point and the plane point, respectively. Note that the input of the residual function must be the transformed point. $\mathbf{p}_{(\cdot)}^{\varepsilon}$, and $\mathbf{p}_{(\cdot)}^{\rho}$ are the neighbouring points of an edge point and the neighbouring points of a plane point, respectively ($\mathbf{p}_{(\cdot)}^{\varepsilon}$, and $\mathbf{p}_{(\cdot)}^{\rho}$ belong to the target point cloud). These neighbouring points are obtained from K-D trees of the target point cloud by using a transformed point in the source point cloud as a searching point.

3.3 Cosntructing a Non-linear Least Square Problem

To correctly register the input point cloud $(\mathcal{P})$, the transformation from the input point cloud's local coordinate frame to the target point cloud's local coordinate frame must be known. The result of the residual functions from the last section can be minimised to find the optimal transformation. Now, the cost function for the optimisation problem can be constructed.

$$\langle \mathbf{t}_i^*, \mathbf{R}_i^* \rangle = \underset{\mathbf{t}_i \in \mathbb{R}^3, \mathbf{R}_i \in \mathbf{SO}(3)}{\arg\min} \sum_{\mathbf{p} \in \mathcal{P}_i^{\varepsilon}} f_{\varepsilon}(\mathbf{R}_i\mathbf{p} + \mathbf{t}_i) + \sum_{\mathbf{p} \in \mathcal{P}_i^{\rho}} f_{\rho}(\mathbf{R}_i\mathbf{p} + \mathbf{t}_i) \tag{8}$$

Furthermore, Lie theory is used here to eliminate the constraint that would be present in the 3D rigid body rotation such as in Special Orthogonal Group ($\mathbf{SO}(3)$). The optimisation technique will be discussed again in section 4.2.

4 C++ Implementation

4.1 Edge Extraction

The code is rewritten from the code from [1] since it is unreadable, and contain many unused lines of code. The procedure is straightforward. First, a K Dimensional Tree (K-D Tree) is constructed from an input point cloud. In this case, the K-D tree has 3 dimensions - X, Y, and Z. Next, neighbouring points is obtained by searching in the K-D tree.

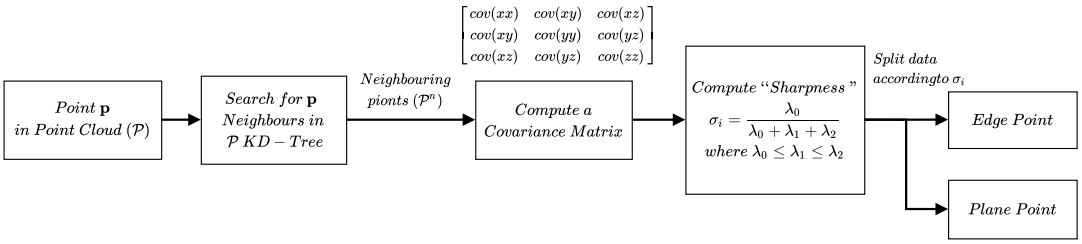


Figure 1: The edge extraction algorithm block diagram (only for a single point).

```
// Location: modules/Utility/VisionUtility2/.../Edge.hpp
for (std::size_t i = 0; i < inputCloud->points.size(); i++) {
    searchPoint = inputCloudXYZ->points[i];
    kdTree.nearestKSearch(searchPoint, kNeighbour, neighbourKdTree,
        neighbourKdTreeEuclideanDist);
```

This procedure will be applied to every point in the input point cloud, hence the for-loop. `kNeighbour` is the number of neighbouring points. The effect of varying `kNeighbour` can be seen is figs. 3. Next, from (4), the average position of each axis ($\bar{p}_i^{n,a}$ and $\bar{p}_i^{n,b}$) is computed.

```
// Location: modules/Utility/VisionUtility2/.../Edge.hpp
for (std::size_t i = 0; i < inputCloud->points.size(); i++) {
    ...
    for (std::size_t j = 0; j < neighbourKdTree.size(); j++) {
        pcl::PointXYZ neighbourPoint = inputCloudXYZ->points[neighbourKdTree[j]];
        xSum += neighbourPoint.x;
        ySum += neighbourPoint.y;
        zSum += neighbourPoint.z;
    }
    float xMean = xSum/kNeighbour;
    float yMean = ySum/kNeighbour;
    float zMean = zSum/kNeighbour;
```

After that, variances and covariances are computed and assigned to the covariance matrix.

```
// Location: modules/Utility/VisionUtility2/.../Edge.hpp
for (std::size_t i = 0; i < inputCloud->points.size(); i++) {
        ...
        for (std::size_t j = 0; j < neighbourKdTree.size(); j++) {
            pcl::PointXYZ neighbourPoint =
                inputCloudXYZ->points[neighbourKdTree[j]];

            // Compute variance of X, Y, and Z
            xxSum += (neighbourPoint.x - xMean) * (neighbourPoint.x - xMean);
            yySum += (neighbourPoint.y - yMean) * (neighbourPoint.y - yMean);
            zzSum += (neighbourPoint.z - zMean) * (neighbourPoint.z - zMean);

            // Compute covatiance of XY, YX, XZ, ZX, YZ, and ZY
            xySum += (neighbourPoint.x - xMean) * (neighbourPoint.y - yMean);
            xzSum += (neighbourPoint.x - xMean) * (neighbourPoint.z - zMean);
            yzSum += (neighbourPoint.y - yMean) * (neighbourPoint.z - zMean);
        }

        float varX = xxSum/kNeighbour;
        float varY = yySum/kNeighbour;
        float varZ = zzSum/kNeighbour;
        float covXY = xySum/kNeighbour;
        float covXZ = xzSum/kNeighbour;
        float covYZ = yzSum/kNeighbour;

        Eigen::Matrix3f covarianceMatrix;
        covarianceMatrix << varX, covXY, covXZ,
                            covXY, varY, covYZ,
                            covXZ, covYZ, varZ;
```

Lastly, eigenvalues are obtained from `Eigen::SelfAdjointEigenSolver` function. Eigenvalues are sorted and the surface variation value (σ_i) is calculated using (5). The result is stored in a newly created type of point called `PointXYZS`, and `S` can be abbreviated from sharpness, surface variation, or sigma.

```
// Location: modules/Utility/VisionUtility2/.../PointTypes.hpp
struct PointXYZS {
    PCL_ADD_POINT4D;
    float s;
    bool isEdge(float sigmaThreshold) {return s >= sigmaThreshold;}
    bool isPlane(float sigmaThreshold) {return s < sigmaThreshold;}
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW
} EIGEN_ALIGN16;
```

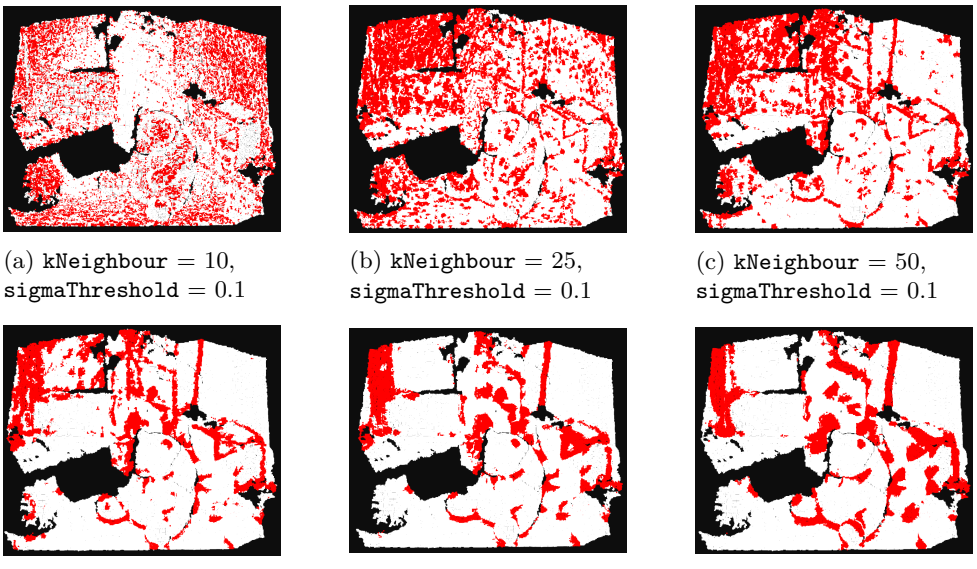
This point type has a method that can be used to separate the procedure between edge points and plane points, as seen below.

```
using namespace cern::utility::visionutility::pointcloud::pointtypes;
pointtypes::PointXYZS new_point(1.0, 3.0, 0.3, 0.04);
if (new_point.isEdge(0.025) {
    // calculate edge residual
} else {
    // calculate plane residual
}
```

In this case, `new_point` will be classified as an edge point since `S` (the last value in the constructor, 0.04) is greater than the threshold (0.025). The value that separate between edge and plane in the code is generally refers as a `simgaThreshold` (σ_t).

Finally, the algorithm is ready to extract edge features from a point cloud. To understand the effect of each parameter, edge point and plane point are separated by colour and plotted. The red coloured point is the former, and the whites are for the latter. In subfigure 3a, there are too few neighbours to calculate a surface variation value. The algorithm picks up little change in the surface as edges, this might be useful in measuring a surface roughness; however, not in finding edge features. For the higher `kNeighbour` value, edge features lose the definition and seem "blurred". Yet, the correlation between how edge features looks and the registration accuracy has not been investigated. However, the time execution time of the function is effect tremendously by the number of neighbours. A large number of neighbours increase the K-D tree's search time and covariance matrix computation time.

Figure 2: The original point cloud.



(a) `kNeighbour` = 10, `sigmaThreshold` = 0.1

(b) `kNeighbour` = 25, `sigmaThreshold` = 0.1

(c) `kNeighbour` = 50, `sigmaThreshold` = 0.1

(d) `kNeighbour` = 100, `sigmaThreshold` = 0.1

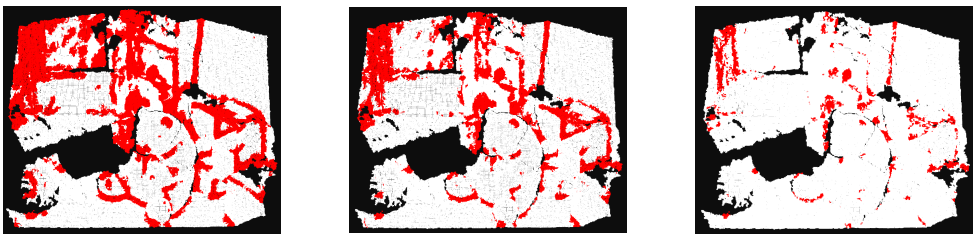
(e) `kNeighbour` = 200, `sigmaThreshold` = 0.1

(f) `kNeighbour` = 400, `sigmaThreshold` = 0.1

Figure 3: The comparison between multiple `kNeighbour` values

`sigmaThreshold` also affect the appearance of edge features as seen in fig. 4. When the `sigmThreshold` increases, edge features appear thinner. In contrast, when the `simgaThreshold` decreases, edge features appear thicker. This action does not seem to increase the background noise as much as changing the `kNeighbour` value.

When deploying the application, the `sigmaThreshold` and `kNeighbour` must be tuned individually. If both values are combined, the tuning procedure should be more trivial and may result in more consistent edge features between multiple frames, which lead to a better match. Another



(a) `kNeighbour` = 125, `sigmaThreshold` = 0.075

(b) `kNeighbour` = 125, `sigmaThreshold` = 0.1

(c) `kNeighbour` = 125, `sigmaThreshold` = 0.15

Figure 4: The comparison between multiple `sigmThreshold` values

easy way to potentially improve the tuning procedure is by normalising a sigmaThreshold value between the lower-bound and the upper-bound of the surface variation value. A correlation between those parameters and the number of edge points "blob" can also be used to improve the algorithm by automatically find the optimal parameters. Analysing an edge point cloud's morphological skeleton might give some insight too.

4.2 Computing Residual

In this part, the code is mostly rewritten from this repository [5] and Ceres Solver references. The repository is written for Robot Operating System (ROS). The repository implementation of this part is a bit questionable. For instance, the whole point cloud is transformed before adding a residual block to the Ceres Solver. This means, when the optimiser increments decision variables, the position of the neighbouring point indices **do not get updated**. Therefore, the author needs to iterate over the optimiser again, which is inefficient and slow. To visualise the problem, I interpret the author's original source code in algorithm 1.

Algorithm 1 SSL_SLAM

INPUTS: $\mathbf{R}_{init}$, $\mathbf{t}_{init}$, $\mathcal{P}$
OUTPUTS: $\mathbf{R}^*$, $\mathbf{t}^*$
$k \leftarrow 1$
$\mathbf{R}_1 \leftarrow \mathbf{R}_{init}$
$\mathbf{t}_1 \leftarrow \mathbf{t}_{init}$
while $k \leq N$ **do**
  for $\mathbf{p}_i \in \mathcal{P}$ **do**
    $\mathbf{p}'_i \leftarrow \mathbf{R}_k \mathbf{p}_i + \mathbf{t}_k$
    Find $\mathbf{p}'_i$ edge neighbours and plane neighbours.
    if $\mathbf{p}_i \in \mathcal{P}^\varepsilon$ **then**
      Add edge residual block $(f_\varepsilon(\mathbf{p}'_i), \nabla f_\varepsilon(\mathbf{p}'_i))$
    else if $\mathbf{p}_i \in \mathcal{P}^\rho$ **then**
      Add plane residual block $(f_\rho(\mathbf{p}'_i), \nabla f_\rho(\mathbf{p}'_i))$
    end if
  end for
  Solve for the optimal decision variable: $(\mathbf{R}^*_k, \mathbf{t}^*_k)$
  $\mathbf{R}_{k+1} \leftarrow \mathbf{R}^*_k$
  $\mathbf{t}_{k+1} \leftarrow \mathbf{t}^*_k$
  $k \leftarrow k + 1$
end while
$\mathbf{R}^* \leftarrow \mathbf{R}^*_k$
$\mathbf{t}^* \leftarrow \mathbf{t}^*_t$

For our application, I rearrange the process such that, the transformation gets update correctly (algorithm 2). In contrary to the algorithm 1, our algorithm searches for the neighbouring

points every time the residual block is evaluated rather than searching for neighbours when once when residual block is added. This remove unnecessarily iteration over the nonlinear least square solver.

Algorithm 2 New Algorithm

INPUTS: $\mathbf{R}_{init},\ \mathbf{t}_{init}, \mathcal{P}$
OUTPUTS: $\mathbf{R}^*,\ \mathbf{t}^*$
$\mathbf{R} \leftarrow \mathbf{R}_{init}$
$\mathbf{t} \leftarrow \mathbf{t}_{init}$
for $\mathbf{p}_i \in \mathcal{P}$ **do**
  if $\mathbf{p}_i \in \mathcal{P}^{\varepsilon}$ **then**
    Add edge residual block $(f_{\varepsilon}(\mathbf{R}\mathbf{p}_i + \mathbf{t}), \nabla f_{\varepsilon}(\mathbf{R}\mathbf{p}_i + \mathbf{t}))$
  else if $\mathbf{p}_i \in \mathcal{P}^{\rho}$ **then**
    Add plane residual block $(f_{\rho}(\mathbf{R}\mathbf{p}_i + \mathbf{t}), \nabla f_{\rho}(\mathbf{R}\mathbf{p}_i + \mathbf{t}))$
  end if
end for
Solve for the optimal decision variable: ($\mathbf{R}^*$, $\mathbf{t}^*$)
(find neighbours every time the residual block is called)

Note that, in the 2, the transformation of the current point happens **internally** every single function call. Although, the original algorithm has few advantages, such as if the residual value is invalid, the residual block is discarded in advance. In contrary, this approach is also prone to edge cases, for instance when either the denominator of (6) or the denominator of (7) is near zero, the function may return a `NaN`. If the number of residual blocks in the Ceres Library is modifiable after the optimisation procedure is initiated, the fix would be trivial. Unfortunately, that is not the case. Hence, I implement a guard after the residual is calculated; in another word, the residual is checked first whether it is a `NaN` or not. If it is, the value is set to zero, since the occurrence of that edge case is relatively rare, and it is usually a few points against thousands. A better workaround in the future is appreciated. Finally, the algorithm can be seen in algorithm 3.

Algorithm 3 New Algorithm with Guards

INPUTS: $\mathbf{R}_{init},\ \mathbf{t}_{init}, \mathcal{P}$
OUTPUTS: $\mathbf{R}^*,\ \mathbf{t}^*$
$\mathbf{R} \leftarrow \mathbf{R}_{init}$
$\mathbf{t} \leftarrow \mathbf{t}_{init}$
for $\mathbf{p}_i \in \mathcal{P}_i$ **do**
  if $\mathbf{p}_i \in \mathcal{P}^{\varepsilon}$ **then**
    $\eta \leftarrow \langle f_{\varepsilon}(\mathbf{R}\mathbf{p}_i + \mathbf{t}), f_{\varepsilon}(\mathbf{R}\mathbf{p}_i + \mathbf{t}) \rangle$
    if η is `NaN` **then**
      $\eta = \langle 0, \mathbf{0} \rangle$
    end if
    Add edge residual block (η)
  else if $\mathbf{p}_i \in \mathcal{P}^{\rho}$ **then**
    $\eta \leftarrow \langle f_{\rho}(\mathbf{R}\mathbf{p}_i + \mathbf{t}), \nabla f_{\rho}(\mathbf{R}\mathbf{p}_i + \mathbf{t}) \rangle$
    if η is `NaN` **then**
      $\eta = \langle 0, \mathbf{0} \rangle$
    end if
    Add plane residual block (η)
  end if
end for
Solve for the optimal decision variable: ($\mathbf{R}^*$, $\mathbf{t}^*$)
(find neighbours every time the residual block is called)

4.3 Solving a Nonlinear Least Square Problem

After adding all the residuals, the nonlinear least square problem must be solved. The Levenberg-Marquardt algorithm from Google's Ceres Solver is used here due to its ability to solve a large-scale nonlinear least square problem fast. Furthermore, the library is specifically designed for this type of application e.g. point cloud registration, and pose graph optimisation. In fact, Google's Cartographer is also implemented with the Ceres Solver. To optimise the 3D spatial transformation of a rigid body there are two parts of the decision variable, namely orientation, and translation. The translation is simple; however, the 3D orientation must be incremented by a rotation matrix. Though this comes with a caveat, as stated in section 3.3, a rotation matrix is a $\mathbf{SO}(3)$ group. To be considered as a $\mathbf{SO}(3)$ group the matrix shall inherit these properties

$$\mathbf{A} \in \mathbb{R}^{3\times3}, \quad \mathbf{A}^\top\mathbf{A} = \mathbf{A}\mathbf{A}^\top = \mathbb{I}^3, \text{ and } \det\mathbf{A} = 1 \tag{9}$$

This can add complexity to the problem because with the cost function from (8), the problem becomes nonlinear constrained optimisation. This problem can be alleviated by using a Lie group of the $\mathbf{SO}(3)$ group, $\mathfrak{so}(3)$ group. This Lie group is also a 3 by 3 matrix ($\mathfrak{so}(3) \in \mathbb{R}^{3\times3}$) and is a skew-symmetric matrix. This can help us because the matrix exponential of any member of a Lie group of $\mathbf{SO}(3)$ is guaranteed to be a rotation matrix. Moreover, any Lie group has a tangent space, which in this case, the space can be represented in a three-element vector (12). This vector can be utilised in the optimisation procedure. Now, the incrementation of the decision variables can be computed by a vector in the tangent space of the Lie group $\mathfrak{so}(3)$ (ω), and a change in the position ($\Delta\mathbf{t}$). The relationship between the tangent-space vector and the Lie group can be seen in (12) and (13). For more information about Lie theory in robotics, I recommend this lecture by Prof. Joan Solà from Universitat Politècnica de Catalunya.

$$\mathbf{R}' = e^{[\omega]_\times}\mathbf{R} \tag{10}$$

$$\mathbf{t}' = \mathbf{R}'\Delta\mathbf{t} + \mathbf{t} \tag{11}$$

$$\omega = \begin{bmatrix}\omega_x \\ \omega_y \\ \omega_z\end{bmatrix} \tag{12}$$

$$[\omega]_\times = \begin{bmatrix}0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0\end{bmatrix} \tag{13}$$

3-by-3 skew-symmetric matrix can be created from a vector easily by using the Eigen library. I wrote a simple function called `cpm3()` (cross-product matrix 3D) in `MathUtility` for this purpose.

```
// Location: modules/Applications/PcMapper/src/SigmaSolver.cpp
namespace linalg = cern::utility::mathutility::linearalgebra;
// Exponential mapping
delta_q = Eigen::Quaterniond((linalg::cpm3(omega)).exp());
// Apply exponential map to the existing quaternion
quater_plus = (delta_q * quater).normalized();
trans_plus = (quater_plus * delta_t) + trans;
```

Even though the optimisation problem in (8) uses a rotation matrix to rotate a point cloud, it is not efficient to use 9 values to explain a three-dimensional orientation. The rotation matrix can also suffer from accumulated errors for the same reason. Euler angles are off the choice since they can suffer from singularities. Therefore, a unit quaternion is used in the implementation in place of a rotation matrix. Be aware that the multiplication of the quaternion in the code is **not** literal, this is merely a syntax in the Eigen library. The quaternion can be constructed with a

rotation matrix; so, `delta_q` is just $e^{[\omega]_\times}$ from (10) in the form of a unit quaternion. In [5], the authors increment the decision variables by using the Euler angles to rotate a unit quaternion. The implementation here is cleaner, faster, and the final cost is lower.

5 Preliminary Result

Unfortunately, I do not have any time left to do an in-depth comparison between various existing algorithms, or comparing results with established scores. Therefore, I compare the `SigmaSolver` with the PCL's built-in Iterative Closest Point Algorithm. The input is a pair of point clouds that subsampled to around 1,600 points. Then, the source point cloud is artificially transformed by rotating $-0.6\,\text{rad}$ in the z-axis and translate $1\,\text{m}$ in the x-axis with respect to the global coordinate frame. The exact transformation matrix in (14), which resulted in fig. 5. ICP, and `SigmaSolver` use their default value.

$$\mathbf{T}_a = \begin{bmatrix} \cos(0.6) & -\sin(0.6) & 0 & 1 \\ \sin(0.6) & \cos(0.6) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{14}$$

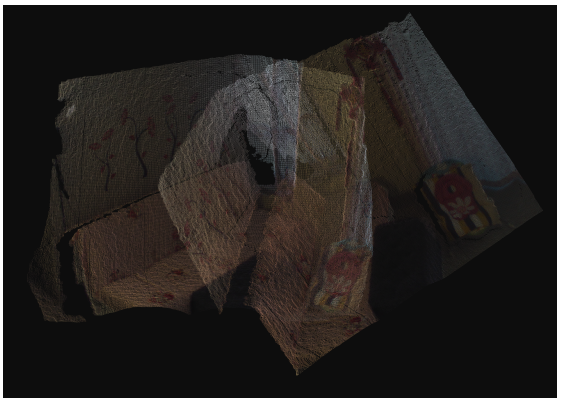


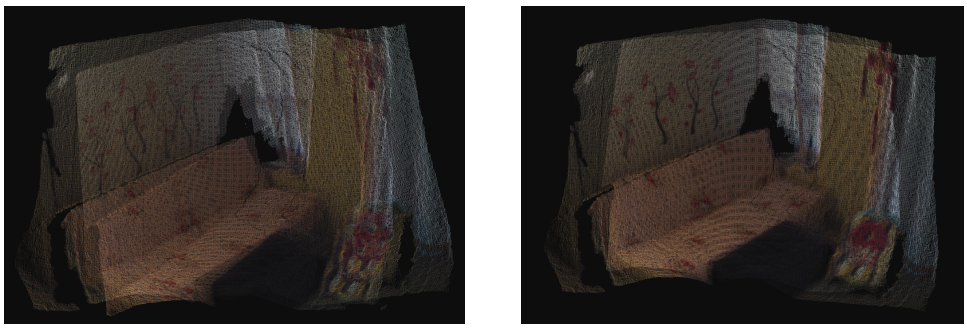
Figure 5: A matching demonstration with a fake transformation.

Next, I plot the result with a transformation that is obtained from (8). Essentially, it is just transforming a local coordinate frame of the source point cloud ($\mathcal{P}_2$) to the local coordinate frame of the target point cloud ($\mathcal{P}_1$). Until here, we use $\mathbf{Rp}+\mathbf{t}$ to transform the point cloud. However, in the implementation, the transformation is usually stored as a homogeneous transformation matrix, which can also transform the point cloud and equivalent to $\mathbf{Rp}+\mathbf{t}$. The equation can be seen in (15).

$$\mathbf{p}' = \begin{bmatrix} \mathbf{R}^* & \mathbf{t}^* \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \quad \forall \mathbf{p} \in \mathcal{P}_2 \tag{15}$$

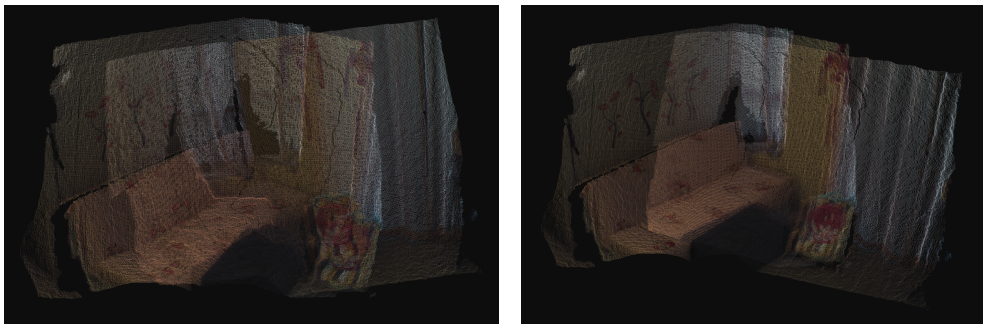
```
#include <pcl/common/transforms.hpp>
pcl::PointCloud<pcl::PointXYZRGB> P1;
pcl::PointCloud<pcl::PointXYZRGB> P1_prime;

pcl::transformPointCloud(P1, P1_prime, tf_optimal); //tf_optimal is an Eigen::Affine3d
```

ICP took $0.012\,\text{s}$, and `SigmaSolver` took $0.097\,\text{s}$. Around $80\,\%$ of the execution time of `SigmaSolver` came from the edge extraction process. Therefore, if real-time behaviour is desired, the improvement in the edge extraction speed should be prioritised. Now, we push the algorithm



(a) Matching done with the ICP (b) Matching done with the `SigmaSolver`

Figure 6: The comparison between ICP and `SigmaSolver`



(a) Matching done with the ICP (b) Matching done with the `SigmaSolver`

Figure 7: The comparison between ICP and `SigmaSolver` with a point cloud that is further away.

harder by using a source point cloud with less overlap area since ICP usually perform poorly on this kind of application.

Lastly, I tried mapping sequentially without odometry data. Point clouds can be visualised by transforming each point cloud's local coordinate frame to the global coordinate frame, in this case, it is the local coordinate frame of the first point cloud. Algorithm 4 shows how to plot point clouds that are matched sequentially. The algorithm performs well, and the result can be seen in figs. 8 and 9.

Algorithm 4 Simple Sequential Mapping

$i \leftarrow 1$
$\mathbf{T} \leftarrow \mathbf{T}_{init}$
while $i \leq$ `pointCloudNum` **do**
  Solve for: $\mathbf{T}^*$ (from Algorithm 3)
  $\mathbf{T} \leftarrow \mathbf{T}\mathbf{T}^*$
  for all $\mathbf{p} \in \mathcal{P}_i$ and $\mathbf{p}' \in \mathcal{P}'_i$ **do**
    $[\mathbf{p}'\ 1]^\top \leftarrow \mathbf{T}[\mathbf{p}\ 1]^\top$
  end for
  Plot: $\mathcal{P}'_i$
  $i \leftarrow i + 1$
end while

One last note, matching between edge features help the point cloud registration algorithm by artificially create a correspondence guide for point clouds; e.g. edge point can not be matched with a plane point, vice versa. This can be exploited a little more by using a plane fitting technique to add another means of creating correspondence from semantics features.

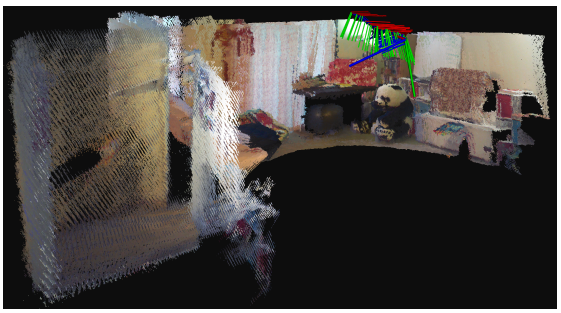


Figure 8: A simple mapping of a living room.

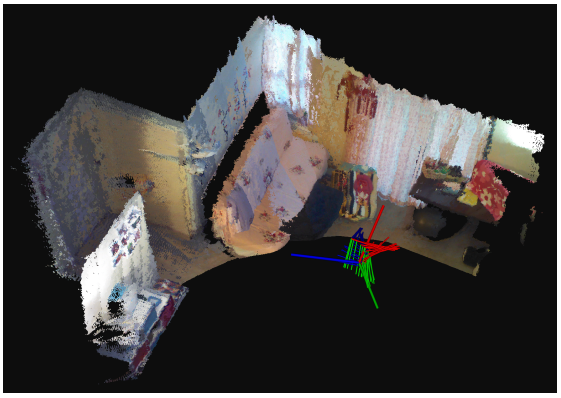


Figure 9: A simple mapping of a living room.

6 Conclusion

The new point cloud registration algorithm is designed and implemented. The algorithm uses edge features of the input point clouds to solve for the optimal transformation matrix. The proposed algorithm performs well compared to the Point Cloud Library's ICP algorithm; however, with more computation time. This algorithm is implemented in C++, then integrated into the CERN Robotics Framework. The resulting codebase needs fewer parameters compared to the predecessor (3DMapper). However, the algorithm has not been extensively tested and benchmarked with many other algorithms, which is advised to do so, in the future. The proposed algorithm can be further improved in several ways. First, optimising the edge extraction algorithm can make this algorithm runs substantially faster since the most computational time is taken in the edge extraction process. The plane can be fitted to further separate the point cloud correspondence which in turn should reduce the chance of the optimiser stuck in local minima. This can also be done to edge points by separating them by the blob. Lastly, the residual function can be simplified. In the current state, the residual function is taken from [5], which is hard to compute and nonlinear. The closest point error as in ICP or probabilistic method as in the coherent point drift might give a comparable result with better time complexity.

7 PcMapper Module Documentation

A copy of the repository's `README.md` file.

The module comprises two main classes, `PcSolver` (Point Cloud Solver), and `PcMappingManager` (Point Cloud Mapping Manager). `PcSolver` class only has one purpose, get the transformation matrix that transforms the coordinate frame of the source point cloud to the coordinate frame of the target point cloud. However, `PcSolver` is just a base class, to solve for the transformation, one of the derived classes must be used. For example, `SigmaSolver` uses edge features to match between two point clouds. It is possible to write more solvers and put them into the `/include/solver` folder.

The `PcMappingManager` does a high-level mapping procedure, for instance, integrating transformations of each scan to get the current pose (useful for mobile robotics). There is not much feature in this class at the moment. Though, in the future, something like pose graph optimisation, occupancy grid construction, or pose extrapolation can be implemented here (functions that are helpful in mapping/SLAM and invariant to the solver algorithm).

7.1 Dependencies

7.1.1 External

- Ceres Solver: Solving a nonlinear least square problem
- Point Cloud Library (pcl): Point cloud manipulation
- Eigen: Linear algebr
- Niels Nlohmann's JSON: Configuration file

7.1.2 Modules, Utilities

- MathUtility: Converting a vector to a skew-symmetric matrix
- VisionUtility2: Point cloud's edge extraction
- EventLogger: Logging

7.2 Example Usage

Create a `PcMappingManager` object for `PointXYZRGB` point type.

```
#include "PcMapper/PcMappingManager.hpp"
#include "PcMapper/SigmaSolver.hpp"

namespace pcmapper = cern::applications::pcmapper;

pcmapper::PcMappingManager<pcl::PointXYZRGB> manager;
```

Set the `SigmaSolver` as a solver. Then, load the configuration file.

```
manager.setSolver(new pcmapper::SigmaSolver<pcl::PointXYZRGB, pcl::PointXYZRGB>);
manager.loadConfigFile(argv[2]);
```

Setup the first point cloud with an initial transformation from the global coordinate frame (`tf_initial`). However, the initial frame also can be set automatically if the buffer is empty.

```
manager.setInitialFrame(p_pc_a, tf_initial);
```

Append point cloud in the buffer. This internally solves for the transformation matrix

```
manager.appendPointCloud(p_pc_b, tf_guess);
```

Now, you can get the current transformation (a transformation matrix between the current point cloud and the last one) by calling

```
Eigen::Affine3d output = manager.getCurrentTransformation()
```

or get the current pose (a transformation matrix between the current point cloud and the global coordinate frame)

```
Eigen::Affine3d output = manager.getCurrentPose()
```

or get every transformation

```
std::deque<Eigen::Affine3d> output = manager.getTransformationList()
```

7.2.1 PcSolver Standalone Usage

In fact, the `PcSolver` derived classes are designed to be a standalone application. The usage is similar to the PCL's ICP. First, set the source and target point clouds.

```
#include "PcMapper/SigmaSolver.hpp"

namespace pcmapper = cern::applications::pcmapper;

pcmapper::SigmaSolver<pcl::PointXYZRGB, pcl::PointXYZRGB> solver;

solver.setInputSource(point_cloud_1);
solver.setInputTarget(point_cloud_2);
```

We have not loaded the configuration, it should use the default values. Then, we can get a transformation by running:

```
Eigen::Affine3d ouput = solver.solveTransform(transformation_guess);
```

7.3 Configuration File Format (JSON)

7.3.1 PcSolver Derived Classes

Configuration must have a `"pc_solver"` field, and in that file, there must exist a `"type"` field that has the same value as the class name in snake-case. For example, if we have a `GenericSolver` class, the JSON file must contain

```
{
    "pc_solver": {
        "type": "generic_solver",
        ...
    }
}
```

because we can not guarantee the same configuration across various `PcSolver` derived classes. Another field that should be filled is the `"name"` field; so that we can extinguish between other configurations in runtime.

7.3.2 PcMappingManager

Currently, PcMappingManager does not have any important configs. The JSON file must have a `"pc_mapping_manager"` field. Currently, `"window_size"` just increases the buffer (sliding window) size and `"extrapolation"` is for the future.

References

[1] Dena Bazazian, Josep R. Casas, and Javier Ruiz-Hidalgo. Fast and robust edge extraction in unorganized point clouds. In *2015 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–8, 2015.

[2] Wolfgang Hess, Damon Kohler, Holger Rapp, and Daniel Andor. Real-time loop closure in 2d lidar slam. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1271–1278, 2016.

[3] M. Pauly, M. Gross, and L.P. Kobbelt. Efficient simplification of point-sampled surfaces. In *IEEE Visualization, 2002. VIS 2002.* IEEE.

[4] Mark Pauly, Richard Keiser, and Markus Gross. Multi-scale feature extraction on point-sampled surfaces. *Computer Graphics Forum*, 22(3):281–289, September 2003.

[5] Han Wang, Chen Wang, and Lihua Xie. Lightweight 3-d localization and mapping for solid-state LiDAR. *IEEE Robotics and Automation Letters*, 6(2):1801–1807, April 2021.


CERN


ไทยเซิร์น
Siam Photon
สวทช
NSTDA
NARIT
สทน
TINT
สสวท
THEP
CENTER
กฟผ.
POWER OF INNOVATION
FOUNDATION