`tams_sei_registration`, Version 1.0/2.0/3.0, are collections of C/C++ routines which align 3D scans based on Spherical Entropy Image (SEI) and Spherical Harmonics. The difference between Version 1.0/2.0 and 3.0 is:

✓ Version 1.0 is the simplest version, computes SEI based on the original point cloud.

✓ Version 2.0 computes SEI based on the translation normalized point cloud.

✓ Version 3.0 is the only one version without theoretical defect, but is the most complicate one. Version 3.0 computes SEI based on the magnitude of FFT of the original point cloud.

The translation estimation in all Versions is based on 3D POMF(Phase-Only Matched Filter). POMF is based on Fourier transform, so samples of larger size are preferred by POMF. Experiments show that the 3D POMF is much more stable than 1D POMF, but cost more memory. The algorithm implemented in this package present in [1].

1 Theoretical Background

1.1 State-of-the-art

In the context of 3D scan registration, scan alignment methods could be classified into *local* alignment methods and *global* alignment methods. If an good initial estimate of the transformation between two input scans is available, the registration problem could be solved using local methods through an iterative process. While the initial guesses are unavailable, normally the feature-based strategies are adopted. The feature-based strategies make use of explicit feature correspondences in the environment and could deal with scan pairs with partial overlap and large offsets. The common procedure of feature-based registration methods includes: key-point extraction, feature description, feature matching, transformation estimation and refinement.

Various features have been adopted for 3D scan registration. But the feature-based registration methods confront several challenges:

✓ how to eliminate the mismatches

✓ parameters must be selected carefully through numerous trials

✓ run-times vary dramatically since use iterative procedure to estimate the result

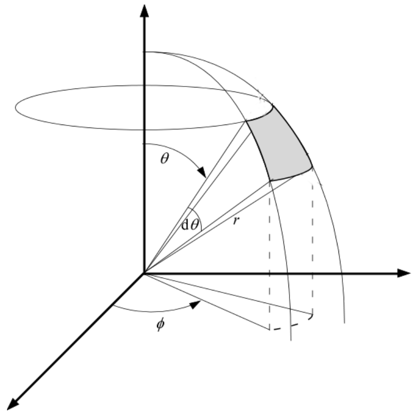

Figure 1: 3D patch when compute SEI

✓ error-prone since only use partial information of scans, and rely on the reliability of key-point extraction and feature description algorithm.

1.2 Spherical Entropy Image (SEI)

The original scan is divided into bins by equally spaced boundaries in the azimuth and elevation dimensions. As shown in Fig. 1, the 3D patch is an analogous square pyramid taking the origin of the scan as the vertex, but the bottom surface is not a plane but the spherical grid. The depth of points belonging to the same 3D patch could be interpreted as the observations of a random variable P, then the information entropy $E(p)$ of the variable P is regarded as the value of structural representation of the patch:

$$E(P) = -\sum_{d \in D} p(P = d) \cdot log\{p(P = d)\} \tag{1}$$

where D is the set of possible values of P, and p is its Probability Density function (PDF).

The process of calculating entropy is:

1. divide the 3D point cloud into several patches according to the polar angle and azimuth angle of points;
2. for each patch, considering the depth of points as the observations of a random variable P, normalize the depth of points, build the histogram and compute the PDF of variable P;
3. compute the entropy of variable P based on the probability distribution.

The structural representation of the 3D point cloud is achieved by computing the entropy of patches in a dense manner. We name this type of representation as

Spherical Entropy Image (SEI).

1.3 Registration of graphs on S^2

Let $SO(3)$ denote the rotation group in 3D space, represented by 3×3 matrices with determinant one. Given a function f_1 on the sphere, and its rotated version f_2 for a rotation $g \in SO(3)$: $f_2 = \wedge(g) \cdot f_1$. Registration of the two functions could be achieved by correlating functions:

$$C(g) = \int_{S^2} f_2(\omega) \bullet \overline{\wedge(g) \cdot f_1(\omega)} d\omega \tag{2}$$

and the g maximizing the integral (2) is the rotation between two functions. However, evaluating $C(g)$ for all possible rotations is a terrific time-consuming task.

It is well known that it is possible to detect the translated duplicates of a pattern in an image by convolving the image with the pattern. And this convolution could be converted to point-wise product via Fourier Transform. In other words, the convolution in time domain equals point-wise multiplication in frequency domain, then the original problem could be solved in frequency domain much more efficiently. Furthermore, this convolution theorem could be generalized to the functions defined on S^2. *Efficient spherical convolution, aided by a fast Spherical Fourier transform and its inverse, contributes to the registration of graphs on* S^2.

Substituting the Spherical Fourier expansions of f_1 and f_2 into equation (2), and then utilizing the *Separation of Variables* technique, the orthogonality and the rotation invariant property between spherical harmonics, the correlation function could be rewritten as:

$$C(g) = \sum_{l} \sum_{|m| \le l} \sum_{|m'| \le l} a_l^m \overline{b_l^{m'}} \cdot \overline{D^l_{m'm}(g)} \tag{3}$$

where a_l^m and $b_l^{m'}$ are respectively the spherical harmonic coefficients of f_1 and f_2; $D^l_{m'm}(g)$ are called *Wigner-D function*, and they are the irreducible unitary representations of $SO(3)$. In some sense, the $D^l_{m'm}(g)$ could be interpreted as the m'-th component of $\wedge(g)$ acting on Y_l^m. Essentially, it is mainly on the strength of the *Separation of Variables* technique and orthogonality of spherical harmonics, and makes full use of the two criteria over and over again.

1.4 Algorithm implemented in this package

The steps of our registration algorithm based on SEI implemented in this package are:

✗ Version1.0

1. compute SEIs based on the input point clouds (Section 1.2)
2. calculate rotation matrix based on Spherical Harmonics of SEIs (Section 1.3)
3. rerotate the point clouds according to the resultant rotation matrix
4. recover the translation based on POMF

✗ Version2.0

1. translation normalization of the original input clouds
2. compute SEIs based on the normalized point clouds (Section 1.2)
3. calculate rotation matrix based on Spherical Harmonics of SEIs (Section 1.3)
4. rerotate the original point clouds according to the resultant rotation matrix
5. recover the translation based on POMF

✗ Version3.0

1. render the input point clouds into volumes
2. compute the magnitude of FFT of volumes
3. compute SEIs based on the magnitude of FFT (Section 1.2)
4. calculate rotation matrix based on Spherical Harmonics of SEIs (Section 1.3)
5. rerotate the point clouds according to the resultant rotation matrix
6. recover the translation based on POMF

use the index of voxel to determine which 3D patch the voxel belongs to

2 Package Structure

✓ CMakeLists.txt
used by **CMake**, the cross-platform, open-source build system. Make this package be easily used under other OS.

✓ FindFFTW3.cmake
.cmake file used by **CMake** to tell compiler the location of **FFTW3** library. Copy it to the **Modules** folder of CMake, then it is unnecessary to set the **FFTW3** library location by hand.

✓ lib1
subdirectory contains the library used for spherical graphs alignment based on Spherical Harmonics.

✓ `tams_soft_fftw_correlation.h`
declaration of the spherical graphs alignment function:
`tams_soft_fftw_correlate`

```
extern void tams_soft_fftw_correlate ( const Eigen::VectorXf &TAMS_sei_sig_real,
                                       const Eigen::VectorXf &TAMS_sei_pat_real,
                                       const int tams_bwIn,
                                       const int tams_bwOut,
                                       const int tams_degLim,
                                       double &alpha,
                                       double &beta,
                                       double &gama);
```

✓ `tams_soft_fftw_correlation.cpp`
implementation of the spherical graphs alignment function:
`tams_soft_fftw_correlate`

✓ `tams_sei_registration.h`
declaration of functions used for scan registration:

```
void tams_cart2sph(const float x, const float y, const float z,
                   float& azimuth, float& polar);
```

```
/** tams_cart2sph(X,Y,Z, azimuth, polar) transforms Cartesian coordinates stored in
  * corresponding elements of arrays X, Y, and Z into spherical coordinates.
  * azimuth and polar are angular displacements in radians.
  * azimuth(longitudinal) is the counterclockwise angle in the x-y plane
  * measured from the positive x-axis.
  * polar(colatitudianl) is the polar angle measured from the z axis.
  *
  * 0 < azimuth < 2*M_PI; 0 < polar < M_PI
  */
```

```
void tams_vector_normalization (std::vector<float> &tams_vector);
```

```
/** tams_vector_normalization} normalize the input vector
  * Parameters:
  * [in]      tams_vector   the input vector
  * [out]     tams_vector   the normalized vector (values are in range [0,1])
  */
```

```
void tams_vector2entropy( const std::vector<float> tams_vector,
                          const size_t hist_bin,
                          float& entropy);
```

```
/** tams_vector2entropy compute the entropy of a vector
  * Parameters:
  * [in]   tams_vector     the input vector
  * [in]   hist_bin        the size of histogram in entropy computation
  * [out]  entropy         the resultant entropy
  */
```

```
void computeSEI( const PointCloudT cloud,
                 size_t sei_dim,
                 size_t hist_bin,
                 Eigen::MatrixXf &entropy);
```

```
computeSEI contained in Version 1.0/2.0
/** computeSEI calculate the SEI of the input point cloud
  * Parameters:
  * [in]     cloud         the input vector
  * [in]     sei_dim       the dimension of SEI(sei_dimXsei_dim)
  * [in]     hist_bin      the size of histogram in entropy computation
  * [out]    entropy       the resultant SEI stored in a (sei_dim X sei_dim) matrix
  */
```

```
void computeSEI( const float *volume,
                 Eigen::Vector3i volumesize,
                 size_t sei_dim,
                 size_t hist_bin,
                 Eigen::MatrixXf &entropy);
```

```
computeSEI contained in Version 3.0
/** computeSEI calculate the SEI of the input volume (magnitude of FFT)
  * Parameters:
  * [in]     volume        the input volume
  * [in]     volumesize    the size of input volume
  * [in]     sei_dim       the dimension of SEI(sei_dim X sei_dim)
  * [in]     hist_bin      the size of histogram in entropy computation
  * [out]    entropy       the resultant SEI stored in a (sei_dim X sei_dim) matrix
  */
```

```
void voxelsize2volumesize ( const PointCloudT cloud,
                            Eigen::Vector3f voxelsize,
                            Eigen::Vector3i &volumesize);
```

```
/** voxelsize2volumesize compute the size of volume
  * the points cloud rendered based on the size of voxel.
  * Parameters:
  * [in]  cloud         the input point cloud
  * [in]  voxelsize     the size of the voxel
  * [out] volumesize    the size of volume the input cloud should be rendered to
  */
```

```
void point2volume (const PointCloudT cloud,
                   Eigen::Vector3f voxelsize,
                   Eigen::Vector3i volumesize,
                   Eigen::Vector3i volumesize_origin,
                   double *volume);
```

```
/** point2volume render the input point cloud
  * to a volume, assign the number of points (or max curvature of points)
  * in a voxel to the value of voxel.
  * Parameters:
  * [in]  cloud              the input point cloud
  * [in]  voxelsize          the size of the voxel
  * [in]  volumesize         the real size of resutant volume
  * [in]  volumesize_origin  the volumesize calculated by function "voxelsize2volumesize"
  * [out] volume             the resultant volume contained in the
  *                          volumesize(0)*volumesize(1)*volumesize(2) matrix
  * [in]  Usecurvature       whether to assign the max curvature of points
                             in a voxel to the value of voxel.
  */
```

```
void PhaseCorrelation3D(const double *signal,
                        const double *pattern,
                        const int height,
                        const int width,
                        const int depth,
                        int &height_offset,
                        int &width_offset,
                        int &depth_offset);
```

```
/** PhaseCorrelation3D compute the offset between two input volumes
  * based on POMF (Phase Only Matched Filter)
  * -->> Q(k) = conjugate(S(k))/|S(k)| * R(k)/|R(k)|
  * -->> q(x) = ifft(Q(k))
  * -->> (xs,ys) = argmax(q(x))
  * Note that the storage order of FFTW is row-order
  * We adopt the RIGHT-hand Cartesian coordinate system.
  * Parameters:
  * [in] signal              the input(signal) volume
  * [in] pattern             the input(pattern) volume
  * [in] height              the height of input volumes(range of x)
  * [in] width               the width of input volumes (range of y)
  * [in] depth               the depth of input volumes (range of z)
  * [out] height_offset      the result offset, we move down (positive x axis)
                             pattern height_offset to match signal
  * [out] width_offset       the result offset, we move right (positive y axis)
                             pattern width_offset to match signal
  * [out] depth_offset       the result offset, we move close to viewer (positive
                             z axis) pattern depth_offset to match signal
  */
```

✓ `tams_sei_registration.hpp`
implemention of the functions declared in `tams_sei_registration.h`

✓ `tams_sei_registration.cpp`
the main function about scan registration happens

3 Package Dependency

The package was developed and tested in the GNU/Linux environment. But we give the *CMakeLists.txt* which used by CMake, the cross-platform, open-source build system. We believe the code could be used under other OS. Some modifications might be required(e.g., tell the complier the locations of dependencies), but I do not think anything drastic should be necessary.

This package depends on **PCL** and **FFTW3**. **FFTW3** is free available on `http://www.fftw.org`. You could tell the compiler the location of **FFTW3** by copying the *FindFFTW3.cmake* to the **Modules** folder of cmake, if you want. Or you could tell the compiler the location of **FFTW3** by hand. **PCL** is free available on `http://www.pointcloud.org`. Normally for **PCL**, *PCLConfig.cmake* is placed in **Share** folder when install **PCL**.

For example, in our laptop (Ubuntu 12.04/14.04, cmake-2.8), **Modules** folder of cmake is under */usr/share/cmake-2.8/Modules*

4 Application Example

5 Discussion

For data registration, the easily overlooked question is how to apply the registration result. In other words, we should apply the registration result to which input to match another input?

There is a convention (but not necessarily true for all packages): we should apply the result to object/pattern/source to match scene/signal/target.

References

[1] Bo Sun, Weiwei Kong, Junhao Xiao, and Jianwei Zhang. A global feature-less scan registration strategy based on spherical entropy images. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, submitted.