Common Folder

The C++ implementation for preparation for the clustering

Generated by Doxygen 1.8.11

Contents

Chapter 1

Common Folder Description

It includes relevant functions

- File I/O operations
- Sampling for streamlines/pathlines
- k-means initialization (from samples, from random coordinates, or k-means++)
- The hierarchical L-method for finding optimal number of clusters
- Different similarity measures for streamlines/pathlines
- The functions to calculate the clustering evaluation metrics, silhouette, the Gamma statics, DB index and normalized validity measurement

Special notice

Distance Matrix

The distance matrix **distanceMatrix** is pre-stored as a 'float∗∗∗' so that every time when calculating the similarity measure between two selected curves, the 'distanceMatrix' will be checked to be NULL or not. If 'distanceMatrix' is NULL, then the similarity measure function will be called otherwise the cached value is called.

MetricPreparation

It is created before calculating the **MetricPreparation** due to the fact that for some similarity measures, e.g., the d_G (2), d_S(14) and d_P(15), either the segmentation on the streamlines/pathlines or the signature histograms should be calculated. In order to avoid repeated calculation of those signatures, we use a cache to pre-calculate the signatures for each line and store them for further pairwise distance value calculation.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 CompareFunc Class Reference

```
#include <PreComputing.h>
```

Public Member Functions

- bool operator() (const CurvatureObject &first, const CurvatureObject &second)

4.1.1 Detailed Description

Definition at line 62 of file PreComputing.h.

4.1.2 Member Function Documentation

4.1.2.1 bool CompareFunc::operator() (const CurvatureObject & *first,* const CurvatureObject & *second*) [inline]

Definition at line 65 of file PreComputing.h.

```
{
    return first.curvature < second.curvature;
}
```

The documentation for this class was generated from the following file:

- PreComputing.h

4.2 CurvatureObject Struct Reference

```
#include <PreComputing.h>
```

Public Member Functions

- CurvatureObject (const float &curvature, const int &i)
- CurvatureObject ()

Public Attributes

- float curvature
- int index

4.2.1 Detailed Description

Definition at line 46 of file PreComputing.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 CurvatureObject::CurvatureObject (const float & *curvature,* const int & *i*) [inline]

Definition at line 51 of file PreComputing.h.

```
51                                                       : curvature(
       curvature), index(i)
52      {}
```

4.2.2.2 CurvatureObject::CurvatureObject ( ) [inline]

Definition at line 54 of file PreComputing.h.

```
55      {}
```

4.2.3 Member Data Documentation

4.2.3.1 float CurvatureObject::curvature

Definition at line 48 of file PreComputing.h.

4.2.3.2 int CurvatureObject::index

Definition at line 49 of file PreComputing.h.

The documentation for this struct was generated from the following file:

- PreComputing.h

4.3 cyl_bessel_j_integral_rep< value_type > Class Template Reference

```
#include <Metric.h>
```

Public Member Functions

- cyl_bessel_j_integral_rep (const value_type &a, const value_type &b, const value_type &c)
- value_type operator() (const value_type &t) const

Private Attributes

- const value_type a
- const value_type b
- const value_type c

4.3.1 Detailed Description

template<typename value_type>
class cyl_bessel_j_integral_rep< value_type >

Definition at line 348 of file Metric.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 template<typename value_type > cyl_bessel_j_integral_rep< value_type >::cyl_bessel_j_integral_rep (const value_type & *a*, const value_type & *b*, const value_type & *c*) [inline]

Definition at line 351 of file Metric.h.

```
351                                                                                              :
      a(a), b(b), c(c)
352     {}
```

4.3.3 Member Function Documentation

4.3.3.1 template<typename value_type > value_type cyl_bessel_j_integral_rep< value_type >::operator() (const value_type & *t*) const [inline]

Definition at line 354 of file Metric.h.

```
355     {
356         // pi * Jn(x) = Int_0^pi [cos(x * sin(t) - n*t) dt]
357         // return cos(x * sin(t) - (n * t));
358         return sqrt(a+2.0*b*t+c*t*t);
359     }
```

4.3.4 Member Data Documentation

4.3.4.1 template<typename value_type > const value_type cyl_bessel_j_integral_rep< value_type >::a [private]

Definition at line 362 of file Metric.h.

4.3.4.2 template<typename value_type > const value_type cyl_bessel_j_integral_rep< value_type >::b [private]

Definition at line 362 of file Metric.h.

4.3.4.3 template<typename value_type > const value_type cyl_bessel_j_integral_rep< value_type >::c [private]

Definition at line 362 of file Metric.h.

The documentation for this class was generated from the following file:

- Metric.h

4.4 DetermClusterNum Class Reference

```
#include <DetermClusterNum.h>
```

Public Member Functions

- DetermClusterNum ()
- virtual ∼DetermClusterNum ()
- const int & getFinalNumOfClusters ()
- void iterativeRefinement (std::map< int, float > &eval_graph)
- void recordLMethodResult (const int &normOption)

Private Member Functions

- const int LMethod (const std::map< int, float > &eval_graph, const int &cutoff)
- void removeExtreme (std::map< int, float > &eval_graph)

Private Attributes

- int finalNumOfClusters

4.4.1 Detailed Description

Definition at line 30 of file DetermClusterNum.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 DetermClusterNum::DetermClusterNum ( )

Definition at line 14 of file DetermClusterNum.cpp.

```
14                                     {
15     // TODO Auto-generated constructor stub
16
17 }
```

4.4.2.2 DetermClusterNum::∼DetermClusterNum ( ) [virtual]

Definition at line 23 of file DetermClusterNum.cpp.

```
23                                      {
24     // TODO Auto-generated destructor stub
25 }
```

4.4.3 Member Function Documentation

4.4.3.1 const int& DetermClusterNum::getFinalNumOfClusters ( ) [inline]

Definition at line 48 of file DetermClusterNum.h.

```
49     {
50         return finalNumOfClusters;
51     }
```

4.4.3.2 void DetermClusterNum::iterativeRefinement (std::map< int, float > & *eval_graph*)

Definition at line 32 of file DetermClusterNum.cpp.

```
33 {
34     // some necessary pre-processing to remove irregular shapes for the L-method
35     removeExtreme(eval_graph);
36
37     // start from the first to search the point with knee
38     int cutoff, lastKnee;
39     int currentKnee = eval_graph.rbegin()->first;
40     cutoff = currentKnee;
41     do  // an iterative refinement for the L-method
42     {
43         lastKnee = currentKnee;
44         currentKnee = LMethod(eval_graph, cutoff);
45         std::cout << "returned value is " << currentKnee <<", cutoff is " << cutoff << std::endl;
46         cutoff = currentKnee*2;
47     }while(currentKnee < lastKnee);
48
49     // get the optimal number of clusters
50     finalNumOfClusters = currentKnee;
51
52     std::cout << finalNumOfClusters << std::endl;
53 }
```

4.4.3.3 const int DetermClusterNum::LMethod (const std::map< int, float > & *eval_graph*, const int & *cutoff*) [private]

Definition at line 62 of file DetermClusterNum.cpp.

```
{
    struct CompObj { float val; int index; };
// #pragma omp declare reduction(minimum : struct CompObj : omp_out = omp_in.val < omp_out.val ? omp_in : omp_out)
    struct CompObj RMSE;
    RMSE.val = FLT_MAX;
    RMSE.index = -1;

    const int& firstIndex = eval_graph.begin()->first;
    /* find the minimal c that minimizes RMSE for the selected cutoff */
#pragma omp parallel num_threads(8)
    {
    #pragma omp nowait
        for(int i=firstIndex;i<=cutoff;++i)
        {
            /* left segment linear least square fitting */
            std::vector<float> index_vec;
            std::vector<float> dist_vec;

            // assign the vector for left segment
            std::map<int, float>::const_iterator iter;
            for(int j=firstIndex;j<=i;++j)
            {
                iter = eval_graph.find(j);
                if(iter!=eval_graph.end())
                {
                    index_vec.push_back(iter->first);
                    dist_vec.push_back(iter->second);
                }
            }
            Eigen::MatrixXf A_sub(2, index_vec.size());
            A_sub.row(0) = Eigen::VectorXf::Map(&(index_vec[0]), index_vec.size()).transpose();
            A_sub.row(1) = Eigen::VectorXf::Constant(index_vec.size(), 1.0).transpose();
            Eigen::VectorXf b_sub = Eigen::VectorXf::Map(&(dist_vec[0]), index_vec.size());
            A_sub.transposeInPlace();
            int firstRows = A_sub.rows();

            // solve the least-square fitting problems
            Eigen::VectorXf c = A_sub.colPivHouseholderQr().solve(b_sub);
            Eigen::VectorXf error = b_sub-A_sub*c;
            float rmse_l = error.transpose()*error;

            /* right segment linear least square fitting */
            index_vec.clear();
            dist_vec.clear();

            // assignment of the vector
            for(int j=i+1;j<=cutoff;++j)
            {
                iter = eval_graph.find(j);
                if(iter!=eval_graph.end())
                {
                    index_vec.push_back(iter->first);
                    dist_vec.push_back(iter->second);
                }
            }
            A_sub = Eigen::MatrixXf(2, index_vec.size());
            A_sub.row(0) = Eigen::VectorXf::Map(&(index_vec[0]), index_vec.size()).transpose();
            A_sub.row(1) = Eigen::VectorXf::Constant(index_vec.size(), 1.0).transpose();
            b_sub = Eigen::VectorXf::Map(&(dist_vec[0]), index_vec.size());
            A_sub.transposeInPlace();
            int secondRows = A_sub.rows();

            // least-square fitting problem
            c = A_sub.colPivHouseholderQr().solve(b_sub);
            error = b_sub-A_sub*c;
            float rmse_r = error.transpose()*error;

            /* compute the total weighted error */
            float rmse = float(firstRows)/float(firstRows+secondRows)*rmse_l+
                    float(secondRows)/float(firstRows+secondRows)*rmse_r;
            // update the rmse value and index
        #pragma omp critical
            if(RMSE.val>rmse)
            {
                RMSE.val=rmse;
                RMSE.index=i;
            }
        }
    }

    return RMSE.index;
}
```

4.4.3.4 void DetermClusterNum::recordLMethodResult (const int & *normOption*)

Definition at line 151 of file DetermClusterNum.cpp.

```
{
    std::ofstream readme("../dataset/LMethod",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "Optimal cluster number of norm " << normOption << " is " << finalNumOfClusters << std::endl;
    readme << std::endl;
    readme.close();
}
```

4.4.3.5 void DetermClusterNum::removeExtreme (std::map< int, float > & *eval_graph*) [private]

Definition at line 169 of file DetermClusterNum.cpp.

```
{
    // find the left index with the maximal distance
    float maxDist = -1.0;
    int leftIndex = -1;
    for(auto iter:eval_graph)
    {
        if(maxDist<iter.second)
        {
            maxDist=iter.second;
            leftIndex=iter.first;
        }
    }
    auto iter_index = eval_graph.find(leftIndex);

    // remove some irregular indices
    for(auto iter=eval_graph.begin();iter!=iter_index;)
    {
        if(iter->first<leftIndex&&iter->second<maxDist)
            eval_graph.erase(iter++);
        else
            ++iter;
    }
}
```

4.4.4 Member Data Documentation

4.4.4.1 int DetermClusterNum::finalNumOfClusters [private]

Definition at line 72 of file DetermClusterNum.h.

The documentation for this class was generated from the following files:

- DetermClusterNum.h
- DetermClusterNum.cpp

4.5 ExtractedLine Struct Reference

```
#include <IOHandler.h>
```

Public Member Functions

- ExtractedLine (const int &pointIndex, const int &cluster)

Public Attributes

- int lineNum
- int cluster

4.5.1 Detailed Description

Definition at line 35 of file IOHandler.h.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 ExtractedLine::ExtractedLine (const int & *pointIndex,* const int & *cluster*) [inline]

Definition at line 39 of file IOHandler.h.

```
               :lineNum(pointIndex),cluster(cluster)
     {}
```

4.5.3 Member Data Documentation

4.5.3.1 int ExtractedLine::cluster

Definition at line 38 of file IOHandler.h.

4.5.3.2 int ExtractedLine::lineNum

Definition at line 37 of file IOHandler.h.

The documentation for this struct was generated from the following file:

- IOHandler.h

4.6 FeatureLine Struct Reference

```
#include <IOHandler.h>
```

Public Member Functions

- FeatureLine ()
- FeatureLine (const std::vector< std::vector< float > > &dataVec)

Public Attributes

- std::vector< MeanLine > centerMass
- std::vector< ExtractedLine > closest
- std::vector< ExtractedLine > furthest
- std::vector< int > group
- std::vector< int > totalNum

4.6.1 Detailed Description

Definition at line 82 of file IOHandler.h.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 FeatureLine::FeatureLine ( ) [inline]

Definition at line 90 of file IOHandler.h.

```
    {

    }
```

4.6.2.2 FeatureLine::FeatureLine (const std::vector< std::vector< float > > & *dataVec*) [inline]

Definition at line 95 of file IOHandler.h.

```
    {
        group = std::vector<int>(dataVec.size());
        totalNum = std::vector<int>(dataVec.size());
    }
```

4.6.3 Member Data Documentation

4.6.3.1 std::vector<MeanLine> FeatureLine::centerMass

Definition at line 84 of file IOHandler.h.

4.6.3.2 std::vector<ExtractedLine> FeatureLine::closest

Definition at line 85 of file IOHandler.h.

4.6.3.3 std::vector< ExtractedLine > FeatureLine::furthest

Definition at line 86 of file IOHandler.h.

4.6.3.4 std::vector< int > FeatureLine::group

Definition at line 87 of file IOHandler.h.

4.6.3.5 std::vector< int > FeatureLine::totalNum

Definition at line 88 of file IOHandler.h.

The documentation for this struct was generated from the following file:

- IOHandler.h

4.7 Initialization Class Reference

```
#include <Initialization.h>
```

Static Public Member Functions

- static void generateRandomPos (MatrixXf &clusterCenter, const int &column, const MatrixXf &cArray, const int &Cluster)
- static void generateFromSamples (MatrixXf &clusterCenter, const int &column, const MatrixXf &cArray, const int &Cluster)
- static void generateFarSamples (MatrixXf &clusterCenter, const int &column, const MatrixXf &cArray, const int &Cluster, const int &normOption, const MetricPreparation &object)

4.7.1 Detailed Description

Definition at line 24 of file Initialization.h.

4.7.2 Member Function Documentation

4.7.2.1 void Initialization::generateFarSamples (MatrixXf & *clusterCenter*, const int & *column*, const MatrixXf & *cArray*, const int & *Cluster*, const int & *normOption*, const **MetricPreparation** & *object*) [static]

Definition at line 108 of file Initialization.cpp.

```
{
    assert(column==cArray.cols());
    const int Total = cArray.rows();
    clusterCenter = MatrixXf(Cluster,column);
    int number[Cluster], selection;
    srand(time(0));
    const int& MaxNum = cArray.rows();
    number[0] = rand()%MaxNum;
    int chosen = 1;

    float percentage, nearest, toCentroid;
    VectorXf distance(Total);
    double squredSummation;
    float left, right;
    while(chosen<Cluster)
    {
        percentage = float(rand()/(double)RAND_MAX);
        for (int i = 0; i < Total; ++i)
        {
            nearest = FLT_MAX;
            for (int j = 0; j < chosen; ++j)
            {
                toCentroid = getDisimilarity(cArray, i, number[j], normOption, object);
                if(nearest>toCentroid)
                    nearest=toCentroid;
            }
            distance(i)=nearest*nearest;
        }
        squredSummation = distance.sum();
        left = 0.0, right = 0.0;
        for (int i = 0; i < Total; ++i)
        {
            left = right;
            right += float((double)distance(i)/squredSummation);
            if(left < percentage && percentage <= right)
            {
                selection = i;
                break;
            }
        }
        number[chosen] = selection;
        chosen++;
    }

#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Cluster; ++i)
    {
        clusterCenter.row(i) = cArray.row(number[i]);
    }

}
```

4.7.2.2 void Initialization::generateFromSamples (MatrixXf & *clusterCenter*, const int & *column*, const MatrixXf & *cArray*, const int & *Cluster*) [static]

Definition at line 54 of file Initialization.cpp.

```
{
    clusterCenter = MatrixXf(Cluster,column);
    std::vector<int> number(Cluster);
    srand(time(0));

    const int& MaxNum = cArray.rows();

    std::cout << MaxNum << std::endl;

    number[0] = rand()%MaxNum;
    int randNum, chosen = 1;
    bool found;
    for (int i = 1; i < Cluster; ++i)
    {
        do
        {
            randNum = rand()%MaxNum;
            found = false;
            for(int j=0;j<chosen;j++)
            {
                if(randNum==number[j])
                {
                    found = true;
                    break;
                }
            }
        }while(found!=false);
        number[i] = randNum;
        ++chosen;
    }
    assert(chosen==Cluster);
    assert(column==cArray.cols());

#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Cluster; ++i)
    {
        clusterCenter.row(i) = cArray.row(number[i]);
    }
}
```

4.7.2.3 void Initialization::generateRandomPos (MatrixXf & *clusterCenter*, const int & *column*, const MatrixXf & *cArray*, const int & *Cluster*) [static]

Definition at line 18 of file Initialization.cpp.

```
{
    clusterCenter = MatrixXf::Random(Cluster, column);
    MatrixXf range(2, column);
    range.row(0) = cArray.colwise().maxCoeff();  //first row contains max
    range.row(1) = cArray.colwise().minCoeff();  //second row contains min
    VectorXf diffRange = range.row(0)-range.row(1);

    MatrixXf diagonalRange = MatrixXf::Zero(column,column);

#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < column; ++i)
    {
        diagonalRange(i,i) = diffRange(i);
    }
    clusterCenter = (clusterCenter+MatrixXf::Constant(Cluster,column,1.0))/2.0;

#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Cluster; ++i)
    {
        clusterCenter.row(i) = clusterCenter.row(i)*diagonalRange+range.row(1);
    }
}
```

The documentation for this class was generated from the following files:

- Initialization.h
- Initialization.cpp

4.8 IOHandler Class Reference

```
#include <IOHandler.h>
```

Static Public Member Functions

- static void readFile (const string &fileName, std::vector< std::vector< float > > &dataVec, int &vertexCount, const int &dimension, int &maxElement)
- static void readFile (const string &fileName, std::vector< std::vector< float > > &dataVec, int &vertexCount, const int &dimension, const int &trajectoryNum, const int &Frame)
- static void printVTK (const string &fileName, const std::vector< std::vector< float > > &dataVec, const int &vertexCount, const int &dimension, const std::vector< int > &clusterNumber, const std::vector< float > &sCluster)
- static void printVTK (const string &fileName, const std::vector< std::vector< float > > &dataVec, const int &vertexCount, const int &dimension)
- static void printVTK (const string &fileName, const std::vector< MeanLine > &dataVec, const int &vertex↩ Count, const int &dimension, const std::vector< float > &sCluster)
- static void printToFull (const std::vector< std::vector< float > > &dataVec, const std::vector< int > &group, const std::vector< int > &totalNum, const string &groupName, const string &fullName, const int &dimension)
- static void printToFull (const std::vector< std::vector< float > > &dataVec, const std::vector< float > &sData, const string &groupName, const string &fullName, const int &dimension)
- static void printToFull (const std::vector< std::vector< float > > &origin, const std::vector< int > &group, const string &fullName, const string &groupName, const int &dimension)
- static void printToFull (const std::vector< std::vector< float > > &dataVec, const std::vector< int > &group, const std::vector< float > &sCluster, const string &groupName, const string &fullName, const int &dimension)
- static void writeReadme (const double &PCA_KMeans_delta, const double &KMeans_delta)
- static void writeReadme (const string &comment, const std::vector< float > &sAverage)
- static void writeReadme (const std::vector< string > &timeName, const std::vector< double > &timeDiff, const int &cluster)
- static void writeReadme (const std::vector< string > &timeName, const std::vector< string > &timeDiff, const int &cluster)
- static void writeReadme (const std::vector< ExtractedLine > &closest, const std::vector< ExtractedLine > &furthest, const int &normOption)
- static void writeReadme (const std::vector< ExtractedLine > &closest, const std::vector< ExtractedLine > &furthest)
- static void writeReadme (const float &closestAverage, const float &furthestAverage)
- static void writeReadme (const string &comments)
- static void writeReadme (const float &entropy, const Silhouette &sil, const string &norm_str)
- static void writeReadMe (const float &value, const string &dataSet, const string &clustering, const string &value_name)
- static void writeGroupSize (const std::vector< int > &storage)
- static void expandArray (MatrixXf &data, const std::vector< std::vector< float > > &dataVec, const int &dimension, const int &maxElements)
- static void expandArray (std::vector< std::vector< float > > &equalArray, const std::vector< std::vector< float > > &trajectories, const int &dimension, const int &maxRowNum)
- static void sampleArray (MatrixXf &data, const std::vector< std::vector< float > > &dataVec, const int &dimension, const int &maxElements)
- static void formArray (float ∗∗∗data, const std::vector< std::vector< float > > &dataVec, const int &dimension)
- static void uniformArcSampling (MatrixXf &data, const std::vector< std::vector< float > > &dataVec, const int &dimension, const int &maxElements)
- static void deleteArray (float ∗∗data, const int &row)
- static void assignVec (std::vector< std::vector< float > > &closestStreamline, std::vector< int > &cluster, const std::vector< ExtractedLine > &closest, int &pointNumber, const std::vector< std::vector< float > > &dataVec)
- static void assignVec (std::vector< int > &cluster, const std::vector< MeanLine > &centerMass)
- static void writeGroup (const std::vector< int > &group, const std::vector< std::vector< float > > &dataVec)
- static void printQuery (const int &normOption, const int &order, const StringQuery &queryResult, const std↩ ::vector< std::vector< float > > &dataVec)
- static void printTXT (float ∗∗data, const int &Row, const int &Column)

- static void printFeature (const string &fileName, const std::vector< std::vector< float > > &array, const std::vector< float > &sCluster, const int &dimension)
- static void printFeature (const string &fileName, const std::vector< std::vector< float > > &array, const std::vector< float > &sCluster, const std::vector< float > &rotation, const int &dimension)
- static void printClusters (const std::vector< std::vector< float > > &dataVec, const std::vector< int > &group, const std::vector< int > &storage, const string &groupName, const string &fullName, const int &dimension)
- static void printClustersNoise (const std::vector< std::vector< float > > &dataVec, const std::vector< int > &group, const std::vector< int > &storage, const string &groupName, const string &fullName, const int &dimension)
- static void generateReadme (const std::vector< string > &activityList, const std::vector< double > &timeList, const int &normOption, const int &numClusters, const float &sValue, const float &threshold)
- static void generateReadme (const std::vector< string > &activityList, const std::vector< string > &timeList)
- static void generateGroups (const std::vector< std::vector< int > > &storage)
- static void generateGroups (const std::vector< std::vector< int > > &storage, const string &fileName)
- static void readClusteringNumber (std::unordered_map< int, int > &clusMap, const string &fileName)

4.8.1 Detailed Description

Definition at line 106 of file IOHandler.h.

4.8.2 Member Function Documentation

4.8.2.1 void IOHandler::assignVec (std::vector< std::vector< float > > & *closestStreamline,* std::vector< int > & *cluster,* const std::vector< ExtractedLine > & *closest,* int & *pointNumber,* const std::vector< std::vector< float > > & *dataVec*) [static]

Definition at line 1048 of file IOHandler.cpp.

```
{
    if(closest.empty())
        return;
    closestStreamline = std::vector<std::vector<float> >(closest.size(), std::vector<float>());
    cluster = std::vector<int>(closest.size());
    pointNumber = 0;
    for (int i = 0; i < closestStreamline.size(); ++i)
    {
        closestStreamline[i] = dataVec[closest[i].lineNum];
        pointNumber+=closestStreamline[i].size();
        cluster[i] = closest[i].cluster;
    }
}
```

4.8.2.2 void IOHandler::assignVec (std::vector< int > & *cluster,* const std::vector< MeanLine > & *centerMass*) [static]

Definition at line 1074 of file IOHandler.cpp.

```
{
    cluster = std::vector<int>(centerMass.size());
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < cluster.size(); ++i)
    {
        cluster[i] = centerMass[i].cluster;
    }
}
```

4.8.2.3 void IOHandler::deleteArray (float ∗∗ *data,* const int & *row*) [static]

Definition at line 797 of file IOHandler.cpp.

```
{
    if(data==NULL)
        return;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < row; ++i)
    {
        delete[] data[i];
    }
    delete[] data;
}
```

4.8.2.4 void IOHandler::expandArray (MatrixXf & *data,* const std::vector< std::vector< float > > & *dataVec,* const int & *dimension,* const int & *maxElements*) [static]

Definition at line 439 of file IOHandler.cpp.

```
{
    data = Eigen::MatrixXf(dataVec.size(), maxElements);
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < dataVec.size(); ++i)
    {
        const std::vector<float>& eachVec = dataVec[i];
        const int& vecSize = eachVec.size();
        //data.row(i) = Eigen::VectorXf::Map(&(eachVec[0]), vecSize);
        for (int j = 0; j<vecSize; j++)
            data(i,j) = eachVec[j];

        for (int j = vecSize; j < maxElements; j=j+dimension)
        {
            for (int k=0; k<dimension; k++)
                data(i,j+k) = eachVec[vecSize-dimension+k];
        }
    }
}
```

4.8.2.5 void IOHandler::expandArray (std::vector< std::vector< float > > & *equalArray,* const std::vector< std::vector< float > > & *trajectories,* const int & *dimension,* const int & *maxRowNum*) [static]

Definition at line 1254 of file IOHandler.cpp.

```
{
    equalArray = std::vector<std::vector<float> >(trajectories.size(),
                 std::vector<float>(maxElement));
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < trajectories.size(); ++i)
    {
        std::vector<float>& tempRow = equalArray[i];
        const std::vector<float>& tempTraj = trajectories[i];
        const int& vecSize = tempTraj.size();
        memcpy(&(tempRow[0]), &(tempTraj[0]), vecSize*sizeof(float));
        for (int j = vecSize; j < maxElement; j=j+dimension)
        {
            memcpy(&(tempRow[j]), &(tempTraj[vecSize-dimension]),
                   dimension*sizeof(float));
        }
    }
}
```

4.8.2.6 void IOHandler::formArray (float *** *data*, const std::vector< std::vector< float > > & *dataVec*, const int & *dimension*) [static]

Definition at line 630 of file IOHandler.cpp.

```
{
    *data = new float*[dataVec.size()];
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < dataVec.size(); ++i)
    {
        const int& arraySize = dataVec[i].size();
        (*data)[i] = new float[arraySize];
        memcpy(&(*data)[i][0], &(dataVec[i][0]), arraySize*sizeof(float));
    }
}
```

4.8.2.7 void IOHandler::generateGroups (const std::vector< std::vector< int > > & *storage*) [static]

Definition at line 1740 of file IOHandler.cpp.

```
{
    if(storage.empty())
        return;
    std::ofstream readme("../dataset/Storage",ios::out|ios::app);
    if(!readme)
    {
        std::cout << "Error creating Storage!" << std::endl;
        exit(1);
    }

    readme << std::endl;
    const int& groupSize = storage.size();
    std::vector<int> element;
    for(int i=0;i<groupSize;++i)
    {
        element = storage[i];
        if(element.empty())
            continue;
        for(int j=0;j<element.size();++j)
            readme << element[j] << " ";
        readme << std::endl;
    }
    std::cout << std::endl;
    readme.close();
}
```

4.8.2.8 void IOHandler::generateGroups (const std::vector< std::vector< int > > & *storage*, const string & *fileName*) [static]

Definition at line 1774 of file IOHandler.cpp.

```
{
    if(storage.empty())
        return;
    std::ofstream readme(("../dataset/"+fileName).c_str(),ios::out);
    if(!readme)
    {
        std::cout << "Error creating Storage!" << std::endl;
        exit(1);
    }

    readme << std::endl;
    const int& groupSize = storage.size();
    std::vector<int> element;
    for(int i=0;i<groupSize;++i)
    {
        element = storage[i];
        if(element.empty())
            continue;
        for(int j=0;j<element.size();++j)
            readme << element[j] << " ";
        readme << std::endl;
    }
    std::cout << std::endl;
    readme.close();
}
```

4.8.2.9 void IOHandler::generateReadme (const std::vector< string > & *activityList*, const std::vector< double > & *timeList*, const int & *normOption*, const int & *numClusters*, const float & *sValue*, const float & *threshold*) [static]

Definition at line 1635 of file IOHandler.cpp.

```
{
    if(activityList.empty()||timeList.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "----------------------------------------------" << std::endl;
    readme << "Norm: " << normOption << std::endl;
    readme << "Clusters: " << numClusters << std::endl;
    readme << "Silhouette: " << sValue << std::endl;
    readme << "Input threshold: " << threshold << std::endl;
    for (int i = 0; i < activityList.size(); ++i)
    {
        readme << activityList[i] << timeList[i] << " s." << std::endl;
    }
    readme << std::endl;
    readme.close();
}
```

4.8.2.10 void IOHandler::generateReadme (const std::vector< string > & *activityList*, const std::vector< string > & *timeList*) [static]

Definition at line 1670 of file IOHandler.cpp.

```
{
    if(activityList.empty()||timeList.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "----------------------------------------------" << std::endl;
    for (int i = 0; i < activityList.size(); ++i)
    {
        readme << activityList[i] << timeList[i] << std::endl;
    }
    readme.close();
}
```

4.8.2.11 void IOHandler::printClusters (const std::vector< std::vector< float > > & *dataVec*, const std::vector< int > & *group*, const std::vector< int > & *storage*, const string & *groupName*, const string & *fullName*, const int & *dimension*) [static]

Definition at line 1526 of file IOHandler.cpp.

```
{
    if(group.empty()||storage.empty())
        return;
    std::ofstream fout(fullName.c_str(), ios::out | ios::app );
    if(!fout)
    {
        std::cout << "Error opening the file!" << std::endl;
        exit(1);
    }

    fout << "SCALARS " << groupName << " int 1" << std::endl;
    fout << "LOOKUP_TABLE " << groupName+string("_table") << std::endl;

    int arraySize;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << group[i] << std::endl;
        }
    }

    fout << "SCALARS " <<  groupName + "_num" << " int 1" << std::endl;
    fout << "LOOKUP_TABLE " <<  groupName+string("_num_table") << std::endl;

    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << storage[group[i]] << std::endl;
        }
    }
    fout.close();
}
```

4.8.2.12 void IOHandler::printClustersNoise (const std::vector< std::vector< float > > & *dataVec*, const std::vector< int > & *group*, const std::vector< int > & *storage*, const string & *groupName*, const string & *fullName*, const int & *dimension*) [static]

Definition at line 1580 of file IOHandler.cpp.

```
{
    /* in case you've noise, so group_id would be -1 */
    if(group.empty()||storage.empty())
        return;
    std::ofstream fout(fullName.c_str(), ios::out | ios::app );
    if(!fout)
    {
        std::cout << "Error opening the file!" << std::endl;
        exit(1);
    }

    fout << "SCALARS " << groupName << " int 1" << std::endl;
    fout << "LOOKUP_TABLE " << groupName+string("_table") << std::endl;

    int arraySize;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << group[i] << std::endl;
        }
    }

    fout << "SCALARS " <<  groupName + "_num" << " int 1" << std::endl;
    fout << "LOOKUP_TABLE " <<  groupName+string("_num_table") << std::endl;

    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << storage[group[i]+1] << std::endl;
        }
    }
    fout.close();
}
```

4.8.2.13 void IOHandler::printFeature (const string & *fileName*, const std::vector< std::vector< float > > & *array*, const std::vector< float > & *sCluster*, const int & *dimension*) [static]

Definition at line 1323 of file IOHandler.cpp.

```
{
    if(array.empty() || sCluster.empty())
        return;
    stringstream ss;
    ss << "../dataset/" << fileName;
    ofstream fout(ss.str().c_str(), ios::out);
    if(!fout)
    {
        std::cout << "Error creating file!" << std::endl;
        exit(-1);
    }

    int vertexCount = 0;
    for (int i = 0; i < array.size(); ++i)
    {
        vertexCount += array[i].size();
    }
    vertexCount /= dimension;

    fout << "# vtk DataFile Version 3.0" << std::endl << "Bernard streamline" << std::endl
         << "ASCII" << std::endl << "DATASET POLYDATA" << std::endl;
    fout << "POINTS " << vertexCount << " float" << std::endl;

    int subSize, arraySize;
    std::vector<float> tempVec;
    for (int i = 0; i < array.size(); ++i)
    {
        tempVec = array[i];
        subSize = tempVec.size()/dimension;
        for (int j = 0; j < subSize; ++j)
        {
            for (int k = 0; k < dimension; ++k)
            {
                fout << tempVec[j*dimension+k] << " ";
            }
            fout << endl;
        }
    }

    fout << "LINES " << array.size() << " " << (vertexCount+array.size()) << std::endl;

    subSize = 0;
    for (int i = 0; i < array.size(); ++i)
    {
        arraySize = array[i].size()/dimension;
        fout << arraySize << " ";
        for (int j = 0; j < arraySize; ++j)
        {
            fout << subSize+j << " ";
        }
        subSize+=arraySize;
        fout << std::endl;
    }
    fout << "POINT_DATA" << " " << vertexCount << std::endl;
    fout << "SCALARS group int 1" << std::endl;
    fout << "LOOKUP_TABLE group_table" << std::endl;

    for (int i = 0; i < array.size(); ++i)
    {
        arraySize = array[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << i << std::endl;
        }
    }

    fout << "SCALARS silhouette float 1" << std::endl;
    fout << "LOOKUP_TABLE silhouette_table" << std::endl;

    for (int i = 0; i < array.size(); ++i)
    {
        arraySize = array[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << sCluster[i] << std::endl;
        }
    }
    fout.close();
}
```

4.8.2.14 void IOHandler::printFeature (const string & *fileName*, const std::vector< std::vector< float > > & *array*, const std::vector< float > & *sCluster*, const std::vector< float > & *rotation*, const int & *dimension*) [static]

Definition at line 1417 of file IOHandler.cpp.

```
{
    if(array.empty() || sCluster.empty())
            return;
        stringstream ss;
        ss << "../dataset/" << fileName;
        ofstream fout(ss.str().c_str(), ios::out);
        if(!fout)
        {
            std::cout << "Error creating file!" << std::endl;
            exit(-1);
        }

        int vertexCount = 0;
        for (int i = 0; i < array.size(); ++i)
        {
            vertexCount += array[i].size();
        }
        vertexCount /= dimension;

        fout << "# vtk DataFile Version 3.0" << std::endl << "Bernard streamline" << std::endl
             << "ASCII" << std::endl << "DATASET POLYDATA" << std::endl;
        fout << "POINTS " << vertexCount << " float" << std::endl;

        int subSize, arraySize;
        std::vector<float> tempVec;
        for (int i = 0; i < array.size(); ++i)
        {
            tempVec = array[i];
            subSize = tempVec.size()/dimension;
            for (int j = 0; j < subSize; ++j)
            {
                for (int k = 0; k < dimension; ++k)
                {
                    fout << tempVec[j*dimension+k] << " ";
                }
                fout << endl;
            }
        }

        fout << "LINES " << array.size() << " " << (vertexCount+array.size()) << std::endl;

        subSize = 0;
        for (int i = 0; i < array.size(); ++i)
        {
            arraySize = array[i].size()/dimension;
            fout << arraySize << " ";
            for (int j = 0; j < arraySize; ++j)
            {
                fout << subSize+j << " ";
            }
            subSize+=arraySize;
            fout << std::endl;
        }
        fout << "POINT_DATA" << " " << vertexCount << std::endl;
        fout << "SCALARS group int 1" << std::endl;
        fout << "LOOKUP_TABLE group_table" << std::endl;

        for (int i = 0; i < array.size(); ++i)
        {
            arraySize = array[i].size()/dimension;
            for (int j = 0; j < arraySize; ++j)
            {
                fout << i << std::endl;
            }
        }

        fout << "SCALARS silhouette float 1" << std::endl;
        fout << "LOOKUP_TABLE silhouette_table" << std::endl;

        for (int i = 0; i < array.size(); ++i)
        {
            arraySize = array[i].size()/dimension;
            for (int j = 0; j < arraySize; ++j)
            {
                fout << sCluster[i] << std::endl;
            }
        }

        fout << "SCALARS rotation float 1" << std::endl;
        fout << "LOOKUP_TABLE rotation_table" << std::endl;

        for (int i = 0; i < array.size(); ++i)
        {
            arraySize = array[i].size()/dimension;
            for (int j = 0; j < arraySize; ++j)
            {
                fout << rotation[i] << std::endl;
            }
        }

        fout.close();
}
```

4.8.2.15 void IOHandler::printQuery (const int & *normOption*, const int & *order*, const StringQuery & *queryResult*, const std::vector< std::vector< float > > & *dataVec*) [static]

Definition at line 1120 of file IOHandler.cpp.

```
{
    stringstream ss;
    ss << "../dataset/norm" << normOption << "_query" << order << "_target.vtk";
    const string& targetStr = ss.str();
    ss.str("");
    ss.clear();

    ss << "../dataset/norm" << normOption << "_query" << order << "_result.vtk";
    const string& resultStr = ss.str();

    /* print out the target streamline vtk file */
    ofstream fTarget(targetStr.c_str(),ios::out);
    if(!fTarget)
    {
        std::cout << "Error creating file!" << std::endl;
        exit(1);
    }
    std::cout << queryResult.index << std::endl;
    std::vector<float> targetVec = dataVec[queryResult.index];
    int pointNumber = targetVec.size()/3;

    fTarget << "# vtk DataFile Version 3.0" << std::endl << "Bernard streamline" << std::endl
            << "ASCII" << std::endl << "DATASET POLYDATA" << std::endl;
    fTarget << "POINTS " << pointNumber << " float" << std::endl;

    for (int i = 0; i < pointNumber; ++i)
    {
        fTarget << targetVec[i*3] << " " << targetVec[i*3+1]
                << " " << targetVec[i*3+2] << std::endl;
    }

    fTarget << "LINES " << 1 << " " << (1+pointNumber) << std::endl;
    fTarget << pointNumber << " ";
    for (int i = 0; i < pointNumber; ++i)
    {
        fTarget << i << " ";
    }
    fTarget << std::endl;
    fTarget.close();


    /* print out the streamline query result vtk file */
    ofstream fResult(resultStr.c_str(),ios::out);
    if(!fResult)
    {
        std::cout << "Error creating file!" << std::endl;
        exit(1);
    }
    pointNumber = 0;
    const std::vector<int>& neighbor = queryResult.neighbor;
    for (int i = 0; i < neighbor.size(); ++i)
    {
        pointNumber += dataVec[neighbor[i]].size()/3;
    }

    fResult << "# vtk DataFile Version 3.0" << std::endl << "Bernard streamline" << std::endl
            << "ASCII" << std::endl << "DATASET POLYDATA" << std::endl;
    fResult << "POINTS " << pointNumber << " float" << std::endl;

    int subArraySize, indexNumber = 0;
    std::vector<float> tempVec;
    for (int i = 0; i < neighbor.size(); ++i)
    {
        tempVec = dataVec[neighbor[i]];
        subArraySize = tempVec.size()/3;
        for (int j = 0; j < subArraySize; ++j)
        {
            fResult << tempVec[3*j] << " " << tempVec[3*j+1] << " "
                    << tempVec[3*j+2] << std::endl;
        }
    }

    fResult << "LINES " << neighbor.size() << " "
            << (neighbor.size()+pointNumber) << std::endl;
    for (int i = 0; i < neighbor.size(); ++i)
    {
        tempVec = dataVec[neighbor[i]];
        subArraySize = tempVec.size()/3;
        fResult << subArraySize << " ";
        for (int j = 0; j < subArraySize; ++j)
        {
            fResult << (indexNumber+j) << " ";
        }
        fResult << std::endl;
        indexNumber += subArraySize;
    }
    fResult.close();

}
```

4.8.2.16 void IOHandler::printToFull (const std::vector< std::vector< float > > & *dataVec*, const std::vector< int > & *group*, const std::vector< int > & *totalNum*, const string & *groupName*, const string & *fullName*, const int & *dimension*) [static]

Definition at line 655 of file IOHandler.cpp.

```
{
    if(group.empty()||totalNum.empty())
        return;
    std::ofstream fout(fullName.c_str(), ios::out | ios::app );
    if(!fout)
    {
        std::cout << "Error opening the file!" << std::endl;
        exit(1);
    }

    fout << "SCALARS " << groupName << " int 1" << std::endl;
    fout << "LOOKUP_TABLE " << groupName+string("_table") << std::endl;

    int arraySize;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << group[i] << std::endl;
        }
    }

    fout << "SCALARS " <<  groupName + "_num" << " int 1" << std::endl;
    fout << "LOOKUP_TABLE " <<  groupName+string("_num_table") << std::endl;

    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << totalNum[i] << std::endl;
        }
    }
    fout.close();
}
```

4.8.2.17 void IOHandler::printToFull (const std::vector< std::vector< float > > & *dataVec*, const std::vector< float > & *sData*, const string & *groupName*, const string & *fullName*, const int & *dimension*) [static]

Definition at line 708 of file IOHandler.cpp.

```
{
    if(sData.empty()||dataVec.empty())
        return;
    std::ofstream fout(fullName.c_str(), ios::out | ios::app );
    if(!fout)
    {
        std::cout << "Error opening the file!" << std::endl;
        exit(1);
    }

    if(!sData.empty())
    {
        fout << "SCALARS " << groupName << " float 1" << std::endl;
        fout << "LOOKUP_TABLE " << groupName+string("_table") << std::endl;

        int arraySize;
        for (int i = 0; i < dataVec.size(); ++i)
        {
            arraySize = dataVec[i].size()/dimension;
            for (int j = 0; j < arraySize; ++j)
            {
                fout << sData[i] << std::endl;
            }
        }
    }

    fout.close();
}
```

4.8.2.18 void IOHandler::printToFull (const std::vector< std::vector< float > > & *origin*, const std::vector< int > & *group*, const string & *fullName*, const string & *groupName*, const int & *dimension*) [static]

Definition at line 1286 of file IOHandler.cpp.

```
{
    std::ofstream fout(fullName.c_str(), ios::out | ios::app );
    if(!fout)
    {
        std::cout << "Error opening the file!" << std::endl;
        exit(1);
    }

    fout << "SCALARS " << groupName << " int 1" << std::endl;
    fout << "LOOKUP_TABLE " << groupName+string("_table") << std::endl;

    int arraySize;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << group[i] << std::endl;
        }
    }
    fout.close();
}
```

4.8.2.19 void IOHandler::printToFull (const std::vector< std::vector< float > > & *dataVec*, const std::vector< int > & *group*, const std::vector< float > & *sCluster*, const string & *groupName*, const string & *fullName*, const int & *dimension*) [static]

Definition at line 753 of file IOHandler.cpp.

```
{
    if(dataVec.empty()||group.empty()||sCluster.empty())
        return;
    std::ofstream fout(fullName.c_str(), ios::out | ios::app );
    if(!fout)
    {
        std::cout << "Error opening the file!" << std::endl;
        exit(1);
    }

    if(!sCluster.empty())
    {
        fout << "SCALARS " <<  groupName << " float 1" << std::endl;
        fout << "LOOKUP_TABLE " <<  groupName+string("_table") << std::endl;

        int arraySize;
        for (int i = 0; i < dataVec.size(); ++i)
        {
            arraySize = dataVec[i].size()/dimension;
            for (int j = 0; j < arraySize; ++j)
            {
                if(group[i]<0)
                    fout << 0 << std::endl;
                else
                    fout << sCluster[group[i]] << std::endl;
            }
        }
    }
    fout.close();
}
```

4.8.2.20 void IOHandler::printTXT (float ** *data*, const int & *Row*, const int & *Column*) [static]

Definition at line 1222 of file IOHandler.cpp.

```
{
    std::ofstream fout("../dataset/full.txt", ios::out);
    if(!fout)
    {
        std::cout << "Error creating a file!" << std::endl;
        exit(1);
    }
    float *array;
    for (int i = 0; i < Row; ++i)
    {
        array = data[i];
        for (int j = 0; j < Column; ++j)
        {
            fout << array[j] << " ";
        }
        fout << std::endl;
    }
    fout.close();
}
```

4.8.2.21 void IOHandler::printVTK (const string & *fileName*, const std::vector< std::vector< float > > & *dataVec*, const int & *vertexCount*, const int & *dimension*, const std::vector< int > & *clusterNumber*, const std::vector< float > & *sCluster*) [static]

Definition at line 186 of file IOHandler.cpp.

```
{
    if(clusterNumber.empty()||sCluster.empty())
        return;
    std::ofstream fout(fileName.c_str(), ios::out);
    if(!fout)
    {
        std::cout << "Error creating a new file!" << std::endl;
        exit(1);
    }
    fout << "# vtk DataFile Version 3.0" << std::endl << "Bernard streamline" << std::endl
```

```
         << "ASCII" << std::endl << "DATASET POLYDATA" << std::endl;
    fout << "POINTS " << vertexCount << " float" << std::endl;

    int subSize, arraySize;
    std::vector<float> tempVec;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        tempVec = dataVec[i];
        subSize = tempVec.size()/dimension;
        for (int j = 0; j < subSize; ++j)
        {
            for (int k = 0; k < dimension; ++k)
            {
                fout << tempVec[j*dimension+k] << " ";
            }
            fout << endl;
        }
    }

    fout << "LINES " << dataVec.size() << " " << (vertexCount+dataVec.size()) << std::endl;

    subSize = 0;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        fout << arraySize << " ";
        for (int j = 0; j < arraySize; ++j)
        {
            fout << subSize+j << " ";
        }
        subSize+=arraySize;
        fout << std::endl;
    }
    fout << "POINT_DATA" << " " << vertexCount << std::endl;
    fout << "SCALARS group int 1" << std::endl;
    fout << "LOOKUP_TABLE group_table" << std::endl;

    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << clusterNumber[i] << std::endl;
        }
    }

    if(!sCluster.empty())
    {
        fout << "SCALARS sCluster float 1" << std::endl;
        fout << "LOOKUP_TABLE sCluster_table" << std::endl;

        for (int i = 0; i < dataVec.size(); ++i)
        {
            arraySize = dataVec[i].size()/dimension;
            for (int j = 0; j < arraySize; ++j)
            {
                fout << sCluster[clusterNumber[i]] << std::endl;
            }
        }
    }

    fout.close();
}
```

4.8.2.22 void IOHandler::printVTK (const string & *fileName*, const std::vector< std::vector< float > > & *dataVec*, const int & *vertexCount*, const int & *dimension*) [static]

Definition at line 275 of file IOHandler.cpp.

```
{
    if(dataVec.empty())
        return;
    std::ifstream fin(fileName.c_str());
    if(fin.good())
        return;
    std::ofstream fout(fileName.c_str(), ios::out);
    if(!fout)
    {
        std::cout << "Error creating a new file!" << std::endl;
        exit(1);
    }
    fout << "# vtk DataFile Version 3.0" << std::endl << "Bernard streamline" << std::endl
         << "ASCII" << std::endl << "DATASET POLYDATA" << std::endl;
    fout << "POINTS " << vertexCount << " float" << std::endl;

    int subSize, arraySize;
    std::vector<float> tempVec;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        tempVec = dataVec[i];
        subSize = tempVec.size()/dimension;
        for (int j = 0; j < subSize; ++j)
        {
            for (int k = 0; k < dimension; ++k)
            {
                fout << tempVec[j*dimension+k] << " ";
            }
            fout << endl;
        }
    }

    fout << "LINES " << dataVec.size() << " " << (vertexCount+dataVec.size()) << std::endl;

    subSize = 0;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        fout << arraySize << " ";
        for (int j = 0; j < arraySize; ++j)
        {
            fout << subSize+j << " ";
        }
        subSize+=arraySize;
        fout << std::endl;
    }
    fout << "POINT_DATA" << " " << vertexCount << std::endl;
    fout << "SCALARS group int 1" << std::endl;
    fout << "LOOKUP_TABLE group_table" << std::endl;

    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << i << std::endl;
        }
    }

    fout.close();
}
```

4.8.2.23 void IOHandler::printVTK (const string & *fileName*, const std::vector< MeanLine > & *dataVec*, const int & *vertexCount*, const int & *dimension*, const std::vector< float > & *sCluster*) [static]

Definition at line 351 of file IOHandler.cpp.

```
{
    if(dataVec.empty())
        return;
    std::ofstream fout(fileName.c_str(), ios::out);
    if(!fout)
    {
        std::cout << "Error creating a new file!" << std::endl;
        exit(1);
    }
    fout << "# vtk DataFile Version 3.0" << std::endl << "Bernard streamline" << std::endl
         << "ASCII" << std::endl << "DATASET POLYDATA" << std::endl;
    fout << "POINTS " << vertexCount << " float" << std::endl;

    int subSize, arraySize;
    std::vector<float> tempVec;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        tempVec = dataVec[i].minCenter;
        subSize = tempVec.size()/dimension;
        for (int j = 0; j < subSize; ++j)
        {
            for (int k = 0; k < dimension; ++k)
            {
                fout << tempVec[j*dimension+k] << " ";
            }
            fout << endl;
        }
    }

    fout << "LINES " << dataVec.size() << " " << (vertexCount+dataVec.size()) << std::endl;

    subSize = 0;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].minCenter.size()/dimension;
        fout << arraySize << " ";
        for (int j = 0; j < arraySize; ++j)
        {
            fout << subSize+j << " ";
        }
        subSize+=arraySize;
        fout << std::endl;
    }
    fout << "POINT_DATA" << " " << vertexCount << std::endl;
    fout << "SCALARS group int 1" << std::endl;
    fout << "LOOKUP_TABLE group_table" << std::endl;

    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].minCenter.size()/dimension;
        for (int j = 0; j < arraySize; ++j)
        {
            fout << dataVec[i].cluster << std::endl;
        }
    }

    if(!sCluster.empty())
    {
        fout << "SCALARS sCluster float 1" << std::endl;
        fout << "LOOKUP_TABLE sCluster_table" << std::endl;

        for (int i = 0; i < dataVec.size(); ++i)
        {
            arraySize = dataVec[i].minCenter.size()/dimension;
            for (int j = 0; j < arraySize; ++j)
            {
                fout << sCluster[dataVec[i].cluster] << std::endl;
            }
        }
    }

    fout.close();
}
```

4.8.2.24 void IOHandler::readClusteringNumber (std::unordered_map< int, int > & *clusMap*, const string & *fileName*) [static]

Definition at line 1832 of file IOHandler.cpp.

```
{
    std::ifstream readme(("../dataset/"+fileName).c_str(), ios::in);
    if(!readme)
    {
        std::cout << "Error creating Storage!" << std::endl;
        exit(1);
    }
    string line;
    int scopePos;
    while(getline(readme, line))
    {
        scopePos = line.find(":");
        if(scopePos==std::string::npos)
        {
            std::cout << "Error for clustering number reader..." << std::endl;
            exit(1);
        }
        clusMap[std::atoi(line.substr(0,scopePos).c_str())] = std::atoi(line.substr(scopePos+1).c_str());
    }
    readme.close();
}
```

4.8.2.25 void IOHandler::readFile (const string & *fileName*, std::vector< std::vector< float > > & *dataVec*, int & *vertexCount*, const int & *dimension*, int & *maxElement*) [static]

Definition at line 19 of file IOHandler.cpp.

```
{

    vertexCount = 0;
    std::ifstream fin(fileName.c_str(), ios::in);
    if(!fin)
    {
        std::cout << "Error creating files!" << std::endl;
        exit(1);
    }
    stringstream ss;
    std::vector<float> tempVec;

    string line, part;

    /* read partial number of streamlines */
    //int MAXNUMBER;
    //std::cout << "Input maximal trajectory numbers: " << std::endl;
    //std::cin >> MAXNUMBER;
    // set currentNumber to record how many streamlines u want to read in
    //int currentNumber = 0;


    /* read partial dimensions of curves */
    //int MAXDIMENSION;
    //std::cout << "Input maximal dimensions: " << std::endl;
    //std::cin >> MAXDIMENSION;
    // set currentNumber to record how many streamlines u want to read in
    //int currentDimensions;

    std::vector<float> vec(3);
    float temp;
    maxElement = 0;
    while(getline(fin, line) /* && currentNumber < MAXNUMBER*/)
    {
        //currentDimensions = 0;
        int tag = 0, count = 0;
        ss.str(line);
        while(ss>>part /*&& currentDimensions<3*MAXDIMENSION*/)
        {
            /* operations below would remove duplicate vertices because that would damage our computation */
            temp = atof(part.c_str());
            if(tag>=3)
            {
                if(count<3)
                {
                    vec[count] = temp;
                    ++tag;
                    ++count;
                }
                if(count==3)
                {
                    int size = tempVec.size();
                    //if(!(abs(vec[0]-tempVec[size-3])<1.0e-5&&abs(vec[1]-tempVec[size-2])<1.0e-5&&abs(vec[2]-tempVec.back())<1.0e-5))
                    //{
                    tempVec.push_back(vec[0]);
                    tempVec.push_back(vec[1]);
                    tempVec.push_back(vec[2]);
                    //}
                    count = 0;
                }
                continue;
            }
            tempVec.push_back(temp);
            ++tag;
            //currentDimensions++;
        }
        /* accept only streamlines with at least three vertices */
        if(tempVec.size()/3>2)
        {
            if(maxElement<tempVec.size())
                maxElement = tempVec.size();
            dataVec.push_back(tempVec);
            vertexCount+=tempVec.size();
        }
        tempVec.clear();
        ss.clear();
        ss.str("");
        //currentNumber++;
    }
    fin.close();


    vertexCount/=dimension;
    std::cout << "File reader has been completed, and it toally has " << dataVec.size() << " trajectories
   and "
              << vertexCount << " vertices!" << std::endl;
    std::cout << "Max dimension is " << maxElement << std::endl;
}
```

4.8.2.26 void IOHandler::readFile (const string & *fileName,* std::vector< std::vector< float > > & *dataVec,* int & *vertexCount,* const int & *dimension,* const int & *trajectoryNum,* const int & *Frame*) [static]

Definition at line 123 of file IOHandler.cpp.

```
{
    vertexCount = trajectoryNum*(Frame-1);
    dataVec = std::vector< std::vector<float> >(trajectoryNum, std::vector<float> ((Frame-1)*dimension));
#pragma omp parallel for schedule(static) num_threads(8)
    /* from 1 to Frame-1 then pay attention to i index */
    for (int i = 1; i < Frame; ++i)
    {
        stringstream ss;
        ss << fileName << i << ".txt";
        std::ifstream fin(ss.str().c_str(), ios::in);
        if(!fin)
        {
            std::cout << "File doesn't exist for this number!" << std::endl;
            exit(1);
        }
        float firstFloat;
        string line, linePart;

        ss.clear();
        ss.str("");
        for (int j = 0; j < trajectoryNum; ++j)
        {
            getline(fin, line);

            assert(!line.empty());

            ss.str(line);
            ss >> linePart;

            ss >> linePart;
            dataVec[j][(i-1)*dimension] = atof(linePart.c_str());

            ss >> linePart;
            dataVec[j][(i-1)*dimension+1] = atof(linePart.c_str());

            ss >> linePart;
            dataVec[j][(i-1)*dimension+2] = atof(linePart.c_str());
        }

        fin.close();
        std::cout << "File " << i << " has been read in successfully!" << std::endl;
    }


}
```

4.8.2.27 void IOHandler::sampleArray (MatrixXf & *data,* const std::vector< std::vector< float > > & *dataVec,* const int & *dimension,* const int & *maxElements*) [static]

Definition at line 551 of file IOHandler.cpp.

```
{
    /*maxElements = INT_MIN;
    int arraySize;
    for (int i = 0; i < dataVec.size(); ++i)
    {
        arraySize = dataVec[i].size();
        if(maxElements < arraySize)
            maxElements = arraySize;
    }
    std::cout << maxElements << std::endl;*/
    //temp.row(i) = Eigen::VectorXf::Map(&each[0], 10); //must match the column size
    data = Eigen::MatrixXf(dataVec.size(), maxElements);
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < dataVec.size(); ++i)
    {
        const std::vector<float>& eachVec = dataVec[i]; //cached vector<float>
        if(eachVec.size()==maxElements)
        {
            data.row(i) = Eigen::VectorXf::Map(&eachVec[0], maxElements);
        }
        else
        {
            const int& pointNum = eachVec.size()/3; //current vec point length
            const int& totalNum = maxElements/3; //totally maximal point length
            const int& segNum = pointNum-1;
            const int& averageAdd = (totalNum-pointNum)/segNum; //average point on each segment

        //# of segments with averageAdd+1 sampled points
            const int& averageRes = (totalNum-pointNum)%segNum;

            int segmentLength;
            int currentPoint = 0;

            Eigen::Vector3f meanLength, insertedPoint;
            Eigen::Vector3f front, end;

            int j;
            for(j=0; j<segNum;j++)  //traverse all segments
            {
                for(int k=0; k<3;k++)
                    data(i,3*currentPoint+k) = eachVec[3*j+k];
                currentPoint++;
                if(j<segNum-averageRes)
                    segmentLength = averageAdd;
                else
                    segmentLength = averageAdd+1;
                if(segmentLength>=1)
                {
                    front << eachVec[3*j], eachVec[3*j+1], eachVec[3*j+2];
                    end << eachVec[3*j+3], eachVec[3*j+4], eachVec[3*j+5];
                    meanLength = (end-front)/(segmentLength+1);
                    for(int k=1; k<=segmentLength; k++)
                    {
                        insertedPoint = front+k*meanLength;
                        for(int s=0; s<3;s++)
                            data(i,3*currentPoint+s) = insertedPoint(s);
                        currentPoint++;
                    }
                }
            }
            assert(currentPoint==totalNum-1);
            for(int k=0; k<3;k++)
                data(i,3*currentPoint+k) = eachVec[3*j+k];
        }
    }
}
```

4.8.2.28 void IOHandler::uniformArcSampling (MatrixXf & *data*, const std::vector< std::vector< float > > & *dataVec*, const int & *dimension*, const int & *maxElements*) [static]

Definition at line 472 of file IOHandler.cpp.

```
{
    const int& numOfRows = dataVec.size();

    /* assign memory for required matrix */
    const int& totalSize = 3*maxElements;
    data = Eigen::MatrixXf(numOfRows, totalSize);
#pragma omp parallel for schedule(static) num_threads(8)
```

```
    /* compute total length of streamline and record each cumulative length information */
    for(int i=0;i<numOfRows;++i)
    {
        float entireLength = 0.0, lineLength;
        const std::vector<float>& eachVec = dataVec[i];
        const int& vecSize = eachVec.size();
        const int& lineNum = vecSize/3-1;
        Eigen::Vector3f lineSeg;

        vector<float> pairwise(lineNum);
        for(int j=0;j<lineNum;++j)
        {
            lineSeg(0)=eachVec[3*j+3]-eachVec[3*j];
            lineSeg(1)=eachVec[3*j+4]-eachVec[3*j+1];
            lineSeg(2)=eachVec[3*j+5]-eachVec[3*j+2];
            lineLength = lineSeg.norm();
            entireLength+=lineLength;
            pairwise[j]=entireLength;
        }

        float eachLength = entireLength/(maxElements-1);
        Eigen::VectorXf row_i(data.row(i).size());

        /* insert starting vertex */
        row_i(0)=eachVec[0];
        row_i(1)=eachVec[1];
        row_i(2)=eachVec[2];

        /* insert ending vertex */
        row_i(totalSize-3)=eachVec[vecSize-3];
        row_i(totalSize-2)=eachVec[vecSize-2];
        row_i(totalSize-1)=eachVec[vecSize-1];

        float tempLength, tempRatio, ratioComplement;
        int preLine, preVertex, postVertex;
        for(int j=1;j<=maxElements-2;++j)
        {
            /* current length */
            tempLength = j*eachLength;

            /* pre-index of coordinates */
            preLine = std::lower_bound(pairwise.begin(), pairwise.end(), tempLength)-pairwise.begin()-1;

            /* locate vertex index */
            preVertex = preLine+1;
            postVertex = preVertex+1;

            /* use linear interpolation to generate new coordinates */
            tempRatio = (tempLength-pairwise[preLine])/(pairwise[preLine+1]-pairwise[preLine]);
            ratioComplement = 1.0-tempRatio;
            row_i(3*j) = tempRatio*eachVec[3*postVertex]+ratioComplement*eachVec[3*preVertex];
            row_i(3*j+1) = tempRatio*eachVec[3*postVertex+1]+ratioComplement*eachVec[3*preVertex+1];
            row_i(3*j+2) = tempRatio*eachVec[3*postVertex+2]+ratioComplement*eachVec[3*preVertex+2];
        }

        data.row(i) = row_i;
    }
}
```

4.8.2.29 void IOHandler::writeGroup (const std::vector< int > & *group*, const std::vector< std::vector< float > > & *dataVec*) [static]

Definition at line 1092 of file IOHandler.cpp.

```
{
    if(group.empty()||dataVec.empty())
        return;
    std::ofstream readme("../dataset/group",ios::out);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    assert(group.size()==dataVec.size());
    for (int i = 0; i < group.size(); ++i)
    {
        readme << group[i] << std::endl;
    }
    readme.close();
}
```

4.8.2.30 void IOHandler::writeGroupSize (const std::vector< int > & *storage*) [static]

Definition at line 1715 of file IOHandler.cpp.

```
{
    if(storage.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "Final cluster size: " << storage.size() << std::endl;
    for (int i = 0; i < storage.size(); ++i)
    {
        readme << storage[i] << " ";
    }
    readme << std::endl;
    readme.close();
}
```

4.8.2.31 void IOHandler::writeReadme (const double & *PCA_KMeans_delta,* const double & *KMeans_delta*) [static]

Definition at line 817 of file IOHandler.cpp.

```
{
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "PCA_KMeans time elapse is " << PCA_KMeans_delta << " s." << std::endl
           << "KMeans time elapse is " << KMeans_delta << " s." << std::endl;
    readme << std::endl;
    readme.close();
}
```

4.8.2.32 void IOHandler::writeReadme (const string & *comment,* const std::vector< float > & *sAverage*) [static]

Definition at line 839 of file IOHandler.cpp.

```
{
    if(sAverage.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << comment << std::endl;
    for (int i = 0; i < sAverage.size(); ++i)
    {
        readme << sAverage[i] << std::endl;
    }
    readme << std::endl;
    readme.close();
}
```

4.8.2.33 void IOHandler::writeReadme (const std::vector< string > & *timeName*, const std::vector< double > & *timeDiff*, const int & *cluster*) [static]

Definition at line 867 of file IOHandler.cpp.

```
{
    if(timeName.empty()||timeDiff.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    assert(timeName.size()==timeDiff.size());
    for (int i = 0; i < timeName.size(); ++i)
    {
        readme << timeName[i] << " is " << timeDiff[i] << " s." << std::endl;
    }
    readme << std::endl;
    readme << "Preset cluster number in K-means is: " << cluster << std::endl;
    readme << std::endl;
    readme.close();
}
```

4.8.2.34 void IOHandler::writeReadme (const std::vector< string > & *timeName*, const std::vector< string > & *timeDiff*, const int & *cluster*) [static]

Definition at line 898 of file IOHandler.cpp.

```
{
    if(timeName.empty()||timeDiff.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    assert(timeName.size()==timeDiff.size());
    for (int i = 0; i < timeName.size(); ++i)
    {
        readme << timeName[i] << " " << timeDiff[i] << std::endl;
    }
    readme << "Preset cluster number in K-means is: " << cluster << std::endl;
    readme << std::endl;
    readme.close();
}
```

4.8.2.35 void IOHandler::writeReadme (const std::vector< ExtractedLine > & *closest*, const std::vector< ExtractedLine > & *furthest*, const int & *normOption*) [static]

Definition at line 928 of file IOHandler.cpp.

```
{
    if(closest.empty()||furthest.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    const string& normStr = "Norm_"+to_string(normOption);
    readme << std::endl;
    readme << normStr+ " closest streamline set has " << closest.size() << " streamlines" << std::endl;
    for (int i = 0; i < closest.size(); ++i)
    {
        readme << closest[i].lineNum << " ";
    }
    readme << std::endl;

    readme << std::endl;
    readme << normStr+ " furthest streamline set has " << furthest.size() << " streamlines" << std::endl;
    for (int i = 0; i < furthest.size(); ++i)
    {
        readme << furthest[i].lineNum << " ";
    }
    readme << std::endl;
    readme.close();
}
```

4.8.2.36 void IOHandler::writeReadme (const std::vector< ExtractedLine > & *closest*, const std::vector< ExtractedLine > & *furthest*) [static]

Definition at line 966 of file IOHandler.cpp.

```
{
    if(closest.empty()||furthest.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << std::endl;
    readme << "PCA closest streamline set has " << closest.size() << " streamlines" << std::endl;
    for (int i = 0; i < closest.size(); ++i)
    {
        readme << closest[i].lineNum << " ";
    }
    readme << std::endl;

    readme << std::endl;
    readme << "PCA furthest streamline set has " << furthest.size() << " streamlines" << std::endl;
    for (int i = 0; i < furthest.size(); ++i)
    {
        readme << furthest[i].lineNum << " ";
    }
    readme << std::endl;
    readme.close();
}
```

4.8.2.37 void IOHandler::writeReadme (const float & *closestAverage*, const float & *furthestAverage*) [static]

Definition at line 1026 of file IOHandler.cpp.

```
{
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "The average rotation of closest is: " << closestAverage
           << ", of furthest is: " << furthestAverage << std::endl;
}
```

4.8.2.38 void IOHandler::writeReadme (const string & *comments*) [static]

Definition at line 1695 of file IOHandler.cpp.

```
{
    if(comments.empty())
        return;
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << comments << std::endl;
    readme.close();
}
```

4.8.2.39 void IOHandler::writeReadme (const float & *entropy*, const Silhouette & *sil*, const string & *norm_str*) [static]

Definition at line 1003 of file IOHandler.cpp.

```
{
    std::ofstream readme("../dataset/README",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << norm_str << std::endl;
    readme << "The average silhouette: " << sil.sAverage
           << ", the gamma statistic is: " << sil.gammaStatistic
           << ", the entropy is: " << entropy
           << ", the DB index is: " << sil.dbIndex
           << std::endl;
}
```

4.8.2.40 void IOHandler::writeReadMe (const float & *value*, const string & *dataSet*, const string & *clustering*, const string & *value_name*) [static]

Definition at line 1810 of file IOHandler.cpp.

```
{
    std::ofstream out_file("../dataset/README", ios::out|ios::app);
    if (!out_file)
    {
        std::cout << "Error for creating README!" << std::endl;
        exit(1);
    }
    out_file << "-----------------------------------------------------------------" << std::endl;
    out_file << "The " << value_name << " of " << clustering << " on dataset "
             << dataSet << " is " << value << std::endl;
    out_file.close();
}
```

The documentation for this class was generated from the following files:

- IOHandler.h
- IOHandler.cpp

4.9 MeanLine Struct Reference

```
#include <IOHandler.h>
```

Public Member Functions

- MeanLine (const std::vector< float > &minCenter, const int &cluster)

Public Attributes

- std::vector< float > minCenter
- int cluster

4.9.1 Detailed Description

Definition at line 50 of file IOHandler.h.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 MeanLine::MeanLine (const std::vector< float > & *minCenter,* const int & *cluster*) [inline]

Definition at line 54 of file IOHandler.h.

```
            :minCenter(minCenter), cluster(cluster)
    {}
```

4.9.3 Member Data Documentation

4.9.3.1 int MeanLine::cluster

Definition at line 53 of file IOHandler.h.

4.9.3.2 std::vector<float> MeanLine::minCenter

Definition at line 52 of file IOHandler.h.

The documentation for this struct was generated from the following file:

- IOHandler.h

4.10 MetricPreparation Struct Reference

```
#include <Metric.h>
```

Public Member Functions

- MetricPreparation (const int &Row, const int &Column)
- MetricPreparation ()
- ∼MetricPreparation ()
- void preprocessing (const Eigen::MatrixXf &data, const int &Row, const int &Column, const int &normOption)

Public Attributes

- std::vector< float > rotation
- std::vector< std::vector< float > > rotationSequence
- std::vector< MultiVariate > normalMultivariate
- std::vector< VectorXf > unitLength
- std::vector< std::vector< float > > pairwise
- std::vector< std::vector< float > > pairwiseNorm
- int row
- int column

4.10.1 Detailed Description

Definition at line 163 of file Metric.h.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 MetricPreparation::MetricPreparation (const int & *Row,* const int & *Column*) [inline]

Definition at line 181 of file Metric.h.

```
    {
        /* don't need to allocate redundant memory since the dataset size would be huge */
        row = Row;
        column = Column;
    }
```

4.10.2.2 MetricPreparation::MetricPreparation ( ) [inline]

Definition at line 192 of file Metric.h.

```
    {
        row = column = 0;
    }
```

4.10.2.3 MetricPreparation::∼MetricPreparation ( ) [inline]

Definition at line 200 of file Metric.h.

```
    {
        row = column = -1;
    }
```

4.10.3 Member Function Documentation

4.10.3.1 void MetricPreparation::preprocessing (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, const int & *normOption*) [inline]

Definition at line 213 of file Metric.h.

```
  {
      switch(normOption)
      {
          case 2:
          case 5:
          case 8:
              {
                  pairwise = std::vector<std::vector<float> >(Row, std::vector<float>(Column-3));
                  pairwiseNorm = std::vector<std::vector<float> >(Row, std::vector<float>((Column-3)/3));
                  computePairWise(data, Row, Column-3, pairwise, pairwiseNorm);
              }
              break;

          case 4:
              {
                  /*  if rotation used for judge similarity difference, has to use pre-defined cache */
                  computeMeanRotation(data, Row, Column, rotation);
              }
              break;

          /*  pre-defined cache for sequence mean and standard deviation */
          case 3:
              {
                  rotationSequence = std::vector<std::vector<float> >(Row, std::vector<float>(2));
                  getRotationSequence(data, Row, Column, rotationSequence);
              }
              break;

          case 6:
              {
                  normalMultivariate = std::vector<MultiVariate>(Row, MultiVariate());
                  getNormalSequence(data, Row, Column, normalMultivariate);
              }
              break;

          case 7:
              {
                  rotationSequence = std::vector<std::vector<float> >(Row, std::vector<float>(2));
                  getFixedSequence(data, Row, Column, rotationSequence);
              }
              break;

          case 9:
              {
                  normalMultivariate = std::vector<MultiVariate>(Row, MultiVariate());
                  getUnnormalizedSequence(data, Row, Column, normalMultivariate);
              }
              break;

          case 10:
              {
                  unitLength = std::vector<VectorXf >(Row, VectorXf(Column));
                  getUnitDirection(data, Row, Column, unitLength);
              }
              break;

          case 14:
              {
                  pairwise = std::vector<std::vector<float> >(Row, std::vector<float>(BIN_SIZE));
                  getSignatureBin(data, Row, Column, pairwise);
              }
```

```
                break;

            case 16:
                {
                    pairwise = std::vector<std::vector<float> >(Row, std::vector<float>(2));
                    getBundleEntropy(data, Row, Column, pairwise);
                }

            default:
                break;
        }
    }
```

4.10.4 Member Data Documentation

4.10.4.1 int MetricPreparation::column

Definition at line 172 of file Metric.h.

4.10.4.2 std::vector<MultiVariate> MetricPreparation::normalMultivariate

Definition at line 167 of file Metric.h.

4.10.4.3 std::vector<std::vector<float> > MetricPreparation::pairwise

Definition at line 169 of file Metric.h.

4.10.4.4 std::vector<std::vector<float> > MetricPreparation::pairwiseNorm

Definition at line 170 of file Metric.h.

4.10.4.5 std::vector<float> MetricPreparation::rotation

Definition at line 165 of file Metric.h.

4.10.4.6 std::vector<std::vector<float> > MetricPreparation::rotationSequence

Definition at line 166 of file Metric.h.

4.10.4.7 int MetricPreparation::row

Definition at line 172 of file Metric.h.

4.10.4.8 std::vector<VectorXf > MetricPreparation::unitLength

Definition at line 168 of file Metric.h.

The documentation for this struct was generated from the following file:

- Metric.h

4.11 MultiVariate Struct Reference

```
#include <PreComputing.h>
```

Public Member Functions

- MultiVariate ()
- ∼MultiVariate ()

Public Attributes

- Matrix3f covariance
- Vector3f meanVec

4.11.1 Detailed Description

Definition at line 32 of file PreComputing.h.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 MultiVariate::MultiVariate ( ) [inline]

Definition at line 36 of file PreComputing.h.

```
{}
```

4.11.2.2 MultiVariate::∼MultiVariate ( ) [inline]

Definition at line 38 of file PreComputing.h.

```
{}
```

4.11.3 Member Data Documentation

4.11.3.1 Matrix3f MultiVariate::covariance

Definition at line 34 of file PreComputing.h.

4.11.3.2 Vector3f MultiVariate::meanVec

Definition at line 35 of file PreComputing.h.

The documentation for this struct was generated from the following file:

- PreComputing.h

4.12 Silhouette Class Reference

```
#include <Silhouette.h>
```

Public Member Functions

- Silhouette ()
- ∼Silhouette ()
- void computeValue (const int &normOption, const MatrixXf &array, const int &Row, const int &Column, const std::vector< int > &group, const MetricPreparation &object, const int &groupNumber, const bool &isPBF)
- void computeValue (const int &normOption, const MatrixXf &array, const int &Row, const int &Column, const std::vector< int > &group, const MetricPreparation &object, const int &groupNumber, const bool &isPBF, const std::vector< vector< int > > &storage)
- void computeValue (const Eigen::MatrixXf &cArray, const std::vector< int > &group, const int &groupNo, const bool &isPBF)
- void reset ()

Public Attributes

- std::vector< float > sData
- std::vector< float > sCluster
- float sAverage
- float dbIndex
- float gammaStatistic = -1.0

Private Member Functions

- const float getDist (const int &first, const int &second, const MetricPreparation &object, const MatrixXf &array, const int &normOption)
- const float getA_i (const std::vector< std::vector< int > > &storage, const std::vector< int > &group, const MatrixXf &array, const int &index, const MetricPreparation &object, const int &normOption)
- const float getB_i (const std::vector< std::vector< int > > &storage, const std::vector< int > &group, const MatrixXf &array, const int &index, const MetricPreparation &object, const int &normOption)
- void getMatrixM (const Eigen::MatrixXf &cArray, const std::vector< int > &group, const std::vector< std↩ ::vector< int > > &storage, Eigen::MatrixXf &distM, Eigen::MatrixXf &idealDistM)
- void getMatrixM (const Eigen::MatrixXf &cArray, const std::vector< int > &group, const std::vector< std↩ ::vector< int > > &storage, Eigen::MatrixXf &idealDistM)
- const float getA_i (const std::vector< std::vector< int > > &storage, const std::vector< int > &group, const Eigen::MatrixXf &array, const int &index, const bool &isPBF, const Eigen::MatrixXf &distM)
- const float getB_i (const std::vector< std::vector< int > > &storage, const std::vector< int > &group, const Eigen::MatrixXf &array, const int &index, const bool &isPBF, const Eigen::MatrixXf &distM)
- void computeSilhouette (const Eigen::MatrixXf &array, const std::vector< int > &group, const bool &isPBF, const std::vector< std::vector< int > > &storage, const Eigen::MatrixXf &distM)
- void computeSilhouette (const Eigen::MatrixXf &array, const std::vector< int > &group, const std::vector< std::vector< int > > &storage, const MetricPreparation &object, const int &normOption)
- void computeDBIndex (const Eigen::MatrixXf &array, const std::vector< int > &group, const std::vector< std::vector< int > > &storage)
- void computeDBIndex (const Eigen::MatrixXf &array, const std::vector< int > &group, const std::vector< std::vector< int > > &storage, const MetricPreparation &object, const int &normOption)
- void computeGammaStatistic (const Eigen::MatrixXf &distM, const Eigen::MatrixXf &idealDistM)
- void computeGammaStatistic (const Eigen::MatrixXf &idealDistM)

4.12.1 Detailed Description

Definition at line 16 of file Silhouette.h.

4.12.2 Constructor & Destructor Documentation

4.12.2.1 Silhouette::Silhouette ( )

Definition at line 12 of file Silhouette.cpp.

```
{
    sData = std::vector<float>();
    sCluster = std::vector<float>();
}
```

4.12.2.2 Silhouette::∼Silhouette ( )

Definition at line 22 of file Silhouette.cpp.

```
{
    reset();
}
```

4.12.3 Member Function Documentation

4.12.3.1 void Silhouette::computeDBIndex (const Eigen::MatrixXf & *array*, const std::vector< int > & *group*, const std::vector< std::vector< int > > & *storage*) [private]

Definition at line 664 of file Silhouette.cpp.

```
{
    dbIndex = 0.0;

    const int& groupNumber = storage.size();

    const int& Column = array.cols();

    /* calculated the projected-space cenroid */
    Eigen::MatrixXf centroid(groupNumber, Column);

    /* average distance of all elements in cluster to its centroid */
    Eigen::VectorXf averageDist(groupNumber);

#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<groupNumber;++i)
    {
        Eigen::VectorXf tempCentroid = Eigen::VectorXf::Zero(Column);

        const std::vector<int>& clusterVec = storage[i];
        const int& clusterSize = clusterVec.size();

        for(int j=0;j<clusterSize;++j)
            tempCentroid+=array.row(clusterVec[j]);

        /* get the centroid coordinates */
        centroid.row(i) = tempCentroid/clusterSize;

        float inClusterSum = 0.0, temp_dist;
        for(int j=0;j<clusterSize;++j)
        {
            //inClusterSum+=getDisimilarity(centroid.row(i),array,clusterVec[j],normOption,object);
            inClusterSum+=(array.row(clusterVec[j])-centroid.row(i)).norm();
        }
        averageDist(i) = inClusterSum/clusterSize;
    }

#pragma omp parallel num_threads(8)
    {
    #pragma omp for nowait
        for (int i = 0; i < groupNumber; ++i)
        {
            float maxValue = (float)INT_MIN, ratioDist;
            for (int j=0;j<groupNumber;++j)
            {
                if(i==j)
                    continue;
                ratioDist = (averageDist(i)+averageDist(j))/(centroid.row(i)-centroid.row(j)).norm();

                if(maxValue<ratioDist)
                    maxValue=ratioDist;
            }

        #pragma omp critical
            dbIndex += maxValue;
        }
    }
    dbIndex/=groupNumber;
}
```

4.12.3.2 void Silhouette::computeDBIndex (const Eigen::MatrixXf & *array*, const std::vector< int > & *group*, const std::vector< std::vector< int > > & *storage*, const MetricPreparation & *object*, const int & *normOption*) [private]

Definition at line 736 of file Silhouette.cpp.

```
{
    dbIndex = 0.0;

    const int& groupNumber = storage.size();

    const int& Column = array.cols();

    /* calculated the projected-space cenroid */
    Eigen::MatrixXf centroid(groupNumber, Column);

    /* average distance of all elements in cluster to its centroid */
    Eigen::VectorXf averageDist(groupNumber);

#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<groupNumber;++i)
    {
        Eigen::VectorXf tempCentroid = Eigen::VectorXf::Zero(Column);

        const std::vector<int>& clusterVec = storage[i];
        const int& clusterSize = clusterVec.size();

        for(int j=0;j<clusterSize;++j)
            tempCentroid+=array.row(clusterVec[j]);

        /* get the centroid coordinates */
        centroid.row(i) = tempCentroid/clusterSize;

        float inClusterSum = 0.0, temp_dist;
        for(int j=0;j<clusterSize;++j)
        {
            inClusterSum+=getDisimilarity(centroid.row(i),array,clusterVec[j],normOption,
  object);
        }
        averageDist(i) = inClusterSum/clusterSize;
    }

#pragma omp parallel num_threads(8)
    {
    #pragma omp for nowait
        for (int i = 0; i < groupNumber; ++i)
        {
            float maxValue = (float)INT_MIN, ratioDist, centDist;
            for (int j=0;j<groupNumber;++j)
            {
                if(i==j)
```

```
                    continue;
                centDist = getDisimilarity(centroid.row(i), centroid.row(j), normOption,
  object);

                //ratioDist = (averageDist(i)+averageDist(j))/(centroid.row(i)-centroid.row(j)).norm();
                ratioDist = (averageDist(i)+averageDist(j))/centDist;

                if(maxValue<ratioDist)
                    maxValue=ratioDist;
            }

        #pragma omp critical
            dbIndex += maxValue;
        }
    }
    dbIndex/=groupNumber;
}
```

4.12.3.3 void Silhouette::computeGammaStatistic (const Eigen::MatrixXf & *distM,* const Eigen::MatrixXf & *idealDistM*) [private]

Definition at line 809 of file Silhouette.cpp.

```
{
    const int& Row = distM.rows();

    const int& totalNum = Row*(Row-1)/2;
    /* mean of values */
    double u_1 = 0.0, u_2 = 0.0;

    /* E(X*X) */
    double s_1 = 0.0, s_2 = 0.0, numerator = 0.0;

    for(int i=0;i<Row-1;++i)
    {
        for(int j=i+1;j<Row;++j)
        {
            /* update the mean u_1, u_2 */
            u_1+=distM(i,j);
            u_2+=idealDistM(i,j);

            /* update the numerator */
            numerator+=distM(i,j)*idealDistM(i,j);

            /* update the deviation */
            s_1+=distM(i,j)*distM(i,j);
            s_2+=idealDistM(i,j)*idealDistM(i,j);
        }
    }

    /* get mean of distM and idealDistM */
    u_1/=totalNum;
    u_2/=totalNum;

    /* get numerator for the computing */
    numerator-=totalNum*u_1*u_2;

    /* get standard deviation */
    s_1=sqrt(s_1/totalNum-u_1*u_1);
    s_2=sqrt(s_2/totalNum-u_2*u_2);

    if(std::isnan(s_1) || std::isnan(s_2))
    {
        std::cout << "standard deviation has nan error!" << std::endl;
        exit(1);
    }
    gammaStatistic = float(numerator/s_1/s_2/totalNum);
}
```

4.12.3.4 void Silhouette::computeGammaStatistic (const Eigen::MatrixXf & *idealDistM*) [private]

Definition at line 863 of file Silhouette.cpp.

```
864 {
865     const int& Row = idealDistM.rows();
866 
867     const int& totalNum = Row*(Row-1)/2;
868 
869     /* mean of values */
870     double u_1 = 0.0, u_2 = 0.0;
871 
872     /* E(X*X) */
873     double s_1 = 0.0, s_2 = 0.0, numerator = 0.0;
874 
875     for(int i=0;i<Row-1;++i)
876     {
877         for(int j=i+1;j<Row;++j)
878         {
879             u_1+=distanceMatrix[i][j];
880             u_2+=idealDistM(i,j);
881 
882             numerator+=distanceMatrix[i][j]*idealDistM(i,j);
883 
884             s_1+=distanceMatrix[i][j]*distanceMatrix[i][j];
885             s_2+=idealDistM(i,j)*idealDistM(i,j);
886         }
887     }
888 
889     /* get mean of distM and idealDistM */
890     u_1/=totalNum;
891     u_2/=totalNum;
892 
893     /* get numerator for the computing */
894     numerator-=totalNum*u_1*u_2;
895 
896     /* get standard deviation */
897     s_1=sqrt(s_1/totalNum-u_1*u_1);
898     s_2=sqrt(s_2/totalNum-u_2*u_2);
899 
900     if(std::isnan(s_1) || std::isnan(s_2))
901     {
902         std::cout << "standard deviation has nan error!" << std::endl;
903         exit(1);
904     }
905     gammaStatistic = float(numerator/s_1/s_2/totalNum);
906 }
```

4.12.3.5 void Silhouette::computeSilhouette (const Eigen::MatrixXf & *array*, const std::vector< int > & *group*, const bool & *isPBF*, const std::vector< std::vector< int > > & *storage*, const Eigen::MatrixXf & *distM*) [private]

Definition at line 540 of file Silhouette.cpp.

```
545 {
546     const int& Row = array.rows();
547 
548 // compute Silhouette value for each data
549     float sSummation = 0;
550 
551 #pragma omp parallel num_threads(8)
552     {
553     #pragma omp for nowait
554         for (int i = 0; i < Row; ++i)
555         {
556             const float& a_i = getA_i(storage, group, array, i, isPBF, distM);
557             const float& b_i = getB_i(storage, group, array, i, isPBF, distM);
558 
559             float s_i;
560             if(abs(a_i-b_i)<1.0e-8)
561                 s_i = 0;
562             else if(a_i<b_i)
563                 s_i = 1 - a_i/b_i;
564             else
565                 s_i = b_i/a_i - 1;
566             if(std::isnan(s_i))
567             {
568                 std::cout << "Error for nan number!" << std::endl;
569                 exit(1);
570             }
571             sData[i] = s_i;
572 
573         #pragma omp critical
574             sSummation += s_i;
```

```
        }
    }
    sAverage = sSummation/group.size();

#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < sCluster.size(); ++i)
    {
        float& eachCluster = sCluster[i];
        eachCluster = 0;
        const std::vector<int>& clustSet = storage[i];
        for (int j = 0; j < clustSet.size(); ++j)
        {
            eachCluster += sData[clustSet[j]];
        }
        eachCluster/=clustSet.size();
    }
}
```

4.12.3.6 void Silhouette::computeSilhouette (const Eigen::MatrixXf & *array*, const std::vector< int > & *group*, const std::vector< std::vector< int > > & *storage*, const **MetricPreparation** & *object*, const int & *normOption*) [private]

Definition at line 603 of file Silhouette.cpp.

```
{
    // compute Silhouette value for each data
    float sSummation = 0;

#pragma omp parallel num_threads(8)
    {
    #pragma omp for nowait
        for (int i = 0; i < group.size(); ++i)
        {
            if(group[i]<0)
                continue;
            const float& a_i = getA_i(storage, group, array, i, object, normOption);
            const float& b_i = getB_i(storage, group, array, i, object, normOption);

            float s_i;
            if(abs(a_i-b_i)<1.0e-8)
                s_i = 0;
            else if(a_i<b_i)
                s_i = 1 - a_i/b_i;
            else
                s_i = b_i/a_i - 1;
            if(std::isnan(s_i))
            {
                std::cout << "Error for nan number!" << std::endl;
                exit(1);
            }
            sData[i] = s_i;

        #pragma omp critical
            sSummation += s_i;
        }
    }
    sAverage = sSummation/(group.size());

#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < sCluster.size(); ++i)
    {
        float& eachCluster = sCluster[i];
        eachCluster = 0;
        const std::vector<int>& clustSet = storage[i];
        for (int j = 0; j < clustSet.size(); ++j)
        {
            eachCluster += sData[clustSet[j]];
        }
        eachCluster/=clustSet.size();
    }
}
```

4.12.3.7 void Silhouette::computeValue (const int & *normOption,* const MatrixXf & *array,* const int & *Row,* const int & *Column,* const std::vector< int > & *group,* const **MetricPreparation** & *object,* const int & *groupNumber,* const bool & *isPBF*)

Definition at line 51 of file Silhouette.cpp.

```
{

    std::vector<std::vector<int> > storage(groupNumber, std::vector<int>());

    //whether some group marked as -1 noise or not
    int noise = 0;
    for (int i = 0; i < group.size(); ++i)
    {
        if(group[i]<0)
        {
            ++noise;
            continue;
        }
        else
            storage[group[i]].push_back(i);
    }
    // compute the value
    computeValue(normOption, array, Row, Column, group, object, groupNumber, isPBF, storage);

}
```

4.12.3.8 void Silhouette::computeValue (const int & *normOption,* const MatrixXf & *array,* const int & *Row,* const int & *Column,* const std::vector< int > & *group,* const **MetricPreparation** & *object,* const int & *groupNumber,* const bool & *isPBF,* const std::vector< vector< int > > & *storage*)

Definition at line 161 of file Silhouette.cpp.

```
{
    sData.clear();
    sCluster.clear();
    sData = std::vector<float>(Row,0);
    assert(Row==group.size());

    //groupNumber doesn't include noise group
    sCluster = std::vector<float>(groupNumber, 0);
    /* if the silhouett computing is not for PBF dataset, then would use distanceMatrix */
    Eigen::MatrixXf idealDistM;
    if(!isPBF)
    {
        getMatrixM(array,group,storage,idealDistM);
    }

    std::cout << "Compute silhouette..." << std::endl;
    /* compute silhouette value */
    computeSilhouette(array, group, storage, object, normOption);
    std::cout << "Silhouette is " << sAverage << std::endl;

    std::cout << "Compute DB index..." << std::endl;
    /* compute DB index */
    computeDBIndex(array, group, storage, object, normOption);
    std::cout << "DB index is " << dbIndex << std::endl;

    /* compute Gamma statistic for distM and idealDistM */
    if(!isPBF)
    {
        std::cout << "Compute gamma statistics..." << std::endl;
        computeGammaStatistic(idealDistM);
        std::cout << "Gamma statistics is " << gammaStatistic << std::endl;
        /* garbage collection for eigen::matrix */
        idealDistM.resize(0,0);
    }
}
```

4.12.3.9 void Silhouette::computeValue (const Eigen::MatrixXf & *cArray,* const std::vector< int > & *group,* const int & *groupNo,* const bool & *isPBF*)

Definition at line 89 of file Silhouette.cpp.

```
{
    sData.clear();
    sCluster.clear();

    /* get Row and Column information */
    const int& Row = array.rows();
    const int& Column = array.cols();

    sData = std::vector<float>(Row,0);

    /* assert information */
    assert(Row==group.size());

    std::vector<std::vector<int> > storage(groupNumber, std::vector<int>());
    for (int i = 0; i < group.size(); ++i)
    {
        storage[group[i]].push_back(i);
    }
    /* record labeling information */
    generateGroups(storage);

    //groupNumber doesn't include noise group
    sCluster = std::vector<float>(groupNumber, 0);

    /* if the silhouett computing is not for PBF dataset, then would use distanceMatrix */
    Eigen::MatrixXf distM, idealDistM;
    if(!isPBF)  // not from PBF, so the distance matrix can be assigned
    {
        getMatrixM(array,group,storage,distM,idealDistM);
    }

    std::cout << "Compute silhouette..." << std::endl;
    /* compute silhouette value */
    computeSilhouette(array, group, isPBF, storage, distM);

    std::cout << "silhouette is " << sAverage << std::endl;

    std::cout << "Compute DB index..." << std::endl;
    /* compute DB index */
    computeDBIndex(array, group, storage);
    std::cout << "DB index is " << dbIndex << std::endl;
    /* compute Gamma statistic for distM and idealDistM */
    if(!isPBF)  // only compute Gamma statistics for non-PBF data set
    {
        std::cout << "Compute gamma statistics..." << std::endl;
        computeGammaStatistic(distM,idealDistM);
        std::cout << "Gamma statistics is " << gammaStatistic << std::endl;
        /* garbage collection for eigen::matrix */
        distM.resize(0,0);
        idealDistM.resize(0,0);
    }

}
```

4.12.3.10 const float Silhouette::getA_i (const std::vector< std::vector< int > > & *storage,* const std::vector< int > & *group,* const MatrixXf & *array,* const int & *index,* const MetricPreparation & *object,* const int & *normOption*) [private]

Definition at line 217 of file Silhouette.cpp.

```
{
    const std::vector<int>& clusterSet = storage[group[index]];
    float inClusterDist = 0.0, dist;
    for (int j = 0; j < clusterSet.size(); ++j)
    {
        if(clusterSet[j]!=index)
        {
            if(distanceMatrix)
            {
                dist = distanceMatrix[index][clusterSet[j]];
            }
            else
            {
                dist = getDist(index, clusterSet[j], object, array, normOption);
            }
            inClusterDist += dist;
        }
    }
    if(std::isnan(inClusterDist))
    {
        std::cout << "a_i has nan error! " << inClusterDist << std::endl;
        exit(1);
    }
    float a_i;
    if(clusterSet.size()==1)
        a_i = 0;
    else
        a_i = inClusterDist/(clusterSet.size()-1);
    return a_i;
}
```

4.12.3.11 const float Silhouette::getA_i (const std::vector< std::vector< int > > & *storage*, const std::vector< int > & *group*, const Eigen::MatrixXf & *array*, const int & *index*, const bool & *isPBF*, const Eigen::MatrixXf & *distM*) [private]

Definition at line 435 of file Silhouette.cpp.

```
{
    const std::vector<int>& clusterSet = storage[group[index]];
    float inClusterDist = 0.0;

    int candidate;
    for (int j = 0; j < clusterSet.size(); ++j)
    {
        candidate = clusterSet[j];
        if(candidate!=index)
        {
            if(!isPBF)
                inClusterDist += distM(index,candidate);
            else
                inClusterDist += (array.row(index)-array.row(candidate)).norm();
        }
    }
    if(std::isnan(inClusterDist))
    {
        std::cout << "a_i has nan error!" << std::endl;
        exit(1);
    }
    float a_i;
    if(clusterSet.size()==1)
        a_i = 0;
    else
        a_i = inClusterDist/(clusterSet.size()-1);
    return a_i;
}
```

4.12.3.12 const float Silhouette::getB_i (const std::vector< std::vector< int > > & *storage*, const std::vector< int > & *group*, const MatrixXf & *array*, const int & *index*, const MetricPreparation & *object*, const int & *normOption*) [private]

Definition at line 265 of file Silhouette.cpp.

```
{
    float outClusterDist = FLT_MAX, perClusterDist = 0;
    std::vector<int> outClusterSet;
    if(storage.size()==1)
        return 0;
    for (int j = 0; j < storage.size(); ++j) //j is group no.
    {
        if(j!=group[index]) //the other cluster
        {
            outClusterSet = storage[j];//get integer list of this group
            perClusterDist = 0;
            for (int k = 0; k < outClusterSet.size(); ++k)
            {
                if(distanceMatrix)
                    perClusterDist+=distanceMatrix[index][outClusterSet[k]];
                else
                    perClusterDist += getDist(index, outClusterSet[k], object, array, normOption);
            }
            if(perClusterDist<0)
            {
                std::cout << "Error for negative distance!" << std::endl;
                exit(1);
            }
            perClusterDist/=outClusterSet.size();
            if(outClusterDist>perClusterDist)
                outClusterDist=perClusterDist;
        }
    }
    return outClusterDist;
}
```

4.12.3.13 const float Silhouette::getB_i (const std::vector< std::vector< int > > & *storage*, const std::vector< int > & *group*, const Eigen::MatrixXf & *array*, const int & *index*, const bool & *isPBF*, const Eigen::MatrixXf & *distM*) [private]

Definition at line 481 of file Silhouette.cpp.

```
{
    float outClusterDist = FLT_MAX, perClusterDist = 0;
    std::vector<int> outClusterSet;

    int candidate, outClusterSize;
    for (int j = 0; j < storage.size(); ++j) //j is group no.
    {
        if(j!=group[index]) //the other cluster
        {
            outClusterSet = storage[j];//get integer list of this group
            perClusterDist = 0;

            outClusterSize = outClusterSet.size();

            /* empty cluster which is erroneous */
            if(outClusterSize==0)
            {
                std::cout << "Found empty clusters!" << std::endl;
                exit(1);
            }

            /* get average dist to all elements inside the cluster */
            for (int k = 0; k < outClusterSize; ++k)
            {
                candidate = outClusterSet[k];
                if(!isPBF)
                    perClusterDist+=distM(index, candidate);
                else
                    perClusterDist += (array.row(index)-array.row(candidate)).norm();
            }
            if(perClusterDist<0)
            {
                std::cout << "Error for negative distance!" << std::endl;
                exit(1);
            }
            perClusterDist/=outClusterSize;
            if(outClusterDist>perClusterDist)
                outClusterDist=perClusterDist;
        }
    }
    return outClusterDist;
}
```

4.12.3.14 const float Silhouette::getDist (const int & *first*, const int & *second*, const MetricPreparation & *object*, const MatrixXf & *array*, const int & *normOption*) [private]

Definition at line 312 of file Silhouette.cpp.

```
{
    float distance = getDisimilarity(array.row(first),array.row(second),
                                     first,second,normOption,object);
    if(distance<0)
    {
        std::cout << "Error for negative distance!" << std::endl;
        exit(1);
    }
    if(isnan(distance) || isinf(distance))
    {
        std::cout << "Error for distance value that is nan or inf!" << std::endl;
        exit(1);
    }
    return distance;
}
```

4.12.3.15 void Silhouette::getMatrixM (const Eigen::MatrixXf & *cArray*, const std::vector< int > & *group*, const std::vector< std::vector< int > > & *storage*, Eigen::MatrixXf & *distM*, Eigen::MatrixXf & *idealDistM*) [private]

Definition at line 343 of file Silhouette.cpp.

```
{
    const int& Row = cArray.rows();
    const int& Column = cArray.cols();

    /* resize matrix size */
    distM = Eigen::MatrixXf::Zero(Row,Row);
    idealDistM = Eigen::MatrixXf::Constant(Row,Row,1.0);

    /* of course here is not related to distanceMatrix which is a global variable */
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<Row;++i)
    {
        for(int j=0;j<Row;++j)
        {
            if(i==j)
                continue;
            /* assign the Euclidean distance of cArray */
            distM(i,j)=(cArray.row(i)-cArray.row(j)).norm();
        }
    }

    const int& groupNumber = storage.size();
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<groupNumber;++i)
    {
        /* if i and j in same cluster, then set it to be zero */
        const std::vector<int>& eachVec = storage[i];
        const int& eachSize = eachVec.size();
        for(int j=0;j<eachSize;++j)
        {
            for(int k=0;k<eachSize;++k)
            {
                idealDistM(eachVec[j],eachVec[k]) = 0;
            }
        }
    }
}
```

4.12.3.16 void Silhouette::getMatrixM (const Eigen::MatrixXf & *cArray*, const std::vector< int > & *group*, const std::vector< std::vector< int > > & *storage*, Eigen::MatrixXf & *idealDistM*) [private]

Definition at line 395 of file Silhouette.cpp.

```
{
    const int& Row = cArray.rows();
    const int& Column = cArray.cols();

    /* resize matrix size */
    idealDistM = Eigen::MatrixXf::Constant(Row,Row,1.0);

    /* find the ideal matrix inside which the idealDistM(i,j)==0 only if i and j in same cluster */
    const int& groupNumber = storage.size();
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<groupNumber;++i)
    {
        /* if i and j in same cluster, then set it to be zero */
        const std::vector<int>& eachVec = storage[i];
        const int& eachSize = eachVec.size();
        for(int j=0;j<eachSize;++j)
        {
            for(int k=0;k<eachSize;++k)
            {
                idealDistM(eachVec[j],eachVec[k]) = 0;
            }
        }
    }
}
```

4.12.3.17 void Silhouette::reset ( )

Definition at line 31 of file Silhouette.cpp.

```
{
    sData.clear();
    sCluster.clear();
    sAverage = 0;
}
```

4.12.4 Member Data Documentation

4.12.4.1 float Silhouette::dbIndex

Definition at line 38 of file Silhouette.h.

4.12.4.2 float Silhouette::gammaStatistic = -1.0

Definition at line 43 of file Silhouette.h.

4.12.4.3 float Silhouette::sAverage

Definition at line 33 of file Silhouette.h.

4.12.4.4 std::vector<float> Silhouette::sCluster

Definition at line 28 of file Silhouette.h.

4.12.4.5 std::vector< float > Silhouette::sData

Definition at line 23 of file Silhouette.h.

The documentation for this class was generated from the following files:

- Silhouette.h
- Silhouette.cpp

4.13 StringQuery Struct Reference

```
#include <IOHandler.h>
```

Public Member Functions

- StringQuery ()
- StringQuery (const int &index, const std::vector< int > &neighbor)

Public Attributes

- int index
- std::vector< int > neighbor

4.13.1 Detailed Description

Definition at line 65 of file IOHandler.h.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 StringQuery::StringQuery ( ) [inline]

Definition at line 69 of file IOHandler.h.

```
    { }
```

4.13.2.2 StringQuery::StringQuery (const int & *index*, const std::vector< int > & *neighbor*) [inline]

Definition at line 71 of file IOHandler.h.

```
                                                    :
                index(index), neighbor(neighbor)
    { }
```

4.13.3 Member Data Documentation

4.13.3.1 int StringQuery::index

Definition at line 67 of file IOHandler.h.

4.13.3.2 std::vector< int > StringQuery::neighbor

Definition at line 68 of file IOHandler.h.

The documentation for this struct was generated from the following file:

- IOHandler.h

4.14 ValidityMeasurement Class Reference

```
#include <ValidityMeasurement.h>
```

Public Member Functions

- ValidityMeasurement ()
- virtual ∼ValidityMeasurement ()
- void computeValue (const int &normOption, const MatrixXf &array, const std::vector< int > &group, const MetricPreparation &object, const bool &isPBF)
- void computeValue (const MatrixXf &array, const std::vector< int > &group)

Public Attributes

- float f_c

Private Member Functions

- void getMST_Parent_Node (std::tuple< float, float, float > &values, const std::vector< int > &clusterNode, const MetricPreparation &object, const int &normOption, const MatrixXf &array, const bool &isPBF)
- void getMST_Parent_Node (std::tuple< float, float, float > &values, const std::vector< int > &clusterNode, const MatrixXf &array)
- const float get_Sc_by_range (const bool &isPBF, const Eigen::MatrixXf &distM, const std::vector< int > &clusterNode, const float &rangeValue, const MetricPreparation &object, const int &normOption, const MatrixXf &array, int &index)
- const float get_Sc_by_range (const Eigen::MatrixXf &distM, const std::vector< int > &clusterNode, const float &rangeValue, const MatrixXf &array, int &index)

Private Attributes

- float min_Sc
- float max_Sc

4.14.1 Detailed Description

Definition at line 23 of file ValidityMeasurement.h.

4.14.2 Constructor & Destructor Documentation

4.14.2.1 ValidityMeasurement::ValidityMeasurement ( )

Definition at line 15 of file ValidityMeasurement.cpp.

```
                                     {
    // TODO Auto-generated constructor stub

}
```

4.14.2.2 ValidityMeasurement::∼ValidityMeasurement ( ) [virtual]

Definition at line 24 of file ValidityMeasurement.cpp.

```
                                      {
    // TODO Auto-generated destructor stub
}
```

4.14.3 Member Function Documentation

4.14.3.1 void ValidityMeasurement::computeValue (const int & *normOption,* const MatrixXf & *array,* const std::vector< int > & *group,* const MetricPreparation & *object,* const bool & *isPBF*)

Definition at line 38 of file ValidityMeasurement.cpp.

```
{
    std::cout << "Compute validity measurement..." << std::endl;
    // get how many different groups it totally has
    int max_group = -1;
    const int& num_node = group.size();
    for(int i=0; i<num_node; ++i)
    {
        if(group[i]==-1)
            continue;
        max_group = std::max(group[i], max_group);
    }
    max_group+=1;

    std::vector<std::vector<int> > storage(max_group);

    for(int i=0; i<num_node; ++i)
    {
        if(group[i]==-1)
            continue;
        storage[group[i]].push_back(i);
    }

    std::vector<std::tuple<float, float, float> > measureVec(max_group);

    for(int i=0; i<max_group; ++i)
    {
        getMST_Parent_Node(measureVec[i], storage[i], object, normOption, array, isPBF);
    }

    float minSc = 0, maxSc = 0, aver_sigma = 0, std_sigma = 0, std_variance;
    for(int i=0; i<max_group; ++i)
    {
        // get the min Sc by summation
        minSc+=std::get<1>(measureVec[i]);
        // get the max Sc by summation
        maxSc+=std::get<2>(measureVec[i]);
        std_variance = std::get<0>(measureVec[i]);
        // get the average variance and standard variation of variance
        aver_sigma+=std_variance;
        std_sigma+=std_variance*std_variance;
    }
    aver_sigma/=max_group;
    std_sigma = std_sigma/float(max_group-1)-float(max_group)/float(max_group-1)*aver_sigma*aver_sigma;

    if(std_sigma<1.0E-10)
    {
        std_sigma=1.0E-10;
    }
    std_sigma=sqrt(std_sigma);

    float h_DDc = aver_sigma+std_sigma;

    minSc/=float(max_group);
    maxSc/=float(max_group);

    // compute g1_Sc
    float g1_Sc = (1.0-minSc)*(1.0-maxSc);
    if(g1_Sc<0)
    {
        std::cout << "Negative number for g1_Sc computation!" << std::endl;
    }
    g1_Sc = aver_sigma*sqrt(g1_Sc);

    // compute g2_Sc
    float g2_Sc = minSc*maxSc;
    if(g2_Sc<0)
    {
        std::cout << "Negative number for g2_Sc computation!" << std::endl;
    }
    g2_Sc = aver_sigma/sqrt(g2_Sc);

    // compute g_Sc
    float g_Sc = (sqrt(g1_Sc*g2_Sc)+(g1_Sc+g2_Sc)/2.0)/2.0;

    // compoute f_c
    f_c = h_DDc*g_Sc;

    if(isnan(f_c) || isinf(f_c))
    {
        std::cout << "Error for f_c to have inf or nan values!" << std::endl;
    }

    /* normalization of validity measurement */
    float min_dist = FLT_MAX, max_dist = -1.0;
    const int& row = array.rows();
#pragma omp parallel for reduction(min:min_dist) num_threads(8)
    for(int i=0; i<row; ++i)
    {
        for(int j=0; j<row; ++j)
        {
            if(i==j)
                continue;
            float dist;
            if(distanceMatrix)
                dist = distanceMatrix[i][j];
            else
                dist = getDisimilarity(array, i, j, normOption, object);
            min_dist = std::min(min_dist, dist);
        }
    }

#pragma omp parallel for reduction(max:max_dist) num_threads(8)
    for(int i=0; i<row; ++i)
    {
        for(int j=0; j<row; ++j)
        {
            if(i==j)
                continue;
            float dist;
            if(distanceMatrix)
                dist = distanceMatrix[i][j];
            else
                dist = getDisimilarity(array, i, j, normOption, object);
            max_dist = std::max(max_dist, dist);
        }
    }
    std::cout << "min dist is " << min_dist << ", and max is " << max_dist << std::endl;
    f_c/=(max_dist-min_dist)*(max_dist-min_dist);

    // try to place the distance range into the file for further batch processing
    std::ofstream fout("../dataset/dist_range", ios::app);
    if(fout.fail())
    {
        std::cout << "Error for file operation!" << std::endl;
        exit(1);
    }

    fout << "For norm " << normOption << ", min is " << min_dist << ", max is " << max_dist << ", and" <<
            " (max - min) is " << (max_dist-min_dist) << std::endl;
    fout << std::endl;
    fout.close();

    std::cout << "Validity measurement is " << f_c << std::endl;
}
```

4.14.3.2 void ValidityMeasurement::computeValue (const MatrixXf & *array*, const std::vector< int > & *group*)

Definition at line 182 of file ValidityMeasurement.cpp.

```
{
    std::cout << "Compute validity measurement..." << std::endl;
    // get how many different groups it totally has
    int max_group = -1;
    const int& num_node = group.size();
    for(int i=0; i<num_node; ++i)
    {
        if(group[i]==-1)
            continue;
        max_group = std::max(group[i], max_group);
    }
    max_group+=1;

    std::vector<std::vector<int> > storage(max_group);
    for(int i=0; i<num_node; ++i)
    {
        if(group[i]==-1)
            continue;
        storage[group[i]].push_back(i);
    }

    std::vector<std::tuple<float, float, float> > measureVec(max_group);

    for(int i=0; i<max_group; ++i)
    {
        getMST_Parent_Node(measureVec[i], storage[i], array);
    }

    float minSc = 0, maxSc = 0, aver_sigma = 0, std_sigma = 0, std_variance;
    for(int i=0; i<max_group; ++i)
    {
        // get the min Sc by summation
        minSc+=std::get<1>(measureVec[i]);
        // get the max Sc by summation
        maxSc+=std::get<2>(measureVec[i]);
        std_variance = std::get<0>(measureVec[i]);
        // get the average variance and standard variation of variance
        aver_sigma+=std_variance;
        std_sigma+=std_variance*std_variance;
    }
    aver_sigma/=float(max_group);
    std_sigma = std_sigma/float(max_group-1)-float(max_group)/float(max_group-1)*aver_sigma*aver_sigma;
    if(std_sigma<1.0E-10)
    {
        std_sigma = 1.0E-10;
    }
    else
        std_sigma=sqrt(std_sigma);

    float h_DDc = aver_sigma+std_sigma;

    minSc/=float(max_group);
    maxSc/=float(max_group);

    // compute g1_Sc
    float g1_Sc = (1.0-minSc)*(1.0-maxSc);
    if(g1_Sc<0)
    {
        std::cout << "Negative number for g1_Sc computation!" << std::endl;
    }
    g1_Sc = aver_sigma*sqrt(g1_Sc);

    // compute g2_Sc
    float g2_Sc = minSc*maxSc;
    if(g2_Sc<0)
    {
        std::cout << "Negative number for g2_Sc computation!" << std::endl;
    }
    g2_Sc = aver_sigma/sqrt(g2_Sc);

    // compute g_Sc
    float g_Sc = (sqrt(g1_Sc*g2_Sc)+(g1_Sc+g2_Sc)/2.0)/2.0;

    // compoute f_c
    f_c = h_DDc*g_Sc;

    if(isnan(f_c) || isinf(f_c))
    {
        std::cout << "Error for f_c to have inf or nan values!" << std::endl;
    }

    /* normalization of validity measurement */
    float min_dist = FLT_MAX, max_dist = -1.0;
    const int& row = array.rows();
#pragma omp parallel for reduction(min:min_dist) num_threads(8)
    for(int i=0; i<row; ++i)
    {
        for(int j=0; j<row; ++j)
        {
            if(i==j)
                continue;
            min_dist = std::min(min_dist, (array.row(i)-array.row(j)).norm());
        }
    }

#pragma omp parallel for reduction(max:max_dist) num_threads(8)
    for(int i=0; i<row; ++i)
    {
        for(int j=0; j<row; ++j)
        {
            if(i==j)
                continue;
            max_dist = std::max(max_dist, (array.row(i)-array.row(j)).norm());
        }
    }
    std::cout << "min dist is " << min_dist << ", and max is " << max_dist << std::endl;
    f_c/=(max_dist-min_dist)*(max_dist-min_dist);

    // try to place the distance range into the file for further batch processing
    std::ofstream fout("../dataset/dist_range", ios::app);
    if(fout.fail())
    {
        std::cout << "Error for file operation!" << std::endl;
        exit(1);
    }

    fout << "For PCA, min is " << min_dist << ", max is " << max_dist << ", and" <<
            " (max - min) is " << (max_dist-min_dist) << std::endl;
    fout << std::endl;
    fout.close();

    std::cout << "Validity measurement is " << f_c << std::endl;
}
```

4.14.3.3 const float ValidityMeasurement::get_Sc_by_range (const bool & *isPBF,* const Eigen::MatrixXf & *distM,* const std::vector< int > & *clusterNode,* const float & *rangeValue,* const MetricPreparation & *object,* const int & *normOption,* const MatrixXf & *array,* int & *index*) [private]

Definition at line 570 of file ValidityMeasurement.cpp.

```
{
    const int& node_number = clusterNode.size();
    float result = 0.0;

    index = 0;
    int inside_whole, inside_cluster;
    for(int i=0; i<node_number; ++i)
    {
        inside_whole = 0, inside_cluster = 0;
        // count how many points in N_epsi(P_i) for the whole dataset
    #pragma omp parallel num_threads(8)
        {
        #pragma omp for nowait
            for(int j=0; j<array.rows(); ++j)
            {
                // don't want to handle duplicates and itself
                if(clusterNode[i]==j)
                    continue;
                float dist;
                if(distanceMatrix)
                    dist = distanceMatrix[clusterNode[i]][j];
                else
                    dist = getDisimilarity(array, clusterNode[i], j, normOption, object);

            #pragma omp critical
                {
                    if(dist<=rangeValue)
                    {
                        ++inside_whole;
                    }
                }
            }

        }

        // count how many points in N_epsi(P_i) for current cluster
    #pragma omp parallel num_threads(8)
        {
        #pragma omp for nowait
            for(int j=0; j<node_number; ++j)
            {
                // don't want to handle duplicates and itself
                if(i==j)
                    continue;
                float dist;
                if(isPBF)
                    dist = distM(i,j);
                else
                    dist = distanceMatrix[clusterNode[i]][clusterNode[j]];

            #pragma omp critical
                if(dist<=rangeValue)
                    ++inside_cluster;
            }

        }
        assert(inside_cluster<=inside_whole);
        if(inside_whole==0)
            continue;
        ++index;
        result+=float(inside_cluster)/float(inside_whole);
    }
    return result;
}
```

4.14.3.4 const float ValidityMeasurement::get_Sc_by_range (const Eigen::MatrixXf & *distM*, const std::vector< int > & *clusterNode*, const float & *rangeValue*, const MatrixXf & *array*, int & *index*) [private]

Definition at line 649 of file ValidityMeasurement.cpp.

```
{
    const int& node_number = clusterNode.size();
    float result = 0.0;

    index = 0;
    int inside_whole, inside_cluster;
    for(int i=0; i<node_number; ++i)
    {
        inside_whole = 0, inside_cluster = 0;
        // count how many points in N_epsi(P_i) for the whole dataset
    #pragma omp parallel num_threads(8)
        {
        #pragma omp for nowait
            for(int j=0; j<array.rows(); ++j)
            {
                // don't want to handle duplicates and itself
                if(i==j)
```

```
                    continue;
                float dist = (array.row(clusterNode[i])-array.row(j)).norm();

            #pragma omp critical
                if(dist<=rangeValue)
                    ++inside_whole;
            }

        }

        // count how many points in N_epsi(P_i) for current cluster
    #pragma omp parallel num_threads(8)
        {
        #pragma omp for nowait
            for(int j=0; j<node_number; ++j)
            {
                // don't want to handle duplicates and itself
                if(i==j)
                    continue;
                float dist = distM(i,j);

            #pragma omp critical
                if(dist<=rangeValue)
                    ++inside_cluster;
            }

        }
        assert(inside_cluster<=inside_whole);
        if(inside_whole==0)
            continue;
        result+=float(inside_cluster)/float(inside_whole);
        ++index;
        assert(!std::isnan(result));
    }
    return result;
}
```

4.14.3.5 void ValidityMeasurement::getMST_Parent_Node (std::tuple< float, float, float > & *values*, const std::vector< int > & *clusterNode*, const MetricPreparation & *object*, const int & *normOption*, const MatrixXf & *array*, const bool & *isPBF*) [private]

Definition at line 318 of file ValidityMeasurement.cpp.

```
{
    using namespace boost;
    typedef adjacency_list < vecS, vecS, undirectedS, no_property, property < edge_weight_t, float > > Graph;
    typedef graph_traits < Graph >::edge_descriptor Edge;
    typedef graph_traits < Graph >::vertex_descriptor Vertex;
    typedef std::pair<int, int> E;

    const int& num_nodes = clusterNode.size();

    if(num_nodes<=1)
    {
        values = std::make_tuple(0.0,0.0,0.0);
        std::cout << "Find 1-candidate cluster!" << std::endl;
        return;
    }

    const int num_edges = num_nodes*(num_nodes-1)/2;
    Eigen::MatrixXf distM;
    // if distanceMatrix is not stored ahead of time
    if(isPBF)
        distM = Eigen::MatrixXf(num_nodes, num_nodes);
    // assign [source, destination] index pair and weight lists
    E *edge_array = new E[num_edges];
    float *weights = new float[num_edges], dist;
    int temp = 0;
    for(int i=0; i<num_nodes-1; ++i)
    {
        for(int j=i+1; j<num_nodes; ++j)
        {
            // assign index pair
            edge_array[temp] = std::make_pair(i, j);
            // assign weight list
            if(distanceMatrix)
                dist = distanceMatrix[clusterNode[i]][clusterNode[j]];
            else
```

```
                dist = getDisimilarity(array, clusterNode[i], clusterNode[j], normOption, object);

            if(isPBF)
            {
                distM(i,j) = dist;
                distM(j,i) = dist;
            }

            weights[temp] = dist;
            ++temp;
        }
    }

#if defined(BOOST_MSVC) && BOOST_MSVC <= 1300
    Graph g(num_nodes);
    property_map<Graph, edge_weight_t>::type weightmap = get(edge_weight, g);
    for (std::size_t j = 0; j < num_edges; ++j) {
        Edge e; bool inserted;
        tie(e, inserted) = add_edge(edge_array[j].first, edge_array[j].second, g);
        weightmap[e] = weights[j];
    }
#else
    Graph g(edge_array, edge_array + num_edges, weights, num_nodes);
#endif
    property_map < Graph, edge_weight_t >::type weight = get(edge_weight, g);
    std::vector < Edge > spanning_tree;
    kruskal_minimum_spanning_tree(g, std::back_inserter(spanning_tree));

    if(edge_array!=NULL)
    {
        delete[] edge_array;
        edge_array = NULL;
    }

    if(weights!=NULL)
    {
        delete[] weights;
        weights = NULL;
    }

    // compute the standard deviation for the distance in MST
    double summation = 0.0, sq_summation = 0.0, average_mst_d, max_d_mst = -1.0;
    const int& MST_EDGE_NUM = num_nodes-1;

    for (std::vector < Edge >::iterator ei = spanning_tree.begin(); ei != spanning_tree.end(); ++ei)
    {
            dist = weight[*ei];
            max_d_mst = std::max(double(dist), max_d_mst);
            summation+=dist;
            sq_summation+=dist*dist;
    }

    float variance;

    if(MST_EDGE_NUM<=1)
    {
        variance = 0;
        average_mst_d = summation;
    }
    else
    {
        average_mst_d = summation/float(MST_EDGE_NUM);
        variance = sq_summation/float(MST_EDGE_NUM-1)-average_mst_d*summation/float(MST_EDGE_NUM-1);

        if(variance<1.0E-10)
        {
            variance = 1.0E-10;
        }
        variance = sqrt(variance);
    }

    // compute the inner Sc value for this cluster
    int min_index, max_index;
    float min_Sc = get_Sc_by_range(isPBF, distM, clusterNode, max_d_mst, object, normOption, array, min_index);
    float max_Sc = get_Sc_by_range(isPBF, distM, clusterNode, average_mst_d, object, normOption, array, max_index);

    min_Sc/=float(min_index);
    max_Sc/=float(max_index);

    // store the standard deviation, min Sc and max Sc in the tuple
    values = std::make_tuple(variance, min_Sc, max_Sc);
}
```

4.14.3.6 void ValidityMeasurement::getMST_Parent_Node (std::tuple< float, float, float > & *values*, const std::vector< int > & *clusterNode*, const MatrixXf & *array*) [private]

Definition at line 447 of file ValidityMeasurement.cpp.

```
{
    using namespace boost;
    typedef adjacency_list < vecS, vecS, undirectedS, no_property, property < edge_weight_t, float > > Graph;
    typedef graph_traits < Graph >::edge_descriptor Edge;
    typedef graph_traits < Graph >::vertex_descriptor Vertex;
    typedef std::pair<int, int> E;
    // get number of points in one cluster
    const int& num_nodes = clusterNode.size();
    if(num_nodes<=1)
    {
        values = std::make_tuple(0.0,0.0,0.0);
        return;
    }
    const int num_edges = num_nodes*(num_nodes-1)/2;

    Eigen::MatrixXf distM = Eigen::MatrixXf(num_nodes, num_nodes);
    // assign [source, destination] index pair and weight lists
    E *edge_array = new E[num_edges];
    float *weights = new float[num_edges], dist;
    int temp = 0;
    for(int i=0; i<num_nodes-1; ++i)
    {
        for(int j=i+1; j<num_nodes; ++j)
        {
            // assign index pair
            edge_array[temp] = std::make_pair(i, j);

            dist = (array.row(clusterNode[i])-array.row(clusterNode[j])).norm();
            distM(i,j) = dist;
            distM(j,i) = dist;

            weights[temp] = dist;
            ++temp;
        }
    }


#if defined(BOOST_MSVC) && BOOST_MSVC <= 1300
    Graph g(num_nodes);
    property_map<Graph, edge_weight_t>::type weightmap = get(edge_weight, g);
    for (std::size_t j = 0; j < num_edges; ++j) {
        Edge e; bool inserted;
        tie(e, inserted) = add_edge(edge_array[j].first, edge_array[j].second, g);
        weightmap[e] = weights[j];
    }
#else
    Graph g(edge_array, edge_array + num_edges, weights, num_nodes);
#endif
    property_map < Graph, edge_weight_t >::type weight = get(edge_weight, g);
    std::vector < Edge > spanning_tree;

    kruskal_minimum_spanning_tree(g, std::back_inserter(spanning_tree));


    if(edge_array!=NULL)
    {
        delete[] edge_array;
        edge_array = NULL;
    }

    if(weights!=NULL)
    {
        delete[] weights;
        weights = NULL;
    }

    // compute the standard deviation for the distance in MST
    double summation = 0.0, sq_summation = 0.0, average_mst_d, max_d_mst = -1.0;
    const int& MST_EDGE_NUM = num_nodes-1;

    for (std::vector < Edge >::iterator ei = spanning_tree.begin(); ei != spanning_tree.end(); ++ei)
    {
        dist = weight[*ei];
        summation+=dist;
        sq_summation+=dist*dist;
        max_d_mst=std::max(max_d_mst, double(dist));
    }

    float variance;
    if(MST_EDGE_NUM<=1)
    {
        variance = 0;
        average_mst_d = summation;
    }
    else
    {
        average_mst_d=summation/float(MST_EDGE_NUM);
        variance = sq_summation/float(MST_EDGE_NUM-1)-average_mst_d*summation/float(MST_EDGE_NUM-1);

        if(variance<1.0E-10)
        {
            variance = 1.0E-10;
        }
        variance = sqrt(variance);
    }

    // compute the inner Sc value for this cluster
    int min_index, max_index;
    float min_Sc = get_Sc_by_range(distM, clusterNode, max_d_mst, array, min_index);
    float max_Sc = get_Sc_by_range(distM, clusterNode, average_mst_d, array, max_index
  );

    min_Sc/=float(min_index);
    max_Sc/=float(max_index);

    // store the standard deviation, min Sc and max Sc in the tuple
    values = std::make_tuple(variance, min_Sc, max_Sc);
}
```

4.14.4 Member Data Documentation

4.14.4.1 float ValidityMeasurement::f_c

Definition at line 29 of file ValidityMeasurement.h.

4.14.4.2 float ValidityMeasurement::max_Sc [private]

Definition at line 70 of file ValidityMeasurement.h.

4.14.4.3 float ValidityMeasurement::min_Sc [private]

Definition at line 70 of file ValidityMeasurement.h.

The documentation for this class was generated from the following files:

- ValidityMeasurement.h
- ValidityMeasurement.cpp

Chapter 5

File Documentation

5.1 CMakeLists.txt File Reference

Functions

- cmake_minimum_required (VERSION 2.6) set(component_SOURCES Distance.h Distance.cpp Initialization.↩ h Initialization.cpp IOHandler.h IOHandler.cpp Metric.h Metric.cpp PreComputing.h PreComputing.cpp Silhouette.h Silhouette.cpp ValidityMeasurement.h ValidityMeasurement.cpp DetermClusterNum.h Determ↩ ClusterNum.cpp) include(CheckCXXCompilerFlag) if(COMPILER_SUPPORTS_CXX11) set(CMAKE_CX↩ X_FLAGS"$

5.1.1 Function Documentation

5.1.1.1 cmake_minimum_required (VERSION 2. *6*)

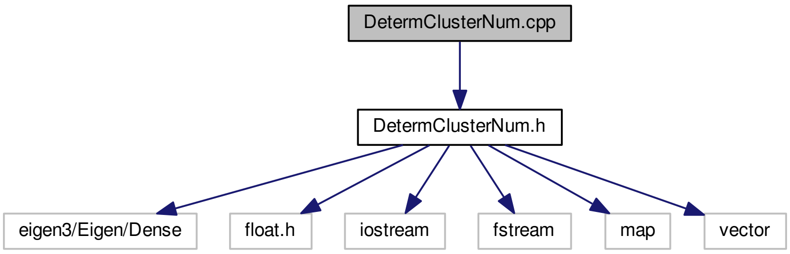
Definition at line 2 of file CMakeLists.txt.

```
27                              {CMAKE_CXX_FLAGS} -std=c++11")
```

5.2 DetermClusterNum.cpp File Reference

```
#include "DetermClusterNum.h"
```

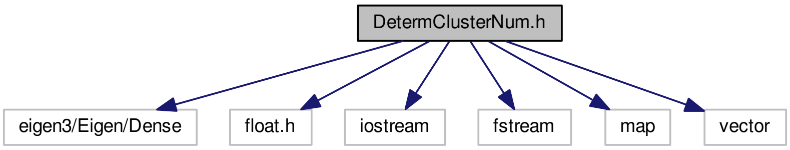
Include dependency graph for DetermClusterNum.cpp:



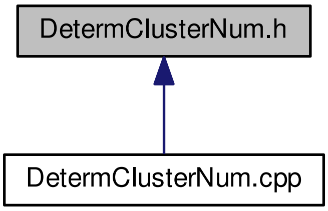
5.3 DetermClusterNum.h File Reference

```
#include <eigen3/Eigen/Dense>
#include <float.h>
#include <iostream>
#include <fstream>
#include <map>
#include <vector>
```

Include dependency graph for DetermClusterNum.h:



This graph shows which files directly or indirectly include this file:



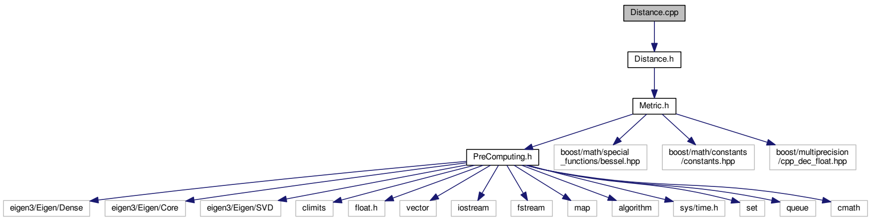
Classes

- class DetermClusterNum

5.4 Distance.cpp File Reference

```
#include "Distance.h"
```

Include dependency graph for Distance.cpp:



Functions

- const float getBMetric_3 (const VectorXf &row, const int &size, const int &i, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_3 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_3 (const int &first, const int &second, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_6 (const VectorXf &row, const int &size, const int &i, const std::vector< MultiVariate > &normalMultivariate)
- const float getBMetric_6 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_6 (const int &first, const int &second, const std::vector< MultiVariate > &normal↩ Multivariate)
- const float getBMetric_7 (const VectorXf &row, const int &size, const int &i, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_7 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_7 (const int &first, const int &second, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_9 (const VectorXf &row, const int &size, const int &i, const std::vector< MultiVariate > &normalMultivariate)
- const float getBMetric_9 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_9 (const int &first, const int &second, const std::vector< MultiVariate > &normal↩ Multivariate)
- const float getBMetric (const std::vector< float > &firstNorm3, const std::vector< float > &secondNorm3)
- const float getBMetric (const MultiVariate &centerNormal, const MultiVariate &neighNormal)
- const float getMetric_10 (const VectorXf &centroid, const int &size, const int &index, const std::vector< VectorXf > &unitLength)
- const float getMetric_10 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getMetric_10 (const int &first, const int &second, const std::vector< VectorXf > &unitLength)
- const float getNorm (const Eigen::VectorXf &centroid, const Eigen::VectorXf &r2, const int &index, const int &normOption, const std::vector< std::vector< float > > &pairwise, const std::vector< std::vector< float > > &objectNorm)
- const float getNorm (const VectorXf &centroid, const VectorXf &r2, const int &firstIndex, const int &second↩ Index, const int &normOption, const std::vector< std::vector< float > > &pairwise, const std::vector< std↩ ::vector< float > > &objectNorm)
- const float getNorm (const Eigen::VectorXf &r1, const Eigen::VectorXf &r2, const int &normOption)
- const float getDisimilarity (const MatrixXf &data, const int &first, const int &second, const int &normOption, const MetricPreparation &object)
- const float getDisimilarity (const VectorXf &others, const MatrixXf &data, const int &index, const int &norm↩ Option, const MetricPreparation &object)
- const float getDisimilarity (const VectorXf &first, const VectorXf &second, const int &firstIndex, const int &secondIndex, const int &normOption, const MetricPreparation &object)
- const float getDisimilarity (const VectorXf &first, const VectorXf &second, const int &normOption, const MetricPreparation &object)
- const float getMetric_MOP (const VectorXf &first, const VectorXf &second)
- const float getMetric_Hausdorff (const VectorXf &first, const VectorXf &second)
- void getDistanceMatrix (const MatrixXf &data, const int &normOption, const MetricPreparation &object)
- void deleteDistanceMatrix (const int &Row)
- const float getRotation (const std::vector< vector< float > > &streamline, std::vector< float > &rotation)
- const float getSignatureMetric (const Eigen::VectorXf &firstArray, const Eigen::VectorXf &secondArray, const std::vector< float > &firstHist, const std::vector< float > &secondHist)
- const float getSignatureMetric (const Eigen::VectorXf &centroid, const Eigen::VectorXf &first, const std↩ ::vector< float > &firstHist)
- const float getSignatureMetric (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getProcrustesMetric (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getProcrustesMetricSegment (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- void generateGroups (const std::vector< std::vector< int > > &storage)

- const float getEntropyMetric (const std::vector< float > &firstEntropy, const std::vector< float > &secondEntropy)
- const float getEntropyMetric (const std::vector< float > &firstEntropy, const Eigen::VectorXf &array)
- const float getEntropyMetric (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getPathline_MCP (const Eigen::VectorXf &first, const Eigen::VectorXf &second)

Variables

- const int & PROCRUSTES_SIZE = 7
- float ** distanceMatrix = NULL

5.4.1 Function Documentation

5.4.1.1 void deleteDistanceMatrix (const int & *Row*)

Definition at line 1382 of file Distance.cpp.

```
{
    if(distanceMatrix)
    {
    #pragma omp parallel for schedule(static) num_threads(8)
        for (int i = 0; i < Row; ++i)
        {
            if(distanceMatrix[i])
            {
                delete[] distanceMatrix[i];
                distanceMatrix[i] = NULL;
            }
        }
        delete[] distanceMatrix;
        distanceMatrix = NULL;
    }
}
```

5.4.1.2 void generateGroups (const std::vector< std::vector< int > > & *storage*)

Definition at line 1808 of file Distance.cpp.

```
{
    if(storage.empty())
        return;
    std::ofstream readme("../dataset/Storage",ios::out|ios::app);
    if(!readme)
    {
        std::cout << "Error creating Storage!" << std::endl;
        exit(1);
    }

    readme << std::endl;
    const int& groupSize = storage.size();
    std::vector<int> element;
    for(int i=0;i<groupSize;++i)
    {
        element = storage[i];
        if(element.empty())
            continue;
        for(int j=0;j<element.size();++j)
            readme << element[j] << " ";
        readme << std::endl;
    }
    std::cout << std::endl;
    readme.close();
}
```

5.4.1.3 const float getBMetric (const std::vector< float > & *firstNorm3*, const std::vector< float > & *secondNorm3*)

Definition at line 284 of file Distance.cpp.

```
{
    // calculate mean and standard deviation of the two arrays
    float u_a, u_b, sig_a, sig_b, sig_a_inverse, sig_b_inverse,
          summation, sum_inverse, tempDist;
    u_a = firstNorm3[0], u_b = secondNorm3[0];
    sig_a = firstNorm3[1], sig_b = secondNorm3[1];
    sig_a *= sig_a, sig_b *= sig_b;
    if(sig_a<=1.0e-8)
    {
        sig_a = 1.0e-8;
        sig_a_inverse = 1.0e8;
    }
    else
        sig_a_inverse = 1.0/sig_a;
    if(sig_b<=1.0e-8)
    {
        sig_b = 1.0e-8;
        sig_b_inverse = 1.0/sig_b;
    }
    summation = sig_a+sig_b;
    sum_inverse = 1.0/summation;
    tempDist = 0.25*log(0.25*(sig_a*sig_b_inverse
                +sig_b*sig_a_inverse+2))
                + 0.25*(u_a-u_b)*(u_a-u_b)*sum_inverse;
    return tempDist;
}
```

5.4.1.4 const float getBMetric (const MultiVariate & *centerNormal*, const MultiVariate & *neighNormal*)

Definition at line 321 of file Distance.cpp.

```
{
    Matrix3f firstCov, secondCov, meanCov, meanCovInverse;
    float sqrtInverse, meanCovDet;
    firstCov = centerNormal.covariance;
    secondCov = neighNormal.covariance;
    meanCov = 0.5*(firstCov+secondCov);
    if(meanCov.determinant()>1.0e-8)
    {
        meanCovInverse = static_cast<Matrix3f>(meanCov.inverse());
        meanCovDet = meanCov.determinant();
    }
    else
    {
        meanCovInverse = pseudoInverse(meanCov);
        meanCovDet = 1.0e8;
    }
    float detMulti = sqrt(firstCov.determinant()*secondCov.determinant());
    sqrtInverse = detMulti>1.0e-8?float(1.0)/detMulti:1.0e8;
    Vector3f meanDiff = centerNormal.meanVec-neighNormal.meanVec;
    float tempDist = 0.125*meanDiff.transpose()*meanCovInverse*meanDiff
                +0.2*log(meanCovDet*sqrtInverse);
    return tempDist;
}
```

5.4.1.5 const float getBMetric_3 (const VectorXf & *row*, const int & *size*, const int & *i*, const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 51 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getSequence(row, size, firstNorm3);
    secondNorm3 = rotationSequence[i];
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.4.1.6 const float getBMetric_3 (const VectorXf & *firstRow,* const int & *size,* const VectorXf & *secondRow*)

Definition at line 72 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getSequence(firstRow, size, firstNorm3);
    getSequence(secondRow, size, secondNorm3);
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.4.1.7 const float getBMetric_3 (const int & *first,* const int & *second,* const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 92 of file Distance.cpp.

```
{
    return getBMetric(rotationSequence[first], rotationSequence[second]);
}
```

5.4.1.8 const float getBMetric_6 (const VectorXf & *row,* const int & *size,* const int & *i,* const std::vector< MultiVariate > & *normalMultivariate*)

Definition at line 110 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getNormalMultivariate(row, size, centerNormal);
    neighNormal = normalMultivariate[i];
    return getBMetric(centerNormal, neighNormal);
}
```

5.4.1.9 const float getBMetric_6 (const VectorXf & *firstRow,* const int & *size,* const VectorXf & *secondRow*)

Definition at line 131 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getNormalMultivariate(firstRow, size, centerNormal);
    getNormalMultivariate(secondRow, size, neighNormal);
    return getBMetric(centerNormal, neighNormal);
}
```

5.4.1.10 const float getBMetric_6 (const int & *first,* const int & *second,* const std::vector< MultiVariate > & *normalMultivariate*)

Definition at line 151 of file Distance.cpp.

```
{
    return getBMetric(normalMultivariate[first], normalMultivariate[second]);
}
```

5.4.1.11 const float getBMetric_7 (const VectorXf & *row*, const int & *size*, const int & *i*, const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 169 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getEachFixedSequence(row, size, firstNorm3);
    secondNorm3 = rotationSequence[i];
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.4.1.12 const float getBMetric_7 (const VectorXf & *firstRow*, const int & *size*, const VectorXf & *secondRow*)

Definition at line 190 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getEachFixedSequence(firstRow, size, firstNorm3);
    getEachFixedSequence(secondRow, size, secondNorm3);
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.4.1.13 const float getBMetric_7 (const int & *first*, const int & *second*, const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 210 of file Distance.cpp.

```
{
    return getBMetric(rotationSequence[first], rotationSequence[second]);
}
```

5.4.1.14 const float getBMetric_9 (const VectorXf & *row*, const int & *size*, const int & *i*, const std::vector< **MultiVariate** > & *normalMultivariate*)

Definition at line 228 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getUnnormalizedMultivariate(row, size, centerNormal);
    neighNormal = normalMultivariate[i];
    return getBMetric(centerNormal, neighNormal);
}
```

5.4.1.15 const float getBMetric_9 (const VectorXf & *firstRow*, const int & *size*, const VectorXf & *secondRow*)

Definition at line 249 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getUnnormalizedMultivariate(firstRow, size, centerNormal);
    getUnnormalizedMultivariate(secondRow, size, neighNormal);
    return getBMetric(centerNormal, neighNormal);
}
```

5.4.1.16 const float getBMetric_9 (const int & *first*, const int & *second*, const std::vector< **MultiVariate** > & *normalMultivariate*)

Definition at line 269 of file Distance.cpp.

```
{
    return getBMetric(normalMultivariate[first], normalMultivariate[second]);
}
```

5.4.1.17 const float getDisimilarity (const MatrixXf & *data*, const int & *first*, const int & *second*, const int & *normOption*, const **MetricPreparation** & *object*)

Definition at line 964 of file Distance.cpp.

```
{
    return getDisimilarity(data.row(first), data.row(second),
                           first, second, normOption, object);
}
```

5.4.1.18 const float getDisimilarity (const VectorXf & *others*, const MatrixXf & *data*, const int & *index*, const int & *normOption*, const **MetricPreparation** & *object*)

Definition at line 985 of file Distance.cpp.

```
{
    float length;
    switch(normOption)
    {
    case 0: // Euclidean distance, d_E
    case 1: // Fraction norm, d_F
    case 2: // geometric similarity measure, d_G
    case 5:
    case 8:
    case 11:
        length = getNorm(others, data.row(index),index,normOption,
                         object.pairwise, object.pairwiseNorm);
        break;

     case 3:
        length = getBMetric_3(others, others.size()/3-2, index,
                              object.rotationSequence);
        break;

     case 4:
        length = abs(object.rotation[index]-
                     getRotation(others, others.size()/3-2));
        break;

     case 6:
        length = getBMetric_6(others, others.size()/3-1, index,
                              object.normalMultivariate);
        break;

     case 7:
        length = getBMetric_7(others, others.size()/3-1, index,
                              object.rotationSequence);
        break;

     case 9:
        length = getBMetric_9(others, others.size()/3-1, index,
                              object.normalMultivariate);
        break;

     case 10:
        length = getMetric_10(others, others.size()/3, index,
                              object.unitLength);
        break;

    case 12:    // the MCP distance, i.e., d_M
        length = getMetric_MOP(others, data.row(index));
        break;

    case 13:    // the Hausdorff distance, i.e., d_H
        length = getMetric_Hausdorff(others, data.row(index));
        break;

    /* signature-based similarity metric with chi-squared test combined with mean-closest */
    case 14:    // the signature-based similarity, i.e., d_S
        length = getSignatureMetric(others,data.row(index),object.pairwise[index]);
        break;

    /* adapted Procrustes distance */
    case 15:    // the Procrustes distance, i.e., d_P
        //length = getProcrustesMetric(others, data.row(index));
        length = std::min(getProcrustesMetricSegment(others, data.row(index)),
                getProcrustesMetricSegment(data.row(index), others));
        break;

    case 16:
        length = getEntropyMetric(object.pairwise[index], others);
        break;

    case 17:    // the time-based MCP, i.e., d_T
        length = getPathline_MCP(others, data.row(index));
        break;

    default:
        exit(1);
        break;
    }

    return length;
}
```

5.4.1.19 const float getDisimilarity (const VectorXf & *first*, const VectorXf & *second*, const int & *firstIndex*, const int & *secondIndex*, const int & *normOption*, const MetricPreparation & *object*)

Definition at line 1082 of file Distance.cpp.

```
{
    float length;
    switch(normOption)
    {
    case 0: // Euclidean distance, d_E
    case 1: // Fraction norm, d_F
    case 2: // Geometric similarity, d_G
    case 5:
    case 8:
    case 11:
        length = getNorm(first, second,firstIndex,secondIndex, normOption,
            object.pairwise, object.pairwiseNorm);
        break;

    case 3:
        length = getBMetric_3(firstIndex, secondIndex,
            object.rotationSequence);
        break;

    case 4:
        length = abs(object.rotation[firstIndex]-
            object.rotation[secondIndex]);
        break;

    case 6:
        length = getBMetric_6(firstIndex, secondIndex,
            object.normalMultivariate);
        break;

    case 7:
        length = getBMetric_7(firstIndex, secondIndex,
            object.rotationSequence);
        break;

    case 9:
        length = getBMetric_9(firstIndex, secondIndex,
            object.normalMultivariate);
        break;

    case 10:
        length = getMetric_10(firstIndex, secondIndex,
                    object.unitLength);
        break;

    case 12:    // the MCP distance, d_M
        length = getMetric_MOP(first, second);
        break;

    case 13:    // the Hausdorff distance, d_H
        length = getMetric_Hausdorff(first, second);
        break;

    case 14:    //  the signature-based distance, d_S
        length = getSignatureMetric(first,second,object.pairwise[firstIndex],object.pairwise[secondIndex]);
        break;

    case 15:    // the Procrutes distance, d_P
        //length = getProcrustesMetric(first, second);
        length = std::min(getProcrustesMetricSegment(first,second), getProcrustesMetricSegment(second,first));
        break;

    case 16:
        length = getEntropyMetric(object.pairwise[firstIndex], object.pairwise[secondIndex]);
        break;

    case 17:    // the time-based MCP, d_T
        length = getPathline_MCP(first, second);
        break;

    default:
        exit(1);
        break;
    }
    return length;
}
```

5.4.1.20 const float getDisimilarity (const VectorXf & *first*, const VectorXf & *second*, const int & *normOption*, const MetricPreparation & *object*)

Definition at line 1174 of file Distance.cpp.

```
{
    float length;
    switch(normOption)
    {
    case 0: // Euclidean distance, d_E
    case 1: // Fraction norm, d_F
    case 2: // Geometric similarity, d_G
    case 5:
    case 8:
    case 11:
        length = getNorm(first, second, normOption);
        break;

    case 3:
        length = getBMetric_3(first, first.size()/3-2, second);
        break;

    case 4:
        length = abs(getRotation(first, first.size()/3-2)-getRotation(second, second.size()/3-2));
        break;

    case 6:
        length = getBMetric_6(first, first.size()/3-1,second);
        break;

    case 7:
        length = getBMetric_7(first, first.size()/3-1, second);
        break;

    case 9:
        length = getBMetric_9(first, first.size()/3-1, second);
        break;

    case 10:
        length = getBMetric_9(first, first.size()/3, second);
        break;

    case 12:    // the MCP distance, d_M
        length = getMetric_MOP(first, second);
        break;

    case 13:    // the Hausdorff distance, d_H
        length = getMetric_Hausdorff(first, second);
        break;

    /* signature-based similarity metric with chi-squared test combined with mean-closest */
    case 14:    // the signature-based similarity, d_S
        length = getSignatureMetric(first, second);
        break;

    /* adapted Procrustes distance */
    case 15:    // the Procrustes distance, d_P
        //length = getProcrustesMetric(first, second);
        length = getProcrustesMetricSegment(first,second);
        break;

    case 16:
        length = getEntropyMetric(first, second);
        break;

    case 17:    // the time-based MCP, d_T
        length = getPathline_MCP(first, second);
        break;

    default:
        exit(1);
        break;
    }

    return length;
}
```

5.4.1.21 void getDistanceMatrix (const MatrixXf & *data,* const int & *normOption,* const MetricPreparation & *object*)

Definition at line 1348 of file Distance.cpp.

```
{
    const int& Row = data.rows();
    distanceMatrix = new float*[Row];

    // assign the distance matrix
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        distanceMatrix[i] = new float[Row];
        for (int j = 0; j < Row; ++j)
        {
            /* don't wish to waste computation on diagonal element */
            if(i==j)
                distanceMatrix[i][j] = 0.0;
            else
                distanceMatrix[i][j] = getDisimilarity(data, i, j, normOption,
   object);
        }
    }
    // help check whether they already been assigned and whether they are symmetric or not
    std::cout << "Distance between 215 and 132 is " << distanceMatrix[215][132] << std::endl;
    std::cout << "Distance between 132 and 215 is " << distanceMatrix[132][215] << std::endl;

    std::cout << "Finished computing distance matrix!" << std::endl;
}
```

5.4.1.22 const float getEntropyMetric (const std::vector< float > & *firstEntropy,* const std::vector< float > & *secondEntropy*)

Definition at line 1845 of file Distance.cpp.

```
{
    assert(firstEntropy.size()==2);
    assert(secondEntropy.size()==2);

    float first = firstEntropy[0]-secondEntropy[0];
    float second = firstEntropy[1]-secondEntropy[1];

    return sqrt(first*first+second*second);
}
```

5.4.1.23 const float getEntropyMetric (const std::vector< float > & *firstEntropy,* const Eigen::VectorXf & *array*)

Definition at line 1868 of file Distance.cpp.

```
{
    assert(firstEntropy.size()==2);

    std::vector<float> secondEntropy;

    getLinearAngularEntropy(array, BUNDLE_SIZE, secondEntropy);

    return getEntropyMetric(firstEntropy, secondEntropy);
}
```

5.4.1.24 const float getEntropyMetric (const Eigen::VectorXf & *first,* const Eigen::VectorXf & *second*)

Definition at line 1890 of file Distance.cpp.

```
{

    std::vector<float> firstEntropy, secondEntropy;

    getLinearAngularEntropy(first, BUNDLE_SIZE, firstEntropy);
    getLinearAngularEntropy(second, BUNDLE_SIZE, secondEntropy);

    return getEntropyMetric(firstEntropy, secondEntropy);
}
```

5.4.1.25 const float getMetric_10 (const VectorXf & *centroid,* const int & *size,* const int & *index,* const std::vector< VectorXf > & *unitLength*)

Definition at line 358 of file Distance.cpp.

```
{
    const VectorXf& x = unitLength[index];
    VectorXf y(size*3);
    getUnitDirection_byEach(centroid,size,y);

    float length = x.dot(y)/x.size();
    length = min(1.0,(double)length);
    length = max(-1.0,(double)length);
    length = acos(length);
    return length;
}
```

5.4.1.26 const float getMetric_10 (const VectorXf & *firstRow*, const int & *size*, const VectorXf & *secondRow*)

Definition at line 383 of file Distance.cpp.

```
{
    VectorXf x(size*3), y(size*3);
    getUnitDirection_byEach(firstRow,size,x);
    getUnitDirection_byEach(secondRow,size,y);

    float length = x.dot(y)/x.size();
    length = min(1.0,(double)length);
    length = max(-1.0,(double)length);
    length = acos(length);
    return length;
}
```

5.4.1.27 const float getMetric_10 (const int & *first*, const int & *second*, const std::vector< VectorXf > & *unitLength*)

Definition at line 407 of file Distance.cpp.

```
{
    const VectorXf& x = unitLength[first];
    const VectorXf& y = unitLength[second];
    float length = x.dot(y)/x.size();
    length = min(1.0,(double)length);
    length = max(-1.0,(double)length);
    length = acos(length);
    return length;
}
```

5.4.1.28 const float getMetric_Hausdorff (const VectorXf & *first*, const VectorXf & *second*)

Definition at line 1305 of file Distance.cpp.

```
{
    // the max of first to second
    const int& vNum = first.size()/3;
    float result, f_to_s=-1.0, s_to_f=-1.0;
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(first(3*i),first(3*i+1),first(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(second(3*j),second(3*j+1),second(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        s_to_f=std::max(s_to_f, minDist);
    }

    // the max of second to first
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(second(3*i),second(3*i+1),second(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(first(3*j),first(3*j+1),first(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        f_to_s=std::max(f_to_s, minDist);
    }

    // max of the max
    result = std::max(f_to_s, s_to_f);
    return result;
}
```

5.4.1.29 const float getMetric_MOP (const VectorXf & *first,* const VectorXf & *second*)

Definition at line 1258 of file Distance.cpp.

```
{
    // The MCP of first to second
    const int& vNum = first.size()/3;
    float result, f_to_s, s_to_f;
    float summation = 0;
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(first(3*i),first(3*i+1),first(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(second(3*j),second(3*j+1),second(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        summation+=minDist;
    }
    s_to_f = summation/vNum;

    // The MCP of second to first
    summation = 0;
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(second(3*i),second(3*i+1),second(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(first(3*j),first(3*j+1),first(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        summation+=minDist;
    }
    f_to_s = summation/vNum;

    // get the average of that
    result = (f_to_s+s_to_f)/2.0;
    return result;
}
```

5.4.1.30 const float getNorm (const Eigen::VectorXf & *centroid,* const Eigen::VectorXf & *r2,* const int & *index,* const int & *normOption,* const std::vector< std::vector< float > > & *pairwise,* const std::vector< std::vector< float > > & *objectNorm*)

Definition at line 432 of file Distance.cpp.

```
{
    assert(centroid.size()==r2.size());
    float length = 0.0;
    switch(normOption)
    {
    case 0: // Euclidean distance
    default:
        length = (centroid-r2).norm();
        break;

    case 11:    // the norm 11
        {
            float dotPro = centroid.dot(r2);
            float firstNorm = centroid.norm();
            float secondNorm = r2.norm();
            float firstInverse, secondInverse;
            if(firstNorm<1.0e-8)
                firstInverse = 1.0e8;
            else
                firstInverse = 1.0/firstNorm;
            if(secondNorm<1.0e-8)
                secondInverse = 1.0e8;
            else
                secondInverse = 1.0/secondNorm;
            dotPro = dotPro*firstInverse*secondInverse;
            dotPro = std::max(dotPro, float(-1.0));
            dotPro = std::min(dotPro, float(1.0));
            length = acos(dotPro)/M_PI;
        }
        break;

    case 1:  /* fraction norm by high-dimensional feature-space */
        {
            for (int i = 0; i < centroid.size(); ++i)
            {
                length += pow(abs(centroid(i)-r2(i)),0.5);
            }
            length = pow(length,2.0);
        }
        break;

    case 2: /* mean value of dot product value, which means it’s rotational invariant, or d_G */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, result;

            std::vector<float> centroidWise;
            std::vector<float> centroidWiseNorm;
            getPairWise_byEach(centroid, pointNum, centroidWise, centroidWiseNorm);

            const std::vector<float>& i_Pairwise = pairwise[index];
            const std::vector<float>& i_PairNorm = objectNorm[index];

            Vector3f left, right;
            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = centroidWiseNorm[i];
                rightNorm = i_PairNorm[i];
                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << centroidWise[3*i], centroidWise[3*i+1], centroidWise[3*i+2];
                    right << i_Pairwise[3*i],i_Pairwise[3*i+1],i_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
        }
        break;

    case 5: /* rotational invariant line-wise acos angle with normal direction for
               measuring whether counterclockwise or clockwise orientation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, normalDot, result;
            Vector3f left, right, normal;

            std::vector<float> centroidWise;
            std::vector<float> centroidNorm;
            getPairWise_byEach(centroid, pointNum, centroidWise, centroidNorm);
            const std::vector<float>& i_Pairwise = pairwise[index];
            const std::vector<float>& i_PairNorm = objectNorm[index];

            left << /*centroid(3)-centroid(0),centroid(4)-centroid(1),centroid(5)-centroid(2)*/
                    centroidWise[0], centroidWise[1], centroidWise[2];
            right << /*r2(3)-r2(0),r2(4)-r2(1),r2(5)-r2(2)*/
                    i_Pairwise[0], i_Pairwise[1], i_Pairwise[2];
            const Vector3f& Normal = left.cross(right);

            for (int i = 0; i < pointNum; ++i)
            {

                leftNorm = centroidNorm[i];
                rightNorm = i_PairNorm[i];
                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << centroidWise[3*i], centroidWise[3*i+1], centroidWise[3*i+2];
                    right << i_Pairwise[3*i],i_Pairwise[3*i+1],i_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    normal = left.cross(right);
                    normalDot = Normal.dot(normal);
                    if(normalDot<0)
                        length+=-acos(result);
                    else
                        length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
            length = abs(length);
        }
        break;

    case 8: /* distance metric defined as mean * standard deviation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, stdevia = 0.0, angle, result;
            Vector3f left, right;

            std::vector<float> centroidWise;
            std::vector<float> centroidNorm;
            getPairWise_byEach(centroid, pointNum, centroidWise, centroidNorm);

            const std::vector<float>& i_Pairwise = pairwise[index];
            const std::vector<float>& i_PairNorm = objectNorm[index];

            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = centroidNorm[i];
                rightNorm = i_PairNorm[i];

                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << centroidWise[3*i], centroidWise[3*i+1], centroidWise[3*i+2];
                    right << i_Pairwise[3*i],i_Pairwise[3*i+1],i_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    angle = acos(result);
                    length+=angle;
                    stdevia+=angle*angle;
                }
                else
                {
                    angle=M_PI;
                    length+=angle;
                    stdevia+=angle*angle;
                }
            }
            length /= pointNum;
            stdevia = stdevia/pointNum-length*length;
            if(stdevia>0)
                stdevia = sqrt(stdevia/pointNum-length*length);
            else
                stdevia = 1.0e-4;
        }
        break;
    }

    return length;
}
```

5.4.1.31 const float getNorm (const VectorXf & *centroid,* const VectorXf & *r2,* const int & *firstIndex,* const int & *secondIndex,* const int & *normOption,* const std::vector< std::vector< float > > & *pairwise,* const std::vector< std::vector< float > > & *objectNorm*)

Definition at line 620 of file Distance.cpp.

```
{
    assert(centroid.size()==r2.size());
    float length = 0.0;
    switch(normOption)
    {
    case 0:
    default:
        length = (centroid-r2).norm();
        break;

    case 1:  /* fraction norm by high-dimensional feature-space, or d_F */
        {
            for (int i = 0; i < centroid.size(); ++i)
            {
                length += pow(abs(centroid(i)-r2(i)),0.5);
            }
            length = pow(length,2.0);
        }
        break;

    case 11:
        {
            float dotPro = centroid.dot(r2);
            float firstNorm = centroid.norm();
            float secondNorm = r2.norm();
            float firstInverse, secondInverse;
            if(firstNorm<1.0e-8)
                firstInverse = 1.0e8;
            else
                firstInverse = 1.0/firstNorm;
            if(secondNorm<1.0e-8)
                secondInverse = 1.0e8;
            else
                secondInverse = 1.0/secondNorm;
            dotPro = dotPro*firstInverse*secondInverse;
            dotPro = std::max(dotPro, float(-1.0));
            dotPro = std::min(dotPro, float(1.0));
            length = acos(dotPro)/M_PI;
        }
        break;

    case 2: /* mean value of dot product value, which means it’s rotational invariant, or d_G */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, result;

            const std::vector<float>& i_Pairwise = pairwise[firstIndex];
            const std::vector<float>& j_Pairwise = pairwise[secondIndex];

            const std::vector<float>& i_PairNorm = objectNorm[firstIndex];
            const std::vector<float>& j_PairNorm = objectNorm[secondIndex];

            Vector3f left, right;
            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = i_PairNorm[i];
                rightNorm = j_PairNorm[i];
                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << i_Pairwise[3*i], i_Pairwise[3*i+1], i_Pairwise[3*i+2];
                    right << j_Pairwise[3*i],j_Pairwise[3*i+1],j_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
        }
        break;

    case 5: /* rotational invariant line-wise acos angle with normal direction for
               measuring whether counterclockwise or clockwise orientation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, normalDot, result;
            Vector3f left, right, normal;

            const std::vector<float>& i_Pairwise = pairwise[firstIndex];
            const std::vector<float>& j_Pairwise = pairwise[secondIndex];

            const std::vector<float>& i_PairNorm = objectNorm[firstIndex];
            const std::vector<float>& j_PairNorm = objectNorm[secondIndex];

            left << /*centroid(3)-centroid(0),centroid(4)-centroid(1),centroid(5)-centroid(2)*/
                    i_Pairwise[0], i_Pairwise[1], i_Pairwise[2];
            right << /*r2(3)-r2(0),r2(4)-r2(1),r2(5)-r2(2)*/
                    j_Pairwise[0], j_Pairwise[1], j_Pairwise[2];
            const Vector3f& Normal = left.cross(right);

            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = i_PairNorm[i];
                rightNorm = j_PairNorm[i];

                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << i_Pairwise[3*i], i_Pairwise[3*i+1], i_Pairwise[3*i+2];
                    right << j_Pairwise[3*i],j_Pairwise[3*i+1],j_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    normal = left.cross(right);
                    normalDot = Normal.dot(normal);
                    if(normalDot<0)
                        length+=-acos(result);
                    else
                        length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
            length = abs(length);
        }
        break;

    case 8: /* distance metric defined as mean * standard deviation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, stdevia = 0.0, angle, result;
            Vector3f left, right;

            const std::vector<float>& i_Pairwise = pairwise[firstIndex];
            const std::vector<float>& j_Pairwise = pairwise[secondIndex];

            const std::vector<float>& i_PairNorm = objectNorm[firstIndex];
            const std::vector<float>& j_PairNorm = objectNorm[secondIndex];

            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = i_PairNorm[i];
                rightNorm = j_PairNorm[i];

                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << i_Pairwise[3*i], i_Pairwise[3*i+1], i_Pairwise[3*i+2];
                    right << j_Pairwise[3*i],j_Pairwise[3*i+1],j_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    angle = acos(result);
                    length+=angle;
                    stdevia+=angle*angle;
                }
                else
                {
                    angle=M_PI;
                    length+=angle;
                    stdevia+=angle*angle;
                }
            }
            length /= pointNum;
            stdevia = stdevia/pointNum-length*length;
            if(stdevia>0)
                stdevia = sqrt(stdevia/pointNum-length*length);
            else
                stdevia = 1.0e-4;
        }
        break;

    }

    return length;
}
```

5.4.1.32 const float getNorm (const Eigen::VectorXf & *r1*, const Eigen::VectorXf & *r2*, const int & *normOption*)

Definition at line 804 of file Distance.cpp.

```
{
    assert(r1.size()==r2.size());
    float length = 0.0;
    switch(normOption)
    {
    case 0:
    default:
        length = (r1-r2).norm();
        break;

    case 1:  /* fraction norm by high-dimensional feature-space */
        {
            for (int i = 0; i < r1.size(); ++i)
            {
                length += pow(abs(r1(i)-r2(i)),0.5);
            }
            length = pow(length,2.0);
        }
        break;

    case 2: /* mean value of dot product value, which means it's rotational invariant, or d_G */
        {
            const int& pointNum = r1.size()/3-1;
            float dotValue, leftNorm, rightNorm, result;
            Vector3f left, right;
            for (int i = 0; i < pointNum; ++i)
            {
                left << r1(3*i+3)-r1(3*i),r1(3*i+4)-r1(3*i+1),r1(3*i+5)-r1(3*i+2);
                right << r2(3*i+3)-r2(3*i),r2(3*i+4)-r2(3*i+1),r2(3*i+5)-r2(3*i+2);
                dotValue = left.dot(right);
                leftNorm = left.norm(), rightNorm = right.norm();
                if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
                {
                    result = dotValue/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
        }
        break;

    case 5: /* rotational invariant line-wise acos angle with normal direction for
               measuring whether counterclockwise or clockwise orientation */
        {
            const int& pointNum = r1.size()/3-1;
            float dotValue, leftNorm, rightNorm, normalDot, result;
            Vector3f left, right, normal;

            left << r1(3)-r1(0),r1(4)-r1(1),r1(5)-r1(2);
            right << r2(3)-r2(0),r2(4)-r2(1),r2(5)-r2(2);

            const Vector3f& Normal = left.cross(right);

            for (int i = 0; i < pointNum; ++i)
            {
                left << r1(3*i+3)-r1(3*i),r1(3*i+4)-r1(3*i+1),r1(3*i+5)-r1(3*i+2);
                right << r2(3*i+3)-r2(3*i),r2(3*i+4)-r2(3*i+1),r2(3*i+5)-r2(3*i+2);
                normal = left.cross(right);
                dotValue = left.dot(right);
                normalDot = Normal.dot(normal);

                leftNorm = left.norm(), rightNorm = right.norm();
                if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
                {
                    result = dotValue/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    if(normalDot<0)
                        length+=-acos(result);
                    else
                        length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
            length = abs(length);
        }
        break;

    case 8: /* distance metric defined as mean * standard deviation */
        {
            const int& pointNum = r1.size()/3-1;
            float dotValue, leftNorm, rightNorm, stdevia = 0.0, angle, result;
            Vector3f left, right;
            for (int i = 0; i < pointNum; ++i)
            {
                left << r1(3*i+3)-r1(3*i),r1(3*i+4)-r1(3*i+1),r1(3*i+5)-r1(3*i+2);
                right << r2(3*i+3)-r2(3*i),r2(3*i+4)-r2(3*i+1),r2(3*i+5)-r2(3*i+2);
                dotValue = left.dot(right);
                leftNorm = left.norm(), rightNorm = right.norm();
                if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
                {
                    result = dotValue/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    angle = acos(result);
                    std::cout << angle << std::endl;
                    length+=angle;
                    stdevia+=angle*angle;
                }
                else
                {
                    angle=M_PI;
                    length+=angle;
                    stdevia+=angle*angle;
                }
            }
            length /= pointNum;
            stdevia = stdevia/pointNum-length*length;
            if(stdevia>0)
                stdevia = sqrt(stdevia/pointNum-length*length);
            else
                stdevia = 1.0e-4;
            length*=stdevia;
        }
        break;

    case 11:
        {
            float dotPro = r1.dot(r2);
            float firstNorm = r1.norm();
            float secondNorm = r2.norm();
            float firstInverse, secondInverse;
            if(firstNorm<1.0e-8)
                firstInverse = 1.0e8;
            else
                firstInverse = 1.0/firstNorm;
            if(secondNorm<1.0e-8)
                secondInverse = 1.0e8;
            else
                secondInverse = 1.0/secondNorm;
            dotPro = dotPro*firstInverse*secondInverse;
            dotPro = std::max(dotPro, float(-1.0));
            dotPro = std::min(dotPro, float(1.0));
            length = acos(dotPro)/M_PI;
        }
        break;
    }

    return length;
}
```

5.4.1.33 const float getPathline_MCP (const Eigen::VectorXf & *first*, const Eigen::VectorXf & *second*)

Definition at line 1910 of file Distance.cpp.

```
{
    /* preset the initial time step is 0, then 1, 2, ... as long as it will be normalized */
    const int& t_M = first.size()/3-1;
    float dist = 0.0, a, b, c;
    Eigen::Vector3f temp, another, diff;
    for(int i=0; i<t_M; ++i)
    {
        temp=Eigen::Vector3f(first(i*3)-second(i*3), first(3*i+1)-second(3*i+1), first(3*i+2)-second(3*i+2));
        another=Eigen::Vector3f(first(i*3+3)-second(i*3+3), first(3*i+4)-second(3*i+4), first(3*i+5)-second(3*i+5));
        diff=another-temp;

        a=temp.transpose()*temp;
        b=temp.transpose()*diff;
        c=diff.transpose()*diff;

        dist+=get_calculus(a, b, c);
    }
    return dist/t_M;
}
```

5.4.1.34 const float getProcrustesMetric (const Eigen::VectorXf & *first,* const Eigen::VectorXf & *second*)

Definition at line 1557 of file Distance.cpp.

```
{
    assert(first.size()==second.size());

    const int& vertexCount = first.size()/3;

    const int& vertexChanged = vertexCount-2*(PROCRUSTES_SIZE/2);
    const int& newSize = 3*vertexChanged;

    /* assign the segment list */
    Eigen::MatrixXf firstSegment(PROCRUSTES_SIZE,3), secondSegment(
PROCRUSTES_SIZE,3), X0;

    int location, rightIndex;

    Eigen::Vector3f first_average, second_average, tempPoint;

    /* A is SVD target, rotation is optimal rotation matrix, and secondPrime is P' after superimposition */
    Eigen::MatrixXf A, rotation, secondPrime = Eigen::MatrixXf(PROCRUSTES_SIZE,3);

    float optimalScaling, traceA, pointDist;

    float result = 0.0;

    /* for all points, assign to them a point set with size of PROCRUSTES_SIZE neighboring points */
    for(int i=0;i<vertexChanged;++i)
    {
        rightIndex = i+PROCRUSTES_SIZE;

        first_average = second_average = Eigen::VectorXf::Zero(3);

        /* get the point set of neighboring 7 points and average */
        for(int j=i;j<rightIndex;++j)
        {
            location = j-i;
            for(int k=0;k<3;++k)
            {
                firstSegment(location,k)=first(3*j+k);
                secondSegment(location,k)=second(3*j+k);
            }

            first_average+=firstSegment.row(location);
            second_average+=secondSegment.row(location);
        }

        first_average/=PROCRUSTES_SIZE;
        second_average/=PROCRUSTES_SIZE;

        /* reserve the matrix */
        X0 = firstSegment;

        /* centralization for the point set */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            firstSegment.row(j) = firstSegment.row(j)-first_average.transpose();
            secondSegment.row(j) = secondSegment.row(j)-second_average.transpose();
        }

        /* get ssqX and ssqY */
        float ssqX = (firstSegment.cwiseProduct(firstSegment)).sum();
        float ssqY = (secondSegment.cwiseProduct(secondSegment)).sum();

        /* check whether negative or not */
        assert(ssqX > 0 && ssqY > 0);

        ssqX = sqrt(ssqX);
        ssqY = sqrt(ssqY);

        /* scaling for the point set */
        firstSegment/=ssqX;
        secondSegment/=ssqY;

        /* get the optimal rotational matrix by othogonal Procrutes analysis */
        A = firstSegment.transpose()*secondSegment;

        /* perform SVD on A */
        JacobiSVD<MatrixXf> svd(A, ComputeThinU | ComputeThinV);

        /* get the optimal 3D rotation */
        rotation = svd.matrixV()*(svd.matrixU().transpose());

        /* get trace for singular value matrix */
        traceA = svd.singularValues().sum();

        /* get optimal scaling */
        optimalScaling = traceA*ssqX/ssqY;

        /* preset the average to the P' */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
            secondPrime.row(j) = first_average;

        /* get P' in superimposed space */
        secondPrime = ssqX*traceA*secondSegment*rotation+secondPrime;

        /* compute the distance and store them in the std::vector<float> */
        pointDist = 0.0;
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            tempPoint = X0.row(j)-secondPrime.row(j);
            pointDist+= tempPoint.transpose()*tempPoint;
        }

        /* get the average of P(x,y')^2 */

        // either by computing the matrix
        //result+=pointDist;

        // or directly using trace of the matrix
        float requiredD = 1.0-traceA*traceA;
        result+=requiredD*requiredD;
    }
    return result/vertexChanged;
}
```

5.4.1.35 const float getProcrustesMetricSegment (const Eigen::VectorXf & *first*, const Eigen::VectorXf & *second*)

Definition at line 1681 of file Distance.cpp.

```
{
    assert(first.size()==second.size());

    const int& vertexCount = first.size()/3;

    const int& vertexChanged = vertexCount/PROCRUSTES_SIZE;
    const int& newSize = 3*vertexChanged;

    /* assign the segment list */
    Eigen::MatrixXf firstSegment(PROCRUSTES_SIZE,3), secondSegment(
  PROCRUSTES_SIZE,3), X0;

    int location, rightIndex;

    Eigen::Vector3f first_average, second_average, tempPoint;

    /* A is SVD target, rotation is optimal rotation matrix, and secondPrime is P' after superimposition */
    Eigen::MatrixXf A, rotation, secondPrime = Eigen::MatrixXf(PROCRUSTES_SIZE,3);

    float optimalScaling, traceA, pointDist;

    float result = 0.0;

    int effective = 0;
    /* for all points, assign to them a point set with size of PROCRUSTES_SIZE neighboring points */
    for(int i=0;i<vertexChanged;++i)
    {
        rightIndex = PROCRUSTES_SIZE*i+PROCRUSTES_SIZE;

        first_average = second_average = Eigen::VectorXf::Zero(3);

        /* get the point set of neighboring 7 points and average */
        for(int j=PROCRUSTES_SIZE*i;j<rightIndex;++j)
        {
            location = j-PROCRUSTES_SIZE*i;
            for(int k=0;k<3;++k)
            {
                firstSegment(location,k)=first(3*j+k);
                secondSegment(location,k)=second(3*j+k);
            }

            first_average+=firstSegment.row(location);
            second_average+=secondSegment.row(location);
        }

        first_average/=PROCRUSTES_SIZE;
        second_average/=PROCRUSTES_SIZE;

        /* reserve the matrix */
        X0 = firstSegment;

        /* centralization for the point set */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            firstSegment.row(j) = firstSegment.row(j)-first_average.transpose();
            secondSegment.row(j) = secondSegment.row(j)-second_average.transpose();
        }

        /* get ssqX and ssqY */
        float ssqX = (firstSegment.cwiseProduct(firstSegment)).sum();
        float ssqY = (secondSegment.cwiseProduct(secondSegment)).sum();

        /* check whether negative or not */
        assert(ssqX > 0 && ssqY > 0);

        if(ssqX<1.0e-14 || ssqY<1.0e-14)
            continue;

        ssqX = sqrt(ssqX);
        ssqY = sqrt(ssqY);

        if(ssqX<1.0e-8 || ssqY<1.0e-8)
            continue;

        /* scaling for the point set */
        firstSegment/=ssqX;
        secondSegment/=ssqY;

        /* get the optimal rotational matrix by othogonal Procrutes analysis */
        A = firstSegment.transpose()*secondSegment;

        /* perform SVD on A */
        JacobiSVD<MatrixXf> svd(A, ComputeThinU | ComputeThinV);

        /* get the optimal 3D rotation */
        rotation = svd.matrixV()*(svd.matrixU().transpose());

        /* get trace for singular value matrix */
        traceA = svd.singularValues().sum();

        /* get optimal scaling */
        optimalScaling = traceA*ssqX/ssqY;

        /* preset the average to the P' */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
            secondPrime.row(j) = first_average;

        /* get P' in superimposed space */
        secondPrime = ssqX*traceA*secondSegment*rotation+secondPrime;

        /* compute the distance and store them in the std::vector<float> */
        pointDist = 0.0;
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            tempPoint = X0.row(j)-secondPrime.row(j);
            pointDist+= tempPoint.transpose()*tempPoint;
        }
        /* get the average of P(x,y')^2 */
        result+=pointDist;
        ++effective;
    }

    if(effective==0)
    {
        return 1.0e-8;
    }
    else
        return result/effective;
}
```

5.4.1.36 const float getRotation (const std::vector< vector< float > > & *streamline,* std::vector< float > & *rotation*)

Definition at line 1406 of file Distance.cpp.

```
{
    if(streamline.empty())
        return -1;
    float result = 0, eachSum;
    const int& size = streamline.size();
    rotation = std::vector<float>(size);
    std::vector<float> eachLine;
    Eigen::Vector3f first, second;
    int lineSize;
    for(int i=0;i<size;++i)
    {
        eachSum = 0;
        eachLine = streamline[i];
        lineSize = eachLine.size()/3-2;
        // calculate the summation of discrete curvatures
        for(int j=0;j<lineSize;++j)
        {
            first<<eachLine[3*j+3]-eachLine[3*j],eachLine[3*j+4]-eachLine[3*j+1],eachLine[3*j+5]-eachLine[3*j+2];
            second<<eachLine[3*j+6]-eachLine[3*j+3],eachLine[3*j+7]-eachLine[3*j+4],eachLine[3*j+8]-eachLine[3*j+5];

            float firstNorm = first.norm(), secondNorm = second.norm();
            if(firstNorm>=1.0e-8 && secondNorm>=1.0e-8)
            {
                float angle = first.dot(second)/firstNorm/secondNorm;
                angle = std::max(angle,float(-1.0));
                angle = std::min(angle,float(1.0));
                eachSum+=acos(angle);
            }
        }
        // get the mean of discrete curvatures
        rotation[i]=eachSum;
        result+=eachSum;
    }
    result/=size;
    return result;
}
```

5.4.1.37 const float getSignatureMetric (const Eigen::VectorXf & *firstArray*, const Eigen::VectorXf & *secondArray*, const std::vector< float > & *firstHist*, const std::vector< float > & *secondHist*)

Definition at line 1454 of file Distance.cpp.

```
{
    /* would choose alpha = 0.5, and 10% of subset vertices for mean_dist */
    const float& Alpha = 0.5;
    const int& SUBSET = 10;

    /* assert whether the size is the same */
    assert(firstArray.size()==secondArray.size());
    assert(firstHist.size()==secondHist.size());

    const int& histSize = firstHist.size();
    const int& vertexCount = firstArray.size()/3;
    const int& size = vertexCount/SUBSET+1;

    Eigen::VectorXf firstSubset(3*size), secondSubset(3*size);

    /* get mean_dist between two sampled subsets */
    int tempPos = 0;
    for(int i=0;i<vertexCount;i+=SUBSET)
    {
        for(int j=0;j<3;++j)
        {
            firstSubset(3*tempPos+j)=firstArray(3*i+j);
            secondSubset(3*tempPos+j)=secondArray(3*i+j);
        }
        ++tempPos;
    }

    /* get mean_dist */
    float result = getMetric_MOP(firstSubset, secondSubset);

    float chi_test = 0.0, histDiff, histSum;

    /* get chi_test for two histograms */
    for(int i=0;i<histSize;++i)
    {
        histDiff = firstHist[i]-secondHist[i];
        histSum = firstHist[i]+secondHist[i];
        /* check numerical error */
        if(histSum<1.0e-8)
            continue;

        chi_test+= histDiff*histDiff/histSum;
    }

    /* get combined distance */
    result = (1-Alpha)*chi_test + Alpha*result;

    return result;
}
```

5.4.1.38 const float getSignatureMetric (const Eigen::VectorXf & *centroid,* const Eigen::VectorXf & *first,* const std::vector< float > & *firstHist*)

Definition at line 1517 of file Distance.cpp.

```
{
    std::vector<float> centroidHist;
    /* get the bin-based histogram for signature */
    getSignatureHist(centroid, BIN_SIZE, centroidHist);

    return getSignatureMetric(centroid,first,centroidHist,firstHist);
}
```

5.4.1.39 const float getSignatureMetric (const Eigen::VectorXf & *first,* const Eigen::VectorXf & *second*)

Definition at line 1536 of file Distance.cpp.

```
{
    std::vector<float> firstHist, secondHist;
    /* get the bin-based histogram for signature */
    getSignatureHist(first, BIN_SIZE, firstHist);
    getSignatureHist(second, BIN_SIZE, secondHist);

    return getSignatureMetric(first,second,firstHist,secondHist);
}
```

5.4.2 Variable Documentation

5.4.2.1 float** distanceMatrix = NULL

Definition at line 39 of file Distance.cpp.

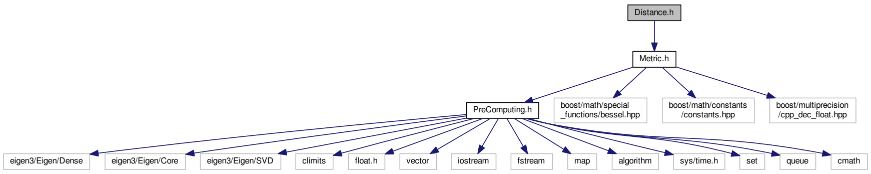
5.4.2.2 const int& PROCRUSTES_SIZE = 7

Definition at line 33 of file Distance.cpp.

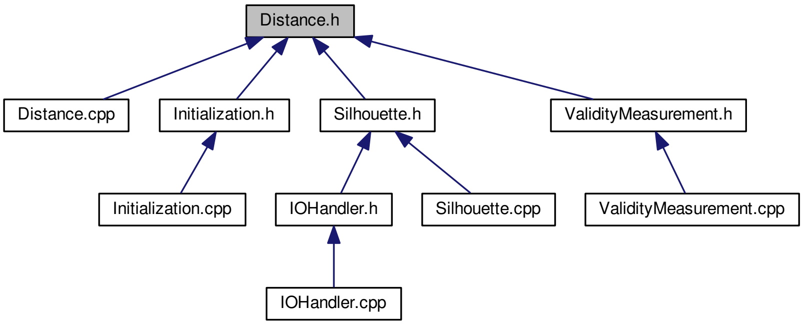
5.5 Distance.h File Reference

`#include "Metric.h"`

Include dependency graph for Distance.h:



This graph shows which files directly or indirectly include this file:



Functions

- const float getBMetric_3 (const VectorXf &row, const int &size, const int &i, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_3 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_3 (const int &first, const int &second, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_6 (const VectorXf &row, const int &size, const int &i, const std::vector< MultiVariate > &normalMultivariate)
- const float getBMetric_6 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_6 (const int &first, const int &second, const std::vector< MultiVariate > &normal↩ Multivariate)
- const float getBMetric_7 (const VectorXf &row, const int &size, const int &i, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_7 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_7 (const int &first, const int &second, const std::vector< std::vector< float > > &rotationSequence)
- const float getBMetric_9 (const VectorXf &row, const int &size, const int &i, const std::vector< MultiVariate > &normalMultivariate)
- const float getBMetric_9 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getBMetric_9 (const int &first, const int &second, const std::vector< MultiVariate > &normal↩ Multivariate)

- const float getBMetric (const std::vector< float > &first, const std::vector< float > &second)
- const float getBMetric (const MultiVariate &first, const MultiVariate &second)
- const float getMetric_10 (const VectorXf &centroid, const int &size, const int &index, const std::vector< VectorXf > &unitLength)
- const float getMetric_10 (const VectorXf &firstRow, const int &size, const VectorXf &secondRow)
- const float getMetric_10 (const int &first, const int &second, const std::vector< VectorXf > &unitLength)
- const float getMetric_MOP (const VectorXf &first, const VectorXf &second)
- const float getMetric_Hausdorff (const VectorXf &first, const VectorXf &second)
- const float getNorm (const Eigen::VectorXf &centroid, const Eigen::VectorXf &r2, const int &index, const int &normOption, const std::vector< std::vector< float > > &object, const std::vector< std::vector< float > > &objectNorm)
- const float getNorm (const VectorXf &centroid, const VectorXf &r2, const int &firstIndex, const int &second↩ Index, const int &normOption, const std::vector< std::vector< float > > &object, const std::vector< std↩ ::vector< float > > &objectNorm)
- const float getNorm (const VectorXf &r1, const VectorXf &r2, const int &normOption)
- const float getSignatureMetric (const Eigen::VectorXf &firstArray, const Eigen::VectorXf &secondArray, const std::vector< float > &firstHist, const std::vector< float > &secondHist)
- const float getSignatureMetric (const Eigen::VectorXf &centroid, const Eigen::VectorXf &first, const std↩ ::vector< float > &firstHist)
- const float getSignatureMetric (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getProcrustesMetric (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getProcrustesMetricSegment (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getEntropyMetric (const std::vector< float > &firstEntropy, const std::vector< float > &second↩ Entropy)
- const float getEntropyMetric (const std::vector< float > &firstEntropy, const Eigen::VectorXf &array)
- const float getEntropyMetric (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getPathline_MCP (const Eigen::VectorXf &first, const Eigen::VectorXf &second)
- const float getDisimilarity (const MatrixXf &data, const int &first, const int &second, const int &normOption, const MetricPreparation &object)
- const float getDisimilarity (const VectorXf &others, const MatrixXf &data, const int &index, const int &norm↩ Option, const MetricPreparation &object)
- const float getDisimilarity (const VectorXf &first, const VectorXf &second, const int &firstIndex, const int &secondIndex, const int &normOption, const MetricPreparation &object)
- void getDistanceMatrix (const MatrixXf &data, const int &normOption, const MetricPreparation &object)
- const float getDisimilarity (const VectorXf &first, const VectorXf &second, const int &normOption, const MetricPreparation &object)
- void deleteDistanceMatrix (const int &Row)
- const float getRotation (const std::vector< vector< float > > &streamline, std::vector< float > &rotation)
- void generateGroups (const std::vector< std::vector< int > > &storage)

Variables

- const int & PROCRUSTES_SIZE
- float ** distanceMatrix

5.5.1 Function Documentation

5.5.1.1 void deleteDistanceMatrix (const int & *Row*)

Definition at line 1382 of file Distance.cpp.

```
{
    if(distanceMatrix)
    {
    #pragma omp parallel for schedule(static) num_threads(8)
        for (int i = 0; i < Row; ++i)
        {
            if(distanceMatrix[i])
            {
                delete[] distanceMatrix[i];
                distanceMatrix[i] = NULL;
            }
        }
        delete[] distanceMatrix;
        distanceMatrix = NULL;
    }
}
```

5.5.1.2 void generateGroups (const std::vector< std::vector< int > > & *storage*)

Definition at line 1808 of file Distance.cpp.

```
{
    if(storage.empty())
        return;
    std::ofstream readme("../dataset/Storage",ios::out|ios::app);
    if(!readme)
    {
        std::cout << "Error creating Storage!" << std::endl;
        exit(1);
    }

    readme << std::endl;
    const int& groupSize = storage.size();
    std::vector<int> element;
    for(int i=0;i<groupSize;++i)
    {
        element = storage[i];
        if(element.empty())
            continue;
        for(int j=0;j<element.size();++j)
            readme << element[j] << " ";
        readme << std::endl;
    }
    std::cout << std::endl;
    readme.close();
}
```

5.5.1.3 const float getBMetric (const std::vector< float > & *first,* const std::vector< float > & *second*)

Definition at line 284 of file Distance.cpp.

```
{
    // calculate mean and standard deviation of the two arrays
    float u_a, u_b, sig_a, sig_b, sig_a_inverse, sig_b_inverse,
          summation, sum_inverse, tempDist;
    u_a = firstNorm3[0], u_b = secondNorm3[0];
    sig_a = firstNorm3[1], sig_b = secondNorm3[1];
    sig_a *= sig_a, sig_b *= sig_b;
    if(sig_a<=1.0e-8)
    {
        sig_a = 1.0e-8;
        sig_a_inverse = 1.0e8;
    }
    else
        sig_a_inverse = 1.0/sig_a;
    if(sig_b<=1.0e-8)
    {
        sig_b = 1.0e-8;
        sig_b_inverse = 1.0/sig_b;
    }
    summation = sig_a+sig_b;
    sum_inverse = 1.0/summation;
    tempDist = 0.25*log(0.25*(sig_a*sig_b_inverse
                +sig_b*sig_a_inverse+2))
                + 0.25*(u_a-u_b)*(u_a-u_b)*sum_inverse;
    return tempDist;
}
```

5.5.1.4 const float getBMetric (const MultiVariate & *first,* const MultiVariate & *second*)

Definition at line 321 of file Distance.cpp.

```
{
    Matrix3f firstCov, secondCov, meanCov, meanCovInverse;
    float sqrtInverse, meanCovDet;
    firstCov = centerNormal.covariance;
    secondCov = neighNormal.covariance;
    meanCov = 0.5*(firstCov+secondCov);
    if(meanCov.determinant()>1.0e-8)
    {
        meanCovInverse = static_cast<Matrix3f>(meanCov.inverse());
        meanCovDet = meanCov.determinant();
    }
    else
    {
        meanCovInverse = pseudoInverse(meanCov);
        meanCovDet = 1.0e8;
    }
    float detMulti = sqrt(firstCov.determinant()*secondCov.determinant());
    sqrtInverse = detMulti>1.0e-8?float(1.0)/detMulti:1.0e8;
    Vector3f meanDiff = centerNormal.meanVec-neighNormal.meanVec;
    float tempDist = 0.125*meanDiff.transpose()*meanCovInverse*meanDiff
               +0.2*log(meanCovDet*sqrtInverse);
    return tempDist;
}
```

5.5.1.5 const float getBMetric_3 (const VectorXf & *row,* const int & *size,* const int & *i,* const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 51 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getSequence(row, size, firstNorm3);
    secondNorm3 = rotationSequence[i];
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.5.1.6 const float getBMetric_3 (const VectorXf & *firstRow,* const int & *size,* const VectorXf & *secondRow*)

Definition at line 72 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getSequence(firstRow, size, firstNorm3);
    getSequence(secondRow, size, secondNorm3);
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.5.1.7 const float getBMetric_3 (const int & *first,* const int & *second,* const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 92 of file Distance.cpp.

```
{
    return getBMetric(rotationSequence[first], rotationSequence[second]);
}
```

5.5.1.8 const float getBMetric_6 (const VectorXf & *row*, const int & *size*, const int & *i*, const std::vector< MultiVariate > & *normalMultivariate*)

Definition at line 110 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getNormalMultivariate(row, size, centerNormal);
    neighNormal = normalMultivariate[i];
    return getBMetric(centerNormal, neighNormal);
}
```

5.5.1.9 const float getBMetric_6 (const VectorXf & *firstRow*, const int & *size*, const VectorXf & *secondRow*)

Definition at line 131 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getNormalMultivariate(firstRow, size, centerNormal);
    getNormalMultivariate(secondRow, size, neighNormal);
    return getBMetric(centerNormal, neighNormal);
}
```

5.5.1.10 const float getBMetric_6 (const int & *first*, const int & *second*, const std::vector< MultiVariate > & *normalMultivariate*)

Definition at line 151 of file Distance.cpp.

```
{
    return getBMetric(normalMultivariate[first], normalMultivariate[second]);
}
```

5.5.1.11 const float getBMetric_7 (const VectorXf & *row*, const int & *size*, const int & *i*, const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 169 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getEachFixedSequence(row, size, firstNorm3);
    secondNorm3 = rotationSequence[i];
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.5.1.12 const float getBMetric_7 (const VectorXf & *firstRow*, const int & *size*, const VectorXf & *secondRow*)

Definition at line 190 of file Distance.cpp.

```
{
    std::vector<float> firstNorm3, secondNorm3;
    getEachFixedSequence(firstRow, size, firstNorm3);
    getEachFixedSequence(secondRow, size, secondNorm3);
    return getBMetric(firstNorm3, secondNorm3);
}
```

5.5.1.13 const float getBMetric_7 (const int & *first*, const int & *second*, const std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 210 of file Distance.cpp.

```
{
    return getBMetric(rotationSequence[first], rotationSequence[second]);
}
```

5.5.1.14 const float getBMetric_9 (const VectorXf & *row*, const int & *size*, const int & *i*, const std::vector< **MultiVariate** > & *normalMultivariate*)

Definition at line 228 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getUnnormalizedMultivariate(row, size, centerNormal);
    neighNormal = normalMultivariate[i];
    return getBMetric(centerNormal, neighNormal);
}
```

5.5.1.15 const float getBMetric_9 (const VectorXf & *firstRow*, const int & *size*, const VectorXf & *secondRow*)

Definition at line 249 of file Distance.cpp.

```
{
    MultiVariate centerNormal, neighNormal;
    getUnnormalizedMultivariate(firstRow, size, centerNormal);
    getUnnormalizedMultivariate(secondRow, size, neighNormal);
    return getBMetric(centerNormal, neighNormal);
}
```

5.5.1.16 const float getBMetric_9 (const int & *first*, const int & *second*, const std::vector< **MultiVariate** > & *normalMultivariate*)

Definition at line 269 of file Distance.cpp.

```
{
    return getBMetric(normalMultivariate[first], normalMultivariate[second]);
}
```

5.5.1.17 const float getDisimilarity (const MatrixXf & *data*, const int & *first*, const int & *second*, const int & *normOption*, const **MetricPreparation** & *object*)

Definition at line 964 of file Distance.cpp.

```
{
    return getDisimilarity(data.row(first), data.row(second),
                           first, second, normOption, object);
}
```

5.5.1.18 const float getDisimilarity (const VectorXf & *others,* const MatrixXf & *data,* const int & *index,* const int & *normOption,* const MetricPreparation & *object*)

Definition at line 985 of file Distance.cpp.

```
{
    float length;
    switch(normOption)
    {
    case 0: // Euclidean distance, d_E
    case 1: // Fraction norm, d_F
    case 2: // geometric similarity measure, d_G
    case 5:
    case 8:
    case 11:
        length = getNorm(others, data.row(index),index,normOption,
                    object.pairwise, object.pairwiseNorm);
        break;

    case 3:
        length = getBMetric_3(others, others.size()/3-2, index,
                              object.rotationSequence);
        break;

    case 4:
        length = abs(object.rotation[index]-
                 getRotation(others, others.size()/3-2));
        break;

    case 6:
        length = getBMetric_6(others, others.size()/3-1, index,
                              object.normalMultivariate);
        break;

    case 7:
        length = getBMetric_7(others, others.size()/3-1, index,
                              object.rotationSequence);
        break;

    case 9:
        length = getBMetric_9(others, others.size()/3-1, index,
                              object.normalMultivariate);
        break;

    case 10:
        length = getMetric_10(others, others.size()/3, index,
                              object.unitLength);
        break;

    case 12:    // the MCP distance, i.e., d_M
        length = getMetric_MOP(others, data.row(index));
        break;

    case 13:    // the Hausdorff distance, i.e., d_H
        length = getMetric_Hausdorff(others, data.row(index));
        break;

    /* signature-based similarity metric with chi-squared test combined with mean-closest */
    case 14:    // the signature-based similarity, i.e., d_S
        length = getSignatureMetric(others,data.row(index),object.pairwise[index]);
        break;

    /* adapted Procrustes distance */
    case 15:    // the Procrustes distance, i.e., d_P
        //length = getProcrustesMetric(others, data.row(index));
        length = std::min(getProcrustesMetricSegment(others, data.row(index)),
                getProcrustesMetricSegment(data.row(index), others));
        break;

    case 16:
        length = getEntropyMetric(object.pairwise[index], others);
        break;

    case 17:    // the time-based MCP, i.e., d_T
        length = getPathline_MCP(others, data.row(index));
        break;

    default:
        exit(1);
        break;
    }

    return length;
}
```

5.5.1.19 const float getDisimilarity (const VectorXf & *first*, const VectorXf & *second*, const int & *firstIndex*, const int & *secondIndex*, const int & *normOption*, const MetricPreparation & *object*)

Definition at line 1082 of file Distance.cpp.

```
{
    float length;
    switch(normOption)
    {
    case 0: // Euclidean distance, d_E
    case 1: // Fraction norm, d_F
    case 2: // Geometric similarity, d_G
    case 5:
    case 8:
    case 11:
        length = getNorm(first, second,firstIndex,secondIndex, normOption,
                    object.pairwise, object.pairwiseNorm);
        break;

    case 3:
        length = getBMetric_3(firstIndex, secondIndex,
                    object.rotationSequence);
        break;

    case 4:
        length = abs(object.rotation[firstIndex]-
                    object.rotation[secondIndex]);
        break;

    case 6:
        length = getBMetric_6(firstIndex, secondIndex,
                    object.normalMultivariate);
        break;

    case 7:
        length = getBMetric_7(firstIndex, secondIndex,
                    object.rotationSequence);
        break;

    case 9:
        length = getBMetric_9(firstIndex, secondIndex,
                    object.normalMultivariate);
        break;

    case 10:
        length = getMetric_10(firstIndex, secondIndex,
                    object.unitLength);
        break;

    case 12:    // the MCP distance, d_M
        length = getMetric_MOP(first, second);
        break;

    case 13:    // the Hausdorff distance, d_H
        length = getMetric_Hausdorff(first, second);
        break;

    case 14:    //  the signature-based distance, d_S
        length = getSignatureMetric(first,second,object.pairwise[firstIndex],object.
pairwise[secondIndex]);
        break;

    case 15:    // the Procrutes distance, d_P
        //length = getProcrustesMetric(first, second);
        length = std::min(getProcrustesMetricSegment(first,second),
getProcrustesMetricSegment(second,first));
        break;

    case 16:
        length = getEntropyMetric(object.pairwise[firstIndex], object.pairwise[secondIndex]
);
        break;

    case 17:    // the time-based MCP, d_T
        length = getPathline_MCP(first, second);
        break;

    default:
        exit(1);
        break;
    }
    return length;
}
```

5.5.1.20 const float getDisimilarity (const VectorXf & *first,* const VectorXf & *second,* const int & *normOption,* const MetricPreparation & *object*)

Definition at line 1174 of file Distance.cpp.

```
{
    float length;
    switch(normOption)
    {
    case 0: // Euclidean distance, d_E
    case 1: // Fraction norm, d_F
    case 2: // Geometric similarity, d_G
    case 5:
    case 8:
    case 11:
        length = getNorm(first, second, normOption);
        break;

    case 3:
        length = getBMetric_3(first, first.size()/3-2, second);
        break;

    case 4:
        length = abs(getRotation(first, first.size()/3-2)-getRotation(second, second.size()/3-2));
        break;

    case 6:
        length = getBMetric_6(first, first.size()/3-1,second);
        break;

    case 7:
        length = getBMetric_7(first, first.size()/3-1, second);
        break;

    case 9:
        length = getBMetric_9(first, first.size()/3-1, second);
        break;

    case 10:
        length = getBMetric_9(first, first.size()/3, second);
        break;

    case 12:    // the MCP distance, d_M
        length = getMetric_MOP(first, second);
        break;

    case 13:    // the Hausdorff distance, d_H
        length = getMetric_Hausdorff(first, second);
        break;

    /* signature-based similarity metric with chi-squared test combined with mean-closest */
    case 14:    // the signature-based similarity, d_S
        length = getSignatureMetric(first, second);
        break;

    /* adapted Procrustes distance */
    case 15:    // the Procrustes distance, d_P
        //length = getProcrustesMetric(first, second);
        length = getProcrustesMetricSegment(first,second);
        break;

    case 16:
        length = getEntropyMetric(first, second);
        break;

    case 17:    // the time-based MCP, d_T
        length = getPathline_MCP(first, second);
        break;

    default:
        exit(1);
        break;
    }

    return length;
}
```

5.5.1.21 void getDistanceMatrix (const MatrixXf & *data,* const int & *normOption,* const MetricPreparation & *object*)

Definition at line 1348 of file Distance.cpp.

```
{
    const int& Row = data.rows();
    distanceMatrix = new float*[Row];

    // assign the distance matrix
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        distanceMatrix[i] = new float[Row];
        for (int j = 0; j < Row; ++j)
        {
            /* don't wish to waste computation on diagonal element */
            if(i==j)
                distanceMatrix[i][j] = 0.0;
            else
                distanceMatrix[i][j] = getDisimilarity(data, i, j, normOption, object);
        }
    }
    // help check whether they already been assigned and whether they are symmetric or not
    std::cout << "Distance between 215 and 132 is " << distanceMatrix[215][132] << std::endl;
    std::cout << "Distance between 132 and 215 is " << distanceMatrix[132][215] << std::endl;

    std::cout << "Finished computing distance matrix!" << std::endl;
}
```

5.5.1.22 const float getEntropyMetric (const std::vector< float > & *firstEntropy*, const std::vector< float > & *secondEntropy*)

Definition at line 1845 of file Distance.cpp.

```
{
    assert(firstEntropy.size()==2);
    assert(secondEntropy.size()==2);

    float first = firstEntropy[0]-secondEntropy[0];
    float second = firstEntropy[1]-secondEntropy[1];

    return sqrt(first*first+second*second);
}
```

5.5.1.23 const float getEntropyMetric (const std::vector< float > & *firstEntropy*, const Eigen::VectorXf & *array*)

Definition at line 1868 of file Distance.cpp.

```
{
    assert(firstEntropy.size()==2);

    std::vector<float> secondEntropy;

    getLinearAngularEntropy(array, BUNDLE_SIZE, secondEntropy);

    return getEntropyMetric(firstEntropy, secondEntropy);
}
```

5.5.1.24 const float getEntropyMetric (const Eigen::VectorXf & *first*, const Eigen::VectorXf & *second*)

Definition at line 1890 of file Distance.cpp.

```
{

    std::vector<float> firstEntropy, secondEntropy;

    getLinearAngularEntropy(first, BUNDLE_SIZE, firstEntropy);
    getLinearAngularEntropy(second, BUNDLE_SIZE, secondEntropy);

    return getEntropyMetric(firstEntropy, secondEntropy);
}
```

5.5.1.25 const float getMetric_10 (const VectorXf & *centroid,* const int & *size,* const int & *index,* const std::vector< VectorXf > & *unitLength*)

Definition at line 358 of file Distance.cpp.

```
{
    const VectorXf& x = unitLength[index];
    VectorXf y(size*3);
    getUnitDirection_byEach(centroid,size,y);

    float length = x.dot(y)/x.size();
    length = min(1.0,(double)length);
    length = max(-1.0,(double)length);
    length = acos(length);
    return length;
}
```

5.5.1.26 const float getMetric_10 (const VectorXf & *firstRow,* const int & *size,* const VectorXf & *secondRow*)

Definition at line 383 of file Distance.cpp.

```
{
    VectorXf x(size*3), y(size*3);
    getUnitDirection_byEach(firstRow,size,x);
    getUnitDirection_byEach(secondRow,size,y);

    float length = x.dot(y)/x.size();
    length = min(1.0,(double)length);
    length = max(-1.0,(double)length);
    length = acos(length);
    return length;
}
```

5.5.1.27 const float getMetric_10 (const int & *first,* const int & *second,* const std::vector< VectorXf > & *unitLength*)

Definition at line 407 of file Distance.cpp.

```
{
    const VectorXf& x = unitLength[first];
    const VectorXf& y = unitLength[second];
    float length = x.dot(y)/x.size();
    length = min(1.0,(double)length);
    length = max(-1.0,(double)length);
    length = acos(length);
    return length;
}
```

5.5.1.28 const float getMetric_Hausdorff (const VectorXf & *first,* const VectorXf & *second*)

Definition at line 1305 of file Distance.cpp.

```
{
    // the max of first to second
    const int& vNum = first.size()/3;
    float result, f_to_s=-1.0, s_to_f=-1.0;
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(first(3*i),first(3*i+1),first(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(second(3*j),second(3*j+1),second(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        s_to_f=std::max(s_to_f, minDist);
    }

    // the max of second to first
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(second(3*i),second(3*i+1),second(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(first(3*j),first(3*j+1),first(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        f_to_s=std::max(f_to_s, minDist);
    }

    // max of the max
    result = std::max(f_to_s, s_to_f);
    return result;
}
```

5.5.1.29 const float getMetric_MOP (const VectorXf & *first*, const VectorXf & *second*)

Definition at line 1258 of file Distance.cpp.

```
{
    // The MCP of first to second
    const int& vNum = first.size()/3;
    float result, f_to_s, s_to_f;
    float summation = 0;
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(first(3*i),first(3*i+1),first(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(second(3*j),second(3*j+1),second(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        summation+=minDist;
    }
    s_to_f = summation/vNum;

    // The MCP of second to first
    summation = 0;
    for(int i=0;i<vNum;++i)
    {
        float minDist = FLT_MAX;
        Vector3f m_i = Vector3f(second(3*i),second(3*i+1),second(3*i+2));
        for(int j=0;j<vNum;++j)
        {
            Vector3f n_j = Vector3f(first(3*j),first(3*j+1),first(3*j+2));
            minDist = std::min((m_i-n_j).norm(),minDist);
        }
        summation+=minDist;
    }
    f_to_s = summation/vNum;

    // get the average of that
    result = (f_to_s+s_to_f)/2.0;
    return result;
}
```

5.5.1.30 const float getNorm (const Eigen::VectorXf & *centroid*, const Eigen::VectorXf & *r2*, const int & *index*, const int & *normOption*, const std::vector< std::vector< float > > & *object*, const std::vector< std::vector< float > > & *objectNorm*)

Definition at line 432 of file Distance.cpp.

```
{
    assert(centroid.size()==r2.size());
    float length = 0.0;
    switch(normOption)
    {
    case 0: // Euclidean distance
    default:
        length = (centroid-r2).norm();
        break;

    case 11:    // the norm 11
        {
            float dotPro = centroid.dot(r2);
            float firstNorm = centroid.norm();
            float secondNorm = r2.norm();
            float firstInverse, secondInverse;
            if(firstNorm<1.0e-8)
                firstInverse = 1.0e8;
            else
                firstInverse = 1.0/firstNorm;
            if(secondNorm<1.0e-8)
                secondInverse = 1.0e8;
            else
                secondInverse = 1.0/secondNorm;
            dotPro = dotPro*firstInverse*secondInverse;
            dotPro = std::max(dotPro, float(-1.0));
            dotPro = std::min(dotPro, float(1.0));
            length = acos(dotPro)/M_PI;
        }
        break;

    case 1:  /* fraction norm by high-dimensional feature-space */
        {
            for (int i = 0; i < centroid.size(); ++i)
            {
                length += pow(abs(centroid(i)-r2(i)),0.5);
            }
            length = pow(length,2.0);
        }
        break;

    case 2: /* mean value of dot product value, which means it's rotational invariant, or d_G */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, result;

            std::vector<float> centroidWise;
            std::vector<float> centroidWiseNorm;
            getPairWise_byEach(centroid, pointNum, centroidWise, centroidWiseNorm);

            const std::vector<float>& i_Pairwise = pairwise[index];
            const std::vector<float>& i_PairNorm = objectNorm[index];

            Vector3f left, right;
            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = centroidWiseNorm[i];
                rightNorm = i_PairNorm[i];
                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << centroidWise[3*i], centroidWise[3*i+1], centroidWise[3*i+2];
                    right << i_Pairwise[3*i],i_Pairwise[3*i+1],i_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
        }
        break;

    case 5: /* rotational invariant line-wise acos angle with normal direction for
               measuring whether counterclockwise or clockwise orientation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, normalDot, result;
            Vector3f left, right, normal;

            std::vector<float> centroidWise;
            std::vector<float> centroidNorm;
            getPairWise_byEach(centroid, pointNum, centroidWise, centroidNorm);
            const std::vector<float>& i_Pairwise = pairwise[index];
            const std::vector<float>& i_PairNorm = objectNorm[index];

            left << /*centroid(3)-centroid(0),centroid(4)-centroid(1),centroid(5)-centroid(2)*/
                    centroidWise[0], centroidWise[1], centroidWise[2];
            right << /*r2(3)-r2(0),r2(4)-r2(1),r2(5)-r2(2)*/
                    i_Pairwise[0], i_Pairwise[1], i_Pairwise[2];
            const Vector3f& Normal = left.cross(right);

            for (int i = 0; i < pointNum; ++i)
            {

                leftNorm = centroidNorm[i];
                rightNorm = i_PairNorm[i];
                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << centroidWise[3*i], centroidWise[3*i+1], centroidWise[3*i+2];
                    right << i_Pairwise[3*i],i_Pairwise[3*i+1],i_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    normal = left.cross(right);
                    normalDot = Normal.dot(normal);
                    if(normalDot<0)
                        length+=-acos(result);
                    else
                        length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
            length = abs(length);
        }
        break;

    case 8: /* distance metric defined as mean * standard deviation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, stdevia = 0.0, angle, result;
            Vector3f left, right;

            std::vector<float> centroidWise;
            std::vector<float> centroidNorm;
            getPairWise_byEach(centroid, pointNum, centroidWise, centroidNorm);

            const std::vector<float>& i_Pairwise = pairwise[index];
            const std::vector<float>& i_PairNorm = objectNorm[index];

            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = centroidNorm[i];
                rightNorm = i_PairNorm[i];

                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << centroidWise[3*i], centroidWise[3*i+1], centroidWise[3*i+2];
                    right << i_Pairwise[3*i],i_Pairwise[3*i+1],i_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    angle = acos(result);
                    length+=angle;
                    stdevia+=angle*angle;
                }
                else
                {
                    angle=M_PI;
                    length+=angle;
                    stdevia+=angle*angle;
                }
            }
            length /= pointNum;
            stdevia = stdevia/pointNum-length*length;
            if(stdevia>0)
                stdevia = sqrt(stdevia/pointNum-length*length);
            else
                stdevia = 1.0e-4;
        }
        break;
    }

    return length;
}
```

5.5.1.31 const float getNorm (const VectorXf & *centroid*, const VectorXf & *r2*, const int & *firstIndex*, const int & *secondIndex*, const int & *normOption*, const std::vector< std::vector< float > > & *object*, const std::vector< std::vector< float > > & *objectNorm*)

Definition at line 620 of file Distance.cpp.

```
{
    assert(centroid.size()==r2.size());
    float length = 0.0;
    switch(normOption)
    {
    case 0:
    default:
        length = (centroid-r2).norm();
        break;

    case 1:  /* fraction norm by high-dimensional feature-space, or d_F */
        {
            for (int i = 0; i < centroid.size(); ++i)
            {
                length += pow(abs(centroid(i)-r2(i)),0.5);
            }
            length = pow(length,2.0);
        }
        break;

    case 11:
        {
            float dotPro = centroid.dot(r2);
            float firstNorm = centroid.norm();
            float secondNorm = r2.norm();
            float firstInverse, secondInverse;
            if(firstNorm<1.0e-8)
                firstInverse = 1.0e8;
            else
                firstInverse = 1.0/firstNorm;
            if(secondNorm<1.0e-8)
                secondInverse = 1.0e8;
            else
                secondInverse = 1.0/secondNorm;
            dotPro = dotPro*firstInverse*secondInverse;
            dotPro = std::max(dotPro, float(-1.0));
            dotPro = std::min(dotPro, float(1.0));
            length = acos(dotPro)/M_PI;
        }
        break;

    case 2: /* mean value of dot product value, which means it's rotational invariant, or d_G */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, result;

            const std::vector<float>& i_Pairwise = pairwise[firstIndex];
            const std::vector<float>& j_Pairwise = pairwise[secondIndex];

            const std::vector<float>& i_PairNorm = objectNorm[firstIndex];
            const std::vector<float>& j_PairNorm = objectNorm[secondIndex];

            Vector3f left, right;
            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = i_PairNorm[i];
                rightNorm = j_PairNorm[i];
                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << i_Pairwise[3*i], i_Pairwise[3*i+1], i_Pairwise[3*i+2];
                    right << j_Pairwise[3*i],j_Pairwise[3*i+1],j_Pairwise[3*i+2];
                    result = left.dot(right)/*leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
        }
        break;

    case 5: /* rotational invariant line-wise acos angle with normal direction for
               measuring whether counterclockwise or clockwise orientation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, normalDot, result;
            Vector3f left, right, normal;

            const std::vector<float>& i_Pairwise = pairwise[firstIndex];
            const std::vector<float>& j_Pairwise = pairwise[secondIndex];

            const std::vector<float>& i_PairNorm = objectNorm[firstIndex];
            const std::vector<float>& j_PairNorm = objectNorm[secondIndex];

            left << /*centroid(3)-centroid(0),centroid(4)-centroid(1),centroid(5)-centroid(2)*/
                    i_Pairwise[0], i_Pairwise[1], i_Pairwise[2];
            right << /*r2(3)-r2(0),r2(4)-r2(1),r2(5)-r2(2)*/
                    j_Pairwise[0], j_Pairwise[1], j_Pairwise[2];
            const Vector3f& Normal = left.cross(right);

            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = i_PairNorm[i];
                rightNorm = j_PairNorm[i];

                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << i_Pairwise[3*i], i_Pairwise[3*i+1], i_Pairwise[3*i+2];
                    right << j_Pairwise[3*i],j_Pairwise[3*i+1],j_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    normal = left.cross(right);
                    normalDot = Normal.dot(normal);
                    if(normalDot<0)
                        length+=-acos(result);
                    else
                        length+=acos(result);
                }
                else
                    length+=M_PI;
            }
            length /= pointNum;
            length = abs(length);
        }
        break;

    case 8: /* distance metric defined as mean * standard deviation */
        {
            const int& pointNum = centroid.size()/3-1;
            float dotValue, leftNorm, rightNorm, stdevia = 0.0, angle, result;
            Vector3f left, right;

            const std::vector<float>& i_Pairwise = pairwise[firstIndex];
            const std::vector<float>& j_Pairwise = pairwise[secondIndex];

            const std::vector<float>& i_PairNorm = objectNorm[firstIndex];
            const std::vector<float>& j_PairNorm = objectNorm[secondIndex];

            for (int i = 0; i < pointNum; ++i)
            {
                leftNorm = i_PairNorm[i];
                rightNorm = j_PairNorm[i];

                if(leftNorm >= 1.0e-8 && rightNorm >= 1.0e-8)
                {
                    left << i_Pairwise[3*i], i_Pairwise[3*i+1], i_Pairwise[3*i+2];
                    right << j_Pairwise[3*i],j_Pairwise[3*i+1],j_Pairwise[3*i+2];
                    result = left.dot(right)/*/leftNorm/rightNorm*/;
                    result = min(1.0,(double)result);
                    result = max(-1.0,(double)result);
                    angle = acos(result);
                    length+=angle;
                    stdevia+=angle*angle;
                }
                else
                {
                    angle=M_PI;
                    length+=angle;
                    stdevia+=angle*angle;
                }
            }
            length /= pointNum;
            stdevia = stdevia/pointNum-length*length;
            if(stdevia>0)
                stdevia = sqrt(stdevia/pointNum-length*length);
            else
                stdevia = 1.0e-4;
        }
        break;

    }

    return length;
}
```

5.5.1.32 const float getNorm (const VectorXf & *r1*, const VectorXf & *r2*, const int & *normOption*)

5.5.1.33 const float getPathline_MCP (const Eigen::VectorXf & *first*, const Eigen::VectorXf & *second*)

Definition at line 1910 of file Distance.cpp.

```
{
    /* preset the initial time step is 0, then 1, 2, ... as long as it will be normalized */
    const int& t_M = first.size()/3-1;
    float dist = 0.0, a, b, c;
    Eigen::Vector3f temp, another, diff;
    for(int i=0; i<t_M; ++i)
    {
        temp=Eigen::Vector3f(first(i*3)-second(i*3), first(3*i+1)-second(3*i+1), first(3*i+2)-second(3*i+2));
        another=Eigen::Vector3f(first(i*3+3)-second(i*3+3), first(3*i+4)-second(3*i+4), first(3*i+5)-second(3*i+5));
        diff=another-temp;

        a=temp.transpose()*temp;
        b=temp.transpose()*diff;
        c=diff.transpose()*diff;

        dist+=get_calculus(a, b, c);
    }
    return dist/t_M;
}
```

5.5.1.34 const float getProcrustesMetric (const Eigen::VectorXf & *first*, const Eigen::VectorXf & *second*)

Definition at line 1557 of file Distance.cpp.

```
{
    assert(first.size()==second.size());

    const int& vertexCount = first.size()/3;

    const int& vertexChanged = vertexCount-2*(PROCRUSTES_SIZE/2);
    const int& newSize = 3*vertexChanged;

    /* assign the segment list */
    Eigen::MatrixXf firstSegment(PROCRUSTES_SIZE,3), secondSegment(PROCRUSTES_SIZE,3), X0;

    int location, rightIndex;

    Eigen::Vector3f first_average, second_average, tempPoint;

    /* A is SVD target, rotation is optimal rotation matrix, and secondPrime is P' after superimposition */
    Eigen::MatrixXf A, rotation, secondPrime = Eigen::MatrixXf(PROCRUSTES_SIZE,3);

    float optimalScaling, traceA, pointDist;

    float result = 0.0;

    /* for all points, assign to them a point set with size of PROCRUSTES_SIZE neighboring points */
    for(int i=0;i<vertexChanged;++i)
    {
        rightIndex = i+PROCRUSTES_SIZE;

        first_average = second_average = Eigen::VectorXf::Zero(3);

        /* get the point set of neighboring 7 points and average */
        for(int j=i;j<rightIndex;++j)
        {
            location = j-i;
            for(int k=0;k<3;++k)
            {
                firstSegment(location,k)=first(3*j+k);
                secondSegment(location,k)=second(3*j+k);
            }

            first_average+=firstSegment.row(location);
            second_average+=secondSegment.row(location);
        }

        first_average/=PROCRUSTES_SIZE;
        second_average/=PROCRUSTES_SIZE;

        /* reserve the matrix */
        X0 = firstSegment;

        /* centralization for the point set */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            firstSegment.row(j) = firstSegment.row(j)-first_average.transpose();
            secondSegment.row(j) = secondSegment.row(j)-second_average.transpose();
        }

        /* get ssqX and ssqY */
        float ssqX = (firstSegment.cwiseProduct(firstSegment)).sum();
        float ssqY = (secondSegment.cwiseProduct(secondSegment)).sum();

        /* check whether negative or not */
        assert(ssqX > 0 && ssqY > 0);

        ssqX = sqrt(ssqX);
        ssqY = sqrt(ssqY);

        /* scaling for the point set */
        firstSegment/=ssqX;
        secondSegment/=ssqY;

        /* get the optimal rotational matrix by othogonal Procrutes analysis */
        A = firstSegment.transpose()*secondSegment;

        /* perform SVD on A */
        JacobiSVD<MatrixXf> svd(A, ComputeThinU | ComputeThinV);

        /* get the optimal 3D rotation */
        rotation = svd.matrixV()*(svd.matrixU().transpose());

        /* get trace for singular value matrix */
        traceA = svd.singularValues().sum();

        /* get optimal scaling */
        optimalScaling = traceA*ssqX/ssqY;

        /* preset the average to the P' */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
            secondPrime.row(j) = first_average;

        /* get P' in superimposed space */
        secondPrime = ssqX*traceA*secondSegment*rotation+secondPrime;

        /* compute the distance and store them in the std::vector<float> */
        pointDist = 0.0;
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            tempPoint = X0.row(j)-secondPrime.row(j);
            pointDist+= tempPoint.transpose()*tempPoint;
        }

        /* get the average of P(x,y')^2 */

        // either by computing the matrix
        //result+=pointDist;

        // or directly using trace of the matrix
        float requiredD = 1.0-traceA*traceA;
        result+=requiredD*requiredD;
    }
    return result/vertexChanged;
}
```

5.5.1.35 const float getProcrustesMetricSegment (const Eigen::VectorXf & *first,* const Eigen::VectorXf & *second*)

Definition at line 1681 of file Distance.cpp.

```
{
    assert(first.size()==second.size());

    const int& vertexCount = first.size()/3;

    const int& vertexChanged = vertexCount/PROCRUSTES_SIZE;
    const int& newSize = 3*vertexChanged;

    /* assign the segment list */
    Eigen::MatrixXf firstSegment(PROCRUSTES_SIZE,3), secondSegment(PROCRUSTES_SIZE,3), X0;

    int location, rightIndex;

    Eigen::Vector3f first_average, second_average, tempPoint;

    /* A is SVD target, rotation is optimal rotation matrix, and secondPrime is P' after superimposition */
    Eigen::MatrixXf A, rotation, secondPrime = Eigen::MatrixXf(PROCRUSTES_SIZE,3);

    float optimalScaling, traceA, pointDist;

    float result = 0.0;

    int effective = 0;
    /* for all points, assign to them a point set with size of PROCRUSTES_SIZE neighboring points */
    for(int i=0;i<vertexChanged;++i)
    {
        rightIndex = PROCRUSTES_SIZE*i+PROCRUSTES_SIZE;

        first_average = second_average = Eigen::VectorXf::Zero(3);

        /* get the point set of neighboring 7 points and average */
        for(int j=PROCRUSTES_SIZE*i;j<rightIndex;++j)
        {
            location = j-PROCRUSTES_SIZE*i;
            for(int k=0;k<3;++k)
            {
                firstSegment(location,k)=first(3*j+k);
                secondSegment(location,k)=second(3*j+k);
            }

            first_average+=firstSegment.row(location);
            second_average+=secondSegment.row(location);
        }

        first_average/=PROCRUSTES_SIZE;
        second_average/=PROCRUSTES_SIZE;

        /* reserve the matrix */
        X0 = firstSegment;

        /* centralization for the point set */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            firstSegment.row(j) = firstSegment.row(j)-first_average.transpose();
            secondSegment.row(j) = secondSegment.row(j)-second_average.transpose();
        }

        /* get ssqX and ssqY */
        float ssqX = (firstSegment.cwiseProduct(firstSegment)).sum();
        float ssqY = (secondSegment.cwiseProduct(secondSegment)).sum();

        /* check whether negative or not */
        assert(ssqX > 0 && ssqY > 0);

        if(ssqX<1.0e-14 || ssqY<1.0e-14)
            continue;

        ssqX = sqrt(ssqX);
        ssqY = sqrt(ssqY);

        if(ssqX<1.0e-8 || ssqY<1.0e-8)
            continue;

        /* scaling for the point set */
        firstSegment/=ssqX;
        secondSegment/=ssqY;

        /* get the optimal rotational matrix by othogonal Procrutes analysis */
        A = firstSegment.transpose()*secondSegment;

        /* perform SVD on A */
        JacobiSVD<MatrixXf> svd(A, ComputeThinU | ComputeThinV);

        /* get the optimal 3D rotation */
        rotation = svd.matrixV()*(svd.matrixU().transpose());

        /* get trace for singular value matrix */
        traceA = svd.singularValues().sum();

        /* get optimal scaling */
        optimalScaling = traceA*ssqX/ssqY;

        /* preset the average to the P' */
        for(int j=0;j<PROCRUSTES_SIZE;++j)
            secondPrime.row(j) = first_average;

        /* get P' in superimposed space */
        secondPrime = ssqX*traceA*secondSegment*rotation+secondPrime;

        /* compute the distance and store them in the std::vector<float> */
        pointDist = 0.0;
        for(int j=0;j<PROCRUSTES_SIZE;++j)
        {
            tempPoint = X0.row(j)-secondPrime.row(j);
            pointDist+= tempPoint.transpose()*tempPoint;
        }
        /* get the average of P(x,y')^2 */
        result+=pointDist;
        ++effective;
    }

    if(effective==0)
    {
        return 1.0e-8;
    }
    else
        return result/effective;
}
```

5.5.1.36 const float getRotation (const std::vector< vector< float > > & *streamline*, std::vector< float > & *rotation*)

Definition at line 1406 of file Distance.cpp.

```
{
    if(streamline.empty())
        return -1;
    float result = 0, eachSum;
    const int& size = streamline.size();
    rotation = std::vector<float>(size);
    std::vector<float> eachLine;
    Eigen::Vector3f first, second;
    int lineSize;
    for(int i=0;i<size;++i)
    {
        eachSum = 0;
        eachLine = streamline[i];
        lineSize = eachLine.size()/3-2;
        // calculate the summation of discrete curvatures
        for(int j=0;j<lineSize;++j)
        {
            first<<eachLine[3*j+3]-eachLine[3*j],eachLine[3*j+4]-eachLine[3*j+1],eachLine[3*j+5]-eachLine[3*j+2];
            second<<eachLine[3*j+6]-eachLine[3*j+3],eachLine[3*j+7]-eachLine[3*j+4],eachLine[3*j+8]-eachLine[3*j+5];

            float firstNorm = first.norm(), secondNorm = second.norm();
            if(firstNorm>=1.0e-8 && secondNorm>=1.0e-8)
            {
                float angle = first.dot(second)/firstNorm/secondNorm;
                angle = std::max(angle,float(-1.0));
                angle = std::min(angle,float(1.0));
                eachSum+=acos(angle);
            }
        }
        // get the mean of discrete curvatures
        rotation[i]=eachSum;
        result+=eachSum;
    }
    result/=size;
    return result;
}
```

5.5.1.37 const float getSignatureMetric (const Eigen::VectorXf & *firstArray,* const Eigen::VectorXf & *secondArray,* const std::vector< float > & *firstHist,* const std::vector< float > & *secondHist*)

Definition at line 1454 of file Distance.cpp.

```
{
    /* would choose alpha = 0.5, and 10% of subset vertices for mean_dist */
    const float& Alpha = 0.5;
    const int& SUBSET = 10;

    /* assert whether the size is the same */
    assert(firstArray.size()==secondArray.size());
    assert(firstHist.size()==secondHist.size());

    const int& histSize = firstHist.size();
    const int& vertexCount = firstArray.size()/3;
    const int& size = vertexCount/SUBSET+1;

    Eigen::VectorXf firstSubset(3*size), secondSubset(3*size);

    /* get mean_dist between two sampled subsets */
    int tempPos = 0;
    for(int i=0;i<vertexCount;i+=SUBSET)
    {
        for(int j=0;j<3;++j)
        {
            firstSubset(3*tempPos+j)=firstArray(3*i+j);
            secondSubset(3*tempPos+j)=secondArray(3*i+j);
        }
        ++tempPos;
    }

    /* get mean_dist */
    float result = getMetric_MOP(firstSubset, secondSubset);

    float chi_test = 0.0, histDiff, histSum;

    /* get chi_test for two histograms */
    for(int i=0;i<histSize;++i)
    {
        histDiff = firstHist[i]-secondHist[i];
        histSum = firstHist[i]+secondHist[i];
        /* check numerical error */
        if(histSum<1.0e-8)
            continue;

        chi_test+= histDiff*histDiff/histSum;
    }

    /* get combined distance */
    result = (1-Alpha)*chi_test + Alpha*result;

    return result;
}
```

5.5.1.38 const float getSignatureMetric (const Eigen::VectorXf & *centroid,* const Eigen::VectorXf & *first,* const std::vector< float > & *firstHist*)

Definition at line 1517 of file Distance.cpp.

```
{
    std::vector<float> centroidHist;
    /* get the bin-based histogram for signature */
    getSignatureHist(centroid, BIN_SIZE, centroidHist);

    return getSignatureMetric(centroid,first,centroidHist,firstHist);
}
```

5.5.1.39 const float getSignatureMetric (const Eigen::VectorXf & *first*, const Eigen::VectorXf & *second*)

Definition at line 1536 of file Distance.cpp.

```
{
    std::vector<float> firstHist, secondHist;
    /* get the bin-based histogram for signature */
    getSignatureHist(first, BIN_SIZE, firstHist);
    getSignatureHist(second, BIN_SIZE, secondHist);

    return getSignatureMetric(first,second,firstHist,secondHist);
}
```

5.5.2 Variable Documentation

5.5.2.1 float** distanceMatrix

Definition at line 39 of file Distance.cpp.

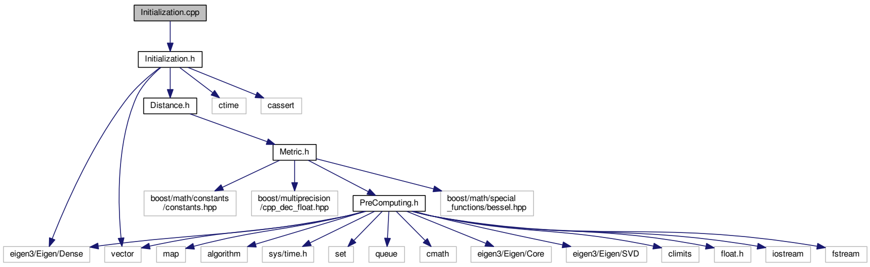
5.5.2.2 const int& PROCRUSTES_SIZE

Definition at line 33 of file Distance.cpp.

5.6 Initialization.cpp File Reference

`#include "Initialization.h"`

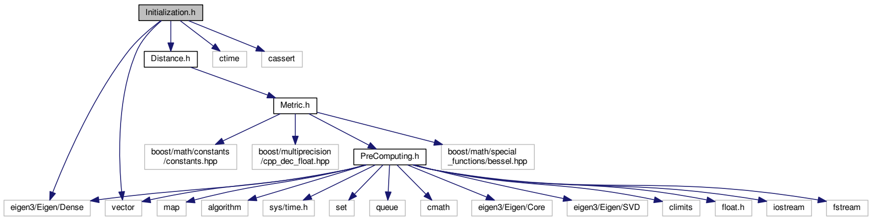
Include dependency graph for Initialization.cpp:



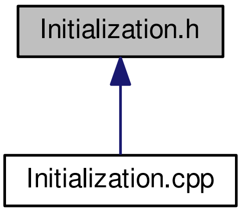
5.7 Initialization.h File Reference

```
#include <eigen3/Eigen/Dense>
#include <vector>
#include <ctime>
#include <cassert>
#include "Distance.h"
```

Include dependency graph for Initialization.h:



This graph shows which files directly or indirectly include this file:



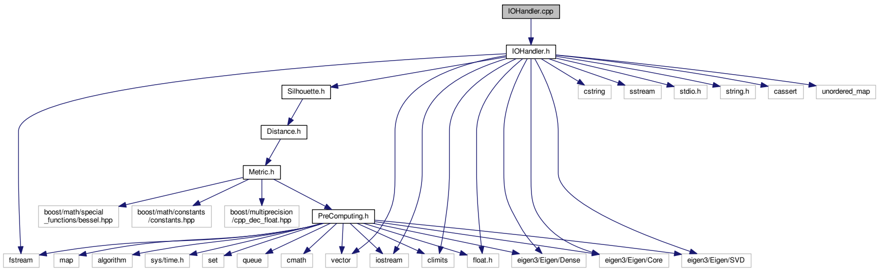
Classes

- class Initialization

5.8 IOHandler.cpp File Reference

```
#include "IOHandler.h"
```

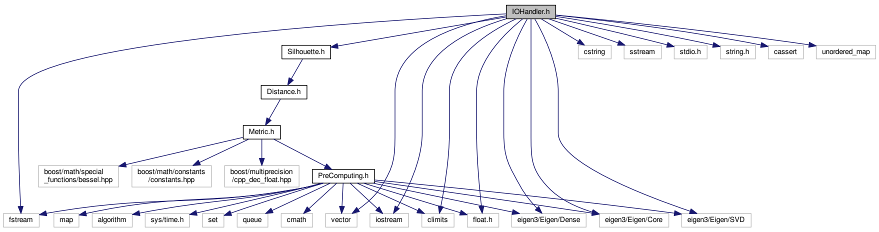
Include dependency graph for IOHandler.cpp:



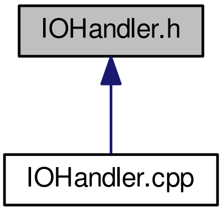
5.9 IOHandler.h File Reference

```
#include <fstream>
#include <vector>
#include <iostream>
#include <cstring>
#include <sstream>
#include <stdio.h>
#include <string.h>
#include <climits>
#include <cassert>
#include <float.h>
#include <unordered_map>
#include <eigen3/Eigen/Dense>
#include <eigen3/Eigen/Core>
#include <eigen3/Eigen/SVD>
#include "Silhouette.h"
```

Include dependency graph for IOHandler.h:



This graph shows which files directly or indirectly include this file:



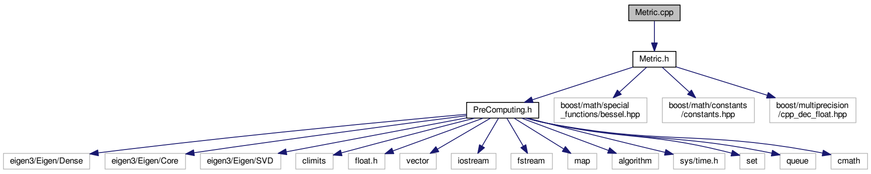
Classes

- struct ExtractedLine
- struct MeanLine
- struct StringQuery
- struct FeatureLine
- class IOHandler

5.10 Metric.cpp File Reference

```
#include "Metric.h"
```

Include dependency graph for Metric.cpp:



Functions

- void computeMeanRotation (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< float > &rotation)
- void getRotationSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std::vector< float > > &rotationSequence)
- void getNormalSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< MultiVariate > &normalMultivariate)
- void getFixedSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &rotationSequence)
- void getUnnormalizedSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std↩ ::vector< MultiVariate > &normalMultivariate)
- void getUnitDirection (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< VectorXf > &unitLength)
- void computePairWise (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &pairwise, std::vector< std::vector< float > > &pairwiseNorm)
- void getSignatureBin (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &pairwise)
- void getBundleEntropy (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &pairwise)
- const float get_calculus (const float &a, const float &b, const float &c)

Variables

- const int & BIN_SIZE = 20
- const int & BUNDLE_SIZE = 20

5.10.1 Function Documentation

5.10.1.1 void computeMeanRotation (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< float > & *rotation*)

Definition at line 30 of file Metric.cpp.

```
{
    rotation = std::vector<float>(Row, 0.0);
    const int& pointNum = Column/3-2;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        rotation[i] = getRotation(data.row(i), pointNum);
    }
}
```

5.10.1.2 void computePairWise (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, std::vector< std::vector< float > > & *pairwise*, std::vector< std::vector< float > > & *pairwiseNorm*)

Definition at line 164 of file Metric.cpp.

```
{
    const int& pointNum = Column/3; // how many line segments
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getPairWise_byEach(data.row(i), pointNum, pairwise[i], pairwiseNorm[i]);
    }
}
```

5.10.1.3 const float get_calculus (const float & *a*, const float & *b*, const float & *c*)

Definition at line 230 of file Metric.cpp.

```
{
    typedef boost::multiprecision::cpp_dec_float_50 mp_type;
    float result = integral(0.0F, 1.0F, 0.00001F,
  cyl_bessel_j_integral_rep<float>(a,b,c));
    return result;
}
```

5.10.1.4 void getBundleEntropy (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, std::vector< std::vector< float > > & *pairwise*)

Definition at line 209 of file Metric.cpp.

```
{
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0;i<Row;++i)
    {
        getLinearAngularEntropy(data.row(i),BUNDLE_SIZE,pairwise[i]);
    }
}
```

5.10.1.5 void getFixedSequence (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 97 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-1;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getEachFixedSequence(data.row(i), pointNum, rotationSequence[i]);
    }
}
```

5.10.1.6 void getNormalSequence (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< MultiVariate > & *normalMultivariate*)

Definition at line 75 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-1;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getNormalMultivariate(data.row(i), pointNum, normalMultivariate[i]);
    }
}
```

5.10.1.7 void getRotationSequence (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 53 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-2;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getSequence(data.row(i), pointNum, rotationSequence[i]);
    }
}
```

5.10.1.8 void getSignatureBin (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< std::vector< float > > & *pairwise*)

Definition at line 187 of file Metric.cpp.

```
{
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0;i<Row;++i)
    {
        //getSignatureHist(data.row(i),BIN_SIZE,pairwise[i]);
        getSignatureHistSampled(data.row(i),BIN_SIZE,pairwise[i]);
    }
}
```

5.10.1.9 void getUnitDirection (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< VectorXf > & *unitLength*)

Definition at line 141 of file Metric.cpp.

```
{
    const int& pointNum = Column/3;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getUnitDirection_byEach(data.row(i), pointNum, unitLength[i]);
    }
}
```

5.10.1.10 void getUnnormalizedSequence (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< MultiVariate > & *normalMultivariate*)

Definition at line 119 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-1;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getUnnormalizedMultivariate(data.row(i), pointNum, normalMultivariate[i]);
    }
}
```

5.10.2 Variable Documentation

5.10.2.1 const int& BIN_SIZE = 20

Definition at line 13 of file Metric.cpp.

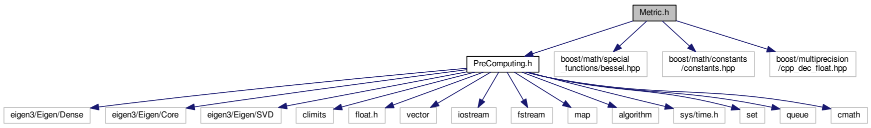
5.10.2.2 const int& BUNDLE_SIZE = 20

Definition at line 19 of file Metric.cpp.

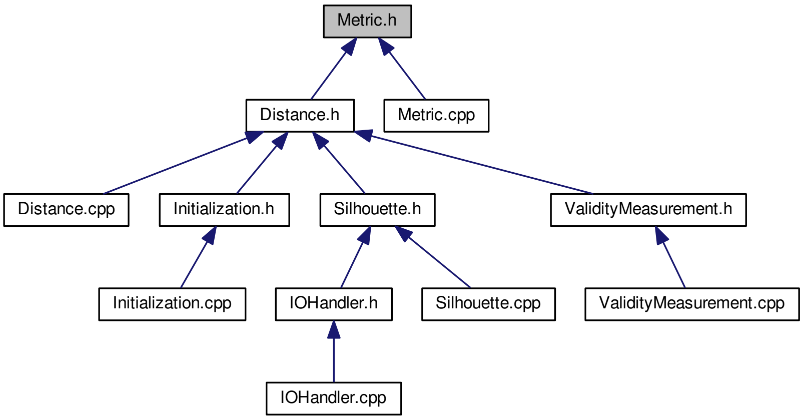
5.11 Metric.h File Reference

```
#include "PreComputing.h"
#include <boost/math/special_functions/bessel.hpp>
#include <boost/math/constants/constants.hpp>
#include <boost/multiprecision/cpp_dec_float.hpp>
```

Include dependency graph for Metric.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct MetricPreparation
- class cyl_bessel_j_integral_rep< value_type >

Functions

- void computeMeanRotation (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< float > &rotation)
- void getRotationSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std::vector< float > > &rotationSequence)
- void getNormalSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< MultiVariate > &normalMultivariate)
- void getFixedSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &rotationSequence)
- void getUnnormalizedSequence (const Eigen::MatrixXf &data, const int &Row, const int &Column, std↩ ::vector< MultiVariate > &normalMultivariate)
- void getUnitDirection (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< VectorXf > &unitLength)
- void computePairWise (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &pairwise, std::vector< std::vector< float > > &pairwiseNorm)
- void getSignatureBin (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &pairwise)
- void getBundleEntropy (const Eigen::MatrixXf &data, const int &Row, const int &Column, std::vector< std↩ ::vector< float > > &pairwise)
- template<typename value_type , typename function_type >
  value_type integral (const value_type a, const value_type b, const value_type tol, function_type func)
- const float get_calculus (const float &a, const float &b, const float &c)

Variables

- const int & BIN_SIZE
- const int & BUNDLE_SIZE

5.11.1 Function Documentation

5.11.1.1 void computeMeanRotation (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< float > & *rotation*)

Definition at line 30 of file Metric.cpp.

```
{
    rotation = std::vector<float>(Row, 0.0);
    const int& pointNum = Column/3-2;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        rotation[i] = getRotation(data.row(i), pointNum);
    }
}
```

5.11.1.2 void computePairWise (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< std::vector< float > > & *pairwise,* std::vector< std::vector< float > > & *pairwiseNorm*)

Definition at line 164 of file Metric.cpp.

```
{
    const int& pointNum = Column/3; // how many line segments
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getPairWise_byEach(data.row(i), pointNum, pairwise[i], pairwiseNorm[i]);
    }
}
```

5.11.1.3 const float get_calculus (const float & *a,* const float & *b,* const float & *c*)

Definition at line 230 of file Metric.cpp.

```
{
    typedef boost::multiprecision::cpp_dec_float_50 mp_type;
    float result = integral(0.0F, 1.0F, 0.00001F,
  cyl_bessel_j_integral_rep<float>(a,b,c));
    return result;
}
```

5.11.1.4 void getBundleEntropy (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< std::vector< float > > & *pairwise*)

Definition at line 209 of file Metric.cpp.

```
{
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0;i<Row;++i)
    {
        getLinearAngularEntropy(data.row(i),BUNDLE_SIZE,pairwise[i]);
    }
}
```

5.11.1.5 void getFixedSequence (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 97 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-1;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getEachFixedSequence(data.row(i), pointNum, rotationSequence[i]);
    }
}
```

5.11.1.6 void getNormalSequence (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, std::vector< **MultiVariate** > & *normalMultivariate*)

Definition at line 75 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-1;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getNormalMultivariate(data.row(i), pointNum, normalMultivariate[i]);
    }
}
```

5.11.1.7 void getRotationSequence (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, std::vector< std::vector< float > > & *rotationSequence*)

Definition at line 53 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-2;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getSequence(data.row(i), pointNum, rotationSequence[i]);
    }
}
```

5.11.1.8 void getSignatureBin (const Eigen::MatrixXf & *data*, const int & *Row*, const int & *Column*, std::vector< std::vector< float > > & *pairwise*)

Definition at line 187 of file Metric.cpp.

```
{
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0;i<Row;++i)
    {
        //getSignatureHist(data.row(i),BIN_SIZE,pairwise[i]);
        getSignatureHistSampled(data.row(i),BIN_SIZE,pairwise[i]);
    }
}
```

5.11.1.9 void getUnitDirection (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< VectorXf > & *unitLength*)

Definition at line 141 of file Metric.cpp.

```
{
    const int& pointNum = Column/3;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getUnitDirection_byEach(data.row(i), pointNum, unitLength[i]);
    }
}
```

5.11.1.10 void getUnnormalizedSequence (const Eigen::MatrixXf & *data,* const int & *Row,* const int & *Column,* std::vector< MultiVariate > & *normalMultivariate*)

Definition at line 119 of file Metric.cpp.

```
{
    const int& pointNum = Column/3-1;
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < Row; ++i)
    {
        getUnnormalizedMultivariate(data.row(i), pointNum, normalMultivariate[i]
  );
    }
}
```

5.11.1.11 template<typename value_type , typename function_type > value_type integral (const value_type *a,* const value_type *b,* const value_type *tol,* function_type *func*) [inline]

Definition at line 305 of file Metric.h.

```
{
    unsigned n = 1U;

    value_type h = (b - a);
    value_type I = (func(a) + func(b)) * (h / 2);

    for(unsigned k = 0U; k < 8U; k++)
    {
        h /= 2;

        value_type sum(0);
        for(unsigned j = 1U; j <= n; j++)
        {
            sum += func(a + (value_type((j * 2) - 1) * h));
        }

        const value_type I0 = I;
        I = (I / 2) + (h * sum);

        const value_type ratio     = I0 / I;
        const value_type delta     = ratio - 1;
        const value_type delta_abs = ((delta < 0) ? -delta : delta);

        if((k > 1U) && (delta_abs < tol))
        {
            break;
        }

        n *= 2U;
    }

    return I;
}
```

5.11.2 Variable Documentation

5.11.2.1 const int& BIN_SIZE

Definition at line 13 of file Metric.cpp.

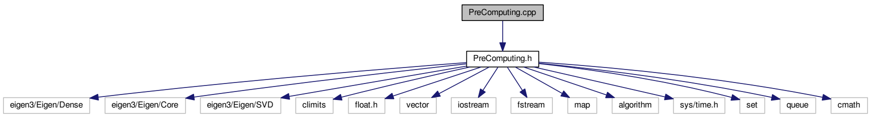
5.11.2.2 const int& BUNDLE_SIZE

Definition at line 19 of file Metric.cpp.

5.12 PreComputing.cpp File Reference

```
#include "PreComputing.h"
```

Include dependency graph for PreComputing.cpp:



Functions

- void getSequence (const VectorXf &array, const int &size, std::vector< float > &rowSequence)
- const float getRotation (const VectorXf &array, const int &size)
- void getNormalMultivariate (const VectorXf &array, const int &size, MultiVariate &rowSequence)
- void getEachFixedSequence (const VectorXf &array, const int &size, std::vector< float > &rowSequence)
- void getUnnormalizedMultivariate (const VectorXf &array, const int &size, MultiVariate &rowSequence)
- void getUnitDirection_byEach (const VectorXf &array, const int &pointNum, VectorXf &direction)
- void getPairWise_byEach (const VectorXf &data, const int &size, std::vector< float > &wiseVec, std::vector< float > &wiseNorm)
- void getSignatureHist (const Eigen::VectorXf &array, const int &binNum, std::vector< float > &histogram)
- void getSignatureHistSampled (const Eigen::VectorXf &array, const int &binNum, std::vector< float > &histogram)
- void getLinearAngularEntropy (const Eigen::VectorXf &array, const int &bundleSize, std::vector< float > &histogram)

5.12.1 Function Documentation

5.12.1.1 void getEachFixedSequence (const VectorXf & *array,* const int & *size,* std::vector< float > & *rowSequence*)

Definition at line 135 of file PreComputing.cpp.

```
{
    rowSequence = std::vector<float>(2);
    float dotValue, leftNorm, meanRotation = 0.0, deviation = 0.0, angle, result;
    Vector3f left, xRay;
    xRay << 1.0,0.0,0.0;
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        dotValue = left.dot(xRay);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            result = dotValue/leftNorm;
            result = min(1.0,(double)result);
            result = max(-1.0,(double)result);
            angle = acos(result);
            meanRotation += angle;
            deviation += angle*angle;
        }
        else
        {
            angle = M_PI;
            meanRotation += angle;
            deviation += angle*angle;
        }
    }
    meanRotation /= size;
    rowSequence[0] = meanRotation;
    int stdDevia = deviation/size-(meanRotation*meanRotation);
    if(stdDevia<0)
        stdDevia = 1.0e-8;
    rowSequence[1] = sqrt(stdDevia);
}
```

5.12.1.2 void getLinearAngularEntropy (const Eigen::VectorXf & *array,* const int & *bundleSize,* std::vector< float > & *histogram*)

Definition at line 483 of file PreComputing.cpp.

```
{
    /* if empty vector, should allocate memory ahead of time */
    if(histogram.empty())
        histogram = std::vector<float>(2);

    /* get how many vertices you'll have */
    const int& segmentNum = array.size()/3-1;

    const int& curvatureNum = segmentNum-1;

    /* should partition the whole streamlines into bunleSize segments, and compute the entropy */

    std::vector<float> segmentVec(segmentNum), curvatureVec(curvatureNum);

    Eigen::Vector3f firstSeg, secondSeg;

    /* discrete curvature */
    float curva;

    float lengthSum = 0.0, curveSum = 0.0;
    int vecIndex = 0;
    for(int i=0;i<curvatureNum;++i)
    {
        for(int j=0;j<3;++j)
        {
            firstSeg(j)=array(3*i+3+j)-array(3*i+j);
            secondSeg(j)=array(3*i+6+j)-array(3*i+3+j);
        }

        if(firstSeg.norm()<1.0e-6 || secondSeg.norm()<1.0e-6)
        {
            segmentVec[i] = 0.0;
            curvatureVec[i]= 0.0;
            continue;
        }

        float firstSegNorm = firstSeg.norm(), secondSegNorm = secondSeg.norm();
        if(firstSegNorm<1.0e-8 || secondSegNorm<1.0e-8)
            curva = 0.0;
        else
```

```
            {
                curva = firstSeg.dot(secondSeg)/firstSegNorm/secondSegNorm;

                /* clip curvature into range [-1.0, 1.0] */
                curva = std::min(float(1.0), curva);
                curva = std::max(float(-1.0), curva);

                curva = acos(curva);
            }

            /* store in the vector */
            curvatureVec[i]=curva;
            curveSum+=curva;

            /* store path */
            segmentVec[i] = firstSeg.norm();
            lengthSum+=segmentVec[i];
        }

        int i = curvatureNum;
        for(int j=0;j<3;++j)
        {
            firstSeg(j)=array(3*i+3+j)-array(3*i+j);
        }
        segmentVec[i] = firstSeg.norm();
        lengthSum+=segmentVec[i];


        /* should deal with exceptional case if lengthSum == 0 or curveSum == 0 */
        if(lengthSum<1.0e-6)
        {
            histogram[0] = 1.0;
        }
        else
        {
            /* get ratio for the vec */
            const int& segmentQuotient = segmentNum/bundleSize;
            const int& segmentResidue = segmentNum%bundleSize;

            /* get the vec for bundleSize */
            std::vector<float> lengthVec(bundleSize);

            float tempLength, linearEntropy = 0.0, prob;
            int left, right;
            for(int k = 0;k<bundleSize-1;++k)
            {
                tempLength = 0.0;
                left = k*segmentQuotient, right = (k+1)*segmentQuotient;
                for(int i = left;i<right;++i)
                    tempLength+=segmentVec[i];

                prob = tempLength/lengthSum;

                if(prob>1.0e-6)
                    linearEntropy += prob*log2f(prob);
            }

            left = (bundleSize-1)*segmentQuotient, right = segmentNum;
            tempLength = 0.0;
            for(int i=left;i<right;++i)
            {
                tempLength+=segmentVec[i];
            }
            if(prob>1.0e-6)
                prob = tempLength/lengthSum;
            linearEntropy += prob*log2f(prob);

            linearEntropy = -linearEntropy/log2f(float(bundleSize));
            histogram[0] = linearEntropy;
        }

        /* deal with curveSum == 1.0 */
        if(curveSum<1.0e-6)
        {
            histogram[1] = 1.0;
        }
        else
        {
            const int& curvatureQuotient = curvatureNum/bundleSize;
            const int& curvatureResidue = curvatureNum%bundleSize;

            /* get the vec for bundleSize */
            std::vector<float> curveVec(bundleSize);

            float tempCurve, angularEntropy = 0.0, prob;
            int left, right;
            for(int k = 0;k<bundleSize-1;++k)
            {
                tempCurve = 0.0;
                left = k*curvatureQuotient, right = (k+1)*curvatureQuotient;
                for(int i=left;i<right;++i)
                    tempCurve+=curvatureVec[i];

                prob = tempCurve/curveSum;
                if(prob>1.0e-6)
                    angularEntropy += prob*log2f(prob);
            }

            left = (bundleSize-1)*curvatureQuotient, right = curvatureNum;
            tempCurve = 0.0;
            for(int i=left;i<right;++i)
            {
                tempCurve+=curvatureVec[i];
            }
            prob = tempCurve/curveSum;
            if(prob>1.0e-6)
                angularEntropy += prob*log2f(prob);

            angularEntropy = -angularEntropy/log2f(float(bundleSize));
            histogram[1] = angularEntropy;
        }
    }
```

5.12.1.3 void getNormalMultivariate (const VectorXf & *array*, const int & *size*, MultiVariate & *rowSequence*)

Definition at line 89 of file PreComputing.cpp.

```
{
    MatrixXf normalDirection(size,3);
    float leftNorm;
    Vector3f left;
    VectorXf unitOne(size);
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            for(int k=0;k<3;k++)
                /* record each line segment normal direction */
                normalDirection(j,k) = left(k)/leftNorm;
        }
        else
        {
            for(int k=0;k<3;k++)
                /* if norm is small, mark them as zero to tell identical points */
                normalDirection(j,k) = 0.0;
        }
        unitOne(j) = 1.0;
    }

    VectorXf meanNormal(3);
    for (int i = 0; i < 3; ++i)
    {
        meanNormal(i) = normalDirection.transpose().row(i).mean();
    }

    MatrixXf tempMatrix = normalDirection-unitOne*meanNormal.transpose();
    rowSequence.covariance = tempMatrix.transpose()*tempMatrix/(size-1);
    rowSequence.meanVec = meanNormal;
}
```

5.12.1.4 void getPairWise_byEach (const VectorXf & *data*, const int & *size*, std::vector< float > & *wiseVec*, std::vector< float > & *wiseNorm*)

Definition at line 263 of file PreComputing.cpp.

```
{
    if(wiseVec.empty())
        wiseVec = std::vector<float>(3*size);

    if(wiseNorm.empty())
        wiseNorm = std::vector<float>(size);

    for (int i = 0; i < size; ++i)
    {
        float leftNorm;
        Vector3f left;
        left << data(3*i+3)-data(3*i),data(3*i+4)-data(3*i+1),data(3*i+5)-data(3*i+2);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            for (int j = 0; j < 3; ++j)
            {
                wiseVec[3*i+j] = left(j)/leftNorm;
            }
            wiseNorm[i] = leftNorm;
        }
        else
        {
            for (int j = 0; j < 3; ++j)
            {
                wiseVec[3*i+j] = 0.0;
            }
            wiseNorm[i] = 0.0;
        }
    }
}
```

5.12.1.5 const float getRotation (const VectorXf & *array*, const int & *size*)

Definition at line 57 of file PreComputing.cpp.

```
{
    float dotValue, leftNorm, rightNorm, meanRotation = 0.0, result;
    Vector3f left, right;
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        right << array(j*3+6)-array(j*3+3), array(j*3+7)-array(j*3+4), array(j*3+8)-array(j*3+5);
        dotValue = left.dot(right);
        leftNorm = left.norm();
        rightNorm = right.norm();
        if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
        {
            result = dotValue/leftNorm/rightNorm;
            result = min(1.0,(double)result);
            result = max(-1.0,(double)result);
            meanRotation += acos(result);
        }
    }
    meanRotation/=size;
    return meanRotation;
}
```

5.12.1.6 void getSequence (const VectorXf & *array*, const int & *size*, std::vector< float > & *rowSequence*)

Definition at line 17 of file PreComputing.cpp.

```
{
    rowSequence = std::vector<float>(2);
    float dotValue, leftNorm, rightNorm, meanRotation = 0.0, deviation = 0.0, angle, result;
    Vector3f left, right;
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        right << array(j*3+6)-array(j*3+3), array(j*3+7)-array(j*3+4), array(j*3+8)-array(j*3+5);
        dotValue = left.dot(right);
        leftNorm = left.norm();
        rightNorm = right.norm();
        if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
        {
            result = dotValue/leftNorm/rightNorm;
            result = min(1.0,(double)result);
            result = max(-1.0,(double)result);
            angle = acos(result);
            meanRotation += angle;
            deviation += angle*angle;
        }
    }
    meanRotation /= size;
    rowSequence[0] = meanRotation;
    int stdDevia = deviation/size-(meanRotation*meanRotation);
    if(stdDevia<0)
        stdDevia = 1.0e-8;
    rowSequence[1] = sqrt(stdDevia);
}
```

5.12.1.7 void getSignatureHist (const Eigen::VectorXf & *array*, const int & *binNum*, std::vector< float > & *histogram*)

Definition at line 307 of file PreComputing.cpp.

```
{
    /* if empty vector, should allocate memory ahead of time */
    if(histogram.empty())
        histogram = std::vector<float>(binNum);

    /* get how many vertices you’ll have */
    const int& segmentNum = array.size()/3-1;

    /* how many vertices on each bin on average */
    const int& binSize = segmentNum/binNum;

    /* first several has binSize+1 vertices, while the rest have binSize vertices */
    const int& residueNum = segmentNum%binNum;

    if(binSize<1)
    {
        std::cout << "Error for bin size calculation!" << std::endl;
        exit(1);
    }

    int totalVertexOnBin = binSize+1, index = 0;
    float dotValue, leftNorm, rightNorm, meanRotation = 0.0, result, rotationSum;
    Vector3f left, right;
    for (int i = 0; i < binNum; ++i)
    {
        /* would reduce that to binSize if i>=residueNum */
        if(i==residueNum)
            totalVertexOnBin = binSize;

        /* reset the rotationSum */
        rotationSum = 0.0;
        for(int j=0;j<totalVertexOnBin;++j)
        {
            left << array(index*3+3)-array(index*3),
                    array(index*3+4)-array(index*3+1),
                    array(index*3+5)-array(index*3+2);
            right << array(index*3+6)-array(index*3+3),
                     array(index*3+7)-array(index*3+4),
                     array(index*3+8)-array(index*3+5);
            dotValue = left.dot(right);
            leftNorm = left.norm();
            rightNorm = right.norm();
            if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
            {
                result = dotValue/leftNorm/rightNorm;
                result = min(1.0,(double)result);
                result = max(-1.0,(double)result);
                rotationSum += acos(result);
            }
            ++index;
        }

        histogram[i] = rotationSum;
    }
    assert(index==segmentNum);
}
```

5.12.1.8 void getSignatureHistSampled (const Eigen::VectorXf & *array,* const int & *binNum,* std::vector< float > & *histogram*)

Definition at line 375 of file PreComputing.cpp.

```
{
    /* if empty vector, should allocate memory ahead of time */
    if(histogram.empty())
        histogram = std::vector<float>(binNum);

    /* get how many vertices you'll have */
    const int& segmentNum = array.size()/3-2;

    /* preset a priority_queue to get the sampled points in maximal curvatures */
    priority_queue<CurvatureObject, std::vector<CurvatureObject>, CompareFunc> pQueue;

    std::vector<float> curvatureVec(segmentNum);

    Eigen::Vector3f firstSeg, secondSeg;

    /* discrete curvature */
    float curva;

    int vecIndex = 0;
    for(int i=0;i<segmentNum;++i)
    {
        for(int j=0;j<3;++j)
        {
            firstSeg(j)=array(3*i+3+j)-array(3*i+j);
            secondSeg(j)=array(3*i+6+j)-array(3*i+3+j);
        }

        float firstSegNorm = firstSeg.norm(), secondSegNorm = secondSeg.norm();
        if(firstSegNorm<1.0e-8 || secondSegNorm<1.0e-8)
            curva = 0.0;
        else
        {
            curva = firstSeg.dot(secondSeg)/firstSegNorm/secondSegNorm;

            /* clip curvature into range [-1.0, 1.0] */
            curva = std::min(float(1.0), curva);
            curva = std::max(float(-1.0), curva);

            curva = acos(curva);
        }
        /* store in the vector */
        curvatureVec[vecIndex++]=curva;

        /* push it into the priority queue */
        pQueue.push(CurvatureObject(curva, i));

    }

    /* get the first binNum-1 object */
    CurvatureObject top;

    /* use ordered_set to sort the index */
    std::vector<int> indexVec;

    int indexNum = 0;
    const int& requiredNum = binNum-1;
    while(indexNum<requiredNum && !pQueue.empty())
    {
        top = pQueue.top();
        indexVec.push_back(top.index);
        pQueue.pop();
        ++indexNum;
    }

    assert(indexVec.size()==requiredNum);

    /* sort the vec */
    std::sort(indexVec.begin(), indexVec.end());

    /* start sampling to make a curvature histogram */
    float curvatureSum = 0.0;

    /* get accumulative curvature */
    int left = 0, right;
    for(int i=0;i<requiredNum;++i)
    {
        right = indexVec[i];

        /* sum up the curvature of left and right */
        curvatureSum = 0.0;
        for(int j=left;j<=right;++j)
        {
            curvatureSum+=curvatureVec[j];
        }

        histogram[i] = curvatureSum;

        left = right+1;
    }

    /* add last element which is from left to last vertex */
    curvatureSum = 0.0;
    for(int i=left;i<segmentNum;++i)
        curvatureSum+=curvatureVec[i];
    histogram[requiredNum] = curvatureSum;
}
```

5.12.1.9 void getUnitDirection_byEach (const VectorXf & *array*, const int & *pointNum*, VectorXf & *direction*)

Definition at line 226 of file PreComputing.cpp.

```
{
    Vector3f left;
    float leftNorm;
    for (int i = 0; i < pointNum; ++i)
    {
        left << array(3*i), array(3*i+1), array(3*i+2);
        leftNorm = left.norm();
        // I Know it's hardly possible to have smaller norm, but just in case
        if(leftNorm>=1.0e-8)
        {
            for (int j = 0; j < 3; ++j)
            {
                direction(3*i+j) = left(j)/leftNorm;
            }
        }
        else
        {
            for (int j = 0; j < 3; ++j)
            {
                direction(3*i+j) = 0;
            }
        }
    }
}
```

5.12.1.10 void getUnnormalizedMultivariate (const VectorXf & *array*, const int & *size*, MultiVariate & *rowSequence*)

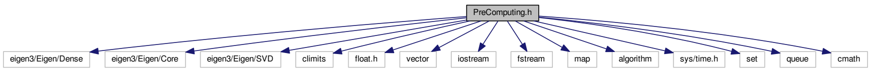
Definition at line 180 of file PreComputing.cpp.

```
{
    MatrixXf normalDirection(size,3);
    float leftNorm;
    Vector3f left;
    VectorXf unitOne(size);
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            for(int k=0;k<3;k++)
                /* record each line segment normal direction */
                normalDirection(j,k) = left(k);
        }
        else
        {
            for(int k=0;k<3;k++)
                /* if norm is small, mark them as zero to tell identical points */
                normalDirection(j,k) = 0.0;
        }
        unitOne(j) = 1.0;
    }

    VectorXf meanNormal(3);
    for (int i = 0; i < 3; ++i)
    {
        meanNormal(i) = normalDirection.transpose().row(i).mean();
    }

    MatrixXf tempMatrix = normalDirection-unitOne*meanNormal.transpose();
    rowSequence.covariance = tempMatrix.transpose()*tempMatrix/(size-1);
    rowSequence.meanVec = meanNormal;
}
```

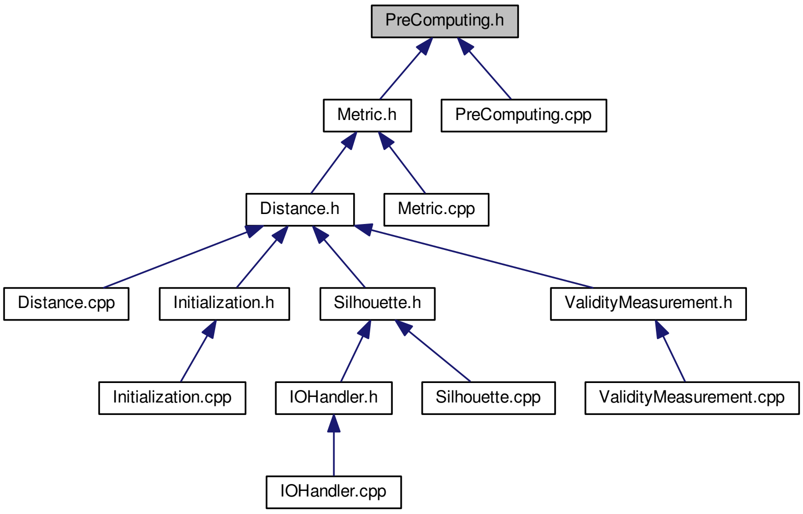
5.13 PreComputing.h File Reference

```
#include <eigen3/Eigen/Dense>
#include <eigen3/Eigen/Core>
#include <eigen3/Eigen/SVD>
#include <climits>
#include <float.h>
#include <vector>
#include <iostream>
#include <fstream>
#include <map>
#include <algorithm>
#include <sys/time.h>
#include <set>
#include <queue>
#include <cmath>
```

Include dependency graph for PreComputing.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct MultiVariate
- struct CurvatureObject
- class CompareFunc

Functions

- void getSequence (const VectorXf &array, const int &size, std::vector< float > &rowSequence)
- const float getRotation (const Eigen::VectorXf &array, const int &size)
- void getNormalMultivariate (const VectorXf &array, const int &size, MultiVariate &rowSequence)
- void getEachFixedSequence (const VectorXf &array, const int &size, std::vector< float > &rowSequence)
- void getUnnormalizedMultivariate (const VectorXf &array, const int &size, MultiVariate &rowSequence)
- void getUnitDirection_byEach (const VectorXf &array, const int &pointNum, VectorXf &direction)
- void getSignatureHist (const Eigen::VectorXf &array, const int &binSize, std::vector< float > &histogram)
- void getSignatureHistSampled (const Eigen::VectorXf &array, const int &binSize, std::vector< float > &histogram)
- void getLinearAngularEntropy (const Eigen::VectorXf &array, const int &bundleSize, std::vector< float > &histogram)
- void getPairWise_byEach (const VectorXf &data, const int &size, std::vector< float > &wiseVec, std::vector< float > &wiseNorm)
- template< typename _Matrix_Type_ >
  _Matrix_Type_ pseudoInverse (const _Matrix_Type_ &a, double epsilon=std::numeric_limits< double >↩
  ::epsilon())

5.13.1 Function Documentation

5.13.1.1 void getEachFixedSequence (const VectorXf & *array*, const int & *size*, std::vector< float > & *rowSequence*)

Definition at line 135 of file PreComputing.cpp.

```
{
    rowSequence = std::vector<float>(2);
    float dotValue, leftNorm, meanRotation = 0.0, deviation = 0.0, angle, result;
    Vector3f left, xRay;
    xRay << 1.0,0.0,0.0;
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        dotValue = left.dot(xRay);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            result = dotValue/leftNorm;
            result = min(1.0,(double)result);
            result = max(-1.0,(double)result);
            angle = acos(result);
            meanRotation += angle;
            deviation += angle*angle;
        }
        else
        {
            angle = M_PI;
            meanRotation += angle;
            deviation += angle*angle;
        }
    }
    meanRotation /= size;
    rowSequence[0] = meanRotation;
    int stdDevia = deviation/size-(meanRotation*meanRotation);
    if(stdDevia<0)
        stdDevia = 1.0e-8;
    rowSequence[1] = sqrt(stdDevia);
}
```

5.13.1.2 void getLinearAngularEntropy (const Eigen::VectorXf & *array*, const int & *bundleSize*, std::vector< float > & *histogram*)

Definition at line 483 of file PreComputing.cpp.

```
{
    /* if empty vector, should allocate memory ahead of time */
    if(histogram.empty())
        histogram = std::vector<float>(2);

    /* get how many vertices you'll have */
    const int& segmentNum = array.size()/3-1;

    const int& curvatureNum = segmentNum-1;

    /* should partition the whole streamlines into bunleSize segments, and compute the entropy */

    std::vector<float> segmentVec(segmentNum), curvatureVec(curvatureNum);

    Eigen::Vector3f firstSeg, secondSeg;

    /* discrete curvature */
    float curva;

    float lengthSum = 0.0, curveSum = 0.0;
    int vecIndex = 0;
    for(int i=0;i<curvatureNum;++i)
    {
        for(int j=0;j<3;++j)
        {
            firstSeg(j)=array(3*i+3+j)-array(3*i+j);
            secondSeg(j)=array(3*i+6+j)-array(3*i+3+j);
        }

        if(firstSeg.norm()<1.0e-6 || secondSeg.norm()<1.0e-6)
        {
            segmentVec[i] = 0.0;
            curvatureVec[i]= 0.0;
            continue;
        }

        float firstSegNorm = firstSeg.norm(), secondSegNorm = secondSeg.norm();
        if(firstSegNorm<1.0e-8 || secondSegNorm<1.0e-8)
            curva = 0.0;
        else
        {
            curva = firstSeg.dot(secondSeg)/firstSegNorm/secondSegNorm;

            /* clip curvature into range [-1.0, 1.0] */
            curva = std::min(float(1.0), curva);
            curva = std::max(float(-1.0), curva);

            curva = acos(curva);
        }

        /* store in the vector */
        curvatureVec[i]=curva;
        curveSum+=curva;

        /* store path */
        segmentVec[i] = firstSeg.norm();
        lengthSum+=segmentVec[i];
    }

    int i = curvatureNum;
    for(int j=0;j<3;++j)
    {
        firstSeg(j)=array(3*i+3+j)-array(3*i+j);
    }
    segmentVec[i] = firstSeg.norm();
    lengthSum+=segmentVec[i];


    /* should deal with exceptional case if lengthSum == 0 or curveSum == 0 */
    if(lengthSum<1.0e-6)
    {
        histogram[0] = 1.0;
    }
    else
    {
        /* get ratio for the vec */
        const int& segmentQuotient = segmentNum/bundleSize;
        const int& segmentResidue = segmentNum%bundleSize;

        /* get the vec for bundleSize */
        std::vector<float> lengthVec(bundleSize);

        float tempLength, linearEntropy = 0.0, prob;
        int left, right;
        for(int k = 0;k<bundleSize-1;++k)
        {
            tempLength = 0.0;
            left = k*segmentQuotient, right = (k+1)*segmentQuotient;
            for(int i = left;i<right;++i)
                tempLength+=segmentVec[i];

            prob = tempLength/lengthSum;

            if(prob>1.0e-6)
                linearEntropy += prob*log2f(prob);
        }

        left = (bundleSize-1)*segmentQuotient, right = segmentNum;
        tempLength = 0.0;
        for(int i=left;i<right;++i)
        {
            tempLength+=segmentVec[i];
        }
        if(prob>1.0e-6)
            prob = tempLength/lengthSum;
        linearEntropy += prob*log2f(prob);

        linearEntropy = -linearEntropy/log2f(float(bundleSize));
        histogram[0] = linearEntropy;
    }

    /* deal with curveSum == 1.0 */
    if(curveSum<1.0e-6)
    {
        histogram[1] = 1.0;
    }
    else
    {
        const int& curvatureQuotient = curvatureNum/bundleSize;
        const int& curvatureResidue = curvatureNum%bundleSize;

        /* get the vec for bundleSize */
        std::vector<float> curveVec(bundleSize);

        float tempCurve, angularEntropy = 0.0, prob;
        int left, right;
        for(int k = 0;k<bundleSize-1;++k)
        {
            tempCurve = 0.0;
            left = k*curvatureQuotient, right = (k+1)*curvatureQuotient;
            for(int i=left;i<right;++i)
                tempCurve+=curvatureVec[i];

            prob = tempCurve/curveSum;
            if(prob>1.0e-6)
                angularEntropy += prob*log2f(prob);
        }

        left = (bundleSize-1)*curvatureQuotient, right = curvatureNum;
        tempCurve = 0.0;
        for(int i=left;i<right;++i)
        {
            tempCurve+=curvatureVec[i];
        }
        prob = tempCurve/curveSum;
        if(prob>1.0e-6)
            angularEntropy += prob*log2f(prob);

        angularEntropy = -angularEntropy/log2f(float(bundleSize));
        histogram[1] = angularEntropy;
    }
}
```

5.13.1.3 void getNormalMultivariate (const VectorXf & *array*, const int & *size*, MultiVariate & *rowSequence*)

Definition at line 89 of file PreComputing.cpp.

```
{
    MatrixXf normalDirection(size,3);
    float leftNorm;
    Vector3f left;
    VectorXf unitOne(size);
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            for(int k=0;k<3;k++)
                /* record each line segment normal direction */
                normalDirection(j,k) = left(k)/leftNorm;
        }
        else
        {
            for(int k=0;k<3;k++)
                /* if norm is small, mark them as zero to tell identical points */
                normalDirection(j,k) = 0.0;
        }
        unitOne(j) = 1.0;
    }

    VectorXf meanNormal(3);
    for (int i = 0; i < 3; ++i)
    {
        meanNormal(i) = normalDirection.transpose().row(i).mean();
    }

    MatrixXf tempMatrix = normalDirection-unitOne*meanNormal.transpose();
    rowSequence.covariance = tempMatrix.transpose()*tempMatrix/(size-1);
    rowSequence.meanVec = meanNormal;
}
```

5.13.1.4 void getPairWise_byEach (const VectorXf & *data*, const int & *size*, std::vector< float > & *wiseVec*, std::vector< float > & *wiseNorm*)

Definition at line 263 of file PreComputing.cpp.

```
{
    if(wiseVec.empty())
        wiseVec = std::vector<float>(3*size);

    if(wiseNorm.empty())
        wiseNorm = std::vector<float>(size);

    for (int i = 0; i < size; ++i)
    {
        float leftNorm;
        Vector3f left;
        left << data(3*i+3)-data(3*i),data(3*i+4)-data(3*i+1),data(3*i+5)-data(3*i+2);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            for (int j = 0; j < 3; ++j)
            {
                wiseVec[3*i+j] = left(j)/leftNorm;
            }
            wiseNorm[i] = leftNorm;
        }
        else
        {
            for (int j = 0; j < 3; ++j)
            {
                wiseVec[3*i+j] = 0.0;
            }
            wiseNorm[i] = 0.0;
        }
    }
}
```

5.13.1.5 const float getRotation (const Eigen::VectorXf & *array*, const int & *size*)

5.13.1.6 void getSequence (const VectorXf & *array*, const int & *size*, std::vector< float > & *rowSequence*)

Definition at line 17 of file PreComputing.cpp.

```
{
    rowSequence = std::vector<float>(2);
    float dotValue, leftNorm, rightNorm, meanRotation = 0.0, deviation = 0.0, angle, result;
    Vector3f left, right;
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        right << array(j*3+6)-array(j*3+3), array(j*3+7)-array(j*3+4), array(j*3+8)-array(j*3+5);
        dotValue = left.dot(right);
        leftNorm = left.norm();
        rightNorm = right.norm();
        if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
        {
            result = dotValue/leftNorm/rightNorm;
            result = min(1.0,(double)result);
            result = max(-1.0,(double)result);
            angle = acos(result);
            meanRotation += angle;
            deviation += angle*angle;
        }
    }
    meanRotation /= size;
    rowSequence[0] = meanRotation;
    int stdDevia = deviation/size-(meanRotation*meanRotation);
    if(stdDevia<0)
        stdDevia = 1.0e-8;
    rowSequence[1] = sqrt(stdDevia);
}
```

5.13.1.7 void getSignatureHist (const Eigen::VectorXf & *array,* const int & *binSize,* std::vector< float > & *histogram*)

Definition at line 307 of file PreComputing.cpp.

```
{
    /* if empty vector, should allocate memory ahead of time */
    if(histogram.empty())
        histogram = std::vector<float>(binNum);

    /* get how many vertices you'll have */
    const int& segmentNum = array.size()/3-1;

    /* how many vertices on each bin on average */
    const int& binSize = segmentNum/binNum;

    /* first several has binSize+1 vertices, while the rest have binSize vertices */
    const int& residueNum = segmentNum%binNum;

    if(binSize<1)
    {
        std::cout << "Error for bin size calculation!" << std::endl;
        exit(1);
    }

    int totalVertexOnBin = binSize+1, index = 0;
    float dotValue, leftNorm, rightNorm, meanRotation = 0.0, result, rotationSum;
    Vector3f left, right;
    for (int i = 0; i < binNum; ++i)
    {
        /* would reduce that to binSize if i>=residueNum */
        if(i==residueNum)
            totalVertexOnBin = binSize;

        /* reset the rotationSum */
        rotationSum = 0.0;
        for(int j=0;j<totalVertexOnBin;++j)
        {
            left << array(index*3+3)-array(index*3),
                    array(index*3+4)-array(index*3+1),
                    array(index*3+5)-array(index*3+2);
            right << array(index*3+6)-array(index*3+3),
                     array(index*3+7)-array(index*3+4),
                     array(index*3+8)-array(index*3+5);
            dotValue = left.dot(right);
            leftNorm = left.norm();
            rightNorm = right.norm();
            if(leftNorm >= 1.0e-8 && rightNorm >=1.0e-8)
            {
                result = dotValue/leftNorm/rightNorm;
                result = min(1.0,(double)result);
                result = max(-1.0,(double)result);
                rotationSum += acos(result);
            }
            ++index;
        }

        histogram[i] = rotationSum;
    }
    assert(index==segmentNum);
}
```

5.13.1.8 void getSignatureHistSampled (const Eigen::VectorXf & *array,* const int & *binSize,* std::vector< float > & *histogram*)

Definition at line 375 of file PreComputing.cpp.

```
{
    /* if empty vector, should allocate memory ahead of time */
    if(histogram.empty())
        histogram = std::vector<float>(binNum);

    /* get how many vertices you'll have */
    const int& segmentNum = array.size()/3-2;

    /* preset a priority_queue to get the sampled points in maximal curvatures */
    priority_queue<CurvatureObject, std::vector<CurvatureObject>, CompareFunc> pQueue;

    std::vector<float> curvatureVec(segmentNum);

    Eigen::Vector3f firstSeg, secondSeg;

    /* discrete curvature */
    float curva;

    int vecIndex = 0;
    for(int i=0;i<segmentNum;++i)
    {
        for(int j=0;j<3;++j)
        {
            firstSeg(j)=array(3*i+3+j)-array(3*i+j);
            secondSeg(j)=array(3*i+6+j)-array(3*i+3+j);
        }

        float firstSegNorm = firstSeg.norm(), secondSegNorm = secondSeg.norm();
        if(firstSegNorm<1.0e-8 || secondSegNorm<1.0e-8)
            curva = 0.0;
        else
        {
            curva = firstSeg.dot(secondSeg)/firstSegNorm/secondSegNorm;

            /* clip curvature into range [-1.0, 1.0] */
            curva = std::min(float(1.0), curva);
            curva = std::max(float(-1.0), curva);

            curva = acos(curva);
        }
        /* store in the vector */
        curvatureVec[vecIndex++]=curva;

        /* push it into the priority queue */
        pQueue.push(CurvatureObject(curva, i));

    }

    /* get the first binNum-1 object */
    CurvatureObject top;

    /* use ordered_set to sort the index */
    std::vector<int> indexVec;

    int indexNum = 0;
    const int& requiredNum = binNum-1;
    while(indexNum<requiredNum && !pQueue.empty())
    {
        top = pQueue.top();
        indexVec.push_back(top.index);
        pQueue.pop();
        ++indexNum;
    }

    assert(indexVec.size()==requiredNum);

    /* sort the vec */
    std::sort(indexVec.begin(), indexVec.end());

    /* start sampling to make a curvature histogram */
    float curvatureSum = 0.0;

    /* get accumulative curvature */
    int left = 0, right;
    for(int i=0;i<requiredNum;++i)
    {
        right = indexVec[i];

        /* sum up the curvature of left and right */
        curvatureSum = 0.0;
        for(int j=left;j<=right;++j)
        {
            curvatureSum+=curvatureVec[j];
        }

        histogram[i] = curvatureSum;

        left = right+1;
    }

    /* add last element which is from left to last vertex */
    curvatureSum = 0.0;
    for(int i=left;i<segmentNum;++i)
        curvatureSum+=curvatureVec[i];
    histogram[requiredNum] = curvatureSum;
}
```

5.13.1.9 void getUnitDirection_byEach (const VectorXf & *array,* const int & *pointNum,* VectorXf & *direction*)

Definition at line 226 of file PreComputing.cpp.

```
{
    Vector3f left;
    float leftNorm;
    for (int i = 0; i < pointNum; ++i)
    {
        left << array(3*i), array(3*i+1), array(3*i+2);
        leftNorm = left.norm();
        // I Know it's hardly possible to have smaller norm, but just in case
        if(leftNorm>=1.0e-8)
        {
            for (int j = 0; j < 3; ++j)
            {
                direction(3*i+j) = left(j)/leftNorm;
            }
        }
        else
        {
            for (int j = 0; j < 3; ++j)
            {
                direction(3*i+j) = 0;
            }
        }
    }
}
```

5.13.1.10 void getUnnormalizedMultivariate (const VectorXf & *array,* const int & *size,* MultiVariate & *rowSequence*)

Definition at line 180 of file PreComputing.cpp.

```
{
    MatrixXf normalDirection(size,3);
    float leftNorm;
    Vector3f left;
    VectorXf unitOne(size);
    for (int j = 0; j < size; ++j)
    {
        left << array(j*3+3)-array(j*3), array(j*3+4)-array(j*3+1), array(j*3+5)-array(j*3+2);
        leftNorm = left.norm();
        if(leftNorm >= 1.0e-8)
        {
            for(int k=0;k<3;k++)
                /* record each line segment normal direction */
                normalDirection(j,k) = left(k);
        }
        else
        {
            for(int k=0;k<3;k++)
                /* if norm is small, mark them as zero to tell identical points */
                normalDirection(j,k) = 0.0;
        }
        unitOne(j) = 1.0;
    }

    VectorXf meanNormal(3);
    for (int i = 0; i < 3; ++i)
    {
        meanNormal(i) = normalDirection.transpose().row(i).mean();
    }

    MatrixXf tempMatrix = normalDirection-unitOne*meanNormal.transpose();
    rowSequence.covariance = tempMatrix.transpose()*tempMatrix/(size-1);
    rowSequence.meanVec = meanNormal;
}
```

5.13.1.11 template<typename _Matrix_Type_ > _Matrix_Type_ pseudoInverse (const _Matrix_Type_ & *a*, double *epsilon* = std::numeric_limits<double>::epsilon())

Definition at line 200 of file PreComputing.h.

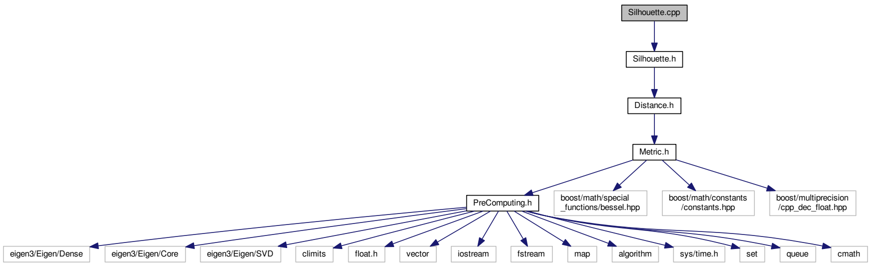
```
{
     Eigen::JacobiSVD< _Matrix_Type_ > svd(a ,Eigen::ComputeThinU | Eigen::ComputeThinV);
     double tolerance = epsilon * std::max(a.cols(), a.rows())
                           *svd.singularValues().array().abs()(0);
     return svd.matrixV() *  (svd.singularValues().array().abs() > tolerance).select
              (svd.singularValues().array().inverse(), 0).matrix().asDiagonal()
              * svd.matrixU().adjoint();
}
```

5.14 README.md File Reference

5.15 Silhouette.cpp File Reference

`#include "Silhouette.h"`

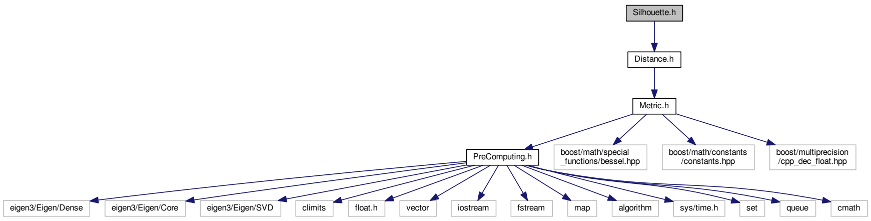
Include dependency graph for Silhouette.cpp:



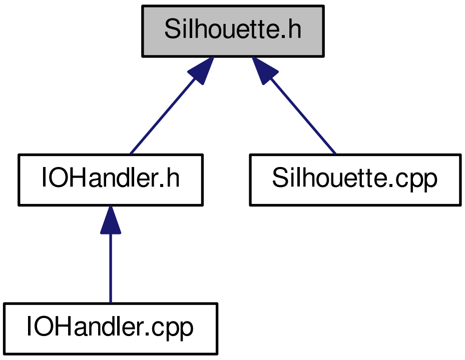
5.16 Silhouette.h File Reference

`#include "Distance.h"`

Include dependency graph for Silhouette.h:



This graph shows which files directly or indirectly include this file:



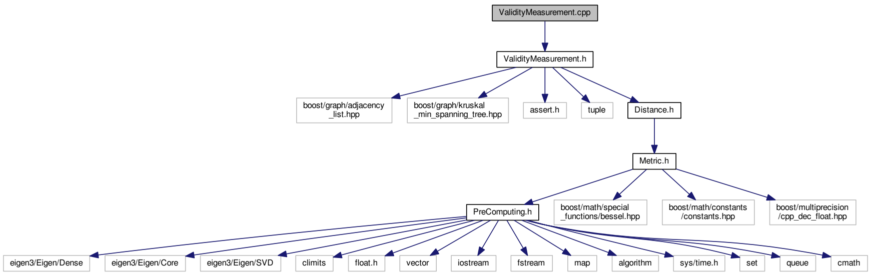
Classes

- class Silhouette

5.17 ValidityMeasurement.cpp File Reference

```
#include "ValidityMeasurement.h"
```

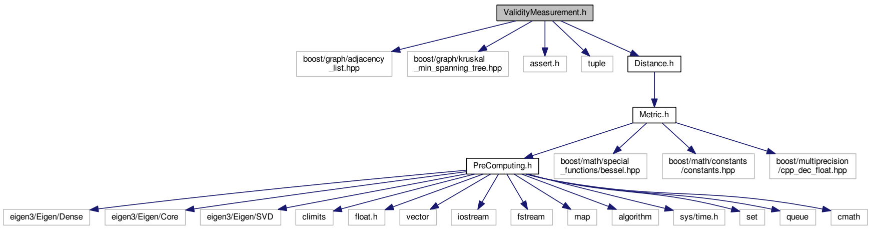
Include dependency graph for ValidityMeasurement.cpp:



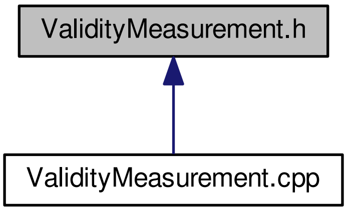
5.18 ValidityMeasurement.h File Reference

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>
#include <assert.h>
#include <tuple>
#include "Distance.h"
```

Include dependency graph for ValidityMeasurement.h:


ValidityMeasurement.h
boost/graph/adjacency _list.hpp
boost/graph/kruskal _min_spanning_tree.hpp
assert.h
tuple
Distance.h
Metric.h
PreComputing.h
boost/math/special _functions/bessel.hpp
boost/math/constants /constants.hpp
boost/multiprecision /cpp_dec_float.hpp
eigen3/Eigen/Dense
eigen3/Eigen/Core
eigen3/Eigen/SVD
climits
float.h
vector
iostream
fstream
map
algorithm
sys/time.h
set
queue
cmath

This graph shows which files directly or indirectly include this file:



Classes

- class ValidityMeasurement

Index