

# 目次

第 1 章	序論	1
1	背景 . . . . .	1
2	目的 . . . . .	1
3	構成 . . . . .	1
第 2 章	提案手法	2
1	はじめに . . . . .	2
2	定義 . . . . .	2
3	eFCMA . . . . .	2
4	sFCMA . . . . .	3
5	qFCMA . . . . .	3
6	アルゴリズム . . . . .	4
7	おわりに . . . . .	4
第 3 章	人工データによる実験	5
1	はじめに . . . . .	5
2	人工データについて . . . . .	5
3	分類関数による特性比較 . . . . .	5
4	おわりに . . . . .	5
第 4 章	実データによる実験	6
1	はじめに . . . . .	6
2	実データについて . . . . .	6
3	ARI による精度比較 . . . . .	6
4	おわりに . . . . .	6
第 5 章	結論	7

目次	ii
----	----

---

参考文献	8
------	---

感想	9
----	---

謝辞	10
----	----

付録 A    プログラムソース	11
------------------	----

vector.h . . . . .	11
--------------------	----

vector.cxx . . . . .	12
----------------------	----

matrix.h . . . . .	18
--------------------	----

matrix.cxx . . . . .	19
----------------------	----

# 图目次

# 表目次

# 第 1 章

## 序論

### 第 1 節 背景

第 1 小節 ファジィクラスタリング

第 2 小節 クラスタサイズ調整変数

### 第 2 節 目的

### 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．

## 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 3 節から 5 節で各手法の最適化問題と，その導出について述べる．最後に第 6 節でアルゴリズムについて述べる．

### 第 2 節 定義

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

### 第 3 節 eFCMA

$$\underset{u,v,\alpha}{\text{minimize}} \sum_{i=1}^C \sum_{k=1}^N u_{i,k} \|x_k - v_i\|_2^2 + \lambda^{-1} \sum_{i=1}^C \sum_{k=1}^N u_{i,k} \log\left(\frac{u_{i,k}}{\alpha_i}\right) \quad (2.1)$$

$$\text{subject to } \sum_{i=1}^C u_{i,k} = 1, \sum_{i=1}^C \alpha_i = 1 \text{ and } \lambda > 0, \quad \alpha_i > 0 \quad (2.2)$$

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

$$u_{i,k} = \frac{\pi_i \exp(-\lambda \|x_k - v_i\|_2^2)}{\sum_{j=1}^C \alpha_j \exp(-\lambda \|x_k - v_j\|_2^2)} \quad (2.4)$$

$$\alpha_i = \frac{\sum_{k=1}^N u_{i,k}}{N} \quad (2.5)$$

## 第 4 節 sFCMA

$$\underset{u,v,\alpha}{\text{minimize}} \sum_{i=1}^C \sum_{k=1}^N (\alpha_i)^{1-m} (u_{i,k})^m \|x_k - v_i\|_2^2 \quad (2.6)$$

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

$$v_i = \frac{\sum_{k=1}^N (u_{i,k})^m x_k}{\sum_{k=1}^N (u_{i,k})^m} \quad (2.8)$$

$$u_{i,k} = \frac{1}{\sum_{j=1}^C \frac{\alpha_j}{\alpha_i} \left( \frac{d_{j,k}}{d_{i,k}} \right)^{\frac{1}{1-m}}} \quad (2.9)$$

$$\alpha_i = \frac{1}{\sum_{j=1}^C \left( \sum_{k=1}^N \frac{(u_{j,k})^m d_{j,k}}{(u_{i,k})^m d_{i,k}} \right)^{\frac{1}{m}}} \quad (2.10)$$

## 第 5 節 qFCMA

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

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

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

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

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

## 第 6 節 アルゴリズム

## 第 7 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 3 節から 5 節で各手法の最適化問題と，その導出について述べた．最後に第 6 節でアルゴリズムについて述べた．



## 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

### 第 3 節 分類関数による特性比較

### 第 4 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．

## 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

### 第 3 節 ARI による精度比較

### 第 4 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] 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).
- [2] Miyamoto, S., Kurosawa, N.: “Controlling Cluster Volume Sizes in Fuzzy c-means Clustering”, Proc. SCIS&ISIS2004, pp. 1–4, (2004).
- [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: “ファジィ  $c$ -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 Classification, 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 Llnorm_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     }
261     return sqrt(result);
262 }
263
264 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     }
299     return result;
300 }
301
302 Matrix Matrix::sub(int row_begin, int row_end,
303                    int col_begin, int col_end) const{
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 }
332 return result;
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     }
347     return;
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     }
356     return result;
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     }
377     return result;
378 }
379
380 Matrix transpose(const Vector &arg){
381     Matrix result(1, arg.size());
382     for(int j=0;j<result.cols();j++){
383         result[0][j]=arg[j];
384     }
385     return result;
386 }
387
388 Matrix operator*(const Vector &lhs, const Matrix &rhs){
389     if(rhs.rows()!=1){
390         std::cerr << "Size unmatched for Vector*Matrix:" << rhs.rows() << ":" << rhs.cols() << "\n";
391         exit(1);
392     }
393     Matrix result(lhs.size(), rhs.cols());
394     for(int i=0;i<result.rows();i++){
395         for(int j=0;j<result.cols();j++){
396             result[i][j]=lhs[i]*rhs[0][j];
397         }
398     }
399     return result;
400 }
```