Fieger, Heumann, Kastner, Watzka:

Generische Bibliothek zur Linearen Algebra und zur Simulation in C++

Sonderforschungsbereich 386, Paper 63 (1997)

Online unter: http://epub.ub.uni-muenchen.de/

Projektpartner

gsf

MAX-PLANCK-GESELLSCHAFT

Generische Bibliothek zur Linearen Algebra und zur Simulation in C++

A. Fieger, C. Heumann, C. Kastner, K. Watzka

Institut für Statistik
Akademiestr. 1
80799 München

August 1997

Abstract

Die hier vorgestellte Bibliothek implementiert in statistischen Anwendungsprogrammen häufig benötigte Bausteine in einer objektorientierten Sprache. Eine solche Bibliothek tritt ihrem Anwender nicht als „black box“ entgegen, sondern ist mit vertretbarem Aufwand erweiterbar und spezialisierbar.

Keywords: Behälterklassen, Beispiele, Generische Programmierung, lineare Algebra, OOP, parametrische Klassen, Pseudo-Zufallszahlen.

1 Einleitung

1.1 Idee

Problemorientierte Programmiersprachen und statistische Programmierprobleme

Programmiersprachen der vierten Generation – sogenannte problemorientierte Programmiersprachen – existieren auch für das Anwendungsgebiet Statistik. Bekannte problemorientierte Sprachen sind GAUSS, S+, die Sprachen IMSL und Datastep aus dem Programmpaket SAS, oder auch MATLAB. Ein typisches Problem, das von solchen Werkzeugen gelöst wird, ist die einfache Handhabung von Matrizen und Vektoren. Die Lösung, die solche Programmiersprachen bieten, ist

oft sehr benutzerfreundlich, gerade für Benutzer mit Anwendungswissen. Problemorientierte Sprachen bleiben in ihrer Syntax und Semantik meist sehr nahe an der Notation, die auch im Anwendungsgebiet üblich ist. So ist die Semantik solcher Sprachen für Benutzer aus dem Anwendungsgebiet selten überraschend.

Leider haben manche der problemorientierten Sprachen für den Anwendungsbereich Statistik den Nachteil, daß die Basis für die Implementation eigener Algorithmen nicht besonders effizient ist. Die „eingebauten“ Algorithmen sind zwar in hochspezialisierter Weise verwirklicht, aber auf die Schnittstelle für Eigenentwicklungen, die über diese Verfahren hinausgehen, wird meist nicht viel Wert gelegt.

Nicht problemorientierte Programmiersprachen

Gerade für moderne, datenintensive Verfahren, ist daher oft eine Implementation in einer nicht problemorientierten Sprache wie Pascal oder C erforderlich, wenn sie nicht „künstlich“ in die Ausdrucksweise der vorhandenen Sprachen der vierten Generation „gepresst“ werden sollen. Nicht problemorientierte Programmiersprachen haben allerdings einen Nachteil: Oft ist relativ viel Aufwand erforderlich, um Operationen zu implementieren, die bei problemorientierten Sprachen mit „eingebaut“ sind.

Ein traditioneller Ansatz, mit dem dieser Nachteil ausgeglichen werden soll, ist die Verwendung von Programmbibliotheken, in denen bestimmte Teilprobleme effizient gelöst sind. Die *Integration* solcher Programmbibliotheken ist allerdings nicht immer einfach zu verwirklichen.

Hier steht ein Anwender, der zwischen dem Einsatz einer problemorientierten Programmiersprache oder einer Programmbibliothek entscheiden muß, vor einem Dilemma: Auf der Seite der Programmbibliotheken steht die stark technische Ausrichtung, auf der der problemorientierten Sprachen die starke Orientierung an der Begriffswelt der Anwendung unter Vernachlässigung von technischen Gesichtspunkten.

Die hier vorgestellte Bibliothek stellt den Versuch dar, durch die Implementation von häufig benötigten *Bausteinen* in einer objektorientierten Sprache, einen Mittelweg anzubieten. Das objektorientierte Paradigma mit den Prinzipien

- der Datenkapselung mit dem Ziel, enge Schnittstellen zu schaffen, und
- der Vererbung, die die Implementation auf *verschiedenen Abstraktionsstufen* ermöglicht,

erleichtert es, Bibliotheken zu entwerfen und zu realisieren, die ihrem Anwender nicht als „black box“ entgegentreten, sondern mit vertretbarem Aufwand erweiterbar und spezialisierbar sind.

Implementation als generische Bibliothek

Beim Entwurf einer portablen Klassenbibliothek zur linearen Algebra stellte sich die Frage, für welche Zahldarstellung diese Bibliothek entworfen werden sollte. Auf manchen Zielplatformen unterscheidet sich die Geschwindigkeit von Fließkomma-Operationen deutlich, je nachdem, welche Zahldarstellung verwendet wird. Andere Zielplatformen verwenden ohnehin nur einen Fließkommadatentyp.

Die Forderungen an die Genauigkeit der Zahldarstellung können von Anwendungsfall zu Anwendungsfall verschieden sein. Die Entscheidung für die maximal mögliche Mantissenbreite bedeutet, je nach Zielplatform, die gleichzeitige Entscheidung für die geringste Verarbeitungsgeschwindigkeit.

Algorithmen zur linearen Algebra sind nicht unbedingt abhängig von der verwendeten Fließkommazahldarstellung. So war es ein logischer Schritt, die Entscheidung über den verwendeten Datentyp dem Anwender zu überlassen. Dieser kann in seine Entscheidung sowohl die Eigenschaften der auf seinem System verfügbaren Datentypen als auch die Eigenschaften des zu lösenden Problems einbeziehen. Die vorhandenen Datentypen legen den Entscheidungsspielraum in Bezug auf Verarbeitungsgeschwindigkeit und Genauigkeit der Darstellung fest. Die Forderungen in diese Parameter ergeben sich aus dem Problem.

Generische Programmierung

Eine Möglichkeit zur Implementation von Algorithmen, die nur bestimmte Forderungen an die verwendeten Datentypen stellen, ohne diese Datentypen selbst festzulegen, bietet die *generische Programmierung*. Anstelle konkreter Datentypen werden „Platzhaltertypen“ verwendet.

Die Forderungen an konkrete Datentypen, auf die ein generischer Algorithmus anwendbar ist, sind durch die auf die Objekte des verwendeten Datentyps angewandten Operationen implizit festgelegt.

Der generische Algorithmus ist auf alle Datentypen anwendbar, die alle Operationen bereitstellen, die bei seiner Implementation benötigt werden. Konkret bedeutet dies, daß die Bibliothek einen Datentyp `T` für die Elemente z. B. einer Matrix verwendet, und daß eine *Instantiierung* der Bibliothek für jeden Typ möglich ist, der gewisse Mindestanforderungen erfüllt. Dies bietet den offensichtlichen Vorteil, daß die Entscheidung bezüglich des Trade-Offs zwischen Mantissenbreite und Verarbeitungsgeschwindigkeit beim Anwender der Bibliothek liegt. Daneben gibt es mindestens einen weiteren Vorteil: Die Klassenbibliothek kann auch für benutzerdefinierte Datentypen instantiiert werden.

Für manche Anwendungen ist eine exakte Rechnung notwendig. Für solche Anwendungen ist es möglich, oder wenigstens sinnvoll, eine Instatiierung mit

einem benutzerdefinierten Typ zur exakten Darstellung rationaler Zahlen vorzunehmen. Dies setzt dann nur voraus, daß keine Operationen auftreten, die Ergebnisse außerhalb des mit dieser Zahldarstellung repräsentierbaren Wertebereichs liefern.

Spezialisierungen

Spezialisierungen von parametrischen Klassen erfolgen, indem die parametrische Klasse nicht nur mit einem Datentyp instantiiert wird, sondern indem darüber hinaus entweder für diesen Datentyp notwendige oder nützliche Methoden mit aufgenommen werden, oder indem bei der Implementation von Methoden die Eigenschaften des zur Instantiierung verwendeten Datentyps berücksichtigt werden.

Da die Masse der Anwendungen mit Fließkommazahlen als Typ eines Matrixelements arbeiten wird, und manche Algorithmen nur für Fließkommazahlen als Elementtyp sinnvoll sind, wurden zwei weitere Designentscheidungen getroffen und bei der Implementation der Bibliothek berücksichtigt:

- Eine Spezialisierung der Klasse `Matrix` für einen Datentyp `real`, bildet die Klasse `realMatrix`. Von dem verwendeten Datentyp `real` wird lediglich erwartet, daß es sich um einen Fließkommatyp handelt. Dieser Datentyp ist noch kein konkreter Typ.

- Zusätzliche Schnittstellenbeschreibungen erleichtern es dem Anweder, eine Instantiierung für den C++-Datentyp `double` zu verwenden, ohne sich näher mit generischen Algorithmen und der Instantiierung beschäftigen zu müssen.

1.2 Design der Klassenbibliothek

`Array2D`, `PreMartrix` und `Matrix`

Die drei Klassen `Array2D`, `PreMatrix` und `Matrix` bilden den „Baumstamm“ der Klassenbibliothek. Eine grobe Betrachtung des Entwurfs kann sich auf diese Klassen beschränken.

Die Klasse `Array2D` ist ein sehr allgemein gehaltenes 2-dimensionales Feld, das auch die Basis einer einfachen Tabellenkalkulation mit zusätzlichen Datenbankfunktionen bilden könnte. Die Operationen einer elementaren relationalen Algebra werden unterstützt. Dies sind insbesondere Projektionen, Selektionen und Vereinigungen, nicht aber das kartesische Produkt.

Die Klasse `PreMatrix` erweitert `Array2D` um alle Matrixoperationen, die mit einfachen arithmetischen Operationen auf dem Elementdatentyp implementiert

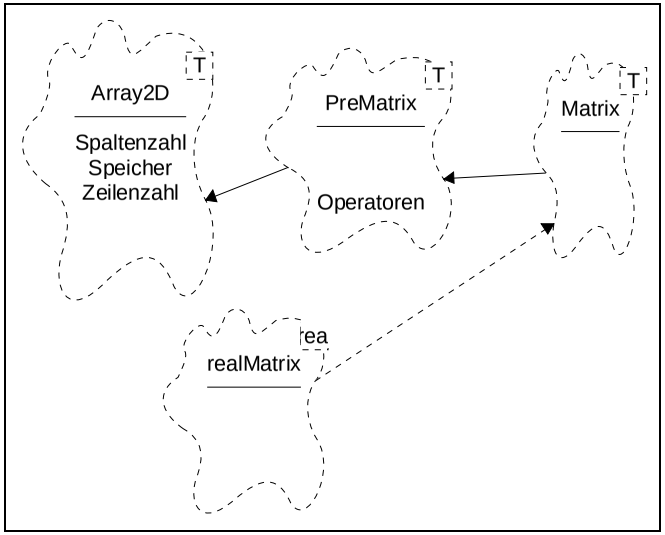

Abbildung 1: Booch-Diagramm der wesentlichen Klassen der Bibliothek

werden können. Sie wurde von Operationen freigehalten, die mehr als einfache arithmetische Operationen von einem Datentyp fordern, der zu ihrer Instantiierung verwendet wird. Dieser Entwurf soll es möglich machen, auf die wesentlichen Operationen zuzugreifen, die mit einer Matrix möglich sind, ohne die Implementation von Funktionen für den Elementdatentyp zu fordern, die mit einer endlichen Zahldarstellung nicht exakt implementiert werden können. Zumindest für die Instantiierung einer `PreMatrix` kann jeder arithmetische Datentyp verwendet werden, der die in Abschnitt 2 näher beschriebenen Forderungen erfüllt.

Die Klasse `Matrix` implementiert schließlich die meisten in der Statistik häufig benötigten Matrixoperationen, soweit dies ohne die Annahme möglich ist, daß die parametrische Klasse mit einem Fließkommatyp instantiiert wird.

1.3 Konventionelle Behälterklassen

Neben den Klassen, die den Kern der Bibliothek bilden, stehen auch verschiedene Behälterklassen einem Benutzer der Bibliothek zur Verfügung, die intern von der Bibliothek ohnehin benötigt werden.

Verantwortung für in einem Behälter abgelegte Objekte

Die Behälter bewahren prinzipiell eigene Kopien der eingefügten Objekte auf. Sie haben demzufolge Wertsemantik. Dies bedeutet, daß eine Instanz in dem Moment freigegeben werden kann, in dem sie erfolgreich in einen Behälter eingefügt wurde. Soll eine Referenzsemantik für Objekte in Behältern erreicht werden, muß dies durch den Anwender der Bibliothek selbst verwaltet werden. Hierzu ist das Entwurfmuster des „smart pointers“ zu empfehlen, d. h. die Referenzen sollten über Referenzzähler für den referierten Speicherplatz verantwortlich bleiben.

Beim Löschen eines Behältertyps werden auch die vom Behältertyp verwalteten Kopien freigegeben. Ein Eingriff des Benutzers ist nicht erforderlich, wenn der Destruktor eines Elementdatentyps ausreichend ist.

`Array`

Vektoren sollten als Matrizen der Zeilen- oder Spaltendimension eins dargestellt werden. Ein gewöhnliches Array, das nicht als Vektor im Sinne der Sprache des Anwendungsgebietes verwendet werden soll, kann durch den parametrischen Behältertyp `Array` gebildet werden.

Verzeigerte Listen und Adaptoren

Die Behälterklassen der Klassenbibliothek beruhen technisch auf der Klasse `LinkedList`, und stellen lediglich Adaptoren für diese Klasse dar. Dies bedeutet, daß Einfügeoperationen von konstanter Komplexität sind, während der Aufwand für Suchoperationen linear in der Anzahl der in einem Behälter abgelegten Elemente wächst.

Stack Die Behälterklasse `Stack` implementiert einen FIFO-Behälter, d. h., Einfügeoperationen, Löschoperationen und Leseoperationen finden am gleichen Ende statt.

Queue Die Behälterklasse `Queue` implementiert einen LIFO-Behälter, d. h., Einfügeoperationen finden am einen, Lese- und Löschoperationen am anderen Ende der Liste statt.

Heap Die Behälterklasse `Heap` verwaltet Objekte nach einer Ordnungsrelation. Eingefügte Objekte werden einsortiert, und der Zugriff zum Lesen und Entfernen erfolgt an dem Ende der sortierten Liste, an dem die nach der verwendeten Ordungsrelation „kleinsten“ Elemente stehen.

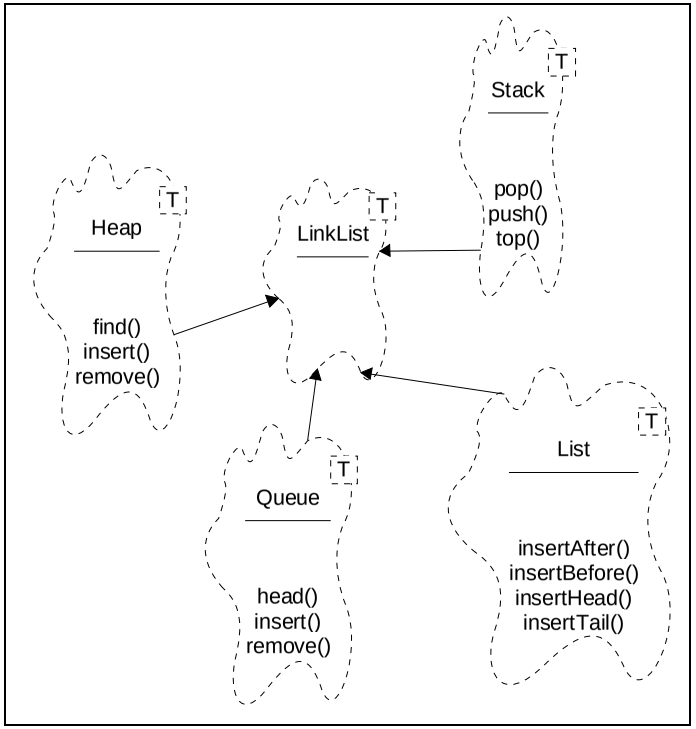

Abbildung 2: Booch-Diagramm der Behälterklassen

List Die Behälterklasse `List` implementiert eine einfach verzeigerte Liste auf der Basis von `LinkedList`. Elemente können am Anfang, am Ende, oder vor oder nach einem angegebenen Element eingefügt werden.

Der Zugriff auf Behälter, und damit das Speichern einer „aktuellen Position“ in einem Behälter, wird in einem objektorientierten Entwurf oft durch sogenannte Iteratoren gekapselt. Der Vorteil der Verwaltung von „aktuellen Positionen“ durch Iteratoren ist, daß innerhalb eines Behälters mehrere „aktuelle Positionen“ aktiv sein können, d. h. daß ein Behälter mehrfach unabhängig durchlaufen werden kann. Diesem Entwurfsmuster folgend erfolgt der Zugriff auf Elemente einer `List` über die parametrische Klasse `ListIterator`, die eine gegebene Liste vorwärts durchläuft. Diese Iteratoren werden durch Einfüge- oder Löschoperationen auf der Liste, zu der sie gehören, ungültig.

Ignorieren der „Standard Template Library“ und neuerer Entwicklungen

Die Standard Template Library (STL) ist der Teil der zukünftigen Standardbibliothek, der unter anderem parametrische Behältertypen – also beispielsweise Listen, Vektoren und Bäume – zur Verfügung stellt. Spätestens bei der Vorstellung einer weiteren Hierarchie von Behältertypen wird ein Leser, der den aktuellen Stand der Definition der Programmiersprache C++ kennt, die Frage stellen, weshalb diese bereits vorhandenen parametrischen Klassen nicht verwendet wurden.

Zum jetzigen Zeitpunkt, also noch vor Abschluß der Sprachdefinition, und damit lange vor deren Umsetzung in existierenden Implementationen der Programmiersprache C++, erschien es uns verfrüht, diesen in Bezug auf die Implementation anspruchsvollen Teil der Sprache bereits einzusetzen.

Auch andere „neuere“ Konstrukte der Sprache wurden bewußt aus der Implementation herausgehalten. In Bezug auf den verwendeten Sprachumfang ist diese Bibliothek also recht konservativ.

Dennoch wurde die Existenz dieser standardisierten Behälter-Bibliothek nicht völlig ignoriert. Die Schnittstelle der Klassenbibliothek ist so gestaltet, daß sie für einen Benutzer, der die generischen Algorithmen der STL kennt, keine überraschende Semantik hat. Die Sichtweise, daß Feldindices mit 0 beginnen, und das Anfang und Ende einer Schleife in „klassischem“ C-Stil angegeben werden („vom Anfang bis vor das Element hinter dem Ende“), sollten zumindest Benutzern vertraut sein, die entweder die ursprüngliche Programmiersprache C, oder die STL kennen:

```
int array[N], *parray;
for (parray = array; parray < array + N; ++parray)
      /*      Behandle ein Element      */
```

Als Anfangswert eines Bereichs wird der Startwert, als Endwert eines Bereichs dagegen der Wert nach dem letzten zu verarbeitenden Wert angegeben.

2 Anforderungen an den arithmetischen Datentyp

Es gibt verschiedene „Ausbaustufen“ von den sehr allgemein gehaltenen doppelt indizierten Behältern `Array2D` zu den spezialisierten Matrizen vom Typ `realMatrix`, auf deren Elemente Funktionen anwendbar sein müssen, die nicht als Polynome mit rationaler Basis darstellbar sind. Diese „Ausbaustufen“ stellen unterschiedliche Anforderungen an die verwendeten Datentypen. Umgekehrt legen die auf einen Datentyp anwendbaren Operationen fest, wieviel Funktionalität des Behältertyps genutzt werden kann.

Manche Implementationen objektorientierter Sprachen nehmen bei der Instantiierung von *parametrischen Klassen* nur solche Instantiierungen vor, die durch die konkrete Anwendung gefordert werden. Solche Implementationen können in der Regel erst zur Laufzeit feststellen, ob alle benötigten Methoden mit dem aktuellen Parameter, der zur Instantiierung eines parametrischen Typs verwendet wurde, verwirklicht werden können. Der Aufruf einer nicht implementierten Botschaft wird solange zum Basistyp weitergereicht, bis ein an der Wurzel des Typensystems stehender Typ feststellt, daß die Methode nicht implementiert ist, und einen Laufzeitfehler erzeugt[1].

Dieses Konzept dieser sehr *späten Bindung* wird von C++ nicht unterstützt. C++ ist eine *stark typisierte Sprache*, und derartige Konflikte werden zum Zeitpunkt der Übersetzung aufgelöst. Dies bedeutet, daß ein Anwender nicht damit rechnen muß, daß ein Laufzeitfehler auftritt, weil eine parametrische Klasse mit einem Datentyp instantiiert wurde, der nicht alle erforderlichen Operationen bereitstellt.

Die Entscheidung für einen zur Implementation der Behälter passenden Datentyp fällt idealerweise bereits in der Entwurfsphase eines Programms. Die folgende Übersicht soll einen Anwender der verschiedenen Behälter, der vor einem solchen Entscheidungsproblem steht, bei dieser Auswahlentscheidung unterstützen.

2.1 Array2D

Klassen, die bei der Instantiierung eines `Array2D` verwendet werden sollen, brauchen mindestens einen parameterlosen Konstruktor, den sogenannten Default-Konstruktor, um Speicherbereiche als Instanzen der Elementtypklasse initialisieren zu können, einen Einfügeoperator für `ostreams` und einen Extraktionsoperator für `istreams`. Zum Kopieren der Elemente wird deren Zuweisungsoperator `operator=` verwendet. Wenn der automatisch bereitgestellte Zuweisungsoperator, der alle Instanzvariablen kopiert, für die verwendete Klasse nicht ausreicht, muß auch ein Zuweisungsoperator definiert werden. Eine sehr rudimentäre Klasse, die zur Instantiierung von `Array2D` verwendet werden kann, ist

```
class Array2DElem
{
public:
   Array2DElem() {}
```

[1] Eine objektorientierte Programmiersprache, die dieses Konzept der späten Bindung verwirklicht, ist Smalltalk. Eine andere objektorinetierte Programmiersprache, die der Programmiersprache C sehr viel ähnlicher ist, aber ebenfalls dem Konzept der späten Bindung folgt, ist die Sprache `Objective C`.

```
   bool operator==(const Array2DElem &) const;
   friend ostream &operator<<(ostream &os, const Array2DElem &)
      { return os;}
   friend istream &operator>>(istream &is, Array2DElem &)
      { return is; }
};
```

Diese Klasse `Array2DElem` implementiert alle Methoden, die die parameterische Klasse `Array2D` für die als Parameterbelegung des formalen Parameters `T` verwendete Klasse voraussetzt. Die Operatoren zur Ein- und Ausgabe mit `stream`-Objekten müssen nicht wirklich zu einer Ein- oder Ausgabe führen, wenn dies für eine konkrete Anwendung nicht sinnvoll erscheint.

Wenn eine Implementation der Sprache C++ die Möglichkeit der „vollständigen Instantiierung“ aller Methoden eine Templateklasse gewählt hat, kann es aber trotzdem erforderlich sein, die oben beschriebenen „blinden“ Implementationen bereitzustellen.

2.2 PreMatrix

Die parametrische Klasse `PreMatrix` stellt bereits deutlich höhere Anforderungen an den Typ, der zu ihrere Instantiierung verwendet wird. Alle einfachen arithmetischen Operatoren müssen definiert sein. Der Typ, der zur Instantiierung von `PreMatrix` verwendet wird, muß daneben auch alle Anforderungen an einen Typ erfüllen, der zur Instantiierung von `Array2D` verwendet werden kann.

```
class PreMatrixElem : Array2DElem
{
public:
   PreMatrixElem() {}
   PreMatrixElem(int);
   PreMatrixElem(const PreMatrixElem &);

   PreMatrixElem operator+(const PreMatrixElem &) const;
   PreMatrixElem operator-(const PreMatrixElem &) const;
   PreMatrixElem operator*(const PreMatrixElem &) const;
   PreMatrixElem operator/(const PreMatrixElem &) const;

   PreMatrixElem operator-() const;

   bool operator<(const PreMatrixElem &) const;
   bool operator<=(const PreMatrixElem &) const;
   bool operator==(const PreMatrixElem &) const;
```



```
   PreMatrixElem operator += (const PreMatrixElem &);
   PreMatrixElem operator -= (const PreMatrixElem &);
   PreMatrixElem operator *= (const PreMatrixElem &);

   friend ostream & operator<<(ostream &, const PreMatrixElem &);
   friend istream & operator>>(istream &, PreMatrixElem &);
};
```

Es muß einen Konstruktor geben, der eine Instanz des verwendeten Elementdatentyps mit einem `int` instantiieren kann. Alle in den für `PreMatrix` definierten Methoden verwendeten Konstanten wurden als Ausdrücke von ganzen Zahlen angegeben. Dies kann bei der Definition von sehr kleinen Konstanten zu Nachteilen führen. An dieser Stelle kann eine Spezialisierung durch den Anwender der Bibliothek sinnvoll sein, wenn die mit ganzen Zahlen erreichbare Genauigkeit nicht ausreichend ist, und der implementierte arithmetische Datentyp die Darstellung von ausreichend kleinen Werten zuläßt.

Es ist sinnvoll und oft auch notwendig, die gewöhnlichen zweistelligen Operatoren so zu implementieren, daß sie ihr Ergebnis als Wert, und nicht als Referenz zurückliefern. Dieser Weg wurde für die hier beschriebene Bibliothek gewählt. So negiert der einstellige Operator `operator-` den Wert seines Operanden und liefert das Ergebnis als Wert zurück.

Wenn der Vergleich der Werte von Instanzen des Elementdatentyps aufweding ist, kann eine einfache Implementation von `operator<=` auch durch die Methoden `operator==` und `operator<` erfolgen. Die Methode `operator<=` kann oft effizienter als

```
bool PreMatrixElem::operator<=(const PreMatrixElem &other) const
{
   return operator<(other) || operator==(other);
}
```

implementiert werden. Daher wurde diese Art von Tests in der Implementation der Methoden von `PreMatrix` als Aufruf der Methode `operator<=`, nicht als logischer Ausdruck mit den Methoden `operator==` und `operator<` codiert. Diese Entscheidung schafft eine etwas breitere Schnittstelle als nötig, führt aber möglicherweise zu einem Effizienzgewinn. Andere Prädikate wie `operator>` oder `operator>=` können durch geeignete Formulierung der Methoden von `PreMatrix` vermieden werden. Dadurch bleibt die geforderte Schnittstelle schmal.

2.3 Matrix

Die parametrische Klasse `Matrix` erweitert die Forderungen an den Typ, der zu ihrer Instantiierung verwendet wird, in einem Punkt. Die Funktion `sqrt` muß für

den verwendeten Typ definiert sein. Dies ist in einer aktuellen Implementation der Sparche C++ für die Typen `float`, `double` und `long double` der Fall. Bei älteren Implementationen können Anpassungsarbeiten erforderlich sein, bei denen die Funktion `sqrt` für andere Typen als `double` geeignet überladen wird.

3 Funktionen für Matrizen

3.1 Elementfunktionen von Matrizen

Die Elementfunktionen der Templateklassen für Matrizen sollen im folgenden genauer vorgestellt werden. Eine Matrix X mit n Zeilen und k Spalten wird hier stets mit

$$\begin{aligned}
\mathbf{X} &= \begin{pmatrix} x_{0,0} & x_{0,1} & \dots & x_{0,k-1} \\ \vdots & & & \vdots \\ x_{n-1,0} & x_{n-1,1} & \dots & x_{n-1,k-1} \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{x}_{\bullet 0} & \mathbf{x}_{\bullet 1} & \dots & \mathbf{x}_{\bullet k-1} \end{pmatrix} \\
&= \begin{pmatrix} \mathbf{x}_{0\bullet} \\ \vdots \\ \mathbf{x}_{n-1\bullet} \end{pmatrix} \\
&= (x_{ij})
\end{aligned}$$

bezeichnet. $\mathbf{x}_{\bullet i}$ bezeichnet also die i-te Zeile, $\mathbf{x}_{j\bullet}$ bezeichnet die j-te Spalte der Matrix $\mathbf{X}$. Die Zeilen- und Spaltenindizes werden stets bei Null beginnend gezählt.

Im folgenden bezeichnet stets `X` das Matrix-Objekt, dessen Elementfunktion vorgestellt wird. Werden in der Parameterliste weitere Matrizen verwendet, so werden diese mit `A`, `B`, usw. bezeichnet. Generell ist zu beachten, daß die Indizierung 'nullbasiert' ist, d. h. anstelle von 1 bis n wird von 0 bis $n-1$ gezählt. Das Matrixelement x_{11} hat den Zeilenindex 0 und den Spaltenindex 0.

Die in dieser Bibliothek implementierten Funktionen von Matrizen liefern drei verschiedene Datentypen zurück.

- `Matrix<T>`-wertige Funktionen.

  Das Ergebnis wird immer als Wert zurückgeliefert. Wenn die Funktion nicht erfolgreich ausgeführt werden konnte, wird eine Instanz vom erwarteten Ergebnistyp zurückgeliefert, für die der `operator int` den Wert `0` und der `operator!` den Wert `true` liefert. Solche Instanzen können mit `if (X)` positiv und mit `if (!X)` negativ auf Gültigkeit geprüft werden. Eine Operation kann fehlschlagen, weil sie von Eigenschaften eines Operanden abhängig ist, die dieser nicht hat, oder weil das Laufzeitsystem eine Resourcenanforderung nicht erfüllen kann[2].

- `bool`-wertige Funktionen.

  Der Typ `bool` wird noch nicht von vielen Implementationen unterstützt. Er kann für die Zwecke dieser Bibliothek durch

  ```
  typedef int bool;
  const bool true = 1;
  const bool false = 0;
  ```

  simuliert werden.

- `unsigned int`-wertige Funktionen.

Die im folgenden beschriebenen Methoden sind echte Methoden der parameterischen Klasse `Matrix<T>`. Sie können nur durch eine Instanz dieser Klasse aufgerufen werden. Um beispielsweise die Zahl der Zeilen einer Matrix zu bestimmen, muß klar sein, welche Matrix gemeint ist. Syntaktisch bedeutet dies, daß vor dem Aufruf einer solchen Methode ein Audruck stehen muß, der den Typ `Matrix<T>` hat.

```
Matrix<double> X(5, 3);
rows();
```

ist deshalb nicht sinnvoll. Es ist nicht klar, auf welche Matrix sich der Aufruf der Methode `rows()` beziehen soll. Dagegen sind

```
Matrix<double> Y(5, 3);
Y.rows();
Y.transposed().rows();
```

sinnvolle Aufrufe. Sie liefern die Zahl der Zeilen der Matrix zurück, die der Ausdruck links vom Punkt-Operator bezeichnet. Dies ist im ersten Fall die Matrix `Y`, im zweiten Fall der Wert den die Methode `transposed()` (angewendet auf die Matrix `Y`) zurückliefert. Eine bildhafte Vorstellung, die diese Sichtweise verdeutlichen hilft, ist die, daß die Elementfunktionen Nachrichten sind, die an Objekte verschickt werden, und diese zu bestimmten Aktivitäten veranlassen.

[2] Im wesentlichen wird es sich dabei um fehlgeschlagene Anforderungen von Hauptspeicher handeln. Diese führen bei moderen Implementationen der Sprache C++ zu „Exceptions“, die innerhalb der Bibliothek nicht behandelt werden. `Matrix<T>`-wertige Funktionen müssen in solchen Implementationen in einen `try`-Block eingebettet werden, oder es muß sichergestellt sein daß fehlgeschlagene Aufrufe des globalen Operators `new` zur Rückgabe eines 0-Zeigers führen.

NAME `Matrix<T>()` – Anlegen einer Matrix

SYNOPSIS

```
Matrix<T>::Matrix<T>(unsigned rows, unsigned cols);
```

BESCHREIBUNG `Matrix<T>()` Anlegen der Matrix mit spezifizierter Zeilen- und Spaltenzahl

`rows` Anzahl der Zeilen der Matrix

`cols` Anzahl der Spalten der Matrix

BEMERKUNGEN Die Matrixelemente werden nicht mit Null vorbesetzt, sondern haben keinen definierten Wert

---

NAME `Matrix<T>()` – Anlegen einer Matrix

SYNOPSIS

```
Matrix<T>::Matrix<T>(
  unsigned rows, unsigned cols,
  T value
);
```

BESCHREIBUNG `Matrix<T>()` Anlegen der Matrix mit spezifizierter Zeilen- und Spaltenzahl und setzen aller Elemente auf den Wert `value`

`rows` Anzahl der Zeilen der Matrix

`cols` Anzahl der Spalten der Matrix

`value` Vorgabewert für alle Elemente der Matrix

NAME `()` – Zugriffsoperator

SYNOPSIS

```
T& Matrix<T>::operator()(unsigned i, unsigned j);
```

BESCHREIBUNG `()` gestattet den lesenden oder schreibenden Zugriff auf das Element x_{ij}

`i` Zeilenindex

`j` Spaltenindex

BEMERKUNGEN Es gibt zwei Varianten dieses Operators. Der Zugriff auf eine Matrix, die nicht verändert werden darf, erlaubt nur einen lesenden Zugriff. Der Zugriff auf eine veränderbare Matrix ermöglicht lesenden und schreibenden Zugriff auf die Matrixelemente.

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `applied()` – Elementweise Operationen

SYNOPSIS

```
Matrix<T> Matrix<T>::applied(T (* func)(T));
```


BESCHREIBUNG `applied()` Anwendung einer Funktion auf die Elemente der Matrix

`func` Anzuwendende Funktion

BEISPIEL `X.applied(sqrt)` ergibt die Matrix $(\sqrt{x_{ij}})$.

NAME `applied()` – Elementweise Verknüpfung

SYNOPSIS

```
Matrix<T> Matrix<T>::applied(
  T (* func)(T), Matrix<T> A
);
```

BESCHREIBUNG `applied()` Anwendung einer Funktion zur elementweisen Verknüpfung mit einer zweiten Matrix

`func` Anzuwendende Funktion

`A` Zweite Matrix

BEMERKUNGEN Es wird vorausgesetzt, daß beide Matrizen die gleiche Dimension haben

BEISPIEL `X.applied(div, A)` ergibt mit der Funktion `T div(T x, T y){ return x/y; }` die Matrix (x_{ij}/a_{ij}).

NAME `blockdiag()` – Blockdiagonalmatrix

SYNOPSIS

```
Matrix<T> Matrix<T>::blockdiag(Matrix<T> A);
```

BESCHREIBUNG `blockdiag()` bildet aus den Matrizen `X` und `A` eine Blockdiagonalmatrix

`A` Matrix die in der unteren rechten Ecke der Ergebnismatrix erscheint

BEMERKUNGEN Das Ergebnis hat die Form $\begin{pmatrix} X & 0 \\ 0 & A \end{pmatrix}$ der Dimension `X.rows+A.rows()` × `X.cols+A.cols()`

BEISPIEL Sei $X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ und $A = 5$, dann ist `X.blockdiag(A)`

$= \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 0 \\ 0 & 0 & 5 \end{pmatrix}$.

NAME `cols()` – Spaltenzahl

SYNOPSIS

```
unsigned Matrix<T>::cols();
```

BESCHREIBUNG `cols()` liefert die Anzahl der Spalten der Matrix zurück

BEMERKUNGEN Die letzte Spalte hat den Index `cols()`−1

SIEHE AUCH `rows()`

NAME `cinverse()` – Inverse

SYNOPSIS

```
Matrix<T> Matrix<T>::cinverse();
```

BESCHREIBUNG `cinverse()` bildet die Inverse einer positiv definiten Matrix

SIEHE AUCH `inverse()`

NAME `csolve()` – Gleichungssystem lösen

SYNOPSIS

```
Matrix<T> Matrix<T>::csolve(const Matrix<T> &b);
```

BESCHREIBUNG `csolve()` löst das Gleichungssystem $Ax = b$, wobei die Matrix `A` positiv definit sein muß

`b` Matrix (Spaltenvektor)

SIEHE AUCH `solve()`

BEISPIEL Die Schätzung $\hat{\beta} = (X'X)^{-1}X'y$ kann mit `(X.sscp()).csolve(X.transposed()*y)` berechnet werden

---

NAME `det()` – Determinante

SYNOPSIS

```
T Matrix<T>::det();
```

BESCHREIBUNG `det()` berechnet die Determinante $|X|$ der Matrix

NAME `diag()` – Diagonalmatrix

SYNOPSIS

```
Matrix<T> Matrix<T>::diag();
```

BESCHREIBUNG `diag()` liefert aus einer Matrix die Diagonale als Spaltenvektor

BEMERKUNGEN Die Elementfunktion ist nicht mit der statischen Funktion `diag()` zu verwechseln, die eine Diagonalmatrix erzeugt.

BEISPIEL Sei $X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$, so liefert `d=X.diag()` den Spaltenvektor $d = \begin{pmatrix} 1 \\ 5 \\ 9 \end{pmatrix}$.

NAME `get()` – Elementzugriff

SYNOPSIS

```
T Matrix<T>::get(unsigned row, unsigned col);
```

BESCHREIBUNG `get()` liefert das Element x_{ij} zurück

`row` Zeilenindedx des Elements

`col` Spaltenindex des Elements

BEMERKUNGEN Auf das Element a_{11} wird mit `A.get(0,0)` zugegriffen

SIEHE AUCH `operator()`, `put()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `getBlock()` – Blockweiser Elementzugriff

SYNOPSIS

```
Matrix<T> Matrix<T>::getBlock(
  unsigned rowfirst, unsigned colfirst,
  unsigned rowlast, unsigned collast
);
```

BESCHREIBUNG `getBlock()` liefert die Teilmatrix (Block) zurück, die durch die Zeilen zwischen `rowfirst` und `rowlast-1` und die Spalten zwischen `colfirst` und `collast-1` beschrieben wird

`rowfirst` Index der ersten Zeile, die in die resultierende Matrix aufgenommen werden soll

`colfirst` Index der ersten Spalte, die in die resultierende Matrix aufgenommen werden soll

`rowlast` Index der Zeile, die nicht mehr in die resultierende Matrix aufgenommen werden soll

`collast` Index der letzten Spalte, die nicht mehr in die resultierende Matrix aufgenommen werden soll

BEMERKUNGEN Es muß `rowfirst` < `rowlast` und `colfirst` < `collast` gelten

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `getCol()` – Blockweiser Elementzugriff

SYNOPSIS

```
Matrix<T> Matrix<T>::getCol(unsigned col);
```

BESCHREIBUNG `getCol()` liefert die Spalte `col` als Spaltenvektor zurück

`col` Index der gewünschten Spalte

BEMERKUNGEN Auf die Spalte $a_{\bullet 1}$ wird mit `A.getCol(0)` zugegriffen

SIEHE AUCH `put()`, `get()`, `putCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `getColBlock()` – Blockweiser Elementzugriff

SYNOPSIS

```
Matrix<T> Matrix<T>::getColBlock(
  unsigned colfirst, unsigned collast
);
```

BESCHREIBUNG `getColBlock()` liefert die Teilmatrix (Block) zurück, die durch die Spalten zwischen `colfirst` und `collast-1` beschrieben wird

`colfirst` Index der ersten Spalte, die in die resultierende Matrix aufgenommen werden soll

`collast` Index der letzten Spalte, die nicht mehr in die resultierende Matrix aufgenommen werden soll

BEMERKUNGEN Es muß `colfirst` < `collast` gelten

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `getRow()` – Blockweiser Elementzugriff

SYNOPSIS

```
Matrix<T> Matrix<T>::getRow(unsigned row);
```

BESCHREIBUNG `getRow()` liefert die Zeile `row` als Matrix zurück

`row` Index der gewünschten Zeile

BEMERKUNGEN Auf die Zeile $a_{1\bullet}$ wird mit `A.getRow(0)` zugegriffen

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `getRowBlock()` – Blockweiser Elementzugriff

SYNOPSIS

```
Matrix<T> Matrix<T>::getRowBlock(
  unsigned rowfirst, unsigned rowlast
);
```

BESCHREIBUNG `getRowBlock()` liefert die Teilmatrix (Block) zurück, die durch die Zeilen zwischen `rowfirst` und `rowlast-1` beschrieben wird

`rowfirst` Index der ersten Zeile, die in die resultierende Matrix aufgenommen werden soll

`rowlast` Index der letzten Zeile, die nicht mehr in die resultierende Matrix aufgenommen werden soll

BEMERKUNGEN Es muß `rowfirst` < `rowlast` gelten

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `getStrikedOut()` – Matrix ohne Zeile und Spalte

SYNOPSIS

```
Matrix<T> Matrix<T>::getStrikedOut(
  unsigned row, unsigned col
);
```

BESCHREIBUNG `getStrikedOut()` liefert die Matrix zurück, die entsteht, wenn die Zeile `row` und die Spalte `col` gestrichen werden

`row` Index der Zeile die gestrichen werden soll

`col` Index der Spalte die gestrichen werden soll

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `getWithoutCol()` – Matrix ohne Spalte

SYNOPSIS

```
Matrix<T> Matrix<T>::getWithoutCol(unsigned col);
```

BESCHREIBUNG `getWithoutCol()` liefert die Matrix zurück, die entsteht, wenn die Spalte `col` gestrichen wird

`col` Index der Spalte, die gestrichen werden soll

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutRow()`

NAME `getWithoutRow()` – Matrix ohne Zeile

SYNOPSIS

```
Matrix<T> Matrix<T>::getWithoutRow(unsigned row);
```

BESCHREIBUNG `getWithoutRow()` liefert die Matrix zurück, die entsteht, wenn die Zeile `row` gestrichen wird

`row` Index der Zeile, die gestrichen werden soll

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`

NAME `hcat()` – Horizontale Konkatenation

SYNOPSIS

```
Matrix<T> Matrix<T>::hcat(Matrix<T> A);
```

BESCHREIBUNG `hcat()` liefert das Ergebnis der horizontalen Konkatenation mit der Matrix `A`

`A` Matrix die 'rechts' angehängt werden soll.

BEMERKUNGEN Der Operator | kann alternativ verwendet werden

SIEHE AUCH `vcat()`

BEISPIEL `X.hcat(A)` liefert $(X\ ,\ A)$ zurück

NAME `inverse()` – Inverse

SYNOPSIS

```
Matrix<T> Matrix<T>::inverse();
```

BESCHREIBUNG `inverse()` bildet die Inverse der Matrix

SIEHE AUCH `cinverse()`

NAME `kronecker()` – Kroneckerprodukt

SYNOPSIS

```
Matrix<T> Matrix<T>::kronecker(Matrix<T> A);
```

BESCHREIBUNG `kronecker()` bildet das Kroneckerprodukt mit der Matrix `A` $(X \otimes A)$

`A` zweite Matrix

NAME `prettyPrint()` – Lesefreundliche Ausgabe in Stream

SYNOPSIS

```
void Matrix<T>::prettyPrint(ostream& out);
```

BESCHREIBUNG `prettyPrint()` schreibt die Matrix in den angegebenen Stream

`out` Stream in den geschrieben werden soll

SIEHE AUCH `prettyScan()`, `print()`

NAME `prettyScan()` – Einlesen in einem lesefreundlichem Format

SYNOPSIS

```
Matrix<T> Matrix<T>::prettyScan(ostream& in);
```

BESCHREIBUNG `prettyScan()` liest die Matrix aus dem angegebenen Stream

`in` Stream aus dem gelesen werden soll

BEMERKUNGEN Das Lesen in dem für einen Menschen leicht erfassbaren Format ist sowohl aufwendiger, als auch weniger sicher als das Lesen aus dem mit dem Stream-Einfügeoperator erzeugten Format. Zum Speichern von Zwischenergebnissen oder zur Ausgabe von Daten, die durch ein Programm auf der Basis dieser Bibliothek weiterverarbeitet werden, ist deshalb die Verwendung der Stream-Operatoren ratsam. `prettyPrint()` und `prettyScan()` sollten nur für die Ein- und Ausgabe von Daten verwendet werden, die von Benutzern oder Fremdprogrammen gelesen oder geschrieben werden.

SIEHE AUCH `prettyPrint()`, `print()`

NAME `print()` – Ausgabe in Stream

SYNOPSIS

```
void Matrix<T>::print(ostream& out, char* delimiter);
```

BESCHREIBUNG `print()` schreibt die Matrix in den angegebenen Stream

`out` Stream in den geschrieben werden soll

`delimiter` einzelnes Zeichen bzw. Zeichenkette als Spaltentrennzeichen

BEMERKUNGEN Das Zeilenende wird durch `endl` und nicht durch `delimiter` definiert

SIEHE AUCH `prettyPrint()`, `prettyScan()`

BEISPIEL Sei $X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$, so schreibt `X.print(out, "&")` in den Stream `1&2` `3&4`

NAME `put()` – Elementzugriff

SYNOPSIS

```
void Matrix<T>::put(unsigned row, unsigned col, T value);
```

BESCHREIBUNG `put()` setzt das Element x_{ij} auf den Wert `value`

`row` Zeilenindex des neu zu setzenden Elements

`col` Spaltenindex des neu zu setzenden Elements

`value` Wert des neu zu setzenden Elements

BEMERKUNGEN Das Element x_{11} wird mit `X.put(0,0,<value>)` neu besetzt

SIEHE AUCH `operator()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `putBlock()` – Blockweiser Elementzugriff

SYNOPSIS

```
void Matrix<T>::putBlock(
  Matrix<T> A, unsigned rowfirst, unsigned colfirst,
  unsigned rowlast, unsigned collast
);
```

BESCHREIBUNG `putBlock()` ersetzt den Block der durch die Zeilen `rowfirst` und `rowlast-1` und die Spalten `colfirst` und `collast-1` gegeben ist durch die Werte, die in der Matrix `A` gegeben sind

`A` Matrix die die neuen Werte enthält

`rowfirst` Erste Zeile des neu zu setzenden Blocks

`colfirst` Erste Spalte des neu zu setzenden Blocks

`rowlast-1` Letzte Zeile des neu zu setzenden Blocks

`collast-1` Letzte Spalte des neu zu setzenden Blocks

BEMERKUNGEN Es muß `rowfirst` < `rowlast` und `colfirst` < `collast` gelten. Die Parameter `collast` und `rowlast` sind optional. Werden `collast` und `rowlast` nicht angegeben, so muß die Matrix `A` die Dimension (`X.rows()`−`rowfirst`) ×

(`X.cols()−colfirst`) oder kleiner haben. Die Matrix `A` wird dann ausgehend vom Element `(colfirst,rowfirst)` in die Matrix `X` eingesetzt. Werden die Parameter `collast` und `rowlast` angegeben, so darf die Matrix `A` die Dimension (`rowlast−rowfirst`) × (`collast−colfirst`) nicht unterschreiten. Es wird dann der Teil (0,0) bis (`rowlast−rowfirst`) × (`collast−colfirst`) aus der Matrix `A` ausgeschnitten und in die Matrix `X` eingesetzt.

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `putCol()` – Spaltenweiser Elementzugriff

SYNOPSIS

```
void Matrix<T>::putCol( unsigned col, Matrix<T> A );
```

BESCHREIBUNG `putCol()` setzt die Werte in der angegebenen Spalte neu
`col` Index der Spalte, deren Werte neu gestezt werden sollen
`A` Matrix mit Werten für die Elemente der zu setzenden Spalte

BEMERKUNGEN Die Matrix `A` muß die Dimension `X.rows()` × 1 haben

SIEHE AUCH `put()`, `get()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `putColBlock()` – Blockweiser Elementzugriff

SYNOPSIS

```
void Matrix<T>::putColBlock(
  unsigned colfirst, unsigned collast, Matrix<T> A
);
```

BESCHREIBUNG `putColBlock()` setzt die Werte in dem Block der durch `colfirst` und `collast-1` gegeben ist neu
`colfirst` Index der ersten Spalte des neu zu besetzenden Blocks
`collast-1` Index der letzten Spalte des neu zu besetzenden Blocks
`A` Matrix mit Werten für die Elemente des neu zu setzenden Blocks

BEMERKUNGEN Die Matrix `A` muß die Dimension `X.rows` × (`collast−colfirst`) haben. Es muß `colfirst` < `collast` gelten

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `putRow()` – Zeilenweiser Elementzugriff

SYNOPSIS

```
void Matrix<T>::putRow(unsigned row, Matrix<T> A);
```


BESCHREIBUNG `putRow()` setzt die Werte in der angegebenen Zeile neu
`row` Index der Zeile, deren Werte neu gestezt werden sollen
`A` Matrix mit Werten für die Elemente der zu setzenden Zeile

BEMERKUNGEN Die Matrix `A` muß die Dimension (1 × `X.cols` haben

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `getRow()`, `putColBlock()`, `getColBlock()`, `putRowBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `putRowBlock()` – Blockweiser Elementzugriff

SYNOPSIS

```
void Matrix<T>::putRowBlock(
  unsigned rowfirst, unsigned rowlast, Matrix<T> A
);
```

BESCHREIBUNG `putRowBlock()` setzt die Werte in dem durch `rowfirst` und `rowlast-1` angegebenen Block neu
`rowfirst` Index der ersten Zeile des neu zu besetzenden Blocks
`rowlast-1` Index der letzten Zeile des neu zu besetzenden Blocks
`A` Matrix mit Werten für die Elemente des neu zu setzenden Blocks

BEMERKUNGEN Die Matrix `A` muß die Dimension (`rowlast−rowfirst`) × `X.cols` haben. Es muß `rowfirst` < `rowlast` gelten

SIEHE AUCH `put()`, `get()`, `putCol()`, `getCol()`, `putRow()`, `getRow()`, `putColBlock()`, `getColBlock()`, `getRowBlock()`, `getStrikedOut()`, `putBlock()`, `getBlock()`, `getWithoutCol()`, `getWithoutRow()`

NAME `root()` – Wurzel

SYNOPSIS

```
Matrix<T> Matrix<T>::root();
```

BESCHREIBUNG `root()` liefert die Cholesky Wurzel der Matrix

NAME `rows()` – Anzahl der Zeilen

SYNOPSIS

```
unsigned Matrix<T>::rows();
```

BESCHREIBUNG `rows()` liefert die Anzahl der Zeilen der Matrix zurück

BEMERKUNGEN Die letzte Zeile hat die Index `rows()`−1

SIEHE AUCH `cols()`

NAME `solve()` – Gleichungssystem lösen

SYNOPSIS

```
Matrix<T> Matrix<T>::solve(const Matrix<T>& b);
```

BESCHREIBUNG `solve()` löst das Gleichungssystem $Ax = b$
`b` Matrix (Spaltenvektor)

BEMERKUNGEN Kann z.B. verwendet werden um die Schätzung $\hat{\beta} = (X'X)^{-1}X'y$ mit `(X.sscp()).solve(X.transposed()*y)` zu berechnen

SIEHE AUCH `csolve()`

NAME `sscp()` – Sum of Squares and Cross Products

SYNOPSIS

```
Matrix<T> Matrix<T>::sscp();
```

BESCHREIBUNG `sscp()` Berechnet die Matrix $X'X$

NAME `symmetric()` – Prüfung auf Symmetrie

SYNOPSIS

```
bool Matrix<T>::symmetric(T epsilon = T(0));
```

BESCHREIBUNG `symmetric()` prüft, ob die Matrix symmetrisch ist
`epsilon` absoluter Wert der Abweichung von der strengen Symmetrie, Vorgabewert ist 0

BEISPIEL Die Matrix $X = \begin{pmatrix} 1 & 1.004 \\ 1 & 2 \end{pmatrix}$ wird mit `X.symmetric(0.01)` als symmetrisch angesehen, bei `X.symmetric()` ist die Matrix nicht symmetrisch.

NAME `trace()` – Spur

SYNOPSIS

```
T Matrix<T>::trace();
```

BESCHREIBUNG `trace()` berechnet die Spur (Summe der Diagonalelemente) der Matrix

BEMERKUNGEN Quadratische Matrix gefordert

NAME `transposed()` – Transponierte

SYNOPSIS

```
Matrix<T> Matrix<T>::transposed();
```

BESCHREIBUNG `transposed()` liefert die Transponierte X' der Matrix

NAME `vcat()` – Vertikale Konkatenation

SYNOPSIS

```
Matrix<T> Matrix<T>::vcat();
```

BESCHREIBUNG `vcat()` liefert das Ergebnis der vertikalen Konkatenation mit der Matrix `A`

`A` Matrix die 'unten' angehängt werden soll.

BEMERKUNGEN Der Operator & kann alternativ verwendet werden

SIEHE AUCH `hcat()`

BEISPIEL `X.vcat(A)` liefert $\begin{pmatrix} X \\ A \end{pmatrix}$ zurück

---

NAME `vec()` – Vektorisierung

SYNOPSIS

```
Matrix<T> Matrix<T>::vec();
```

BESCHREIBUNG `vec()` liefert die Matrix als Spaltenvektor zurück

BEMERKUNGEN Die Dimension der Ergebnismatrix ist `X.rows()`·`X.cols()` × 1

SIEHE AUCH `vech()`

BEISPIEL Sei $X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ dann liefert `X.vec()` den Spaltenvektor
$(1\ 4\ 2\ 5\ 3\ 6)'$

NAME `vech()` – Vektorisierung

SYNOPSIS

```
Matrix<T> Matrix<T>::vech();
```

BESCHREIBUNG `vech()` liefert die Matrix als Zeilenvektor zurück

BEMERKUNGEN Die Dimension der Ergebnismatrix ist 1 × `X.rows()`·`X.cols()`

SIEHE AUCH `vec()`

BEISPIEL Sei $X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ dann liefert `X.vech()`: den Zeilenvektor
$(1\ 2\ 3\ 4\ 5\ 6)$

NAME `zero()` – Matrixelemente gleich 0

SYNOPSIS

```
bool Matrix<T>::zero(T epsilon = T(0));
```

BESCHREIBUNG `zero()` prüft, ob die Matrixelemente gleich 0 sind `epsilon` absoluter Wert der Abweichung von 0.

BEISPIEL Die Matrix $X = \begin{pmatrix} 0 & 0.004 \\ 0 & 0 \end{pmatrix}$ wird mit `X.zero(0.01)` als gleich 0 angesehen, bei `X.zero()` ist die Matrix ungleich 0.

3.2 Statische Funktionen für Matrizen

Neben den Methoden, die auf eine Instanz einer Klasse anwendbar sind, kennt das objektorientierte Paradigma auch solche Methoden, die auf die Metaklasse dieser Klasse angewendet werden. Eine andere Sichtweise, die aber vom Ergebnis her keine andere Bedeutung hat, ist, daß solche Methoden Methoden „der Klasse selbst“ sind. In C++ werden solche „Klassenmethoden“ oder Methoden der Metaklasse einer Klasse durch statische Funktionen innerhalb der Klasse dargestellt. Diese sind an die Klasse gebunden, und haben wie andere Methoden, die auf eine Instanz der Klasse anwendbar sind, Zugriff auf die „Interna“ der Implementation. Sie existieren aber nicht im globalen Namensraum, sondern gehören zum Namensraum der Klasse, und müssen daher durch einen qualifizierten Bezeichner aufgerufen werden. Wenn `foo()` eine statische Funktion der Klasse `Bar` ist, wird sie durch `Bar::foo()` aufgerufen. Nachdem Klassenmethoden keine Instanz der Klasse benötigen, durch die sie aufgerufen werden, „beziehen“ sie sich nicht auf eine bestimmte Instanz.

NAME `diag()` – Diagonalmatrix

SYNOPSIS

```
Matrix<T> Matrix<T>::diag(unsigned int dim, value <T>);
```

BESCHREIBUNG `diag()` bildet eine Diagonalmatrix deren Elemente x_{ii} den Wert `value` besitzen

`dim` Dimension der Matrix

`value` Vorgabewert für die Diagonalelemente

NAME `diag()` – Diagonalmatrix

SYNOPSIS

```
Matrix<T> Matrix<T>::diag(Matrix<T> values);
```

BESCHREIBUNG `diag()` bildet eine Diagonalmatrix deren Elemente x_{ii} die Werte besitzen, die in der Matrix `values` angegeben sind

`values` Spaltenvektor mit den gewünschten Werten der Diagonalmatrix

BEMERKUNGEN Die Resultierende Matrix hat die Dimension `values.rows()` × `values.rows()`

NAME `tridiag()` – Tridiagonalmatrix

SYNOPSIS

```
Matrix<T> Matrix<T>::tridiag(Matrix<T> diag,
                             Matrix<T> upper, Matrix<T> lower);
```

BESCHREIBUNG `tridiag()` bildet eine Diagonalmatrix aus den Spaltenvektoren `diag`, `upper` und `lower`. Die Werte von `diag` werden auf der Winkelhalbierenden eingetragen, die Werte von `upper` werden auf der Diagonalen oberhalb der Winkelhalbierenden, die Werte von `lower` auf der Diagonalen unterhalb der Winkelhalbierenden eingetragen, die restlichen Elemente sind ‘0’

`diag` Spaltenvektor mit den gewünschten Werten der Winkelhalbierenden

`upper` Spaltenvektor mit den gewünschten Werten oberhalb der Winkelhalbierenden

`lower` Spaltenvektor mit den gewünschten Werten unterhalb der Winkelhalbierenden

BEMERKUNGEN Die Resultierende Matrix hat die Dimension `diag.rows()` × `diag.rows()`

BEISPIEL Sei $A = (1\ 2\ 3)'$, $B = (4\ 5)'$ und $C = (6\ 7)'$, dann liefert

```
Matrix<float> A, B, C, D;
/* Matrizen */
D = Matrix<float>::tridiag(A,B,C);
```

die Matrix $D = \begin{pmatrix} 1 & 4 & 0 \\ 6 & 2 & 5 \\ 0 & 7 & 3 \end{pmatrix}$

3.3 Funktionen die Interna von Matrizen kennen

Funktionen, die durch einen bevorzugten Zugriff auf Interna von Matrizen ausgezeichnet sind, erlauben es, einfache Algorithmen effizienter zu implementieren. Sie werden allerdings Bestandteil der Schnittstelle der parametrischen Klassen. Diese starke Abhängigkeit von externen Funktionen und der Implementation der parametrischen Klassen dient dazu, syntaktische Schwierigkeiten auszuräumen und die Grenze der Datenkapselung an wenigen ausgesuchten Stellen zu durchbrechen.

NAME `<<` – Einfügeoperator für Streams

SYNOPSIS

```
ostream& operator<< (ostream& os, const Matrix<T>& ob);
```

BESCHREIBUNG `<<` fügt eine Matrix in einem Format in einen Stream ein

`os` gültiger Output-Stream

`ob` Objekt, das eingefügt werden soll

BEMERKUNGEN Der Operator `<<` ermöglicht die für C++ Streams typische Ausgabe

SIEHE AUCH `>>`

BEISPIEL

```
Matrix<float> Y(2,2,0.0F);
cout << Y;
```

NAME >> – Extraktionsoperator für Streams

SYNOPSIS

```
istream& operator>> (istream& is, Matrix<T>& ob);
```

BESCHREIBUNG >> liest eine mit << geschriebene Matrix aus einem Stream

`is` gültiger Input-Stream

`ob` Objekt in das das Ergebnis eingefügt werden kann

BEMERKUNGEN Der Operator >> ermöglicht die für C++ Streams typische Eingabe. Der Input-Stream kann bei Fehlern in der Eingabe ungültig werden. Dies kann mit der Methode `bad()` des Input-Streams abgeprüft werden

SIEHE AUCH <<

BEISPIEL

```
Matrix<float> Y;
cin >> Y;
```

NAME * – Skalarmultiplikation

SYNOPSIS

```
Matrix<T> operator* (T value, const Matrix<T>& Y);
```

BESCHREIBUNG * leistet die Skalarmultiplikation von links

`value` Skalar

`Y` Matrix

SIEHE AUCH >>

3.4 Spezielle Funktionen für Real-Matrizen

Die Instantiierung mit einem Fließkommatyp ermöglicht einige zusätzliche Operationen die nur für Fließkommazahlen sinnvoll sind, oder für diese deutlich einfacher implementiert werden können, so daß auf eine allgemeinere Implementation verzichtet wurde. Die Spezialisierung von `Matrix<real>` mit einem Benutzerdefinierten Typ `real` heißt `realMatrix` und ist ein von `Matrix<real>` abgeleiteter Typ.

NAME `containsNaNs()` – Prüfung auf 'NotANumber'

SYNOPSIS

```
bool realMatrix::containsNaNs();
```

BESCHREIBUNG `containsNaNs()` durchsucht die Matrix nach vorhandenen IEEE-NotANumber Ausnahmen. Diese entstehen, wenn eine Operation nicht definiert ist und auch kein „sinnvoller" Ersatzwert wie z.B. $-\infty$ oder $+\infty$ existiert

BEMERKUNGEN Eine IEEE-NotANumber Ausnahme tritt z.B. bei einer Division von Null durch Null auf

---

NAME `eigenvalSym()` – Eigenwerte

SYNOPSIS

```
realMatrix realMatrix::eigenvalSym();
```

BESCHREIBUNG `eigenvalSym()` liefert die Eigenwerte einer symmetrischen Matrix als Spaltenvektor zurück.

BEMERKUNGEN Die Matrix X muß dabei symmetrisch im strengen Sinn sein.

NAME `eigenvecSym()` – Eigenvektoren

SYNOPSIS

```
realMatrix realMatrix::eigenvecSym();
```

BESCHREIBUNG `eigenvecSym()` berechnet die Eigenvektoren einer symmetrischen Matrix

BEMERKUNGEN Die Matrix X muß dabei symmetrisch im strengen Sinn sein.

---

4 Erzeugen von Zufallszahlen

4.1 Pseudozufallszahlen

Random numbers are to important to be left to chance. *(Aus einer Signatur im Usenet)*

John v. Neumann hat bereits festgestellt, daß sich jeder, der einen Algorithmus zur Erzeugung von Zufallszahlen entwirft, im Zustand der Sünde befindet. Das Ziel eines Verfahrens zur Erzeugung von Pseudo-Zufallszahlen kann es immer nur sein, einen bestimmten Aspekt des Zufalls möglichst gut nachzubilden, so beispielsweise die Unabhängigkeit zweier in einem bestimmten Abstand erzeugter Zahlen, oder deren Symmetrie.

Neben dem Grundproblem des Erzeugens von Zufallszahlen mit den für den angestrebten Zweck erforderlichen Eigenschaften gibt es auch eine Reihe von praktischen Problemen, die bei der Verwendung von Pseudozufallszahlen im Rahmen einer Simulation ebenfalls behandelt werden müssen. Ein Verfahren zur Erzeugung von Pseudo-Zufallszahlen wird dabei als Ausgangspunkt für weitere Verfahren zur Erzeugung von Pseudo-Zufallszahlen aus bestimmten Verteilungen verwendet.

Während die Auswahl eines geeigneten Verfahrens zur Erzeugung von Pseudo-Zufallszahlen, die eine Gleichverteilung auf dem Intervall $[0;1]$ nachbilden, sehr

von der Laufzeitumgebung oder im Idealfall auch von der vorhandenen Hardware abhängig ist, kann die weniger anspruchsvolle Aufgabe der Transformation solcher Pseudo-Zufallszahlen in Zufallszahlen, die eine bestimmte Verteilung nachbilden, auch von einer portablen Bibliothek übernommen werden.

4.2 Abstrakte Basisklassen

Das Paradigma der objektorientierten Programmierung bietet das Konzept einer *abstraken Basisklasse* an, um einen benutzerdefinierten Typ zu spezifizieren, der einzelne Aspekte seiner Implementation vollständig an andere benutzerdefinierte Typen delegiert, die den abstrakten Typ konkretisieren. Der Vorteil einer solchen abstrakten Basisklasse ist es, daß konkrete Klassen, die eine solche

Basisklasse spezialisieren, die an sie deligierten Aspekte nicht nur konkretisieren dürfen, sondern sogar konkretisieren müssen, wenn der speziellere benutzerdefinierte Typ instantiierbar sein soll.

Auf diese Weise läßt sich sehr gut ausdrücken, daß die Klasse zur Erzeugung von Pseudo-Zufallszahlen nur die portabel implementierbaren Werkzeuge, nicht aber den Algorithmus zur Erzeugung von Zufallszahlen bereitstellt. Dieser muß vom Anwender nach den Möglichkeiten gewählt werden, die seine Laufzeitumgebung oder seine Hardware bietet.

4.3 Die abstrakte Klasse `RNG`

Auch die abstrakte Basisklasse der Pseudo-Zufallszahlengeneratoren ist als parameterische Klasse implementiert. Als aktuelle Parameter für den formalen Typ-Parameter `T` eines Zufallszahlengenerators kommen jedoch nur solche Typen in Frage, für die die Funktionen der C-Standardbibliothek implementiert sind. Ohne zusätzliche eigene Unterstützung ist dies in älteren Implementation der Typ `double`, in neueren Implementationen jeder Fließkommatyp.

Rein virtuelle Methoden

Die Klasse `RNG` kann selbst nicht instantiiert werden, weil sie rein *rein virtuelle* Methoden besitzt. Die rein virtuelle Methode `double rndu()` soll eine Pseudo-Zufallszahl aus einer Gleichverteilung auf dem Intervall $[0;1]$ liefern. Die rein virtuelle Methode `void initrnd(int seed = 0)` soll den Zufallszahlengenerator initialisieren, wobei ein Wert von `0` für `seed` bedeutet, daß ein zufälliger Startwert gewählt werden soll, der aus der Systemzeit oder einer anderen externen Quelle abgeleitet ist. Andere Werte für `seed` sollen einen definierten, reproduzierbaren Anfangszustand liefern.

Eine Spezialisierung dieses abstrakten Zufallszahlengenerators mit Hilfe der C++-Standardbibliothek könnte für einen Compiler, der das letzte öffentlich zugängliche Draft Working Paper realisiert, etwa so aussehen:

```
#include "rng.h"
#inlcude <cstdlib>
#include <climits>
#include <ctime>

template <class T>
class CRNG : public RNG<T>
{
   virtual double rndu()
   { return static_cast<T>(rand())/static_cast<T>(RAND_MAX); }
protected:
   virtual void initrnd(int seed = 0);
};

inline void CRNG::initrnd(int seed)
{
   if (seed == 0) {
      time_t tm;
      struct tm *pt;

      time(&tm);
      pt = localtime(&tm);
      seed = ptm->tm_sec + 60 * ptm->tm_min;
   }
   srand(seed);
}
```

Die Anforderungen an die Implementation von `rand()` in der C++-Standardbibliothek sind minimal. Ein solcher Generator hängt also sehr von der Qualität der Implementation ab. Wenn die Laufzeitumgebung einen zuverlässigen Generator für Pseudo-Zufallszahlen bereitstellt, sollte diese „Standardlösung“ höchstens dann verwendet werden, wenn die Portabilität des Quelltextes die entscheidende Forderung für die geplante Anwendung ist.

Methoden, die Pseudo-Zufallszahlen oder Pseudo-Zufallsfelder liefern

---

NAME `contunif()` – stetige Gleichverteilung

SYNOPSIS

```
T contunif(T a, T b);
```

BESCHREIBUNG `contunif()` Liefert eine Pseudo-Zufallszahl aus einer Gleichverteilung auf dem Intervall $[a; b]$.

`a` Untere Grenze des Intervalls

`b` Obere Grenze des Intervalls

NAME `contunif()` – stetige Gleichverteilung

SYNOPSIS

```
void contunif(Array<T> &m, T a, T b);
```

BESCHREIBUNG `contunif()` Füllt `m` mit Pseudo-Zufallszahlen aus einer Gleichverteilung auf dem Intervall $[a; b]$.

`m` Array der mit Zufallszahlen gefüllt werden soll

`a` Untere Grenze des Intervalls

`b` Obere Grenze des Intervalls

NAME `discunif()` – diskrete Gleichverteilung

SYNOPSIS

```
T discuinf(T l, T u);
```

BESCHREIBUNG `discunif()` Liefert eine Pseudo-Zufallszahl aus einer diskreten Gleichverteilung auf $\{l, l+1, \ldots, u-1, u\}$.

`u` Untere Grenze des Intervalls

`l` Obere Grenze des Intervalls

NAME `discunif()` – diskrete Gleichverteilung

SYNOPSIS

```
void discunif(Array<T> &m, T l, T u);
```

BESCHREIBUNG `discunif()` Füllt `m` mit Pseudozufallszahlen aus einer diskreten Gleichverteilung auf $\{l, l+1, \ldots, u-1, u\}$.

`m` Array der mit Zufallszahlen gefüllt werden soll

`u` Untere Grenze des Intervalls

`l` Obere Grenze des Intervalls

NAME `gamma()` – Gammaverteilung

SYNOPSIS

```
T gamma(T a);
```


BESCHREIBUNG `gamma()` Liefert eine Pseudo-Zufallszahl aus einer Gammaverteilung mit Parameter `a`.

`a` Parameter der Gammaverteilung

NAME `gamma()` – Gammaverteilung

SYNOPSIS

```
void gamma(Array<T> &m, T a);
```

BESCHREIBUNG `gamma()` Füllt `m` mit Pseudo-Zufallszahlen aus einer Gammaverteilung mit Parameter `a`.

`m` Array der mit Zufallszahlen gefüllt werden soll

`a` Parameter der Gammaverteilung

NAME `normal()` – Normalverteilung

SYNOPSIS

```
T normal(T mu, T sigmasq);
```

BESCHREIBUNG `normal()` Liefert eine Pseudo-Zufallszahl aus einer Normalverteilung mit Erwartungswert `mu` und Varianz `sigmasq`.

`mu` Erwartungswert μ

`sigmasq` Varianz σ^2

NAME `normal()` – Normalverteilung

SYNOPSIS

```
void normal(Array<T> &m, T mu, T sigmasq);
```

BESCHREIBUNG `normal()` Füllt `m` mit Pseudo-Zufallszahlen aus einer Normalverteilung mit Erwartungswert `mu` und Varianz `sigmasq`.

`m` Array der mit Zufallszahlen gefüllt werden soll

`mu` Erwartungswert μ

`sigmasq` Varianz σ^2

WARNUNGEN Im Gegensatz zu der sonst üblichen Notation wird hier zuerst die Varianz angegeben. Dadurch kann der häufige Fall einer Normalverteilung mit Erwartungswert Null und beliebiger Varianz über Defaultparameter behandelt werden.

NAME `normal()` – Normalverteilung

SYNOPSIS

```
void normal(Array2D<T> &m,
            const PreMatrix<T> &mu, const Matrix<T> &S);
```

BESCHREIBUNG `normal()` Füllt `m` mit Pseudozufallszahlen aus einer multivariaten Normalverteilung mit Erwatungswertvektor `mu` und Kovarianzmatrix `S`.

`m` Array der mit Zufallszahlen gefüllt werden soll

`mu` Erwartungswertvektor $\boldsymbol{\mu}$
`S` Kovarianzmatrix $\mathbf{S}$

BEMERKUNGEN `m` hat die gleiche Spaltendimension wie `mu`, und `S` ist eine quadratische Matrix dieser Dimension, die positiv definit ist.

NAME `poisson()` – Poissonverteilung

SYNOPSIS

```
T poisson(T lambda);
```

BESCHREIBUNG `poisson()` Liefert eine Pseudo-Zufallszahl aus einer Poissonverteilung mit Parameter `lambda`.
`lambda` Parameter λ der Poissonverteilung

BEMERKUNGEN Wenn mehrere Zufallszahlen aus der gleichen Poissonverteilung benötigt werden, ist die erste überladene Methode deutlich effizienter.

NAME `poisson()` – Poissonverteilung

SYNOPSIS

```
void poisson(Array<T> &m, T lambda);
```

BESCHREIBUNG `poisson()` Füllt `m` mit Pseudo-Zufallszahlen aus einer Poissonverteilung mit Parameter `lambda`.
`m` Array der mit Zufallszahlen gefüllt werden soll
`lambda` Parameter λ der Poissonverteilung

---

5 Beispiele

Normalanwendung: Matrixmultiplikation mit 'einfach' instantiierter `matrix`. Es wird die Matrix `X` von der Standardeingabe und die Matrix `y` aus der Datei `foo.asc` eingelesen. Die Matrizen werden dann miteinander multipliziert und schließlich wird das Ergebnis auf die Standardausgebe geschrieben.

```
#include <bool.h>
#include <matrix.h>
#include <iostream.h>
#include <fstream.h>

int main() {
  matrix X, y;
  ifstream in("foo.asc")
```



```
  X.prettyScan(cin);
  y.prettyScan(in);
  (X * y).prettyPrint(cout);
  cout << endl;
  return 0;
}
```

Vergleichsoperator: Test auf Gleichheit von Arrays. Mit `[4] { A b c d }` wird ein `Array<char>` der Länge 4 angegeben:
Eingabe: `[4] { F r e d } [4] { F r e d }`
Erwartete Ausgabe: `gleich`
Eingabe: `[4] { F r e d } [4] { F r e y }`
Erwartete Ausgabe: `verschieden`

```
#include <bool.h>
#include <matrix.h>
#include <iostream.h>
#include <tarray.h>

int main() {
  Array<char> lineA, lineB;

  cin >> lineA >> ws >> lineB;
  if (lineA == lineB)
    cout << "gleich" << endl;
  else
    cout << "verschieden" << endl;
  return 0;
}
```

Elementweise Anwendung: Eine zweistellige Funktion wird auf zwei Matrizen gleicher Dimension angewendet. Hier wird die Funktion `power(a,b)`, die das Ergebnis a^b als `float` (durch Aufruf der für `double` definierten Funktion `pow(a,b)`) liefert betrachtet.

```
#include <bool.h>
#include <matrix.h>
#include <iostream.h>
#include <math.h>

float power(float a, float b) {
  return float(pow(double(a), double(b)));
}
```

```
int main() {
  Array2D<float> A(2, 2), B(2,2);

  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j) {
      A(i, j) = float(i * 2 + j + 1);
      B(i, j) = 1 / A(i, j);
    }
  Array2D<float> C = A.applied(B, power);
  cout << A << "pow" << B << " = " << C << endl;
  return 0;
}
```

Zeilen- und spaltenweise Operationen: Zugriff und Ausgabe von Elementen der von Array2D abgeleiteten Klasse `PreMatrix`.

```
#include <bool.h>
#include <matrix.h>
#include <iostream.h>

int main() {
  typedef PreMatrix<float> mat;

  mat A(4,4);
  for (unsigned int i = 0; i < 16; ++i)
    A(i % 4, i / 4) = float(i);
  A.prettyPrint(cout);
  cout << endl;
  A.getRow(0).prettyPrint(cout);
  cout << endl;
  A.getCol(3).prettyPrint(cout);
  cout << endl;
  A.putCol(2, A.getCol(1));
  A.prettyPrint(cout);
  cout << endl;
  return 0;
}
```

Einfache Listenanwendung: Einfügen von Elementen in die Liste und Durchlaufen der Liste (vorwärts) mit einem Iterator.

```
#include <bool.h>
#include <matrix.h>
#include <iostream.h>
#include <tlinklst.h>
```



```
const unsigned int upperBound = 128;

int main() {
  List<int> A;

  for (unsigned int i = 0; i < upperBound; ++i)
    A.insert(i + 1);
  ListIterator<int> p = A;
  while(p) {
    cout << p() << endl;
    ++p;
  }
  return 0;
}
```

Diagonalmatrix: Einfügen und Extrahieren von Diagonalmatrizen (zu beachten ist der unterschiedliche Aufruf und die verschiedene Bedeutung der Elementfunktion `A.diag()` und der statischen Funktion `mat::diag(B)`).

```
#include <bool.h>
#include <matrix.h>
#include <iostream.h>

int main() {
  typedef PreMatrix<float> mat;
  mat X(4, 4);
  mat A = mat::diag(5, 1.0F) * 3.14F;
  mat B = A.diag();
  mat z(1,1, 0.5F);

  A.prettyPrint(cout);
  cout  << endl << endl;
  (A - mat::diag(B)).prettyPrint(cout);
  cout << endl;
  mat::diag(B.transposed()).prettyPrint(cout);
  cout << endl;
  mat::diag(z).prettyPrint(cout);
  return 0;
}
```

Horizontale und vertikale Konkatenation von Matrizen.

```
#include <bool.h>
#include <matrix.h>
```

```
#include <iostream.h>

typedef PreMatrix<float> mat;

int main() {
  mat X(5, 1, 1.0F);

  for (unsigned int j = 2; j < 10; ++j) {
    mat b(X.rows(), 1, float(j));
    X = X | b;
  }
  X.prettyPrint(cout);
  for (unsigned int i = 10; i < 15; ++i) {
    mat c(1, X.cols(), float(i));
    X = X & c;
  }
  X.prettyPrint(cout);
  return 0;
}
```

Einfache Matrixarithmetik: Lösen eines linearen Gleichungssystems der Form $Ax = b$.

```
#include <bool.h>
#include <matrix.h>
#include <iostream.h>

typedef PreMatrix<arithm> mat;

int main() {
   mat A, b, x;

   A.prettyScan(cin);
   b.prettyScan(cin);
   x = A.solve(b);
   x.prettyPrint(cout);
   cout << endl;
   return 0;
}
```

Vergleich verschiedener Algorithmen zur Inversion positiv definiter Matrizen ($S = X'X$ ist positiv definit).

```
#include <bool.h>
#include <matrix.h>
```



```
#include <iostream.h>
#include <math.h>
#include <float.h>

typedef Matrix<double> matrix;

matrix& trunc(matrix& m, double limit) {
  unsigned int i, j;

  for (i = 0; i < m.rows(); ++i)
    for (j = 0; j < m.cols(); ++j)
      m.put(i, j, fabs(m.get(i, j)) > limit ? 1.0 : 0.0);
  return m;
}

int main() {
  matrix X;

  X.prettyScan(cin);
  matrix S = X.sscp();
  matrix I = S.inverse();
  matrix J = S.cinverse();
  matrix temp = I - J;
  matrix K = trunc(temp, sqrt(DBL_EPSILON));
  K.prettyPrint(cout);
  return 0;
}
```

Lizenzbestimmungen

Die Autoren dieser Bibliothek geben jeder nicht-kommerziellen Organisation und jedem Individuum das Nutzungsrecht und das Recht, eine beliebige Anzahl von Kopien dieser Software zu erstellen. Die Bibliothek darf ohne schriftliche Genehmigung der Autoren nicht verkauft werden. Die Autoren übernehmen keine Verantwortung für Schäden, die durch die Verwendung dieser Bibliothek entstehen und geben keine Garantie dafür, daß die Bibliothek für irgendeinen speziellen Zweck geeignet ist.

Verfügbarkeit

Die hier vorgestellten Klassen und Funktionen können über ftp auf `ftp.stat.uni-muenchen.de` bezogen werden. Die Dateien befinden sich im Verzeichnis `/pub/src/libtempl`.

Anerkenntnis

Diese Arbeit ist im Sonderforschungsbereich 386, Diskrete Datenstrukturen, München, entstanden und wurde auf seine Veranlassung unter Verwendung der ihm von der Deutschen Forschungsgemeinschaft zur Verfügung gestellten Mittel gedruckt.