Klassen für sparse-Matrizen

Horst Hollatz

1. September 2005

Zusammenfassung

Mittels der Programmiersprache `C++` werden zunächst Klassen für eindimensionale und zweidimensionale Listen definiert, aus denen sparse-Vektoren, sparse-Rechteckmatrizen, quadratische sparse-Matrizen und symmetrische sparse-Matrizen abgeleitet sind. Die Funktionen und Operatoren dieser Klassen gestatten die Darstellung von Operationen analog zur Matrizenrechnung. Zum Gleichungslösen (bzw. Lösen linearer Ausgleichsprobleme) gibt es in Abhängigkeit von der Klasse das Lösen mittels `LU`-Zerlegung (mit/ohne Pivotisierung, Regularisierung), mittels `LDLT`-Zerlegung (mit/ohne Pivotisierung, Regularisierung) und das konjugierte Gradientenverfahren (mit/ohne Vorkonditionierung, Regularisierung). Das System kann ohne wesentliche objektorientierte Programmierkenntnisse angewendet und durch Streichen oder Hinzufügen von Funktionen spezifischen Aufgaben angepaßt werden.

1 Basisklassen

Das Klassenkonzept folgt weitgehend dem Klassenkonzept `LS` für lineare Systeme vom gleichen Autor. Der hier gegebene Text setzt die Kenntnis dieses Konzeptes voraus. Grundlage sind hier die Klassen `sp_list1<T>`, `sp_list2<T>`, `sp_listU<T>`, in denen ein- und zweidimensionale sparse-Objekte mit beliebigem Datentyp abgelegt werden können. Dabei bildet die letzte Klasse zweidimensionale, symmetrische sparse-Objekte ab. In Objekten gibt es häufig ein ausgezeichnetes Element σ; oft ist dies das Nullelement. Ein Objekt mit $\mathcal{O}(n^p)$ Elementen (p nat. Zahl) heißt **sparse** oder **schwach besetzt** von der Ordnung $r < p$ bezüglich σ, falls es nicht mehr als $\mathcal{O}(n^r)$ Nicht-σ-Elemente enthält. Ein Objekt mit $\mathcal{O}(n^p)$ Elementen soll **sparse** heißen, falls es ein Element σ enthält, zu σ ein $r < p$ existiert und das Objekt höchstens $\mathcal{O}(n^r)$ Nicht-σ-Elemente enthält. Die Nicht-σ-Elemente heißen hier NNE. So ist z. B. jede (n, n)-Matrix mit $\mathcal{O}(n)$ NNE sparse.

1.1 Eindimensionale Liste

Diese Klasse ist grundlegend und zeigt insbesondere die verwendete Ideologie.

```
#ifndef SP_LIST1
#define SP_LIST1
#include "sp_names.h"
#include LS_ARRAY1_H
#include SP_LIST_H

template <class T>
class sp_list1: public sp_list<T>
{ protected:
  list1<T> *a;           // a->i: Dimension des Feldes
                         // a->next: Zeiger b auf 1. Datenelement (oder auf a)
                         // b->i: Index des 1. Datenelementes
                         // b->x: Datenelement
```

```
                         // b->next: Zeiger auf nächstes Datenlement
                         // oder auf a
  char ONAME[ls_len];    // Feld für den Objektnamen
  void set_data(list1<T> *aa){ a=aa;}
  public:
  char *name;            // Zeiger auf den Objektnamen
  sp_list1(ls_UINT =0, char* ="sp_list1");
  sp_list1(sp_list1<T> &);
  sp_list1(T*, ls_UINT,char* ="sp_list1");
  ~sp_list1();
  sp_list1<T>& swap(sp_list1<T> &);
  operator const ls_array1<T>()const
  { ls_array1<T> v(a->i); get_array(v); return v;}
  list1<T>* asList()const{ return a;}
  ls_UINT dimension() const{ return a->i;}
  const sp_list1<T>& operator=(const sp_list1<T> &);
  sp_list1<T>& put(T, ls_UINT=0);
  sp_list1<T>& put_array(const sp_list1<T>&, ls_UINT =0);
  sp_list1<T>& put_array(const ls_array1<T>&, ls_UINT =0);
  sp_list1<T>& put_array(T, ls_UINT =0);
  T get(ls_UINT) const;
  const sp_list1<T>& get_array(sp_list1<T>&, ls_UINT =0) const;
  const sp_list1<T>& get_array(ls_array1<T>&, ls_UINT =0) const;
  sp_list1<T>& append(ls_UINT =1);
  sp_list1<T>& remove(ls_UINT);
  sp_list1<T>& remove();
  const sp_list1<T>& write(ostream &) const;
  sp_list1<T>& read(istream &);
  const sp_list1<T>& operator>> (char *) const;
  sp_list1<T>& operator<< (char *);
};
#ifndef SP_LIB
#include SP_LIST1_C
#endif
#endif
```

Bei fixiertem Datentyp `T` erhalten alle Objekte der Klasse ihre Speicherplätze aus dem gleichen Fundus, nämlich der abstrakten Klassenvorlage `sp_list<T>`:

```
#ifndef SP_LIST
#define SP_LIST
#include "sp_names.h"
#include LS_ERROR_H

template <class T>
class list1
{ public:
  T x;         // Wert
  ls_UINT i;   // Index
  list1 *next; // Zeiger auf das nächste Element
};

template <class T>
class list2
```

```
{ public:
     T x;           // Wert
     ls_UINT i,     // Zeilenindex
             j;     // Spaltenindex
     list2 *r_next, // Zeiger auf nächstes Element in Zeile
           *c_next; // Zeiger auf nächstes Element in Spalte
};
```

```
template <class T>
class sp_list
{ protected:
  static list1<T> *ff1,    // Zeiger auf Leerkette
                  *fa1;    // Zeiger auf letzten Block
  static list2<T> *ff2,    // Zeiger auf Leerkette
                  *fa2;    // Zeiger auf letzten Block
  static ls_UINT anz1,anz2;// Instanzen-Zähler
  enum{ blks = 1024 };     // Größe eines Blockes
  sp_list(ls_UINT l);
  public:
  ~sp_list();
  list1<T>* get1_el();
  void ret1_el(list1<T>*aa);
  list2<T>* get2_el();              // Anfordern Element
  list2<T>* get2_el(ls_UINT ll);
  void ret2_el(list2<T>*aa);
  void ret2_el(list2<T> *aa, ls_UINT l);
};
#include SP_LIST_C
#endif
```

Aus Laufzeitgründen erfolgt die eigentliche Speicherplatzanforderung blockweise; die sparse-Objekte fordern jedoch den Speicherplatz elementweise an und geben ihn auch elementweise zurück. Daher werden die unbenutzten Datenelemente in einer Leerliste verkettet; ein zurückgegebenes Datenelement wird an den Anfang gekettet, damit es als erstes wieder benutzt werden kann.
Instanzenzähler werden benötigt, um den angeforderten Speicherplatz an das Betriebssystem zurückzugeben, falls kein Objekt des betreffenden Typs mehr existiert.
In einer eindimensionalen Liste erhält jedes Datenelement einen nichtnegativen Index, der seinen eigentlichen Platz innerhalb des Feldes charakterisiert. Damit besteht ein Listenelement aus einem Datenelement `T x`, seinem Index `uint i` und einem Zeiger `list1 *next` auf das nächste Listenelement. Der Anker `list1 *a` hat den gleichen Aufbau; als Index ist jedoch `dim` gespeichert, um ein einfaches Durchlaufen der Liste zu ermöglichen; der Zeiger `next` zeigt auf das erste Listenelement. Der Zeiger im letzten Listenelement zeigt auf den Anker. Die Listenelemente sind mit aufsteigendem Index verkettet, so daß beim Durchlauf als letzter Index stets `dim` erscheint.
Für die Verwaltung einer eindimensionalen Liste `v` benötigt man die Eingabe des `i`-ten Datenelementes `s`: `v.put(s,i)`. Mit `s=v.get(i)` wird das `i`-te Datenelement ausgegeben. Mittels `v.remove()` wird die gesamte Liste entfernt; ein einzelnes Element wird mittels `v.remove(i)` gestrichen und die Dimension des Feldes um `1` erniedrigt.
Die Methoden-Namen sind so gewählt, daß sich ihre Funktionen weitgehend selbst erklären.

1.2 Zweidimensionale Liste

Eine zweidimensionale Liste ist das Abbild eines zweidimensionalen Feldes, wobei die Null-Elemente aus dem Urbild nicht abgespeichert sind. Die Speicherplatzverwaltung einer zweidimensionalen Liste wird in der gleichen Klasse `sp_list2<T>` durchgeführt.

```
#ifndef SP_LIST2
#define SP_LIST2
#include <string.h>
#include "sp_names.h"
#include LS_ARRAY2_H
#include SP_LIST1_H

template<class T>
class sp_list2: public sp_list<T>
{ protected:
  list2<T>  *a; // a->i=m; a->j=n; a->r_next=a->c_next=b;
                // b+i : Anker der i-ten Zeile und i-ten Spalte
                // (b+i)->r_next : 1. Datenelement Zeile i;
                // (b+i)->c_next : 1. Datenelement Spalte i;
                // (b+i)->i : Element-Anzahl der i-ten Zeile;
                // (b+i)->j : Element-Anzahl der i-ten Spalte;
                // das letzte r_next bzw. c_next zeigt auf a;
  char ONAME[ls_len];    // Feld für den Objektnamen
  void set_data(list2<T> *aa){ a=aa;}
  public:
  char *name;
  sp_list2(ls_UINT =0, ls_UINT =0, char* ="sp_list2");
  sp_list2(sp_list2<T> &);
  sp_list2(T*, ls_UINT, ls_UINT, char* ="sp_list2");
  ~sp_list2();
  sp_list2<T>& swap(sp_list2<T> &);
  ls_UINT number_of_rows() const{ return a->i; }
  ls_UINT number_of_columns() const{ return a->j;}
  list2<T>* asList()const{ return a;}
  const sp_list2<T>& operator=(const sp_list2<T> &);
  sp_list2<T>& put(T, ls_UINT, ls_UINT);
  sp_list2<T>& put_row(const sp_list1<T> &, ls_UINT,
                       ls_UINT =0);
  sp_list2<T>& put_row(const ls_array1<T> &, ls_UINT,
                       ls_UINT =0);
  sp_list2<T>& put_row(T, ls_UINT, ls_UINT =0);
  sp_list2<T>& put_column(const sp_list1<T>&, ls_UINT, ls_UINT);
  sp_list2<T>& put_column(const ls_array1<T>&, ls_UINT,
                          ls_UINT);
  sp_list2<T>& put_column(T, ls_UINT, ls_UINT);
  sp_list2<T>& put_diagonal(const sp_list1<T> &, ls_UINT =0,
                            ls_UINT =0);
  sp_list2<T>& put_diagonal(const ls_array1<T> &, ls_UINT =0,
                            ls_UINT =0);
  sp_list2<T>& put_diagonal(T, ls_UINT =0, ls_UINT =0);
  sp_list2<T>& put_array(const sp_list2<T> &, ls_UINT =0,
                         ls_UINT =0);
  sp_list2<T>& put_array(const ls_array2<T> &, ls_UINT =0,
                         ls_UINT =0);
```

```
  sp_list2<T>& put_array(T, ls_UINT =0, ls_UINT =0);
  T get(ls_UINT, ls_UINT) const;
  const sp_list2<T>& get_row(sp_list1<T> &,ls_UINT,
                             ls_UINT =0) const;
  const sp_list2<T>& get_row(ls_array1<T> &,ls_UINT,
                             ls_UINT =0) const;
  const sp_list2<T>& get_column(sp_list1<T> &,ls_UINT,
                                ls_UINT) const;
  const sp_list2<T>& get_column(ls_array1<T> &,ls_UINT,
                                ls_UINT) const;
  const sp_list2<T>& get_diagonal(sp_list1<T> &,ls_UINT =0,
                                  ls_UINT =0) const;
  const sp_list2<T>& get_diagonal(ls_array1<T> &,ls_UINT =0,
                                  ls_UINT =0) const;
  const sp_list2<T>& get_array(sp_list2<T>&, ls_UINT =0,
                               ls_UINT =0) const;
  const sp_list2<T>& get_array(ls_array2<T>&, ls_UINT =0,
                               ls_UINT =0) const;
  sp_list2<T>& append_row(ls_UINT =1);
  sp_list2<T>& append_column(ls_UINT =1);
  sp_list2<T>& append_array(ls_UINT =1, ls_UINT =1);
  sp_list2<T>& remove_row(ls_UINT);
  sp_list2<T>& remove_column(ls_UINT);
  sp_list2<T>& remove();
  sp_list2<T>& swap_row(ls_UINT, ls_UINT);
  sp_list2<T>& swap_column(ls_UINT, ls_UINT);
  const sp_list2<T>& write_row(ostream &) const;
  sp_list2<T>& read_row(istream &);
  const sp_list2<T>& write_column(ostream &) const;
  sp_list2<T>& read_column(istream &);
  const sp_list2<T>& operator>> (char *) const;
  sp_list2<T>& operator<< (char *);
  ls_REAL filling_density();
};
#ifndef SP_LIB
#include SP_LIST2_C
#endif
#endif
```

In einer zweidimensionalen Liste der Größe `(m,n)`, wo `m` die Zeilenanzahl und `n` die Spaltenanzahl darstellen, ist die Position eines Datenelementes durch zwei Indices `(i,j)` definiert: Der erste charakterisiert die Zeile, der zweite die Spalte, in der das Datenelement steht. Dabei beziehen sich diese Indices stets auf das entsprechende Urbild. Die Zeilen und Spalten sind jeweils vorwärts verkettet; in `a->r_next` und in `a->c_next` findet man den Zeiger auf das Ankerfeld `b`. Im Zeilenindex `(b+k)->i` des Ankers wird die Anzahl der verketteten Datenelemente in der `k`-ten Zeile, im Spaltenindex `(b+k)->j` die Anzahl der verketteten Datenelemente in der `k`-ten Spalte abgelegt. Der `next`-Zeiger des letzten Datenelementes einer Zeile oder Spalte zeigt auf `a`; außerdem gilt `a->i = m` und `a->j = n`. Typische Verarbeitungsprozesse einer Liste sind das zeilen- oder spaltenweise Suchen eines Elementes mit eventuellem Ein- oder Ausketten des betreffenden Elementes und das Durchlaufen einer Zeile oder Spalte. Mit der gewählten Organisation ist dies leicht möglich:

1. Gesucht sei das Datenelement aus der Zeile `i` mit dem Spaltenindex `j`:

```
list2<T> *ap=asList(), *aa; ap=ap->r_next+i;
while((aa=ap->r_next)->j < j) ap=aa;
```

```
if(aa->j == j) // vorhanden ...
```

2. Durchlaufen der vorhandenen Listenelemente der `j`-ten Spalte:

```
list2<T> *aa=asList(); aa=aa->c_next+j;
while((aa=aa->c_next) != a) ...
```

Die Ein-, Ausgabe- und Lösch-Funktionen benötigen jeweils eine Startposition, von der an sie zeilen-, spalten- oder diagonalweise arbeiten.

1.3 Obere Dreiecksstruktur

Bei symmetrischen, zweidimensionalen sparse-Objekten ist es zweckmäßig, nur das obere Dreieck abzuspeichern, wobei aber die logische Sicht hinsichtlich der Indizierung von Datenelementen so erfolgt, als ob die Datenelemente aus dem unteren Dreieck vorhanden wären. Dadurch gelingt es insbesondere, den Funktionen und Operationen die gleiche äußere Form wie in `sp_list2<T>` zu geben.

```
#ifndef SP_LISTU
#define SP_LISTU
#include "sp_names.h"
#include LS_ARRAYU_H
#include SP_LIST2_H

template<class T>
class sp_listU: public sp_list<T>
{ protected:
  list2<T> *a;// a->i=n; a->j=n; a->r_next=a->c_next=b oder a;
              // b+i Anker der i-ten Zeile und i-ten Spalte
              // (b+i)->r_next 1. Datenelement der i-ten Zeile;
              // (b+i)->c_next 1. Datenelement der i-ten Spalte;
              // (b+i)->i Element-Anzahl der i-ten Zeile;
              // (b+i)->j Element-Anzahl der i-ten Spalte;
              // das letzte r_next bzw. c_next zeigt auf a;
  char ONAME[ls_len];    // Feld für den Objektnamen
  void set_data(list2<T> *aa){ a=aa;}
  public:
  char *name;
  sp_listU(ls_UINT =0, char* ="sp_listU");
  sp_listU(sp_listU<T> &);
  ~sp_listU();
  sp_listU<T>& swap(sp_listU<T>&);
  ls_UINT dimension() const{ return a->i;}
  ls_REAL filling_density();
  const sp_listU<T>& operator=(const sp_listU<T>&);
  list2<T>* asList() const{ return a;}
  sp_listU<T>& put(T, ls_UINT, ls_UINT);
  sp_listU<T>& put_row(const sp_list1<T>&, ls_UINT, ls_UINT =0);
  sp_listU<T>& put_row(const ls_array1<T>&, ls_UINT, ls_UINT =0);
  sp_listU<T>& put_row(T, ls_UINT, ls_UINT =0);
  sp_listU<T>& put_diagonal(const sp_list1<T>&, ls_UINT, ls_UINT);
  sp_listU<T>& put_diagonal(const ls_array1<T>&, ls_UINT, ls_UINT);
  sp_listU<T>& put_diagonal(T, ls_UINT =0, ls_UINT =0);
  sp_listU<T>& put_array(const sp_listU<T>&, ls_UINT =0, ls_UINT =0);
  sp_listU<T>& put_array(const ls_arrayU<T>&, ls_UINT =0, ls_UINT =0);
```

```
  T get(ls_UINT, ls_UINT) const;
  const sp_listU<T>& get_row(sp_list1<T>&, ls_UINT, ls_UINT =0) const;
  const sp_listU<T>& get_row(ls_array1<T>&, ls_UINT, ls_UINT =0) const;
  const sp_listU<T>& get_diagonal(sp_list1<T>&, ls_UINT, ls_UINT) const;
  const sp_listU<T>& get_diagonal(ls_array1<T>&, ls_UINT, ls_UINT) const;
  const sp_listU<T>& get_array(sp_listU<T> &, ls_UINT =0, ls_UINT =0) const;
  const sp_listU<T>& get_array(ls_arrayU<T>&, ls_UINT =0, ls_UINT =0) const;
  sp_listU<T>& append_row(ls_UINT =1);
  sp_listU<T>& swap_row(ls_UINT, ls_UINT);
  sp_listU<T>& remove_row(ls_UINT);
  sp_listU<T>& remove();
  const sp_listU<T>& write_row(ostream &) const ;
  sp_listU<T>& read_row(istream &);
  const sp_listU<T>& write_column(ostream &) const ;
  sp_listU<T>& read_column(istream &);
  const sp_listU<T>& operator>> (char *) const ;
  sp_listU<T>& operator<< (char *);
};
#ifndef SP_LIB
#include SP_LISTU_C
#endif
#endif
```

Die Analogie zur zweidimensionalen Liste ist nicht vollständig: Insbesondere stimmt die `i`-te Zeile mit der `i`-ten Spalte überein; folglich braucht man nur eine zeilenorientierte Ein- und Ausgabe.

2 Die Klasse für sparse-Vectoren

In dieser Klasse sind sparse-Vektoren abgelegt. Dabei werden nur die NNE gespeichert. Es gibt analoge Operationen und Funktionen wie in der Klasse `ls_Vector`.

```
#ifndef SP_VECTOR
#define SP_VECTOR
#include "sp_names.h"
#include SP_LIST1_H

ls_REAL ausl(ls_REAL,ls_REAL);

class sp_Vector: public sp_list1<ls_REAL>
{ public:
  ls_REAL eps;
  sp_Vector(ls_UINT =0,char* ="sp_Vector");
  sp_Vector(sp_Vector&);
  sp_Vector(ls_REAL*, ls_UINT ,char* ="sp_Vector");
  const sp_Vector& operator=(const sp_Vector &);
  sp_Vector operator-() const;                 // - x
  sp_Vector operator*(ls_REAL) const;          // s * x
  sp_Vector& operator*=(ls_REAL);              // x=x * s
  sp_Vector operator+(const sp_Vector &) const;// x + y
  sp_Vector operator-(const sp_Vector &) const;// x - y
  sp_Vector& operator+=(const sp_Vector &);    // x=x + y
  sp_Vector& operator-=(const sp_Vector &);    // x=x - y
  ls_REAL operator*(const sp_Vector &) const;  // x*y Skalarprodukt
```

```
};
sp_Vector operator*(ls_REAL, const sp_Vector &);
#ifndef SP_LIB
#include SP_VECTOR_C
#endif
#endif
```

Für `ls_REAL` darf `float` oder `double` gesetzt werden. Diese Klassen gestatten nun die programmtechnische Darstellung von Vektoroperationen analog zur linearen Algebra. Dazu ein kleines Beispiel. Bildung einer Konvexkombination zweier Vektoren, die zunächst auf die Länge `1` zu normieren sind.

```
sp_Vector konv_vector(const sp_Vector &x, const sp_Vector &y, ls_REAL s)
{ return x*(s/sqrt(x*x)) +y*((1-s)/sqrt(y*y));}
```

3 Die Klasse für sparse-Matrizen

Rechteck-Matrizen werden wie Datentypen deklariert: `...; sp_Matrix A(m,n); ...`. Dabei sind `m` die Zeilen- und `n` die Spaltenanzahl der Matrix `A` mit `m` $\geqq$ `n`, die erst zur Laufzeit ihre Werte erhalten (dynamische Klassen-Komponenten). An den Operationen und Funktionen sind nur die NNE beteiligt. Zur Ein- und Ausgabe dienen die Funktionen `read_row` und `write_row`:

```
...; sp_Matrix A(3,2); ...
ofstream fo("bb"); A.write_row(fo); ...
ifstream fi("aa"); A.read_row(fi); ....
```

Hier wird die Matrix `A` mit ihrem Objektnamen `sp_Matrix` in die Datei `bb` geschrieben; sodann wird in der Datei `aa` das Wort `sp_Matrix` gesucht; danach muß zeilenweise die einzulesende Matrix als in geschweiften Klammern eingeschlossener Block folgen. Der entsprechende Block für die `(6,5)`-Matrix

$$\begin{pmatrix} 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & -2 & 1 & 0 \\ -3 & 2 & 0 & 0 & 4 \\ 0 & -4 & 1 & -5 & 2 \\ -6 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 \end{pmatrix}$$

hat den folgenden Aufbau:

```
{ number_of_rows: 6  number_of_columns: 5
  row: 0  number_of_elements: 2
       0  1  2 -1
  row: 1  number_of_elements: 2
       2 -2  3  1
  row: 2  number_of_elements: 3
       0 -3  1  2  4  4
  row: 3  number_of_elements: 4
       1 -4  2  1  3 -5  4  2
  row: 4  number_of_elements: 2
       0 -6  1  2
  row: 5  number_of_elements: 2
       3 -1  4 -1
}
```

Die Matrixelemente sind zeilenweise geschrieben mit Zeilennummer und NNE-Anzahl in der Zeile; danach folgen die NNE mit Spaltenindex und Wert. Zwischen den Daten sind die üblichen Trennzeichen erlaubt: Leerzeichen, Tabulatorzeichen, neue-Zeile-Zeichen. Natürlich gibt es auch die spaltenweise Version (`read_column`, `write_column`). Falls man keine Ansprüche an die Lese- oder Schreibeinstellung der Datei hat, darf man für Schreiben `v >> "bb"` und für Lesen `v << "aa"` verwenden; hier wird die Zeilen-Version aufgerufen. Alle Funktionen und Operationen, die nicht von der Tatsache Gebrauch machen, daß es sich bei den Daten um reelle Zahlen handelt, werden von der Klasse `sp_list2<ls_REAL>` geerbt; hierin steht `ls_REAL` für `float` oder `double`. Weitere Operationen und Funktionen sind die folgenden: **Multiplikation einer sparse-Matrix mit einem Vektor:** `x = A*y`.
Multiplikation einer transponierten sparse-Matrix mit einem Vektor: `x = y*A`.

Bei diesen Operationen müssen beide Vektoren entweder normale Vektoren oder sparse-Vektoren sein.

```
#ifndef SP_MATRIX
#define SP_MATRIX
#include "sp_names.h"
#include LS_MATRIX_H
#include SP_VECTOR_H
#include SP_LIST2_H

class sp_Matrix: public sp_list2<ls_REAL>
{ public:
  sp_Matrix(ls_UINT =0, ls_UINT =0, char* ="sp_Matrix");
  sp_Matrix(ls_REAL*, ls_UINT, ls_UINT, char* ="sp_Matrix");
  sp_Matrix(sp_Matrix &);
  ls_Vector row(ls_UINT i)const
  { ls_Vector u(a->j); get_row(u,i,0); return u;}
  ls_Vector column(ls_UINT j)const
  { ls_Vector u(a->i); get_column(u,0,j); return u;}
  ls_Vector upper_diagonal(ls_UINT j) const
  { ls_UINT n=(a->i<a->j)?a->i:a->j; ls_Vector u(n);
    get_diagonal(u,0,j); return u;}
  ls_Vector lower_diagonal(ls_UINT i)const
  { ls_UINT n=(a->i<a->j)?a->i:a->j; ls_Vector u(n);
    get_diagonal(u,i,0); return u;}
  const sp_Matrix& operator=(const sp_Matrix &);
  sp_Matrix operator+(const sp_Matrix &)const;   //A+B
  sp_Matrix operator-(const sp_Matrix &)const;   //A-B
  sp_Matrix& operator+=(const sp_Matrix &);      //A=A+B
  sp_Matrix& operator-=(const sp_Matrix &);      //A=A-B
  sp_Matrix operator*(const sp_Matrix &)const;   //A*B
  sp_Vector operator* (const sp_Vector &)const;  //A*x
  ls_Vector operator* (const ls_Vector &)const;  //A*x
  ls_UINT solve(ls_Vector&, ls_Vector&) const; //cg-Verfahren
};
sp_Vector operator* (const sp_Vector &, const sp_Matrix &);
ls_Vector operator* (const ls_Vector &, const sp_Matrix &);
sp_Matrix operator*(ls_REAL, const sp_Matrix &);  // s*A
#ifndef SP_LIB
#include SP_MATRIX_C
#endif
#endif
```

In jeder Matrix-Klasse gibt es zum Lösen eines linearen Gleichungssystems bzw. Lösen eines linearen Ausgleichsproblems die Methode `solve`, die das konjugierte Gradienteverfahren verwendet. In Abhängigkeit von

der speziellen Struktur der Koeffizientenmatrix kann man verschiedene Faktorisierungen verwenden. Sofern eine Faktorisierung ohne Pivotisierung durchführbar ist, sollte man diese nutzen.
`LU`**-Faktorisierung ohne Pivotisierung**:

```
#ifndef SP_LU
#define SP_LU
#include "sp_names.h"
#include SP_MATRIX_H

class sp_LU
{ protected:
  sp_Matrix A, F;
  char ONAME[ls_len];    // Feld für den Objektnamen
  public:
  char *name, *A_name, *F_name;
  ls_REAL eps;
  int rc;
  sp_LU(char* ="sp_LU");
  sp_LU(sp_Matrix &, ls_REAL =0., char* ="sp_LU");
  sp_LU(sp_LU &);
  unsigned char good()const{ return !rc;}
  sp_LU& swap(sp_LU &);
  const sp_LU& operator=(const sp_LU &);
  const sp_LU& solve(ls_Vector &, const ls_Vector &) const;
  ls_UINT post_iteration(ls_Vector &, ls_Vector &) const;
  ls_Vector residuum(const ls_Vector &, const ls_Vector &) const;
  const sp_LU& write_row(ostream &) const;
  sp_LU& read_row(istream &);
  const sp_LU& write_column(ostream &) const;
  sp_LU& read_column(istream &);
  const sp_LU& operator>> (char *) const;
  sp_LU& operator<< (char *);
};
#ifndef SP_LIB
#include SP_LU_C
#endif
#endif
```

Es empfiehlt sich gelegentlich, eine Nachiteration zu versuchen. Dabei ist zu beachten, daß im Falle der Konvergenz die Lösung der im Rechner befindlichen Aufgabe angenähert wird.
Zusätzlich sind 3 weitere Varianten implementiert:
- `LU`-Faktorisierung mit fiktiv skalierter Zeilen-Pivotisierung (Klasse `sp_LU_row`),
- `LU`-Faktorisierung mit fiktiv skalierter Spalten-Pivotisierung (Klasse `sp_LU_column`),
- `LU`-Faktorisierung mit Diagonal-Pivotisierung (Klasse `sp_LU_diagonal`).

Bei schlechtkonditionierten Matrizen sollte man die Regularisierung anwenden. Der Regularisierungsparameter ist bei der Instanziierung der Faktorisierung anzugeben; als Empfehlung könnte $\alpha \in$ `(1.e-9, 1.e-12)` gelten.

4 Die Klasse für symmetrische sparse-Matrizen

In dieser Klasse sind symmetrische sparse-Matrizen abgelegt; es ist nur das obere Dreieck gespeichert; sie ist von der Klasse `sp_listU<float>` bzw. `sp_listU<double>` abgeleitet. Eine symmetrische `(n,n)`-

Matrix wird durch `sp_sMatrix A(n)` deklariert. Der Klassenaufbau folgt dem der Klasse `sp_Matrix`.

```
#ifndef SP_SMATRIX
#define SP_SMATRIX
#include "sp_names.h"
#include SP_MATRIX_H
#include LS_SMATRIX_H
#include SP_LISTU_H

class sp_sMatrix: public sp_listU<ls_REAL>
{ public:
  sp_sMatrix(ls_UINT n=0,char *nam="sp_sMatrix"):
  sp_listU<ls_REAL>(n,nam){}
  sp_sMatrix(sp_sMatrix &B):sp_listU<ls_REAL>(B){}
  ls_Vector row(ls_UINT i)const
  { ls_Vector u(a->i); get_row(u,i); return u;}
  ls_Vector diagonal(ls_UINT i)const
  { ls_Vector u(a->i); get_diagonal(u,0,i); return u;}
  const sp_sMatrix& operator=(const sp_sMatrix &);
  sp_sMatrix operator+(const sp_sMatrix &)const;   //A+B
  sp_sMatrix operator-(const sp_sMatrix &)const;   //A-B
  sp_sMatrix& operator+=(const sp_sMatrix &);      //A=A+B
  sp_sMatrix& operator-=(const sp_sMatrix &);      //A=A-B
  sp_Matrix operator*(const sp_sMatrix &)const;    //A*B
  sp_Vector operator* (const sp_Vector &) const;
  ls_Vector operator* (const ls_Vector &) const;
  ls_UINT solve(ls_Vector &, ls_Vector &) const;
};
#ifndef SP_LIB
#include SP_SMATRIX_C
#endif
#endif
```

Als Lösungsmethoden gibt es Klassen für
- das vorkonditionierte konjugierte Gradientenverfahren (`sp_sCG_precond`),
- die LDLT-Faktorisierung (`sp_LDLT()`,
- die LDLT-Faktorisierung mit Diagonal-Pivotisierung (`LDLT_diagonal()`).
Beispielhaft sei hier eine Klasse notiert.

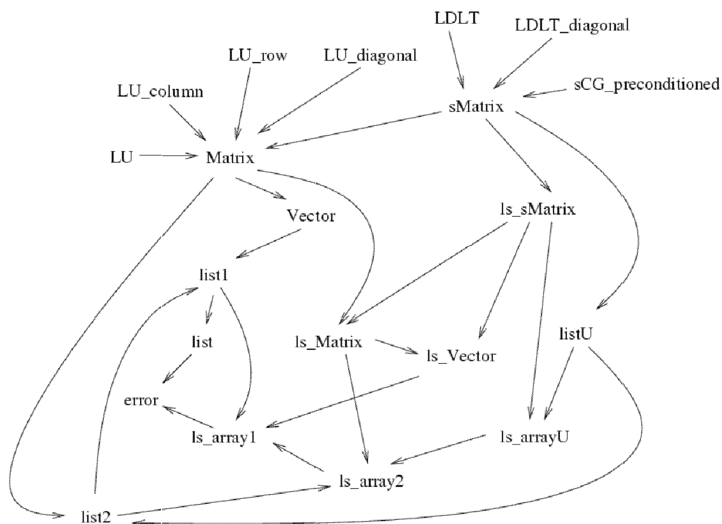
```
#ifndef SP_LDLT
#define SP_LDLT
#include "sp_names.h"
#include SP_SMATRIX_H

class sp_LDLT
{ protected:
  sp_sMatrix A, F;
  char ONAME[ls_len];    // Feld für den Objektnamen
  public:
  char *name, *A_name, *F_name;
  ls_REAL eps;
  int rc;
  sp_LDLT(char* ="sp_LDLT");
  sp_LDLT(sp_sMatrix &, ls_REAL =0., char* ="sp_LDLT");
```

```
  sp_LDLT(sp_LDLT &);
  unsigned char good()const{ return !rc;}
  sp_LDLT& swap(sp_LDLT &);
  const sp_LDLT& operator=(const sp_LDLT &);
  const sp_LDLT& solve(ls_Vector &, const ls_Vector &) const;
  ls_UINT post_iteration(ls_Vector &, ls_Vector &) const;
  ls_Vector residuum(const ls_Vector &, const ls_Vector &) const;
  const sp_LDLT& write_row(ostream &) const;
  sp_LDLT& read_row(istream &);
  const sp_LDLT& write_column(ostream &) const;
  sp_LDLT& read_column(istream &);
  const sp_LDLT& operator>> (char *) const;
  sp_LDLT& operator<< (char *);
};
#ifndef SP_LIB
#include SP_LDLT_C
#endif
#endif
```

Hinsichtlich einer schlechten Kondition der Koeffizientematrix gilt analoges zu oben.

5 Allgemeine Hinweise

Hinsichtlich der Anwendung dieser Klassen wurde das bei den `LS`-Klassen dokumentierte Konzept realisiert. Das folgende Bild zeigt den Nachladegraphen.



Eventuelle Fehler werden durch den `try-catch`-Mechanismus aufgefangen:

`main(){... try{ ...;}catch(...){};...}.`

Hier werden jene Fehler aufgefangen, die innerhalb des `try`-Blocks auftreten.

6 Programmdokumentation

6.1 Datentyp `list1<T>`

Öffentliche Daten:

`T x`

Wert des Datenelementes.

`ls_UINT i`

Wahrer Index des Datenlementes.

`list1<T>* next`

Zeiger auf das nächste Datenelement.

6.2 Datentyp list2<T>

Öffentliche Daten:

`T x`

Wert des Datenelementes.

`ls_UINT i, j`

Wahrer Zeilen- und Spaltenindex des Datenlementes.

`list2<T>* r_next, c_next`

Zeiger auf das nächste Datenelement in der Zeile, Spalte.

6.3 Abstrakte Klassen-Vorlage `sp_list<T>` Speicherplatzverwaltung

Nichtöffentliche Daten:

`static list1<T> *ff1`

Zeiger auf Leerkette

`static list1<T> *fa1`

Zeiger auf letzten Anforderungsblock

`static ls_UINT anz1, anz2`

Instanzen-Zähler

`enum{ blks = 1024 }`

Größe eines Anforderungsblockes

`static list2<T> *ff2`

Zeiger auf Leerkette.

`static list2<T> *fa2`

Zeiger auf den letzten Anforderungsblock.

Öffentliche Daten/Methoden:

`list1<T>* get1_el()`

Speicherplatz für ein Datenelement anfordern.

`void ret1_el(list1<T>*aa)`

Freigeben eines Speicherplatzes.

`list2<T> *get2_el()`

Liefert Speicherplatz für ein Datenelement.

`list2<T> *get2_el(ls_UINT l)`

Liefert Speicherplatz für `l` Datenelemente.

`void ret2_el(list2<T>*aa)`

Gibt den Speicherplatz für ein Datenelement zurück.

`void ret2_el(list2<T>*aa, ls_UINT l)`

Gibt den Speicherplatz für ein Datenelementefeld zurück.

6.4 Die Klassen-Vorlage `sp_list1<T>` eindimensionales sparse-Feld beliebigen Typs `T`

Abgeleitet von: `sp_list<T>`

Nichtöffentliche Daten:

`list1<T> *a`

Datenfeld mit folgendem Aufbau:

`a->i`: Dimension des Feldes
`a->next`: Zeiger `b` auf 1. Datenelement (oder auf `a`)
`b->i`: Index des 1. Datenelementes
`b->x`: Datenelement
`b->next`: Zeiger auf nächstes Datenelement oder auf `a`

`char ONAME[ls_len]`

Feld für den Objektnamen.

`void set_data(list1<T>*)`

Daten setzen.

Öffentliche Daten/Methoden:

`char* name`

Zeiger auf den Objektnamen.

```
sp_list1(ls_UINT =0, char* ="sp_list1")
```

Standardkonstruktor.

```
sp_list1(sp_list1<T>&)
```

move-Konstruktor.

```
sp_list1(T*, ls_UINT, char* ="sp_list1")
```

Instanziierung mit einem Datenfeld.

```
sp_list1<T>& swap(sp_list1<T> &)
```

Datentausch zwischen zwei Objekten.

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

Konvertierung eines sparse-Feldes in ein eindimensionales Datenfeld.

```
list1<T>* asList()const
```

Liefert den Zeiger auf die dynamische Komponente.

```
ls_UINT dimension() const
```

Liefert die Dimension des sparse-Feldes.

```
const sp_list1<T>& operator=(const sp_list1<T> &)
```

Zuweisungsoperator.

```
sp_list1<T>& put(T, ls_UINT=0)
```

Setzen eines Datenelementes.

```
sp_list1<T>& put_array(const sp_list1<T>&, ls_UINT =0)
```

Einspeichern eines sparse-Feldes ab der angegebenen Position.

```
sp_list1<T>& put_array(const ls_array1<T>&, ls_UINT =0)
```

Einspeichern eines eindimensionalen Feldes ab der angegebenen Position.

```
sp_list1<T>& put_array(T, ls_UINT =0)
```

Ab der angegebenen Position erhalten alle Datenelemente den angegebenen Wert.

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

Liefert den Wert des sparse-Feldes auf der angegebenen Position.

```
const sp_list1<T>& get_array(sp_list1<T>&, ls_UINT =0) const
```

Datenausgabe. Das Programm ist die Umkehrung der Methode

```
put_array(const sp_list1<T>&, ls_UINT =0)
```

`const sp_list1<T>& get_array(ls_array1<T>&, ls_UINT =0) const`

Datenausgabe. Das Programm ist die Umkehrung der Methode
`put_array(const ls_array1<T>&, ls_UINT =0)`

`sp_list1<T>& append(ls_UINT l=1)`

Anhängen von `l` Nullelementen.

`sp_list1<T>& remove(ls_UINT)`

Entfernen des angegebenen Elementes.

`sp_list1<T>& remove()`

Entfernen des gesamten sparse-Feldes.

`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 `a->i`,
Schlüsselwort `number_of_elements:`, Datenelemente in der Form `Index Zahl`.

`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 Aufparameter spezifizierte Datei; die Ausgabe erfolgt durch den Aufruf der Methode `write(ostream&)`.

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

Dateneingabe aus der durch den Aufparameter spezifizierten Datei; die Eingabe erfolgt durch den Aufruf der Methode `read(istream&)`.

6.5 Die Klassen-Vorlage `sp_list2<T>` zweidimensionales sparse-Feld beliebigen Typs `T`

Abgeleitet von: `sp_list<T>`

Nichtöffentliche Daten:
`list2<T> *a`

Zeiger auf sparse-Liste des sparse-Feldes mit folgendem Aufbau:

a->i: Zeilenanzahl,
a->j: Spaltenanzahl,
a->r_next: Anker b für das Zeilenfeld,
a->r_next: Anker b für das Spaltenfeld (a->r_next=a->c_next.

b+i: Anker der i-ten Zeile und i-ten Spalte,
(b+i)->r_next: 1. Datenelement der i-ten Zeile,
(b+i)->c_next: 1. Datenelement der i-ten Spalte,
(b+i)->i: NNE-Anzahl der i-ten Zeile,
(b+i)->j: NNE-Anzahl der i-ten Spalte,
das letzte r_next bzw. c_next zeigt auf a.

```
char ONAME[ls_len]
```

Feld für den Objektnamen.

```
void set_data(list2<T> *aa)
```

Daten neu setzen.

```
char* name
```

Zeiger auf den Objektnamen.

```
sp_list2(ls_UINT mm=0, ls_UINT nn=0, char* ="sp_list2")
```

Standardkonstruktor; es wird ein zweidimensionales sparse-Feld mit mm Zeilen und nn Spalten instanziiert.

```
sp_list2(sp_list2<T> &)
```

move-Konstruktor; die Objekt-Daten werden übernommen und aus der Quelle entfernt.

```
sp_list2(T*, ls_UINT, ls_UINT, char* ="sp_list2")
```

Konstruktor, der das angegebene zweidimensionale Feld in ein sparse-Feld konvertiert.

```
sp_list2<T>& swap(sp_list2<T> &)
```

Austausch der Objekt-Daten.

```
ls_UINT number_of_rows() const
```

Liefert die Zeilenanzahl.

```
ls_UINT number_of_columns() const
```

Liefert die Spaltenanzahl.

```
list2<T>* asList()const
```

Liefert den Anker der dynamischen Komponente.

```
const sp_list2<T>& operator=(const sp_list2<T> &)
```

Zuweisungsoperator; Objektnamen bleiben erhalten.

sp_list2<T>& put(T, ls_UINT, ls_UINT)

Einspeichern eines Datenelementes an die angegebene Position.

sp_list2<T>& put_row(const sp_list1<T> &, ls_UINT, ls_UINT =0)

Einspeichern eines eindimensionalen sparse-Feldes als Zeile ab der angegebenen Position.

sp_list2<T>& put_row(const ls_array1<T> &, ls_UINT, ls_UINT =0)

Einspeichern eines eindimensionalen Feldes als Zeile ab der angegebenen Position.

sp_list2<T>& put_row(T, ls_UINT, ls_UINT =0)

Einspeichern eines Wertes als Zeile ab der angegebenen Position.

sp_list2<T>& put_column(const sp_list1<T>&, ls_UINT, ls_UINT)

Einspeichern eines eindimensionalen sparse-Feldes als Spalte ab der angegebenen Position.

sp_list2<T>& put_column(const ls_array1<T>&, ls_UINT, ls_UINT)

Einspeichern eines eindimensionalen Feldes als Spalte ab der angegebenen Position.

sp_list2<T>& put_column(T, ls_UINT, ls_UINT)

Einspeichern eines Wertes als Spalte ab der angegebenen Position.

sp_list2<T>& put_diagonal(const sp_list1<T> &,ls_UINT =0,ls_UINT =0)

Einspeichern eines eindimensionalen sparse-Feldes als Diagonale ab der angegebenen Position.

sp_list2<T>& put_diagonal(const ls_array1<T>&,ls_UINT =0,ls_UINT =0)

Einspeichern eines eindimensionalen Feldes als Diagonale ab der angegebenen Position.

sp_list2<T>& put_diagonal(T, ls_UINT =0, ls_UINT =0)

Einspeichern eines Wertes als Diagonale ab der angegebenen Position.

sp_list2<T>& put_array(const sp_list2<T> &, ls_UINT =0, ls_UINT =0)

Einspeichern eines zweidimensionalen sparse-Feldes ab der angegebenen Position.

sp_list2<T>& put_array(const ls_array2<T> &, ls_UINT =0, ls_UINT =0)

Einspeichern eines zweidimensionalen Feldes ab der angegebenen Position.

sp_list2<T>& put_array(T, ls_UINT =0, ls_UINT =0)

Einspeichern eines Wertes als zweidimensionales Feld ab der angegebenen Position.

T get(ls_UINT, ls_UINT) const

Liefert den Wert, der durch die angegebenen Daten (Zeile, Spalte) indiziert ist.

sp_list2<T>& get_row(sp_list1<T> &,ls_UINT, ls_UINT =0) const

Liefert die indizierte Zeile als eindimensionales sparse-Feld.

```
sp_list2<T>& get_row(ls_array1<T> &,ls_UINT, ls_UINT =0) const
```

Liefert die indizierte Zeile als eindimensionales Feld.

```
sp_list2<T>& get_column(sp_list1<T> &,ls_UINT, ls_UINT) const
```

Liefert die indizierte Spalte als eindimensionales sparse-Feld.

```
sp_list2<T>& get_column(ls_array1<T> &,ls_UINT, ls_UINT) const
```

Liefert die indizierte Spalte als eindimensionales Feld.

```
sp_list2<T>& get_diagonal(sp_list1<T>&,ls_UINT =0,ls_UINT =0)const
```

Liefert die indizierte Diagonale als eindimensionales sparse-Feld.

```
sp_list2<T>& get_diagonal(ls_array1<T>&,ls_UINT =0,ls_UINT =0)const
```

Liefert die indizierte Diagonale als eindimensionales Feld.

```
sp_list2<T>& get_array(sp_list2<T>&,ls_UINT =0,ls_UINT =0)const
```

Liefert das indizierte Teilfeld als zweidimensionales sparse-Feld.

```
sp_list2<T>& get_array(ls_array2<T>&,ls_UINT =0,ls_UINT =0)const
```

Liefert das indizierte Teilfeld als zweidimensionales Feld.

```
sp_list2<T>& append_row(ls_UINT l=1)
```

Es werden `l` Nullzeilen angehängt.

```
sp_list2<T>& append_column(ls_UINT l=1)
```

Es werden `l` Nullspalten angehängt.

```
sp_list2<T>& append_array(ls_UINT l=1, ls_UINT k=1)
```

Es werden `l` Nullzeilen und `k` Nullspalten angehängt.

```
sp_list2<T>& remove_row(ls_UINT)
```

Streichen der indizierten Zeile.

```
sp_list2<T>& remove_column(ls_UINT)
```

Streichen der indizierten Spalte.

```
sp_list2<T>& remove()
```

Streichen der gesamten dynamischen Komponente.

```
sp_list2<T>& swap_row(ls_UINT, ls_UINT)
```

Zeilenaustausch.

```
sp_list2<T>& swap_column(ls_UINT, ls_UINT)
```

Spaltenaustausch.

```
const sp_list2<T>& write_row(ostream &) const
```

Die Objekt-Daten werden in standardisierter Form in die durch den angegebenen Dateideskriptor definierte Datei in folgender Reihenfolge geschrieben:

Objektname,
geschweifte Klammer auf,
Schlüsselwort `number_of_rows:`, Zeilenzahl,
Schlüsselwort `number_of_columns:`, Spaltenzahl,
Schlüsselwort `row:`, Zeilennummer,
Schlüsselwort `number_of_elements:`, Anzahl,
Spaltenindex, Zahl, ... geschweifte Klammer zu.

Die Zeilen müssen lückenlos, in aufsteigender Reihenfolge, mit dem Index `0` beginnend erscheinen. Innerhalb einer Zeile müssen die Spaltenindices in aufsteigender Reihenfolge auftreten; die erste Spalte hat den Index `0`.

```
sp_list2<T>& read_row(istream &)
```

Die Objekt-Daten werden zeilenweise aus der durch den angegebenen Dateideskriptor definierte Datei gelesen. Die Daten müssen in der durch das zeilenweise Schreiben definierten Reihenfolge stehen. Das Zielobjekt muß leer sein; falls es einen Objektnamen hat, wird dieser in der Datei gesucht; andernfalls gilt das erste gelesene Wort als Objektname. Die Daten sind durch übliche Trennzeichen zu trennen.

```
const sp_list2<T>& write_column(ostream &) const
```

Spaltenweises Schreiben in die angegebene Datei.
Es wird analog zur Methode `write_row` verfahren: Die Kopfdaten stehen in der Reihenfolge Spaltenzahl, Zeilenzahl; danach folgen die Spalten in der Form `column:` Spaltenindex, `number_of_elements:`, Anzahl, Zeilenindex, Zahl, ...

```
sp_list2<T>& read_column(istream &)
```

Die Objekt-Daten werden spaltenweise aus der durch den angegebenen Dateideskriptor definierte Datei gelesen. Die Daten müssen in der durch das spaltenweise Schreiben definierten Reihenfolge stehen. Das Zielobjekt muß leer sein; falls es einen Objektnamen hat, wird dieser in der Datei gesucht; andernfalls gilt das erste gelesene Wort als Objektname. Die Daten sind durch übliche Trennzeichen zu trennen.

```
const sp_list2<T>& operator>> (char *) const
```

Zeilenweises Schreiben in die angegebene Datei; es wird die Methode `write_row` aufgerufen. Falls als Dateiname das Leerwort angegeben ist, wird in die Standardausgabe geschrieben.

```
sp_list2<T>& operator<< (char *)
```

Zeilenweises Lesen aus der angegebenen Datei; es wird die Methode `read_row` aufgerufen. Falls als Dateiname das Leerwort angegeben ist, wird von der Standardeingabe gelesen.

```
ls_REAL filling_density()
```

Liefert die Besetzungsdichte des sparse-Feldes.

6.6 Die Klassen-Vorlage `sp_listU<T>` symmetrisches, oberes sparse-Dreiecksfeld beliebigen Typs `T`

Abgeleitet von: `sp_list<T>`

Nichtöffentliche Daten:

`static list2<T> *ff`

Zeiger auf Leerkette.

`static list2<T> *fa`

Zeiger auf den letzten Anforderungsblock.

`static ls_UINT anz`

Instanzen-Zähler.

`enum{ blks = 1024}`

Größe eines Anforderungsblockes.

`list2<T> *a`

Zeiger auf sparse-Liste des sparse-Feldes mit folgendem Aufbau:

`a->i`: Zeilenanzahl,
`a->j`: Spaltenanzahl (=Zeilenanzahl),
`a->r_next`: Anker `b` für das Zeilenfeld,
`a->r_next`: Anker `b` für das Spaltenfeld (`a->r_next=a->c_next`.

`b+i`: Anker der `i`-ten Zeile und `i`-ten Spalte,
`(b+i)->r_next`: 1. Datenelement der `i`-ten Zeile,
`(b+i)->c_next`: 1. Datenelement der `i`-ten Spalte,
`(b+i)->i`: NNE-Anzahl der `i`-ten Zeile,
`(b+i)->j`: NNE-Anzahl der `i`-ten Spalte,
das letzte `r_next` bzw. `c_next` zeigt auf `a`.

`char ONAME[ls_len]`

Feld für den Objektnamen.

`void set_data(list2<T> *aa)`

Daten neu setzen.

`char* name`

Zeiger auf den Objektnamen.

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

Standardkonstruktor; es wird ein zweidimensionales sparse-Feld mit `mm` Zeilen und `nn` Spalten instanziert.

`sp_listU(sp_listU<T> &)`

move-Konstruktor; die Objekt-Daten werden übernommen und aus der Quelle entfernt.

```
list2<T> *get_el()
```

Liefert Speicherplatz für ein Datenelement.

```
list2<T> *get_el(ls_UINT l)
```

Liefert Speicherplatz für l Datenelemente.

```
void ret_el(list2<T>*aa)
```

Gibt den Speicherplatz für ein Datenelement zurück.

```
void ret_el(list2<T>*aa, ls_UINT l)
```

Gibt den Speicherplatz für ein Datenelementefeld zurück.

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

Austausch der Objekt-Daten.

```
ls_UINT number_of_rows() const
```

Liefert die Zeilenanzahl.

```
ls_UINT number_of_columns() const
```

Liefert die Spaltenanzahl.

```
list2<T>* asList()const
```

Liefert den Anker der dynamischen Komponente.

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

Zuweisungsoperator; Objektnamen bleiben erhalten.

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

Einspeichern eines Datenelementes an die angegebene Position; der Zeilenindex darf nicht größer als der Spaltenindex sein.

```
sp_listU<T>& put_row(const sp_list1<T> &, ls_UINT, ls_UINT =0)
```

Einspeichern eines eindimensionalen sparse-Feldes als Zeile ab der angegebenen Position. Es wird der Teil bis zum Diagonalelement als Spalte und der Rest als Zeile eingespeichert.

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

Einspeichern eines eindimensionalen Feldes als Zeile ab der angegebenen Position.

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

Einspeichern eines Wertes als Zeile ab der angegebenen Position.

```
sp_listU<T>& put_diagonal(const sp_list1<T> &,ls_UINT =0,ls_UINT =0)
```

Einspeichern eines eindimensionalen sparse-Feldes als Diagonale ab der angegebenen Position.

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

Einspeichern eines eindimensionalen Feldes als Diagonale ab der angegebenen Position.

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

Einspeichern eines Wertes als Diagonale ab der angegebenen Position.

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

Einspeichern eines zweidimensionalen sparse-Dreiecksfeldes ab der angegebenen Position.

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

Einspeichern eines zweidimensionalen Dreiecksfeldes ab der angegebenen Position.

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

Einspeichern eines Wertes als zweidimensionales Dreiecksfeld ab der angegebenen Position.

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

Liefert den Wert, der durch die angegebenen Daten (Zeile, Spalte) indiziert ist.

```
sp_listU<T>& get_row(sp_list1<T> &,ls_UINT, ls_UINT =0) const
```

Liefert die indizierte Zeile als eindimensionales sparse-Feld.

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

Liefert die indizierte Zeile als eindimensionales Feld.

```
sp_listU<T>& get_diagonal(sp_list1<T>&,ls_UINT =0,ls_UINT =0)const
```

Liefert die indizierte Diagonale als eindimensionales sparse-Feld.

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

Liefert die indizierte Diagonale als eindimensionales Feld.

```
sp_listU<T>& get_array(sp_listU<T>&,ls_UINT =0,ls_UINT =0)const
```

Liefert das indizierte Teilfeld als zweidimensionales sparse-Dreiecksfeld.

```
sp_listU<T>& get_array(ls_arrayU<T>&,ls_UINT =0,ls_UINT =0)const
```

Liefert das indizierte Teilfeld als zweidimensionales Dreiecksfeld.

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

Es werden `l` Nullzeilen und Spalten angehängt.

```
sp_listU<T>& remove_row(ls_UINT)
```

Streichen der indizierten Zeile und Spalte.

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

Streichen der gesamten dynamischen Komponente.

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

Zeilen- und Spaltenaustausch.

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

Die explizit vorhandenen Objekt-Daten werden in standardisierter Form in die durch den angegebenen Dateideskriptor definierte Datei in folgender Reihenfolge geschrieben:

Objektname,
geschweifte Klammer auf,
Schlüsselwort `dimension:`, Zeilenzahl,
Schlüsselwort `row:`, Zeilennummer,
Schlüsselwort `number_of_elements:`, Anzahl,
Spaltenindex, Zahl, ...

Die Zeilen müssen lückenlos, in aufsteigender Reihenfolge, mit dem Index `0` beginnend erscheinen. Innerhalb einer Zeile müssen die Spaltenindices in aufsteigender Reihenfolge auftreten; die erste Spalte hat den Index `0`.

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

Die Objekt-Daten werden zeilenweise aus der durch den angegebenen Dateideskriptor definierte Datei gelesen. Die Daten müssen in der durch das zeilenweise Schreiben definierten Reihenfolge stehen. Das Zielobjekt muß leer sein; falls es einen Objektnamen hat, wird dieser in der Datei gesucht; andernfalls gilt das erste gelesene Wort als Objektname. Die Daten sind durch übliche Trennzeichen zu trennen.

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

Spaltenweises Schreiben in die angegebene Datei.
Es wird analog zur Methode `write_row` verfahren:
Die Spalten stehen in der Form `column:` Spaltenindex, `number_of_elements:`, Anzahl, Zeilenindex, Zahl, ...

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

Die Objekt-Daten werden spaltenweise aus der durch den angegebenen Dateideskriptor definierte Datei gelesen. Die Daten müssen in der durch das spaltenweise Schreiben definierten Reihenfolge stehen. Das Zielobjekt muß leer sein; falls es einen Objektnamen hat, wird dieser in der Datei gesucht; andernfalls gilt das erste gelesene Wort als Objektname. Die Daten sind durch übliche Trennzeichen zu trennen.

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

Zeilenweises Schreiben in die angegebene Datei; es wird die Methode `write_row` aufgerufen. Falls als Dateiname das Leerwort angegeben ist, wird in die Standardausgabe geschrieben.

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

Zeilenweises Lesen aus der angegebenen Datei; es wird die Methode read_row aufgerufen. Falls als Dateiname das Leerwort angegeben ist, wird von der Standardeingabe gelesen.

ls_REAL filling_density()

Liefert die Besetzungsdichte des sparse-Feldes.

6.7 Die Klasse sp_Vector - sparse-Vektor

Abgeleitet von: sp_list1<ls_REAL>

Öffentliche Daten:

ls_REAL eps

Genauigkeitsschranke.

sp_Vector(ls_UINT =0,char* ="sp_Vector")

Standardkonstruktor.

sp_Vector(sp_Vector&)

move-Konstruktor.

sp_Vector(ls_REAL*, ls_UINT ,char* ="sp_Vector")

Konstruktor, durch den ein Feld übernommen wird.

const sp_Vector& operator=(const sp_Vector &)

Zuweisungsoperator.

sp_Vector operator-() const

Negativer Vektor.

sp_Vector operator*(ls_REAL) const

Multiplikation eines Vektors mit einer Zahl.

sp_Vector& operator*=(ls_REAL)

Speichernde Multiplikation eines Vektors mit einer Zahl.

sp_Vector operator+(const sp_Vector &) const

Addition zweier Vektoren.

sp_Vector operator-(const sp_Vector &) const

Differenz zweier Vektoren.

sp_Vector& operator+=(const sp_Vector &)

Speichernde Addition zweier Vektoren.

sp_Vector& operator-=(const sp_Vector &)

Speichernde Differenz zweier Vektoren.

`ls_REAL operator*(const sp_Vector &) const`

Skalarprodukt.

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

Multiplikation eines Vektors mit einer Zahl.

6.8 Die Klasse `sp_Matrix` - sparse-Matrix

Abgeleitet von: `sp_list2<ls_REAL>`

Öffentliche Daten:

`sp_Matrix(ls_UINT m=0, ls_UINT n=0, char* ="sp_Matrix")`

Standardkonstruktor für eine sparse-Matrix mit `m` Zeilen und `n` Spalten.

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

Instanziierung einer sparse-Matrix, indem das angegebene zweidimensionale Feld mit der angebenen Zeilen- und Spaltenzahl in ein zweidimensionales sparse-Feld konvertiert wird.

`sp_Matrix(sp_Matrix &)`

`move`-Konstruktor. Die Objekt-Daten der Quelle werden übernommen und in der Quelle gestrichen.

`ls_Vector row(ls_UINT i)const`

Liefert die `i`-te Zeile als normalen Vektor.

`ls_Vector column(ls_UINT j)const`

Liefert die `j`-te Spalte als normalen Vektor.

`ls_Vector upper_diagonal(ls_UINT j) const`

Liefert die `j`-te obere Subdiagonale als normalen Vektor (die Dimension des erzeugten Vektors ist dabei maximal, d. h. gleich der Spaltenanzahl der Matrix).

`ls_Vector lower_diagonal(ls_UINT i)const`

Liefert die `i`-te untere Subdiagonale als normalen Vektor (die Dimension des erzeugten Vektors ist dabei maximal, d. h. gleich der Spaltenanzahl der Matrix).

`const sp_Matrix& operator=(const sp_Matrix &)`

Zuweisungsoperator.

`sp_Matrix operator+(const sp_Matrix &)const`

Matrix-Addition.

`sp_Matrix operator-(const sp_Matrix &)const`

Matrix-Subtraktion.

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

Speichernde Matrix-Addition.

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

Speichernde Matrix-Subtraktion.

`sp_Matrix operator*(const sp_Matrix &)const`

Matrizenmultiplikation.

`sp_Vector operator* (const sp_Vector &)const`

Matrix-mal-Vektor.

`ls_Vector operator* (const ls_Vector &)const`

Matrix-mal-Vektor.

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

Lösen eines linearen Gleichungssystems mit der angegebenen rechten Seite `b` und dem Startvektor `x`. Es wird das konjugierte Gradientenverfahren angewendet. Nach Rückkehr aus dem Programm steht `x` die gefundene Näherungslösung und auf `b` das Residuum. Der Rückkehrwert ist die Anzahl der ausgeführten Schritte. Falls die Zeilenanzahl größer als die Spaltenanzahl ist, wird das entsprechende lineare Ausgleichsproblem gelöst.

`sp_Vector operator* (const sp_Vector &, const sp_Matrix &)`

Transponierte Matrix mal Vektor.

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

Transponierte Matrix mal Vektor.

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

Wert mal Matrix.

6.9 Die Klasse `sp_sMatrix` - symmetrische sparse-Matrix

Abgeleitet von: `sp_listU<ls_REAL>`

Öffentliche Daten:

`sp_sMatrix(ls_UINT n=0,char *nam="sp_sMatrix")`

Instanziierung einer symmetrischen, `n`-diemensionalen Matrix.

`sp_sMatrix(sp_sMatrix &B)`

`move`-Konstruktor.

`ls_Vector row(ls_UINT i)const`

Liefert die i-te Zeile als normalen Vketor.

`ls_Vector diagonal(ls_UINT i)const`

Liefert die i-te Subdiagonale als normalen Vketor.

`const sp_sMatrix& operator=(const sp_sMatrix &)`

Zuweisungsoperator.

`sp_sMatrix operator+(const sp_sMatrix &)const`

Matrix-Addition.

`sp_sMatrix operator-(const sp_sMatrix &)const`

Matrix-Subtraktion.

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

Speichernde Matrix-Addition.

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

Speichernde Matrix-Subtraktion.

`sp_Matrix operator*(const sp_sMatrix &)const`

Matrix-Multiplikation.

`sp_Vector operator* (const sp_Vector &) const`

Matrix mal Vektor.

`ls_Vector operator* (const ls_Vector &) const`

Matrix mal Vektor.

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

Lösen eines linearen Gleichungssystems mit der angegebenen rechten Seite b und dem Startvektor x. Es wird das konjugierte Gradientenverfahren angewendet. Nach Rückkehr aus dem Programm steht x die gefundene Näherungslösung und auf b das Residuum. Der Rückkehrwert ist die Anzahl der ausgeführten Schritte.

6.10 Die Klasse sp_LU LU-Faktorisierung ohne Pivotisierung

Nichtöffentliche Daten:

`sp_Matrix A, F`

Matrix und ihre Faktorisierung.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten:

`char* name, A_name, F_name`

Zeiger auf alle Objektnamen.

`ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert nach der Faktorisierung.

`sp_LU(char* ="sp_LU")`

Instanziierung eines leeren Objektes.

`sp_LU(sp_Matrix &, ls_REAL alpha=0., char* ="sp_LU")`

`move`-Konstruktor; die übergebene quadratische Matrix wird übernommen und faktorisiert. Falls `alpha` eine kleine, positive Zahl ist, wird eine aus der übergebenen Matrix erzeugte, regularisierte Matrix faktorisiert.

`sp_LU(sp_LU &)`

`move`-Konstruktor.

`unsigned char good()const`

Liefert den Fehlercode, der aussagt, ob die Faktorisierung erfolgreich war; der Wert selbst gibt die Schrittnummer an, in der kein Pivotelement gefunden wurde.

`sp_LU& swap(sp_LU &)`

Objekt-Daten-Tausch.

`const sp_LU& operator=(const sp_LU &)`

Zuweisungsoperator.

`const sp_LU& solve(ls_Vector &x, const ls_Vector &b) const`

Lösen eines linearen Gleichungssystems mit der rechten Seite `b`; auf `x` wird die Lösung zurückgegeben.

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

Nachiteration einer Lösung `x`; auf `x` wird die nachiterierte Lösung und auf `b` das Residuum zurückgegeben; der Rückkehrwert ist die ausgeführte Schrittzahl.

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

Liefert das Residuum `b - A*x`.

`const sp_LU& write_row(ostream &) const`

Zeilenweises Schreiben der im Objekt auftretenden Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU& read_row(istream &)`

Zeilenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

`const sp_LU& write_column(ostream &) const`

Spaltenweises Schreiben der im Objekt auftretenden Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU& read_column(istream &)`

Spaltenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

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

Schreiben in die namentlich gegebene Datei mit der Methode `write_row`.

`sp_LU& operator<< (char *)`

Lesen aus der namentlich gegebenen Datei mit der Methode `read_row`.

6.11 Die Klasse `sp_LU_row` `LU`-Faktorisierung mit Zeilen-Pivotisierung

Nichtöffentliche Daten:

`sp_Matrix A, F`

Matrix und ihre Faktorisierung.

`ls_array1<ls_UINT> irow, icol`

Felder mit den wahren Zeilen- und Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten:

`char* name, A_name, F_name, irow_name, icol_name`

Zeiger auf alle Objektnamen.

`ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert nach der Faktorisierung.

`sp_LU_row(char* ="sp_LU_row")`

Instanziierung eines leeren Objektes.

`sp_LU_row(sp_Matrix &, ls_REAL alpha=0., char* ="sp_LU_row")`

`move`-Konstruktor; die übergebene quadratische Matrix wird übernommen und faktorisiert. Falls `alpha` eine kleine, positive Zahl ist, wird eine aus der übergebenen Matrix erzeugte, regularisierte Matrix faktorisiert.

`sp_LU_row(sp_LU_row &)`

`move`-Konstruktor.

`unsigned char good()const`

Erfolgssignal nach der Faktorisierung.

`sp_LU_row& swap(sp_LU_row &)`

Objekt-Daten-Tausch.

`const sp_LU_row& operator=(const sp_LU_row &)`

Zuweisungsoperator.

`const sp_LU_row& solve(ls_Vector &x, const ls_Vector &b) const`

Lösen eines linearen Gleichungssystems mit der rechten Seite `b`; auf `x` wird die Lösung zurückgegeben.

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

Nachiteration einer Lösung `x`; auf `x` wird die nachiterierte Lösung und auf `b` das Residuum zurückgegeben; der Rückkehrwert ist die ausgeführte Schrittzahl.

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

Liefert das Residuum `b - A*x`.

`const sp_LU_row& write_row(ostream &) const`

Zeilenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU_row& read_row(istream &)`

Zeilenweises Lesen der im Objekt auftretenden Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

`const sp_LU_row& write_column(ostream &) const`

Spaltenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU_row& read_column(istream &)`

Spaltenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

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

Schreiben in die namentlich gegebene Datei mit der Methode `write_row`.

`sp_LU_row& operator<< (char *)`

Lesen aus der namentlich gegebenen Datei mit der Methode `read_row`.

6.12 Die Klasse `sp_LU_column` `LU`-Faktorisierung mit Spalten-Pivotisierung

Nichtöffentliche Daten:

`sp_Matrix A, F`

Matrix und ihre Faktorisierung.

`ls_array1<ls_UINT> irow, icol`

Felder mit den wahren Zeilen- und Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten:

`char* name, A_name, F_name, irow_name, icol_name`

Zeiger auf alle Objektnamen.

`ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert nach der Faktorisierung.

`sp_LU_column(char* ="sp_LU_column")`

Instanziierung eines leeren Objektes.

`sp_LU_column(sp_Matrix &, ls_REAL alpha=0., char* ="sp_LU_column")`

`move`-Konstruktor; die übergebene quadratische Matrix wird übernommen und faktorisiert. Falls `alpha` eine kleine, positive Zahl ist, wird eine aus der übergebenen Matrix erzeugte, regularisierte Matrix faktorisiert.

`sp_LU_column(sp_LU_column &)`

`move`-Konstruktor.

`unsigned char good()const`

Erfolgssignal nach der Faktorisierung.

`sp_LU_column& swap(sp_LU_column &)`

Objekt-Daten-Tausch.

`const sp_LU_column& operator=(const sp_LU_column &)`

Zuweisungsoperator.

`const sp_LU_column& solve(ls_Vector &x, const ls_Vector &b) const`

Lösen eines linearen Gleichungssystems mit der rechten Seite `b`; auf `x` wird die Lösung zurückgegeben.

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

Nachiteration einer Lösung `x`; auf `x` wird die nachiterierte Lösung und auf `b` das Residuum zurückgegeben; der Rückkehrwert ist die ausgeführte Schrittzahl.

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

Liefert das Residuum `b - A*x`.

`const sp_LU_column& write_row(ostream &) const`

Zeilenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU_column& read_row(istream &)`

Zeilenweises Lesen der im Objekt auftretenden Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

`const sp_LU_column& write_column(ostream &) const`

Spaltenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU_column& read_column(istream &)`

Spaltenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

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

Schreiben in die namentlich gegebene Datei mit der Methode `write_row`.

`sp_LU_column& operator<< (char *)`

Lesen aus der namentlich gegebenen Datei mit der Methode `read_row`.

6.13 Die Klasse `sp_LU_diagonal` `LU`-Faktorisierung mit Diagonal-Pivotisierung

Nichtöffentliche Daten:

`sp_Matrix A, F`

Matrix und ihre Faktorisierung.

`ls_array1<ls_UINT> ind`

Feld mit den wahren Zeilen/Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten:

`char* name, A_name, F_name, ind_name`

Zeiger auf alle Objektnamen.

`ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert nach der Faktorisierung.

`sp_LU_diagonal(char* ="sp_LU_diagonal")`

Instanziierung eines leeren Objektes.

`sp_LU_diagonal(sp_Matrix &,ls_REAL s=0.,char* ="sp_LU_diagonal")`

`move`-Konstruktor; die übergebene quadratische Matrix wird übernommen und faktorisiert. Falls `s` eine kleine, positive Zahl ist, wird eine aus der übergebenen Matrix erzeugte, regularisierte Matrix faktorisiert.

`sp_LU_diagonal(sp_LU_diagonal &)`

`move`-Konstruktor.

`unsigned char good()const`

Erfolgssignal nach der Faktorisierung.

`sp_LU_diagonal& swap(sp_LU_diagonal &)`

Objekt-Daten-Tausch.

`const sp_LU_diagonal& operator=(const sp_LU_diagonal &)`

Zuweisungsoperator.

`const sp_LU_diagonal& solve(ls_Vector &x, const ls_Vector &b) const`

Lösen eines linearen Gleichungssystems mit der rechten Seite `b`; auf `x` wird die Lösung zurückgegeben.

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

Nachiteration einer Lösung `x`; auf `x` wird die nachiterierte Lösung und auf `b` das Residuum zurückgegeben; der Rückkehrwert ist die ausgeführte Schrittzahl.

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

Liefert das Residuum `b - A*x`.

`const sp_LU_diagonal& write_row(ostream &) const`

Zeilenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU_diagonal& read_row(istream &)`

Zeilenweises Lesen der im Objekt auftretenden Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

`const sp_LU_diagonal& write_column(ostream &) const`

Spaltenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LU_diagonal& read_column(istream &)`

Spaltenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

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

Schreiben in die namentlich gegebene Datei mit der Methode `write_row`.

`sp_LU_diagonal& operator<< (char *)`

Lesen aus der namentlich gegebenen Datei mit der Methode `read_row`.

6.14 Die Klasse `sp_LDLT` `LDLT`-Faktorisierung einer symmetrischen Matrix ohne Pivotisierung

Nichtöffentliche Daten:

`sp_Matrix A, F`

Matrix und ihre Faktorisierung.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten:

`char* name, A_name, F_name`

Zeiger auf alle Objektnamen.

`ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert nach der Faktorisierung.

`sp_LDLT(char* ="sp_LDLT")`

Instanziierung eines leeren Objektes.

`sp_LDLT(sp_Matrix &, ls_REAL alpha=0., char* ="sp_LDLT")`

`move`-Konstruktor; die übergebene quadratische Matrix wird übernommen und faktorisiert. Falls `alpha` eine kleine, positive Zahl ist, wird eine aus der übergebenen Matrix erzeugte, regularisierte Matrix faktorisiert.

`sp_LDLT(sp_LDLT &)`

`move`-Konstruktor.

`unsigned char good()const`

Erfolgssignal nach der Faktorisierung.

`sp_LDLT& swap(sp_LDLT &)`

Objekt-Daten-Tausch.

`const sp_LDLT& operator=(const sp_LDLT &)`

Zuweisungsoperator.

`const sp_LDLT& solve(ls_Vector &x, const ls_Vector &b) const`

Lösen eines linearen Gleichungssystems mit der rechten Seite `b`; auf `x` wird die Lösung zurückgegeben.

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

Nachiteration einer Lösung `x`; auf `x` wird die nachiterierte Lösung und auf `b` das Residuum zurückgegeben; der Rückkehrwert ist die ausgeführte Schrittzahl.

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

Liefert das Residuum `b - A*x`.

`const sp_LDLT& write_row(ostream &) const`

Zeilenweises Schreiben der im Objekt auftretenden Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LDLT& read_row(istream &)`

Zeilenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

`const sp_LDLT& write_column(ostream &) const`

Spaltenweises Schreiben der im Objekt auftretenden Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LDLT& read_column(istream &)`

Spaltenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

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

Schreiben in die namentlich gegebene Datei mit der Methode `write_row`.

`sp_LDLT& operator<< (char *)`

Lesen aus der namentlich gegebenen Datei mit der Methode `read_row`.

6.15 Die Klasse `sp_LDLT_diagonal` `LDLT`-Faktorisierung einer symmetrischen Matrix mit Diagonal-Pivotisierung

Nichtöffentliche Daten:

`sp_Matrix A, F`

Matrix und ihre Faktorisierung.

`ls_array1<ls_UINT> ind`

Feld mit den wahren Zeilen/Spaltenindices.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten:

`char* name, A_name, F_name, ind_name`

Zeiger auf alle Objektnamen.

`ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert nach der Faktorisierung.

`sp_LDLT_diagonal(char* ="sp_LDLT_diagonal")`

Instanziierung eines leeren Objektes.

`sp_LDLT_diagonal(sp_Matrix &,ls_REAL s=0.,char* ="sp_LDLT_diagonal")`

`move`-Konstruktor; die übergebene quadratische Matrix wird übernommen und faktorisiert. Falls `s` eine kleine, positive Zahl ist, wird eine aus der übergebenen Matrix erzeugte, regularisierte Matrix faktorisiert.

`sp_LDLT_diagonal(sp_LDLT_diagonal &)`

`move`-Konstruktor.

`unsigned char good()const`

Erfolgssignal nach der Faktorisierung.

`sp_LDLT_diagonal& swap(sp_LDLT_diagonal &)`

Objekt-Daten-Tausch.

`const sp_LDLT_diagonal& operator=(const sp_LDLT_diagonal &)`

Zuweisungsoperator.

`const sp_LDLT_diagonal& solve(ls_Vector &x, const ls_Vector &b)const`

Lösen eines linearen Gleichungssystems mit der rechten Seite `b`; auf `x` wird die Lösung zurückgegeben.

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

Nachiteration einer Lösung `x`; auf `x` wird die nachiterierte Lösung und auf `b` das Residuum zurückgegeben; der Rückkehrwert ist die ausgeführte Schrittzahl.

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

Liefert das Residuum `b - A*x`.

`const sp_LDLT_diagonal& write_row(ostream &) const`

Zeilenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LDLT_diagonal& read_row(istream &)`

Zeilenweises Lesen der im Objekt auftretenden Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

`const sp_LDLT_diagonal& write_column(ostream &) const`

Spaltenweises Schreiben der Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_LDLT_diagonal& read_column(istream &)`

Spaltenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

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

Schreiben in die namentlich gegebene Datei mit der Methode `write_row`.

`sp_LDLT_diagonal& operator<< (char *)`

Lesen aus der namentlich gegebenen Datei mit der Methode `read_row`.

6.16 Die Klasse `sp_sCG_precond` Vorkonditioniertes konjugiertes Gradientenverfahren für eine symmetrischen Matrix

Nichtöffentliche Daten:

`sp_Matrix A, F`

Matrix und ihre Vorkonditionierung.

`char ONAME[ls_len]`

Feld für den Objektnamen.

Öffentliche Daten:

`char* name, A_name, F_name`

Zeiger auf alle Objektnamen.

`ls_REAL eps`

Genauigkeitsschranke.

`int rc`

Rückkehrwert nach der Faktorisierung.

`sp_sCG_precond(char* ="sp_sCG_precond")`

Instanziierung eines leeren Objektes.

`sp_sCG_precond(sp_Matrix &,ls_REAL s=0.,char* ="sp_sCG_precond")`

`move`-Konstruktor; die übergebene quadratische Matrix wird übernommen und faktorisiert. Falls `s` eine kleine, positive Zahl ist, wird eine aus der übergebenen Matrix erzeugte, regularisierte Matrix faktorisiert.

`sp_sCG_precond(sp_sCG_precond &)`

`move`-Konstruktor.

`unsigned char good()const`

Erfolgssignal nach der Faktorisierung.

`sp_sCG_precond& swap(sp_sCG_precond &)`

Objekt-Daten-Tausch.

`const sp_sCG_precond& operator=(const sp_sCG_precond&)`

Zuweisungsoperator.

`const sp_sCG_precond& solve(ls_Vector &x,ls_Vector &b) const`

Lösen eines linearen Gleichungssystems mit der rechten Seite `b`; auf `x` wird die Lösung zurückgegeben.

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

Liefert das Residuum `b - A*x`.

`const sp_sCG_precond& write_row(ostream &) const`

Zeilenweises Schreiben der im Objekt auftretenden Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_sCG_precond& read_row(istream &)`

Zeilenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

`const sp_sCG_precond& write_column(ostream &) const`

Spaltenweises Schreiben der im Objekt auftretenden Objekte in die durch den Dateidedeskriptor definierte Datei in der auftretenden Reihenfolge. Die Objekte werden umrahmt vom Objektnamen und geschweiften Klammern.

`sp_sCG_precond& read_column(istream &)`

Spaltenweises Lesen der Objekte aus der durch den Dateidedeskriptor definierten Datei. Die Objekte müssen umrahmt sein vom Objektnamen und geschweiften Klammern.

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

Schreiben in die namentlich gegebene Datei mit der Methode `write_row`.

`sp_sCG_precond& operator<< (char *)`

Lesen aus der namentlich gegebenen Datei mit der Methode `read_row`.