Klassen für lineare Systeme

Horst Hollatz

1. September 2005

Zusammenfassung

Mittels der Programmiersprache C++ werden Klassenvorlagen für ein- und zweidimensionale sowie obere Dreiecksfelder definiert, die der Verwaltung beliebiger Datentypen dienen. Davon abgeleitet sind die Klassen für Vektoren, allgemeine Matrizen, quadratische Matrizen und symmetrische Matrizen (inklusive symmetrische Bandmatrizen). Die Methoden und Operatoren dieser Klassen gestatten die Darstellung von Operationen analog zur Matrizenrechnung. Als Gleichungslöser gibt es in Abhängigkeit von der Klasse das Lösen mittels LU-Zerlegung, mittels LDLT-Zerlegung, mittels QR-Zerlegung (auch Lösen linearer Ausgleichsprobleme, Regularisierung) und mittels konjugiertem Gradientenverfahren. Die Klassen arbeiten mit dynamischen Komponenten; durch Streichen oder Hinzufügen von Funktionen lassen sie sich leicht spezifischen Aufgaben anpassen.

1. Vektoren auf Rechnern

Vektoren sind i. a. nicht auf Rechnern darstellbar. Dies erkennt man bereits daran, daß man z. B. keine stetigen Funktionen auf einem Rechner darstellen kann. Man ist daher gut beraten, sich auf solche Vektoren zu beschränken, die n-Tupel mit Komponenten gleichen Typs sind. Außerdem sollte der Speicherplatz für ein n-Tupel dynamisch angefordert sein. Schließlich sollten die n-Tupel im Interesse einer schnelleren Fehlerfindung logisch unterscheidbar sein. Damit erfolgt die Klassen-Darstellung von Vektoren auf einem Rechner in 2 Stufen: Zunächst braucht man eine Klasse für n-Tupel von Daten und zu ihnen gehörende Methoden, die für jeden Datentyp ausführbar sind. Falls mit den Daten solche Operationen ausführbar sind, wie sie für n-dimensionale Vektoren gelten, ergeben sich aus den n-Tupeln Vektoren. Zur Darstellung von Vektoren auf Rechnern wird hier die Programmiersprache C++ verwendet. Selbst wenn dem Leser die Konstrukte dieser objektorientierten Sprache fremd sein sollten, wird er diesen Text mit Gewinn studieren können, sofern Interesse und Verständnis für eine Programmiersprache vorhanden sind.
Die folgenden Darlegungen gelten in analoger Weise für weitere Klassen und ihre Objekte.
Die Aneinanderreihung von Daten gleichen Typs ist ein n-Tupel. Da der Typ der in einem n-Tupel abgelegten Daten zunächst unbekannt ist, wird eine Klassenvorlage `ls_array1<T>` für dynamische Felder des Typs T definiert.

```
#ifndef LS_ARRAY1
#define LS_ARRAY1
#include "ls_names.h"
#include LS_ERROR_H

template <class T>
class ls_array1 // Eindimensionale Felder
{ protected:
  T *a;
```

```
  char ONAME[ls_len];                  // Objektname
  ls_UINT dim;                         // Felddimension
  void set_data(T *aa=NULL,ls_UINT d=0){ a=aa, dim=d;}
  public:
  char* name;                          // Zeiger auf Objektnamen
  ls_array1(ls_UINT =0,char* ="ls_array1");// Standardkonstruktor
  ls_array1(ls_array1<T> &);           // move-Konstruktor
  ls_array1(T*,ls_UINT,char* ="ls_array1");// Feld-Übernahme
  ~ls_array1();
  ls_array1<T>& swap(ls_array1<T> &);  // Datenfelder-Austausch
  ls_UINT dimension() const{ return dim;}
  T* asArray()const{ return a;}
  T& operator[](ls_UINT) const;        // indizierter Zugriff
  const ls_array1<T>& operator=(const ls_array1<T> &);
  ls_array1<T>& put_array(const ls_array1<T>&,ls_UINT =0);
                                      // Feldeingabe ab Position
  ls_array1<T>& put_array(T, ls_UINT =0); // Elementeing. ab Pos.
  ls_array1<T>& put(T, ls_UINT);       // Elementeingabe auf Pos.
  T get(ls_UINT);                      // Elementausgabe von Pos.
  const ls_array1<T>& get_array(ls_array1<T> &,
                                ls_UINT =0) const;
                                       // Feldausgabe ab Pos.
  ls_array1<T>& append(ls_UINT =1);    // Nullelemente anhängen
  ls_array1<T>& remove(ls_UINT);       // Element entfernen
  ls_array1<T>& remove();              // Feld entfernen
  const ls_array1<T>& write(ostream&)const;// in Datei schreiben
  ls_array1<T>& read(istream &);       // aus Datei lesen
  const ls_array1<T>& operator>>(char*)const;// in namentl. Datei
  ls_array1<T>& operator<<(char*);     // aus namentl. Datei
};
#ifndef LS_LIB
#include LS_ARRAY1_C
#endif
#endif
```

Wie hat man sich die Funktionsweise dieser Klassenvorlage vorzustellen? Aus der Vorlage wird eine Klasse, indem der Parameter `T` einen Wert erhält; dies geschieht dadurch, daß in einem Programm ein Konstrukt der Form `ls_array1<int> kind` auftritt. Wir wählen als Datentyp `int`. Zunächst sei eine Instanz deklariert:

```
ls_array1<int> v(n);.
```

Diese Deklaration wird als Aufruf des Unterprogramms `ls_array1<int>` mit dem Parameter `n` und Referenzen, die auf die Daten des Objektes zeigen, übersetzt. Danach ist `v` logisch ein `n`-Tupel des Typs `int` und der Anfangsbelegung `0`. Jedes Objekt erhält einen Namen; standardmäßig wird der Klassenname als Objektname vergeben; er darf mittels `strcpy` geändert werden:

```
strcpy(v.name,"v-Feld").
```

Das `n`-Tupel ist dynamisch angelegt; die Komponente `a` (Zeiger) enthält die Referenz auf das angeforderte Feld; die Komponente `dim` enthält die aktuelle Dimension des Datenfeldes in Einheiten des Datentyps. Eine Instanz darf auch mit einem Datenfeld der Form `T *A` und seiner Feldlänge instanziiert werden. Die Dimension der dynamischen Komponente wird durch `v.dimension()` erfragt.
Die Tatsache, daß dynamische Komponenten verwendet werden, zwingt dazu, der Klasse eigene Kon-

struktoren, einen eigenen Zuweisungsoperator und einen eigenen Destruktor zu geben. In einem Konstruktor wird u. a. die dynamische Komponente angelegt und im Destruktor der Speicherplatz für die dynamische Komponente freigegeben. Üblicherweise ist auch ein sog. `copy`-Konstruktor vonnöten, der aufgerufen wird, falls man ein Objekt mittels eines anderen instanziiert.
Sehr wichtig ist es zu bemerken, daß hier in Klassen mit dynamischen Komponenten der `copy`-Konstruktor als `move`-Konstruktor arbeitet: Die Datenfelder werden in das neue Objekt übernommen; der Objektname wird kopiert; danach ist das Quellobjekt ohne Daten. Diese Form wird vorteilhaft beim Instanziieren des Rückkehrwertes verwendet: Angenommen, die in einem Unterprogramm erzeugte Instanz `A` mit dynamischen Komponenten soll Rückkehrwert sein. Bei der Anweisung `return A;` passiert folgendes: Durch den Aufruf des `copy`-Konstruktors wird auf dem Stack eine Kopie von `A` erzeugt; danach wird mittels Destruktor das Original zerstört und in das aufrufende Programm zurückgekehrt. Durch den realisierten `copy`-Konstruktor übernimmt die Stack-Kopie von `A` die dynamischen Komponenten ohne neuen Speicherplatz anzufordern und der für `A` aufgerufene Destruktor kann die dynamischen Komponenten nicht freigeben. Dadurch vermeidet man zeitweilige Doppelungen von Datenfeldern und Laufzeitfehler wegen fehlendem Speicherplatz. Soll der so implementierte `copy`-Konstruktor nicht aufgerufen werden, sollte man die Objekte per `call_by_reference` an ein Programm übergeben oder den Inhalt des Objektes vor dem Aufruf retten. Soll das Quellobjekt erhalten bleiben, so hat man anstelle von `ls_array1<int> A(B)` die Anweisungen `ls_array1<int> A; A=B;` zu codieren. Diese Anweisungen dürfen nicht zu `ls_array<int> A=B` verkürzt werden, da bei letzterer der `copy`-Konstruktor aufgerufen wird.
Eine weitere Möglichkeit, dynamische Datenfelder an andere Objekte zu übergeben, besteht darin, die Methode `swap` anzuwenden: Durch den Aufruf `A.swap(B)` werden alle Datenfelder aus der Instanz `B`, einschließlich Objektname, mit dem Objekt `A` getauscht. Durch Anwenden der `swap`-Methode innerhalb eines Unterprogramms auf zwei Objekte, wobei das eine innerhalb und das andere außerhalb des Unterprogramms instanziiert wurde, werden Daten an das aufrufende Programm übermittelt, ohne (zeitweilig) zusätzlich Speicherplatz anzufordern. Der Vorteil einer solchen Vorgehensweise ist offensichtlich: Wenn ein Objekt `A` mehrere Objekte anderer Klassen enthält, können diese zunächst außerhalb von `A` erzeugt werden, um sie dann beim Instanziieren von `A` ohne Anfordern von zusätzlichem Speicherplatz zu übernehmen. Hätte man diesen Mechanismus nicht, müßte man im Interesse von Speicherplatzeinsparung Methoden zur Konstruktion der einzelnen Unterobjekte in die Klassendefinition von `A` aufnehmen.
Datenaustausch gibt es mit anderen Instanzen und mit Dateien. Zunächst soll der Datenaustausch mit anderen Instanzen kommentiert werden. Besonders wichtig ist der elementweise Zugriff:

```
i = v[3]; v[3] = k; .
```

Zur Ein- und Ausgabe mehrerer Datenelemente dienen die Funktionen `get_array, put_array`. Im Aufruf beider sind das Datenfeld und die Position des ersten Datenelementes in der Quelle bzw. im Ziel anzugeben. Beim Aufruf `u.put_array(v,3)` werden Daten ab `v[0]` nach `u` ab `u[3]` kopiert; die Elementanzahl richtet sich nach dem Minimum aus der Dimension von `v` und der Dimension von `u` minus Anfangsposition. Der Aufruf `u.get_array(v,3)` schreibt die ab `u[3]` stehenden Daten in das Feld `v` ab `v[0]`; die geschriebene Elementanzahl berechnet sich in analoger Weise. Das Belegen aller Datenelemente von `u` mit dem Wert `s` wird durch `u.put_array(s)` oder `u.put_array(s,0)` erreicht.
Die Funktionen `write`, `read`, `<<` und `>>` dienen dem Datenaustausch mit Dateien. Will man z. B. die Instanz `v` in eine Datei namens `hallo` schreiben, so hat man einfach `v >> "hallo"` zu codieren; die Anweisung `v >> ""` führt zur Ausgabe auf das Standard-Ausgabemedium. Sollte man besondere Wünsche an die Voreinstellung der Datei haben, wie z. B. an die formatierte Ausgabe von Gleitpunktzahlen, sind die Funktionen `v.write(fout)` bzw. `v.read(fin)` zu verwenden; hierin bedeuten `fout` und `fin` Filedeskriptoren. In einer Datei haben Objektdaten eine standardisierte Darstellung. Sie beginnen

mit dem Objektnamen, gefolgt von einem in geschweiften Klammern eingeschlossenen Block; der Block beginnt mit einer Zeile, in der die Werte der Konstruktionsparameter mit vorangestelltem Schlüsselwort stehen, gefolgt von den Datenelementen. So hat das 4-Tupel

```
(1.1, 2.22, 3.333, 4.4444)
```

als Objekt mit dem Namen `huhu` die externe Darstellung

```
huhu { dimension: 4 1.1 2.22 3.333 4.4444 }.
```

Die einzelnen Daten sind durch übliche Trennzeichen getrennt. Sollte ein Objekt Instanzen enthalten, stehen im Block die Daten der entsprechenden Instanzen in analoger Form.
Ein Objekt, das Daten einlesen möchte, muß leer sein. Hat das Zielobjekt keinen Namen, wird das erste eingelesene Wort als Objektname verwendet; andernfalls wird zunächst nach dem Objektnamen gesucht. Durch diese Technik braucht das Programm vor dem Einlesen nicht den Objektnamen zu kennen:

```
ls_array1<int> k; strcpy(k.name,""); k<<"meine_datei";.
```

Vor Anwendung einer Klassenvorlage auf einen konkreten Datentyp müssen folgende Bedingungen erfüllt sein: Für den Datentyp muß ein Zuweisungsoperator definiert sein. Die `>>`- und `<<`-Operatoren für Dateien müssen definiert sein:

```
fin << v; fout >> v.
```

Für die standardmäßig vorhandenen Basistypen sind diese Bedingungen erfüllt.
Die Klassenvorlage `ls_array1<T>` vereinigt in sich Daten und Methoden, die unabhängig vom konkreten Datentyp sind. So kann die Klasse `ls_array1<unsigned short>` Basisklasse für eindimensionale Felder sein, deren Daten natürliche Zahlen sind. Eine konkrete Klasse mit diesen Datenelementen wird jedoch noch weitere Methoden beinhalten, wie z. B. die komponentenweise Addition von Feldern.
Wir werden erkennen, daß diese Klassenvorlage Basis für die Darstellung weiterer Objekte der linearen Algebra ist.
Basisklasse für einen Vektor ist die Klasse `ls_array1<ls_REAL>`, wobei `ls_REAL` für `float` bzw. `double` steht:

```
#ifndef LS_VECTOR
#define LS_VECTOR
#include "ls_names.h"
#include LS_ARRAY1_H

class ls_Matrix; class ls_sMatrix;
class ls_Vector: public ls_array1<ls_REAL>
{ public:
  ls_REAL eps;
  ls_Vector(ls_UINT =0, char* ="ls_Vector");
  ls_Vector(ls_Vector &);
  ls_Vector(ls_REAL*, ls_UINT, char* ="ls_Vector");
  ~ls_Vector(){}
  const ls_Vector& operator=(const ls_Vector &);
  ls_Vector operator-() const;                 //-x
  ls_Vector operator*(ls_REAL) const;          //x*s
  ls_Vector& operator*=(ls_REAL);              //x=x*s
  ls_Vector operator+(const ls_Vector &)const; //x+y
  ls_Vector operator-(const ls_Vector &)const; //x-y
  ls_Vector& operator+=(const ls_Vector &);    //x=x+y
```

```
  ls_Vector& operator-=(const ls_Vector &);    //x=x-y
  ls_REAL operator*(const ls_Vector &)const;   //x*y
  ls_Matrix dyad(const ls_Vector &)const;      //dyad.
  ls_sMatrix dyad()const;                      //dyad.
};
ls_Vector operator*(ls_REAL, const ls_Vector &); //s*x
#ifndef LS_LIB
#include LS_VECTOR_C
#endif
#endif
```

Die Klasse `ls_Vector` erbt zunächst alle Daten und Methoden ihrer Basisklasse. Zusätzliche Methoden – hier meist arithmetische – führen von der Basisklasse zur Vektor-Klasse. Sind nun `x`, `y` Instanzen gleicher Dimension der Klasse `ls_Vector` und `s` eine `ls_REAL`-Zahl, so sind auch `x + y`, `x - y`, `s*x`, `x*s`, `u += s*x`, `u -= y`, `u *= s` Instanzen der gleichen Klasse. Daraus folgt insbesondere, daß sich Linearkombinationen von Vektoren analog zu mathematischen Formeln darstellen lassen:

```
x = y + s*u + t*v.
```

Die `dyad`-Funktionen erzeugen als dyadisches Produkt eine Matrix.

2. Matrizen auf Rechnern

Um eine Matrix auf einem Rechner darstellen zu können, benötigt man zunächst ein zweidimensionales Datenfeld mit Daten beliebigen Typs. Auf dem Rechner gibt es aber nur die aufeinander folgende Anordnung von Datenelementen, wie sie durch die Klassenvorlage `ls_array1<T>` erfaßt ist. Folglich muß ihr eine neue Struktur aufgeprägt werden: Wir wollen von Zeilen und Spalten sprechen dürfen und brauchen Methoden, die dieser Struktur angepaßt sind. Daraus ergibt sich eine Klassenvorlage `ls_array2<T>`, die aus `ls_array1<T>` abgeleitet ist:

```
#ifndef LS_ARRAY2
#define LS_ARRAY2
#include "ls_names.h"
#include LS_ARRAY1_H

template <class T>
class ls_array2: public ls_array1<T>      // Zweidim. Felder
{ protected:
  ls_UINT m, n;
  void set_data(ls_UINT mm=0, ls_UINT nn=0){ m=mm; n=nn;}
  public:
  ls_array2(ls_UINT =0, ls_UINT =0, char* ="ls_array2");
  ls_array2(ls_array2<T> &);                // move-Konstruktor
  ls_array2<T>(T*, ls_UINT, ls_UINT, char* ="ls_array2");
                                            // Feld-Übernahme
  ~ls_array2(){}
  ls_array2<T>& swap(ls_array2<T> &);
  T* operator[](ls_UINT i) const;         // indizierter Zugriff
  ls_UINT number_of_rows() const{ return m;}
  ls_UINT number_of_columns() const{ return n;}
  const ls_array2<T>& operator=(const ls_array2<T> &);
  ls_array2<T>& put(T, ls_UINT, ls_UINT);   // Elementeingabe
  ls_array2<T>& put_row(const ls_array1<T>&, ls_UINT,
```

```
                        ls_UINT =0);
  ls_array2<T>& put_row(T, ls_UINT, ls_UINT =0);
  ls_array2<T>& put_column(const ls_array1<T>&, ls_UINT,
                           ls_UINT);
  ls_array2<T>& put_column(T, ls_UINT, ls_UINT);
  ls_array2<T>& put_diagonal(const ls_array1<T> &,
                             ls_UINT =0, ls_UINT =0);
  ls_array2<T>& put_diagonal(T, ls_UINT =0, ls_UINT =0);
  ls_array2<T>& put_array(const ls_array2<T> &,
                             ls_UINT =0, ls_UINT =0);
  ls_array2<T>& put_array(T, ls_UINT =0, ls_UINT =0);
  T get(ls_UINT, ls_UINT) const;          // Element-Ausgabe
  const ls_array2<T>& get_row(ls_array1<T>&,
                              ls_UINT, ls_UINT =0) const;
  const ls_array2<T>& get_column(ls_array1<T>&,
                              ls_UINT, ls_UINT) const;
  const ls_array2<T>& get_diagonal(ls_array1<T>&,
                              ls_UINT =0, ls_UINT =0) const;
  const ls_array2<T>& get_array(ls_array2<T>&,
                              ls_UINT =0, ls_UINT =0) const;
  ls_array2<T>& append_row(ls_UINT =1);   // Null-Zeilen anhäng.
  ls_array2<T>& append_column(ls_UINT =1);// Null-Spalten anhäng.
  ls_array2<T>& append_array(ls_UINT =1, ls_UINT =1);
                                          // Null-Feld anhängen
  ls_array2<T>& remove_row(ls_UINT i);    // Zeile streichen
  ls_array2<T>& remove_column(ls_UINT i); // Spalte streichen
  ls_array2<T>& remove();                 // Datenfeld streichen
  ls_array2<T>& swap_row(ls_UINT, ls_UINT);// Zeilentausch
  ls_array2<T>& swap_column(ls_UINT, ls_UINT);// Spaltentausch
  const ls_array2<T>& write_row(ostream &) const;
  ls_array2<T>& read_row(istream &);
  const ls_array2<T>& write_column(ostream &) const;
  ls_array2<T>& read_column(istream &);
  const ls_array2<T>& operator>>(char *)const;// Zeilen in Datei
  ls_array2<T>& operator<<(char *);          // Zeilen aus Datei
};
#ifndef LS_LIB
#include LS_ARRAY2_C
#endif
#endif
```

Wählen wir als Datentyp `int`, so lautet die Deklaration einer Instanz:

```
ls_array2<int> A(mm, nn).
```

Hierin geben `mm` die Zeilenanzahl und `nn` die Spaltenanzahl des Datenfeldes an; diese Daten werden auf `m` und `n` abgelegt, so daß sich als Gesamtlänge des angeforderten Feldes `dim = m*n` ergibt. Ein Vergleich mit der Klassenvorlage `ls_array1<T>` zeigt die Analogien. Der Direktzugriff erfolgt über Doppelindices: `A[i][j] = s; s = A[i][j];`. Die Zeilen- bzw. Spaltenanzahl wird mit `A.number_of_rows()` bzw. `A.number_of_columns()` abgefragt. Methoden der Basisklasse werden entweder übernommen oder sinnvoll durch andere überlagert. Da dieses Datenfeld in Zeilen und Spalten strukturiert ist, gibt es auch Diagonalen. Dem wird dadurch entsprochen, daß es Methoden gibt, die mit Zeilen, Spalten oder Diagonalen arbeiten:

- Eingabe von Werten und Feldern als Zeile, Spalte oder Diagonale:

`put_row`, `put_column`, `put_diagonal`,

- Anhängen von Null-Zeilen bzw. Null-Spalten:
  `append_row`, `append_column`,
- Eingabe eines zweidimensionalen Feldes als Unterfeld:
  `put_array`,
- Ausgabe von Feldern, die als Zeilen, Spalten, Diagonalen, Unterfeldern in der Instanz vorkommen:
  `get_row`, `get_column`, `get_diagonal`, `get_array`,
- Vertauschen von Zeilen oder Spalten:
  `swap_row`, `swap_column`,
- Streichen von Zeilen oder Spalten:
  `remove_row`, `remove_column`.

Der externe Datenaustausch (Instanz mit Datei) erfolgt nun zeilen- oder spaltenweise, jedoch in der gleichen äußeren Form wie in der Basisklasse.
Beispiel: Das zweidimensionale Datenfeld mit 4 Zeilen und 5 Spalten

$$\begin{pmatrix} 1 & 1 & 2 & 2 & 0 \\ 1 & 2 & 3 & 2 & 1 \\ 2 & 1 & 0 & 3 & 1 \\ 0 & 0 & 1 & 1 & 3 \end{pmatrix}$$

und dem Namen `hello` hat die externe Darstellung

```
hello
{ number_of_rows: 4 number_of_columns: 5
  1 1 2 2 0
  1 2 3 2 1
  2 1 0 3 1
  0 0 1 1 3
}.
```

Bis hier enthält diese Klassenvorlage weitgehend datentyp-unabhängige Methoden. Aus ihr wird eine **Matrix** (Rechteckmatrix), wenn die Datenelemente aus einem algebraischen Körper genommen werden. Diese Datentyp-Spezifikation erlaubt es, arithmetische Operationen auszuführen und man erhält die abgeleitete Klasse `ls_Matrix`.

```
#ifndef LS_MATRIX
#define LS_MATRIX
#include "ls_names.h"
#include LS_VECTOR_H
#include LS_ARRAY2_H

class ls_Matrix: public ls_array2<ls_REAL>
{ public:
  ls_Matrix(ls_UINT =0, ls_UINT =0, char* ="ls_Matrix");
  ls_Matrix(ls_Matrix &);
```

```
  const ls_Matrix& operator=(const ls_Matrix &);
  ls_Vector row(ls_UINT i)const
  { ls_Vector u(n); get_row(u,i); return u;}
  ls_Vector column(ls_UINT j)const
  { ls_Vector u(n); get_column(u,0,j); return u;}
  ls_Vector upper_diagonal(ls_UINT j)const
  { ls_Vector u(n); get_diagonal(u,0,j); return u;}
  ls_Vector lower_diagonal(ls_UINT i)const
  { ls_Vector u(n); get_diagonal(u,i,0); return u;}
  ls_Matrix& operator+=(const ls_Matrix &);    //A=A+B
  ls_Matrix operator-(const ls_Matrix &)const; //A-B
  ls_Matrix& operator-=(const ls_Matrix &);    //A=A-B
  ls_Matrix operator+(const ls_Matrix &) const;//A+B
  ls_Matrix operator*(const ls_Matrix &) const;//A*B
  ls_Vector operator*(const ls_Vector &) const;//A*x
  ls_Matrix operator*(ls_REAL) const;          //A*s
  ls_Matrix& operator*=(ls_REAL);              //A=A*s
  ls_UINT solve(ls_Vector &x, ls_Vector &b) const;//cg-Verfahren

};
ls_Matrix operator*(ls_REAL, const ls_Matrix &);  // s*A
ls_Vector operator*(const ls_Vector &, const ls_Matrix &);// x*A
#ifndef LS_LIB
#include LS_MATRIX_C
#endif
#endif
```

Man sieht, daß die Klasse `ls_array2<ls_REAL>` lediglich um arithmetische Operationen mit Vektoren, Matrizen und Zahlen ergänzt ist. Sind nun `A`, `B`, `C` Rechteckmatrizen (Instanzen der Klasse `ls_Matrix`), `x`, `y`, `z` Vektoren passender Dimension und `s, t` Zahlen, so wird durch `A*x` ein Vektor erzeugt (Matrix mal Vektor); ebenso durch `y*A` (transponierte Matrix mal Vektor); bei `C = A*B` wird der Matrix `C` ein Matrizenprodukt Matrix mal Matrix zugewiesen. Auch zusammengesetzte Operationen lassen sich problemlos notieren, wie z. B. `z = s*(A*x) - t*(y*B)`.

3. Symmetrische Matrizen auf Rechnern

Symmetrische Matrizen zeichnen sich gegenüber quadratischen Matrizen dadurch aus, daß die Daten an der Hauptdiagonalen gespiegelt sind; daher braucht auch nur das obere Dreieck der Matrix abgespeichert zu werden. Dem obigen Vorgehen folgend ist also zunächst aus der Klassenvorlage `ls_array1<T>` eine Klassenvorlage für ein zweidimensionales, oberes Dreiecksfeld abzuleiten.

```
// oberes Dreiecksfeld

#ifndef LS_ARRAYU
#define LS_ARRAYU
#include "ls_names.h"
#include LS_ARRAY2_H

template <class T>
class ls_arrayU: public ls_array1<T>
{ protected:
  ls_UINT n;
```

```
  void set_data(ls_UINT nn=0){ n=nn;}
  public:
  ls_arrayU(ls_UINT =0, char* ="ls_arrayU");
  ls_arrayU(ls_arrayU<T>&);
  ~ls_arrayU(){}
  ls_arrayU<T>& swap(ls_arrayU<T>&);
  T* operator[](ls_UINT k) const;
  ls_UINT dimension() const{ return n;}
  const ls_arrayU<T>& operator=(const ls_arrayU<T> &);
  ls_arrayU<T>& put(T, ls_UINT, ls_UINT);
  ls_arrayU<T>& put_row(const ls_array1<T>&, ls_UINT =0,
                        ls_UINT =0);
  ls_arrayU<T>& put_row(T, ls_UINT, ls_UINT);
  ls_arrayU<T>& put_column(const ls_array1<T>&,
                           ls_UINT =0, ls_UINT =0);
  ls_arrayU<T>& put_column(T, ls_UINT, ls_UINT);
  ls_arrayU<T>& put_diagonal(const ls_array1<T>&,
                             ls_UINT =0, ls_UINT =0);
  ls_arrayU<T>& put_diagonal(T, ls_UINT =0, ls_UINT =0);
  T get(ls_UINT, ls_UINT) const;
  const ls_arrayU<T>& get_row(ls_array1<T>&,
                              ls_UINT =0, ls_UINT =0) const;
  const ls_arrayU<T>& get_column(ls_array1<T>&,
                              ls_UINT =0, ls_UINT =0) const;
  const ls_arrayU<T>& get_diagonal(ls_array1<T>&,
                              ls_UINT =0, ls_UINT =0) const;
  const ls_arrayU<T>& get_array(ls_array2<T>&) const;
  ls_arrayU<T>& append_column(ls_UINT =1);
  ls_arrayU<T>& swap(ls_UINT, ls_UINT);
  ls_arrayU<T>& remove(ls_UINT);
  ls_arrayU<T>& remove();
  const ls_arrayU<T>& write_row(ostream &) const;
  ls_arrayU<T>& read_row(istream &);
  const ls_arrayU<T>& write_column(ostream &) const;
  ls_arrayU<T>& read_column(istream &);
  const ls_arrayU<T>& operator>> (char *) const;
  ls_arrayU<T>& operator<< (char *);
};
#ifndef LS_LIB
#include LS_ARRAYU_C
#endif
#endif
```

Man erkennt die große Ähnlichkeit mit der Vorlage `ls_array2<T>`. Es gibt aber wichtige Änderungen: Die Indizierung der Datenelemente erfolgt so, als ob die Daten aus dem unteren Dreieck vorhanden wären: Bei jeder Positionsangabe mittels Zeilenindex `i` und Spaltenindex `j` muß stets `i <= j` gelten. Dem Nutzer ist es überlassen, ob es sich dabei um ein symmetrisches oder unsymmetrisches oberes Dreiecksfeld handelt. Aus dieser Klasse wird eine symmetrische Matrix abgeleitet.

```
// symmetrische Matrix (nur oberes Dreieck)

#ifndef LS_SMATRIX
#define LS_SMATRIX
#include "ls_names.h"
```

```
#include LS_VECTOR_H
#include LS_ARRAYU_H

class ls_Matrix;

class ls_sMatrix: public ls_arrayU<ls_REAL>
{ public:
  ls_sMatrix(ls_UINT =0, char* ="ls_sMatrix");
  ls_sMatrix(ls_sMatrix &);
  ls_Vector row(ls_UINT i)const
  { ls_Vector u(n); get_row(u,i,i+1), get_column(u,0,i);
    return u;}
  ls_Vector diagonal(ls_UINT j)const
  { ls_Vector u(n); get_diagonal(u,0,j); return u;}
  const ls_sMatrix& operator=(const ls_sMatrix &); //A=B
  ls_sMatrix& operator+=(const ls_sMatrix&);       //A=A+B
  ls_sMatrix operator-(const ls_sMatrix&) const;   //A-B
  ls_sMatrix& operator-=(const ls_sMatrix&);       //A=A-B
  ls_sMatrix operator+(const ls_sMatrix&) const;   //A+B
  ls_sMatrix& operator*=(ls_REAL);                 //A=A*s
  ls_Vector operator*(const ls_Vector&) const;     //A*x
  ls_sMatrix operator*(ls_REAL) const;             //A*s
  ls_Matrix operator*(const ls_sMatrix &) const;   //A*B
  ls_UINT solve(ls_Vector &x, ls_Vector &b);     //cg-Verfahren
};
ls_sMatrix operator*(ls_REAL, const ls_sMatrix&);  //s*A
#include LS_MATRIX_H
#ifndef LS_LIB
#include LS_SMATRIX_C
#endif
#endif
```

Hier wird das obere Dreiecksfeld symmetrisch interpretiert. Dies wirkt sich insbesondere auf das Lösen eines entsprechenden linearen Gleichungssystems aus. Desweiteren wird aus der Klassenvorlage `ls_arrayU<T>` eine obere Dreiecksmatrix abgeleitet.

```
// obere Dreiecksmatrix

#ifndef LS_UMATRIX
#define LS_UMATRIX
#include "ls_names.h"
#include LS_VECTOR_H
#include LS_ARRAYU_H

class ls_Matrix;

class ls_uMatrix: public ls_arrayU<ls_REAL>
{ public:
  ls_uMatrix(ls_UINT =0, char* ="ls_uMatrix");
  ls_uMatrix(ls_uMatrix &);
  ls_Vector row(ls_UINT i)const
  { ls_Vector u(n); get_row(u,i,i); return u;}
  ls_Vector column(ls_UINT j)const
  { ls_Vector u(n); get_column(u,0,j); return u;}
```

```
  ls_Vector diagonal(ls_UINT j)const
  { ls_Vector u(n); get_diagonal(u,0,j); return u;}
  const ls_uMatrix& operator=(const ls_uMatrix &); //A=B
  ls_uMatrix& operator+=(const ls_uMatrix&);       //A=A+B
  ls_uMatrix operator-(const ls_uMatrix&) const;   //A-B
  ls_uMatrix& operator-=(const ls_uMatrix&);       //A=A-B
  ls_uMatrix operator+(const ls_uMatrix&) const;   //A+B
  ls_uMatrix& operator*=(ls_REAL);                 //A=A*s
  ls_Vector operator*(const ls_Vector&) const;     //A*x
  ls_uMatrix operator*(ls_REAL) const;             //A*s
  ls_Matrix operator*(const ls_uMatrix &) const;   //A*B
  ls_UINT solve(ls_Vector &, ls_Vector &);       //cg-Verfahren
  void backward(ls_Vector &, ls_Vector &);     // Rückwärtseins.
};
ls_uMatrix operator*(ls_REAL, const ls_uMatrix&);  //s*A
ls_Vector operator*(const ls_Vector&, const ls_uMatrix&); //x*A
#include LS_MATRIX_H
#ifndef LS_LIB
#include LS_UMATRIX_C
#endif
#endif
```

Symmetrische Bandmatrizen mit Diagonalspeicherung sind für Diskretisierungsmethoden wichtig; daher ist ihnen eine besondere Klasse gewidmet.
Zunächst sei eine Klassenvorlage namens `ls_array1M<T>` definiert, in der eine Sammlung von eindimensionalen Datenfeldern verwaltet wird:

```
#ifndef LS_ARRAY1M
#define LS_ARRAY1M
#include "ls_names.h"
#include LS_ARRAY1_H

template <class T>
struct array1M{ ls_UINT l; T *f;};

template <class T>
class ls_array1M          // Reihung Eindimensionaler Felder
{ protected:
  array1M<T> *a;          // Datenfeld
  ls_UINT   n;            // Dimension von a
  char ONAME[ls_len];
  void set_data(array1M<T> *aa=NULL,ls_UINT nn=0){ a=aa; n=nn;}
  public:
  char *name;             // Objektname
  ls_array1M(ls_UINT =0, char* ="ls_array1M");
  ls_array1M(ls_array1M<T> &);
  ~ls_array1M();
  ls_array1M<T>& swap(ls_array1M<T> &);
  ls_UINT number_of_arrays() const{ return n;}
  ls_UINT length(ls_UINT i) const{ return (a+i)->l;}
                          // Länge des i-ten Feldes
  array1M<T>* asArray()const{ return a;}
  T* operator[](ls_UINT i){ return (a+i)->f;}
                        // indizierter Zugriff auf das i-te Feld
```

```
  const ls_array1M<T>& operator=(const ls_array1M<T>&);
  ls_array1M<T>& put_array(const ls_array1<T>&v, ls_UINT i);
                          // Eingabe des i-ten Feldes
  ls_array1M<T>& put(const T*, ls_UINT l, ls_UINT i);
                          // Eingabe eines Feldes als i-tes Feld
  ls_array1M<T>& put(T, ls_UINT);
                          // Eingabe eines Wertes als i-tes Feld
  const ls_array1M<T>& get_array(ls_array1<T>&v,
                                 ls_UINT i) const;
                          // Ausgabe des i-ten Feldes
  ls_array1M<T>& remove_array(ls_UINT i);
                                  // i-tes Feld entfernen
  ls_array1M<T>& remove();        // alle Felder entfernen
  const ls_array1M<T>& write(ostream &) const;
  ls_array1M<T>& read(istream &);
  const ls_array1M<T>& operator>> (char *) const;
  ls_array1M<T>& operator<< (char *);
};
#ifndef LS_LIB
#include LS_ARRAY1M_C
#endif
#endif
```

Ein Objekt dieser Klasse enthält n eindimensionale Datenfelder; jedes Einzelfeld hat eine eigene Dimension.
Aus dieser Klasse wird eine Klasse `ls_bMatrix` abgeleitet, die symmetrische Bandmatrizen repräsentiert, wobei in den eindimensionalen Feldern die oberen Diagonalen abgespeichert sind. Beim Initialisieren ist die Anzahl der Diagonalen anzugeben. Als Operation ist hier nur die Operation Matrix mal Vektor hinzugefügt.

```
#ifndef LS_BMATRIX
#define LS_BMATRIX
#include "ls_names.h"
#include LS_VECTOR_H
#include LS_ARRAY1M_H

class ls_bMatrix: public ls_array1M<ls_REAL>
{ public:
  ls_bMatrix(ls_UINT =0,char* ="ls_bMatrix");
  ls_bMatrix(ls_bMatrix &);
  operator ls_array1M<ls_REAL>&(){ return *this;}
  const ls_bMatrix& operator=(const ls_bMatrix &);//A=B
  ls_UINT number_of_diagonals() const{ return n;}
  ls_Vector operator* (ls_Vector &);              //A*x
  ls_UINT solve(ls_Vector &x, ls_Vector &b);      //cg-Verfahren
};
#ifndef LS_LIB
#include LS_BMATRIX_C
#endif
#endif
```

Das Speicherabbild einer symmetrischen Bandmatrix mit Diagonalspeicherung hat eine Besonderheit: Falls in einer Diagonalen nur ein Element abgelegt ist, werden alle Elemente der Diagonalen als gleich diesem angesehen. Extremal kann so jede Diagonale durch jeweils ein Element repräsentiert sein. Damit

belegen solche Matrizen minimalen Speicherplatz. In der Operation Matrix mal Vektor ist diese Möglichkeit entsprechend berücksichtigt.

4. Gleichungslöser

Die Spezifik einer Matrix findet in den Methoden zur Lösung eines linearen Gleichungssystems ihre Fortsetzung. Zunächst gehört zu jeder Matrix-Klasse als Methode das konjugierte Gradientenverfahren (`solve`). Beim Aufruf dieser Methode ist als 1. Parameter der Startvektor und als 2. Parameter die rechte Seite anzugeben; auf dem Startvektor findet man als Ausgabe die gefundene Lösung, auf der rechten Seite das Residuum. Im Falle einer allgemeinen Koeffizientenmatrix wird gegebenenfalls die Quadratmittellösung bestimmt.
Jede andere Methode verändert die Koeffizientenmatrix; daher entpricht ihr eine Klasse; alle Löser-Klassen haben einen einheitlichen Aufbau; sie enthalten insbesondere die Koeffizientenmatrix in unveränderter Form und die zugehörige Faktorisierung, gegebenenfalls mit Hilfsfeldern, damit eine Lösungsberechnung möglich ist. Betrachten wir z. B. die `QR`-Faktorisierung nach Householder:

```
#ifndef LS_QR
#define LS_QR
#include "ls_names.h"
#include LS_MATRIX_H

class ls_QR
{ protected:
  ls_Matrix A, F;        // Matrix und QR-Faktorisierung
  ls_array1<ls_REAL> gamma, rho;  // Hilfsfelder
  char ONAME[ls_len];
  public:
  char *name, *A_name, *F_name, *gamma_name, *rho_name;
  ls_REAL eps;
  int rc;                // Rückkehrwert nach Faktorisierung
  ls_QR(char* ="ls_QR");
  ls_QR(ls_Matrix &, ls_REAL =0., char* ="ls_QR");
                         // Matrix-Übernahme (move-Konstruktor)
                         // Faktorisierung (mit Regularisierung)
  ls_QR(ls_QR &);
  ls_QR& swap(ls_QR &);
  unsigned char good()const{ return !rc;} // Erfolgssignal
  const ls_QR& operator=(const ls_QR &);
  const ls_QR& solve(ls_Vector &x, const ls_Vector &b) const;
                         // Gleichungslöser
  ls_UINT post_iteration(ls_Vector &x, ls_Vector &b) const;
                         // Nachiteration
  ls_Vector residuum(const ls_Vector &x,
                     const ls_Vector &b) const;
  ls_Vector Qx(const ls_Vector &x) const; // Q*x
  ls_Vector xQ(const ls_Vector &x) const; // x*Q
  ls_Vector Rx(const ls_Vector &x) const; // R*x
  ls_Vector xR(const ls_Vector &x) const; // x*R
  const ls_QR& write_row(ostream &) const;
  ls_QR& read_row(istream &);
  const ls_QR& write_column(ostream &) const;
  ls_QR& read_column(istream &);
```

```
  const ls_QR& operator>> (char *) const;
  ls_QR& operator<< (char *);
};
#ifndef LS_LIB
#include LS_QR_C
#endif
#endif
```

Um einen Zugriff auf die Objektnamen der eingebetteten Objekte zu ermöglichen, gibt es die entsprechenden Zeiger: `A_name` enthält einen Zeiger auf den Objektnamen des eingebetteten Objektes `A` usw. . Bei der Anweisung `ls_QR C(B);` mit einer Matrix `B` wird der `move`-Konstruktor aufgerufen und die Faktorisierung ausgeführt. Nach erfolgreicher Deklaration, was mittels der `good`-Funktion überprüft werden kann, ruft man die Funktion `solve` mit zwei Vektoren auf, wobei auf dem ersten die gefundene Lösung abgelegt wird und auf dem zweiten die rechte Seite für das entsprechende Gleichungssystem zu übergeben ist. Im Falle einer quadratischen Koeffizientenmatrix wird die Lösung des Gleichungssystems, bei einem überbestimmten Gleichungssystem die Quadratmittel-Lösung bestimmt.
Bei der Anweisung `ls_QR C(B,s);` mit einer kleinen, positiven, reellen Zahl `s` wird eine Regularisierung angewendet. Diese Vorgehensweise empfiehlt sich sehr bei Gleichungssystemen mit schlechtkonditionierter Koeffizientenmatrix.
Gegebenenfalls darf eine Nachiteration durchgeführt werden:

`post_iteration(ls_Vector &x, ls_Vector &b)`.

Dabei ist auf `x` die aktuelle Lösung und auf `b` die rechte Seite zu übergeben. Als Ergebnis erhält man die nachiterierte Lösung und das Residuum; der Rückkehrwert liefert die Anzahl der ausgeführten Iterationen. Bei der Nachiteration ist zu berücksichtigen, daß versucht wird, die gefundene Rechnerlösung der im Rechner befindlichen Aufgabe anzupassen. Hat die Koeffizientenmatrix eine große Kondition, wird eine Nachiteration wenig erfolgreich sein. Meist wird eine schlechtkonditionierte Aufgabe bereits bei der Zerlegungsberechnung dadurch erkannt, daß die Berechnung abbricht, da die Matrix numerisch singulär ist. Es ist dringend empfohlen, nach der Zerlegungsberechnung den Rückkehrwert zu testen.
Für weitere Anwendungen mit quadratischen Faktorisierungen benötigt man oft bei gegebener `QR`-Faktorisierung die Operationen `Q*x`, `x*Q`, `R*x` und `x*R`; daher sind die entsprechenden Funktionen hinzugefügt worden.
Über die Lese-Schreib-Funktionen kann man die Faktorisierung (und die Matrix) retten, um zu einem späteren Zeitpunkt weitere Lösungen zu berechnen. Alle anderen Klassen für Gleichungslöser arbeiten nach dem gleichen Muster und enthalten analoge Funktionen.
Die `LU`-Faktorisierung für eine quadratische Matrix gibt es in 5 Varianten:

- ohne Pivotisierung (`ls_LU`),
- Spalten-Pivotisierung (`ls_LU_column`),
- Zeilen-Pivotisierung (`ls_LU_row`),
- Diagonal-Pivotisierung (`ls_LU_diagonal`).
- Total-Pivotisierung (`ls_LU_total`).

Die verschiedenen Pivotisierungen verwenden dabei eine fiktive Skalierung.
Beispielhaft sei die Klasse `ls_LU_column` notiert.

```
#ifndef LS_LU_COLUMN
#define LS_LU_COLUMN
```

```
#include "ls_names.h"
#include LS_MATRIX_H

class ls_LU_column
{ protected:
  ls_Matrix A, F;
  ls_array1<ls_UINT> ind;
  char ONAME[ls_len];
  public:
  char *name, *A_name, *F_name, *ind_name;
  ls_REAL eps;           // Genauigkeitsschranke
  int rc;                // Rückkehrwert nach Faktorisierung
  ls_LU_column(char* ="ls_LU_column");
  ls_LU_column(ls_Matrix &, ls_REAL =0., char* ="ls_LU_column");
                         // Matrix-Übernahme (move-Konstruktor)
                         // LU-Faktor. mit Spalten-Pivotisierung
  unsigned char good()const{ return !rc;}// Erfolgssignal
  ls_LU_column(ls_LU_column &);    // move-Konstruktor
  ls_LU_column& swap(ls_LU_column &);
  const ls_LU_column& operator=(const ls_LU_column &);
  const ls_LU_column& solve(ls_Vector &x,
                            const ls_Vector &b) const;
                         // Gleichungslöser
  ls_UINT post_iteration(ls_Vector &, ls_Vector &) const;
                         // Nachiteration
  ls_Vector residuum(const ls_Vector &x,
                     const ls_Vector &b) const;
  ls_Vector Lx(const ls_Vector &x) const; // L*x
  ls_Vector xL(const ls_Vector &x) const; // x*L
  ls_Vector Ux(const ls_Vector &x) const; // U*x
  ls_Vector xU(const ls_Vector &x) const; // x*U
  const ls_LU_column& write_row(ostream &) const;
  ls_LU_column& read_row(istream &);
  const ls_LU_column& write_column(ostream &) const;
  ls_LU_column& read_column(istream &);
  const ls_LU_column& operator>> (char *) const;
  ls_LU_column& operator<< (char *);
};
#ifndef LS_LIB
#include LS_LUCOLUMN_C
#endif
#endif
```

Man erkennt die Analogie zur Klasse `ls_QR`.
Daneben gibt es mit gleichen Pivotisierungsvarianten die Invertierung einer Matrix (`ls_INV`).
Ein kleines Beispielprogramm möge die Anwendung illustrieren.

```
//: $CC +eh -o beispiel1 beispiel1.cpp -lm
//#define ls_REAL float
#include "ls_QR.h"
int main()
{ ls_UINT n=8; ls_REAL s=1.e-10;
;
try
```

```
  {
    ls_Vector x(n,"Lösung"), b(n,"rechte_Seite");
    ls_Matrix A(n,n,"Koeffizienten-Matrix");
    for(ls_UINT j, i=0; i < n; i++)
      for(j=i; j < n; j++) A[i][j] = A[j][i]=1./(i+j+1.);
                        // Hilbert-Matrix
    x.put_array(1.);    // alle Komponenten von x gleich 1
    b = A*x;            // rechte Seite ist die Zeilensumme
    b >> "";            // Ausgabe auf Standard-Ausgabe-Medium
    x.put_array(0.);    // alle Komponenten von x sind gleich 0
    ls_QR AA(A,s);      // QR-Faktorisierung mit Regularisierung
    if(AA.good())       // normal weiter, falls erfolgreich
    { AA.solve(x,b);    // Gleichungssystem lösen; alle gleich 1.
      x >> "";          // Lösung anschauen
      AA.residuum(x,b) >> "";// Residuum anschauen
    }
    else cerr << "Methode sagt: Matrix ist singulär." << endl;
  }
  catch(...){};         // erkannte Fehler auffangen.
  cin >> n;
  return 0;
}
```

Es sei erwähnt, daß selbst beim Lösen eines Systems mit einer `(500,500)`-Hilbertmatrix mittels Regularisierung eine Lösung berechnet wird, die `5` erste richtige Ziffern hat.
Zum Gleichungslösen von Systemen mit symmetrischer Koeffizienten-Matrix dienen die Klassen

- `ls_LDLT`: `LDLT`-Faktorisierung ohne Pivotisierung,
- `ls_LDLT_diagonal`: `LDLT`-Faktorisierung mit Pivotisierung entlang der Hauptdiagonalen.

```
#ifndef LS_LDLT_DIAGONAL
#define LS_LDLT_DIAGONAL
#include "ls_names.h"
#include LS_SMATRIX_H

class ls_LDLT_diagonal
{ protected:
  ls_sMatrix A, F;                // Matrix und Faktorisierung
  ls_array1<ls_UINT> ind;         // wahre Diagonal-Indices
  char ONAME[ls_len];
  public:
  char *name, *A_name, *F_name, *ind_name;
  ls_REAL eps;
  int rc;                  // Rückkehrwert nach Faktorisierung
  ls_LDLT_diagonal(char* ="ls_LDLT_diagonal");
  ls_LDLT_diagonal(ls_sMatrix &, ls_REAL =0.,
                   char* ="ls_LDLT_diagonal");
                         // Matrix-Übernahme (move-Konstruktor)
                         // LDLT-Faktor. mit Diagonal-Pivot.
  unsigned char good()const{ return !rc;} // Erfolgssignal
  ls_LDLT_diagonal(ls_LDLT_diagonal &);
  ls_LDLT_diagonal& swap(ls_LDLT_diagonal &);
  const ls_LDLT_diagonal& operator=(const ls_LDLT_diagonal&);
```

```
  const ls_LDLT_diagonal& solve(ls_Vector &x,
                                const ls_Vector &b) const;
                                  // Gleichungslöser
  ls_UINT post_iteration(ls_Vector &, ls_Vector &) const;
                                  // Nachiteration
  ls_Vector residuum(const ls_Vector &x,
                     const ls_Vector &b) const;
  ls_Vector Lx(const ls_Vector &x) const;
                         // L*x (L: Cholesky-Faktor)
  ls_Vector xL(const ls_Vector &x) const;
                         // x*L (L: Cholesky-Faktor)
  const ls_LDLT_diagonal& write_row(ostream &) const;
  ls_LDLT_diagonal& read_row(istream &);
  const ls_LDLT_diagonal& write_column(ostream &) const;
  ls_LDLT_diagonal& read_column(istream &);
  const ls_LDLT_diagonal& operator>> (char *) const;
  ls_LDLT_diagonal& operator<< (char *);
};
#ifndef LS_LIB
#include LS_LDLTDIAGONAL_C
#endif
#endif
```

Das obige Beispiel läßt sich sehr leicht zu einem Beispiel für die LDLT-Faktorisierung machen.

```
//: $CC -o beispiel2 beispiel2.cpp -lm
//#define ls_REAL float
#include "ls_LDLT_diagonal.h"
int main()
{ ls_UINT n=6, i, j; ls_REAL s=1.e-10;
try
  {
    ls_Vector x(n,"Lösung"), b(n,"rechte_Seite");
    ls_sMatrix A(n,"Koeffizienten-Matrix");
    for(i=0; i < n; i++)
      for(j=i; j < n; j++) A[i][j] = 1./(i+j+1.); // Hilbert-Matrix
    x.put_array(1.);     // alle Komponenten von x gleich 1
    b = A*x;             // rechte Seite ist die Zeilensumme
    b >> "";             // Ausgabe auf Standard-Ausgabe-Medium
    x.put_array(0.);     // alle Komponenten von x sind gleich 0
    ls_LDLT_diagonal AA(A,s);// LDLT-Faktorisierung mit Diagonalpiv.
    if(AA.good())        // normal weiter, falls erfolgreich
    { AA.solve(x,b);     // System lösen; alle gleich 1.
      x >> "";           // Lösung ansehen
      AA.residuum(x,b) >> "";// Residuum ansehen
    }
    else cerr << "Methode meint: Matrix ist singulär." << endl;
  }
  catch(...){};         // erkannte Fehler auffangen.
  cin >> n;
  return 0;
}
```

Die folgende Tabelle belegt, in welcher Weise bei den obigen Gleichungslösern die Regularisierung wirkt, sofern man mit reellen Zahlen vom Typ double rechnet. Dazu wurde für wachsende Ordnung n

der Koeffizienten-Hilbertmatrix (n=6, 10, 50, 100, 500) das entsprechende Gleichungssystem mit den Regularisierungsparametern

s=0., 1.e-7, 1.e-8, 1.e-9, 1.e-10, 1.e-11, 1.e-12

in der Methoden-Reihenfolge

LU (1), LU_column (2), LU_row (3), LU_diagonal (4), LU_total (5), INV (6), INV_column (7), INV_row (8), INV_diagonal (9), INV_total (10), QR (11),LDLT (12), LDLT_diagonal (13)

gelöst. Gemessen und ausgegeben wurde der relative Fehler.

	0.00e+00	1.00e-7	1.00e-8	1.00e-09	1.00e-10	1.00e-11	1.00e-12	1.00e-13
n=6								
(1)	7.45e-11	1.87e-4	3.25e-5	3.52e-06	3.54e-07	3.56e-08	3.55e-09	1.97e-10
(2)	4.64e-10	1.87e-4	3.25e-5	3.52e-06	3.54e-07	3.57e-08	3.16e-09	4.19e-10
(3)	2.42e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.56e-08	2.98e-09	5.07e-10
(4)	3.91e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.53e-08	3.51e-09	9.18e-11
(5)	1.90e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.51e-08	3.19e-09	4.63e-10
(6)	8.07e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.60e-08	2.42e-09	3.93e-10
(7)	4.25e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.47e-08	9.51e-10	1.53e-09
(8)	3.80e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.52e-08	2.75e-09	4.36e-10
(9)	1.10e-09	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.42e-08	2.02e-09	3.81e-10
(10)	1.10e-09	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.42e-08	2.02e-09	3.81e-10
(11)	4.11e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.53e-08	3.19e-09	1.54e-10
(12)	2.63e-10	1.87e-4	3.25e-5	3.52e-06	3.54e-07	3.53e-08	3.46e-09	1.18e-10
(13)	2.60e-10	1.87e-4	3.25e-5	3.52e-06	3.55e-07	3.53e-08	3.45e-09	1.58e-10
n=10								
(1)	6.30e-04	1.82e-4	5.13e-5	1.74e-05	4.68e-06	4.06e-06	5.73e-05	1.76e-04
(2)	4.12e-04	1.82e-4	5.13e-5	1.74e-05	5.50e-06	4.51e-06	4.28e-05	1.51e-04
(3)	4.41e-04	1.82e-4	5.13e-5	1.74e-05	5.16e-06	3.32e-06	4.97e-05	2.44e-05
(4)	8.01e-06	1.82e-4	5.13e-5	1.74e-05	5.15e-06	5.02e-06	6.11e-05	1.70e-04
(5)	2.00e-04	1.82e-4	5.13e-5	1.74e-05	4.74e-06	4.92e-06	2.17e-05	3.83e-04
(6)	6.10e-03	1.82e-4	5.13e-5	1.77e-05	4.76e-06	4.48e-05	7.56e-05	3.80e-03
(7)	9.78e-03	1.82e-4	5.14e-5	1.74e-05	6.02e-06	2.66e-05	8.62e-04	2.71e-03
(8)	4.44e-04	1.82e-4	5.13e-5	1.74e-05	5.22e-06	1.71e-05	6.69e-05	4.78e-04
(9)	2.22e-04	1.82e-4	5.13e-5	1.74e-05	5.43e-06	2.78e-06	6.77e-05	2.35e-04
(10)	2.22e-04	1.82e-4	5.13e-5	1.74e-05	5.43e-06	2.78e-06	6.77e-05	2.35e-04
(11)	3.38e-04	1.82e-4	5.13e-5	1.74e-05	5.22e-06	2.79e-06	2.98e-05	1.45e-04
(12)	2.08e-04	1.82e-4	5.13e-5	1.74e-05	5.18e-06	4.16e-06	2.65e-05	2.38e-04
(13)	1.22e-04	1.82e-4	5.13e-5	1.74e-05	5.15e-06	4.96e-06	2.64e-05	2.18e-04
n=50								
(1)	5.24e+02	1.79e-4	5.67e-5	1.80e-05	6.63e-06	2.76e-05	2.61e-04	2.09e-03
(2)	1.01e+02	1.79e-4	5.67e-5	1.80e-05	6.05e-06	1.87e-05	2.23e-04	1.65e-03
(3)	9.72e+02	1.79e-4	5.67e-5	1.80e-05	5.96e-06	2.11e-05	2.35e-04	2.00e-03
(4)	1.05e+03	1.79e-4	5.67e-5	1.79e-05	6.38e-06	2.61e-05	2.17e-04	2.20e-03
(5)	1.56e+02	1.79e-4	5.67e-5	1.80e-05	6.21e-06	2.38e-05	1.50e-04	2.38e-03
(6)	2.10e+09	1.79e-4	5.68e-5	8.69e-05	1.80e-03	3.85e-02	9.59e-01	1.63e+01
(7)	5.77e+09	1.79e-4	9.10e-5	1.83e-03	7.20e-02	2.30e+00	5.81e+01	1.42e+03
(8)	3.99e+03	1.79e-4	5.67e-5	1.80e-05	9.99e-06	8.48e-05	1.08e-03	7.15e-03
(9)	1.50e+03	1.79e-4	5.67e-5	1.80e-05	9.23e-06	6.07e-05	5.33e-04	6.19e-03
(10)	9.59e+16	1.79e-4	5.67e-5	1.80e-05	9.23e-06	6.07e-05	5.33e-04	6.19e-03
(11)	1.23e+02	1.79e-4	5.67e-5	1.80e-05	6.02e-06	1.44e-05	1.44e-04	1.36e-03
(12)	8.25e+01	1.79e-4	5.67e-5	1.80e-05	5.87e-06	1.49e-05	1.53e-04	1.22e-03
(13)	singular	1.79e-4	5.67e-5	1.80e-05	5.96e-06	1.42e-05	1.49e-04	1.24e-03
n=100								
(1)	singular	1.79e-4	5.64e-5	1.79e-05	6.98e-06	3.12e-05	3.48e-04	3.47e-03
(2)	1.59e+02	1.79e-4	5.64e-5	1.79e-05	6.49e-06	2.35e-05	2.35e-04	2.30e-03
(3)	7.06e+02	1.79e-4	5.64e-5	1.79e-05	6.49e-06	3.14e-05	3.49e-04	2.49e-03
(4)	2.43e+02	1.79e-4	5.64e-5	1.79e-05	6.91e-06	3.65e-05	3.20e-04	3.45e-03

(5)	9.26e+02	1.79e-4	5.64e-5	1.79e-05	7.53e-06	3.44e-05	3.45e-04	4.25e-03
(6)	singular	1.79e-4	5.66e-5	1.58e-04	3.94e-03	1.03e-01	2.09e+00	8.09e+01
(7)	5.30e+11	1.78e-4	3.40e-4	8.98e-03	2.24e-01	1.46e+01	1.34e+03	3.27e+04
(8)	2.25e+03	1.79e-4	5.64e-5	1.77e-05	1.63e-05	1.35e-04	1.29e-03	1.60e-02
(9)	singular	1.79e-4	5.64e-5	1.79e-05	1.05e-05	8.38e-05	9.38e-04	8.20e-03
(10)	singular	1.79e-4	5.64e-5	1.79e-05	1.05e-05	8.38e-05	9.38e-04	8.20e-03
(11)	1.27e+02	1.79e-4	5.64e-5	1.79e-05	6.09e-06	1.95e-05	1.71e-04	1.61e-03
(12)	8.94e+03	1.79e-4	5.64e-5	1.79e-05	6.15e-06	1.92e-05	1.85e-04	1.68e-03
(13)	singular	1.79e-4	5.64e-5	1.79e-05	6.19e-06	1.97e-05	1.93e-04	1.68e-03
n=500								
(1)	singular	1.79e-4	5.65e-5	1.79e-05	1.04e-05	8.22e-05	8.14e-04	8.00e-03
(2)	singular	1.79e-4	5.65e-5	1.79e-05	9.02e-06	5.84e-05	6.54e-04	6.28e-03
(3)	singular	1.79e-4	5.65e-5	1.79e-05	9.29e-06	6.43e-05	6.09e-04	6.35e-03
(4)	singular	1.79e-4	5.65e-5	1.79e-05	9.93e-06	7.56e-05	7.39e-04	8.52e-03
(5)	singular	1.79e-4	5.65e-5	1.79e-05	1.19e-05	9.78e-05	9.83e-04	9.88e-03
(6)	singular	1.79e-4	5.70e-5	2.31e-04	6.94e-03	1.92e-01	5.73e+00	1.67e+02
(7)	singular	1.79e-4	1.08e-3	4.17e-02	2.08e+00	1.69e+02	1.32e+04	6.86e+05
(8)	6.91e+04	1.79e-4	5.65e-5	2.21e-05	8.72e-05	6.91e-04	9.62e-03	5.84e-02
(9)	singular	1.79e-4	5.65e-5	1.80e-05	2.57e-05	2.54e-04	2.00e-03	2.52e-02
(10)	singular	1.79e-4	5.65e-5	1.80e-05	2.57e-05	2.54e-04	2.00e-03	2.52e-02
(11)	singular	1.79e-4	5.65e-5	1.79e-05	8.15e-06	4.49e-05	3.81e-04	3.98e-03
(12)	singular	1.79e-4	5.65e-5	1.79e-05	7.61e-06	4.04e-05	3.89e-04	3.68e-03
(13)	singular	1.79e-4	5.65e-5	1.79e-05	7.43e-06	3.92e-05	3.74e-04	3.69e-03

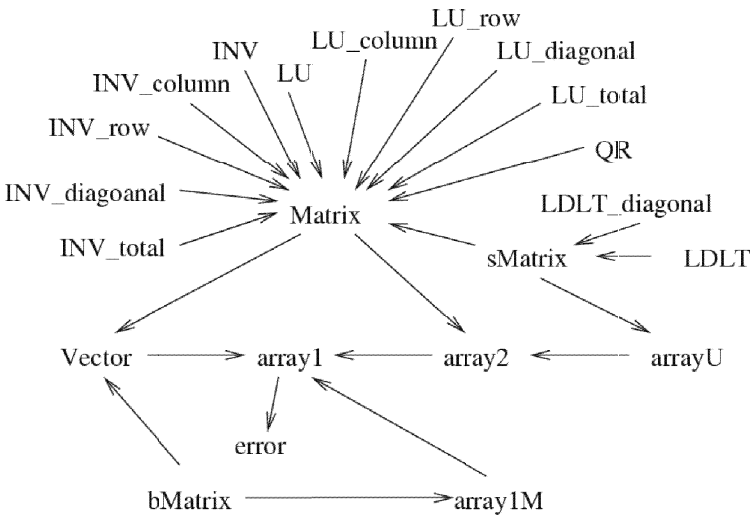
Für dieses Beispiel belegt die Tabelle, daß die Regularisierung wesentlich stärker die Genauigkeit erhöht als die Pivotisierung. Selbst beim Rechnen mit Zahlen vom Typ `float` liefert die Regularisierung noch brauchbare Ergbenisse, wie die folgende Tabelle zeigt.

	0.00e+00	1.00e-3	1.00e-4	1.00e-05	1.00e-06	1.00e-07	1.00e-08	1.00e-09
n=6								
(1)	3.32e-03	1.82e-2	5.22e-3	1.97e-03	1.10e-02	1.58e-02	2.75e-02	3.32e-03
(2)	6.69e-02	1.82e-2	5.11e-3	2.71e-03	9.93e-03	6.73e-02	3.20e-02	6.69e-02
(3)	2.86e-01	1.82e-2	5.29e-3	2.15e-03	1.20e-02	1.39e-01	2.06e-01	2.86e-01
(4)	7.03e-03	1.82e-2	5.16e-3	3.49e-03	7.03e-04	3.62e-02	1.09e-01	7.03e-03
(5)	9.20e-03	1.82e-2	5.17e-3	2.57e-03	5.49e-03	3.50e-02	3.18e-01	9.20e-03
(6)	9.98e-02	1.82e-2	5.19e-3	6.84e-03	1.11e-02	4.61e-01	5.42e-01	9.98e-02
(7)	3.06e-01	1.82e-2	5.09e-3	1.12e-02	2.85e-02	5.74e-02	6.32e-01	3.06e-01
(8)	4.79e-01	1.82e-2	5.02e-3	2.88e-03	2.99e-02	1.17e-01	4.62e-01	4.79e-01
(9)	0.00e+00	1.82e-2	4.93e-3	2.79e-03	6.38e-03	7.25e-02	2.17e-01	0.00e+00
(10)	2.93e-01	1.82e-2	5.34e-3	1.83e-03	1.28e-02	1.02e-01	2.99e-01	2.93e-01
(11)	3.47e-02	1.82e-2	5.14e-3	2.16e-03	1.09e-02	2.12e-02	2.72e-03	3.47e-02
(12)	3.07e-02	1.82e-2	5.17e-3	2.56e-03	4.35e-03	2.67e-02	5.06e-02	3.07e-02
(13)	7.95e-02	1.82e-2	5.13e-3	2.68e-03	4.32e-03	3.78e-02	8.80e-02	7.95e-02
n=10								
(1)	1.36e+01	1.77e-2	5.68e-3	3.51e-03	2.94e-02	2.18e-01	7.21e-01	1.36e+01
(2)	1.57e+01	1.77e-2	5.72e-3	4.69e-03	4.90e-02	4.84e-01	2.91e+00	1.57e+01
(3)	1.52e+01	1.77e-2	5.90e-3	4.94e-03	4.14e-02	2.49e-01	5.07e+00	1.52e+01
(4)	5.34e+00	1.77e-2	5.98e-3	4.84e-03	1.43e-02	7.11e-01	4.94e+00	5.34e+00
(5)	2.44e+01	1.77e-2	5.84e-3	4.46e-03	2.86e-02	2.10e-01	8.08e-01	2.44e+01
(6)	2.51e+02	1.77e-2	5.51e-3	2.33e-02	4.09e-01	5.27e+00	5.60e+01	2.51e+02
(7)	1.02e+03	1.77e-2	6.01e-3	5.06e-02	5.06e-01	4.23e+00	4.32e+01	1.02e+03
(8)	6.34e+01	1.77e-2	5.97e-3	8.65e-03	1.09e-01	4.63e-01	2.28e+01	6.34e+01
(9)	3.83e+01	1.77e-2	5.72e-3	5.52e-03	4.84e-02	5.32e-01	1.23e+01	3.83e+01
(10)	1.03e+02	1.77e-2	5.94e-3	7.57e-03	5.04e-02	9.91e-01	6.86e+00	1.03e+02
(11)	1.10e+01	1.77e-2	5.79e-3	3.47e-03	1.83e-02	1.44e-01	8.12e-01	1.10e+01
(12)	1.44e+01	1.77e-2	5.79e-3	2.98e-03	2.67e-02	5.93e-02	3.28e+00	1.44e+01
(13)	1.12e+01	1.77e-2	5.85e-3	3.33e-03	1.53e-02	1.43e-01	3.28e+00	1.12e+01
n=50								
(1)	9.43e+01	1.80e-2	5.87e-3	1.48e-02	1.15e-01	1.09e+00	3.12e+01	1.21e+02

(2)	6.53e+01	1.80e-2	5.68e-3	9.96e-03	9.51e-02	8.48e-01	4.21e+01	5.49e+02
(3)	9.97e+01	1.80e-2	5.79e-3	8.92e-03	9.01e-02	1.00e+00	1.70e+02	2.97e+02
(4)	1.28e+02	1.80e-2	5.85e-3	1.25e-02	1.44e-01	1.38e+00	4.72e+01	9.26e+01
(5)	1.90e+02	1.80e-2	5.79e-3	1.77e-02	1.10e-01	1.31e+00	5.06e+01	2.36e+02
(6)	1.73e+05	1.80e-2	7.51e-3	1.09e-01	2.62e+00	6.39e+01	1.28e+04	5.70e+06
(7)	6.29e+06	1.81e-2	1.30e-2	5.46e-01	2.99e+01	8.16e+02	1.27e+05	1.48e+06
(8)	4.58e+03	1.80e-2	7.71e-3	5.99e-02	3.80e-01	5.96e+00	4.20e+02	4.72e+02
(9)	3.95e+04	1.80e-2	6.48e-3	3.03e-02	2.96e-01	3.17e+00	1.83e+02	2.33e+03
(10)	2.20e+08	1.80e-2	6.01e-3	3.14e-02	2.32e-01	3.44e+00	3.06e+07	5.28e+08
(11)	6.78e+01	1.80e-2	5.65e-3	4.25e-03	4.33e-02	4.29e-01	8.10e+00	5.23e+02
(12)	1.21e+02	1.80e-2	5.61e-3	4.23e-03	4.29e-02	5.23e-01	2.05e+01	4.64e+01
(13)	4.92e+01	1.80e-2	5.64e-3	3.92e-03	4.33e-02	5.20e-01	3.25e+01	3.09e+01
n=100								
(1)	5.87e+02	1.80e-2	5.90e-3	1.69e-02	1.56e-01	1.50e+00	1.45e+02	3.00e+02
(2)	4.63e+02	1.80e-2	5.86e-3	8.55e-03	9.19e-02	1.05e+00	9.74e+01	1.31e+03
(3)	2.02e+03	1.80e-2	5.74e-3	1.05e-02	1.21e-01	1.20e+00	3.60e+02	1.37e+03
(4)	5.87e+03	1.80e-2	5.94e-3	1.90e-02	1.55e-01	1.94e+00	1.66e+02	5.19e+02
(5)	2.44e+02	1.80e-2	6.03e-3	2.00e-02	2.91e-01	2.23e+00	1.61e+02	3.90e+02
(6)	2.25e+06	1.81e-2	8.57e-3	1.59e-01	3.57e+00	1.02e+02	1.14e+05	5.62e+07
(7)	1.30e+07	1.81e-2	1.80e-2	7.05e-01	8.98e+01	2.06e+03	5.44e+05	7.08e+06
(8)	7.23e+03	1.80e-2	8.56e-3	7.58e-02	9.04e-01	8.69e+00	6.75e+02	3.13e+03
(9)	8.22e+03	1.80e-2	7.54e-3	5.13e-02	5.01e-01	4.29e+00	1.23e+03	1.16e+04
(10)	5.20e+08	1.80e-2	6.99e-3	4.28e-02	4.67e-01	4.46e+00	7.05e+08	1.32e+09
(11)	1.32e+03	1.80e-2	6.13e-3	8.74e-03	6.58e-02	4.31e-01	1.32e+01	1.54e+02
(12)	6.07e+01	1.80e-2	5.66e-3	6.27e-03	6.33e-02	6.07e-01	1.75e+01	1.19e+02
(13)	1.35e+02	1.80e-2	5.64e-3	5.61e-03	5.55e-02	5.74e-01	4.11e+01	8.94e+01
n=500								
(1)	2.10e+03	1.80e-2	6.94e-3	3.58e-02	3.71e-01	3.95e+00	1.73e+04	4.95e+03
(2)	5.66e+03	1.80e-2	7.13e-3	2.12e-02	3.12e-01	3.11e+00	6.98e+01	6.43e+03
(3)	4.08e+03	1.80e-2	6.78e-3	2.13e-02	2.85e-01	2.92e+00	9.86e+01	2.96e+03
(4)	5.98e+04	1.80e-2	7.23e-3	6.05e-02	8.81e-01	6.30e+00	1.60e+02	7.41e+03
(5)	9.30e+03	1.80e-2	9.19e-3	6.06e-02	2.23e+00	1.34e+01	2.59e+02	3.81e+03
(6)	4.31e+08	1.80e-2	1.29e-2	2.03e-01	5.16e+00	1.48e+02	1.14e+05	2.94e+08
(7)	3.60e+09	1.80e-2	2.46e-2	8.93e-01	1.01e+02	5.92e+03	1.13e+06	3.04e+07
(8)	8.98e+04	1.81e-2	1.64e-2	1.93e-01	2.68e+00	4.18e+01	4.03e+03	4.96e+04
(9)	8.87e+04	1.80e-2	1.49e-2	1.24e-01	1.28e+00	1.29e+01	2.99e+03	2.36e+05
(10)	4.19e+09	1.81e-2	1.55e-2	1.27e-01	1.26e+00	1.29e+01	2.98e+07	3.15e+09
(11)	8.93e+02	1.80e-2	5.97e-3	5.34e-02	3.23e-01	2.66e+00	4.71e+03	5.68e+04
(12)	1.79e+02	1.80e-2	5.77e-3	1.01e-02	8.92e-02	8.73e-01	6.60e+01	7.88e+02
(13)	1.16e+03	1.80e-2	5.72e-3	6.84e-03	6.29e-02	6.96e-01	1.55e+01	7.65e+02

5. Hinweise

Eine Klasse ist hier durch eine h-Datei und eine cpp-Datei repräsentiert. Die h-Datei enthält die Klassendefinition; in der cpp-Datei findet man die Implementation der Methoden. In der Anwendung der Klassen ist es nur erforderlich, die h-Datei der jeweils höchsten Klasse in ein Programm einzubinden, da die sonst noch benötigten Klassen nachgeladen werden. Die folgende Darstellung zeigt den Nachladegraphen.



Jeder Quell-Datei ist ein Identifikator zugeordnet: Der Identifikator für die Datei `ls_Vector.h` ist `LS_VECTOR`; für die Datei `ls_Vector.cpp` ist es `LS_VECTORC` usw.. Ein solcher Identifikator ist genau dann definiert, wenn die betreffende Datei geladen ist. Durch Abfragen des Identifikators vermeidet man, daß eine Datei doppelt geladen wird. Zu jeder `h`-Datei gehört eine `cpp`-Datei. Falls eine `h`-Datei geladen ist, wird an ihrem Ende die zugeordnete `cpp`-Datei geladen. Auf diese Weise wird einerseits erreicht, daß alle und nur die Dateien geladen sind, die zum Übersetzen benötigt werden; andererseits kann jede `cpp`-Datei separat übersetzt werden. Außerdem braucht das System beim Übersetzen eines Programms keine separat übersetzte `cpp`-Datei, da die benötigten Dateien während der Übersetzung eingebunden werden. Dies kann man als Vor- oder als Nachteil ansehen. Ein Nachteil ist, daß mit einer `h`-Datei stets auch die `cpp`-Datei geladen wird, was inhaltlich bedeutet, daß beide zu einer Datei vereinigt und die in einer Objektmodulbibliothek abgelegten Übersetzungen unbrauchbar sind. Der Identifikator `LS_LIB` behebt diesen Nachteil: Wenn er gesetzt ist, wird keine `cpp`-Datei geladen. Es ist sehr zu empfehlen, in einem solchen Falle für die `LS`-Klassen eine separate Objektmodulbibliothek zu erstellen.

6. Programmdokumentation

6.1. Die Klassen-Vorlage `ls_array1<T>` eindimensionales Datenfeld beliebigen Typs `T`

Nichtöffentliche Daten:

`T *a`

Zeiger auf das Datenfeld.

`ls_UINT dim`

Dimension des Feldes.

`char ONAME[ls_len]`

Feld für den Objektnamen.

`void set_data(T *aa=NULL, ls_UINT d=0)`

Öffentliche Daten/Methoden:

`char *name`

Zeiger auf den Objektnamen.

`ls_array1(ls_UINT l=0, char* ="ls_array1")`

Standard-Konstruktor; es wird ein `l`-Tupel kreiert.

`ls_array1(ls_array1<T> &)`

`move`-Konstruktor. Die dynamische Komponente des im Aufruf übergebenen Objektes wird übernommen.

`ls_array1(T*, ls_UINT, char* ="ls_array1")`

Das im Aufruf angegebene Datenfeld (mit der angegebenen Dimension) wird als dynamische Komponente kopiert.

`ls_array1<T> & swap(ls_array1<T> &)`

Datenfelder-Austausch.

`ls_UINT dimension() const`

Liefert die Dimension des Datenfeldes.

`T & operator[](ls_UINT) const`

indizierter Zugriff auf Datenelement

`T* asArray() const`

Liefert den Zeiger auf das Datenfeld.

`const ls_array1<T> & operator=(const ls_array1<T> &)`

Zuweisungsoperator; Objektname wird nicht übernommen.

`ls_array1<T> & put(T, ls_UINT)`

Ab der durch den 2. Parameter angebenen Stelle werden alle Datenelemente mit dem angegebenen Datum belegt.

`ls_array1<T> & put_array(const ls_array1<T> &, ls_UINT =0)`

Dateneingabe aus dem angegebenen Objekt.
Der `ls_UINT`- Parameter gibt den Index an, ab dem die Daten einzuspeichern sind. Die Anzahl der einzuspeichernden Daten ergibt sich aus der maximal möglichen Anzahl.

`T get(ls_UINT)`

Liefert den Wert an der angegebenen Stelle.

`const ls_array1<T> & get_array(ls_array1<T>&,ls_UINT) const`

Datenausgabe aus dem angegebenen Objekt.
Das Programm ist die Umkehrung der Methode
`put_array(const ls_array1<T> &, ls_UINT)`.

`ls_array1<T> & append(ls_UINT l=1)`

Das aktuelle Feld wird um `l` Nullelemente erweitert.

`ls_array1<T> & remove(ls_UINT i)`

Die `i`-te Komponete wird entfernt (Kürzung um ein Element).

`ls_array1<T> & remove()`

Das Datenfeld wird entfernt.

`const ls_array1<T> & write(ostream &) const`

Externe Datenausgabe.
Die Instanz wird in die durch den Aufruf spezifizierte Datei geschrieben. Das Schreiben geschieht in standardisierter Form: Den Daten wird der Objektname vorangestellt; danach folgen in geschweiften Klammern eingeschlossen - durch übliche Trennzeichen getrennt - die wesentlichen Daten der Instanz in folgender Form:
Schlüsselwort `dimension:`, Wert der Komponente `dim`, Datenelemente.

`ls_array1<T> & read(istream &)`

Externe Dateneingabe in eine leere Instanz.
Das Programm ist die Umkehrung der Methode `write(ostream &)`. Das Objekt muß leer sein; falls es einen Namen hat, wird dieser zunächst gesucht, andernfalls wird das als erstes gelesene Wort als Objektname genommen.

`const ls_array1<T> & operator >> (char *) const`

Datenausgabe in die durch den Aufrufparameter namentlich spezifizierte Datei; die Ausgabe erfolgt durch den Aufruf der Methode `write(ostream &)`.

`ls_array1<T> & operator << (char *)`

Dateneingabe aus der durch den Aufrufparameter namentlich spezifizierten Datei durch den Aufruf der Methode `read(istream &)`.

6.2. Die Klassen-Vorlage `ls_array2<T>` zweidimensionales Datenfeld beliebigen Typs

Abgeleitet von: `ls_array1<T>`

Nichtöffentliche Daten:

`ls_UINT m`

Zeilenanzahl.

`ls_UINT n`

Spaltenanzahl.

`void set_data(ls_UINT mm=0, ls_UINT nn=0)`

Daten neu setzen.

Öffentliche Methoden:

`ls_array2(ls_UINT mm=0, ls_UINT nn=0, char* ="ls_array2")`

Konstruktor, der eine Instanz mit `mm` Zeilen, `nn` Spalten erzeugt.

`ls_array2( T*, ls_UINT, ls_UINT, char* ="ls_array2")`

Konstruktor, der aus dem angegebenen Feld sein Datenfeld erzeugt.

`ls_array2(ls_array2<T> &)`

`move`-Konstruktor.

`ls_array2<T> & swap(ls_array2<T> &)`

Datenfelder-Austausch.

`T* operator[](ls_UINT) const`

Indizierter Zugriff auf ein Datenelement.

`ls_UINT number_of_rows() const`

Liefert die Zeilenanzahl.

`ls_UINT number_of_columns() const`

Liefert die Spaltenanzahl.

`const ls_array2<T> & operator=(const ls_array2<T> &)`

Zuweisungsoperator; Objektname wird nicht übernommen.

`ls_array2<T> & put(T, ls_UINT, ls_UINT)`

Eingabe eines Wertes auf die angegebene Stelle.

`ls_array2<T> & put_row(const ls_array1<T>&, ls_UINT, ls_UINT =0)`

Eingabe in eine Zeile ab angegebener Position.

`ls_array2<T> & put_row(T, ls_UINT, ls_UINT =0)`

Eingabe eines Wertes in eine Zeile ab angegebener Position.

`ls_array2<T> & put_column(const ls_array1<T> &, ls_UINT, ls_UINT)`

Eingabe in eine Spalte ab angegebener Position.

`ls_array2<T> & put_column(T, ls_UINT, ls_UINT)`

Eingabe eines Wertes in eine Spalte ab angegebener Position.

`ls_array2<T> & put_diagonal(const ls_array1<T> &,ls_UINT =0,ls_UINT =0)`

Eingabe in eine Diagonale ab angegebener Position.

`ls_array2<T> & put_diagonal(T, ls_UINT =0, ls_UINT =0)`

Eingabe eines Wertes in eine Diagonale ab angegebener Position.

`ls_array2<T> & put_array(const ls_array2<T> &, ls_UINT =0, ls_UINT =0)`

Eingabe eines zweidimensionalen Datenfeldes. Die letzten beiden Parameter bestimmen die Startposition (Zeile, Spalte) in der Zielinstanz.

`ls_array2<T> & put_array(T, ls_UINT =0, ls_UINT =0)`

Eingabe eines Wertes als zweidimensionales Datenfeld. Die letzten beiden Parameter bestimmen die Startposition (Zeile, Spalte) in der Zielinstanz.

`T get(ls_UINT, ls_UINT) const`

Elementausgabe aus der angegebenen Stelle.

`const ls_array2<T> & get_row(ls_array1<T> &, ls_UINT, ls_UINT =0) const`

Datenausgabe aus einer Zeile ab angegebener Position.

`const ls_array2<T> & get_column(ls_array1<T> &, ls_UINT, ls_UINT) const`

Datenausgabe aus einer Spalte ab angegebener Position.

`ls_array2<T> get_diagonal(ls_array1<T> &, ls_UINT =0, ls_UINT =0) const`

Datenausgabe aus einer Diagonalen ab angegebener Position.

`const ls_array2<T> & get_array(ls_array2<T> &, ls_UINT, ls_UINT)const`

Datenausgabe eines zweidimensionalen Teilfeldes.

`ls_array2<T> & append_row(ls_UINT l=1)`

Anhängen von `l` Null-Zeilen.

`ls_array2<T> & append_column(ls_UINT l=1)`

Anhängen von `l` Null-Spalten.

`ls_array2<T> & remove_column(ls_UINT)`

Entfernen einer Spalte.

`ls_array2<T> & remove_row(ls_UINT)`

Entfernen einer Zeile.

`ls_array2<T> & remove()`

Entfernen des gesamten Feldes.

`ls_array2<T> & swap_row(ls_UINT, ls_UINT)`

Zeilentausch.

`ls_array2<T> & swap_column(ls_UINT, ls_UINT)`

Spaltentausch.

`const ls_array2<T> & write_row(ostream &) const`

Externe, zeilenweise Datenausgabe.
Das Schreiben geschieht in standardisierter Form: Den Daten wird der Objektname vorangestellt; danach folgen in geschweiften Klammern eingeschlossen - durch übliche Trennzeichen getrennt - die wesentlichen Daten der Instanz in folgender Form:
Schlüsselwort `number_of_rows:`, Wert der Komponente `m`,
Schlüsselwort `number_of_columns:`, Wert der Komponente `n`,
Datenelemente.

`ls_array2<T> & read_row(istream &)`

Das Programm ist die Umkehrung der Methode `write_row(ostream &)`.

`const ls_array2<T> & write_column(ostream &) const`

Externe, spaltenweise Datenausgabe.
Die Ausgabe erfolgt analog zur zeilenweisen Ausgabe, jedoch werden die Kopfdaten in der Reihenfolge

`number_of_columns:`, `number_of_rows:`

geschrieben.

`ls_array2<T> & read_column(istream &)`

Externe, spaltenweise Dateneingabe.
Das Programm ist die Umkehrung der Methode
`write_column(ostream &)`.

`const ls_array2<T> & operator >> (char *) const`

Datenausgabe in die durch den Aufrufparameter namentlich spezifizierte Datei durch den Aufruf der Methode `write_row(ostream &)`.

`ls_array2<T> & operator << (char *)`

Dateneingabe aus der durch den Aufrufparameter namentlich spezifizierten Datei durch den Aufruf der Methode `read_row(istream &)`.

6.3. Die Klassen-Vorlage `ls_arrayU<T>` symmetrisches, zweidimensionales, oberes Dreiecksfeld

Abgeleitet von: `ls_array1<T>`

Nichtöffentliche Daten:

`ls_UINT n`

Zeilen- und Spaltenanzahl.

```
void set_data(ls_UINT)
```

Daten neu setzen.

Öffentliche Methoden:

```
ls_arrayU(ls_UINT nn=0, char* ="ls_arrayU")
```

Konstruktor, der eine Instanz mit `nn` Zeilen und Spalten erzeugt.

```
arrayU(ls_arrayU<T> &)
```

`move`-Konstruktor.

```
ls_arrayU<T> & swap(ls_arrayU<T> &)
```

Datenfelder-Austausch.

```
T* operator[](ls_UINT) const
```

Indizierter Zugriff auf ein Datenelement. Dabei ist zu berücksichtigen, daß so indiziert wird, als ob die Datenelemente des unteren Dreiecks abgespeichert wären.

```
ls_UINT dimension() const
```

Liefert die Zeilenanzahl (= Spaltenanzahl).

```
const ls_arrayU<T> & operator=(const ls_arrayU<T> &)
```

Zuweisungsoperator; Objektname wird nicht übernommen.

```
ls_arrayU<T> & put(T, ls_UINT, ls_UINT)
```

Elementeingabe.

```
ls_arrayU<T> & put_row(const ls_array1<T> &, ls_UINT,ls_UINT =0)
```

Eingabe in eine Zeile ab angegebener Position.

```
ls_arrayU<T> & put_row(T, ls_UINT,ls_UINT)
```

Eingabe des angegebenen Wertes in eine Zeile ab angegebener Position.

```
ls_arrayU<T> & put_diagonal(const ls_array1<T> &, ls_UINT =0, ls_UINT =0
```

Eingabe in eine Diagonale ab angegebener Position.

```
ls_arrayU<T> & put_diagonal(T, ls_UINT =0, ls_UINT =0
```

Eingabe des angegebenen Wertes in eine Diagonale ab angegebener Position.

```
T get(ls_UINT, ls_UINT) const
```

Elementausgabe.

```
ls_arrayU<T> & get_row(ls_array1<T> &, ls_UINT, ls_UINT =0) const
```

Datenausgabe aus einer Zeile ab angegebener Position.

```
ls_arrayU<T> & get_diagonal(ls_array1<T> &,ls_UINT =0,ls_UINT =0)const
```

Datenausgabe aus einer Diagonalen ab angegebener Position.

```
const ls_arrayU<T> & get_array(ls_array2<T> &) const
```

Konvertierung in ein zweidimensionales Feld.

```
append_row(ls_UINT =1)
```

Anhängen von Null-Zeilen (Spalten).

```
ls_arrayU<T> & remove(ls_UINT)
```

Entfernen einer Zeile (Spalte).

```
ls_arrayU<T> & remove()
```

Entfernen des gesamten Datenfeldes.

```
ls_arrayU<T> & swap(ls_UINT, ls_UINT)
```

Symmetrischer Zeilen/Spaltentausch.

```
const ls_arrayU<T> & write_row(ostream &) const
```

Externe, zeilenweise Datenausgabe.
Das Schreiben geschieht in standardisierter Form: Den Daten wird der Objektname vorangestellt; danach folgen in geschweiften Klammern eingeschlossen - durch übliche Trennzeichen getrennt - die wesentlichen Daten der Instanz in folgender Form:
Schlüsselwort `dimension:`, Wert der Komponente `n`,
Datenelemente.

```
ls_arrayU<T> & read_row(istream &)
```

Externe, zeilenweise Dateneingabe.
Das Programm ist die Umkehrung der Methode `write_row(ostream &)`.

```
const ls_arrayU<T> & write_column(ostream &) const
```

Externe, spaltenweise Datenausgabe.

```
ls_arrayU<T> & read_column(istream &)
```

Externe, spaltenweise Dateneingabe.
Das Programm ist die Umkehrung der Methode
`write_column(ostream &)`.

```
const ls_arrayU<T> & operator >> (char *) const
```

Datenausgabe in die durch den Aufrufparameter namentlich spezifizierte Datei durch den Aufruf der Methode `write_row(ostream &)`.

```
ls_arrayU<T> & operator << (char *)
```

Dateneingabe aus der durch den Aufrufparameter namentlich spezifizierten Datei durch den Aufruf der Methode `read_row(istream &)`.

6.4. Klassen-Vorlage `ls_array1M<T>` mehrere, eindimensionale Datenfelder

Nichtöffentliche Daten:

`array1M<T> *a`

Zeiger auf die Datenfelder; jedes Datenfeld hat die Struktur

`struct array1M{ls_UINT l; T *f;}`

hierin findet man auf `l` die Feldlänge und auf `f` den Zeiger auf das Datenfeld.

`ls_UINT n`

Dimension des Feldes `a`, d. h. Anzahl der zu verwaltenden Felder.

`char ONAME[ls_len]`

Feld für den Objektnamen.

`void set_data(ls_UINT nn)`

Daten neu setzen.

Öffentliche Daten/Methoden:

`char *name`

Zeiger auf den Objektnamen.

`ls_array1M(ls_UINT nn, char* ="ls_array1M")`

Konstruktor, der eine Instanz mit `nn` Datenfeldern erzeugt.

`ls_array1M(ls_array1M<T> &)`

`move`-Konstruktor.

`ls_array1M<T> & swap(ls_array1M<T> &)`

Datenfelder-Austausch.

`ls_UINT number_of_arrays() const`

Liefert die Anzahl der Datenfelder.

`ls_UINT length(ls_UINT i) const`

Liefert die Dimension des `i`-ten Datenfeldes.

`array1M<T>* asArray() const`

Liefert den Zeiger auf das Datenfeld `a`.

`const ls_array1M<T> & operator=(ls_array1M<T> &)`

Zuweisungsoperator; Objektname wird nicht übernommen.

`T* operator[](ls_UINT i)`

indizierter Zugriff auf das `i`-te Datenfeld.

```
ls_array1M<T> & put_array(const ls_array1<T> &v, ls_UINT i)
```

Eingabe des Datenfeldes aus der Instanz `v` als `i`-tes Datenfeld.

```
ls_array1M<T> & put(T w, ls_UINT i)
```

Das `i`-te Datenfeld erhält den Wert `w`.

```
ls_array1M<T> & put(T* w, ls_UINT l, ls_UINT i)
```

Das `i`-te Datenfeld erhält das Feld w mit der Länge `l`.

```
ls_arrayM<T> & get_array(ls_array1<T> &v, ls_UINT i) const
```

Das `i`-te Datenfeld wird ausgegeben.

```
ls_array1M<T> & remove_array(ls_UINT)
```

Ein Datenfeld wird entfernt.

```
ls_array1M<T> & remove()
```

Alle Datenfelder werden entfernt.

```
write(ostream &)
```

Externe Datenausgabe.
Das Schreiben geschieht in standardisierter Form: Den Daten wird der Objektname vorangestellt; danach folgen in geschweiften Klammern eingeschlossen - durch übliche Trennzeichen getrennt - die wesentlichen Daten der Instanz in folgender Form: Nach der öffnenden geschweiften Klammer steht das Schlüsselwort `dimension:`; dem folgt der Wert der Komponente `dim`;
danach stehen das Schlüsselwort `maximal_length:`, der Wert der Komponente `n`; das `i`-te Datenfeld wird durch `array:`, `i`, `elements:` und Feldlänge eingeleitet, wonach die Daten folgen.

```
ls_array1M<T> & read(istream &)
```

Externe Dateneingabe.
Das Programm ist die Umkehrung der Methode `write(ostream &)`.

```
const ls_array1M<T> & operator >> (char *) const
```

Datenausgabe in die durch den Aufrufparameter namentlich spezifizierte Datei; die Ausgabe erfolgt durch den Aufruf der Methode `write(ostream &)`.

```
ls_array1M<T> & operator << (char *)
```

Dateneingabe aus der durch den Aufrufparameter namentlich spezifizierten Datei durch den Aufruf der Methode `read(istream &)`.

6.5. `ls_Vector` - Vektoren

Abgeleitet von: `ls_array1<ls_REAL>`

Öffentliche Daten:

`static ls_REAL eps`

Genauigkeitsschranke bei arithmetischen Operationen;

Öffentliche Methoden:

`ls_Vector(ls_UINT =0, char* ="ls_Vector")`

Instanziierung eines (Null-)Vektors mit der im Aufruf angegebenen Dimension.

`ls_Vector(ls_Vector &)`

`move`-Konstruktor.

`ls_Vector(ls_REAL *, ls_UINT, char* ="ls_Vector")`

Die Daten des im Aufruf angegebenen Datenfeldes (mit der angegebenen Dimension) werden als dynamische Komponente übernommen.

`const ls_Vector & operator= (const ls_Vector &)`

Zuweisungsoperator; Objektname wird nicht übernommen.

`ls_Vector operator-() const`

Negativer Vektor.

`ls_Vector operator*(ls_REAL) const`

Multiplikation eines Vektors `v` mit einer Zahl `s`: `v * s`.

`ls_Vector & operator*=(ls_REAL)`

Speichernde Multiplikation eines Vektors `v` mit einer Zahl `s`: `v=v * s`.

`ls_Vector operator+(const ls_Vector &) const`

Addition zweier Vektoren.

`ls_Vector operator-(const ls_Vector &) const`

Subtraktion zweier Vektoren.

`ls_Vector & operator+=(const ls_Vector &)`

Speichernde Addition eines Vektors: `v=v + w`.

`ls_Vector & operator-=(const ls_Vector &)`

Speichernde Subtraktion eines Vektors: `v=v - w`.

`ls_REAL operator*(const ls_Vector &) const`

Skalarpodukt zweier Vektoren: `s=v * w`.

`ls_Matrix dyad(const ls_Vector &) const`

Dyadisches Produkt zweier Vektoren; das Ergebnis (= Rückgabewert) ist eine Matrix.

`ls_sMatrix dyad() const`

Dyadisches Produkt eines Vektors mit sich; das Ergebnis (= Rückgabewert) ist eine symmetrische Matrix.

`ls_Vector operator*(ls_REAL, const ls_Vector &)`

Multiplikation eines Vektors mit einer Zahl: `w=s * v`.

6.6. `ls_Matrix` - allgemeine Matrizen

Abgeleitet von: `ls_array2<ls_REAL>`

Öffentliche Methoden:

`ls_Matrix(ls_UINT mm=0, ls_UINT nn=0, char* ="ls_Matrix")`

Instanziierung einer (Null-)Matrix mit `mm` Zeilen und `nn` Spalten.

`ls_Matrix(ls_Matrix &)`

`move`-Konstruktor.

`ls_Vector row(ls_UINT) const`

Liefert eine Zeile der Matrix.

`ls_Vector column(ls_UINT) const`

Liefert eine Spalte der Matrix.

`ls_Vector upper_diagonal(ls_UINT) const`

Liefert eine obere Diagonale der Matrix.

`ls_Vector lower_diagonal(ls_UINT) const`

Liefert eine untere Diagonale der Matrix.

`const ls_Matrix & operator= (const ls_Matrix &)`

Zuweisungsoperator; Objektname wird nicht übernommen.

`ls_Matrix & operator+=(const ls_Matrix &)`

Speichernde Addition zweier Matrizen `A` und `B`: `A=A + B`.

`ls_Matrix operator-(const ls_Matrix &) const`

Subtraktion zweier Matrizen.

`ls_Matrix & operator-=(const ls_Matrix &)`

Speichernde Subtraktion zweier Matrizen `A` und `B`: `A=A - B`.

`ls_Matrix operator+(const ls_Matrix &) const`

Addition zweier Matrizen.

`ls_Matrix operator*(const ls_Matrix &) const`

Multiplikation zweier Matrizen.

`ls_Vector operator*(const ls_Vector &) const`

Multiplikation einer Matrix mit einem Vektor `v`: `A * v`.

`ls_Matrix operator*(ls_REAL) const`

Multiplikation einer Matrix mit einer Zahl `s`: `A * s`.

`ls_Matrix & operator*=(ls_REAL)`

Speichernde Multiplikation einer Matrix mit einer Zahl `s`: `A=A * s`.

`ls_UINT solve(ls_Vector &x,ls_Vector &b)`

Lösen eines linearen Gleichungssystem mit der rechten Seite `b`. Es wird das konjugierte Gradientenverfahren benutzt; dabei ist `x` der Startvektor. Bei Rückkehr aus dem Programm stehen auf `x` die gefundene Lösung und auf `b` das Residuum. Der Rückgabewert ist die Anzahl der ausgeführten Schritte.
Falls die Koeffizientenmatrix mehr Zeilen als Spalten hat, wird das entsprechende lineare Ausgleichsproblem gelöst.

`ls_Matrix operator*(ls_REAL, const ls_Matrix &)`

Multiplikation einer Matrix mit einer Zahl `s`: `s * A`.

`ls_Vector operator*(const ls_Vector &, const ls_Matrix &)`

Multiplikation der transponierten Matrix mit Vektor `v`: `v * A`.

6.7. `ls_sMatrix` - symmetrische Matrizen

Abgeleitet von: `ls_arrayU<ls_REAL>`

Öffentliche Methoden:

`ls_sMatrix(ls_UINT nn=0, char* ="ls_sMatrix")`

Instanziierung einer symmetrischen (Null-)Matrix mit `nn` Zeilen/Spalten.

`ls_sMatrix(ls_sMatrix &)`

`move`-Konstruktor.

`ls_Vector row(ls_UINT) const`

Liefert eine (vollständige) Zeile der Matrix.

`ls_Vector diagonal(ls_UINT) const`

Liefert eine (vollständige) Diagonale der Matrix.

`const ls_sMatrix & operator= (const ls_sMatrix &)`

Zuweisungsoperator; Objektname wird nicht übernommen.

`ls_sMatrix & operator+=(const ls_sMatrix &)`

Speichernde Addition zweier Matrizen: `A=A + B`.

`ls_sMatrix operator-(const ls_sMatrix &) const`

Subtraktion zweier Matrizen.

`ls_sMatrix & operator-=(const ls_sMatrix &)`

Speichernde Subtraktion zweier Matrizen : `A=A - B`.

`ls_sMatrix operator+(const ls_sMatrix &) const`

Addition zweier Matrizen.

`ls_Vector operator*(const ls_Vector &) const`

Multiplikation einer Matrix mit einem Vektor `v`: `A * v`.

`ls_sMatrix operator*(ls_REAL) const`

Multiplikation einer Matrix mit einer Zahl `s`: `A * s`.

`ls_sMatrix & operator*=(ls_REAL)`

Speichernde Multiplikation einer Matrix mit einer Zahl `s`: `A=A * s`.

`ls_UINT solve(ls_Vector &x,ls_Vector &b)`

Lösen eines linearen Gleichungssystem mit der rechten Seite `b`. Es wird das konjugierte Gradientenverfahren benutzt; dabei ist `x` der Startvektor. Bei Rückkehr aus dem Programm stehen auf `x` die gefundene Lösung und auf `b` das Residuum. Der Rückgabewert ist die Anzahl der ausgeführten Schritte.

`ls_sMatrix operator*(ls_REAL, const ls_sMatrix &)`

Multiplikation einer symmetrischen Matrix mit einer Zahl `s`: `s * A`.

6.8. `ls_bMatrix` - symmetrische Band-Matrizen

Abgeleitet von: `ls_array1M<ls_REAL>`

Öffentliche Methoden:

`ls_bMatrix(ls_UINT d=0)`

Instanziierung einer symmetrischen (Null-)Band-Matrix mit einer oberen Bandbreite von `d`. Die Instanz widerspiegelt eine symmetrische Band-Matrix, bei der nur das obere Dreieck abgespeichert ist und dieses diagonalweise. Diagonalen, bei denen alle Elemente gleich sind, dürfen als einelementig eingespeichert sein.

`ls_bMatrix(ls_bMatrix &)`

move-Konstruktor.

`const ls_bMatrix & operator=(const ls_bMatrix &)`

Zuweisungsoperator.

`ls_Vector operator*(const ls_Vector &) const`

Multiplikation einer Matrix mit einem Vektor `v`: `A * v`.

`ls_UINT solve(ls_Vector &x,ls_Vector &b)`

Lösen eines linearen Gleichungssystem mit der rechten Seite `b`. Es wird das konjugierte Gradientenverfahren benutzt; dabei ist `x` der Startvektor. Bei Rückkehr aus dem Programm stehen auf `x` die gefundene Lösung und auf `b` das Residuum. Der Rückgabewert ist die Anzahl der ausgeführten Schritte.

6.9. `ls_QR` - Householder-Faktorisierung

Nichtöffentliche Daten/Methoden:

`char ONAME[ls_len]`

Feld für den Objektnamen.

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_REAL> gamma, rho`

Hilfsfelder.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *gamma_name, *rho_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_QR(char* ="ls_QR")`

Standard-Konstruktor.

`ls_QR(ls_Matrix &, ls_REAL =0., char* ="ls_QR")`

move- Konstruktor. Es wird eine `QR`-Faktorisierung der Matrix berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_QR(ls_QR &)`

`move`- Konstruktor.

`ls_QR & swap(ls_QR &)`

Datenfelder-Austausch.

`const ls_QR & operator= (const ls_QR &)`

Zuweisungsoperator.

`const ls_QR & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Qx(const ls_Vector &x) const`

Multiplikation der Orthogonalmatrix mit einem Vektor.

`ls_Vector xQ(const ls_Vector &x) const`

Multiplikation eines Vektors mit der Orthogonalmatrix.

`ls_Vector Rx(const ls_Vector &x) const`

Multiplikation der oberen Dreiecksmatrix mit einem Vektor.

`ls_Vector xR(const ls_Vector &x) const`

Multiplikation eines Vektors mit der oberen Dreiecksmatrix.

`const ls_QR & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_QR & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_QR & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_QR & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_QR & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_QR & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei in eine leere Instanz.

6.10. `ls_LU` - Faktorisierung ohne Pivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_LU(char* ="ls_LU")`

Standard-Konstruktor.

`ls_LU(ls_Matrix &, ls_REAL =0., char* ="ls_LU")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und eine Faktorisierung berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_LU(ls_LU &)`

move-Konstruktor.

`ls_LU & swap(ls_LU &)`

Datenfelder-Austausch.

`const ls_LU & operator= (const ls_LU &)`

Zuweisungsoperator.

`const ls_LU & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Lx(const ls_Vector &x) const`

Multiplikation der unteren Dreiecksmatrix mit einem Vektor.

`ls_Vector xL(const ls_Vector &x) const`

Multiplikation eines Vektors mit der unteren Dreiecksmatrix.

`ls_Vector Ux(const ls_Vector &x) const`

Multiplikation der oberen Dreiecksmatrix mit einem Vektor.

`ls_Vector xU(const ls_Vector &x) const`

Multiplikation eines Vektors mit der oberen Dreiecksmatrix.

`const ls_LU & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_LU & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_LU & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_LU & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.11. `ls_LU_column` - Faktorisierung mit Spalten-Pivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> ind`

Indexfeld für die Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *ind_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_LU_column(char* ="ls_LU_column")`

Standard-Konstruktor.

`ls_LU_column(ls_Matrix &, ls_REAL =0., char* ="ls_LU_column")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und eine Faktorisierung mit Spaltenpivotisierung und virtueller Skalierung berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_LU_column(ls_LU_column &)`

move-Konstruktor.

`ls_LU_column & swap(ls_LU_column &)`

Datenfelder-Austausch.

`const ls_LU_column & operator= (const ls_LU_column &)`

Zuweisungsoperator.

`const ls_LU_column & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Lx(const ls_Vector &x) const`

Multiplikation der unteren Dreiecksmatrix mit einem Vektor.

`ls_Vector xL(const ls_Vector &x) const`

Multiplikation eines Vektors mit der unteren Dreiecksmatrix.

`ls_Vector Ux(const ls_Vector &x) const`

Multiplikation der oberen Dreiecksmatrix mit einem Vektor.

`ls_Vector xU(const ls_Vector &x) const`

Multiplikation eines Vektors mit der oberen Dreiecksmatrix.

`const ls_LU_column & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_column & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_LU_column & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_column & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_LU_column & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_LU_column & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.12. `ls_LU_diagonal` - Faktorisierung mit Diagonalpivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> ind`

Indexfeld für die Diagonalindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *ind_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_LU_diagonal(char* ="ls_LU_diagonal")`

Standard-Konstruktor.

`ls_LU_diagonal(ls_Matrix &, ls_REAL =0.,char* ="ls_LU_diagonal")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und eine Faktorisierung mit Diagonal-Pivotisierung berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_LU_diagonal(ls_LU_diagonal &)`

move-Konstruktor.

`ls_LU_diagonal & swap(ls_LU_diagonal &)`

Datenfelder-Austausch.

`const ls_LU_diagonal & operator= (const ls_LU_diagonal &)`

Zuweisungsoperator.

`const ls_LU_diagonal & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Lx(const ls_Vector &x) const`

Multiplikation der unteren Dreiecksmatrix mit einem Vektor.

`ls_Vector xL(const ls_Vector &x) const`

Multiplikation eines Vektors mit der unteren Dreiecksmatrix.

`ls_Vector Ux(const ls_Vector &x) const`

Multiplikation der oberen Dreiecksmatrix mit einem Vektor.

`ls_Vector xU(const ls_Vector &x) const`

Multiplikation eines Vektors mit der oberen Dreiecksmatrix.

`const ls_LU_diagonal & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_diagonal & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LU_diagonal & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_diagonal & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LU_diagonal & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_LU_diagonal & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.13. `ls_LU_row` - Faktorisierung mit Zeilen-Pivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> ind`

Indexfeld für die Zeilenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *ind_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_LU_row(char* ="ls_LU_row")`

Standard-Konstruktor.

`ls_LU_row(ls_Matrix &, ls_REAL =0., char* ="ls_LU_row")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und eine Faktorisierung mit Zeilen-Pivotisierung und virtueller Skalierung berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_LU_row(ls_LU_row &)`

move-Konstruktor.

`ls_LU_row & swap(ls_LU_row &)`

Datenfelder-Austausch.

`const ls_LU_row & operator= (const ls_LU_row &)`

Zuweisungsoperator.

`const ls_LU_row & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Lx(const ls_Vector &x) const`

Multiplikation der unteren Dreiecksmatrix mit einem Vektor.

`ls_Vector xL(const ls_Vector &x) const`

Multiplikation eines Vektors mit der unteren Dreiecksmatrix.

`ls_Vector Ux(const ls_Vector &x) const`

Multiplikation der oberen Dreiecksmatrix mit einem Vektor.

`ls_Vector xU(const ls_Vector &x) const`

Multiplikation eines Vektors mit der oberen Dreiecksmatrix.

`const ls_LU_row & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_row & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LU_row & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_row & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LU_row & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_LU_row & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.14. `ls_LU_total` - Faktorisierung mit Total-Pivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> indr, indc`

Indexfelder für die wahren Zeilen- und Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *indr_name, *indc_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_LU_total(char* ="ls_LU_total")`

Standard-Konstruktor.

`ls_LU_total(ls_Matrix &, ls_REAL =0.,char* ="ls_LU_total")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und eine Faktorisierung mit Total-Pivotisierung berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_LU_total(ls_LU_total &)`

move-Konstruktor.

`ls_LU_total & swap(ls_LU_total &)`

Datenfelder-Austausch.

`const ls_LU_total & operator= (const ls_LU_total &)`

Zuweisungsoperator.

`const ls_LU_total & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Lx(const ls_Vector &x) const`

Multiplikation der unteren Dreiecksmatrix mit einem Vektor.

`ls_Vector xL(const ls_Vector &x) const`

Multiplikation eines Vektors mit der unteren Dreiecksmatrix.

`ls_Vector Ux(const ls_Vector &x) const`

Multiplikation der oberen Dreiecksmatrix mit einem Vektor.

`ls_Vector xU(const ls_Vector &x) const`

Multiplikation eines Vektors mit der oberen Dreiecksmatrix.

`const ls_LU_total & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_total & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LU_total & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LU_total & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LU_total & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_LU_total & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.15. `ls_INV` - Invertierung ohne Pivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_INV(char* ="ls_INV")`

Standard-Konstruktor.

`ls_INV(ls_Matrix &, ls_REAL =0., char* ="ls_INV")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und invertiert, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_INV(ls_INV &)`

`move`-Konstruktor.

`ls_INV & swap(ls_INV &)`

Datenfelder-Austausch.

```
const ls_INV & operator= (const ls_INV &)
```

Zuweisungsoperator.

```
const ls_INV & solve(ls_Vector &, const ls_Vector &) const
```

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

```
ls_UINT post_iteration(ls_Vector &, ls_Vector &) const
```

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

```
ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const
```

Berechnung des Residuums bei vorliegender Faktorisierung.

```
ls_Vector Fx(const ls_Vector &x) const
```

Multiplikation der inversen Matrix mit einem Vektor.

```
ls_Vector xF(const ls_Vector &x) const
```

Multiplikation eines Vektors mit der inversen Matrix.

```
const ls_INV & write_row(ostream &) const
```

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

```
ls_INV & read_row(istream &)
```

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

```
const ls_INV & write_column(ostream &) const
```

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

```
ls_INV & read_column(istream &)
```

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

```
const ls_INV & operator>> (char *) const
```

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

```
ls_INV & operator<< (char *)
```

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.16. `ls_INV_column` - Invertierung mit Spaltenpivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> ind`

Feld mit den wahren Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *ind_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_INV_column(char* ="ls_INV_column")`

Standard-Konstruktor.

`ls_INV_column(ls_Matrix &, ls_REAL =0., char* ="ls_INV_column")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und invertiert, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_INV_column(ls_INV_column &)`

`move`-Konstruktor.

`ls_INV_column & swap(ls_INV_column &)`

Datenfelder-Austausch.

`const ls_INV_column & operator= (const ls_INV_column &)`

Zuweisungsoperator.

`const ls_INV_column & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Fx(const ls_Vector &x) const`

Multiplikation der inversen Matrix mit einem Vektor.

`ls_Vector xF(const ls_Vector &x) const`

Multiplikation eines Vektors mit der inversen Matrix.

`const ls_INV_column & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_INV_column & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_column & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_INV_column & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_column & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_INV_column & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.17. `ls_INV_row` - Invertierung mit Zeilenpivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> ind`

Feld mit den wahren Zeilenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *ind_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_INV_row(char* ="ls_INV_row")`

Standard-Konstruktor.

`ls_INV_row(ls_Matrix &, ls_REAL =0., char* ="ls_INV_row")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und invertiert, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_INV_row(ls_INV_row &)`

`move`-Konstruktor.

`ls_INV_row & swap(ls_INV_row &)`

Datenfelder-Austausch.

`const ls_INV_row & operator= (const ls_INV_row &)`

Zuweisungsoperator.

`const ls_INV_row & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Fx(const ls_Vector &x) const`

Multiplikation der inversen Matrix mit einem Vektor.

`ls_Vector xF(const ls_Vector &x) const`

Multiplikation eines Vektors mit der inversen Matrix.

`const ls_INV_row & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_INV_row & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_row & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_INV_row & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_row & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_INV_row & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.18. `ls_INV_diagonal` - Invertierung mit Diagonalpivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> ind`

Feld mit den wahren Diagonalindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *ind_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_INV_diagonal(char* ="ls_INV_diagonal")`

Standard-Konstruktor.

`ls_INV_diagonal(ls_Matrix &, ls_REAL =0., char* ="ls_INV_diagonal")`

`move`-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und invertiert, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_INV_diagonal(ls_INV_diagonal &)`

`move`-Konstruktor.

`ls_INV_diagonal & swap(ls_INV_diagonal &)`

Datenfelder-Austausch.

`const ls_INV_diagonal & operator= (const ls_INV_diagonal &)`

Zuweisungsoperator.

`const ls_INV_diagonal & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Fx(const ls_Vector &x) const`

Multiplikation der inversen Matrix mit einem Vektor.

`ls_Vector xF(const ls_Vector &x) const`

Multiplikation eines Vektors mit der inversen Matrix.

`const ls_INV_diagonal & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_INV_diagonal & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_diagonal & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_INV_diagonal & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_diagonal & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_INV_diagonal & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.19. `ls_INV_total` - Invertierung mit Totalpivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> indr, indc`

Felder mit den wahren Zeilen- und Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *indr_name, *indc_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

```
ls_INV_total(char* ="ls_INV_total")
```

Standard-Konstruktor.

```
ls_INV_total(ls_Matrix &, ls_REAL =0., char* ="ls_INV_total")
```

move-Konstruktor. Die angegebene, quadratische Matrix wird übernommen und invertiert, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

```
unsigned char good() const
```

Liefert Signal für erfolgreiche Faktorisierung.

```
ls_INV_total(ls_INV_total &)
```

move-Konstruktor.

```
ls_INV_total & swap(ls_INV_total &)
```

Datenfelder-Austausch.

```
const ls_INV_total & operator= (const ls_INV_total &)
```

Zuweisungsoperator.

```
const ls_INV_total & solve(ls_Vector &, const ls_Vector &) const
```

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

```
ls_UINT post_iteration(ls_Vector &, ls_Vector &) const
```

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

```
ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const
```

Berechnung des Residuums bei vorliegender Faktorisierung.

```
ls_Vector Fx(const ls_Vector &x) const
```

Multiplikation der inversen Matrix mit einem Vektor.

```
ls_Vector xF(const ls_Vector &x) const
```

Multiplikation eines Vektors mit der inversen Matrix.

```
const ls_INV_total & write_row(ostream &) const
```

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

```
ls_INV_total & read_row(istream &)
```

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_total & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_INV_total & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_INV_total & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_INV_total & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.20. `ls_LDLT` - Faktorisierung einer symmetrischen Matrix

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Faktorisierte Matrix.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`ls_LDLT(char* ="ls_LDLT")`

Standard-Konstruktor.

`ls_LDLT(ls_sMatrix &, ls_REAL =0., char* ="ls_LDLT")`

`move`-Konstruktor. Die angegebene, symmetrische Matrix wird übernommen und eine Faktorisierung berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_LDLT(ls_LDLT &)`

move-Konstruktor.

`ls_LDLT & swap(ls_LDLT &)`

Datenfelder-Austausch.

`const ls_LDLT & operator= (const ls_LDLT &)`

Zuweisungsoperator.

`const ls_LDLT & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Lx(const ls_Vector &x) const`

Multiplikation des Cholesky-Faktors mit einem Vektor.

`ls_Vector xL(const ls_Vector &x) const`

Multiplikation eines Vektors mit dem Cholesky-Faktor.

`const ls_LDLT & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LDLT & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_LDLT & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LDLT & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei in eine leere Instanz.

`const ls_LDLT & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_LDLT & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.

6.21. `ls_LDLT_diagonal`
Faktorisierung einer symmetrischen Matrix mit Diagonalpivotisierung

Nichtöffentliche Daten/Methoden:

`ls_Matrix A, F`

Koeffizienten- und faktorisierte Matrix.

`ls_array1<ls_UINT> ind`

Indexfeld für die Diagonalindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten/Methoden:

`char *name, *A_name, *F_name, *ind_name`

Zeiger auf alle Objektnamen.

`static ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert für die Faktorisierung.

`ls_LDLT_diagonal(char* ="ls_LDLT_diagonal")`

Standard-Konstruktor.

`ls_LDLT_diagonal(ls_sMatrix &, ls_REAL =0., char* ="ls_LDLT_diagonal")`

`move`-Konstruktor. Die angegebene, symmetrische Matrix wird übernommen und eine Faktorisierung mit Diagonal-Pivotisierung berechnet, gegebenenfalls mit Regularisierung (der Regularisierungsparameter ist als positive Zahl zu übergeben). Auf `rc` wird der Fehlercode abgelegt.

`unsigned char good() const`

Liefert Signal für erfolgreiche Faktorisierung.

`ls_LDLT_diagonal(ls_LDLT_diagonal &)`

`move`-Konstruktor.

`ls_LDLT_diagonal & swap(ls_LDLT_diagonal &)`

Datenfelder-Austausch.

`const ls_LDLT_diagonal & operator= (const ls_LDLT_diagonal &)`

Zuweisungsoperator.

`const ls_LDLT_diagonal & solve(ls_Vector &, const ls_Vector &) const`

Zur im Aufruf übergebenen rechten Seite (2. Parameter) wird das entprechende lineare Gleichungssystem gelöst und die gefundene Lösung (1. Parameter) zurückgegeben.

`ls_UINT post_iteration(ls_Vector &, ls_Vector &) const`

Zur im Aufruf als 1. Parameter übergebenen Lösung und der rechten Seite wird eine Nachiteration ausgeführt; die gefundene Lösung nebst Residuum werden zurückgegeben. Der Rückkehrwert gibt die Iterationszahl an.

`ls_Vector residuum(const ls_Vector &x, const ls_Vector &b) const`

Berechnung des Residuums bei vorliegender Faktorisierung.

`ls_Vector Lx(const ls_Vector &x) const`

Multiplikation des Cholesky-Faktors mit einem Vektor.

`ls_Vector xL(const ls_Vector &x) const`

Multiplikation eines Vektors mit dem Cholesky-Faktor.

`const ls_LDLT_diagonal & write_row(ostream &) const`

Zeilenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LDLT_diagonal & read_row(istream &)`

Zeilenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LDLT_diagonal & write_column(ostream &) const`

Spaltenweises Schreiben der Objekt-Daten in die durch den Dateideskriptor bestimmte Datei.

`ls_LDLT_diagonal & read_column(istream &)`

Spaltenweises Lesen der Objekt-Daten aus der durch den Dateideskriptor bestimmten Datei.

`const ls_LDLT_diagonal & operator>> (char *) const`

Zeilenweises Schreiben der Objekt-Daten in die namentlich angegebene Datei.

`ls_LDLT_diagonal & operator<< (char *)`

Zeilenweises Lesen der Objekt-Daten aus der namentlich angegebenen Datei.