Kapitel 4
Lineare Algebra/mehrdimensionale Felder

4.1 Matrizen in C++

Eine Reihe von Anwendungen benötigen anstelle von eindimensionalen Feldern (*Vektoren*) zwei- (*Matrizen*) oder seltener auch drei- oder höher dimensionale Felder (*Tensoren*). Der Sprachstandard von C/C++ unterstützt zwar statische mehrdimensionale Deklarationen wie

```
double f[5][5][5]
```

aber nicht dynamische, die erst zur Laufzeit erfolgen. Außerdem muss die Bereitstellung eines kompletten mehrdimensionalen Feldes nicht immer die beste Lösung sein. Sind beispielsweise sehr viele Felder mit dem Wert Null belegt, wird unnötigerweise sehr viel Speicherplatz verbraucht. Zugriffe auf Teilbereiche von Feldern und Rechenoperationen, an denen Felder unterschiedlicher Dimension teilnehmen, vervollständigen die Aufgabenstellung

4.1.1 Normal besetzte Matrizen

Wir beginnen mit der Diskussion dynamisch deklarierter Felder und Anwendungen, wie sie vorzugsweise in der linearen Algebra auftreten.

4.1.1.1 Eine Matrixklasse

Der vermeintliche Nachteil – dynamische Deklaration mehrdimensionaler Felder – lässt sich relativ leicht durch eine Indexarithmetik beseitigen. Eine Matrix mit m Zeilen und n Spalten lässt sich auch dadurch realisieren, dass man die Zeilen (oder die Spalten) hintereinander auf einem linearen Feld anordnet. Um ein bestimmtes Matrixelement a_{ik} zu indizieren, müssen $(i - 1)$ Zeilen übersprungen und dann auf den Index k in der Zeile zugegriffen werden (alternativ lässt sich die gleiche Argumentation bei spaltenweiser Speicherung aufbauen).Wir erhalten damit folgenden Grundtyp der Matrixklasse:

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_4, © Springer-Verlag Berlin Heidelberg 2010

```
template <typename T> class Matrix {
public:
  Matrix(int n, int m): zeilen(n),spalten(m)
     {a= new T[n*m];}

  // Elementzugriffe über Indizes
  T& operator()(int i,int j)
       {return a[i*spalten+j];}

protected:
  int zeilen,spalten;
  T* a;
};//end class
```

Benötigt man mehr Dimensionen, kann in der gleichen Weise vorgegangen werden, beispielsweise.

```
Matrix(int n, int m, int k): X(n), Y(m), Z(k)
     {a= new T[n*m*k];}

T& operator()(int i,int j, int k)
     {return a[(i*X+j)*Y+k];}
```

Bei der Organisation der Daten auf dem Feld ist zu berücksichtigen, ob die Daten zu einem späteren Zeitpunkt mit anderen Anwendungen ausgetauscht werden sollen. Das zweidimensionale Beispiel organisiert die Reihenfolge der Daten zeilenweise, d.h. die Elemente einer Zeile der Matrix folgend direkt aufeinander. Alternativ kann jedoch auch eine spaltenweise Reihenfolge der Daten auf dem Feld implementiert werden. Wenn die weitere Bearbeitung von Daten mit anderen Programmen erfolgen soll, müssen Sie das Schema passend einstellen.

Aufgabe. Implementieren Sie eine Klasse mit zwei Dimensionen.[1] „Universalisieren“ Sie die Definition durch Konstruktoren und Zugriffsoperatoren für den eindimensionalen Fall (Vektoren). Beachten Sie, dass auch Methoden für `const`-Zugriffe notwendig sind.
Vergessen Sie nicht, `to_string`- und `from_string`-Methoden zu implementieren, beispielsweise in der Form

```
<matrix>
     <dim><int>...</int>....</dim>
     <row><T>...</T>...</row>
     ...
</matrix>
```

[1] Sie können auch mehr Dimensionen vorsehen, wenn Sie Anwendungen im Auge haben, die dies benötigen. Die weitere Diskussion in diesem Kapitel beschränkt sich jedoch auf zwei Dimensionen.

Anmerkung. Die meisten Algorithmen bereiten zwar wenig Probleme, da insbesondere in der linearen Algebra vieles einfach nur von der mathematischen Notation in die der Programmiersprache zu übersetzen ist. Trotzdem ist es natürlich möglich, dass man sich bei den Indizes irgendwo einmal vergriffen hat und eigenartige Ergebnisse erhält, wobei die Masse der Daten und Operationen bei der Fehlersuche hinderlich ist. Aus Geschwindigkeitsgründen ist es aber nicht ratsam, generelle Kontrollen auf gültige Indizes durchzuführen. Während der Entwicklungsphase können jedoch mit Hilfe eines `assert`-Makros Kontrollen eingebaut werden:

```
T* at(int const& i, int const& j){
    assert(i<dm1 && j<dm2);
    return a+(i+j*dm1);
}// end function
```

Das `assert`-Makro wird über `#define NDEBUG` gesteuert und enthält entweder einen Funktionsaufruf, der bei Nichterfüllen der Bedingung durchgeführt wird, oder eine leere Anweisung, die beim Optimieren verschwindet, da das Ergebnis der Ausdrucksauswertung nirgendwo benötigt wird.[2]

```
#ifdef NDEBUG
   #define assert(cond) \
      if(!(cond)) _assert_message();
#else
   #define assert(cond)
#endif
```

4.1.1.2 Iteratoren

Die Verwendung von Algorithmen der STL erfordert neben einer schnellen Indexarithmetik auch die Definition von Iteratoren. Das Iteratorenkonzept ist bezüglich der `begin`()- und `end`()-Funktionen allerdings aufwändiger als in der STL, da wir von einem beliebigen Matrixelement in Spalten- oder Zeilenrichtung iterieren müssen. Die Iteratoren selbst sind dann auch keine einfachen Zeiger auf Speicherstellen mehr, sondern müssen ihre Arithmetik mitführen.

Der Vorschub eines Iterators hängt davon ab, ob es sich um einen Zeilen- oder einen Spalteniterator handelt. Spalteniteratoren rücken um eine Einheit vor, Zeileniteratoren um die Dimension der Spalten. Wir vereinigen beide Konzepte in einer Iteratorklasse, indem wir den Vorschub auf einem Attribut hinterlegen.[3] Alles zusammengefasst lautet das Grundkonzept des Iterators daher:

---

[2] Vergleiche die Mechanismen der TRACE-Steuerung im letzten Kapitel.

[3] Es wären zwar auch getrennte Iteratoren für einfachen und mehrfachen Vorschub denkbar, aber den damit verbundenen Aufwand bei der Definition sparen wir uns hier.

```
template <typename T> class MatrixIterator{
public:
  typedef ptrdiff_t            difference_type;
  typedef forward_iterator_tag iterator_category;
  MatrixIterator():_act(0),incr(0){}

  explicit
  MatrixIterator(MatrixIterator<T> const& it):
       _act(it._act),incr(it.incr){}

  template <typename U>
  MatrixIterator(MatrixIterator<U> const& it):
      _act(it._act),incr(it.incr){}

  reference operator*() const { return *_act; }
  pointer   operator->() const{ return _act; }

  inline MatrixIterator<T>& operator++(){
    _act+=incr;
    return *this;
  }//end function

  inline MatrixIterator<T> operator++(int){...}

 template <typename U> inline
 bool operator==(MatrixIterator<U> const& it)const{
   assert(incr==it.incr);
   return _act==it._act;
 }//end function

protected:
  pointer _act;
  int incr;
  MatrixIterator(pointer n, int d):
         _act(n),incr(d){}
´

 template <typename U> friend class MatrixIterator;
 template <typename U> friend class Matrix;
};;//end class
```

Wie Sie bemerken, sind ein Kopierkonstruktor sowie der Vergleichsoperator als innere Template-Deklarationen angelegt. Aufgrund der vom Compiler akzeptierten Konvertierung `T*` ->`T const*` (*Attribut* `_act`) sind Kombinationen von Iteratoren und konstanten Iteratoren in Ausdrücken damit kein Problem.

Im Kopfbereich befinden sich Deklarationen für den Iteratortyp und den Distanztyp. Dies erlaubt dem Compiler bei der Verwendung von Algorithmen aus der STL die Auswahl der korrekten Algorithmen. Weiterhin werden für den Zugriff auf die Datenobjekte die internen Typen `value_type, reference` und `pointer`

definiert (*hier nicht aufgeführt*). Diese Typen sind aus dem Template-Parameter zu generieren, was bei Verwendung einfacher Typen kein großes Problem darstellt, wird ein solcher Typ doch einfach per `typedef`-Befehl umtypisiert. Template-Techniker erlauben jedoch, auch abgeleitete Typen sicher zu Typisieren, und da diese Technik auch außerhalb der Matrixklasse interessant ist, haben wir sie in einem späteren Teilkapitel untergebracht.

Die Iteratoren-Methoden `begin()` und `end()` in der Matrixklasse sind in folgende Versionen vorzusehen (*ergänzen Sie bitte selbst die const-Iteratoren und die end-Funktionen*):

```
iterator begin();
iterator  begin1(int const& index2);
iterator  begin1(int const& index1,
                 int const& index2);
iterator  begin2(int const& index1,
                 int index2=0) const;
```

Die Implementierungen ergeben sich leicht durch einfache arithmetische Überlegungen, z.B.:

```
template <typename T> typename
Matrix<T>::iterator Matrix<T>::begin2(
                  int const& index1, int index2){
    return iterator(at(index1,index2),dm1);
}//end function

template <typename T> inline typename
Matrix<T>::iterator Matrix<T>::end2(
                  int const& index1, int index2){
    return iterator(e+index1,dm1);
}//end function
```

Aufgabe. Implementieren und Testen sie die Iteratoren. Zum Testen können Sie auch einige der STL-Algorithmen nutzen.

4.1.1.3 Arithmetische Grundoperationen

Wie für andere mathematische Objekte, die durch C++ Klassen dargestellt werden, so können auch für Matrizen die Rechenoperatoren für Addition, Multiplikation usw. überschrieben werden. Da Matrizen aber recht speicherintensiv sind, sollte ab einer gewissen Größe darauf geachtet werden, nicht zu großzügig mit dem Speicher umzugehen. Das betrifft nicht nur Operatoren, die temporäre Variable definieren, sondern auch die Matrizenmultiplikation, die einige Besonderheiten aufweist.[4] Die Multiplikation ist definiert durch den mathematischen Ausdruck

[4] Die Addition/Subtraktion ist unkritisch, die Division über die Multiplikation mit der inversen Matrix, sofern eine solche existiert, definiert.

$$(c_{ik}) = \left(\sum_{j=1}^{n} a_{ij} * b_{jk} \right)$$

Wir können nun auf drei verschiedene Situationen bei der Multiplikation treffen, die wir hier durch eine Multiplikationsroutine anstelle der Operatoren beschreiben.

```
mul(a,b,c);     // 1
mul(a,a,c);     // 2
mul(a,a,a);     // 3
```

Im ersten Fall werden zwei Matrizen multipliziert und das Ergebnis auf einer dritten gespeichert. Da drei Matrizen existieren, ist die Umsetzung kein Problem:

```
int i,j;
dest.resize(s1.dim1(),s2.dim2());
for(i=0;i<s1.dim1();i++){
  for(j=0;j<s2.dim2();j++){
    dest(i,j)=inner_product(s1.begin2(i,0),
                            s1.end2(i,0),
                            s2.begin1(0,j),
                            null<T>());
  }//endfor
}//endfor
```

Aufgabe. Der Algorithmus kann natürlich viel einfacher durch die direkte Umsetzung der mathematischen Formel in Ausdrücke mit Indexoperatoren implementieren, während eine Formulierung mit Iteratoren schon etwas Sorgfalt erfordert. Andererseits sollte die Iteratorversion effektiver Arbeiten, da die Indexarithmetik durch eine Zeigerinkrementierung ersetzt ist.
Implementieren Sie trotzdem hier und in anderen Algorithmen jeweils eine Version mit Indexoperatoren. Diese dienen als Referenz zur Kontrolle, ob die Iteratorversionen korrekt umgesetzt sind (Gleichheit der Ergebnisse). Prüfen Sie experimentell, welchen Zeitgewinn Sie durch Iteratorversionen erhalten. Wenn Sie weitere Algorithmen implementieren, können Sie anhand dieser Ergebnisse Entscheidungskriterien aufstellen, ob sich für eine spezielle Aufgabe eine Optimierung lohnt.

Auch Fall 3 lässt sich so lösen, allerdings nur, indem vor Durchführung der Multiplikation die Matrix a kopiert wird und nur die Kopie in der Summe Verwendung findet. Würden wir dies nicht machen, so würden in den ersten Operationen Matrixelemente überschrieben, später jedoch wiederverwendet und damit das Ergebnis falsch.

Bei Fall 2 drängt sich ebenfalls die Vermutung auf, zunächst eine Kopie herstellen zu müssen, jedoch

Aufgabe. Wenn Sie sich die Matrixmultiplikation genau anschauen, können Sie feststellen, dass bei geschickter Summierung eine zusätzliche Zeile oder eine Spalte für die Zwischenspeicherung von Daten genügt, die nach vollständiger Füllung in die entsprechende Zeile/Spalte des Ergebnisse kopiert wird. Arbeiten Sie diesen Algorithmus aus.

Bei der Optimierung des Speicherplatzbedarfs ist also darauf zu achten, dass die Operationen „speichersicher“ ausgeführt werden, d.h. das Daten nicht überschrieben werden, die zu einem späteren Zeitpunkt nochmals benötigt werden. Das trifft auf einige Algorithmen der linearen Algebra zu und erfordert, wie das Beispiel zeigt, oft etwas Überlegung.[5] Darauf müssen Sie im Übrigen auch achten, wenn Sie die Aufgabe durch überschriebene Operatoren erledigen. Welcher der drei Fälle vorliegt, lässt sich intern durch Vergleich der Speicheradressen der Daten ermitteln.

4.1.1.4 Untermatrizen (Slices)

In einigen Arbeitsgebieten, beispielsweise der Computergrafik, werden Untermatrizen benötigt, die aus einer bestimmten Anzahl von Elementen um ein Zentralelement bestehen, bei einigen mathematischen Problemen der linearen Algebra kann eine Matrix in quasidiagonale Teile zerfallen, d.h. längs der Hauptdiagonalen reihen sich kleine Matrizen aneinander, während alle anderen Element Null werden. Um mit diesen Teilmatrizen sinnvoll arbeiten zu können, sind Einschränkungswerkzeuge sinnvoll.

Eine einfache Definition eines solchen `slice` ist die Templateklasse

```
template <class T, bool normal=TRUE> class slice {
public:
    slice(Matrix<T>& a, int xx, int yy,
                        int dxx, int dyy):
           m(a), x(xx), y(yy), dx(dxx), dy(dyy) {}
    T& operator()(int i, int j){return m(x+i,y+j);}
    ...
protected:
    enum {centered=!normal};
    Matrix<T>& m;
    int x,y,dx,dy;
};
```

[5] Alternativ können Sie natürlich wie bei den Indexoperatoren auch zunächst auf die Optimierung verzichten und großzügig mit dem Speicher umgehen, d.h. immer nach Fall 3 operieren. Da der Speicherplatz, den ein Programm verwenden kann, ohne dass es zu temporären Auslagerungen auf die Festplatte kommt, inzwischen beachtliche Größen erreichen kann, stellt sich die Optimierungsfrage bei Ihren Problemen möglicherweise gar nicht. Auch dazu können Sie natürlich wie bei den Indexoperatoren Messungen durchführen und Kriterienkataloge entwickeln.

Sie übernimmt lediglich die geänderte Adressarithmetik, wobei man je nach Anwendung die Indizes unterschiedlich interpretieren kann:

- Bei der Standardindizierung (`normal=TRUE`) geben die übergebenen Indizes das Element (0,0) der Untermatrix an. Die Indizes im `slice` sind positive Zahlen (`0..dxx-1,0..dyy-1`) wie in den Matrizen selbst.
- bei der zentrierten Indizierung (`normal=FALSE`) wird das zentrale Element indiziert und die im Operator `operator()(..)` übergebenen Elementindizes können positiv und negativ sein. Der zulässige Indexbereich ist `(-dxx..dxx,-dyy..dyy)`.

Aufgabe. Implementieren Sie eine vollständige mit den Matrizen identische Schnittstelle. Beachten Sie,dass die Start- und Enditeratoren nur andere Werte erhalten müssen, aber ansonsten von den Matrizen übernommen werden können.

Um Matrizen und Slices beliebig gemischt einsetzen zu können, müssen nur die bestehenden Algorithmen in Templates umgesetzt werden, z.B.

```
template <class T1, class T2, class T3>
void mul(T1& a,T2& b,T3& c){ ...
```

In den Iteratorversionen sollten sich Matrizen und Slices gleich verhalten, in Indexoperatorversionen leichte Verschlechterungen ergeben. Prüfen Sie dies experimentell.

4.1.1.5 Dateneingabe in Matrizen

Recht mühsam ist häufig die Eingabe von Daten in eine Matrix, und nicht nur in Testumgebungen ist es wünschenswert, bei kleineren Matrizen über eine Eingabemöglichkeit zu verfügen, die wie eine Matrix in der Theorie aussieht und so eine einfache Kontrolle der Daten erlaubt:

```
template <typename T> class Matrix {
    typedef Initialiser<Matrix<T> >    initialiser;

Matrix<int>::initialiser an;
an(a) = 1 , 6 , 11 , 16 , 21,
        2 , 7 , 12 , 17 , 22,
        3 , 8 , 13 , 18 , 23,
        4 , 9 , 14 , 19 , 24,
        5 ,10 , 15 , 20 , 25;
```

In diesem Wunschbild wird das erste Element mittels `operator=` zugewiesen und anschließend der Kommaoperator `operator,(. .)` dazu verwendet, auf die Folgepositionen ebenfalls Werte zuzuweisen, wobei im Texteditor das Ganze zu aufgebaut wird, dass es einer Matrix in mathematischer Notation auf Papier gleicht.

Für die Umsetzung definieren wir eine spezielle Klasse

```
template <typename T> class Initialiser {
public:
   ...
private:
    T* ref;
    mutable typename T::iterator it;
};
```

die mittels Iteratoren eine sukzessive Füllung eines Containers bewerkstelligt und damit universell einsetzbar ist. Bei der Konstruktion oder mittels des Klammeroperators wird der zu bearbeitende Container registriert.

```
Initialiser(): ref(0) {}
Initialiser(T& t): ref(&t) {}

Initialiser<T>& operator()(T& t){
    ref=&t;
}//end function
```

Der Zuweisungsoperator übernimmt die Initialisierung des Iterators. Wir implementieren gleich zwei Versionen, die mit Datentypen und mit Stringversionen arbeiten können:

```
Initialiser<T>& operator=(
           typename T::value_type const& a){
    if(ref){
        it=ref->begin();
        *it=a;
    }//end function
    return *this;
}//end function

Initialiser<T>& operator=(char const* s){
    if(ref){
        it=ref->begin();
        *it=from_string<typename T::value_type>(s);
    }//end function
    return *this;
}//end function
```

Der Kommaoperator führt jeweils zuerst eine Iteratorinkrementierung durch, bevor er den Wert abspeichert.

```
Initialiser<T>& operator,(
             typename T::value_type const& a){
    if(ref && it!=ref->end()){
        ++it;
        if(it!=ref->end()){
            *it=a;
```

```
        }//endif
    }//endif
    return *this;
}//end function

Initialiser<T>& operator,(char const* s){
    if(ref && it!=ref->end()){
        ++it;
        if(it!=ref->end()){
            *it=from_string<typename T::value_type>(s);
        }//endif
    }//endif
    return *this;
}//end function
```

4.1.2 Schwach besetzte Matrizen

In einigen Anwendungsbereichen hat man es mit Matrizen zu tun, die sehr große Maximalindizes besitzen, aber im Gegenzug nur wenige Elemente aufweisen, die von Null verschieden sind. Hier kann es sich lohnen, tatsächlich nur die Elemente zusammen mit ihrem Index zu speichern, die nicht Null sind.

Man muss an dieser Stelle festhalten, dass es sich bei der Festlegung des Matrixmodells um eine entweder-oder-Entscheidung handelt. Es macht keinen Sinn, normal besetzte mit schwach besetzten Matrizen zu mischen, da bei Rechnungen in der Regel normal besetzte Matrizen übrig bleiben. Wir müssen also im Folgenden nicht auf Kompatibilität mit dem vorhandenen Modell achten (die auch nur in wenigen Ausnahmefällen möglich ist). Andererseits müssen auch die Algorithmen auf das neue Modell umgeschrieben werden.

4.1.2.1 Die Matrixklasse

Das Speichermodell ist recht einfach: statt `dim1*dim2` Elemente anzulegen, legt man nur eine komplette Zeile oder Spalte an und speichert dann die wenigen Spalten- oder Zeilenelement als Index/Wertpaar.[6] Dazu greift man zweckmäßigerweise auf STL-Containerklassen zurück:

```
template <class T> class SBMatrix{
  ...
protected:
  vector<map<int,T> > mat;
};//end class
```

[6] Auch bei schwacher Besetzung kann man davon ausgehen, dass pro Zeile/Spalte mindestens ein Element vorhanden ist. Eine Zeile/Spalte dann fest vorzusehen, statt das komplette Indexpaar (i,j) als Schlüssel eines Baums zu interpretieren, beschleunigt dann nicht nur das indizierte Zugreifen, sondern vermeidet auch eine Menge Ärger bei der Definition von Iteratoren.

Formal beginnt die Einsparung von Speicherplatz, wenn die Spalten bzw. Zeilen weniger als zur Hälfte besetzt sind, aber die Zugriffe sind natürlich weniger effektiv als in einer vollbesetzten Matrix. Der Zugriff über `operator()(. .)` ist allerdings auf den lesenden Zugriff zu beschränken. Zum einen sind ja nicht alle Matrixelemente tatsächlich vorhanden, und für nicht vorhandene wird Null zurückgegeben, zum anderen können die Einträge in der schwach besetzten Matrix auch gelöscht werden, indem einem Element der Wert Null zugewiesen wird. Der Lesezugriff besitzt dann folgenden Code:

```
typedef typename map<int,T>::const_iterator
                MapConstIt;
typedef typename vector<map<int,T»::const_iterator
                    VecConstIt;

template <typename T> inline
T const& SBMatrix<T>::operator()(int i, int j) const{
    MapConstIt it;
    it=mat[i].find(j);
    if(it!=mat[i].end())
        return it->second;
    else
        return null<T>();
}//end function
```

Beim schreibenden Zugriff ist zu prüfen, ob ein Element verändert, eingefügt oder gelöscht wird:

```
template <typename T> inline
SBMatrix<T>& SBMatrix<T>::set(int i, int j,
                              T const& t){
    typename map<int,T>::iterator it;
    it=mat[i].find(j);
    if(it==mat[i].end()){
        if(t!=null<T>()){
            mat[i].insert(pair<int,T>(j,t));
        }//endif
    }else{
        if(t==null<T>()){
            mat[i].erase(it);
        }else{
            it->second=t;
        }//endif
    }//endif
    return *this;
}//end function
```

Nun ist ein indizierter Zugriff in einer schwach besetzten Matrix nicht gerade eine optimale Wahl für Algorithmen, denn in den meisten Fällen wird ja eine Null zurückgegeben.[7] Es ist daher sinnvoll, von einem besetzten Element zum nächsten springen zu können. Wie sich aber schnell zeigt, sind Iteratoren im herkömmlichen Sinn hier auch kein geeignetes Mittel, da in Algorithmen meist auf mehrere Matrizen zugegriffen werden muss. Für den ersten Zugriff bestehen häufig keine besonderen Vorgaben, die folgenden Zugriffe sind aber an bestimmte Indizes gebunden. Wenn man nicht gerade Iteratoren konstruieren möchte, die Zahlentripel zurückliefern, und überhaupt ein einheitliches Zugriffsbild in allen Matrizen haben möchte, sind andere Wege einzuschlagen.

Wir realisieren dies durch drei Initialisierungsfunktionen und eine Vorschubfunktion, die jeweils neue Koordinaten und den Rückgabewert `true` liefern, wenn weitere Matrixelemente vorhanden sind.

```
bool find_init(int& index1, int& index2) const;
bool row_init(int& index1, int const& index2)const;
bool line_init(int const& index1,int& index2)const;
bool find_next(int& index1, int& index2) const;
```

Die Indexvariablen werden bei der Initialisierung gesetzt und anschließend fortgeschrieben, dürfen aber im rufenden Programm nicht verändert werden. Intern verwenden wir nämlich keine Indexvariablen, sondern Iteratoren auf den beiden Containertypen:

```
mutable vector<map<int,T> >::const_iterator
                  VecConstIt vit;
mutable typename map<int,T>::const_iterator mit;
mutable enum stype { all, line, row} tp;
```

Da die Zugriffe in `const`-Methoden erfolgen, müssen alle Attribute einschließlich der Vorschubart als `mutable` deklariert werden. Würde nämlich ohne diese Deklaration versucht, beispielsweise das Attribut `mit` in der Methode `line_init(. .)` neu zu belegen, so hätte der Compiler berechtigte einwände dagegen. Das Schlüsselwort unterdrückt diese Compilerkontrolle, sollte also nur mit entsprechender Sorgfalt angewandt werden.

Die Methode `find_init(..)` setzt die Indizes auf das erste Matrixelement, wobei der Zeilenvorschub vor dem Spaltenvorschub erfolgt:

```
template <typename T>
bool SBMatrix<T>::find_init(int& index1,
                            int& index2) const{
    index1=index2=0;
    tp=all;
```

[7] Natürlich kann der indizierte Zugriff wieder zum Prüfen der korrekten Umsetzung von Algorithmen eingesetzt werden und stellt sogar die einzige Verträglichkeit mit dem normal besetzten Modell dar, sofern die schwach besetzte Matrix zu den als **const** deklarierten Parametern gehört.

```
    vit=mat.begin();
    mit=vit->begin();
    if(mit!=vit->end()){
        index2=mit->first;
        return true;
    }else{
        return find_next(index1,index2);
    }//endif
}//end fuction
```

Bei der Methode `line_init(..)` ist der erste Index fest vorgegeben, bei `row_init(..)` der zweite, d.h. die Indizes befinden sich auf dem ersten vorhandenen Matrixelement in der Zeile bzw. der Spalte (*oder es wird* `false` *zurückgegeben, wenn in der Zeile/Spalte keine Elemente vorhanden sind*).

Aufgabe. Implementieren Sie die beiden Methoden.

Die Methode `find_next(. .)` orientiert sich an der enum-Variablen `tp`, welche Vorschubart initialisiert wurde. Bei der Vorschubart `all` wird der nächste Eintrag auf der aktuellen Zeile gesucht, ist dort keiner vorhanden, der erste Eintrag auf der nächsten Zeile mit Elementen. Bei den anderen beiden Vorschubarten wird nur auf das nächste Element der Zeile vorgerückt (*am Ende der Zeile erfolgt kein weiterer Vorschub*) bzw. auf die nächste Zeile, die einen Eintrag in der angegebenen Spalte aufweist.

Aufgabe. Implementieren Sie die Vorschubmethode. Sinnvollerweise wird zu Beginn der Funktion bzw. vor einem Iteratorvorschub geprüft, ob die Bedingungen `vit!=mat.end()` bzw. `mit!=vit->end()` zutreffen, da es ansonsten bei einen überzähligen Aufruf zu einem unerlaubten Speicherzugriff kommen kann. Auch bei der Dimensionierung der Matrix empfiehlt es sich, `vit=mat.end()` zu setzen.

Der Einsatz der Methoden in Schleifen erfolgt aufgrund des logischen rückgabewertes in Form einer logischen Variablen:

```
int i,j;
bool loop;
for(loop=find_init(i,j);loop;loop=find_next(i,j))
```

Das sieht zwar etwas ungewohnt aus, aber Sie haben sich ja auch schon an Iteratoren in Schleifen gewöhnt. Bei der Verwendung von `while`-Schleifen kommt man allerdings sowohl bei Kopf- als auch bei Fußsteuerung nicht um eine `if(. .)`-Abfrage herum.

4.1.2.2 Addition und Subtraktion

Der Einsatz von schwach besetzten Matrizen zahlt sich nur aus, wenn sich an dem Besetzungsgrad während der Rechnungen wenig ändert. Ist eine anfangs schwach

besetzte Matrix nach einigen Rechenvorgängen gut gefüllt, sollte lieber auf diese Modell verzichtet werden. Wir sehen deshalb keine Verknüpfung zwischen normal und schwach besetzten Matrizen vor; alle Algorithmen müssen daher für den jeweiligen Typ neu implementiert werden (*sofern sich nicht ohnehin andere Algorithmen aufgrund der Besetzungszahlen anbieten*).

Die Addition ist mittels des Gesamtiterators einfach zu realisieren, da nur jedes Element zu dem entsprechenden in der Zielmatrix addiert werden muss (*wir sehen für die Addition nur zwei Parameter vor*):

```
template <typename T> inline
SBMatrix<T>& add(SBMatrix<T>& dest,SBMatrix<T> const& s1)
    {
    assert(dest.dim1()==s1.dim1());
    int i,j;
    bool loop;
    for(loop=s1.find_init(i,j);
                 loop;loop=s1.find_next(i,j)){
        dest.set(i,j,dest(i,j)+s1(i,j));
    }//endfor
    return dest;
}//end function
```

Bei der Multiplikation gehen wir ähnlich vor: in der ersten Matrix werden nacheinander alle vorhandenen Elemente aufgerufen, anschließend in der zweiten alle Elemente auf der durch den ersten Spaltenindex angegebenen Zeile:

```
template <typename T>
SBMatrix<T>& mul(SBMatrix<T>& dest,
                 SBMatrix<T> const& s1,
                 SBMatrix<T> const& s2){
    if(&dest==&s1 || &dest==&s2){
        SBMatrix<T> tmp;
        mul(tmp,s1,s2);
        dest=tmp;
    }else{
        int i,j,k;
        bool lp1,lp2;
        dest.resize(s1.dim1());
        for(lp1=s1.find_init(i,j);
                     lp1;lp1=s1.find_next(i,j)){
            for(lp2=s2.line_init(j,k);
                     lp2;lp2=s2.find_next(j,k)){
                dest.set(i,k,
                       dest(i,k)+s1(i,j)*s2(j,k));
            }//endfor
        }//endfor
```

```
    }//endif
    return dest;
}//end function
```

Weitere Algorithmen für schwach besetzte Matrizen werden wir nicht diskutieren, da einerseits solche Matrizen nur in recht speziellen Anwendungen auftreten, andererseits ein kurzer Blick in ein Lehrbuch der numerischen Mathematik zeigt, dass anstelle der bekannten Standardalgorithmen auch sehr spezielle Methoden verwendet werden.

4.1.3 Compilezeitoptimierungen – Vektoren und Matrizen

Die Matrixklasse erlaubt eine einheitliche Bearbeitung von Vektoren (=*einzeilige oder einspaltige Matrizen*) und Matrizen, die Sliceklasse erlaubt darüber hinaus auch die Interpretation der Zeilen oder Spalten einer Matrix als Zeilen- oder Spaltenvektoren. Separate Datentypen für Vektoren und Matrizen müssen daher nicht verwendet werden, wie ja auch in der linearen Algebra ziemlich früh gezeigt wird, dass man zwischen den Begriffen eigentlich nicht unterscheiden muss (*oder in der Geometrie eigentlich noch die Unterscheidung Punkt-Vektor hinzufügen müsste, ohne dass mathematisch etwas Neues geschieht*).

Bei den Indexzugriffen und Iteratoren lassen sich noch Verbesserungen erreichen, wenn die Dimensionierung der einzelnen Variablen nicht zur Laufzeit, sondern bereits zur Compilezeit vorgenommen wird. Insbesondere bei mehr als zwei Dimensionen sind die Aufgabenstellungen meist so speziell, dass mit festen Größen gearbeitet werden kann, so dass wir beispielsweise zu folgender Klassendefinition gelangen:

```
template <class T, int di, int dj, int dk>
struct Tensor {
    T m[di][dj][dk];
    T& operator()(int i, int j, int k)
         { return m[i][j][k];}
    ...
};
```

Bei indexgestützten Algorithmen führt die CPU zwar auch in diesem Fall eine Indexarithmetik durch, jedoch arbeitet diese mit konstanten Maximalindizes anstelle von in Variablen gespeicherten Werten, und manche CPU-Typen bieten zusätzlich spezielle hardwaregestützte Indexarithmetiken an. Die Compilezeitfestlegung der Größenparameter wird somit auf jeden Fall durch eine gesteigerte Effizienz belohnt.

Spalteniteratoren, die auf die nächste Zeile springen müssen, können davon ebenfalls Gebrauch machen.

```
template <typename T, int inkr>
class MatrixIterator {
  ...
```

```
MatrixIterator<T,inkr>T& operator++(int){
  _actr+=inkr;
   return *this;
}
```

Auch hier ist das Inkrement nun nicht mehr auf einem Attribut abgelegt, sondern eine Konstante. In ähnlicher Weise lassen sich Slices definieren, ohne dass wir dies hier ausführen wollen.

> **Aufgabe.** Implementieren Sie mit Hilfe der Template-Technik Matrizen sowie Felder mit drei oder mehr Dimensionen (für jede Dimension wird ein weiterer Templateparameter eingeführt). Achten Sie auf kompatible Schnittstellen zu den normal besetzten Matrizen, damit die bereits entwickelten Algorithmen weiterverwendet werden können. Führen Sie einige Laufzeittest zur Ermittlung der Effizienz durch.

Sollten Sie in der Praxis auf Anwendungsaufgaben stoßen, die die Verwendung derartiger Strukturen erfordert, können Sie auf die blitz++ Bibliothek zurückgreifen, mit der einige Übungen an dieser Stelle empfohlen seien. Zur weiteren Optimierung von Algorithmen werden dort noch anderen Softwaretechniken eingesetzt, die wir jedoch erst weiter hinten im Kapitel „Ausdrücke“ diskutieren werden.

4.2 Numerisch–Mathematische Klassen

4.2.1 Das Rundungsproblem

In den schwach besetzten Matrizen haben wir Matrixelemente nur dann gelöscht, wenn eine Null eingetragen wird. Hierzu ist anzumerken, dass bei Verwendung von Fließkommazahlen die Prüfung $a_{rs} = 0$ nicht trivial ist. Wir schauen uns das Phänomen einmal für die verschiedenen Zahlenklassen an (*weitere Betrachtungen dazu folgen in einem späteren Kapitel*).

Rechnungen mit ganzen Zahlen können auf dem Computer exakt durchgeführt werden, wenn Divisionen vermieden werden, bei denen der Divisor nicht als Faktor im Dividenden enthalten ist (*vielfach lässt sich dies vermeiden, indem beide Größen zuvor mit geeigneten Faktoren multipliziert werden*). Probleme mit der Nullprüfung existieren nicht.

Noch günstiger sieht es bei der Verwendung von rationalen Zahlen aus. Auch die Division ist nun immer korrekt durchführbar, und die Nullprüfung bereitet ebenfalls keine Probleme. Mit der Korrektheit ist es jedoch vorbei, sobald Wurzeln, Logarithmen oder trigonometrische Funktionen in Spiel kommen. Die Ergebnisse sind reelle oder komplexe Zahlen, die grundsätzlich nicht korrekt repräsentiert werden können.

Da die Beschränkung auf rationale Zahlen in der Praxis nur selten gelingt und Rechnungen mit rationalen Zahlen in der Darstellungsform

`(ganzzahliger_Zähler, ganzzahliger_Nenner)` nicht gerade besonders effektiv sind, verzichtet man meist auf diesen Zahlentyp und verwendet Fließkommadarstellungen, die auch für reelle Zahlen verwandt werden. Eine Fließkommazahl auf einem Rechner besitzt die allgemeine Darstellung

$$z = \pm \left(\sum_{k=1}^{m} a_k * 2^{-k} \right) * 2^e$$

Während durch den Exponenten e die Zahl in einem sehr großen Intervall liegen kann (*bei dem Datentyp* `double` *gilt* $|e| \leq 1.024$ *oder* $10^{-300} < |z| < 10^{300}$), stehen nur m Stellen für die Genauigkeit der Darstellung zur Verfügung (*wieder für* `double` *:* $m = 52$. *Für* $e = 0$ *entspricht das erste Bit dem Zahlenwert* 0, 5*, das letzte dem Zahlenwert*

$$\epsilon = 2{,}2 * 10^{-16}$$

Zwei Zahlen, die sich um weniger als den zweiten Wert unterscheiden, sind für den Rechner identisch). Die Beschränkung in der Genauigkeit führt dazu, dass bei Rechnungen laufend gerundet werden muss:

- Bei der Multiplikation entstehen zunächst Zahlen mit 2^*m Stellen, von denen die hintere Hälfte abgeschnitten werden muss (*ähnliches gilt für die Division*),
- bei der Addition von Zahlen mit $e_1 \neq e_2$ muss eine Zahl so verschoben werden, dass korrespondierende Bits addiert werden können, was auch wiederum zu einem Überhang führt, der der Rundung unterliegt (*ähnliches gilt wieder für die Subtraktion*)[8]

$$0{,}331 * 10^0 + 0{,}225 * 10^{-1} \Rightarrow 0{,}331 * 10^0 + 0{,}023 * 10^0$$

Aufgrund der Rundungen gelten die grundlegenden mathematischen Gesetze der Assoziativität und der Distributivität von Rechnungen auf einem Rechner für die meisten Zahlen nicht mehr. In den meisten Fällen findet man[9]

$$(a \circ b) \circ c \neq a \circ (b \circ c)$$
$$a * (b + c) \neq a * b + a * c$$

Insbesondere wird in den seltensten Fällen bei einer Rechnung der Wert Null entstehen, auch wenn dies nach der Theorie so sein müsste. Für unser spezielles

[8] Da gilt sogar noch wesentlich schlimmeres. Hierzu kommen wir aber erst in einem späteren Kapitel.

[9] Natürlich gibt es auch Zahlen, die sich mit Fliesskommazahlen exakt darstellen lassen und bei Rechnungen nichts an Genauigkeit verlieren. Das sind jedoch die Ausnahmen.

Problem –verschwindende Matrixelemente in schwach besetzten Matrizen- werden wir deshalb das Prüfverfahren ändern müssen. Alle Zahlen, deren Absolutwerte eine noch festzulegende vorgegebene Schranke unterschreiten, sind als Null betrachten:

$$a =_R 0 \Leftrightarrow |a| \leq \delta$$

Ist der Wert ϵ die Grenze für die Unterscheidung zweier Zahlen der Größe Eins, so ergibt sich die Grenze für Zahlen anderer Größe, indem wir ihren Absolutwert mit ϵ multiplizieren. Allerdings ist dieser Wert im Grunde nur dann brauchbar, wenn die Zahlen noch keine Geschichte haben. Sind sie in Rechnungen verwendet worden, so sind sie dabei Rundungsvorgängen unterworfen gewesen, die etwas größere oder etwas kleinere Werte als den korrekten Wert ergeben. Da wir nicht ausschließen können, dass mehrere aufeinander folgende Rechnungen jeweils in die gleiche Richtung runden, kann die resultierende Abweichung vom korrekten Wert auch größer als ϵ werden. Wir berücksichtigen dies durch einen zusätzlichen Faktor f und definieren

$$a =_R b \Leftrightarrow |a - b| \leq \epsilon * max(\,|a|\,,|b|\,) * f$$

Wie wir später begründen werden, erhalten wir mit $f \approx 100 \ldots 1000$ meist vernünftige Ergebnisse.

Fassen wir nochmals zusammen:

- Ganzzahlige Typen lassen immer exakte Rechnungen zu (*das heißt* $\in= 0$),
- Rationale Zahlen lassen ebenfalls exakte Rechnungen zu, so lange keine algebraischen oder transzendenten Zahlen (*Wurzeln*, π) berücksichtigt werden müssen,
- Fließkommazahlen sind (*nahezu*) grundsätzlich ungenau.

4.2.2 Algebraische Eigenschaften

Da wir in unseren Matrizen den Grundzahlentyp des Vektorraumes als `template`–Parameter übergeben, bleibt nichts anderes übrig, als die algebraischen Eigenschaften der Basis auf ähnliche Art zu definieren, wenn wir nicht doch für jeden Zahlentyp eine Spezialisierung für die Nullprüfung einführen wollen. Hierzu definiert bereits die C++–Standardbibliothek eine Vorlagenklasse, von der wir hier allerdings nur einige der für uns interessanten Attribute oder Methoden angeben:

```
// Allgemeine Klassendefinition
// ============================
template <class T> class numeric_limits {
public:
    enum { is_specialized    = false };
    enum { is_exact        = true };
    enum { is_integer    = true };
```

```
    enum { has_signaling_NaN    = false };
    inline static T epsilon(){return 0;};
    inline static T max(){...};
    inline static T min(){...};
    inline static signaling_NaN() {..};

    ...
};//end class

// Spezialisierung für den Datentyp „rational"
// ============================================
class numeric_limits<rational> {
public:
    enum { is_specialized    = true };
    enum { is_exact        = true };
    enum { is_integer    = false };
    inline static double epsilon(){return 0;};
    ...
};//end class

// Spezialisierung für den Datentyp„double"
// ========================================
class numeric_limits<double> {
public:
    enum { is_specialized    = true };
    enum { is_exact        = false };
    enum { is_integer    = false };
    inline static double epsilon()
           {return 2.2*e-16;};
    ...
};//end class
```

Die Attribute besitzen folgende Bedeutung:

- `is_specialised` gibt an, ob die Eigenschaftsklasse für den als Vorlagenparameter angegebenen Datentyp spezialisiert ist.[10]
- `is_exact` signalisiert, ob eine Rechnung korrekt durchführbar ist oder Rundungsfehlern unterliegt,
- `is_integer` kann als Hinweis genommen werden, ob es sich um Ringe oder Körper handelt, das heißt die Division vollständig oder nur mit Rest durchführbar ist,
- `has_signaling_NaN` gibt an, ob der Datentyp speziell ausgezeichnete Bitmuster besitzt, die im Fall ungültiger Operationen ausgegeben werden. Dies ist bei Fließkommazahlen etwa dann der Fall, wenn durch Null dividiert wird.

[10] Ist das nicht der Fall, kann beispielsweise eine Warnung vom betroffenen Algorithmus ausgegeben werden.

Die statischen Methoden erklären sich von selbst. Die Spezialisierungen für die Standarddatentypen sind im Bibliotheksheader `<limits>` implementiert (*und fassen die in* `<math.h>` *definierten C–Konstanten etwas handlicher zusammen*). Weitere Typen können bei Bedarf selbst spezialisiert werden.

Da die Typklasse sich vorzugsweise an rechentechnischen Eigenschaften orientiert und weniger an mathematischen, besteht möglicherweise Bedarf an weiteren Festlegungen. Da man tunlichst nicht an Bibliotheksklassen herumfummelt (schließlich gibt es irgendwann ein Update), erledigt man dies durch eine eigene Erweiterung:

```
template <class T>
class algebraic_properties:
      public numeric_limits<T> {
...
```

Das aber nur als Randbemerkung, da wir hier vorläufig keinen Bedarf an Erweiterungen haben.

4.2.3 Konstantenvereinbarungen

Da allerdings trotz eines recht großen Umfangs an Attributen und Methoden nicht alles in der Standardklasse vorhanden ist, was wir (*auch in weiteren Anwendungen*) benötigen werden und einige Begriffe (*wie* `is_integer`) nur annähernd das beschreiben, was wir brauchen, definieren wir ergänzend einige eigene Klassen und Methoden. Zunächst ist es recht sinnvoll, auf eine Reihe von Konstanten typunabhängig zugreifen zu können:

```
template <typename T> struct Constant {
    static inline T const& null(){
        static T Null(0);
        return Null;
    }//end constant

    static inline T const& eins(){
        static T EINS(1);
        return EINS;
    }//end constant
    ....
};//end struct
```

Dieses allgemeine Template können wir beliebig spezialisieren:

```
template <typename T> struct Constant<complex<T> >{
    static inline complex<T> const& null(){
       static complex<T> Null(Constant<T>::null(),
                              Constant<T>::null());
```

```
        return Null;
    }//end constant

    static inline complex<T> const& eins(){
       static complex<T> EINS(Constant<T>::eins(),
                              Constant<T>::null());
        return EINS;
    }//end constant
};//end struct
```

Auch wenn in den meisten Fällen dies kein Unterschied zur direkten Verwendung von 0 oder 1 darstellt und ein wenig mehr Schreibarbeit darstellt, ist eine solche Klasse aus verschiedenen Gründen nützlich:

- Die Konstanten werden nur einmalig instanziiert (interessant bei komplexeren Datentypen).
- Die Initialisierung für komplexe Typen kann individuell erfolgen.[11]
- Die Liste von Konstanten kann beliebig und typindividuell erweitert werden.
- Der (versehentliche) Einsatz eines Datentyps in einer für ihn ungeeigneten Umgebung oder das Fehlen der Spezifizierung einer Konstanten fällt dem Compiler auf, so dass man besser vor unliebsamen Berechnungsergebnissen geschützt ist.

Die umständliche Schreibweise kann noch mit Hilfe einer Templatefunktion abgekürzt werden:

```
template <class T> inline
T const& null() { return Constant<T>::null(); }

double r = null<double>();
```

Die Vereinfachung ist allerdings ein zusätzlicher Schritt. Der Umweg über eine Klasse zur Definition einer Konstanten ist notwendig, da diese in einer anderen Klasse zum `friend` erklärt werden kann und nur so die Möglichkeit einer beliebigen Programmierung einer Konstanten bietet.

4.2.4 Vergleiche und Nullprüfungen

Um nun die Rundungsfehler berücksichtigen zu können, ersetzen wir die Vergleichsoperatoren ebenfalls durch Klassen, die wir später spezialisieren können.

```
template <class T>
struct compare {
    compare(){}

    void load(T const& t){}
```

[11] Beispielsweise kann die Klasse `constant` als `friend` erklärt werden, was ihr beliebige Manipulationen während der Initialisierung erlaubt.

```
    inline bool equal(const T& s, const T& t)
            {return s==t;}
    inline bool less(const T& s, const T& t)
            {return s<t;}
    inline bool zero(const T& s)
            {return s==Constant<T>::null();}
};//end struct
```

Weitere Vergleiche können nach Bedarf implementiert werden. Wichtig ist hier die leere Methode `load()`. Die Standardimplementation macht noch nichts anderes als die Vergleichsoperatoren. Bei Datentypen mit Rundungsfehlern ändert sich das allerdings, denn hier eben wird nicht mehr auf absolute Übereinstimmung getestet. Im Falle einer `double`-Prüfung sieht das folgendermaßen aus:

```
inline double& cmp_eps_double(){
    static double r=1.0e-13;
    return r;
}//endfunction

inline double& cmp_ref_double(){
    static double r=1.0;
    return r;
}//endfunction

template <> struct compare<double> {
    compare(){ load(cmp_ref_double());
               frexp(cmp_eps_double(),
               &cmp_exp_double);}

    inline void load(const double& s)const
             { frexp(s,&ref);}
    inline bool equal(const double& s,
                      const double& t) const {
        load(s);
        return zero(s-t);
    }//end function

    inline bool less(const double& s,
                     const double& t) const {
        return (s<t) && !equal(s,t);
    }//end function

    inline bool zero(const double& s) const {
        int e;
        if(s==0)
            return true;
        frexp(s,&e);
```

```
        return e<=ref+cmp_exp_double;
    }//end function
private:
    mutable int ref, cmp_exp_double;
};//end struct
```

Das sieht zunächst etwas kompliziert aus, was aber daran liegt, dass die normierte Struktur des `double`-Typs zur Optimierung der Prüfung herangezogen wurde und Teile der Prüfung nur auf dem Exponenten stattfinden und nicht auf der kompletten Zahl.[12]

Wichtig ist, dass für die Prüfungen jeweils ein Referenzwert benötigt wird. Werden zum Beispiel zwei im betrachteten Sinn gleiche Zahlen in der Größenordnung 10^{20} voneinander subtrahiert, so kann das Ergebnis immer noch im Bereich 10^4 liegen, obwohl es als Null zu betrachten ist. Hier kommt nun die Methode `load()` ins Spiel, die sich den Exponenten der Vergleichsgröße merkt, gegen den später der Nullvergleich durchgeführt wird.

Der benötigte Referenzwert hat jedoch zur Folge, dass die Klasse nicht mehr statisch eingesetzt werden kann, sondern ein Objekt erzeugt werden muss, das vor der Verwendung mit dem Referenzwert geladen wird.

```
compare<double> cmp;
cmp.load(r);
....
if(cmp.zero(a)) ...
```

Dies muss in allen Algorithmen erfolgen, in denen derartige Prüfungen vorgenommen werden sollen, und betrifft dann auch Implementationen, in denen der Algorithmus mit dem Datentyp `int` instanziiert wird. Wie eine Inspektion des Kodes zeigt, ist das aber nur mit Schreibarbeit, jedoch nicht mit einem Laufzeitaufwand verbunden, da die Objekte bei rundungsfehlerfreien Typen leer sind.

Im Konstruktor werden die Objekte auf plausible Standardwerte vorgeladen, die ihrerseits als Singletons angelegt sind und anwendungsspezifisch eingerichtet werden können. In Algorithmen, in denen diese Bezugswerte ausreichen, kann auf die aufwändige Anlage von Vergleichsobjekten verzichtet werden:

```
template <class T> inline
bool zero(T const & t)
    {return compare<T>().zero(t) ;}
```

Aufgabe. Implementieren Sie Konstanten und Vergleichsklassen für die Template-Klasse `complex`.

[12] Es sind einige C-Bibliotheksmethoden beteiligt, und das Handbuch der C-Bibliothek gibt Ihnen weitere Einblicke in die Art der Optimierung.

4.2.5 Anwendung auf schwach besetzten Matrizen

Mit diesen Erweiterungen kann nun versucht werden, die grundsätzlich unterschiedlichen Zugriffsschemata zwischen normal und schwach besetzten Matrizen aufzuheben und das Löschen nicht mehr benötigter Elemente in schwach besetzten Matrizen zu automatisieren, indem die Nullprüfung in der `set`-Methode durch eine `zero`-Prüfung ersetzt wird.

Das Problem hierbei ist die Festlegung eines passenden Bezugswertes. In einigen Anwendungen mag es möglich sein, diesen zentral für alle Vergleiche zu definieren, aber in anderen Umgebungen kann man damit auch auf Probleme stoßen. Betrachten wir beispielsweise Matrizen, deren Elemente durch die Relation

$$\left(a_{k,1} \sim x, a_{k,2} \sim x^2, a_{k,3} \sim x^3, \ldots a_{k,n} \sim x^n\right)$$

verknüpft sind, so sind ohne weiteres Elemente möglich, die in einer zentral gesteuerten Prüfung die Nullbedingung erfüllen, ohne dass sie anwendungstechnisch tatsächlich als Null zu betrachten sind und gelöscht werden dürfen. Vielmehr kann es hier notwendig werden, für unterschiedliche Bereiche der Matrix unterschiedliche Bezugswerte vorzugeben. Das kann dann allerdings auch nur individuell im Anwendungsprogramm erfolgen.

Des weiteren verändert sich die Vertrauenswürdigkeit von Daten häufig im Laufe der Rechnung, weil sich Fehler aufschaukeln können. War beispielsweise zu Beginn der Rechnung ein Wert von 10^{-13} noch ein korrekter Eingabewert, kann es passieren, dass nach einer Reihe von Rechenschritten alles unterhalb 10^{-12} als Null betrachtet werden und gelöscht werden kann.

Mit anderen Worten: bevor Sie nun die numerisch-mathematischen Klassen in ihre Matrizenalgorithmen einbauen, sollten Sie zunächst einmal prüfen, ob das für Ihre spezielle Anwendung auch sinnvoll ist. Diese numerisch-mathematischen Probleme sind in der Praxis von hoher Bedeutung, weshalb ich ihnen gegen Ende des Buches auch ein eigenes Hauptkapitel widme. Arbeiten Sie ggf. daher auch dieses Kapitel durch, damit Sie mit Ihren Implementationen auch das erreichen, was anwendungstechnisch gefordert ist.

4.3 Einige Algorithmen der linearen Algebra

Die Basisalgorithmen für Addition und Multiplikation haben wir bereits bei der Implementation der Matrixklassen vorgestellt. Ebenfalls wenig Probleme dürften folgende Algorithmen beinhalten:

> **Aufgabe.** Implementieren Sie Methoden für die Erzeugung einer Nullmatrix, einer Einheitsmatrix, einer transponierten Matrix, einer Multiplikation mit einem Faktor (*bei diesen Methoden wird die übergebene Matrix in diese Form überführt, aber keine neue Matrix erzeugt*) und für die Erzeugung einer Kopie von einer Matrix.

Wir stellen in diesem Abschnitt zwei Algorithmen vor, die zur Lösung von in der Praxis erstaunlich häufig auftretenden Problemen verwendet werden können:

- Den Gaussalgorithmus zur Lösung linearer Gleichungssysteme sowie
- einen einfachen Algorithmus zur Berechnung von Eigenwerten und Eigenvektoren symmetrischer Matrizen.

Die hier vorgestellten Algorithmen sind aus didaktischen Gründen ausgewählt, weil sie auch ohne tiefes mathematisches Verständnis verständlich sein sollten. Bei komplexeren realen Problemen ist daher etwas Vorsicht und notfalls der Griff zu einem Buch über numerischen Mathematik angesagt.

Bezüglich des Gaussalgorithmus sei noch angemerkt, dass im Kapitel über Zahlendarstellungen im Unterkapitel Körper noch eine Template-Spezialisierung vorgestellt wird, die bei kleinen Gleichungssystemen komplett ohne Schleifenkonstrukte auskommt. Sie können nach dem Studium des Gaussalgorithmus bereits einen vorsichtigen Blick in dieses Kapitel werfen.

4.3.1 Lineare Gleichungssysteme

4.3.1.1 Zur Theorie des Gaussalgorithmus

Wir diskutieren nun den Gaussalgorithmus zur Lösung von linearen Gleichungssystemen. Er ist gewissermaßen die elementare Grundlage für die lineare Algebra, und verlangt aber doch soviel Verständnis bei der Umsetzung, dass der eine oder andere Anfänger Probleme damit hat.[13] Die Lösung der Aufgabe

$$A\,\vec{x} = \vec{b}$$

mit bekanntem $\boldsymbol{A}$ und $\vec{b}$ verfolgt das Ziel, die Matrix in eine rechte obere Dreiecksmatrix umzuformen, d.h. unterhalb der Hauptdiagonale sind nur Nullwerte zu finden. Dies entspricht dem reduzierten Gleichungssystem

$$\begin{gathered} a_{11} * x_1 + a_{12} * x_2 + \ldots + a_{1n} * x_n = b_1 \\ a_{22} * x_2 + \ldots + a_{2n} * x_n = b_2 \\ \ldots \\ a_{n-1n-1} * x_{n-1} + a_{n-1n} * x_n = b_{n-1} \\ a_{nn} * x_n = b_n \end{gathered}$$

Erlaubte Manipulationen – sie ändern an den Lösungswerten x_k nichts – die Addition von Gleichungen, das Vertauschen von Gleichungen (*Spaltentausch ist auch*

[13] Für das Folgende ist es notwendig, dass Sie die mathematische Seite des Algorithmus kennen und in seinen Grundprinzipien verstanden haben. Aus Platz- und Themengründen kann ich hier nicht die mathematische Theorie komplett aufrollen.

erlaubt, erfordert aber eine entsprechende Umnumerierung der x_k) und die Multiplikation kompletter Gleichungen mit beliebigen Faktoren. Um zur Dreiecksform zu gelangen, wird zunächst mit der ersten Spalten beginnend die Zeile mit dem betragsmäßig größten Element in einer Spalte auf die Diagonalstelle verschoben. Anschließend werden geeignete Vielfache dieser Zeile so von den restlichen Zeilen subtrahiert, dass die Elemente unterhalb des Diagonalelementes Null werden. Die Zeilentauschungen und Faktoren können gesichert werden, um das Gleichungssystem für andere Werten b_k zu Lösen, ohne noch einmal von vorne anfangen zu müssen.

4.3.1.2 Zur Implementation des Gaussalgorithmus

Die folgenden Darlegungen beschränke ich auf voll besetzte Matrizen und Rechnungen mit Fließkommazahlen. Im ersten Schritt wird im Gleichungssystem

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots a_{1n}x_n = b_1 \\ &a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots a_{2n}x_n = b_2 \\ &\ldots \end{aligned}$$

die erste Zeile mit (a_{21}/a_{11}) multipliziert und von der zweiten abgezogen, wodurch der erste Term verschwindet. Wenn zunächst nur die Matrix umgeformt wird, muss der Faktor, der dann später noch für die Berechnung der Vektorkomponenten $(\ldots, b_k \ldots)$ benötigt wird, zwischengespeichert werden. Dies kann formal auf den frei werdenden Positionen der Matrix erfolgen, wir werden hier jedoch eine zweite Matrix dazu erzeugen, die nach Abschluss aller Berechnungen eine linke untere Dreiecksmatrix (*L-Matrix*) mit 1 auf der Diagonale darstellt. Zwischen den Matrizen besteht der Zusammenhang

$$A = L * R$$

Soll der Gaussalgorithmus mit ganzzahligen Koeffizienten durchgeführt werden, so sind anstelle des Quotienten beide Zeilen mit Faktoren zu multiplizieren, um die Spaltenwerte korrekt verschwinden lassen zu können. Das passt aber nicht zu den hier verwendeten Speicherstrukturen, so dass man sich – wenn solche Anwendungen jemals auftreten sollten – andere Gedanken dazu machen muss.

Wir sehen ein zweistufiges Lösungssystem vor: zunächst wird die Matrix in zwei Dreicksmatrizen zerlegt:

```
template <class T>
bool gauss_LR(Matrix<T>& AR,
              Matrix<T>& L,
              vector<int>& v);
```

Der Vektor `v` hält die Zeilentauschungen fest. Mit Hilfe von Algorithmen aus der STL lässt sich ein Zeilentausch, der das größte Element unterhalb der Hauptdiagonalen auf die Diagonalposition bringt, leicht realisieren:

```
template <class T> struct abs_gt:
  public binary_function<T,T,T> {
    inline bool operator()(T const& t1,
                           T const& t2){
        return abs(t1)<abs(t2);
    }//end function
};//end class

template <class T>
void pivot_line(Matrix<T>& m,
                int ind1,int ind2,
                vector<int>& v){
    typename Matrix<T>::iterator it;
    int diff;
    it=max_element(m.begin1(ind1,ind2),
                   m.end1(ind1,ind2),abs_gt<T>());
    diff=distance(m.begin1(ind1,ind2),it);
    if(v.size()<ind1)
        v.resize(ind1);
    v[ind1]=ind1+diff;
    if(diff!=0){
        swap_ranges(m.begin2(ind1,0),
            m.end2(ind1,0),m.begin2(ind1+diff,0));
    }//endif
}//end function
```

Die Implementation der Gauss-Algorithmus fällt unter Verwendung dieser Verschiebungsfunktion recht kurz aus: nach Verschieben wird die Verschiebung auch auf die L-Matrix angewandt. Anschließend werden für alle unterhalb des Hauptdiagonalelementes liegenden Zeilen die Faktoren berechnet und in der L-Matrix abgespeichert sowie die Zeilentransformation ausgeführt:

```
template <class T>
struct gauss_f: public binary_function<T,T,T> {
    gauss_f(T const& f): faktor(f) {};
    inline T operator()(T const& t1, T const& t2){
        return t1-faktor*t2;
    }//end function
private:
    T const& faktor;
    gauss_f();
};//end classtemplate <class T>
bool gauss_LR(Matrix<T>& AR, Matrix<T>& L,
                             vector<int>& v){
  ...
  for(ii=0;ii<AR.dim1()-1;ii++){
```

```
    pivot_line(AR,ii,ii,v);
    if(v[ii]!=ii)
        swap_ranges(L.begin2(ii,0),L.end2(ii,0),
                                  L.begin2(v[ii],0));
    if(AR(ii,ii)==0)
        return false;
    for(j=ii+1;j<AR.dim1();++j){
      L(j,ii)=AR(j,ii)/AR(ii,ii);
      AR(j,ii)=null<T>();
      transform(AR.begin2(j,ii+1),AR.end2(j,ii),
                      AR.begin2(ii,ii+1),
                      AR.begin2(j,ii+1),
                      gauss_f<T>(L(j,ii)));
        }//endfor
    }//endfor
    for(j=0;j<L.dim1();j++)
        L(j,j)=eins<T>();
    return true;
}//endif
```

Beachten Sie, wie wir auch hier wieder die STL-Algorithmen ausgenutzt haben.

Im zweiten Teil sind nun die Lösungen zu ermitteln, die mathematisch folgendermaßen berechnet werden:

$$x_k = \left(b_k - \sum_{j=k+1}^{n} x_j * a_{kj} \right) / a_{kk},\ k = n, n-1, \ldots 1$$

Wir gestalten den Algorithmus so, dass direkt eine ganze Serie von Lösungen berechnet werden kann. Die rechten Seiten der Gleichungen werden durch eine Matrix vorgegeben, deren jede Spalte ein Ergebnis präsentiert. Die Einträge sind zunächst entsprechend den im Vektor v notierten Zeilentauschungen in die richtige Reihenfolge zu bringen und mit den in der L-Matrix notierten Faktoren zu verrechnen. Als Gesamtalgorithmus folgt:

```
template <class T>
void solve_LR(Matrix<T> const& R,
              Matrix<T> const& L,
              Matrix<T> const& b,
              Matrix<T>& x,
             vector<int> const& v){
  ...
  for(i=0;i<bc.dim1()-1;++i)
    if(v[i]!=0)
      swap_ranges(bc.begin2(i,0),bc.end2(i,0),
                            bc.begin2(v[i],0));
```

```
  for(i=0;i<bc.dim1()-1;i++)
    for(j=i+1;j<bc.dim1();j++)
      transform(bc.begin2(j,0),bc.end2(j,0),
                bc.begin2(i,0),
                bc.begin2(j,0),
                gauss_f<T>(L(j,i)));
  for(i=0;i<bc.dim2();++i)
    for(j=bc.dim1()-1;j>=0;--j)
      xx(j,i)=(bc(j,i)-
               inner_product(R.begin2(j,j+1),
                             R.end2(j,j+1),
                             xx.begin1(j+1,i),
                             null<T>()))/R(j,j);
  x=xx;
}//end function
```

Die drei Schritte sind im Code wohl recht deutlich zu erkennen, so dass sich weitere Kommentare erübrigen. Die Matrix **x** enthält alle Lösungsvektoren.

Wie durch die Schachtelung der Schleifen (*eine ist jeweils durch einen* `transform` *-Algorithmus ersetzt*) erkennen lässt, steigt der Rechenaufwand proportional zur 3. Potenz der Zeilenanzahl der Matrix, wird also schnell größer.

4.3.1.3 Grenzen und Alternativen

Der Gaussalgorithmus bereitet Probleme, wenn während der Bearbeitung ein Diagonalelement trotz Pivotsuche relativ zu den übrigen Matrixelementen in die Nähe der Null gerät.[14] Wird es exakt Null, ist die Matrix singulär und das zugehörige Gleichungssystem nicht lösbar. Ist ein Matrixelement nach einem Rechenschritt zwar nicht als Null anzusehen, jedoch auch nicht weit davon entfernt, und rechnet man mit diesem Wert weiter, so kann es passieren, dass das Ergebnis kaum etwas mit der korrekten Lösung zu tun hat. Als Beispiel betrachten Sie

$$\begin{aligned} 64.919.121x - 159.018.721y &= 1 \\ 41.869.520{,}5x - 102.558.961y &= 0 \end{aligned}$$

mit der Lösung

	X	*Y*
FP-Arithmetik	102.558.961	41.869.520,5
Korrekte Lösung	205.117.992	83.739.041

[14] Ohne dass wir dies hier nun besonders nachweisen, bedeutet dies, dass zwei Zeilenvektoren der Matrix mehr oder weniger kolinear sind.

Die mit der gebräuchlichen Fließkommaarithmetik berechnete Lösung stimmt in diesem Extrembeispiel in keiner Ziffer mit der korrekten Lösung überein. Ursachen und Kontrollen dieser numerischen Instabilitäten werden wir in einem eigenen Kapitel ausführlich untersuchen, und ich begnüge mich daher hier mit der Darstellung des Phänomens.

Um solche Effekte zumindest zu erkennen, sollte anschließend zumindest überprüft werden, ob der Vektor

$$\vec{r} = \vec{b} - A * \vec{x}$$

hinreichend gut dem Nullvektor entspricht. Versuchsweise kann auch das Gleichungssystem erneut mit $\vec{r}$ an der Stelle von $\vec{b}$ gelöst und die Lösung zur ursprünglichen Lösung $\vec{x}$ addiert werden. Durch derartige Nachiteration lässt sich das Ergebnis in einem gewissen Umfang verbessern.

In der Praxis wird der Gaussalgorithmus häufig durch den Householder-Algorithmus ersetzt, der diese Effekte nicht aufweist, auf den wir hier aber nicht eingehen können. Für schwach besetzte sehr große Gleichungssysteme existieren auf noch weitere spezielle Algorithmen, die den Gaussalgorithmus substituieren.

Neben den Lösungen eines Gleichungssystems liefert der Algorithmus noch zwei weitere Ergebnisse:

(a) Das Produkt der Diagonalelemente der R-Matrix ist der Wert der Determinante. Dieser ist auf anderem Wege kaum zu ermitteln, wird aber zum Glück auch recht selten benötigt, obwohl Determinanten in verschiedenen theoretischen Bereichen häufig auftreten.
(b) Wird im zweiten Teil für **b** eine Einheitsmatrix übergeben, so ist das Ergebnis die inverse Matrix $\boldsymbol{A}^{-1}$. Der Algorithmus ist nicht ungünstiger als der speziellere Gauss-Jordan-Algorithmus, erspart aber eine Menge weiterer Programmierarbeit. Auch inverse Matrizen werden übrigens relativ selten tatsächlich benötigt.

4.3.2 Eigenwerte von Matrizen

Ein relativ häufig auftretendes Problem ist die Bestimmung von Eigenwerten und Eigenvektoren von Matrizen. Eigenwert und Eigenvektor sind definiert durch

$$A * \vec{v} = \lambda * \vec{v}$$

Die Anwendung der Transformation $\boldsymbol{A}$ führt also nur zu einer Längenänderung des Eigenvektors. Schreibt man das Problem in anderer Form auf, so gelangt man zu einem Nullstellenproblem für Polynome:

$$(A - \lambda * E)\vec{v} = 0 \quad \Leftrightarrow \quad det\,(A - X * E) = 0$$

Das durch die Determinante definiert Polynom heißt charakteristisches Polynom der Matrix. Die Determinantenform nützt nun für die Erarbeitung einer Lösung nichts, hat aber dennoch Auswirkungen auf das Verfahren: bekanntlich existieren für Nullstellenprobleme von Polynomen vom Grad 4 oder mehr keine geschlossenen Lösungen, sondern die Nullstellen sind iterativ zu ermitteln. Aufgrund der Äquivalenz der Ausdrücke gilt dies auch für Eigenwerte/Eigenvektoren (*sonst wäre das Nullstellenproblem doch geschlossen lösbar*).

Das allgemeine Eigenwertproblem ist recht komplex zu lösen *(beispielsweise nützt hier der Gaussalgorithmus, von einer nicht sehr günstigen Iterationsmöglichkeit abgesehen, ausnahmsweise mal gar nichts*), aber glücklicherweise sind sehr viele Probleme mit einer symmetrischen Matrix verbunden. Bei diesen Matrizen lassen sich die Nichtdiagonalelemente einzeln mit Hilfe einfacher Drehmatrizen, wie wir sie bei der Computergrafik kennen lernen werden, auf Null bringen. Wie Drehmatrizen formal genau aussehen, können Sie dort nachschlagen. Allerdings erfordert es die Theorie, dass die Drehmatrizen in der Iteration in der Form

$$A^{(i+1)} = D_i^{-1} * A^{(i)} * D_i$$

eingesetzt werden,was nun wiederum in dieser einfachen Art nur bei symmetrischen Matrizen funktioniert.

Durch eine solche Operation wird zwar ein Nichtdiagonalelement auf beiden Seiten der Diagonale gelöscht, allerdings werden dabei Matrixelemente, die schon einmal Null waren, möglicherweise wieder reaktiviert. Die reaktivierten Elemente sind jedoch auf jeden Fall betragsmäßig kleiner als das verschwundene Element. Iterativ nähert sich der Inhalt der Nichtdiagonalelemente damit der Null, während auf der Diagonalen die Eigenwerte stehen bleiben und das Produkt aller Drehmatrizen die Eigenvektoren ergibt. Das führt auf den einfachen Algorithmus

```
template <typename T>
void eigenwert_J(Matrix<T> const& A,
                 Matrix<T>& ev, Matrix<T>& v){
  ...
  E1(v);
  cp.load(*max_element(a.begin(),a.end(),
                          abs_gt<T>()));
  while(true){
    for(i=0,k=0,l=0,tmax=null<T>();i<a.dim1();i++)
      for(j=0;j<i;j++)
        if(tmax<abs(a(i,j))){
            tmax=abs(a(i,j));k=i;l=j;
        }//endif
    if(cp.zero(tmax))
      break;
    E1(vn);
    te=(a(k,k)-a(l,l))/(a(k,l)*zwei<T>());
```

```
    t=sign(te)/(abs(te)+sqrt(eins<T>()+
                                    square(te)));
    c=eins<T>()/sqrt(eins<T>()+square(t));
    s=t*c;
    vn(k,k)=vn(l,l)=c; vn(k,l)=-s; vn(l,k)=s;
    mul(v,v,vn);
    tvn=vn;
    std::swap(tvn(k,l),tvn(l,k));
    a=mul(a,tvn,mul(a,a,vn));
  }//endfor
  ev.resize(a.dim1(),1);
  for(i=0;i<a.dim1();i++)
    ev(i,0)=a(i,i);
}//end function
```

Aufgabe. Diesmal ist die Aufgabenstellung anders herum als normalerweise. Mit der üblichen Entwicklungstaktik wie in den anderen Kapiteln hätte hier eine zu große Menge Mathematik stehen müssen, um Ihnen die Hintergründe verständlich zu machen. Deshalb hier der fertige Algorithmus mit minimaler Mathematik. Führen Sie ein „reverse Engineering“ an diesem Code durch und identifizieren Sie die Drehmatrizen, die Sie im Abschnitt über Computergraphik kennenlernen können. Stellen Sie den kompletten Algorithmus mathematisch zusammen.

Dieser Algorithmus ist zwar aus numerisch-mathematischer Sicht nicht die erste Wahl, ist aber leicht zu verstehen und hilft, eine Reihe einfacher Probleme erfolgreich anzugehen.