目次

図目次

表目次

第 1 章 序論

第 1節　背景

近年，情報通信社会の発展に伴いデータ量が増大し，日々多様なデータがコンピュータに蓄積されている．検索エンジンなどのインターネット上のサービスでは，蓄積されたビッグデータの解析や分類を行うことで，利用者に適切な情報を素早く送ることを可能にしている．ビッグデータを人の手によって分類することには困難が伴うため，計算機を用いて自動的にデータの分類を行うための技術であるクラスタリングが必要となる．クラスタリングとは，与えられたデータの個体間に存在する類似性に基づいて，個体をいくつかのクラスタと呼ばれるグループに分割を行う教師なし機械学習の手法である (図 1.1).

データをクラスタに分類した際に，それぞれのデータが各クラスタに属す度合いを表した値を帰属度と呼ぶ．帰属度が 0 と 1 のみで表され，それぞれのデータが各クラスタに明確に分類されるクラスタリングをハードクラスタリングと呼び，一方で帰属度が 0 と 1 の間の値で表され，データが属するクラスタを柔軟に表すことができるクラスタリングをファジィクラスタリングと呼ぶ．現実に存在しているデータには，明確に分類できるものだけでなく本質的に分類できない複雑なものも存在し，そういったデータの分類にはファジィクラスタリングが有効である．

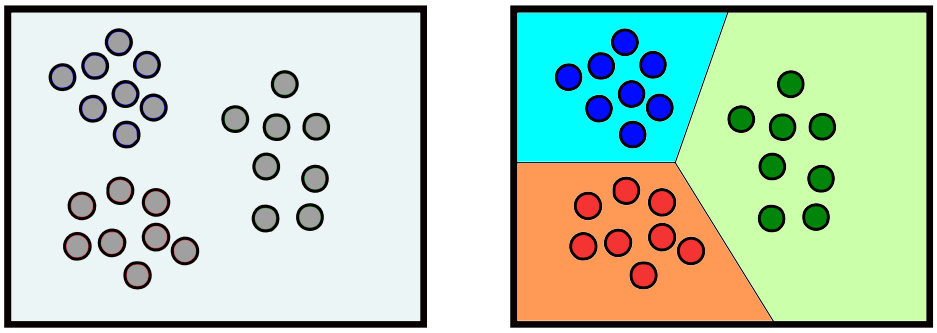
第 2節　目的

既存の手法における課題として，各クラスタのサイズに差がある場合，クラスタリングから有意な結果が得られないというものがある．ここで，クラスタのサイズとは，クラスタに属するデータの数と，そのクラスタに属するデータ間の類似度に基づくものであり，データの数が多い，または類似度が小さいクラスタをサイズが大きいクラスタとし，データの数が少ない，または類似度が大きいクラスタをサイズが大きいクラスタとする．現在，各クラスタのサイズを考慮してクラスタリングを行う手法が複数提案されており，本研究はそれらの手法について

各手法の特性を把握するとともに，最も有用な手法を発見することを目的とする．

第 3節　構成

本文書の構成を次に示す．第 2 章では，提案手法について説明する．第 3 章では，人工データ実験による各手法の特性比較を行う．第 4 章では，実データ実験による各手法の精度比較を行う．最後に第 5 章では，本文書の結論を述べる．また，付録では，プログラムソースを掲載している．



(a) クラスタリング前　　(b) クラスタリング後

図 1.1: クラスタリングについて

第 2 章
提案手法

第 1節　はじめに

本章では，本研究で提案するファジィクラスタリング手法について説明する．まず第 2 節で定義を示し，次に第 4 節から第 5 節で各手法の最適化問題と，各変数の更新式について述べる．

第 2節　定義

次節で述べるファジィクラスリングの最適化問題における各変数の定義について，表 2.1 に示す．

表 2.1: ファジイクラスタリングの最適化問題における定義

N	データ数	x_k	データ
C	クラスタ数	v_i	クラスタ中心
λ, m	ファジィ化パラメータ	$u_{i,k}$	帰属度
α_i	クラスタサイズ調整変数		

第 3節　sFCMA

Standard Fuzzy c-Means with vAriable controlling cluster size (sFCMA) [1] の最適化問題を以下に示す.

$$\underset{u,v,\alpha}{\text{minimize}} \sum_{i=1}^{C}\sum_{k=1}^{N}(\alpha_i)^{1-m}(u_{i,k})^m||x_k - v_i||_2^2 \tag{2.1}$$

$$\text{subject to } \sum_{i=1}^{C} u_{i,k} = 1,\ \sum_{i=1}^{C}\alpha_i = 1 \text{ and } m > 1,\ \alpha_i > 0 \tag{2.2}$$

次に，クラスタ中心 v_i の更新式を以下に示す.

$$v_i = \frac{\sum_{k=1}^{N}(u_{i,k})^m x_k}{\sum_{k=1}^{N}(u_{i,k})^m} \tag{2.3}$$

次に，帰属度 $u_{i,k}$ の更新式を以下に示す.

$$u_{i,k} = \frac{1}{\sum_{j=1}^{c}\frac{\alpha_j}{\alpha_i}\left(\frac{d_{j,k}}{d_{i,k}}\right)^{\frac{1}{1-m}}} \tag{2.4}$$

次に，クラスタサイズ調整変数 α_i の更新式を以下に示す.

$$\alpha_i = \frac{1}{\sum_{j=1}^{C}\left(\sum_{k=1}^{N}\frac{(u_{j,k})^m d_{j,k}}{(u_{i,k})^m d_{i,k}}\right)^{\frac{1}{m}}} \tag{2.5}$$

第 4節　eFCMA

Entropy-regularized Fuzzy c-Means vAriable controlling clusters size (eFCMA) [2] の最適化問題を以下に示す.

$$\underset{u,v,\alpha}{\text{minimize}} \sum_{i=1}^{C}\sum_{k=1}^{N}u_{i,k}||x_k - v_i||_2^2 + \lambda^{-1}\sum_{i=1}^{C}\sum_{k=1}^{N}u_{i,k}\log\left(\frac{u_{i,k}}{\alpha_i}\right) \tag{2.6}$$

$$\text{subject to } \sum_{i=1}^{C} u_{i,k} = 1,\ \sum_{i=1}^{C}\alpha_i = 1 \text{ and } \lambda > 0, \quad \alpha_i > 0 \tag{2.7}$$

次に，クラスタ中心 v_i の更新式を以下に示す.

$$v_i = \frac{\sum_{k=1}^{N}u_{i,k}x_k}{\sum_{k=1}^{N}u_{i,k}} \tag{2.8}$$

次に，帰属度 $u_{i,k}$ の更新式を以下に示す．

$$u_{i,k} = \frac{\pi_i \exp(-\lambda||x_k - v_i||_2^2)}{\sum_{j=1}^{C} \alpha_j \exp(-\lambda||x_k - v_j||_2^2)} \tag{2.9}$$

次に，クラスタサイズ調整変数 α_i の更新式を以下に示す．

$$\alpha_i = \frac{\sum_{k=1}^{N} u_{i,k}}{N} \tag{2.10}$$

第 5節　qFCMA

q-divergence based Fuzzy c-Means with vAriable controlling cluster size (qFCMA) [3] の最適化問題を以下に示す．

$$\underset{u,v,\alpha}{\text{minimize}} \sum_{i=1}^{C}\sum_{k=1}^{N} (\alpha_i)^{1-m}(u_{i,k})^m ||x_k - v_i||_2^2 + \frac{\lambda^{-1}}{m-1} \sum_{i=1}^{C}\sum_{k=1}^{N} (\alpha_i)^{1-m}(u_{i,k})^m \tag{2.11}$$

$$\text{subject to } \sum_{i=1}^{C} u_{i,k} = 1,\ \sum_{i=1}^{C} \alpha_i = 1 \text{ and } \lambda > 0,\ m > 1,\ \alpha_i > 0 \tag{2.12}$$

次に，クラスタ中心 v_i の更新式を以下に示す．

$$v_i = \frac{\sum_{k=1}^{N} (u_{i,k})^m x_k}{\sum_{k=1}^{N} (u_{i,k})^m} \tag{2.13}$$

次に，帰属度 $u_{i,k}$ の更新式を以下に示す．

$$u_{i,k} = \frac{\alpha_i (1 + \lambda(1-m)||x_i - v_k||_2^2)^{\frac{1}{1-m}}}{\sum_{j=1}^{C} \alpha_j (1 + \lambda(1-m)||x_j - v_k||_2^2)^{\frac{1}{1-m}}} \tag{2.14}$$

次に，クラスタサイズ調整変数 α_i の更新式を以下に示す．

$$\alpha_i = \frac{1}{\sum_{j=1}^{C} \left(\sum_{k=1}^{N} \frac{(u_{j,k})^m (1-\lambda(1-m)d_{j,k})}{(u_{i,k})^m (1-\lambda(1-m)d_{i,k})} \right)^{\frac{1}{m}}} \tag{2.15}$$

第 6節　おわりに

本章では，本研究で提案するファジィクラスタリング手法について説明した．まず第 2 節で定義を示し，次に第 4 節から 5 節で各手法の最適化問題と，各変数の更新式について述べた．

第 3 章
人工データによる実験

第 1節　はじめに

本章では，人工データを用いた実験について述べる．まず第 2 節で本実験で用いる人工データについて説明する．次に第 3 節でアルゴリズムについて述べる．最後に第 4 節で実験により得られた分類関数を用いて各手法の特性比較を行う．

第 2節　人工データについて

人工データとして，クラス数 2，各クラスのデータ数 50，合計データ数 100 のデータを平均値 $(-1, -1)$，標準偏差 (0.5, 0.5) 及び平均値 (1, 1)，標準偏差 (0.5, 0.5) のガウスサンプリングで生成したデータを用いた (図 3.1)．

第 3節　アルゴリズム

1. クラスタ中心をランダムに与える．
2. クラスタ中心を用いて帰属度を更新する．
3. 帰属度を用いてクラスタ中心及びクラスタサイズ調整変数を更新する．
4. 収束すれば終了し，そうでない場合は 2 に戻る．

第 4節　分類関数による特性比較

sFCMA の実験結果を図 3.2a, 3.2b に示す．パラメータ m を 2.00 から 1.01 に変化させたところ，分類関数は m の値が大きいほどファジィになり，小さいほどクリスプになることが分かった．

次に，eFCMA の実験結果を図 3.3a, 3.3b に示す．垂直軸は分類関数値を，底面はデータ空間を表す．網掛けで示されるのが分類関数であり，各点がデータを表している．パラメータ λ

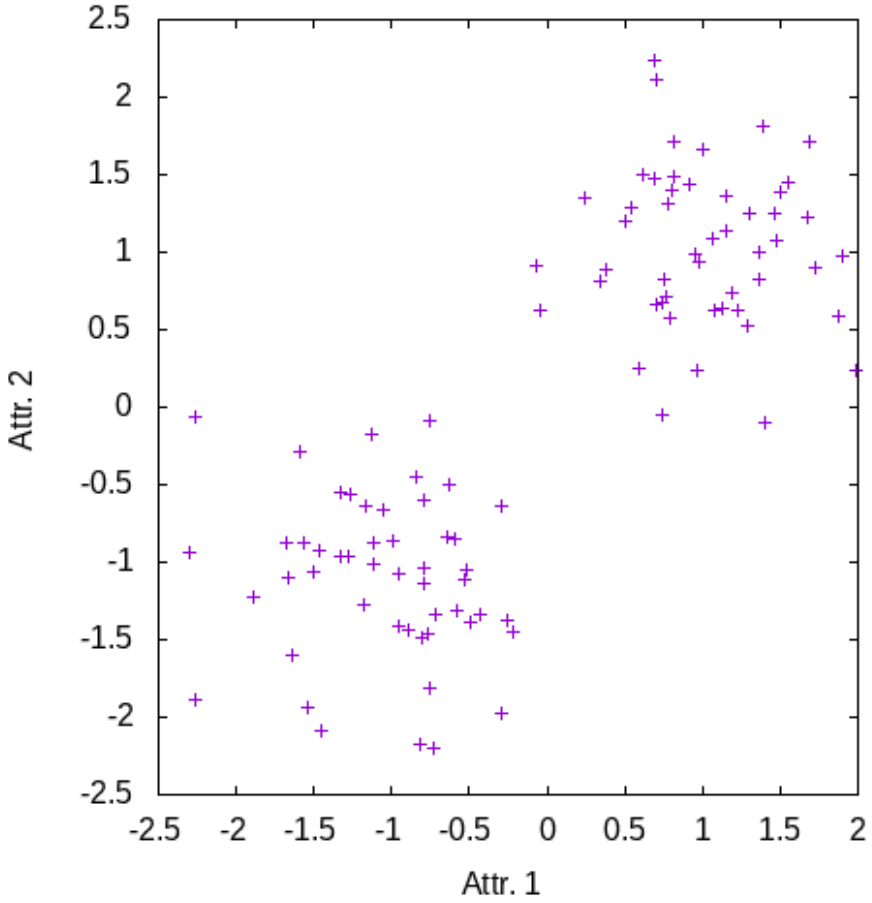
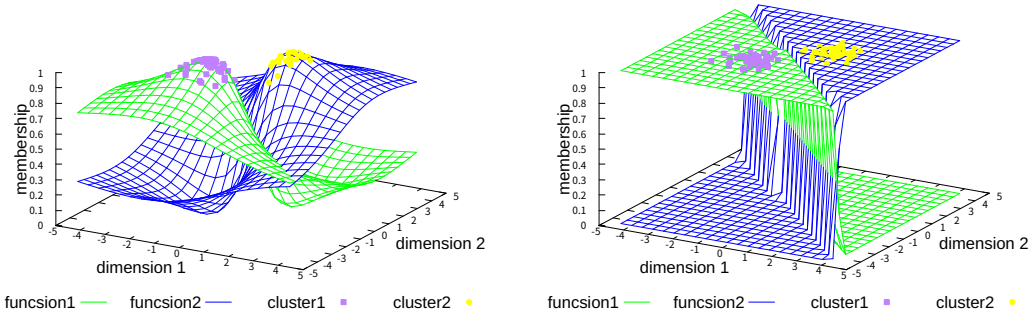


図 3.1: 人工データ

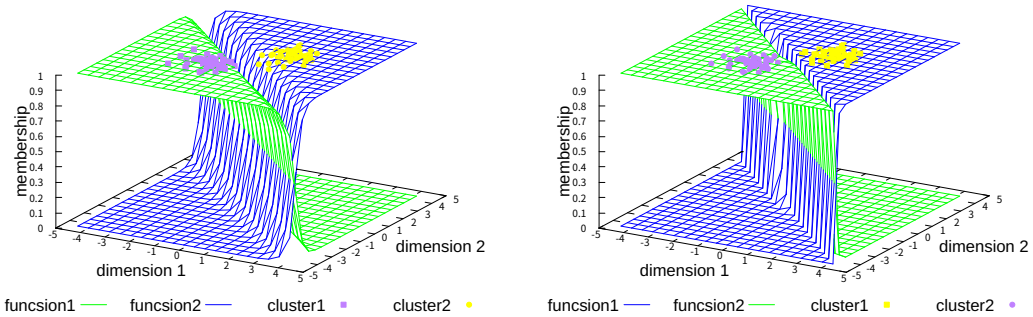
を 1.00 から 10.00 に変化させたところ，分類関数は λ の値が小さいほどファジィになり，大きいほどクリスプになることが分かった．

qFCMA の実験結果を図 3.4a, 3.4b, 3.4c に示す．こちらは，パラメータ (m, λ) の組み合わせとして，$(2.00, 1.00)$, $(1.01, 1.00)$, $(1.01, 10.00)$ の 3 通りでクラスタリングを行った．図 3.4a 及び図 3.4b の分類関数より，m の値が大きいほどファジィになり，小さいほどクリスプになることが分かった．また，図 3.4b 及び図 3.4c の分類関数より，λ の値が小さいほどファジィになり，大きいほどクリスプになることが分かった．また，qFCMA において $m-1 \to +0$ とすると sFCMA と同じ特性が得られ，$\lambda \to \infty$ とすると eFCMA と同様の特性を示すことがわかった．これらの実験結果より qFCMA は sFCMA と eFCMA の特性を併せ持つと言える．



(a) m=2.00　　(b) m=1.01

図 3.2: sFCMA の人工データの実験結果

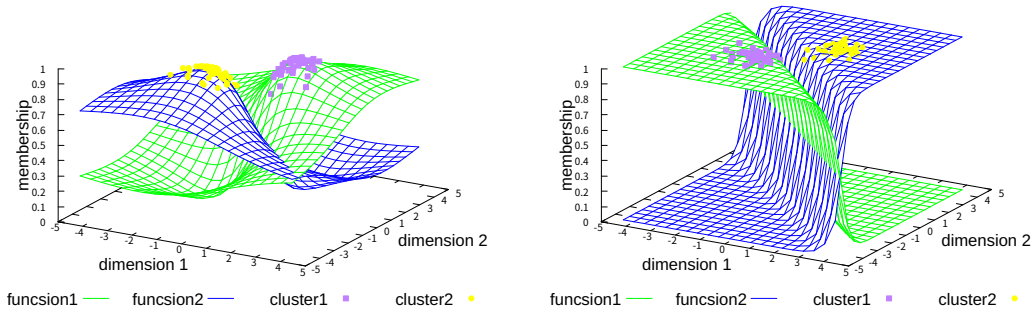


(a) λ=1.00　　(b) λ=10.00

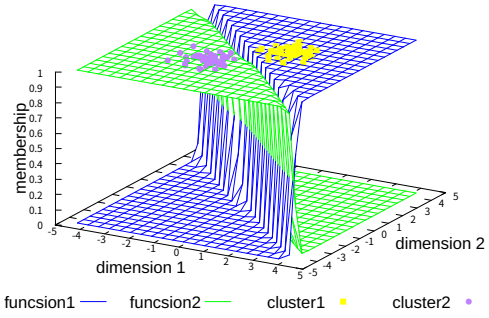
図 3.3: eFCMA の人工データの実験結果

第 5節　おわりに

本章では，人工データを用いた実験について述べた．まず第 2 節で本実験で用いる人工データについて説明した．次に第 3 節でアルゴリズムについて述べた．最後に第 4 節で実験により得られた分類関数を用いて各手法の特性比較を行った．



(a) $m = 2.00$, $\lambda = 1.00$　　(b) $m = 1.01$, $\lambda = 1.00$



(c) $m = 1.01$, $\lambda = 10.00$

図 3.4: qFCMA の人工データの実験結果

第 4 章
実データによる実験

第 1節　はじめに

本章では，実データを用いた実験について述べる．まず第 2 節で本実験で用いる実データについて説明する．次に第 3 節でアルゴリズムについて述べる．最後に第 4 節で実験により得られた評価指標を用いて各手法の精度比較を行う．

第 2節　実データについて

実データとしては，個体数 403，クラス数 4 の，被験者の勉強時間や試験結果などの 5 属性を収録した “User Knowledge Modeling Dasta Set” を用いた．

第 3節　アルゴリズム

1. 正解帰属度を用いて帰属度を初期化する．
2. 帰属度を用いてクラスタ中心及びクラスタサイズ調整変数を更新する．
3. 収束すれば終了し，そうでない場合は 2 に戻る．

第 4節　ARI による精度比較

sFCMA, eFCMA, qFCMA の実データ実験の結果について，それぞれ図 4.1, 4.2, 4.3 に示す．sFCMA では m の値を 1.1 から 3.0 まで 0.1 刻み，eFCMA では λ の値を 1 から 100 まで 1 刻み，qFCMA では m の値を 1.1 から 3.0 まで 0.1 刻み，λ の値を 1 から 100 まで 1 刻みで変化させた．

それぞれの手法の最高 ARI を表 4.1 に示す．

最も高い ARI を示した手法は sFCMA であり，他の 2 手法と比較して ARI に 0.4 以上の差が見られた．

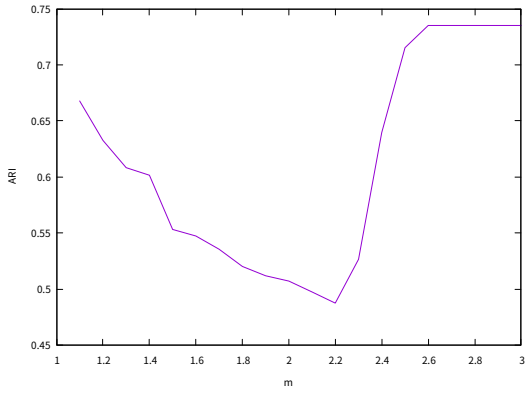


図 4.1: sFCMA の実データの実験結果

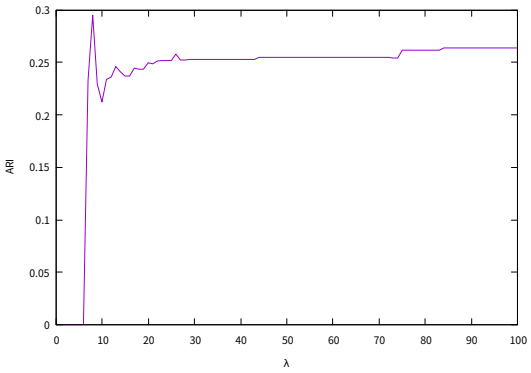


図 4.2: eFCMA の実データの実験結果

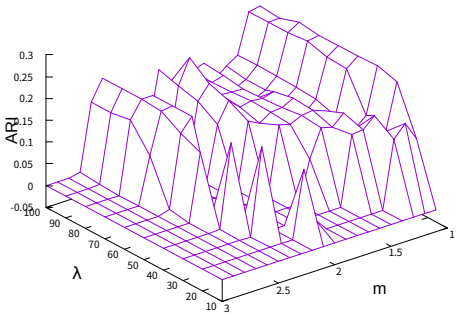


図 4.3: qFCMA の実データの実験結果

表 4.1: 各手法の ARI の最高値とパラメータ

手法名	ARI の最高値	パラメータ値
sFCMA	0.73515	$m = 3$
eFCMA	0.29500	$\lambda = 8$
qFCMA	0.26286	$\lambda = 80$, $m = 1.1$

第 5節　おわりに

本章では，実データを用いた実験について述べた．まず第 2 節で本実験で用いる人工データについて説明した．次に第 3 節でアルゴリズムについて述べた．最後に第 4 節で実験により得られた評価指標を用いて各手法の精度比較を行った．

第 5 章 結論

本文書では，第 2 章では，提案手法について説明した．第 3 章では，人工データ実験により各手法の特性比較を行った．第 4 章では，実データ実験により各手法の精度比較を行った．最後に第 5 章では，本文書の結論を述べた．また，付録では，プログラムソースを掲載した．

参考文献

[1] Miyamoto, S., Kurosawa, N.: “Controlling Cluster Volume Sizes in Fuzzy c-means Clustering”, Proc. SCIS&ISIS2004, pp. 1–4, (2004).

[2] Ichihashi, H., Honda, K., Tani, N.: “Gaussian Mixture PDF Approximation and Fuzzy c-means Clustering with Entropy Regularization”, Proc. 4th Asian Fuzzy System Symposium, pp. 217–221, (2000).

[3] Miyamoto, S., Ichihashi, H., and Honda, K.: Algorithms for Fuzzy Clustering, Springer (2008).

[4] 宮本 定明, 馬屋原 一孝, 向殿 政男:“ファジイ c-平均法とエントロピー正則化法におけるファジィ分類関数”, 日本ファジィ学会誌 Vol. 10, No. 3 pp. 548–557, (1998).

[5] Hubert, L., and Arabie, P.:“Comparing Partitions”, Journal of Classifcation, Vol. 2, No. 1, pp. 193–218, (1985).

感想

おいしかった.

謝辞

ありがとう.

付録 A プログラムソース

vector.h

```
1   #include<iostream>
2   #include<cstring>
3
4   #ifndef __VECTOR__
5   #define __VECTOR__
6
7   class Matrix;
8
9
10  class Vector{
11   private:
12    int Size;
13    double *Element;
14   public:
15    Vector(int size=0);
16    ~Vector(void);
17    explicit Vector(int dim, double arg, const char *s);
18    Vector(const Vector &arg);
19    Vector &operator=(const Vector &arg);
20    Vector(Vector &&arg);
21    Vector &operator=(Vector &&arg);
22    int size(void) const;
23    double operator[](int index) const;
24    double &operator[](int index);
25    Vector operator+(void) const;
26    Vector operator-(void) const;
27    Vector &operator+=(const Vector &rhs);
28    Vector &operator-=(const Vector &rhs);
29    Vector &operator*=(double rhs);
30    Vector &operator/=(double rhs);
31    Vector operator+(const Vector &rhs) const;
32    Vector operator-(const Vector &rhs) const;
```

```
33    double operator*(const Vector &rhs) const;
34    bool operator==(const Vector &rhs) const;
35    bool operator!=(const Vector &rhs) const;
36    Vector sub(int begin, int end) const;
37    void set_sub(int begin, int end, const Vector &arg);
38  };
39
40  Vector operator*(double lhs, const Vector &rhs);
41  Vector operator/(const Vector &lhs, double rhs);
42  std::ostream &operator<<(std::ostream &os, const Vector &rhs);
43  double max_norm(const Vector &arg);
44  double squared_norm(const Vector &arg);
45  double norm_square(const Vector &arg);
46  double L1norm_square(const Vector &arg);
47  Vector fraction(const Vector &arg);
48
49  #endif
```

vector.cxx

```
1   #include<iostream>
2   #include<cstdlib>
3   #include<cmath>
4   #include"vector.h"
5
6
7   Vector::Vector(int size) try :
8     Size(size), Element(new double[Size]){
9   }
10  catch(std::bad_alloc){
11    std::cerr << "Vector::Vector(int size): Out of Memory!" << std::endl;
12    throw;
13   }
14
15  Vector::~Vector(void){
16    delete []Element;
17  }
18
19  Vector::Vector(const Vector &arg) try :
20    Size(arg.Size), Element(new double[Size]){
21    for(int i=0;i<Size;i++){
22      Element[i]=arg.Element[i];
23    }
24  }
25  catch(std::bad_alloc){
26    std::cerr << "Vector::Vector(int size): Out of Memory!" << std::endl;
27    throw;
28   }
```

```
29
30  Vector::Vector(Vector &&arg)
31    : Size(arg.Size), Element(arg.Element){
32    arg.Size=0;
33    arg.Element=nullptr;
34  }
35
36  Vector::Vector(int dim, double arg, const char *s) try :
37    Size(dim), Element(new double[Size]){
38    if(strcmp(s, "all")!=0){
39      std::cerr << "Invalid string parameter" << std::endl;
40      exit(1);
41    }
42    for(int i=0;i<Size;i++){
43      Element[i]=arg;
44    }
45  }
46  catch(std::bad_alloc){
47    std::cerr << "Vector::Vector(int size): Out of Memory!" << std::endl;
48    throw;
49   }
50
51  Vector &Vector::operator=(const Vector &arg){
52    if(this==&arg)        return *this;
53    if(this->Size != arg.Size){
54      Size=arg.Size;
55      delete []Element;
56      try{
57        Element=new double[Size];
58      }
59      catch(std::bad_alloc){
60        std::cerr << "Out of Memory" << std::endl;
61        throw;
62      }
63    }
64    for(int i=0;i<Size;i++){
65      Element[i]=arg.Element[i];
66    }
67    return *this;
68  }
69
70  Vector &Vector::operator=(Vector &&arg){
71    if(this!=&arg){
72      Size=arg.Size;
73      Element=arg.Element;
74      arg.Size=0;
75      arg.Element=nullptr;
76    }
77    return *this;
78  }
```

```
79
80  int Vector::size(void) const{
81    return Size;
82  }
83
84  double Vector::operator[](int index) const{
85    return Element[index];
86  }
87
88  double &Vector::operator[](int index){
89    return Element[index];
90  }
91
92  Vector Vector::operator+(void) const{
93    return *this;
94  }
95
96  Vector Vector::operator-(void) const{
97    Vector result=*this;
98    for(int i=0;i<result.Size;i++){
99      result[i]*=-1.0;
100   }
101   return result;
102 }
103
104 Vector &Vector::operator+=(const Vector &rhs){
105   if(rhs.Size==0){
106     std::cout << "Vector::operator+=:Size 0" << std::endl;
107     exit(1);
108   }
109   else if(Size!=rhs.Size){
110     std::cout << "Vector::operator+=:Size Unmatched" << std::endl;
111     exit(1);
112   }
113   else{
114     for(int i=0;i<Size;i++){
115       Element[i]+=rhs[i];
116     }
117   }
118   return *this;
119 }
120
121 Vector &Vector::operator*=(double rhs){
122   for(int i=0;i<Size;i++){
123     Element[i]*=rhs;
124   }
125   return *this;
126 }
127
128 Vector &Vector::operator/=(double rhs){
```

```
129     for(int i=0;i<Size;i++){
130       Element[i]/=rhs;
131     }
132     return *this;
133   }
134
135
136   Vector &Vector::operator-=(const Vector &rhs){
137     if(rhs.Size==0){
138       std::cout << "Vector::operator-=:Size 0" << std::endl;
139       exit(1);
140     }
141     else if(Size!=rhs.Size){
142       std::cout << "Vector::operator-=:Size Unmatched" << std::endl;
143       exit(1);
144     }
145     else{
146       for(int i=0;i<Size;i++){
147         Element[i]-=rhs[i];
148       }
149     }
150     return *this;
151   }
152
153   Vector Vector::operator+(const Vector &rhs) const{
154     Vector result=*this;
155     return result+=rhs;
156   }
157
158   Vector Vector::operator-(const Vector &rhs) const{
159     Vector result=*this;
160     return result-=rhs;
161   }
162
163   Vector operator*(double lhs, const Vector &rhs){
164     if(rhs.size()==0){
165       std::cout << "Vector operator*:Size 0" << std::endl;
166       exit(1);
167     }
168     Vector result=rhs;
169     for(int i=0;i<result.size();i++){
170       result[i]*=lhs;
171     }
172     return result;
173   }
174
175   Vector operator/(const Vector &lhs, double rhs){
176     if(lhs.size()==0){
177       std::cout << "Vector operator/:Size 0" << std::endl;
178       exit(1);
```

```
179     }
180     Vector result=lhs;
181     return (result/=rhs);
182   }
183
184
185   std::ostream &operator<<(std::ostream &os, const Vector &rhs){
186     os << "(";
187     if(rhs.size()>0){
188       for(int i=0;;i++){
189         os << rhs[i];
190         if(i>=rhs.size()-1) break;
191         os << ", ";
192       }
193     }
194     os << ')';
195     return os;
196   }
197
198   bool Vector::operator==(const Vector &rhs) const{
199     if(Size!=rhs.size())  return false;
200     for(int i=0;i<Size;i++){
201       if(Element[i]!=rhs[i])      return false;
202     }
203     return true;
204   }
205
206   double max_norm(const Vector &arg){
207     if(arg.size()<1){
208       std::cout << "Can't calculate norm for 0-sized vector" << std::endl;
209       exit(1);
210     }
211     double result=fabs(arg[0]);
212     for(int i=1;i<arg.size();i++){
213       double tmp=fabs(arg[i]);
214       if(result<tmp)      result=tmp;
215     }
216     return result;
217   }
218
219   double squared_norm(const Vector &arg){
220     return sqrt(norm_square(arg));
221   }
222
223   double norm_square(const Vector &arg){
224     double result=0.0;
225     for(int i=0;i<arg.size();i++){
226       result+=arg[i]*arg[i];
227     }
228     return result;
```

```
229  }
230
231  double L1norm_square(const Vector &arg){
232    double result=0.0;
233    for(int i=0;i<arg.size();i++){
234      result+=fabs(arg[i]);
235    }
236    return result;
237  }
238
239  double Vector::operator*(const Vector &rhs) const{
240    if(Size<1 || rhs.size()<1 || Size!=rhs.size()){
241      std::cout << "Can't calculate innerproduct";
242      std::cout << "for 0-sized vector";
243      std::cout << "or for different sized vector";
244      std::cout << std::endl;
245      exit(1);
246    }
247    double result=Element[0]*rhs[0];
248    for(int i=1;i<Size;i++){
249      result+=Element[i]*rhs[i];
250    }
251    return result;
252  }
253
254  bool Vector::operator!=(const Vector &rhs) const{
255    if(Size!=rhs.size())  return true;
256    for(int i=0;i<Size;i++){
257      if(Element[i]!=rhs[i])      return true;
258    }
259    return false;
260  }
261
262  Vector Vector::sub(int begin, int end) const{
263    if(end<begin){
264      std::cerr << "Vector::sub:invalid parameter" << std::endl;
265      exit(1);
266    }
267    Vector result(end-begin+1);
268    for(int i=0;i<result.size();i++){
269      result[i]=Element[begin+i];
270    }
271    return result;
272  }
273
274  void Vector::set_sub(int begin, int end, const Vector &arg){
275    if(end<begin){
276      std::cerr << "Vector::sub:invalid parameter" << std::endl;
277      exit(1);
278    }
```

```
279     if(end-begin+1!=arg.size()){
280       std::cerr << "Vector::sub:invalid parameter" << std::endl;
281       exit(1);
282     }
283     for(int i=0;i<arg.size();i++){
284       Element[begin+i]=arg[i];
285     }
286     return;
287   }
288
289   Vector fraction(const Vector &arg){
290     Vector result(arg.size());
291     for(int i=0;i<result.size();i++){
292       result[i]=1.0/arg[i];
293     }
294     return result;
295   }
```

matrix.h

```
1   #include<iostream>
2   #include<cstring>
3   #include"vector.h"
4
5   #ifndef __MATRIX__
6   #define __MATRIX__
7
8   class Matrix{
9    private:
10    int Rows;
11    Vector *Element;
12   public:
13    //Matrix(int rows=0);
14    Matrix(int rows=0, int cols=0);
15    explicit Matrix(int dim, const char *s);
16    explicit Matrix(const Vector &arg, const char *s);
17    ~Matrix(void);
18    Matrix(const Matrix &arg);
19    Matrix &operator=(const Matrix &arg);
20    Matrix(Matrix &&arg);
21    Matrix &operator=(Matrix &&arg);
22    int rows(void) const;
23    int cols(void) const;
24    Vector operator[](int index) const;
25    Vector &operator[](int index);
26    Matrix operator+(void) const;
27    Matrix operator-(void) const;
28    Matrix &operator+=(const Matrix &rhs);
```

```
29    Matrix &operator-=(const Matrix &rhs);
30    Matrix &operator*=(double rhs);
31    Matrix &operator/=(double rhs);
32    Matrix sub(int row_begin,
33                    int row_end,
34                    int col_begin,
35                    int col_end) const;
36    void set_sub(int row_begin,
37                 int row_end,
38                 int col_begin,
39                 int col_end,
40                 const Matrix &arg);
41  };
42
43  Matrix operator+(const Matrix &lhs, const Matrix &rhs);
44  Matrix operator-(const Matrix &lhs, const Matrix &rhs);
45  Matrix operator*(double lhs, const Matrix &rhs);
46  Vector operator*(const Matrix &lhs, const Vector &rhs);
47  Matrix operator*(const Matrix &lhs, const Matrix &rhs);
48  Matrix operator*(const Matrix &lhs, double rhs);
49  Matrix operator/(const Matrix &lhs, double rhs);
50  std::ostream &operator<<(std::ostream &os, const Matrix &rhs);
51  bool operator==(const Matrix &lhs, const Matrix &rhs);
52  double max_norm(const Matrix &arg);
53  double frobenius_norm(const Matrix &arg);
54  Matrix transpose(const Matrix &arg);
55  Vector diag(const Matrix &arg);
56  Matrix pow(const Matrix &arg, double power);
57  Matrix transpose(const Vector &arg);
58  Matrix operator*(const Vector &lhs, const Matrix &rhs);
59
60  #endif
```

matrix.cxx

```
1   #include<iostream>
2   #include<cstdlib>
3   #include<cmath>
4   #include"vector.h"
5   #include"matrix.h"
6
7   Matrix::Matrix(int rows, int cols) try :
8     Rows(rows), Element(new Vector[Rows]){
9     for(int i=0;i<Rows;i++){
10      Element[i]=Vector(cols);
11    }
12  }
13  catch(std::bad_alloc){
```

```
14      std::cerr << "Out of Memory" << std::endl;
15      throw;
16    }
17
18  Matrix::Matrix(int dim, const char *s) try :
19    Rows(dim), Element(new Vector[Rows]){
20    if(strcmp(s, "I")!=0){
21      std::cerr << "Invalid string parameter" << std::endl;
22      exit(1);
23
24    }
25    for(int i=0;i<Rows;i++){
26      Element[i]=Vector(dim);
27    }
28    for(int i=0;i<Rows;i++){
29      for(int j=0;j<Rows;j++){
30        Element[i][j]=0.0;
31      }
32      Element[i][i]=1.0;
33    }
34  }
35  catch(std::bad_alloc){
36      std::cerr << "Out of Memory" << std::endl;
37      throw;
38    }
39
40  Matrix::Matrix(const Vector &arg, const char *s) try :
41    Rows(arg.size()), Element(new Vector[Rows]){
42    if(strcmp(s, "diag")!=0){
43      std::cerr << "Invalid string parameter" << std::endl;
44      exit(1);
45
46    }
47    for(int i=0;i<Rows;i++){
48      Element[i]=Vector(Rows);
49    }
50    for(int i=0;i<Rows;i++){
51      for(int j=0;j<Rows;j++){
52        Element[i][j]=0.0;
53      }
54      Element[i][i]=arg[i];
55    }
56  }
57  catch(std::bad_alloc){
58      std::cerr << "Out of Memory" << std::endl;
59      throw;
60    }
61
62  Matrix::~Matrix(void){
63    delete []Element;
```

```
64  }
65
66  Matrix::Matrix(const Matrix &arg) try :
67    Rows(arg.Rows), Element(new Vector[Rows]){
68    for(int i=0;i<Rows;i++){
69      Element[i]=arg.Element[i];
70    }
71  }
72  catch(std::bad_alloc){
73      std::cerr << "Out of Memory" << std::endl;
74      throw;
75   }
76
77  Matrix::Matrix(Matrix &&arg)
78    : Rows(arg.Rows), Element(arg.Element){
79    arg.Rows=0;
80    arg.Element=nullptr;
81  }
82
83  Matrix &Matrix::operator=(Matrix &&arg){
84    if(this==&arg){
85      return *this;
86    }
87    else{
88      Rows=arg.Rows;
89      Element=arg.Element;
90      arg.Rows=0;
91      arg.Element=nullptr;
92      return *this;
93    }
94  }
95
96  Matrix &Matrix::operator=(const Matrix &arg){
97    if(this==&arg)        return *this;
98    //Rows=arg.Rows; ここでは Rows を更新してはいけない
99    if(this->Rows != arg.Rows || this->cols() != arg.cols()){
100     Rows=arg.Rows;
101     delete []Element;
102     try{
103       Element=new Vector[Rows];
104     }
105     catch(std::bad_alloc){
106       std::cerr << "Out of Memory" << std::endl;
107       throw;
108     }
109   }
110   for(int i=0;i<Rows;i++){
111     Element[i]=arg.Element[i];
112   }
113   return *this;
```

```
114  }
115
116  int Matrix::rows(void) const{
117    return Rows;
118  }
119
120  int Matrix::cols(void) const{
121    return Element[0].size();
122  }
123
124  Vector Matrix::operator[](int index) const{
125    return Element[index];
126  }
127
128  Vector &Matrix::operator[](int index){
129    return Element[index];
130  }
131
132  Matrix Matrix::operator+(void) const{
133    return *this;
134  }
135
136  Matrix Matrix::operator-(void) const{
137    Matrix result=*this;
138    for(int i=0;i<result.Rows;i++){
139      result[i]=-1.0*result[i];
140    }
141    return result;
142  }
143
144  Matrix &Matrix::operator+=(const Matrix &rhs){
145    if(rhs.Rows==0){
146      std::cout << "Rows 0" << std::endl;
147      exit(1);
148    }
149    else if(Rows!=rhs.Rows){
150      std::cout << "Rows Unmatched" << std::endl;
151      exit(1);
152    }
153    else{
154      for(int i=0;i<Rows;i++){
155        Element[i]+=rhs[i];
156      }
157    }
158    return *this;
159  }
160
161  Matrix &Matrix::operator-=(const Matrix &rhs){
162    if(rhs.Rows==0){
163      std::cout << "Rows 0" << std::endl;
```

```
164       exit(1);
165     }
166     else if(Rows!=rhs.Rows){
167       std::cout << "Rows Unmatched" << std::endl;
168       exit(1);
169     }
170     else{
171       for(int i=0;i<Rows;i++){
172         Element[i]-=rhs[i];
173       }
174     }
175     return *this;
176   }
177
178   Matrix operator+(const Matrix &lhs, const Matrix &rhs){
179     Matrix result=lhs;
180     return result+=rhs;
181   }
182
183   Matrix operator-(const Matrix &lhs, const Matrix &rhs){
184     Matrix result=lhs;
185     return result-=rhs;
186   }
187
188   Matrix operator*(double lhs, const Matrix &rhs){
189     if(rhs.rows()==0){
190       std::cout << "Rows 0" << std::endl;
191       exit(1);
192     }
193     Matrix result=rhs;
194     for(int i=0;i<result.rows();i++){
195       result[i]=lhs*result[i];
196     }
197     return result;
198   }
199
200   Matrix &Matrix::operator/=(double rhs){
201     for(int i=0;i<Rows;i++){
202       Element[i]/=rhs;
203     }
204     return *this;
205   }
206
207   Matrix operator/(const Matrix &lhs, double rhs){
208     Matrix result(lhs);
209     return result/=rhs;
210   }
211
212   std::ostream &operator<<(std::ostream &os, const Matrix &rhs){
213     os << "(";
```

```
214     if(rhs.rows()>0){
215       for(int i=0;;i++){
216         os << rhs[i];
217         if(i>=rhs.rows()-1) break;
218         os << "\n";
219       }
220     }
221     os << ')';
222     return os;
223   }
224
225   bool operator==(const Matrix &lhs, const Matrix &rhs){
226     if(lhs.rows()!=rhs.rows())    return false;
227     for(int i=0;i<lhs.rows();i++){
228       if(lhs[i]!=rhs[i])  return false;
229     }
230     return true;
231   }
232
233   double abssum(const Vector &arg){
234     double result=fabs(arg[0]);
235     for(int i=1;i<arg.size();i++){
236       result+=fabs(arg[i]);
237     }
238     return result;
239   }
240
241   double max_norm(const Matrix &arg){
242     if(arg.rows()<1){
243       std::cout << "Can't calculate norm for 0-sized vector" << std::endl;
244       exit(1);
245     }
246     double result=abssum(arg[0]);
247     for(int i=1;i<arg.rows();i++){
248       double tmp=abssum(arg[i]);
249       if(result<tmp)      result=tmp;
250     }
251     return result;
252   }
253
254   double frobenius_norm(const Matrix &arg){
255     double result=0.0;
256     for(int i=0;i<arg.rows();i++){
257       for(int j=0;j<arg.cols();j++){
258         result+=arg[i][j]*arg[i][j];
259       }}
260     return sqrt(result);
261   }
262
263   Vector operator*(const Matrix &lhs, const Vector &rhs){
```

```
264     if(lhs.rows()<1 || lhs.cols()<1 || rhs.size()<1 || lhs.cols()!=rhs.size()){
265       std::cout << "operator*(const Matrix &, const Vector &):";
266       std::cout << "Can't calculate innerproduct ";
267       std::cout << "for 0-sized vector ";
268       std::cout << "or for different sized vector:";
269       std::cout << "lhs.Cols=" << lhs.cols() << ", ";
270       std::cout << "lhs.Rows=" << lhs.rows() << ", ";
271       std::cout << "rhs.Size=" << rhs.size();
272       std::cout << std::endl;
273       exit(1);
274     }
275     Vector result(lhs.rows());
276     for(int i=0;i<lhs.rows();i++){
277       result[i]=lhs[i]*rhs;
278     }
279     return result;
280   }
281
282   Matrix operator*(const Matrix &lhs, const Matrix &rhs){
283     if(lhs.rows()<1 || rhs.cols()<1 || lhs.cols()!=rhs.rows()){
284       std::cout << "Can't calculate innerproduct";
285       std::cout << "for 0-sized vector";
286       std::cout << "or for different sized vector";
287       std::cout << std::endl;
288       exit(1);
289     }
290     Matrix result(lhs.rows(), rhs.cols());
291     for(int i=0;i<result.rows();i++){
292       for(int j=0;j<result.cols();j++){
293         result[i][j]=0.0;
294         for(int k=0;k<lhs.cols();k++){
295           result[i][j]+=lhs[i][k]*rhs[k][j];
296         }
297       }}
298     return result;
299   }
300
301   Matrix Matrix::sub(int row_begin, int row_end,
302                              int col_begin, int col_end) const{
303
304     if(row_end<row_begin || col_end<col_begin){
305       std::cerr << "Matrix::sub:invalid parameter" << std::endl;
306       std::cerr << "row_begin:" << row_begin << std::endl;
307       std::cerr << "row_end:" << row_end << std::endl;
308       std::cerr << "col_begin:" << col_begin << std::endl;
309       std::cerr << "col_end:" << col_end << std::endl;
310       exit(1);
311     }
312     if(row_end>=this->rows() || col_end>=this->cols()){
313       std::cerr << "Matrix::sub:invalid parameter" << std::endl;
```

```
314       std::cerr << "row_end:" << row_end << std::endl;
315       std::cerr << "Rows:" << this->rows() << std::endl;
316       std::cerr << "col_end:" << col_end << std::endl;
317       std::cerr << "Cols:" << this->cols() << std::endl;
318       exit(1);
319     }
320     if(row_begin<0 || col_begin<0){
321       std::cerr << "Matrix::sub:invalid parameter" << std::endl;
322       std::cerr << "row_begin:" << row_begin << std::endl;
323       std::cerr << "col_begin:" << col_begin << std::endl;
324       exit(1);
325     }
326     Matrix result(row_end-row_begin+1, col_end-col_begin+1);
327     for(int i=0;i<result.rows();i++){
328       for(int j=0;j<result.cols();j++){
329         result[i][j]=Element[i+row_begin][j+col_begin];
330       }}
331     return result;
332   }
333
334   void Matrix::set_sub(int row_begin, int row_end,
335                        int col_begin, int col_end,
336                        const Matrix &arg){
337
338     if(row_end<row_begin || col_end<col_begin){
339       std::cerr << "Matrix::sub:invalid parameter" << std::endl;
340       exit(1);
341     }
342     for(int i=row_begin;i<=row_end;i++){
343       for(int j=col_begin;j<=col_end;j++){
344         Element[i][j]=arg[i-row_begin][j-col_begin];
345       }}
346     return;
347   }
348
349   Matrix transpose(const Matrix &arg){
350     Matrix result(arg.cols(), arg.rows());
351     for(int i=0;i<result.rows();i++){
352       for(int j=0;j<result.cols();j++){
353         result[i][j]=arg[j][i];
354       }}
355     return result;
356   }
357
358   Vector diag(const Matrix &arg){
359     if(arg.rows()!=arg.cols()){
360       std::cerr << "No Diag" << std::endl;
361       exit(1);
362     }
363     Vector result(arg.rows());
```

```
364     for(int i=0;i<result.size();i++){
365       result[i]=arg[i][i];
366     }
367     return result;
368   }
369
370   Matrix pow(const Matrix &arg, double power){
371     Matrix result(arg);
372     for(int i=0;i<result.rows();i++){
373       for(int j=0;j<result.cols();j++){
374         result[i][j]=pow(result[i][j],power);
375       }}
376     return result;
377   }
378
379   Matrix transpose(const Vector &arg){
380     Matrix result(1, arg.size());
381     for(int j=0;j<result.cols();j++){
382       result[0][j]=arg[j];
383     }
384     return result;
385   }
386
387   Matrix operator*(const Vector &lhs, const Matrix &rhs){
388     if(rhs.rows()!=1){
389       std::cerr << "Size unmatched for Vector*Matrix:" << rhs.rows() << ":" << rhs.cols
390       exit(1);
391     }
392     Matrix result(lhs.size(), rhs.cols());
393     for(int i=0;i<result.rows();i++){
394       for(int j=0;j<result.cols();j++){
395         result[i][j]=lhs[i]*rhs[0][j];
396       }}
397     return result;
398   }
```

hcm.cxx

```
1   #include"hcm.h"
2   #include <boost/math/special_functions/binomial.hpp>
3
4   Hcm::Hcm(int dimension,
5            int data_number,
6            int centers_number):
7     Data(data_number, dimension),
8     Centers(centers_number, dimension),
9     Tmp_Centers(centers_number, dimension),
10    Membership(centers_number, data_number),
```

```
11    Tmp_Membership(centers_number, data_number),
12    Alpha(centers_number),
13    Tmp_Alpha(centers_number),
14    Dissimilarities(centers_number, data_number),
15    CrispMembership(centers_number, data_number),
16    CorrectCrispMembership(centers_number, data_number),
17    ContingencyTable(centers_number+1, centers_number+1),
18    Iterates(0){
19    /***↓収束判定のために DBL_MAX に設定***/
20    for(int i=0;i<centers_number;i++){
21      Centers[i]=Vector(dimension);
22      for(int ell=0;ell<dimension;ell++){
23        Centers[i][ell]=DBL_MAX;
24      }
25    }
26    /***↓収束判定のために DBL_MAX に設定***/
27    for(int i=0;i<centers_number;i++){
28      for(int k=0;k<data_number;k++){
29        Membership[i][k]=DBL_MAX;
30      }
31    }
32  }
33
34  void Hcm::revise_dissimilarities(void){
35    for(int i=0;i<centers_number();i++){
36      for(int k=0;k<data_number();k++){
37        Dissimilarities[i][k]=norm_square(Data[k]-Centers[i]);
38      }}
39    return;
40  }
41
42  void Hcm::revise_membership(void){
43    Tmp_Membership=Membership;
44    for(int k=0;k<data_number();k++){
45      int min_index=0; double min_dissimilarity=Dissimilarities[0][k];
46      for(int i=1;i<centers_number();i++){
47        if(min_dissimilarity>Dissimilarities[i][k]){
48          min_index=i;
49          min_dissimilarity=Dissimilarities[i][k];
50        }
51      }
52      for(int i=0;i<centers_number();i++){
53        Membership[i][k]=0.0;
54      }
55      Membership[min_index][k]=1.0;
56    }
57    return;
58  }
59
60  void Hcm::revise_centers(void){
```

```
61    Tmp_Centers=Centers;
62    for(int i=0;i<centers_number();i++){
63      double denominator=0.0;
64      Vector numerator(Centers[i].size());
65      for(int ell=0;ell<numerator.size();ell++){
66        numerator[ell]=0.0;
67      }
68      for(int k=0;k<data_number();k++){
69        denominator+=Membership[i][k];
70        numerator+=Membership[i][k]*Data[k];
71      }
72      Centers[i]=numerator/denominator;
73    }
74    return;
75  }
76
77  int Hcm::dimension(void) const{
78    return Data[0].size();
79  }
80
81  int Hcm::data_number(void) const{
82    return Data.rows();
83  }
84
85  int Hcm::centers_number(void) const{
86    return Centers.rows();
87  }
88
89  Matrix Hcm::centers(void) const{
90    return Centers;
91  }
92
93  Matrix Hcm::tmp_centers(void) const{
94    return Tmp_Centers;
95  }
96
97  Matrix Hcm::data(void) const{
98    return Data;
99  }
100
101 Matrix Hcm::membership(void) const{
102   return Membership;
103 }
104
105 Matrix Hcm::tmp_membership(void) const{
106   return Tmp_Membership;
107 }
108
109 int &Hcm::iterates(void){
110   return Iterates;
```

```
111  }
112
113  Matrix Hcm::dissimilarities(void) const{
114    return Dissimilarities;
115  }
116
117  double &Hcm::data(int index1, int index2){
118    return Data[index1][index2];
119  }
120
121  double &Hcm::centers(int index1, int index2){
122    return Centers[index1][index2];
123  }
124
125  double &Hcm::membership(int row, int col){
126    return Membership[row][col];
127  }
128
129  double Hcm::objective(void) const{
130    return Objective;
131  }
132
133  void Hcm::set_objective(void){
134    Objective=0.0;
135    for(int i=0;i<centers_number();i++){
136      for(int k=0;k<data_number();k++){
137        Objective+=Membership[i][k]*Dissimilarities[i][k];
138      }}
139    return;
140  }
141
142  double &Hcm::dissimilarities(int index1, int index2){
143    return Dissimilarities[index1][index2];
144  }
145
146  void Hcm::set_crispMembership(void){
147    for(int k=0;k<data_number();k++){
148      for(int i=0;i<centers_number();i++){
149        CrispMembership[i][k]=0.0;
150      }
151      double max=-DBL_MAX;
152      int max_index=-1;
153      for(int i=0;i<centers_number();i++){
154        if(Membership[i][k]>max){
155          max=Membership[i][k];
156          max_index=i;
157        }
158      }
159      CrispMembership[max_index][k]=1.0;
160    }
```

```
161     return;
162   }
163
164   Matrix Hcm::crispMembership(void) const{
165     return CrispMembership;
166   }
167
168   double &Hcm::crispMembership(int index1, int index2){
169     return CrispMembership[index1][index2];
170   }
171
172   Matrix Hcm::correctCrispMembership(void) const{
173     return CorrectCrispMembership;
174   }
175
176   double &Hcm::correctCrispMembership(int index1, int index2){
177     return CorrectCrispMembership[index1][index2];
178   }
179
180   void Hcm::set_contingencyTable(void){
181     ContingencyTable.set_sub(0,centers_number()-1, 0, centers_number()-1,CrispMembershi
182
183     for(int i=0;i<ContingencyTable.rows()-1;i++){
184       ContingencyTable[i][ContingencyTable.cols()-1]=0.0;
185       for(int j=0;j<ContingencyTable.cols()-1;j++){
186         ContingencyTable[i][ContingencyTable.cols()-1]+=ContingencyTable[i][j];
187       }
188     }
189     for(int j=0;j<ContingencyTable.cols()-1;j++){
190       ContingencyTable[ContingencyTable.rows()-1][j]=0.0;
191       for(int i=0;i<ContingencyTable.rows()-1;i++){
192         ContingencyTable[ContingencyTable.rows()-1][j]+=ContingencyTable[i][j];
193       }
194     }
195     ContingencyTable[ContingencyTable.rows()-1][ContingencyTable.cols()-1]=data_number(
196     return;
197   }
198
199   Matrix Hcm::contingencyTable(void) const{
200     return ContingencyTable;
201   }
202
203   double combination(int n, int k){
204     if(n<k) return 0.0;
205    return boost::math::binomial_coefficient<double>(n, k);
206   }
207
208   double Hcm::ARI(void) const{
209     double Index=0.0;
210     for(int i=0;i<ContingencyTable.rows()-1;i++){
```

```
211       for(int j=0;j<ContingencyTable.cols()-1;j++){
212         Index+=ContingencyTable[i][j]*ContingencyTable[i][j];
213       }
214     }
215     Index=0.5*(Index-ContingencyTable[ContingencyTable.rows()-1][ContingencyTable.cols(
216     //  std::cout << "Index:" << Index << std::endl;
217     double ExpectedIndexI=0.0;
218     for(int i=0;i<ContingencyTable.rows()-1;i++){
219       ExpectedIndexI+=combination(ContingencyTable[i][ContingencyTable.cols()-1], 2);
220     }
221     //  std::cout << "ExpectedIndexI:" << ExpectedIndexI << std::endl;
222     double ExpectedIndexJ=0.0;
223     for(int j=0;j<ContingencyTable.cols()-1;j++){
224       ExpectedIndexJ+=combination(ContingencyTable[ContingencyTable.rows()-1][j], 2);
225     }
226     //  std::cout << "ExpectedIndexJ:" << ExpectedIndexJ << std::endl;
227     double ExpectedIndex=ExpectedIndexI*ExpectedIndexJ/combination(ContingencyTable[Con
228     //  std::cout << "Denom:" << combination(ContingencyTable[ContingencyTable.rows()-1
229     double MaxIndex=0.5*(ExpectedIndexI+ExpectedIndexJ);
230
231     return (Index-ExpectedIndex)/(MaxIndex-ExpectedIndex);
232   }
233
234   Vector &Hcm::data(int index1){
235     return Data[index1];
236   }
237
238   Vector &Hcm::centers(int index1){
239     return Centers[index1];
240   }
241
242   Vector Hcm::alpha(void) const{
243     return Alpha;
244   }
245
246   double &Hcm::alpha(int index1){
247     return Alpha[index1];
248   }
249
250   Vector Hcm::tmp_alpha(void) const{
251     return Tmp_Alpha;
252   }
```

hcm.h

```
1   #include<cmath>
2   #include<cfloat>
3   #include"matrix.h"
```

```
 4
 5  #ifndef __HCM__
 6  #define __HCM__
 7
 8  class Hcm{
 9   protected:
10    Matrix Data, Centers, Tmp_Centers;
11    Matrix Membership, Tmp_Membership, Dissimilarities;
12    Matrix CrispMembership, CorrectCrispMembership, ContingencyTable;
13    Vector Alpha,Tmp_Alpha;
14    int Iterates;
15    double Objective;
16   public:
17    Hcm(int dimension,
18        int data_number,
19        int centers_number);
20    virtual void revise_membership(void);
21    virtual void revise_dissimilarities(void);
22    virtual void revise_centers(void);
23    int dimension(void) const;
24    int data_number(void) const;
25    int centers_number(void) const;
26    Matrix centers(void) const;
27    Matrix tmp_centers(void) const;
28    Matrix data(void) const;
29    Matrix membership(void) const;
30    Matrix tmp_membership(void) const;
31    Vector alpha(void) const;
32    double &alpha(int index);
33    Vector tmp_alpha(void) const;
34    int &iterates(void);
35    Matrix dissimilarities(void) const;
36    double &data(int index1, int index2);
37    Vector &data(int index1);
38    double &centers(int index1, int index2);
39    Vector &centers(int index1);
40    double &membership(int index1, int index2);
41    double &dissimilarities(int index1, int index2);
42    void set_objective(void);
43    double objective(void) const;
44    void set_crispMembership(void);
45    Matrix crispMembership(void) const;
46    double &crispMembership(int index1, int index2);
47    Matrix correctCrispMembership(void) const;
48    double &correctCrispMembership(int index1, int index2);
49    void set_contingencyTable(void);
50    Matrix contingencyTable(void) const;
51    double ARI(void) const;
52  };
53
```

```
54  #endif
```

sfcma.h

```
1   #include"hcm.h"
2   #include"sfcm.h"
3
4   #ifndef __SFCMA__
5   #define __SFCMA__
6
7   class Sfcma: virtual public Hcm, public Sfcm{
8    public:
9     Sfcma(const int &dimension,
10          const int &data_number,
11          const int &centers_number,
12          const double &fuzzifierEm);
13    virtual void revise_membership(void);
14    virtual void revise_centers(void);
15    virtual void revise_alpha(void);
16  };
17  #endif
```

sfcma.cxx

```
1   #include"sfcma.h"
2
3   Sfcma::Sfcma(const int &dimension,
4                const int &data_number,
5                const int &centers_number,
6                const double &fuzzifierEm) :
7     Hcm(dimension, data_number, centers_number),
8     Sfcm(dimension, data_number, centers_number,fuzzifierEm){
9   }
10
11  void Sfcma::revise_membership(void){
12    Tmp_Membership=Membership;
13    for(int k=0;k<data_number();k++){
14      int numZeroDissimilarities=0;
15      Vector indexZeroDissimilarities(centers_number(), 0.0, "all");
16      for(int i=0;i<centers_number();i++){
17        if(Dissimilarities[i][k]==0.0){
18          numZeroDissimilarities++;
19          indexZeroDissimilarities[i]=1.0;
20        }
21      }
```

```
22      if(numZeroDissimilarities!=0){
23        for(int i=0;i<centers_number();i++){
24          Membership[i][k]=indexZeroDissimilarities[i]/numZeroDissimilarities;
25        }
26      }
27      else{
28        for(int i=0;i<centers_number();i++){
29          double denominator=0.0;
30          for(int j=0;j<centers_number();j++){
31            denominator+=Alpha[j]/Alpha[i]
32              *pow(Dissimilarities[i][k]/Dissimilarities[j][k],
33              1.0/(FuzzifierEm-1.0));
34          }
35          Membership[i][k]=1.0/denominator;
36        }
37      }
38    }
39    return;
40  }
41
42  void Sfcma::revise_centers(void){
43    Sfcm::revise_centers();
44    return;
45  }
46
47  void Sfcma::revise_alpha(void){
48    Tmp_Alpha=Alpha;
49    double denominator=0.0;
50    for(int j=0;j<centers_number();j++){
51      double tmp1=0.0;
52      for(int k=0;k<data_number();k++){
53        tmp1+=pow(Membership[j][k],FuzzifierEm)*Dissimilarities[j][k];
54      }
55      denominator+=pow(tmp1,1.0/FuzzifierEm);
56    }
57    for(int i=0;i<centers_number();i++){
58      double tmp2=0.0;
59      for(int k=0;k<data_number();k++){
60        tmp2+=pow(Membership[i][k],FuzzifierEm)*Dissimilarities[i][k];
61      }
62      Alpha[i]=pow(tmp2,1.0/FuzzifierEm)/denominator;
63    }
64    return;
65  }
```

efcma.h

```
1
```

```
 2  #include"efcm.h"
 3
 4  #ifndef __EFCMA__
 5  #define __EFCMA__
 6
 7  class Efcma: public Efcm {
 8  public:
 9    Efcma(
10         int dimension,
11         int data_number,
12         int centers_number,
13         double fuzzifierLambda);
14    void revise_membership(void);
15    void revise_alpha(void);
16  };
17
18  #endif
19
```

efcma.cxx

```
 1
 2  #include"efcma.h"
 3
 4  Efcma::Efcma(int dimension,
 5               int data_number,
 6               int centers_number,
 7               double fuzzifierLambda)
 8    : Hcm(dimension, data_number, centers_number),
 9      Efcm(dimension, data_number, centers_number, fuzzifierLambda){
10  }
11
12  void Efcma::revise_membership(void){
13    Tmp_Membership=Membership;
14    for(int k=0;k<data_number();k++){
15      for(int i=0;i<centers_number();i++){
16        double denominator=0.0;
17        for(int j=0;j<centers_number();j++){
18          denominator+=(Alpha[j]/Alpha[i])*exp(FuzzifierLambda*(Dissimilarities[i][k]-D
19        }
20        Membership[i][k]=1.0/denominator;
21      }
22    }
23    return;
24  }
25
26  void Efcma::revise_alpha(void){
27    Tmp_Alpha=Alpha;
```

```
28    for(int i=0;i<centers_number();i++){
29      double numerator=0;
30      for(int k=0;k<data_number();k++){
31        numerator+=Membership[i][k];
32      }
33      Alpha[i]=numerator/data_number();
34    }
35    return;
36  }
```

qfcma.h

```
1   #include"efcma.h"
2   #include"sfcma.h"
3
4   #ifndef __QFCMA__
5   #define __QFCMA__
6
7   class Qfcma: public Efcma, public Sfcma{
8
9   public:
10    Qfcma(int dimension,
11          int data_number,
12          int centers_number,
13          double fuzzifierEm,
14          double fuzzifierLambda);
15    virtual void revise_membership(void);
16    virtual void revise_centers(void);
17    virtual void revise_alpha(void);
18  };
19
20  #endif
21
```

qfcma.cxx

```
1   #include"qfcma.h"
2
3   Qfcma::Qfcma(int dimension,
4                int data_number,
5                int centers_number,
6                double fuzzifierEm,
7                double fuzzifierLambda) :
8     Hcm(dimension, data_number, centers_number),
9     Sfcma(dimension, data_number, centers_number,fuzzifierEm),
```

```
10    Efcma(dimension, data_number, centers_number,fuzzifierLambda){
11  }
12
13  void Qfcma::revise_membership(void){
14    Tmp_Membership=Membership;
15    for(int k=0;k<data_number();k++){
16      int numZeroDissimilarities=0;
17      Vector indexZeroDissimilarities(centers_number(), 0.0, "all");
18      for(int i=0;i<centers_number();i++){
19        if(Dissimilarities[i][k]==0.0){
20          numZeroDissimilarities++;
21          indexZeroDissimilarities[i]=1.0;
22        }
23      }
24      if(numZeroDissimilarities!=0){
25        for(int i=0;i<centers_number();i++){
26          Membership[i][k]=indexZeroDissimilarities[i]/numZeroDissimilarities;
27        }
28      }
29      else{
30        for(int i=0;i<centers_number();i++){
31          double denominator=0.0;
32          for(int j=0;j<centers_number();j++){
33            denominator+=Alpha[j]/Alpha[i]
34              *pow((1.0-FuzzifierLambda*(1.0-FuzzifierEm)
35                    *Dissimilarities[j][k])
36                  /(1.0-FuzzifierLambda*(1.0-FuzzifierEm)
37                    *Dissimilarities[i][k])
38                   ,1.0/(1.0-FuzzifierEm));
39          }
40          Membership[i][k]=1.0/denominator;
41        }
42      }//else
43    }//k
44    return;
45  }
46
47  void Qfcma::revise_centers(void){
48    Sfcma::revise_centers();
49    return;
50  }
51
52  void Qfcma::revise_alpha(void){
53    Tmp_Alpha=Alpha;
54    double denominator=0.0;
55    for(int j=0;j<centers_number();j++){
56      double tmp1=0.0;
57      for(int k=0;k<data_number();k++){
58        tmp1+=pow(Membership[j][k],FuzzifierEm)
59          *(1.0-FuzzifierLambda*(1.0-FuzzifierEm)*Dissimilarities[j][k]);
```

```
60        }
61        denominator+=pow(tmp1,1.0/FuzzifierEm);
62      }
63      for(int i=0;i<centers_number();i++){
64        double tmp2=0.0;
65        for(int k=0;k<data_number();k++){
66          tmp2+=pow(Membership[i][k],FuzzifierEm)
67            *(1.0-FuzzifierLambda*(1.0-FuzzifierEm))*Dissimilarities[i][k];
68        }
69        Alpha[i]=pow(tmp2,1/FuzzifierEm)/denominator;
70      }
71      return;
72    }
```