Spectral Clustering

The C++ implmentation for spectral clustering

Generated by Doxygen 1.8.11

Contents

Chapter 1

Spectral Clustering

Spectral clustering is a popular clustering technique that builds the normalized cut minimization for the input distance matrix. It includes two popular versions in flow visualization

- Spectral clustering (SC) with eigenrotation minimization (SC-eigen)
  - It can find the optimal number of clusters given the distance matrix and a preset bound k
  - It is very time consuming with complicated eigenrotation minimization inside the range
- k-means (SC k-means)
  - It finds the natural clusters with user input parameters after the generation of embedding space

Possible in the future we will implement a third version of spectral clustering, k-way normalized cut which has been found in flow visualization literature.

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 DataSet Struct Reference

```
#include <Predefined.h>
```

Public Attributes

- vector< vector< float > > dataVec
- Eigen::MatrixXf dataMatrix
- int maxElements = -1
- int vertexCount = -1
- int dimension = -1
- string strName
- string fullName
- string dataName

4.1.1 Detailed Description

Definition at line 20 of file Predefined.h.

4.1.2 Member Data Documentation

4.1.2.1 Eigen::MatrixXf DataSet::dataMatrix

Definition at line 23 of file Predefined.h.

4.1.2.2 string DataSet::dataName

Definition at line 30 of file Predefined.h.

4.1.2.3 vector< vector< float > > DataSet::dataVec

Definition at line 22 of file Predefined.h.

4.1.2.4 int DataSet::dimension = -1

Definition at line 26 of file Predefined.h.

4.1.2.5 string DataSet::fullName

Definition at line 29 of file Predefined.h.

4.1.2.6 int DataSet::maxElements = -1

Definition at line 24 of file Predefined.h.

4.1.2.7 string DataSet::strName

Definition at line 28 of file Predefined.h.

4.1.2.8 int DataSet::vertexCount = -1

Definition at line 25 of file Predefined.h.

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

- Predefined.h

4.2 Ensemble Struct Reference

```
#include <Predefined.h>
```

Public Attributes

- int size
- std::vector< int > element

4.2.1 Detailed Description

Definition at line 38 of file Predefined.h.

4.2.2 Member Data Documentation

4.2.2.1 std::vector<int> Ensemble::element

Definition at line 41 of file Predefined.h.

4.2.2.2 int Ensemble::size

Definition at line 40 of file Predefined.h.

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

- Predefined.h

4.3 Evrot Class Reference

```
#include <Evrot.h>
```

Public Member Functions

- Evrot (const Eigen::MatrixXf &X, int method)
- virtual ∼Evrot ()
- float getQuality ()
- std::vector< std::vector< int > > getClusters ()
- Eigen::MatrixXf & getRotatedEigenVectors ()

Protected Member Functions

- void evrot ()
- void cluster_assign ()
- float evqual (const Eigen::MatrixXf &X)
- float evqualitygrad (const Eigen::VectorXf &theta, const int &angle_index)
- Eigen::MatrixXf rotate_givens (const Eigen::VectorXf &theta)
- Eigen::MatrixXf build_Uab (const Eigen::VectorXf &theta, const int &a, const int &b)
- Eigen::MatrixXf gradU (const Eigen::VectorXf &theta, const int &k)

Protected Attributes

- int mMethod
- const int mNumDims
- const int mNumData
- int mNumAngles
- Eigen::VectorXi ik
- Eigen::VectorXi jk
- Eigen::MatrixXf mX
- Eigen::MatrixXf mXrot
- float mQuality
- std::vector< std::vector< int > > mClusters

4.3.1 Detailed Description

Definition at line 30 of file Evrot.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 Evrot::Evrot (const Eigen::MatrixXf & *X*, int *method*)

Definition at line 12 of file Evrot.cpp.

```
12                                              :
13     mMethod(method),
14     mNumDims(X.cols()),
15     mNumData(X.rows()),
16     mNumAngles((int)(mNumDims*(mNumDims-1)/2)), // get the number of angles
17     ik(Eigen::VectorXi(mNumAngles)),
18     jk(Eigen::VectorXi(mNumAngles)),
19     mX(X),
20     mClusters(std::vector<std::vector<int> >(mNumDims)) //allocate clusters vector
21 {
22     // build index mapping (to index upper triangle)
23     int k = 0;
24     for( int i=0; i<mNumDims-1; i++ ){
25         for( int j=i+1; j<=mNumDims-1; j++ ){
26             ik[k] = i;
27             jk[k] = j;
28             k++;
29         }
30     }
31
32     evrot();
33 }
```

4.3.2.2 Evrot::∼Evrot ( ) [virtual]

Definition at line 37 of file Evrot.cpp.

```
38 {
39
40 }
```

4.3.3 Member Function Documentation

4.3.3.1 Eigen::MatrixXf Evrot::build_Uab (const Eigen::VectorXf & *theta*, const int & *a*, const int & *b*) [protected]

Definition at line 243 of file Evrot.cpp.

```
244 {
245     int k,i;
246     //set Uab to be an identity matrix
247     Eigen::MatrixXf Uab(mNumDims,mNumDims);
248     Uab.setZero();
249     Uab.setIdentity();
250
251     if( b < a ) {
252         return Uab;
253     }
254
255     float tt,u_ik;
256     for( k=a; k<=b; k++ ){
257         tt = theta[k];
258     #pragma omp parallel for schedule(static) num_threads(8)
259         for( i=0; i<mNumDims; i++ ) {
260             u_ik =          Uab(i,ik[k]) * cos(tt) - Uab(i,jk[k]) * sin(tt);
261             Uab(i,jk[k]) =    Uab(i,ik[k]) * sin(tt) + Uab(i,jk[k]) * cos(tt);
262             Uab(i,ik[k]) = u_ik;
263         }
264     }
265     return Uab;
266 }
```

4.3.3.2 void Evrot::cluster_assign ( ) [protected]

Definition at line 127 of file Evrot.cpp.

```
{
    // find max of each row
    Eigen::VectorXi max_index_col(mNumData);
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0; i<mNumData; i++ )
    {
        int col=0;
        float mValue = mXrot.row(i).cwiseAbs().maxCoeff(&col);
        /*
        int row, col;
        mXrot.row(i).cwise().abs().maxCoeff(&row, &col);
        */
        max_index_col[i] = col;
    }

    // prepare cluster assignments
#pragma omp parallel for schedule(static) num_threads(8)
    for(int j=0; j<mNumDims; j++ )
    {  // loop over all columns
        for(int i=0; i<mNumData; i++ )
        { // loop over all rows
            if( max_index_col[i] == j ){
                mClusters[j].push_back(i);
            }
        }
    }

/* delete cluster that has zero elements in case that vanishing vector won't create trouble */
    std::vector<std::vector<int> > tempCluster;
    for(int i=0;i<mClusters.size();++i)
        if(!mClusters[i].empty())
            tempCluster.push_back(mClusters[i]);
    mClusters.clear();
    mClusters = tempCluster;
}
```

4.3.3.3 float Evrot::evqual (const Eigen::MatrixXf & *X*) [protected]

Definition at line 165 of file Evrot.cpp.

```
{
    // take the square of all entries and find max of each row
    Eigen::MatrixXf X2(X.rows(), X.cols());
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<X.rows();++i)
    {
        for(int j=0;j<X.cols();++j)
        {
            X2(i,j)=X(i,j)*X(i,j);
        }
    }

    Eigen::VectorXf max_values(X.rows());

#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<X.rows();++i)
        max_values(i)=X2.row(i).maxCoeff();

    // compute cost
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0; i<mNumData; i++ )
    {
        X2.row(i) = X2.row(i) / max_values[i];
    }

    float J = 1.0 - (X2.sum()/mNumData -1.0)/mNumDims;

    return J;
}
```

4.3.3.4 float Evrot::evqualitygrad (const Eigen::VectorXf & *theta*, const int & *angle_index*) [protected]

Definition at line 198 of file Evrot.cpp.

```
{
    // build V,U,A
    Eigen::MatrixXf V = gradU(theta, angle_index);

    Eigen::MatrixXf U1 = build_Uab(theta, 0,angle_index-1);
    Eigen::MatrixXf U2 = build_Uab(theta, angle_index+1,mNumAngles-1);

    Eigen::MatrixXf A = mX*U1*V*U2;

    // rotate vecs according to current angles
    Eigen::MatrixXf Y = rotate_givens(theta);

    // find max of each row
    Eigen::VectorXf max_values(mNumData);
    Eigen::VectorXi max_index_col(mNumData);
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0; i<mNumData; i++ ) {
        int row, col;
        Y.row(i).cwiseAbs().maxCoeff(&row, &col);
        max_values[i] = Y(i,col);
        max_index_col[i] = col;
    }

    // compute gradient
    float dJ=0, tmp1, tmp2;
    for( int j=0; j<mNumDims; j++ ){  // loop over all columns
        for( int i=0; i<mNumData; i++ ){ // loop over all rows
            tmp1 = A(i,j) * Y(i,j) / (max_values[i]*max_values[i]);
            tmp2 = A(i,max_index_col[i]) * (Y(i,j)*Y(i,j)) / (max_values[i]*max_values[i]*max_values[i]);
            dJ += tmp1-tmp2;
        }
    }
    dJ = 2*dJ/mNumData/mNumDims;

    return dJ;
}
```

4.3.3.5 void Evrot::evrot ( ) [protected]

Definition at line 44 of file Evrot.cpp.

```
{

    // definitions
    int max_iter = 100;
    float dQ,Q,Q_new,Q_old1,Q_old2,Q_up,Q_down;
    float alpha;
    int iter,d;

    Eigen::VectorXf theta = Eigen::VectorXf::Zero(mNumAngles);
    Eigen::VectorXf theta_new = Eigen::VectorXf::Zero(mNumAngles);

    Q = evqual(mX); // initial quality

    Q_old1 = Q;
    Q_old2 = Q;
    iter = 0;

    while( iter < max_iter ){ // iterate to refine quality
        iter++;
        for( d = 0; d < mNumAngles; d++ ){
            if( mMethod == 2 ){ // descend through numerical drivative
                alpha = 0.1;
                {
                    // move up
                    theta_new[d] = theta[d] + alpha;
                    Eigen::MatrixXf Xrot = rotate_givens(theta_new);
                    Q_up = evqual(Xrot);
                }
                {
                    // move down
                    theta_new[d] = theta[d] - alpha;
```

```
                   Eigen::MatrixXf Xrot = rotate_givens(theta_new);
                   Q_down = evqual(Xrot);
               }

               // update only if at least one of them is better
               if( Q_up > Q || Q_down > Q){
                   if( Q_up > Q_down ){
                       theta[d] = theta[d] + alpha;
                       theta_new[d] = theta[d];
                       Q = Q_up;
                   } else {
                       theta[d] = theta[d] - alpha;
                       theta_new[d] = theta[d];
                       Q = Q_down;
                   }
               }
           } else { // descend through true derivative
               alpha = 1.0;
               dQ = evqualitygrad(theta, d);
               theta_new[d] = theta[d] - alpha * dQ;
               Eigen::MatrixXf Xrot = rotate_givens(theta_new);
               Q_new = evqual(Xrot);
               if( Q_new > Q){
                   theta[d] = theta_new[d];
                    Q = Q_new;
                }
                else{
                    theta_new[d] = theta[d];
                }
            }
        }
        // stopping criteria
        if( iter > 2 ){
            if( Q - Q_old2 < 1e-3 ){
                break;
            }
        }
        Q_old2 = Q_old1;
        Q_old1 = Q;
    }


    mXrot = rotate_givens(theta_new);
    cluster_assign();

    //output
    mQuality = Q;
}
```

4.3.3.6 std::vector<std::vector<int> > Evrot::getClusters ( ) [inline]

Definition at line 36 of file Evrot.h.

```
36 { return mClusters; }
```

4.3.3.7 float Evrot::getQuality ( ) [inline]

Definition at line 35 of file Evrot.h.

```
35 { return mQuality; }
```

4.3.3.8 Eigen::MatrixXf& Evrot::getRotatedEigenVectors ( ) [inline]

Definition at line 37 of file Evrot.h.

```
37 { return mXrot; }
```

4.3.3.9 Eigen::MatrixXf Evrot::gradU (const Eigen::VectorXf & *theta*, const int & *k*) [protected]

Definition at line 268 of file Evrot.cpp.

```
{
    Eigen::MatrixXf V(mNumDims,mNumDims);
    V.setZero();

    V(ik[k],ik[k]) = -sin(theta[k]);
    V(ik[k],jk[k]) = cos(theta[k]);
    V(jk[k],ik[k]) = -cos(theta[k]);
    V(jk[k],jk[k]) = -sin(theta[k]);

    return V;
}
```

4.3.3.10 Eigen::MatrixXf Evrot::rotate_givens (const Eigen::VectorXf & *theta*) [protected]

Definition at line 236 of file Evrot.cpp.

```
{
    Eigen::MatrixXf G = build_Uab(theta, 0, mNumAngles-1);
    Eigen::MatrixXf Y = mX*G;
    return Y;
}
```

4.3.4 Member Data Documentation

4.3.4.1 Eigen::VectorXi Evrot::ik [protected]

Definition at line 54 of file Evrot.h.

4.3.4.2 Eigen::VectorXi Evrot::jk [protected]

Definition at line 55 of file Evrot.h.

4.3.4.3 std::vector<std::vector<int> > Evrot::mClusters [protected]

Definition at line 62 of file Evrot.h.

4.3.4.4 int Evrot::mMethod [protected]

Definition at line 50 of file Evrot.h.

4.3.4.5 int Evrot::mNumAngles [protected]

Definition at line 53 of file Evrot.h.

4.3.4.6 const int Evrot::mNumData `[protected]`

Definition at line 52 of file Evrot.h.

4.3.4.7 const int Evrot::mNumDims `[protected]`

Definition at line 51 of file Evrot.h.

4.3.4.8 float Evrot::mQuality `[protected]`

Definition at line 60 of file Evrot.h.

4.3.4.9 Eigen::MatrixXf Evrot::mX `[protected]`

Definition at line 58 of file Evrot.h.

4.3.4.10 Eigen::MatrixXf Evrot::mXrot `[protected]`

Definition at line 59 of file Evrot.h.

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

- Evrot.h
- Evrot.cpp

4.4 Para Struct Reference

```
#include <SpectralClustering.h>
```

Public Attributes

- int sampled
- int LaplacianOption
- bool isDistSorted
- int postProcessing
- int mMethod
- int extractOption

4.4.1 Detailed Description

Definition at line 38 of file SpectralClustering.h.

4.4.2 Member Data Documentation

4.4.2.1 int Para::extractOption

Definition at line 69 of file SpectralClustering.h.

4.4.2.2 bool Para::isDistSorted

Definition at line 54 of file SpectralClustering.h.

4.4.2.3 int Para::LaplacianOption

Definition at line 49 of file SpectralClustering.h.

4.4.2.4 int Para::mMethod

Definition at line 64 of file SpectralClustering.h.

4.4.2.5 int Para::postProcessing

Definition at line 59 of file SpectralClustering.h.

4.4.2.6 int Para::sampled

Definition at line 44 of file SpectralClustering.h.

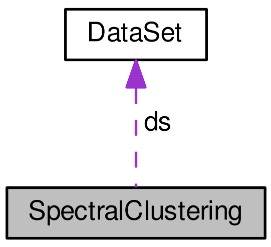
The documentation for this struct was generated from the following file:

- SpectralClustering.h

4.5 SpectralClustering Class Reference

```
#include <SpectralClustering.h>
```

Collaboration diagram for SpectralClustering:



Public Member Functions

- SpectralClustering ()
- SpectralClustering (const int &argc, char ∗∗argv, const Para &p, bool &automatic)
- ∼SpectralClustering ()
- void performClustering ()

Private Member Functions

- void extractFeatures (const std::vector< int > &storage, const std::vector< std::vector< int > > &neighbor←Vec, const Eigen::MatrixXf &centroid)
- void setDataset (const int &argc, char ∗∗argv)
- void getParameterUserInput ()
- void setParameterAutomatic (const Para &p)
- void clusterByNorm (const int &norm)
- void setLabel (vector< vector< int > > &neighborVec, vector< int > &storage, Eigen::MatrixXf &centroid)
- void getAdjacencyMatrix (Eigen::MatrixXf &adjacencyMatrix)
- void getDegreeMatrix (const Eigen::MatrixXf &adjacencyMatrix, Eigen::DiagonalMatrix< float, Dynamic > &degreeMatrix)
- void getLaplacianMatrix (const Eigen::MatrixXf &adjacencyMatrix, Eigen::DiagonalMatrix< float, Dynamic > &degreeMatrix, Eigen::MatrixXf &laplacianMatrix)
- void getEigenClustering (const Eigen::MatrixXf &laplacianMatrix, const int &norm)
- void getSigmaList ()
- void getEntropyRatio (const std::vector< int > &storage, float &EntropyRatio)
- void recordPreset (const int &number)
- void recordOptimalResult (const int &normOption, const int &clusNum)
- void normalizeEigenvec (Eigen::MatrixXf &eigenVec)
- void performKMeans (const Eigen::MatrixXf &eigenVec, std::vector< int > &storage, std::vector< std←::vector< int > > &neighborVec)
- void getEigvecRotation (std::vector< int > &storage, std::vector< std::vector< int > > &neighborVec, Eigen::MatrixXf &clusterCenter, const Eigen::MatrixXf &X)

Private Attributes

- MetricPreparation object
- int normOption = -1
- std::vector< int > group
- std::vector< string > activityList
- std::vector< string > timeList
- DataSet ds
- int numberOfClusters = -1
- int initializationOption = -1
- std::vector< float > distRange
- std::vector< float > sigmaVec
- int LaplacianOption = -1
- bool isDistSorted = -1
- int postProcessing = -1
- int extractOption = -1
- int SCALING
- bool isOptimal
- int presetNumber
- bool readCluster
- bool isPathlines
- float mMaxQuality = 0
- int mMethod = -1

4.5.1 Detailed Description

Definition at line 76 of file SpectralClustering.h.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 SpectralClustering::SpectralClustering ( )

Definition at line 21 of file SpectralClustering.cpp.

```
{

}
```

4.5.2.2 SpectralClustering::SpectralClustering (const int & *argc*, char ∗∗ *argv*, const Para & *p*, bool & *automatic*)

Definition at line 35 of file SpectralClustering.cpp.

```
{
    setDataset(argc, argv);

    if(automatic)
        setParameterAutomatic(p);
    else
        getParameterUserInput();

}
```

4.5.2.3 SpectralClustering::∼SpectralClustering ( )

Definition at line 50 of file SpectralClustering.cpp.

```
{
    deleteDistanceMatrix(ds.dataMatrix.rows());
}
```

4.5.3 Member Function Documentation

4.5.3.1 void SpectralClustering::clusterByNorm (const int & *norm*) [private]

Definition at line 173 of file SpectralClustering.cpp.

```
{
    normOption = norm;

    /* very hard to decide whether needed to perform such pre-processing */
    object = MetricPreparation(ds.dataMatrix.rows(), ds.dataMatrix.cols());
    object.preprocessing(ds.dataMatrix, ds.dataMatrix.rows(),
  ds.dataMatrix.cols(), normOption);

    /* would store distance matrix instead because it would save massive time */
    struct timeval start, end;
    double timeTemp;
    gettimeofday(&start, NULL);

    deleteDistanceMatrix(ds.dataMatrix.rows());

    std::ifstream distFile(("../dataset/"+to_string(normOption)).c_str(), ios::in);
    if(distFile.fail())
    {
        distFile.close();
        getDistanceMatrix(ds.dataMatrix, normOption, object);
        std::ofstream distFileOut(("../dataset/"+to_string(normOption)).c_str(), ios::out);
        for(int i=0;i<ds.dataMatrix.rows();++i)
        {
            for(int j=0;j<ds.dataMatrix.rows();++j)
            {
                distFileOut << distanceMatrix[i][j] << " ";
            }
            distFileOut << std::endl;
        }
        distFileOut.close();
    }
    else
    {
        std::cout << "read distance matrix..." << std::endl;

        distanceMatrix = new float*[ds.dataMatrix.rows()];
    #pragma omp parallel for schedule(static) num_threads(8)
        for (int i = 0; i < ds.dataMatrix.rows(); ++i)
        {
            distanceMatrix[i] = new float[ds.dataMatrix.rows()];
        }
        int i=0, j;
        string line;
        stringstream ss;
        while(getline(distFile, line))
        {
            j=0;
            ss.str(line);
            while(ss>>line)
            {
                if(i==j)
                    distanceMatrix[i][j]=0;
                else
                    distanceMatrix[i][j] = std::atof(line.c_str());
                ++j;
            }
            ++i;
            ss.str("");
            ss.clear();
        }
        distFile.close();
    }


    gettimeofday(&end, NULL);
    timeTemp = ((end.tv_sec  - start.tv_sec) * 1000000u
              + end.tv_usec - start.tv_usec) / 1.e6;
    activityList.push_back("Distance matrix computing for norm "+to_string(norm)+" takes: ");
    timeList.push_back(to_string(timeTemp)+" s");

    getSigmaList();


    Eigen::MatrixXf adjacencyMatrix, laplacianMatrix;
    Eigen::DiagonalMatrix<float,Dynamic> degreeMatrix;

    /* get weighted adjacency matrix by Gaussian kernel */
    getAdjacencyMatrix(adjacencyMatrix);

    /* get degree matrix */
    getDegreeMatrix(adjacencyMatrix, degreeMatrix);

    /* get Laplacian matrix */
    getLaplacianMatrix(adjacencyMatrix, degreeMatrix, laplacianMatrix);

    getEigenClustering(laplacianMatrix, norm);
}
```

4.5.3.2 void SpectralClustering::extractFeatures (const std::vector< int > & *storage,* const std::vector< std::vector< int > > & *neighborVec,* const Eigen::MatrixXf & *centroid*) [private]

Definition at line 314 of file SpectralClustering.cpp.

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

    std::cout << "Final group number information: " << std::endl;
    for (int i = 0; i < storage.size(); ++i)
    {
        std::cout << storage[i] << " ";
    }
    std::cout << std::endl;

    string pprocessing;
    switch(postProcessing)
    {
    case 1:
        pprocessing="Kmeans";
        break;

    case 2:
        pprocessing="EigenRot";
        break;
    }

    float EntropyRatio;
    getEntropyRatio(storage, EntropyRatio);

    /* record labeling information */
    // IOHandler::generateGroups(neighborVec);


    IOHandler::printClusters(ds.dataVec,group,storage,"SC_"+pprocessing+"_norm"+to_string(
  normOption),ds.fullName,ds.dimension);

    struct timeval start, end;
    double timeTemp;

    /* compute the centroid coordinates of each clustered group */

    gettimeofday(&start, NULL);

    vector<vector<float> > closest(numberOfClusters);
    vector<vector<float> > furthest(numberOfClusters);

    /* extract the closest and furthest streamlines to centroid */

#pragma omp parallel for schedule(static) num_threads(8)
    for (int i=0;i<numberOfClusters;++i)
    {
        float minDist = FLT_MAX;
        float maxDist = -10;
        int minIndex = -1, maxIndex = -1;
        const std::vector<int>& groupRow = neighborVec[i];
        const Eigen::VectorXf& eachCentroid = centroid.row(i);
        for (int j = 0; j < groupRow.size(); ++j)
        {
            float distance = getDisimilarity(eachCentroid,ds.dataMatrix,groupRow[j],
  normOption,object);
            if(minDist>distance)
            {
                minDist = distance;
                minIndex = groupRow[j];
            }
            if(maxDist<distance)
            {
                maxDist = distance;
                maxIndex = groupRow[j];
            }
        }
        closest[i] = ds.dataVec[minIndex];
        furthest[i] = ds.dataVec[maxIndex];
    }

    std::vector<std::vector<float> > center_vec(numberOfClusters, vector<float>(Column));
#pragma omp parallel for schedule(static) num_threads(8)
    for (int i = 0; i < center_vec.size(); ++i)
    {
        for (int j = 0; j < Column; ++j)
        {
            center_vec[i][j] = centroid(i,j);
        }
    }

    gettimeofday(&end, NULL);
    timeTemp = ((end.tv_sec  - start.tv_sec) * 1000000u
               + end.tv_usec - start.tv_usec) / 1.e6;
    activityList.push_back("Feature extraction takes: ");
    timeList.push_back(to_string(timeTemp)+" s");

    ValidityMeasurement vm;
    vm.computeValue(normOption, ds.dataMatrix, group, object, false);
    activityList.push_back("SC Validity measure is: ");
    stringstream fc_ss;
    fc_ss << vm.f_c;
    timeList.push_back(fc_ss.str());

    gettimeofday(&start, NULL);
    Silhouette sil;
    sil.computeValue(normOption,ds.dataMatrix,ds.dataMatrix.rows(),ds.dataMatrix.cols(),group,object,
                     numberOfClusters, false, neighborVec);
    gettimeofday(&end, NULL);
    timeTemp = ((end.tv_sec  - start.tv_sec) * 1000000u + end.tv_usec - start.tv_usec) / 1.e6;
    activityList.push_back("Silhouette calculation takes: ");
    timeList.push_back(to_string(timeTemp)+" s");

    std::cout << "Finishing extracting features!" << std::endl;

    stringstream ss;
    ss << "norm_" << normOption;

    std::vector<float> closestRotation, furthestRotation;
    const float& closestAverage = getRotation(closest, closestRotation);
    const float& furthestAverage = getRotation(furthest, furthestRotation);

    /* save closest, furthest and centroid representative streamlines */
    IOHandler::printFeature(ds.dataName+"_SC"+pprocessing+"_closest_"+ss.str()+".vtk", closest, sil.sCluster,
            closestRotation, ds.dimension);
    IOHandler::printFeature(ds.dataName+"_SC"+pprocessing+"_furthest_"+ss.str()+".vtk", furthest, sil.sCluster,
            furthestRotation, ds.dimension);
    IOHandler::printFeature(ds.dataName+"_SC"+pprocessing+"_centroid_"+ss.str()+".vtk", center_vec, sil.sCluster,ds.dimension);

    IOHandler::printToFull(ds.dataVec, sil.sData, "SC"+pprocessing+"_SValueLine_"+ss.str(), ds.fullName, ds.dimension);
    IOHandler::printToFull(ds.dataVec, group, sil.sCluster, "SC"+pprocessing+"_SValueCluster_"+ss.str(), ds.fullName, ds.dimension);

    activityList.push_back("numCluster is: ");
    timeList.push_back(to_string(numberOfClusters));

    activityList.push_back("Norm option is: ");
    timeList.push_back(to_string(normOption));

    activityList.push_back("SC post-processing is: ");
    switch(postProcessing)
    {
    case 1:
        timeList.push_back("k-means");
        break;

    case 2:
        timeList.push_back("vector rotation");
        break;
    }

    activityList.push_back("Average rotation of closest is: ");
    timeList.push_back(to_string(closestAverage));

    activityList.push_back("Average rotation of furthest is: ");
    timeList.push_back(to_string(furthestAverage));

    IOHandler::generateReadme(activityList,timeList);

    IOHandler::writeReadme(EntropyRatio, sil, "For norm "+to_string(normOption));
}
```

4.5.3.3 void SpectralClustering::getAdjacencyMatrix (Eigen::MatrixXf & *adjacencyMatrix*) [private]

Definition at line 579 of file SpectralClustering.cpp.

```
{
    //in case of diagonal matrix element is not assigned
    adjacencyMatrix = Eigen::MatrixXf::Zero(ds.dataMatrix.rows(), ds.dataMatrix.rows());
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<adjacencyMatrix.rows();++i)
    {
        for(int j=0;j<adjacencyMatrix.cols();++j)
        {
            float dist_ij;
            if(i==j)
                continue;
            else if(distanceMatrix)
            {
                dist_ij = distanceMatrix[i][j];
            }
            else
                dist_ij = getDisimilarity(ds.dataMatrix, i, j, normOption, object);
            adjacencyMatrix(i,j)=exp(-dist_ij*dist_ij/sigmaVec[i]/sigmaVec[j]);
        }
    }

    std::cout << "Finish computing adjacency matrix!" << std::endl;
}
```

4.5.3.4 void SpectralClustering::getDegreeMatrix (const Eigen::MatrixXf & *adjacencyMatrix*, Eigen::DiagonalMatrix< float, Dynamic > & *degreeMatrix*) [private]

Definition at line 611 of file SpectralClustering.cpp.

```
{
    degreeMatrix = Eigen::DiagonalMatrix<float,Dynamic>(ds.dataMatrix.rows());
    Eigen::VectorXf v = VectorXf::Zero(ds.dataMatrix.rows());
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<v.size();++i)
    {
        float summation = 0;
        for(int j=0;j<adjacencyMatrix.cols();++j)
        {
            summation+=adjacencyMatrix(i,j);
        }
        v(i) = summation;
    }

    degreeMatrix.diagonal() = v;

    std::cout << "Fnish computing degree matrix!" << std::endl;
}
```

4.5.3.5 void SpectralClustering::getEigenClustering (const Eigen::MatrixXf & *laplacianMatrix*, const int & *norm*) [private]

Definition at line 669 of file SpectralClustering.cpp.

```
{
    struct timeval start, end;
    gettimeofday(&start, NULL);

    /* eigen decomposition for Hermite matrix (real and symmetric matrix) */
    std::cout << "Eigen decomposition starts!..." << std::endl;
    SelfAdjointEigenSolver<MatrixXf> eigensolver(laplacianMatrix);
    std::cout << "Eigen decomposition ends!..." << std::endl;

    gettimeofday(&end, NULL);
    float timeTemp = ((end.tv_sec-start.tv_sec)*1000000u+end.tv_usec-start.tv_usec)/1.e6;
    activityList.push_back("Eigen decomposition takes: ");
    timeList.push_back(to_string(timeTemp)+" s");

    const int& eigenRows = presetNumber;
    std::cout << "Eigen rows are: " << eigenRows << std::endl;
    //const int& eigenRows = 5;

    Eigen::MatrixXf eigenVec(eigenRows, ds.dataMatrix.rows());

    const int& Row = laplacianMatrix.rows();

    /* from paper we know it should get largest eigenvalues, and from eigen library we know it's latter */
    for(int i=Row-1;i>Row-eigenRows-1;--i)
        eigenVec.row(Row-1-i) = eigensolver.eigenvectors().col(i).transpose();
    eigenVec.transposeInPlace();

    /* how many elements in each cluster */
    std::vector<int> storage;

    /* which elements stored in each cluster */
    std::vector<std::vector<int> > neighborVec;

    /* centroid cluster */
    Eigen::MatrixXf clusterCenter;

    /* k-means as a post-processing */
    if(postProcessing==1)
    {
        normalizeEigenvec(eigenVec);

        performKMeans(eigenVec,storage,neighborVec);

        setLabel(neighborVec, storage, clusterCenter);

        extractFeatures(storage,neighborVec,clusterCenter);
    }
    /* eigenvector rotation */
    else if(postProcessing==2)
    {
        getEigvecRotation(storage,neighborVec,clusterCenter,eigenVec);

        if(neighborVec.empty())
            return;

        setLabel(neighborVec, storage, clusterCenter);

        if(isOptimal)
            recordOptimalResult(norm, neighborVec.size());
        else
            extractFeatures(storage,neighborVec,clusterCenter);
    }
}
```

4.5.3.6 void SpectralClustering::getEigvecRotation (std::vector< int > & *storage*, std::vector< std::vector< int > > & *neighborVec*, Eigen::MatrixXf & *clusterCenter*, const Eigen::MatrixXf & *X*) [private]

Definition at line 888 of file SpectralClustering.cpp.

```
{
    mMaxQuality = 0;
    Eigen::MatrixXf vecRot;
    Eigen::MatrixXf vecIn = X.block(0,0,X.rows(),2);
    Evrot *e = NULL;

    struct timeval start, end;
    gettimeofday(&start, NULL);

    const int& xCols = X.cols();

    std::cout << "Eigenvector rotation starts within " << xCols << " columns..." << std::endl;
    for (int g=2; g <= xCols; g++)
    {
        // make it incremental (used already aligned vectors)
        std::cout << "column " << g << ":";
        if( g > 2 )
        {
            vecIn.resize(X.rows(),g);
            vecIn.block(0,0,vecIn.rows(),g-1) = e->getRotatedEigenVectors();
            vecIn.block(0,g-1,X.rows(),1) = X.block(0,g-1,X.rows(),1);
            delete e;
        }
        //perform the rotation for the current number of dimensions
        e = new Evrot(vecIn, mMethod);

        //save max quality
        if (e->getQuality() > mMaxQuality)
        {
            mMaxQuality = e->getQuality();
        }

        if(isnan(e->getQuality())||isinf(e->getQuality()))
        {
            std::cout << "Meet with nan or inf! Stop! " << std::endl;
            return;
        }

        std::cout << " max quality is " << mMaxQuality << ", Evrot has quality " << e->
    getQuality() << std::endl;
        //save cluster data for max cluster or if we're near the max cluster (so prefer more clusters)
        if ((e->getQuality() > mMaxQuality) || (mMaxQuality - e->
    getQuality() <= 0.001))
        {
            neighborVec = e->getClusters();
            vecRot = e->getRotatedEigenVectors();
        }
    }

    gettimeofday(&end, NULL);
    float timeTemp = ((end.tv_sec-start.tv_sec)*1000000u+end.tv_usec-start.tv_usec)/1.e6;
    activityList.push_back("Eigenvector rotation takes: ");
    timeList.push_back(to_string(timeTemp)+" s");

    if(neighborVec.empty())
        return;

    clusterCenter = Eigen::MatrixXf::Zero(neighborVec.size(),vecRot.cols());
    storage = std::vector<int>(neighborVec.size());

#pragma omp parallel for schedule(static) num_threads(8)
    for (unsigned int i=0; i < neighborVec.size(); i++)
    {
        storage[i] = neighborVec[i].size();
        for (unsigned int j=0; j < neighborVec[i].size(); j++)
        {
            //sum points within cluster
            clusterCenter.row(i) += vecRot.row(neighborVec[i][j]);
        }
    }

#pragma omp parallel for schedule(static) num_threads(8)
    for (unsigned int i=0; i < neighborVec.size(); i++) {
        //find average point within cluster
        clusterCenter.row(i) = clusterCenter.row(i) / neighborVec[i].size();
    }

    numberOfClusters = neighborVec.size();
}
```

4.5.3.7 void SpectralClustering::getEntropyRatio (const std::vector< int > & *storage*, float & *EntropyRatio*) [private]

Definition at line 561 of file SpectralClustering.cpp.

```
{
    EntropyRatio = 0;
    const int& Row = ds.dataMatrix.rows();
    for (int i = 0; i < storage.size(); ++i)
    {
        float ratio = float(storage[i])/float(Row);
        EntropyRatio-=ratio*log2f(ratio);
    }
    EntropyRatio/=log2f(storage.size());
}
```

4.5.3.8 void SpectralClustering::getLaplacianMatrix (const Eigen::MatrixXf & *adjacencyMatrix*, Eigen::DiagonalMatrix< float, Dynamic > & *degreeMatrix*, Eigen::MatrixXf & *laplacianMatrix*) [private]

Definition at line 639 of file SpectralClustering.cpp.

```
{
    switch(LaplacianOption)
    {
    default:
    case 1:
    /* L = D^(-1)A */
        getMatrixPow(degreeMatrix, -1.0);
        laplacianMatrix=degreeMatrix*adjacencyMatrix;
        break;

    case 2:
        Eigen::MatrixXf dMatrix = Eigen::MatrixXf(adjacencyMatrix.rows(),adjacencyMatrix.cols());
        const Eigen::VectorXf& m_v = degreeMatrix.diagonal();
        for(int i=0;i<dMatrix.rows();++i)
            dMatrix(i,i) = m_v(i);
        laplacianMatrix = dMatrix-adjacencyMatrix;
        break;
    }
}
```

4.5.3.9 void SpectralClustering::getParameterUserInput ( ) [private]

Definition at line 1015 of file SpectralClustering.cpp.

```
{
    std::cout << "It is a pathline data set? 1.Yes, 0.No." << std::endl;
    int pathlineOption;
    std::cin >> pathlineOption;
    assert(pathlineOption==1||pathlineOption==0);
    isPathlines = (pathlineOption==1);

    int sampleOption;

    if(isPathlines)
        sampleOption = 1;
    else
    {
        std::cout << "choose a sampling method for the dataset?" << std::endl
                  << "1.directly filling with last vertex; 2. uniform sampling." << std::endl;
        std::cin >> sampleOption;
    }
    assert(sampleOption==1||sampleOption==2);

    if(sampleOption==1)
        IOHandler::expandArray(ds.dataMatrix,ds.dataVec,ds.dimension,ds.maxElements);
    else if(sampleOption==2)
        IOHandler::sampleArray(ds.dataMatrix,ds.dataVec,ds.dimension,ds.maxElements);
    else if(sampleOption==3)
        IOHandler::uniformArcSampling(ds.dataMatrix,ds.dataVec,ds.dimension,ds.maxElements);

    group = std::vector<int>(ds.dataMatrix.rows());

    /* the default value for streamline clustering is 2 normalized Laplacian */
    std::cout << "---------------------------" << std::endl;
    std::cout << "Laplacian option: 1.Normalized Laplacian, 2.Unsymmetric Laplacian" << std::endl;
    std::cout << "..And in streamline clustering people tend to choose 1.Normalized Laplacian!-----------" << std::endl;
    std::cin >> LaplacianOption;
    assert(LaplacianOption==1||LaplacianOption==2);


    int sortedOption;
    std::cout << "Please choose whether local scaling by sorted distance: 1. yes, 2. no: " << std::endl;
    std::cin >> sortedOption;
    assert(sortedOption==1||sortedOption==2);
    if(sortedOption==1)
        sortedOption = true;
    else if(sortedOption==2)
        sortedOption = false;

    std::cout << "------------------------------------------------------------------------" << std::endl;
    std::cout << "Input a desired cluster number among [1, " << ds.dataMatrix.rows() << "]: ";
    std::cin >> presetNumber;
    assert(presetNumber>1 && presetNumber<ds.dataMatrix.rows()/10);

    std::cout << "------------------------------------------------------------------------" << std::endl;
    std::cout << "Input a post-processing method: 1.k-means, 2.eigenvector rotation: " << std::endl;
    std::cin >> postProcessing;
    assert(postProcessing==1||postProcessing==2);

    if(postProcessing==2)
    {
        std::cout << "------------------------------------------------" << std::endl;
        std::cout << "Please input derivative method: 1.numerical derivative, 2.true derivative." <<
      std::endl;
        std::cin >> mMethod;
        assert(mMethod==1 || mMethod==2);
    }

    std::cout << "Please choose cluster number method, 0.user input, 1.read clustering: " << std::endl;
    int clusterInput;
    std::cin >> clusterInput;
    assert(clusterInput==0 || clusterInput==1);
    readCluster = (clusterInput==1);

}
```

4.5.3.10 void SpectralClustering::getSigmaList ( ) [private]

Definition at line 498 of file SpectralClustering.cpp.

```
{
    const int& Row = ds.dataMatrix.rows();
    sigmaVec = std::vector<float>(Row);

    if(isDistSorted)
    {
        /* get SCALING-th smallest dist */
    #pragma omp parallel for schedule(static) num_threads(8)
        for(int i=0;i<Row;++i)
        {
            /* instead we implement a n*logk priority_queue method for finding k-th smallest element */
            std::priority_queue<float> limitQueue;
            float tempDist;
            for(int j=0;j<Row;++j)
            {
                if(i==j)
                    continue;
                if(distanceMatrix)
                    tempDist = distanceMatrix[i][j];
                else
                    tempDist = getDisimilarity(ds.dataMatrix, i, j,
      normOption, object);
                // element is even larger than the biggest
                limitQueue.push(tempDist);
                if(limitQueue.size()>SCALING)
                    limitQueue.pop();
            }

            sigmaVec[i] = limitQueue.top();
        }
    }
    else
    {
        /* directly by index since in both papers only mention i-th neighboring point */
    #pragma omp parallel for schedule(static) num_threads(8)
        for(int i=0;i<Row;++i)
        {
            if(i<SCALING)
            {
                if(distanceMatrix)
                    sigmaVec[i]=distanceMatrix[i][SCALING];
                else
                    sigmaVec[i]=getDisimilarity(ds.dataMatrix, i,
      SCALING, normOption, object);
            }
            else
            {
                if(distanceMatrix)
                    sigmaVec[i]=distanceMatrix[i][SCALING-1];
                else
                    sigmaVec[i]=getDisimilarity(ds.dataMatrix, i,
      SCALING-1, normOption, object);
            }
        }
    }
    std::cout << "Finish local scaling..." << std::endl;
}
```

4.5.3.11 void SpectralClustering::normalizeEigenvec (Eigen::MatrixXf & *eigenVec*) [private]

Definition at line 755 of file SpectralClustering.cpp.

```
{
    const int& rows = eigenVec.rows();
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<rows;++i)
    {
        eigenVec.row(i)/=eigenVec.row(i).norm();
    }
}
```

4.5.3.12 void SpectralClustering::performClustering ( )

Definition at line 59 of file SpectralClustering.cpp.

```
{
    //distance metric type
    /*  0: Euclidean Norm
        1: Fraction Distance Metric
        2: piece-wise angle average
        3: Bhattacharyya metric for rotation
        4: average rotation
        5: signed-angle intersection
        6: normal-direction multivariate distribution
        7: Bhattacharyya metric with angle to a fixed direction
        8: Piece-wise angle average \times standard deviation
        9: normal-direction multivariate un-normalized distribution
        10: x*y/|x||y| borrowed from machine learning
        11: cosine similarity
        12: Mean-of-closest point distance (MCP)
        13: Hausdorff distance min_max(x_i,y_i)
        14: Signature-based measure from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6231627
        15: Procrustes distance take from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6787131
        16: entropy-based distance metric taken from http://vis.cs.ucdavis.edu/papers/pg2011paper.pdf
        17: time-series MCP distance from https://www.sciencedirect.com/science/article/pii/S0097849318300128
            for pathlines only
    */
    if(postProcessing==2)
    {
        std::cout << "Find optimal activated? 0. No, 1. Yes: " << std::endl;
        int optimalOption;
        std::cin >> optimalOption;
        assert(optimalOption==0 || optimalOption==1);
        isOptimal = (optimalOption==1);

        std::cout << "Please input the preset number of clusters in [2, " << ds.dataVec.size() << "]: " << std::endl;
        std::cin >> presetNumber;
        assert(presetNumber>=2 && presetNumber<=ds.dataVec.size());

        /* record initial number of clusters of user input */
        recordPreset(presetNumber);

        readCluster = false;
    }
    else if(postProcessing==1)
    {
        std::cout << "Please choose cluster number method, 0.user input, 1.read clustering: " << std::endl;
        int clusterInput;
        std::cin >> clusterInput;
        assert(clusterInput==0 || clusterInput==1);
        readCluster = (clusterInput==1);
    }

    std::unordered_map<int,int> clusterMap;
    if(readCluster)
    {
        IOHandler::readClusteringNumber(clusterMap, "cluster_number");
    }


    for(int i=0;i<=17;++i)
    {
        if(isPathlines)
        {
            /* don't want to deal with many too naive metrics */
            if(i!=0 && i!=1 && i!=2 && i!=4 && i!=12 && i!=13 && i!=14 && i!=15 && i!=17)
                continue;
        }
        else
        {
            if(i!=0 && i!=1 && i!=2 && i!=4 && i!=12 && i!=13 && i!=14 && i!=15)
                continue;
        }

        if(postProcessing==1)
        {
            if(readCluster)
                presetNumber = clusterMap[i];
            else
            {
                std::cout << "Please input the preset number of clusters for norm " << i << " among [2, "
                        << ds.dataVec.size() << "]: " << std::endl;
                std::cin >> presetNumber;
            }
            assert(presetNumber>=2 && presetNumber<=ds.dataVec.size());
        }

        std::cout << "-----------------------------------------------------" << std::endl;
        std::cout << "Experiment on norm " << i << " starts!--------------" << std::endl;

        activityList.clear();
        timeList.clear();

        activityList.push_back("Preset numOfClusters for norm "+to_string(i) +" is: ");
        timeList.push_back(to_string(presetNumber));

        struct timeval start, end;
        double timeTemp;
        gettimeofday(&start, NULL);

        clusterByNorm(i);

        gettimeofday(&end, NULL);
        timeTemp = ((end.tv_sec  - start.tv_sec) * 1000000u
                           + end.tv_usec - start.tv_usec) / 1.e6;
        activityList.push_back("SC for "+to_string(i) +" takes: ");
        timeList.push_back(to_string(timeTemp)+"s");

        std::cout << std::endl;
    }
}
```

4.5.3.13 void SpectralClustering::performKMeans (const Eigen::MatrixXf & *eigenVec,* std::vector< int > & *storage,* std::vector< std::vector< int > > & *neighborVec*) [private]

Definition at line 773 of file SpectralClustering.cpp.

```
{

    const int& Row = eigenVec.rows();
    const int& Column = eigenVec.cols();

    float moving=1000, tempMoving, before;

    numberOfClusters = presetNumber;

    storage = std::vector<int>(numberOfClusters);

    /* centerTemp is temporary term for storing centroid position, clusterCenter is permanent */
    MatrixXf centerTemp, clusterCenter;

    /* chosen from sample for initialization of k-means */
    Initialization::generateFromSamples(clusterCenter,Column,eigenVec,
  numberOfClusters);

    int tag = 0;

    neighborVec=std::vector< std::vector<int> >(numberOfClusters);

    float PCA_KMeans_delta, KMeans_delta;

    std::cout << "...k-means started!" << std::endl;

    struct timeval start, end;
    gettimeofday(&start, NULL);

    do
    {
        before = moving;
        /* preset cluster number recorder */
        std::fill(storage.begin(), storage.end(), 0);

        centerTemp = MatrixXf::Zero(numberOfClusters, Column);

    #pragma omp parallel for schedule(static) num_threads(8)
        for (int i = 0; i < numberOfClusters; ++i)
        {
            neighborVec[i].clear();
        }

    #pragma omp parallel num_threads(8)
        {
        #pragma omp for nowait
            for (int i = 0; i < Row; ++i)
            {
                float dist = FLT_MAX;
                float temp;
                int clusTemp;
                for (int j = 0; j < numberOfClusters; ++j)
                {
                    temp = (eigenVec.row(i)-clusterCenter.row(j)).norm();
                    if(temp<dist)
                    {
                        dist = temp;
                        clusTemp = j;
                    }
                }

            #pragma omp critical
                {
                    storage[clusTemp]++;
                    neighborVec[clusTemp].push_back(i);
                    group[i] = clusTemp;
                    centerTemp.row(clusTemp)+=eigenVec.row(i);
                }
            }
        }

        moving = FLT_MIN;

    #pragma omp parallel for reduction(max:moving) num_threads(8)
        for (int i = 0; i < numberOfClusters; ++i)
        {
            if(storage[i]>0)
            {
                centerTemp.row(i)/=storage[i];
                tempMoving = (centerTemp.row(i)-clusterCenter.row(i)).norm();
                clusterCenter.row(i) = centerTemp.row(i);
                if(moving<tempMoving)
                    moving = tempMoving;
            }
        }
        std::cout << "K-means iteration " << ++tag << " completed, and moving is "
        << moving << "!" << std::endl;
    }while(abs(moving-before)/before >= 1.0e-3 && tag < 50 && moving>0.01);

    gettimeofday(&end, NULL);
    float timeTemp = ((end.tv_sec-start.tv_sec)*1000000u+end.tv_usec-start.tv_usec)/1.e6;
    activityList.push_back("K-means takes: ");
    timeList.push_back(to_string(timeTemp)+" s");

    for(auto iter=storage.begin(); iter!=storage.end();)
    {
        if(*iter==0)
            storage.erase(iter);
        else
            ++iter;
    }
    numberOfClusters = storage.size();
}
```

4.5.3.14 void SpectralClustering::recordOptimalResult (const int & *normOption,* const int & *clusNum*) [private]

Definition at line 1113 of file SpectralClustering.cpp.

```
{
    std::ofstream readme("../dataset/optimal.txt",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "Optimal number of cluster for norm " << normOption << " with sc eigen-rotation
  minimization is "
           << clusNum << std::endl;
    readme << std::endl;
    readme.close();
}
```

4.5.3.15 void SpectralClustering::recordPreset (const int & *number*) [private]

Definition at line 1093 of file SpectralClustering.cpp.

```
{
    std::ofstream readme("../dataset/optimal.txt",ios::out | ios::app);
    if(!readme)
    {
        std::cout << "Error creating readme!" << std::endl;
        exit(1);
    }
    readme << "Preset cluster number is: " << number << std::endl;
    readme << std::endl;
    readme.close();
}
```

4.5.3.16 void SpectralClustering::setDataset (const int & *argc,* char ∗∗ *argv*) [private]

Definition at line 473 of file SpectralClustering.cpp.

```
{
    if(argc!=3)
    {
        std::cout << "Input argument should have 3!" << endl
                  << "./cluster inputFile_name(in dataset folder) "
                  << "data_dimension(3)" << endl;
        exit(1);
    }
    ds.strName = string("../dataset/")+string(argv[1]);
    ds.dataName = string(argv[1]);
    ds.dimension = atoi(argv[2]);

    IOHandler::readFile(ds.strName,ds.dataVec,ds.vertexCount,
  ds.dimension,ds.maxElements);

    ds.fullName = ds.strName+"_full.vtk";
    IOHandler::printVTK(ds.fullName, ds.dataVec, ds.
  vertexCount, ds.dimension);

    SCALING = 0.05*ds.dataVec.size();
}
```

4.5.3.17 void SpectralClustering::setLabel (vector< vector< int > > & *neighborVec,* vector< int > & *storage,* Eigen::MatrixXf & *centroid*) [private]

Definition at line 269 of file SpectralClustering.cpp.

```
{
    std::vector<Ensemble> nodeVec(storage.size());
    std::cout << "Cluster label setting begins with " << numberOfClusters << " clusters..." << std::endl;
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<nodeVec.size();++i)
    {
        nodeVec[i].size = storage[i];
        nodeVec[i].element = neighborVec[i];
    }

    /* sort group index by size of elements containd inside */
    std::sort(nodeVec.begin(), nodeVec.end(), [](const Ensemble& first, const Ensemble& second)
    {return first.size<second.size|| (first.size==second.size&&first.element[0]<second.element[0]);});

    neighborVec = std::vector<std::vector<int> >(nodeVec.size());
    storage = std::vector<int>(nodeVec.size());
    centroid = Eigen::MatrixXf(nodeVec.size(), ds.dataMatrix.cols());

#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<nodeVec.size();++i)
    {
        neighborVec[i] = nodeVec[i].element;
        storage[i] = nodeVec[i].size;
        Eigen::VectorXf tempVec = Eigen::VectorXf::Zero(ds.dataMatrix.cols());
        for(int j=0;j<storage[i];++j)
        {
            tempVec+=ds.dataMatrix.row(i).transpose();
            /* don't forget to re-compute the group tag */
            group[neighborVec[i][j]]=i;
        }
        centroid.row(i) = tempVec/storage[i];
    }

    std::cout << "Cluster label setting ends..." << std::endl;
}
```

4.5.3.18 void SpectralClustering::setParameterAutomatic (const Para & *p*) [private]

Definition at line 974 of file SpectralClustering.cpp.

```
{
    std::cout << "It is a pathline data set? 1.Yes, 0.No." << std::endl;
    int pathlineOption;
    std::cin >> pathlineOption;
    assert(pathlineOption==1||pathlineOption==0);
    isPathlines = (pathlineOption==1);

    if(isPathlines)
    {
        IOHandler::expandArray(ds.dataMatrix,ds.dataVec,ds.dimension,ds.maxElements);
    }
    else
    {
        if(p.sampled==1)
            IOHandler::expandArray(ds.dataMatrix,ds.dataVec,ds.dimension,ds.maxElements);
        else if(p.sampled==2)
            IOHandler::sampleArray(ds.dataMatrix,ds.dataVec,ds.dimension,ds.maxElements);
        else if(p.sampled==3)
            IOHandler::uniformArcSampling(ds.dataMatrix,ds.dataVec,ds.dimension,ds.maxElements);
    }

    group = std::vector<int>(ds.dataMatrix.rows());

    /* the default value for streamline clustering is 2 normalized Laplacian */
    LaplacianOption = p.LaplacianOption;

    isDistSorted = p.isDistSorted;

    postProcessing = p.postProcessing;

    mMethod = p.mMethod;

    extractOption = p.extractOption;
}
```

4.5.4 Member Data Documentation

4.5.4.1 std::vector< string > SpectralClustering::activityList `[private]`

Definition at line 129 of file SpectralClustering.h.

4.5.4.2 std::vector< float > SpectralClustering::distRange `[private]`

Definition at line 154 of file SpectralClustering.h.

4.5.4.3 DataSet SpectralClustering::ds `[private]`

Definition at line 139 of file SpectralClustering.h.

4.5.4.4 int SpectralClustering::extractOption = -1 `[private]`

Definition at line 179 of file SpectralClustering.h.

4.5.4.5 std::vector< int > SpectralClustering::group `[private]`

Definition at line 124 of file SpectralClustering.h.

4.5.4.6 int SpectralClustering::initializationOption = -1 `[private]`

Definition at line 149 of file SpectralClustering.h.

4.5.4.7 bool SpectralClustering::isDistSorted = -1 `[private]`

Definition at line 169 of file SpectralClustering.h.

4.5.4.8 bool SpectralClustering::isOptimal `[private]`

Definition at line 189 of file SpectralClustering.h.

4.5.4.9 bool SpectralClustering::isPathlines `[private]`

Definition at line 204 of file SpectralClustering.h.

4.5.4.10 int SpectralClustering::LaplacianOption = -1 `[private]`

Definition at line 164 of file SpectralClustering.h.

4.5.4.11 float SpectralClustering::mMaxQuality = 0 `[private]`

Definition at line 355 of file SpectralClustering.h.

4.5.4.12 int SpectralClustering::mMethod = -1 `[private]`

Definition at line 360 of file SpectralClustering.h.

4.5.4.13 int SpectralClustering::normOption = -1 `[private]`

Definition at line 119 of file SpectralClustering.h.

4.5.4.14 int SpectralClustering::numberOfClusters = -1 `[private]`

Definition at line 144 of file SpectralClustering.h.

4.5.4.15 MetricPreparation SpectralClustering::object `[private]`

Definition at line 114 of file SpectralClustering.h.

4.5.4.16 int SpectralClustering::postProcessing = -1 `[private]`

Definition at line 174 of file SpectralClustering.h.

4.5.4.17 int SpectralClustering::presetNumber `[private]`

Definition at line 194 of file SpectralClustering.h.

4.5.4.18 bool SpectralClustering::readCluster `[private]`

Definition at line 199 of file SpectralClustering.h.

4.5.4.19 int SpectralClustering::SCALING `[private]`

Definition at line 184 of file SpectralClustering.h.

4.5.4.20 std::vector<float> SpectralClustering::sigmaVec `[private]`

Definition at line 159 of file SpectralClustering.h.

4.5.4.21 std::vector<string> SpectralClustering::timeList `[private]`

Definition at line 134 of file SpectralClustering.h.

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

- SpectralClustering.h
- SpectralClustering.cpp

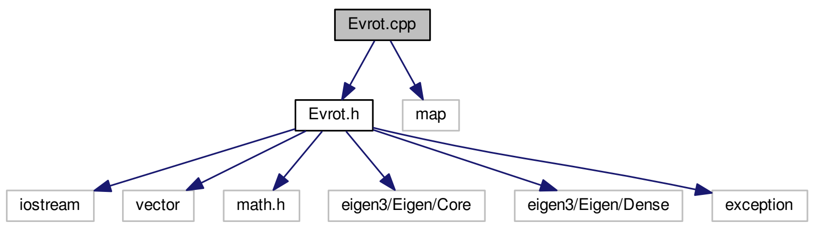
Chapter 5

File Documentation

5.1 Evrot.cpp File Reference

```
#include "Evrot.h"
#include <map>
```

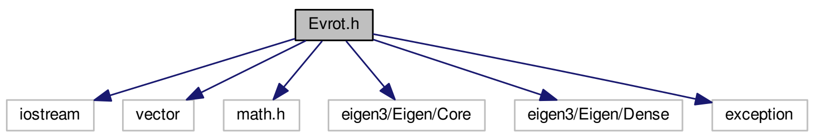
Include dependency graph for Evrot.cpp:



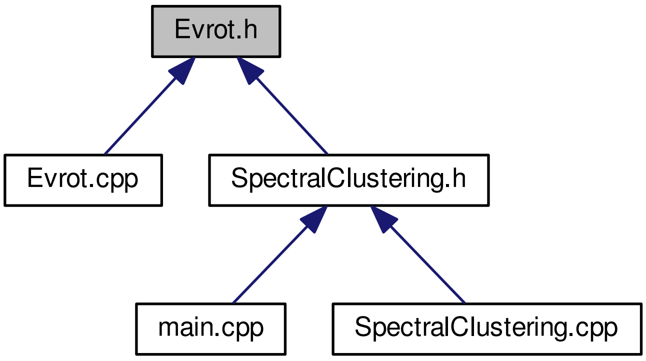
5.2 Evrot.h File Reference

```
#include <iostream>
#include <vector>
#include <math.h>
#include <eigen3/Eigen/Core>
#include <eigen3/Eigen/Dense>
#include <exception>
```

Include dependency graph for Evrot.h:



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



Classes

- class Evrot

Macros

- #define EPS 2.2204e-8

5.2.1 Macro Definition Documentation

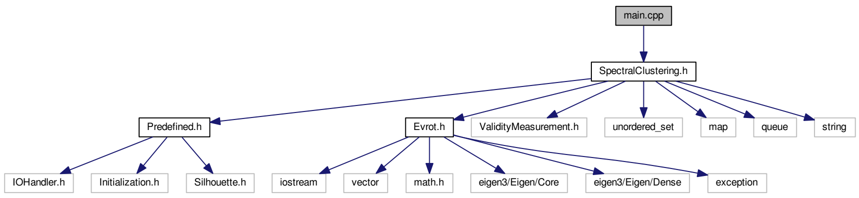
5.2.1.1 #define EPS 2.2204e-8

Definition at line 28 of file Evrot.h.

5.3 main.cpp File Reference

```
#include "SpectralClustering.h"
```

Include dependency graph for main.cpp:



Functions

- void setPara (Para &p)
- int main (int argc, char **argv)

5.3.1 Function Documentation

5.3.1.1 int main (int *argc,* char ** *argv*)

Definition at line 18 of file main.cpp.

```
{
    Para p;

    setPara(p);

    /* enable automatic option */
    bool automatic = true;

    SpectralClustering spectClus(argc, argv, p, automatic);

    spectClus.performClustering();

    return 0;
}
```

5.3.1.2 void setPara (Para & *p*)

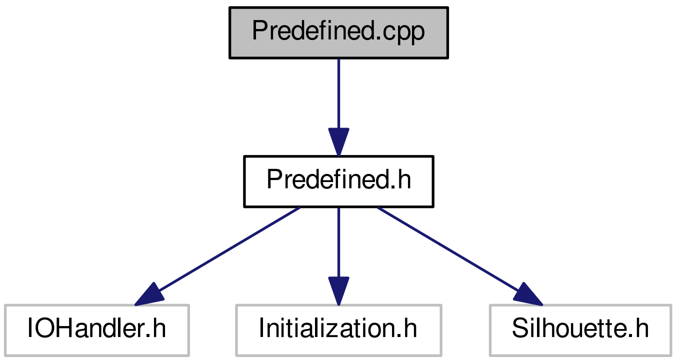
Definition at line 40 of file main.cpp.

```
{
    /* 1.directly filling with last vertex; 2. uniform sampling, 3. equal-arc sampling */
    p.sampled = 2;

    /* Laplacian option: 1.Normalized Laplacian, 2.Unsymmetric Laplacian */
    p.LaplacianOption = 1;

    /* local scaling by sorted distance: true, false */
    p.isDistSorted = true;

    /* post-processing method: 1.k-means, 2.eigenvector rotation*/
    std::cout << "Input the post-processing: 1.k-means, 2.eigenvector rotation: " << std::endl;
    std::cin >> p.postProcessing;
    assert(p.postProcessing==1 || p.postProcessing==2);

    /* derivative method for eigen rotation: 1.numerical derivative, 2.true derivative */
    p.mMethod = 2;

    /* extraction option, 1. centroid, closest and furthest, 2. median, 3. statistical representation */
    p.extractOption = 1;
}
```

5.4 Predefined.cpp File Reference

`#include "Predefined.h"`

Include dependency graph for Predefined.cpp:



Functions

- template<class T >
  void deleteVecElements (std::vector< T > &original, const T &first, const T &second)

5.4.1 Function Documentation

5.4.1.1 template<class T > void deleteVecElements (std::vector< T > & *original,* const T & *first,* const T & *second*)

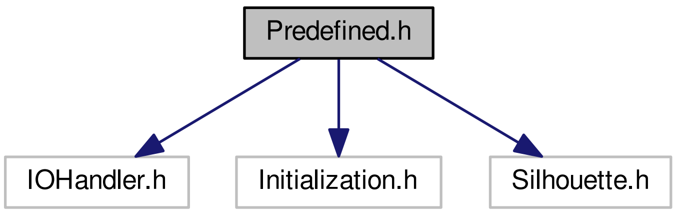
Definition at line 19 of file Predefined.cpp.

```
{
    std::size_t size = original.size();
    assert(size>2);
    vector<T> result(size-2);
    int tag = 0;
    for(int i=0;i<size;++i)
    {
        //meet with target elements, not copied
        if(original[i]==first || original[i]==second)
            continue;
        result[tag++]=original[i];
    }
    assert(tag==size-2);
    original = result;
}
```

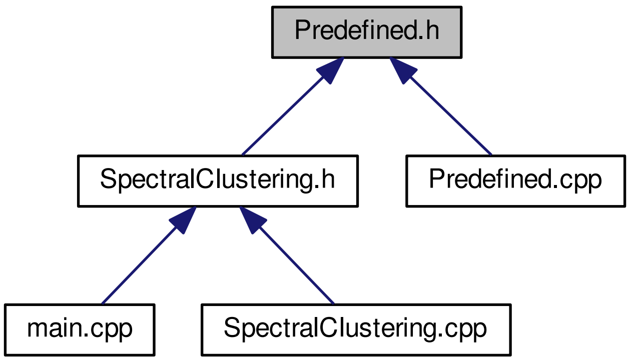
5.5 Predefined.h File Reference

```
#include "IOHandler.h"
#include "Initialization.h"
#include "Silhouette.h"
```

Include dependency graph for Predefined.h:



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



Classes

- struct DataSet
- struct Ensemble

Functions

- template<class T >
  void deleteVecElements (std::vector< T > &original, const T &first, const T &second)

5.5.1 Function Documentation

5.5.1.1 template< class T > void deleteVecElements (std::vector< T > & *original,* const T & *first,* const T & *second*)

Definition at line 19 of file Predefined.cpp.

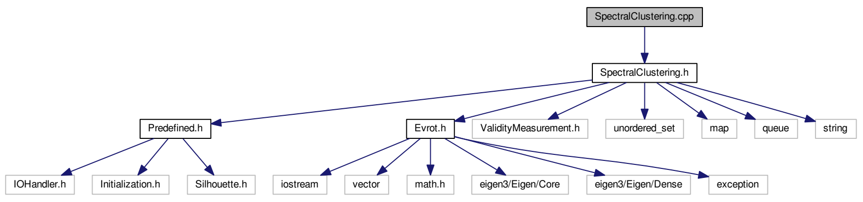
```
{
    std::size_t size = original.size();
    assert(size>2);
    vector<T> result(size-2);
    int tag = 0;
    for(int i=0;i<size;++i)
    {
        //meet with target elements, not copied
        if(original[i]==first || original[i]==second)
            continue;
        result[tag++]=original[i];
    }
    assert(tag==size-2);
    original = result;
}
```

5.6 README.md File Reference

5.7 SpectralClustering.cpp File Reference

`#include "SpectralClustering.h"`

Include dependency graph for SpectralClustering.cpp:



Functions

- void getMatrixPow (Eigen::DiagonalMatrix< float, Dynamic > &matrix, const float &powNumber)

5.7.1 Function Documentation

5.7.1.1 void getMatrixPow (Eigen::DiagonalMatrix< float, Dynamic > & *matrix,* const float & *powNumber*)

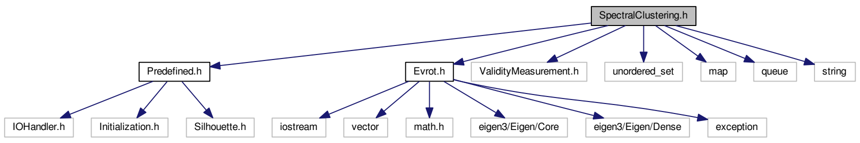
Definition at line 741 of file SpectralClustering.cpp.

```
{
    Eigen::VectorXf& m_v = matrix.diagonal();
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<m_v.size();++i)
        m_v(i) = pow(m_v(i), powNumber);
}
```

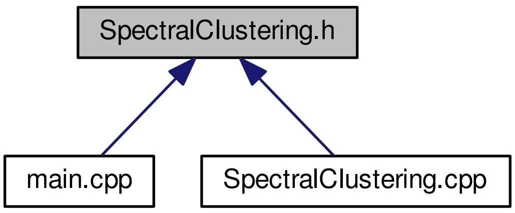
5.8 SpectralClustering.h File Reference

```
#include "Predefined.h"
#include "Evrot.h"
#include "ValidityMeasurement.h"
#include <unordered_set>
#include <map>
#include <queue>
#include <string>
```

Include dependency graph for SpectralClustering.h:



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



Classes

- struct Para
- class SpectralClustering

Macros

- #define GradientStep 0.3

Functions

- void getMatrixPow (Eigen::DiagonalMatrix< float, Dynamic > &matrix, const float &powNumber)

5.8.1 Macro Definition Documentation

5.8.1.1 #define GradientStep 0.3

Definition at line 31 of file SpectralClustering.h.

5.8.2 Function Documentation

5.8.2.1 void getMatrixPow (Eigen::DiagonalMatrix< float, Dynamic > & *matrix,* const float & *powNumber*)

Definition at line 741 of file SpectralClustering.cpp.

```
{
    Eigen::VectorXf& m_v = matrix.diagonal();
#pragma omp parallel for schedule(static) num_threads(8)
    for(int i=0;i<m_v.size();++i)
        m_v(i) = pow(m_v(i), powNumber);
}
```

Index