Optimization methods in Simultaneous Localization and Mapping

ZiChen Liu, Mentor: Yifu Tao

February 2024

Contents

1 Introduction

As the technological innovation in the field of computation and material science. Robots have shown it's advancement in many industrial field including medical operation, military research as well as self-driving car. Letting robots have an eyes and navigate itself in the environment have been a classical, fundamental and yet a challenging problem in the field of robotics, known as the Simultaneous localisation and mapping (SLAM) [3]. In general a well constructed robotic

system can be decompose into three section while performing an task, which are, perception, planning and action. Each playing an important role allow the robot to complete the task successfully.

The problems of SLAM also consist an cooperation of this three stage. In perception, different sensor choice may highly affect the final result. Camera and radar are two main categories used, each providing you with different data type, where monocular camera provide you with a 2d projection of the 3d space, with much more color information for object identification however suffered in determine the scale of the scene. While radar are scan by light beans gives you 3d information usually store in point cloud. There is other sensor include stereo and RGB-D camera however they are more advanced to modeling the data structure therefore not include.Then in planning stage, mobile robots use the data collected to estimate the structure of the environment as well as it's on position and motion in the space based on a sequence of algorithm which will be detailed discussed in the following section. Finally, At the action stage, the robot have to make sure it's operator mechanism perform the given plan at lowest delay and expected speed as precise as possible.

In this report, our main goal is to give a tour of the perception methods, introducing how to construct an odometry to calibrate the transformation and reconstruction from the LiDAR data using both non-linear and linear optimisation methods. And give some intuitive interpretation over the mathematical detail behind this problem.

2 Preliminaries

2.1 Rotation matrix and Lie algebra

2.1.1 Vector and spatial rotation/Rigid body transformation

For a robot observing the surroundings, it's natural to eliminate the representation of the surrounding and the robot itself to be rigid body (no extra moving in the scene in this early stage) where the motion can be described in a Cartesian coordinate with 3-axis x,y,z without any deformation. a point. In this specific problem, we have 2 set of base for linear space one represent of the world coordinate and the other for the robots coordinate $B := (\hat{i}, \hat{j}, \hat{k})$ Any point's location can be indicate given the vector $p := [x, y, z]^T \in \mathbb{R}^3$ with it's base as the how the dot product been calculate $d \cdot p$ (notice that the base is not addable).

In $\mathbb{R}^3$, we denote the relation between two product sign $\cdot$ and $\times$ as the fol-

lowing operation between two vectors.

$$a \cdot b := \sum_{i=1}^{3} a_i b_i \quad |a \cdot b| := |a||b|cos\theta \tag{1}$$

$$a \times b := \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} b \quad |a \cdot b| := |a||b|sin\theta \tag{2}$$

with θ denote the angle between the two angle. Notice that the dot product is a scalar value while cross product is defined by a skew-symmetric matrix of a, return a vector point at normal direction of the parallelogram extend by a,b and with a mode of surface area. This operation can only be defined in $\mathbb{R}^3$ and $\mathbb{R}^7$ I won't discuss this in detail, which is beyond the topic.

Now the robots is about to move in the 3d space. Lets denote the stationary world coordinate as B_w and robotic perception as B_r at the beginning of the time there are the same and all the operation are acting on B_r. In the case of translation of translation with a distance of $T \in \mathbb{R}^3$, a point of $p \in \mathbb{R}^3$ in any coordinate after this translation become $p' = p + T$.

The rotation is not straight forward as the translation, we start by consider the property of a rotation denote by R act on any vector p denote as $R(p)$, we know that a rotation is linear action therefore for vector thus R can be represent by a matrix $R \in \mathbb{R}^{n \times n}$

$$v, w \in \mathbb{R}^3 \quad \lambda \in \mathbb{R}$$
$$v + w = (Rv) + (Rw)$$
$$R(\lambda v) = \lambda Rv$$

we also knows that rotation are preserves in lengths and angles, this lead to no change in dot product after two vector under same transformation

$$\begin{aligned} v \cdot w &= v^T w \\ &= (Rv)^T (Rw) \\ &= (v^T R^T)(Rw) \\ &= v^T (R^T R) w \end{aligned} \tag{3}$$

observing that equation21 shows that $R^T R = I$ where I is a identity matrix, we then define it with notation of $O_{(n)} = \{R \in \mathbb{R}^{n \times n} | R^T R = I\}$ where O stand for orthogonal. Since the area of 2 vector won't be affected by rotation, $det(R)$ should be 1, this can be further notated as $SO_{(n)} = \{R \in O_{(n)}, det(R) = 1\}$, named as special orthogonal group. With both the rotation and translation any movement in space can be calculate by the following equation extended

in homogeneous coordinate in order to perform the whole transformation as a linear equation

$$p' = Rp + T \tag{4}$$

$$\begin{bmatrix} p' \\ 1 \end{bmatrix} = \begin{bmatrix} R & T \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} p \\ 1 \end{bmatrix} \tag{5}$$

the whole transformation M are define as a special euclidean group

$$SE(3) = \{M = \begin{bmatrix} R & T \\ 0^T & 1 \end{bmatrix} \in \mathbb{R}^{4\times4} | R \in SO(3), T \in \mathbb{R}^3\} \tag{6}$$

this representation of transformation is been consider as the linear projection of the rotation and translation in a homogeneous coordinate. which we can easily compute it's inverse as

$$M^{-1} = \begin{bmatrix} R^T & -R^T T \\ 0^T & 1 \end{bmatrix} \tag{7}$$

2.1.2 Euler angle and quaternion

This work doesn't require the use of this two representation, however understanding them will help you understand robot motion in other perspective. Euler angle consist three angle include pitch yaw and row to describe a rotation,it might encounter problem of singular value invariant quaternion is an extension of imaginary number which represent 4 dimensional space using 4 orthogonal base while in practical quatrain are usually used to represent rotation since only 4 flout need to store in the memory space of the robots.

2.1.3 Lie group and Lie algebra

To begin with, I have to admit that the idea of lie algebra is vase and complicated topic, and almost impossible for me to finish in this report, I will try to minimise the derive and going through some essential part of it in order to understand the purpose and benefit of using and applying it.

There is 16 variable needed for a matrix to represent a transformation of 6 degree of freedom. The natural restrain of orthogonality also trouble us when applied calculus to it, which is somewhat important in the world of engineering. Naturally, people are more comfortable doing things like optimization and interpolation for real number. Is there a way of applied the some mathematical system into different object in the real world application? This is the time for group theory show it's magic.

Intuitively, a group is generally a set of element obey a specific rule of arithmetic. For any mathematical object with the same properties, we been able to

applied all the methods developed previously onto it. The proper definition of a group G with an arithmetic operation $*$ follows the following rules:

$$closure : \forall g, h \in G \quad g * h \in G \tag{8}$$
$$associativity : \forall g, h, k \in G \quad (g * h) * k = g * (h * k) \tag{9}$$
$$identity : \exists e \in G \quad \forall g \in Ge * g = g * e = g \tag{10}$$
$$inverse : \forall g \in G \quad \exists h \in Gh * g = g * h = e \tag{11}$$

Lie group are groups with smooth and consecutive properties, we named those space manifolds.

Since Lie group are not closure on + operation, in order to compute calculus on the manifold, we need to define a further operation that allow us to do it, which by definition, given as a algebra for a specific lie group: which consist of a set $\mathbb{V}$ and a field $\mathbb{F}$, a lie algebra of lie group G is $(\mathbb{V}, \mathbb{F}, [,])$ denoted by $\mathfrak{g}$

$$closure : \forall X, Y \in \mathbb{V} \quad [X, Y] \in \mathbb{V} \tag{12}$$
$$bilinear : \forall X, Y, Z \in \mathbb{V} \quad a, b \in \mathbb{F} \quad [aX + bY, Z] = a[X, Z] + b[Y, Z], \quad [Z, aX + bY] = a[Z, X] + b[Z, Y] \tag{13}$$
$$reflexivity : \forall X \in \mathbb{V}, [X, X] = 0 \tag{14}$$
$$jacobian \quad equivalent : \forall X, Y, Z \in \mathbb{V} \quad [X, [X, Y]] + [Z, [Y, X]] + [Y, [Z, X]] = 0 \tag{15}$$

Intuitively lie algebra is a structure of the tangent space of it's lie group's at the identity. In order to find the lie algebra of our 3D transformation, we start by taking the constrain of the rotation:

$$\frac{\partial RR^T}{\partial t} = \frac{\partial I}{\partial t} \tag{16}$$
$$\dot{R}R^T + R\dot{R}^T = 0 \tag{17}$$
$$R^T\dot{R} = -(\dot{R}R^T)^T \tag{18}$$
$$\dot{R} = \begin{bmatrix} 0 & -w_3 & w_2 \\ w_3 & 0 & -w_1 \\ -w_2 & w_1 & 0 \end{bmatrix} R \tag{19}$$

denote the skew-symmetric matrix to be $w^\wedge$ and from matrix to vector be $w = (w^\wedge)^\vee$. notice that we need 3 more parameters of $m \in \mathbb{R}^3$ translation to linearised $SE(3)$ around the neighbourhood of identity.

$$X_{w_1} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \frac{\partial T}{\partial w_1}|_{(w,p)=(0^6)}$$

$$X_{w_2} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \frac{\partial T}{\partial w_2}|_{(w,p)=(0^6)}$$

$$X_{w_3} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \frac{\partial T}{\partial w_3}|_{(w,p)=(0^6)}$$

$$X_{m_1} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \frac{\partial T}{\partial m_1}|_{(w,p)=(0^6)}$$

$$X_{m_2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \frac{\partial T}{\partial m_2}|_{(w,p)=(0^6)}$$

$$X_{m_3} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \frac{\partial T}{\partial m_3}|_{(w,p)=(0^6)}$$

$$(w_1, w_2, w_3, m_1, m_2, m_3) \to (\delta w_1, \delta w_2, \delta w_3, \delta m_1, \delta m_2, \delta m_3)$$

$$(\delta w_1, \delta w_2, \delta w_3, \delta m_1, \delta m_2, \delta m_3) \to I_4 + \sum_{i=0}^{3} w_i X_{w_i} + \sum_{i=0}^{3} m_i X_{m_i}$$

$$X = \begin{bmatrix} w^\wedge & p \\ 0^T & 0 \end{bmatrix}$$

thus the lie algebra for $SE(3)$ can be defined as follow

$$\mathfrak{se}(3) = \{v = \begin{bmatrix} w \\ m \end{bmatrix} \in \mathbb{R}^6, v^\wedge = \begin{bmatrix} w^\wedge & p \\ 0^T & 0 \end{bmatrix} \in \mathbb{R}^{4\times 4}\}$$

given a small step of ϵ around identity $I + \epsilon v$ and iterative apply the group operation of X multiple times which gives, also know as the exponential map of the lie algebra

$$\lim_{k\to\infty} (I + \frac{1}{k} X)^k = \sum_{n=0}^{\infty} \frac{X^n}{n!} = exp(X)$$

$$exp(X) = \begin{bmatrix} \sum_{n=0}^{\infty} \frac{(w^\wedge)^n}{n!} & \sum_{n=0}^{\infty} \frac{(w^\wedge)^n}{(n+1)!} p \\ 0^T & 1 \end{bmatrix}$$

let $w = \theta a$ where $|a| = 1$, observing that:

$$a^\wedge a^\wedge = aa^T - I$$
$$a^\wedge a^\wedge a^\wedge = -a^\wedge$$

the first summation calculate by:

$$\begin{aligned}\sum_{n=0}^{\infty} \frac{(w^\wedge)^n}{n!} &= \sum_{n=0}^{\infty} \frac{(\theta a^\wedge)^n}{n!} \\ &= I + \theta a^\wedge + \frac{1}{2!}\theta^2 a^\wedge a^\wedge + \frac{1}{3!}\theta^3 a^\wedge a^\wedge a^\wedge + ... \\ &= aa^T - a^\wedge a^\wedge + \frac{1}{2!}\theta^2 a^\wedge a^\wedge - \frac{1}{3!}\theta^3 a^\wedge + ... \\ &= aa^T + (\theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - ...)a^\wedge - (1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - ...)a^\wedge a^\wedge \\ &= aa^T + sin\theta a^\wedge - cos\theta a^\wedge a^\wedge\end{aligned} \tag{20}$$

intuitively you can consider this as rotation of θ degree along the axis given by direction a further rearranging it gives the famous Rodrigues' rotation formula[4]:

$$exp(a^\wedge) = cos\theta I + (1 - cos\theta)aa^T + sin\theta a^\wedge$$

for the summation before the translation term:

$$\begin{aligned}\sum_{n=0}^{\infty} \frac{(w^\wedge)^n}{(n+1)!} &= \sum_{n=0}^{\infty} \frac{(\theta a^\wedge)^n}{(n+1)!} \\ &= I + \frac{1}{2!}\theta a^\wedge + \frac{1}{3!}\theta^2 a^\wedge a^\wedge + \frac{1}{4!}\theta^3 a^\wedge a^\wedge a^\wedge + ... \\ &= aa^T + \frac{a^\wedge}{\theta} - \frac{1}{\theta}(1 - \frac{1}{2!}\theta^2 + \frac{1}{4!}\theta^4 - ...)a^\wedge - \frac{1}{\theta}(\theta - \frac{1}{3!}\theta^3 + \frac{1}{5!}\theta^5 - ...)a^\wedge a^\wedge \\ &= aa^T + (\frac{1 - cos(\theta)}{\theta})a^\wedge - \frac{sin(\theta)}{\theta}a^\wedge a^\wedge\end{aligned} \tag{21}$$

we can defined this term as the left Jacobian:

$$J = (\frac{1 - cos(\theta)}{\theta})a^\wedge - (1 - \frac{sin(\theta)}{\theta})aa^T + \frac{sin(\theta)}{\theta}I$$

The inverse map of the exponential map is called logarithm map defined on Taylor expansion of space around identity to the tangent space, however in practice,

we can also calculate the rotation angle from the upper right component that determines the rotation using the Rodrigues' rotation formula:

$$\begin{aligned} tr(R) =& cos\theta tr(I) + (1 - cos\theta)tr(aa^T) + sin\theta tr(a^\wedge) \\ &= 1 + 2cos(\theta) \end{aligned} \tag{22}$$

$$\theta = arccos(\frac{tr(R) - 1}{2})$$

since $Ra = a$, a is the eigenvector of R, while $m = J^{-1}t$.

2.1.4 Derivative in Lie algebra

The next step is to construct calculus for our lie group based on the lie algebra representation. More specifically, the derivative of lie group since it will be a powerful tool in the construction of non-linear optimization.

For any $a, b \in \mathbb{R}$ we have $exp(a)exp(b) = exp(a + b)$, unfortunately, this equilibrium does not hold in the case of exponential map from lie algebra to lie group, Instead, we have what called Baker-Campbell-Hausdorff formula [1] for any $v, u \in \mathfrak{so}(3)$:

$$ln(exp(v)exp(u)) = v + u + \frac{1}{2}[u, v] + \frac{1}{12}[v, [v, u]] + \frac{1}{12}[u, [v, u]] + ... \tag{23}$$

which gives us the left linear approximation of:

$$\lim_{u \to 0} exp(u^\wedge)exp(v^\wedge) \approx exp((J^{-1}u + v)^\wedge) \tag{24}$$

the Jacobin J here is a more complex 6×6 matrix which is sightly different from the J above. By definion of derivative:

$$\begin{aligned} \frac{\partial exp(u^\wedge)p}{\partial u} &= \lim_{\delta u \to 0} \frac{exp((u + \delta u)^\wedge)p - exp(u^\wedge)p}{\delta u} \\ &= \lim_{\delta u \to 0} \frac{exp((J\delta u)^\wedge)exp(u^\wedge)p - exp(u^\wedge)p}{\delta u} \\ &\approx \lim_{\delta u \to 0} \frac{(I + (J\delta u)^\wedge)exp(u^\wedge)p - exp(u^\wedge)p}{\delta u} \\ &= \lim_{\delta u \to 0} \frac{(J\delta u)^\wedge exp(u^\wedge)p}{\delta u} \end{aligned} \tag{25}$$

notice that this finally leave us with a Jacobin and since $\wedge$ here is not satisfied cross product rearrangement, we introduce a more practical way of doing the derivative by multiply a left variation:

$$
\begin{aligned}
\frac{\partial exp(u^\wedge)p}{\partial u} &= \lim_{\delta u \to 0} \frac{exp(\delta u^\wedge)exp(u^\wedge)p - exp(u^\wedge)p}{\delta u} \\
&\approx \lim_{\delta u \to 0} \frac{(I + \delta u^\wedge)exp(u^\wedge)p - exp(u^\wedge)p}{\delta u} \\
&= \lim_{\delta u \to 0} \frac{\delta u^\wedge exp(u^\wedge)p}{\delta u} \\
&= \lim_{\delta u \to 0} \frac{\begin{bmatrix} \delta w^\wedge & \delta p \\ 0^T & 0 \end{bmatrix} \begin{bmatrix} Rp + t \\ 1 \end{bmatrix}}{\delta u} \\
&= \begin{bmatrix} I & -(Rp - t)^\wedge \\ 0^T & 0^T \end{bmatrix}
\end{aligned}
\tag{26}
$$

2.2 SVD

Singular value decomposition is a powerful tool in linear algebra, which allow you to represent a matrix in to smaller pieces and represent information more efficiently. Now lets assume you have a image represent by 1 in a 3 by 3 matrix, it's straight forward to reducing the 9 number into a product of 2 vector with six 1:

$$
\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \tag{27}
$$

however for a matrix that beyond 1 rank, we might need a linear combination to representing them. The methods mathematician come up with is the SVD. For any rectangular matrix $A \in \mathbb{R}^{m \times n}$ with rank $r \in \mathbb{Z}$ we take u to be the orthonormal basis and the eigen vector of AA^T, then the singular value σ^2 is been calculate with the first r's σ be the eigen value of A (as well as for A^T). considering single u_i

$$
AA^T u_i = \sigma_i^2 u_i \tag{28}
$$

notice that

$$
A^T(AA^T u_i) = A^T \sigma^2 u \tag{29}
$$

$$
(A^T A)(A^T u) = A^T \sigma^2 u \tag{30}
$$

to reduce the A^T beside u on both side, it's eiser to set

$$
u = {A^T}^{-1} v \tag{31}
$$

further more, since now v is just orthogonal not orthonomal, we use $u = {A^T}^{-1} \sigma v$

$$
(A^T A)(A^T {A^T}^{-1} \sigma v) = \sigma^3 A^T {A^T}^{-1} v \tag{32}
$$

$$
A^T A v = \sigma^2 v \tag{33}
$$

where v_i is now the eigenvector of $A^T A$. Rearranging the value into matrix form we obtian the following:

$$A \begin{bmatrix} v_1 & ... & v_r \end{bmatrix} = \begin{bmatrix} u_1 & ... & u_r \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & ... & \\ & & \sigma_r \end{bmatrix}$$

(34)

where the matrix for u is $U \in \mathbb{R}^{m \times r}$, matrix for v as $V \in \mathbb{R}^{n \times r}$ and $sigma$ to the diagonal matrix $\Sigma \in \mathbb{R}^{r \times r}$. since V is orthonormal, we obtian the decomposition by applied the two side onto a V^T:

$$A = U \Sigma V^T \tag{35}$$

3 experiment

3.1 Odometry

odometry's job is to track the translation of the robot based on it's chronological collected data (using lidar in this experiment)

For our experiment setup, we construct our linear optimisation methods using python language and test on an online data source of point cloud collected using lidar.[2] If you are interesting in the implementation please referring to the code base pose on the appendix at the end of this paper.

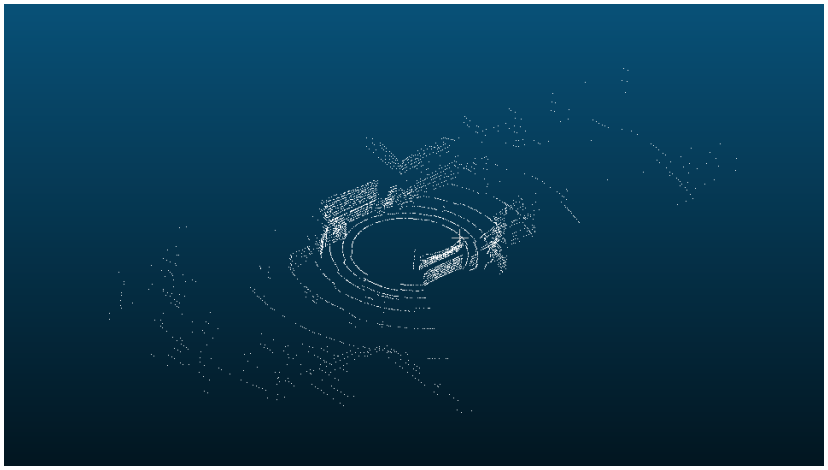

Figure 1: single frame of point cloud

Algorithm 1 odometry

Require: sequence of point cloud $P \in \mathbb{R}^{m \times n \times 3}$
  $base \leftarrow P_0$
  $map \leftarrow base$
  for $i \leftarrow 1$ to m **do**
    $T \leftarrow icp(base, P_i)$
    if fit well **then**
      $base \leftarrow T \cdot P_i$
      $map \leftarrow map + base$
    end if
  end for
  $a = 1$

3.2 3D-3D ICP

3.2.1 optimisation target

The optimisation target of ICP is to find best fit transformation in between two frames

3.2.2 non-linear optimization

our optimization target is given by:

$$\arg\min_{v} = \frac{1}{2} \sum_{i=1}^{n} ||p_i - exp(v^\wedge) p_i||_2^2 \tag{36}$$

Algorithm 2 non-linear optimization

Require: Randomly initialize parameter $v = [w, m]^T \in \mathbb{R}^6$, α is settled based on experiment
Ensure: $p \in \mathbb{R}^{n \times 3}, n \in \mathbb{Z}^+$
  $loss \leftarrow \frac{1}{2} \sum_{i=1}^{n} \sqrt{(p_i - exp(v^\wedge) p_i)^2}$
  while loss not converge **do**
    $v \leftarrow v - \alpha \frac{\partial loss}{\partial v}$
    $loss \leftarrow \frac{1}{2} \sum_{i=1}^{n} \sqrt{(p_i - exp(v^\wedge) p_i)^2}$
  end while

Once we have construct our mathematical approach in doing calculus for the transformation in the manifold, non-linear optimization is a rather straight forward approach since it's a direct process of reduce loss in the along the gradient of the loss function you give.

3.2.3 SVD approach

given a point set p_i, the second frame after transformation gives us another set of point:

$$p_i^{'} = Rp_i + T + J \tag{37}$$

where J represent a noise vector which is the error between the real position of the point after transformation compare to the current value. Based on this, we are able to obtain an mean square error function of the error we want to minimize:

$$\underset{R,T}{min} J = \frac{1}{2}\sum_{i=1}^{n} ||p_i^{'} - (Rp_i + T)||_2^2 \tag{38}$$

where the generalised formula is given by [5] for which n is the number of point in the set next we calculate the centre of mass of the two set of point:

$$p = \frac{1}{n}\sum_{i=1}^{n} p_i \tag{39}$$

$$p^{'} = \frac{1}{n}\sum_{i=1}^{n} p_i^{'} \tag{40}$$

intuitively, notice that the decentering coordinate of point before and after transformation only different by the rotation. And if we have obtain the maximum rotation, the translation can be calculate by the different from the two centre of mass after transformation.

$$\underset{R}{min} J = \frac{1}{2}\sum_{i=1}^{n} ||(p_i^{'} - p^{'}) - R(p_i - p)||_2^2 \tag{41}$$

$$T = p^{'} - Rp \tag{42}$$

define that $q_i^{'} = p_i^{'} - p^{'}$ and $q_i = p_i - p$.

$$\begin{aligned}
\underset{R}{min} J &= \frac{1}{2}\sum_{i=1}^{n} ||q_i^{'} - Rq_i||_2^2 \\
&= \frac{1}{2}\sum_{i=1}^{n} (q_i^{'} - Rq_i)\cdot(q_i^{'} - Rq_i) \\
&= \frac{1}{2}\sum_{i=1}^{n} q_i^{'}\cdot q_i^{'} - 2q_i^{'}\cdot(Rq_i) + (Rq_i)\cdot(Rq_i) \\
&= \frac{1}{2}\sum_{i=1}^{n} (q_i^{'T} q_i^{'} - 2(q_i^{'})^T(Rq_i) + (Rq_i)^T(Rq_i)) \\
&= \frac{1}{2}\sum_{i=1}^{n} (q_i^{'T} q_i^{'} - 2(q_i^{'})^T(Rq_i) + q_i^T R^T Rq_i) \\
&= \frac{1}{2}\sum_{i=1}^{n} (q_i^{'T} q_i^{'} - 2(q_i^{'})^T(Rq_i) + q_i^T q_i)
\end{aligned} \tag{43}$$

thus minimising J with R equivalent to :

$$\begin{aligned} \min_R J &= \frac{1}{2}\sum_{i=1}^{n}(-q_i^{'T}(Rq_i)) \\ &= \frac{1}{2}\sum_{i=1}^{n} -tr(Rq_i^{'}q_i^T) \\ &= -tr(R(\frac{1}{2}\sum_{i=1}^{n} Rq_i^{'}q_i^T)) \\ &= -tr(RH) \end{aligned} \tag{44}$$

having this we denoted the column of the any matrix by the minuscule of it for example i'th column of H will be h_i. Now we can apply the singular value decomposition for H and obtain an expression of

$$\begin{aligned} &\min_R J - tr(RH) \\ &\quad = -tr(RU\Sigma V^T) \\ &\quad = -tr(XV\Sigma V^T) \\ &\quad = -tr((\Sigma V^T)XV) \\ &\quad = -\sigma_{ii}\sum_i v_i^T(Xv_i) \end{aligned} \tag{45}$$

using Cauchy-Buniakowsky-Schwarz inequality:

$$\begin{aligned} \min_R J &= -\sigma_{ii}\sum_i v_i^T(Xv_i) \\ &\geq -\sigma_{ii}\sum_i \sqrt{(v_i^T v_i)(v^T X^T X v_i)} \\ &= -\sigma_{ii}\sum_i v_i^T v_i \\ &= -tr(V\Sigma V^T) \end{aligned} \tag{46}$$

thus when $X = I$ the error will be minimize. Thus $RU = V$ therefore:

$$R = VU^T \tag{47}$$

and to make sure it's a rotational matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{48}$$

will be used to make sure $det(R) = 1$ by definition

step-wise approach of linear optimization is specially useful in the point cloud dataset, it's suffer less with heavy loaded computation compare to the non-linear approach since it already assume every match fit the point to it's best position in one step.

Algorithm 3 best-fit transform

Require: Point cloud $A, B \in \mathbb{R}^{n \times 3}$
Ensure: : $n \in \mathbb{Z}^+$
$c_A \leftarrow A - \frac{\sum_{i=1}^{n} A}{n}$
$c_B \leftarrow B - \frac{\sum_{i=1}^{n} B}{n}$ ▷ neighborhood of center of mass
$H \leftarrow c_A^T \cdot c_B$ ▷ H is the covariance matrix
$U, S, V^T = svd(H)$
$M \leftarrow I$
$M[-1, -1] \leftarrow det(V \cdot (U^T)$
$R \leftarrow V \cdot M \cdot U^T$
$t \leftarrow c_B - R \cdot c_A$
$T \leftarrow I$
$T[: n, : n] \leftarrow R$
$T[: n, n] \leftarrow t$

Algorithm 4 optimization

Require: Point cloud $A, B \in \mathbb{R}^{n \times 3}$
Ensure: : $n \in \mathbb{Z}^+$
initialise $loss$
$A_i \leftarrow copy(A)$
while loss not converge **do**
$distance, indices \leftarrow KNN(A, B)$
$T \leftarrow best - fit(A[indices], B)$
$A \leftarrow T \cdot A$
$loss \leftarrow \frac{distance}{n}$
end while
$T \leftarrow best - fit(A_i, A)$

3.3 Result

(please refer to the result attach at the appendix section in this report) We select a continuous frames and combined them in to a single set. We will obtain the messy looking shape, where all point scatter around the approximate center of mass and form a disk shape without the correct translation and rotation.

After applying ICP between each frame and take the first frame as the world coordinate, we been able to obtain a clearer constructed map of the real surroundings of the robots.

However, by increasing the current frame number, our methods suffer from calibrating the correct level of the ground. There still left us with the physical constrain to consider in the future works.

4 Conclusion

This research have delivered a comprehensive derivation in to the vast detail and the idea behind Simultaneous Localization and Mapping, discussed and decompose the mathematical prior consisting of Lie algebra as well as singular value decomposition, and experienced on python environment for prototype. This report provides the studies of robotics with the perspective of math axioms, encourage researcher to follow a standard philosophical constrain which helps the construction of more generalised model in the field not only robotics but also a wider research area in computer vision, machine learning and deep learning for the ultimate goal of building generalized artificial intelligence. Nonetheless, the research in general are stay in a ground level of basic idea, without a detailed and well-designed product that can be used by industrial application. The experiment section expose many difficulties within before it's been constructed successfully and tested empirically. For instance, in real world scenario, simple point cloud data consist many implicit physical constrain for example the sensor should be stay on the ground, and it highly depend on the precondition of stationary and rigid body environment without texture. A robust SLAM should be able to adopted itself into complicated light scene, texture as well as detection of moving object for practical usage. Therefore, the future of robotics perception can combined and studied on wider modality.

References

[1] R Gilmore. "Baker-Campbell-Hausdorff formulas". In: *Journal of Mathematical Physics* 15.12 (1974), pp. 2090–2092.

[2] Leyao Huang. "Review on LiDAR-based SLAM techniques". In: *2021 International Conference on Signal Processing and Machine Learning (CONF-SPML)*. IEEE. 2021, pp. 163–168.

[3] Iman Abaspur Kazerouni et al. "A survey of state-of-the-art on visual SLAM". In: *Expert Systems with Applications* 205 (2022), p. 117734.

[4] Johan Ernest Mebius. “Derivation of the Euler-Rodrigues formula for three-dimensional rotations from the general formula for four-dimensional rotations”. In: *arXiv preprint math/0701759* (2007).

[5] Shinji Umeyama. “Least-squares estimation of transformation parameters between two point patterns”. In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 13.04 (1991), pp. 376–380.

5 Appendix

Code

ODOMETRY.py

```
1
2  import os
3  import numpy as np
4  from ICP import myicp
5  from simpleicp import PointCloud, SimpleICP
6
7  def loadPlyToNumpy(file_name):
8      with open(file_name, 'r') as f:
9          lines = f.readlines()
10
11     data_start = lines.index('end_header\n') + 1
12
13     data = [list(map(float, line.split())) for line in lines[
       data_start:]]
14     data_array = np.array(data)
15
16     return data_array
17
18 def savePlyFromPtsRGB(pts, file_name, RGB=None, alpha=None):
19     with open(file_name, 'w') as f:
20         f.write('ply\n')
21         f.write('format␣ascii␣1.0\n')
22         f.write('element␣vertex␣%d\n' % pts.shape[0])
23         f.write('property␣float␣x\n')
24         f.write('property␣float␣y\n')
25         f.write('property␣float␣z\n')
26         if RGB is not None:
27             f.write('property␣uchar␣red\n')
28             f.write('property␣uchar␣green\n')
29             f.write('property␣uchar␣blue\n')
30         if alpha is not None:
31             f.write('property␣uchar␣alpha\n')
32         f.write('end_header\n')
33         # write data
34         for i in range(pts.shape[0]):
35             f.write('%f␣%f␣%f' % (pts[i,0], pts[i,1], pts[i,2]))
36             if RGB is not None:
37                 f.write('␣%d␣%d␣%d' % (RGB[i,0]*255, RGB[i,1]*255,
       RGB[i,2]*255))
38             if alpha is not None:
39                 f.write('␣%d' % (alpha[i]*255))
40             f.write('\n')
```

```
41
42
43 current__path = os.path.dirname(os.path.abspath(__file__))
44 print('Current␣folder␣path␣is:␣{}'.format(current__path))
45 point_cloud_folder_path = "/home/pc/Blender_project/SLAM/
       original_point_cloud/original"
46 saved_point_cloud_folder_path = "/home/pc/Blender_project/SLAM/
       after_ICP"
47
48
49
50 ply_file_list = [f for f in os.listdir(point_cloud_folder_path) if
       f.endswith('.ply')]#sorted([f for f in os.listdir(
       point_cloud_folder_path) if f.endswith('.ply')])
51 ply_file_list = list(filter(lambda f:f[-8:-4] != "copy",
       ply_file_list))
52 ply_index = list(map(lambda f:float(f[:-4]),ply_file_list))
53 comb_file = list(zip(ply_index,ply_file_list))
54 ply_file_list = [item[1] for item in sorted(comb_file, key=lambda x
       :x[0])]
55 print(f"{len(ply_file_list)}␣ply␣files␣found␣in␣{
       point_cloud_folder_path}")
56 print(ply_file_list)
57
58 frame_ini = 0
59 nm_iter = 5# len(ply_file_list)
60 ref_cloud = None
61 data = None
62 ori_data = None
63 ini = True
64 noise_sigma = 0.01
65 traj = []
66 mi = True
67
68
69 for i in range(frame_ini,frame_ini+nm_iter): # fram to choice
70     # iterate through all the input clouds
71     current_file_name = os.path.join(point_cloud_folder_path,
       ply_file_list[i])
72     print(f"Loading␣{current_file_name}")
73     input_cloud = loadPlyToNumpy(current_file_name)
74     print(f"Loaded␣{input_cloud.shape[0]}␣points")
75     # Now use ICP to do SLAM!
76     print(i)
77     if ini: # set up initial frame
78         ini = False
79         ref_cloud = input_cloud
80         print(ref_cloud.shape[0])
81         data = ref_cloud
82         ori_data = ref_cloud
83
84     else:
85         temp = np.copy(input_cloud)
86         print(input_cloud.shape[0],ref_cloud.shape[0])
87         num_choice = min(input_cloud.shape[0],ref_cloud.shape[0])
88         print(num_choice)
89
```

```
90
91          if mi:
92              T,_,_ = myicp(input_cloud[:num_choice,:], ref_cloud[:
    num_choice,:])
93          ### simple icp
94          else:
95              pc_fix = PointCloud(ref_cloud[:num_choice,:],columns=["
    x","y","z"])
96              pc_mov = PointCloud(input_cloud[:num_choice,:],columns
    =["x","y","z"])
97
98              icp = SimpleICP()
99              icp.add_point_clouds(pc_fix, pc_mov)
100             T,_,_,_ = icp.run(max_overlap_distance=1)
101         ###
102
103         traj.append(T)
104
105         after = np.ones((num_choice,4))
106         after[:,0:3] = input_cloud[:num_choice,:]
107         after = T.dot(after.T).T[:,0:3]
108
109         ori_data = np.concatenate((ori_data,temp),0)
110         data = np.concatenate((data,after),0)
111         # After you finish the ICP part, you can save the
    transformed point clouds to ply files
112         #if(input_cloud.shape[0]>2000):
113         ref_cloud = after
114
115         print("shape:␣", data.shape)
116
117 #savePlyFromPtsRGB(ori_data, os.path.join(current__path,
    saved_point_cloud_folder_path, ply_file_list[i]))
118 savePlyFromPtsRGB(data, os.path.join(current__path,
    saved_point_cloud_folder_path, ply_file_list[i]))
119 print(np.array(traj).shape)
```

ICP.py

```
1     import numpy as np
2 from sklearn.neighbors import NearestNeighbors
3 #from icp_best_transform import best_fit_transform
4 import time
5
6 def rotation_matrix(axis, theta):
7     axis = axis/np.sqrt(np.dot(axis, axis))
8     a = np.cos(theta/2.)
9     b, c, d = -axis*np.sin(theta/2.)
10
11    return np.array([[a*a+b*b-c*c-d*d, 2*(b*c-a*d), 2*(b*d+a*c)],
12                     [2*(b*c+a*d), a*a+c*c-b*b-d*d, 2*(c*d-a*b)],
13                     [2*(b*d-a*c), 2*(c*d+a*b), a*a+d*d-b*b-c*c]])
14
15 def nearest_neighbor(src, dst):
16     '''
17     Find the nearest (Euclidean) neighbor in dst for each point in
    src
18     Input:
```

```
        src: Nxm array of points
        dst: Nxm array of points
    Output:
        distances: Euclidean distances of the nearest neighbor
        indices: dst indices of the nearest neighbor
    '''

    assert src.shape == dst.shape

    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(dst)
    distances, indices = neigh.kneighbors(src, return_distance=True
    )
    return distances.ravel(), indices.ravel()

def best_fit_transform(A,B):
    m = A.shape[1]
    centroid_A = np.mean(A, axis=0)
    centroid_B = np.mean(B, axis=0)
    AA = A - centroid_A
    BB = B - centroid_B
    sig_a = np.linalg.norm(AA)**2 / m
    sig_b = np.linalg.norm(BB)**2 / m

    H = np.dot(AA.T,BB)
    U, S, Vt = np.linalg.svd(H)
    S = np.diag(S)

    M = np.identity(m)
    M[m-1,m-1] = np.linalg.det(Vt.T.dot(U.T))

    R = Vt.T.dot(M.dot(U.T))
    s = np.trace(M.dot(S))/sig_a
    s=1


    t = centroid_B - np.multiply(s,R.dot(centroid_A))
    # compute the transformation between the current source and
    nearest destination points
    T=np.identity(m+1)
    T[:m,:m] = R
    T[:m,m] = t

    return T

def myicp(A, B, init_pose=None, max_iterations=50, tolerance=0.001)
    :
    '''
    The Iterative Closest Point method: finds best-fit transform
    that maps points A on to points B
    Input:
        A: Nxm numpy array of source mD points
        B: Nxm numpy array of destination mD point
        init_pose: (m+1)x(m+1) homogeneous transformation
        max_iterations: exit algorithm after max_iterations
        tolerance: convergence criteria
    Output:
```

```
        T: final homogeneous transformation that maps A on to B
        distances: Euclidean distances (errors) of the nearest neighbor
        nm_iter: number of iterations to converge
    '''

    assert A.shape == B.shape

    # get number of dimensions
    m = A.shape[1]

    # make points homogeneous, copy them to maintain the originals
    src = np.ones((m+1,A.shape[0]))
    dst = np.ones((m+1,B.shape[0]))
    src[:m,:] = np.copy(A.T)
    dst[:m,:] = np.copy(B.T)

    # apply the initial pose estimation
    if init_pose is not None:
        src = np.dot(init_pose, src)

    prev_error = 0

    nm_iter = 0

    for i in range(max_iterations):
        # find the nearest neighbors between the current source and destination points
        distances, indices = nearest_neighbor(src[:m,:].T,dst[:m,:].T)
        #print(indices)
        #print(np.mean(distances))

        T = best_fit_transform(src[:m,:].T, dst[:m,indices].T)
        #print(T)

        # update the current source
        src = T.dot(src)

        # check error
        mean_error = np.mean (distances)
        if np.abs(prev_error - mean_error) < tolerance:
            break
        prev_error = mean_error

        nm_iter += 1

    T = best_fit_transform(A, src[:m,:].T) # best fit piring match, calculate the final rtansformation
    #print(indices)
    #print(mean_error)

    return T,distances,nm_iter
```

Figure

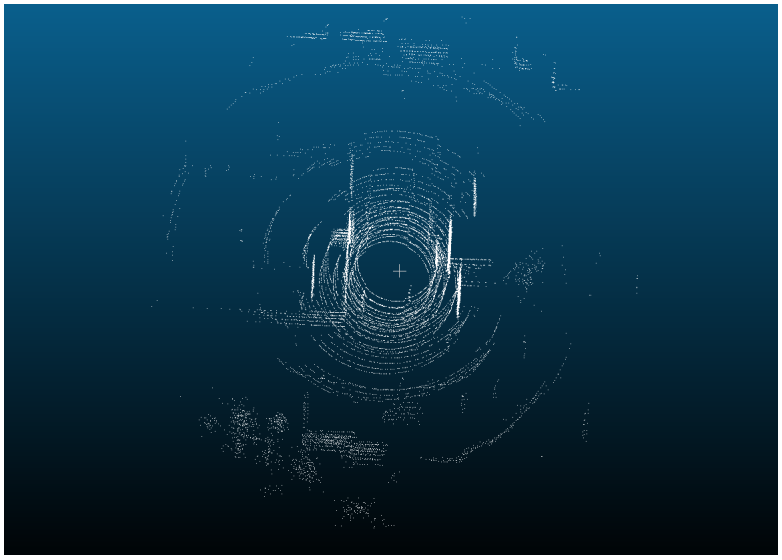

Figure 2: 10 frame calibration(top down view)

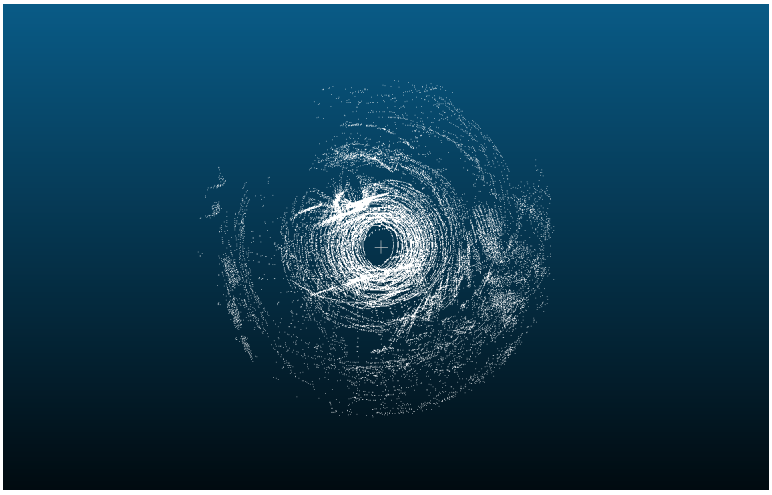

Figure 3: 30 frame calibration(top down view)

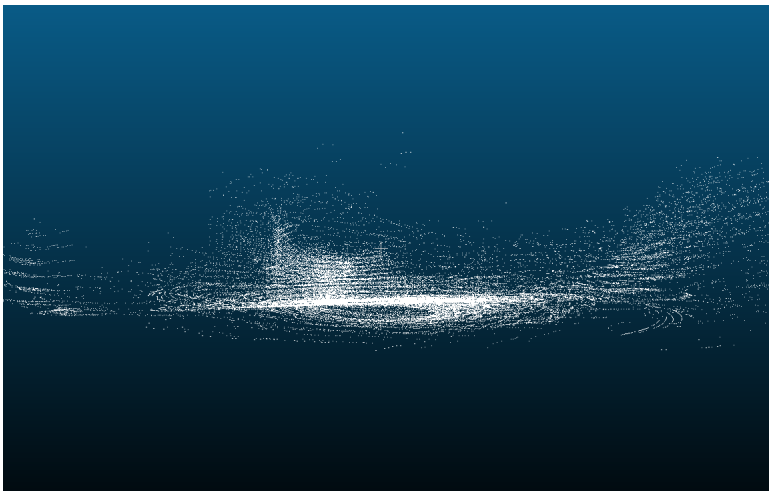

Figure 4: 30 frame calibration(side view)