Technische Universität Chemnitz-Zwickau

DFG-Forschergruppe „SPC" · Fakultät für Mathematik

A. Meyer · M. Pester

Verarbeitung von Sparse-Matrizen in Kompaktspeicherform (KLZ / KZU)

Preprint-Reihe der Chemnitzer DFG-Forschergruppe „Scientific Parallel Computing"

SPC 94_12 Juni 1994

Inhaltsverzeichnis

1 Einleitung **1**

2 Die „Kompaktliste zeilenweise“ **1**

3 Implementation der Kompaktspeicherung **2**

4 Eine Auswahl wichtiger Bibliotheksroutinen **5**

4.1 `klz_init / awithj / jfroma -` Elementare Routinen 5
4.2 `axmklz / axmkzu -` Matrix-Vektor-Multiplikation 5
4.3 `axtkzu -` Matrix-Vektor-Multiplikation (transponiert) 5
4.4 `d_out_klz / d_out_kzu -` Diagonale extrahieren 6
4.5 `makekzu -` Matrix-Rahmen definieren 6
4.6 `hdiagkzu -` Hauptdiagonale initialisieren 6
4.7 `akkuij -` Akkumulation eines Matrix-Elements 7
4.8 `packklz / packkzu -` Verdichten einer Matrix 7
4.9 `sortkzu -` Sortieren von Matrix-Elementen 7
4.10 `cvbklz -` Konvertieren in Profilspeicherform 8
4.11 `mtvklz / mcvklz -` Teilmultiplikationen (`KLZ`) 8
4.12 `mtvkzu / mcvkzu -` Teilmultiplikationen (`KZU`) 9
4.13 `aitocklz -` Kopieren einer Teilmatrix 9
4.14 `maddklz -` Addition zweier Matrizen 9
4.15 `zerokzu -` Matrixelemente Null setzen 10
4.16 `ilukzu -` Unvollständige LU-Zerlegung 10
4.17 `vorkzu / ruekzu -` Lösen von Dreieckssystemen 10
4.18 `ic0klz -` Unvollständige Cholesky-Zerlegung 11
4.19 `micklz -` Modifizierte unvollständige Cholesky-Zerlegung 11
4.20 `rueklz / vorklz -` Lösen von Dreieckssystemen 11
4.21 `mafklz -` Diagonale der MAF-Vorkonditionierung 12
4.22 `vodklz / rudklz -` Lösen von Dreieckssystemen 12

5 Beispiel eines `KLZ`-Unterprogramms **13**

6 Verfügbarkeit der Bibliothek **14**

Arnd Meyer, Matthias Pester

Technische Universität Chemnitz-Zwickau
Fakultät für Mathematik
DFG-Forschergruppe SPC
Reichenhainer Straße 41
PSF 964
D-09009 Chemnitz

e-mail:
a.meyer@mathematik.tu-chemnitz.de
m.pester@mathematik.tu-chemnitz.de

VERARBEITUNG VON SPARSE-MATRIZEN IN KOMPAKTSPEICHERFORM (KLZ / KZU)

Arnd Meyer · Matthias Pester

1 Einleitung

Mit dieser Dokumentation soll eine an der TU Chemnitz seit vielen Jahren und in vielen Anwendungen erprobte Methode der Speicherung und Verarbeitung von schwachbesetzten Matrizen in Fortran einem breiteren Nutzerkreis zugänglich und verständlich gemacht werden.

Die Notwendigkeit zur Verwendung von Sparse-Matrix-Techniken in modernen numerischen Verfahren ist unumstritten, trotz zunehmender Speicherressourcen. Neben der ökonomischen Speicherausnutzung ist auch der Vorteil der Beschränkung der arithmetischen Operationen auf ein Minimum zu nennen. Häufig findet man spezielle Algorithmen für Matrizen mit Bandstruktur oder mit einem speziellen Profil. Wir betrachten hier eine Speicherform, bei der ausschließlich die Nichtnull-Elemente der Matrix in Form einer Liste gespeichert werden.

Nach einer Beschreibung der verwendeten Methode zur Sparse-Matrix-Speicherung werden einige der gebräuchlichsten verfügbaren Routinen vorgestellt.

2 Die „Kompaktliste zeilenweise“

Unter der Speicherform **KLZ** *Kompaktliste zeilenweise* wollen wir die zeilenweise Anordnung aller von Null verschiedenen Elemente a_{ij} des rechten oberen Dreiecks einer reellen symmetrischen Matrix A verstehen. Die gleiche Anordnung *aller* Nichtnull-Elemente einer *unsymmetrischen* Matrix bezeichnen wir zur besseren Unterscheidung mit **KZU**.

Wir betrachten eine schwachbesetzte Matrix $A \in \mathbb{R}^{n\times m}$. Die Anzahl der (*zu speichernden*) Nichtnull-Elemente sei *NNE*. Die zur Speicherung und Verarbeitung einer solchen Matrix notwendigen Informationen sind pro Element:

- der Wert des Elementes a_{ij},
- seine Position (i, j) innerhalb der Matrix A, d. h. Zeilen- und Spaltenindizes, sowie

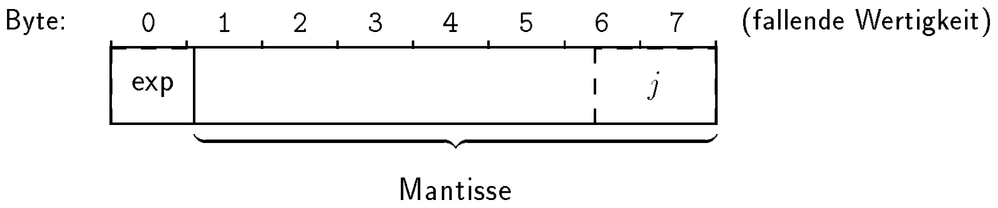
- seine Position k im Speicher, also innerhalb einer Liste `A` (der Länge *NNE*) aller Nichtnull-Elemente.

Bei einer zeilenweisen Anordnung der zu speichernden Elemente a_{ij} kann auf die explizite Speicherung der *NNE* Indizes i verzichtet werden; es genügt ein Vektor L der Länge n („*Leitvektor*") mit je einem Zeiger (Index) l_i auf das Ende (oder den Beginn) der Zeile i. Die Spaltenindizes j sind jedoch für jedes Element zu speichern, so daß im Normalfall ein Speicherbedarf von

$$(NNE + n) \times \texttt{Integer} + NNE \times \texttt{Real}$$

entsteht. Vielfach ist es aber (besonders für den Fortran-Programmierer) ein Handicap, zwei Listen mit einer im voraus unbekannten Länge *NNE* anzulegen.

In der hier beschriebenen Speicherform KLZ (bzw. KZU) wurden deshalb die Werte a_{ij} und j zu einem Listenelement $\mathtt{A}_k$ zusammengefaßt, indem der Index j in der rechnerinternen (`DoublePrecison-`) Darstellung den niederwertigen Teil der Mantisse von a_{ij} überlagert.



Die Reihenfolge der Bytes im Speicher ist jedoch maschinenabhängig (vgl. Abschnitt 3). Der durch die Kompaktspeicherung verursachte relative Fehler bei a_{ij} liegt (bei einer Mantissenlänge von 51 Bits) in der Größenordnung von

$$\varepsilon(n) = O\left(\frac{n}{2^{51}}\right)$$

(z. B. für $n = 100000$: $\varepsilon \approx 10^{-11}$) und damit grundsätzlich weit unter dem Eingangsfehler, der durch die Diskretisierung oder bereits durch das physikalische Modell entsteht. Für die meisten numerischen Anwendungen spielt daher eine solche geringfügige Störung der Matrixelemente keine Rolle. Für Rechnungen, die exakte Daten voraussetzen, ist diese Methode jedoch nicht zu empfehlen.

Mit dieser Vorgehensweise reduziert sich zugleich der Speicherbedarf auf

$$n \times \texttt{Integer} + NNE \times \texttt{Real}$$

für den Leitvektor L und die Liste `A` der Nichtnull-Elemente.

Der Leitvektor L sei so definiert, daß dessen Element L_i die Position k des **letzten** gespeicherten Elementes der Zeile i angibt.

3 Implementation der Kompaktspeicherung

Ein entscheidender Nachteil der ersten Implementationen dieser Methode unter Verwendung der Fortran-`EQUIVALENCE`-Anweisung war die Abhängigkeit von der rechnerinternen Zahlendarstellung. Bekanntlich unterscheiden sich verschiedene Prozessortypen bzw.

Real-Formate auch in der Anordnung der Bytes innerhalb des Adreßraumes der Maschine. So waren die beiden gängigen Varianten für auf- und absteigende Bytefolge (hier z.B. für $n \leq 32767$):

```
DOUBLE PRECISION  A(*), AIJ
INTEGER*2         JH
EQUIVALENCE       (AIJ,JH)
```

bei aufsteigender, bzw.

```
DOUBLE PRECISION  A(*), AIJ
INTEGER*2         JH, J4(4)
EQUIVALENCE       (AIJ,J4(1)), (J4(4),JH)
```

bei absteigender Byte-Anordnung (bzgl. der Wertigkeit). In beiden Fällen könnten die Matrixelemente dann in folgender Art und Weise verarbeitet werden:

```
 ...
AIJ  = A(k)
J    = JH          !  Index "auspacken"
 ...
AIJ  = <neuer Wert>
JH   = J           !  Index "einpacken"
A(k) = AIJ
```

Der nächste Schritt zur Verbesserung wurde durch die in Fortran77 vorhandenen bitorientierten logischen Verknüpfungen möglich:

```
DOUBLE PRECISION  A(*), AIJ
INTEGER*4         JOVL(2), Mask_0, Mask_1
EQUIVALENCE       ( AIJ,JOVL(1) )
DATA              Mask_0 / Z FFFF 8000 /
DATA              Mask_1 / Z 0000 7FFF /
DATA              I_ovl  / 2 /
 ...
AIJ  = A(k)
J    = IAND ( JOVL(I_ovl), Mask_1 )
 ...
AIJ  = <neuer Wert>
JOVL(I_ovl) = IOR ( J, IAND(JOVL(I_ovl),Mask_0 )
A(k) = AIJ
 ...
```

Diese Variante ist wiederum für die absteigende Wertigkeit der Bytes im Speicher gültig. Bei der anderen Variante sind nur die drei DATA-Zeilen auszutauschen:

```
DATA              Mask_0 / Z 0080 FFFF /
DATA              Mask_1 / Z FF7F 0000 /
DATA              I_ovl  / 1 /
```

Allerdings ist auch die Schreibweise der Hexadezimalkonstanten nicht einheitlich definiert. Um diese Maschinenabhängigkeit zu vermeiden, wurde eine Initialisierungsroutine geschaffen, die zunächst die für die konkrete Hardware zu verwendende Speichertechnik automatisch erkennt und in Abhängigkeit von der vorgesehen Matrixordnung n die erforderlichen Bitmasken erzeugt:

- `call KLZ_INIT(N)`

Nun soll der Nutzer auch von der Notwendigkeit befreit werden, die oben erwähnten `EQUIVALENCE`-Anweisungen verwenden zu müssen. Dazu stehen für den Zugriff auf ein Element der Matrix die folgenden beiden Funktionen zur Verfügung:

mit der Vereinbarung:
`DOUBLE PRECISION  AwithJ, AA, A(*)`

- `J = JfromA ( A(K) )`
  liefert den Spaltenindex aus dem gepackten Matrix-Element.
- `A(K) = AwithJ ( AA,J )`
  packt den `Real`-Wert `AA` mit dem Index `J` in das Element `A(k)`.

Somit wird im Hinblick auf die Fortran-Realisierung eine symmetrische KLZ-Matrix (bzw. eine unsymmetrische KZU-Matrix) durch drei Parameter `N, A, L` beschrieben. Ein kurzes Programmstück veranschaulicht deren Verwendung zur Erzeugung einer einfachen Matrix, z. B.:

$$\begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & 4 & -1 \\ & & & -1 & 4 \end{pmatrix}$$

Wenngleich solche einfach strukturierten Matrizen nicht der typische Anwendungsfall für diese Speicherform sind, wird die prinzipielle Verfahrensweise damit klar:

```
      Subroutine KLZ_Matrix ( N, A, L )
      double precision  A(*), AwithJ
      integer           L(N), i, k
      call KLZ_Init(N)          ! spaetestens hier initialisieren
      k=0
      DO 1 i = 1,N
        k=k+1
        A(k) = AwithJ(4d0,i)    ! Spaltenindizes einpacken
        IF (i .LT. N) THEN
          k=k+1                 ! nicht zu viele Elemente!
          A(k) = AwithJ(-1d0,i+1)
        ENDIF
        L(i) = k                ! Zeilen-Ende markieren
1     CONTINUE
      return
      end
```

Mit *NNE* =`L(N)` steht danach für das rufende Programm die Anzahl der auf `A` gespeicherten Elemente fest.

4 Eine Auswahl wichtiger Bibliotheksroutinen

Bei der folgenden kurzen Darstellung der einzelnen Routinen beziehen sich die Kürzel `KLZ` bzw. `KZU` entsprechend auf eine symmetrische bzw. unsymmetrische nach oben beschriebenem Verfahren gespeicherte schwachbesetzte Matrizen, wobei die zeilenweise Darstellung symmetrischer Matrizen die Elemente des **oberen** Dreiecks enthält. Teilweise verwenden diese Routinen Vektoroperationen der Bibliothek `libvbasmod.a`[1].

4.1 `klz_init / awithj / jfroma` - Elementare Routinen

Initialisierung : `call klz_init (N)`

Die maximal vorgesehene Spaltenzahl für Matrizen in Kompaktspeicherung wird vorgegeben. Intern werden die entsprechenden Bitmasken bereitgestellt.
Wichtig! - Vor Aufruf dieses Programms sind alle anderen nicht verwendbar!

Spaltenindex einpacken: `A(k) = AwithJ(AA,J)`

Der Wert der `REAL*8`-Variablen `A` wird zusammen mit dem Spaltenindex `J` zu einem `REAL*8`-Wert gepackt. `AwithJ` ist mit `DOUBLE PRECISION` zu deklarieren.

Spaltenindex auspacken: `J = JfromA(AA)`

Dem `REAL*8`-Element einer `KLZ/KZU`-Matrix wird der Spaltenindex entnommen.

4.2 `axmklz / axmkzu` - Matrix-Vektor-Multiplikation

Rufzeile: `call axmklz / axmkzu (Add,N,A,L,X,Y)`

Parameter:

- `Add` - Steuergröße in Form eines Zeichens, `Add='0'` (Zeichen *Null*) zeigt an, daß der Zielvektor `Y` vor Ausführung der Operation Null gesetzt wird.
- `N` - Vektorlänge bzw. Matrixdimension
- `A,L` - Matrix in Kompaktspeicherung mit zugehörigem Leitvektor
- `X` - Vektor
- `Y` - Ergebnisvektor

Funktion: Multiplikation der Matrix `A` mit dem Vektor `X`:

`Y := Y + A*X` oder `Y := A*X` `(bei Add='0')`

4.3 `axtkzu` - Matrix-Vektor-Multiplikation (transponiert)

Rufzeile: `call axtkzu (Add,N,A,L,X,Y)`

Parameter: siehe `axmklz` (4.2)

Funktion: Multiplikation der Matrix $\mathtt{A}^\top$ mit dem Vektor `X`:

$\mathtt{Y := Y + A^\top *X}$ oder $\mathtt{Y := A^\top *X}$ `(bei Add='0')`

Bemerkung: Der Parameter `Add='0'` darf hier nur für quadratische Matrizen angewendet werden, da sonst die Länge des Vektors `Y` nicht im voraus bekannt ist.

[1] siehe **SPC 94_4** (Bibliotheken zur Entwicklung paralleler Algorithmen)

4.4 d_out_klz / d_out_kzu - Diagonale extrahieren

Rufzeile: `call d_out_klz / d_out_kzu (N,A,L,D)`

Parameter:

- `N` - Vektorlänge bzw. Matrixdimension
- `A,L` - Matrix in Kompaktspeicherung mit zugehörigem Leitvektor
- `D` - Vektor, auf den die Diagonale von `A` gespeichert wird

Funktion: Aus der Matrix `A` wird die Hauptdiagonale auf den Vektor `D` kopiert (z. B. zwecks Jacobi-Vorkonditionierung):
`D := diag(A)`

4.5 makekzu - Matrix-Rahmen definieren

Rufzeile: `call makekzu (NZ,N,L,A,MAX,IER)`

Parameter:

- `NZ` - vorgesehene maximale Anzahl der Nichtnull-Elemente pro Zeile
- `N` - Anzahl der Zeilen der Matrix
- `L` - `INTEGER`-Feld (Länge `N`), auf dem der Leitvektor zurückgegeben wird
- `A` - `REAL*8`-Feld, das durch `makekzu` initialisiert wird
- `MAX` - Länge des Feldes `A` (Anzahl `REAL*8`-Elemente)
- `IER` - Fehleranzeiger, wird auf 1 gesetzt, wenn Speicherplatz (`MAX`) nicht ausreicht, sonst 0.

Funktion: Das Feld `A` wird mit `N*NZ` Nullen gefüllt und der Leitvektor `L` wird initialisiert, als hätte die Matrix in jeder Zeile genau `NZ` Nichtnull-Elemente. Diese Routine dient als Vorbereitung für `akkuij` (4.7). Später müssen die in `A` verbliebenen Nullen durch Komprimieren der Matrix mittels `packklz` bzw. `packkzu` (4.8) eliminiert werden.

4.6 hdiagkzu - Hauptdiagonale initialisieren

Rufzeile: `call hdiagkzu (N,L,A)`

Parameter:

- `N` - Anzahl der Zeilen der Matrix
- `L` - Leitvektor von `A`
- `A` - `KZU`-Matrix (durch `makekzu` definiert)

Funktion: Die Diagonalelemente der `KLZ/KZU`-Matrix werden mit einem sehr kleinen Wert (`1D-25`) belegt. Damit wird sichergestellt, daß bei nachfolgender Akkumulation (`akkuij`) die Diagonalelemente jeweils als erste Elemente der Zeile gespeichert sind. Bei der Generierung symmetrischer Matrizen (`KLZ`) wird damit auch ausgeschlossen, daß Leerzeilen (ohne gespeichertes Element) entstehen, wodurch einige der `KLZ`-Unterprogramme versagen könnten. In diesen wird meist vorausgesetzt, daß pro Zeile das Diagonalelement als erstes auch tatsächlich gespeichert ist.

4.7 `akkuij` - Akkumulation eines Matrix-Elements

Rufzeile: `call akkuij (NZ,NS,L,A,I,J,T,IER)`

Parameter:

- `NZ` - Anzahl der Zeilen von `A`
- `NS` - Anzahl der Spalten von `A`
- `L` - Leitvektor zu `A`
- `A` - `KLZ/KZU`-Matrix
- `I,J` - Index des zu akkumulierenden Elements a_{ij}
- `T` - Wert, der zu a_{ij} zu addieren ist
- `IER` - Fehleranzeiger:
  - `IER=1`: einer der Indizes `I,J` liegt außerhalb der zulässigen Grenzen
  - `IER=2`: Die laut Leitvektor vorgesehene maximale Anzahl der Elemente pro Zeile wurde überschritten,

Funktion: In den durch `makekzu` reservierten Rahmen einer `KZU`-Matrix werden durch wiederholten Aufruf von `akkuij` die Matrixelemente eingetragen bzw. aufsummiert.

4.8 `packklz / packkzu` - Verdichten einer Matrix

Rufzeile: `call packklz / packkzu (N,L,A)`

Parameter:

- `N` - Anzahl der Zeilen der Matrix
- `L` - Leitvektor zu `A`
- `A` - Input: `KZU`-Matrix, die noch Nullen enthalten kann
  - Output: verdichtete `KLZ`- bzw. `KZU`-Matrix

Funktion: Falls in der `KZU`-Matrix `A` noch Nullen enthalten sind, werden diese eliminiert. Im Fall von `packklz` bleibt nur der rechte obere Teil der (als symmetrisch vorausgesetzten) Matrix erhalten. Der Leitvektor `L` wird entsprechend aktualisiert.

4.9 `sortkzu` - Sortieren von Matrix-Elementen

Rufzeile: `call sortkzu (N,L,A)`

Parameter:

- `N` - Zeilenanzahl von `A`
- `L` - Leitvektor zu `A`
- `A` - `KZU`- oder `KLZ`-Matrix

Funktion: Innerhalb der Zeilen der Matrix A werden die Elemente nach aufsteigendem Spaltenindex sortiert.

Bemerkung: Durch Anwendung von `akkuij` entstehen die Elemente innerhalb der Zeilen im allgemeinen in ungeordneter Reihenfolge. Eine Sortierung ist auch beispielsweise für die Matrix-Vektor-Multiplikation nicht notwendig. Dagegen setzen andere

Algorithmen (unvollständige Zerlegungen) der Einfachheit halber eine solche Anordnung voraus.

4.10 `cvbklz` - Konvertieren in Profilspeicherform

Rufzeile: `call cvbklz (Aout,Ain,L,N,Max,IER)`

Parameter:

`Aout` - Output: Matrix `A` in Speicherform `VBZ`
`Ain` - Input: Matrix `A` in Speicherform `KLZ`
`L` - Input: Leitvektor zur `KLZ`-Matrix `A`
- Output: Leitvektor zur `VBZ`-MAtrix `A`
`Max` - verfügbare Gesamtlänge des Feldes `Aout` (in Doppelworten)
`IER` - Fehleranzeiger: ungleich Null bedeutet, daß der Speicherplatz (`Max`) nicht ausreichte.

Funktion: Die kompakt gespeicherte Matrix A wird in Profilspeicherung (`VBZ` = variable Bandweite zeilenweise) überführt, indem zeilenweise alle Nullen zwischen Diagonalelement und äußerstem Nichtnull-Element aufgefüllt werden. Das Programm wurde so implementiert, daß die Parameter `Aout` und `Ain` beim Aufruf identisch sein können. Der Leitvektor `L` wird gemäß der neuen Speicherbelegung verändert.

Bemerkung: Für die Speicherform `VBZ` ist es offensichtlich nicht notwendig, die Spaltenindizes explizit mit abzuspeichern. Sie ist speziell für Algorithmen mit Fill-In besser geeignet als `KLZ`, benötigt aber bedeutend mehr Platz.

4.11 `mtvklz / mcvklz` - Teilmultiplikationen (`KLZ`)

Rufzeile: `call mtvklz(NC,YC,XC,A,L,ZI)`
`call mcvklz(NI,NC,A,L,XC,YI)`

Parameter:

`NC` - Dimension des oberen Diagonalblocks
`NI` - Dimension des unteren Diagonalblocks
`XC` - Vektor der Länge `NC`
`ZI` - Vektor der Länge `NI`
`Y` - Ergebnis der Multiplikation bei `mcvklz` (Länge `NC+NI`)
`YC` - Ergebnis der Multiplikation bei `mtvklz` (Länge `NC`)
`L` - Leitvektor zu `A`
`A` - `KLZ`-Matrix (erste `NC` Zeilen)

Funktion: Diese beiden Unterprogramme führen jeweils eine Teil-Matrixmultiplikation von `Y = A*X` aus, wobei `A` eine symmetrische Matrix (in Kompaktspeicherung `KLZ`) der folgenden Art ist:

$$A = \begin{pmatrix} A_{CC} & A_{CI} \\ A_{IC} & A_{II} \end{pmatrix},$$

mit $A_{CC}:\ \mathtt{NC}\times\mathtt{NC},\ A_{CI}:\ \mathtt{NC}\times\mathtt{NI},\ A_{II}:\ \mathtt{NI}\times\mathtt{NI},\ A_{IC}=A_{CI}^{\top}$.

Damit berechnet

$$\begin{array}{lrcl} \texttt{mtvklz:} & Y_C & := & X_C - A_{CI} * Z_I \qquad \text{und} \\ \texttt{mcvklz:} & Y & := & \begin{pmatrix} A_{CC} \\ A_{IC} \end{pmatrix} * X_C \end{array}$$

Bemerkung: Diese Operationen treten bei der Verwendung des Schurkomplements auf. Die Teilmatrix A_{II} wird für beide Routinen nicht benötigt, sie wird im allgemeinen wegen der besseren Behandlung als `KLZ`-Matrix separat gespeichert (d. h. mit *eigenem* Leitvektor, vgl. 4.13)

4.12 `mtvkzu / mcvkzu` - Teilmultiplikationen (`KZU`)

Rufzeile:
```
call mtvkzu(NC,YC,XC,A,L,ZI)
call mcvkzu(NI,NC,A,L,XC,YI)
```

Funktion: Parameter und Funktion entsprechen denen von (4.11), allerdings für die unsymmetrische Speicherform `KZU`.

4.13 `aitocklz` - Kopieren einer Teilmatrix

Rufzeile: `call aitocklz (NC,N,A,LA,C,LC)`

Parameter:

- `NC` - Anzahl der Zeilen des oberen Diagonalblocks
- `N` - Anzahl der Zeilen der Matrix `A` insgesamt
- `A` - `KLZ`-Matrix mit `N` Zeilen
- `LA` - Leitvektor zu `A` (Länge `N`)
- `C` - Output: `KLZ`-Matrix mit `NI=N-NC` Zeilen
- `LC` - Output: Leitvektor zu `C` (Länge `NI=N-NC`)

Funktion: Kopiert die Teilmatrix A_{II} (vgl. 4.11) aus A heraus und generiert den dazugehörigen Leitvektor `LC`

Bemerkung: Der Aufruf ist auch ohne zusätzlichen Speicherplatz für `C` und `LC` möglich:
```
call aitocklz ( NC, N, A, L, A(L(NC)+1), L(NC+1) )
```
In dem zu A_{II} gehörenden Teil von `A` bzw. `L` werden dabei nur die Spaltenindizes sowie der entsprechende Teil des Leitvektors modifiziert.

4.14 `maddklz` - Addition zweier Matrizen

Rufzeile: `call maddklz (N,L,A,B,alpha,C)`

Parameter:

- `N` - Anzahl der Zeilen der Matrizen `A,B,C`
- `L` - Leitvektor zu `A, B` und `C`
- `A,B,C` - `KLZ`- oder `KZU`-Matrizen gleicher Struktur
- `alpha` - skalarer Faktor

Funktion: Berechnung von `A=B+alpha*C` für Matrizen mit gleichem Besetzungsmuster.

4.15 `zerokzu` - Matrixelemente Null setzen

Rufzeile: `call zerokzu (N,L,A)`

Parameter:

- `N` - Anzahl der Zeilen der Matrix
- `L` - Leitvektor zu `A`
- `A` - `KLZ-` oder `KZU`-Matrix

Funktion: Unter Beibehaltung der Spaltenindizes einer bereits definierten Matrix `A` werden alle Elemente „*Null*" gesetzt (zur Vermeidung unerwünschter Effekte wird `1D-30` statt Null verwendet), z. B. als Start für nachfolgende Addition von Matrizen gleicher Struktur.

4.16 `ilukzu` - Unvollständige LU-Zerlegung

Rufzeile: `call ilukzu (N,A,LA,omega,IER)`

Parameter:

- `N` - Dimension der Matrix `A`
- `A` - Input: `KZU`-Matrix
  - Output: Faktoren `L` und `U`: `L·U` $\approx$ `A`
- `LA` - Leitvektor zu `A`
- `omega` - skalarer Faktor, (`REAL*4`), $0 \leq \omega \leq 1$
- `IER` - Fehleranzeiger: Ein Wert $i > 0$ zeigt an, daß das Element u_{ii} zu klein wurde, d. h.: ILU nicht durchführbar.

Funktion: Berechnung der einfachen (`omega=0.0`) oder modifizierten (`omega=1.0`) unvollständigen LU-Zerlegung für eine unsymmetrische (`KZU-`) Matrix. Das ω-fache des vernachlässigten Fill-In wird in den Diagonalelementen berücksichtigt.
Als Diagonalelemente werden die reziproken Werte u_{ii}^{-1} gespeichert.

4.17 `vorkzu / ruekzu` - Lösen von Dreieckssystemen

Rufzeile: `call vorkzu / ruekzu (N,A,L,X,Y)`

Parameter:

- `N` - Dimension der Matrix `A`
- `A` - `KZU`-Matrix, die die Faktoren L und U enthält.
- `L` - Leitvektor zu `A`
- `X` - Lösungsvektor
- `Y` - rechte Seite

Funktion: Durch `vorkzu` bzw. `ruekzu` werden entsprechend die Gleichungssysteme

$$Lx = y \quad \text{bzw.} \quad Rx = y$$

durch Vorwärts- bzw. Rückwärtseinsetzen gelöst (z. B. für `ILU`). Die obere Dreiecksmatrix R und die untere Dreiecksmatrix L sind gemeinsam als `KZU`-Matrix `A` gespeichert (mit reziproken Elementen u_{ii}^{-1} auf der Diagonalen).

4.18 `ic0klz` - Unvollständige Cholesky-Zerlegung

Rufzeile: `call ic0klz (N,A,B,L,eps,IPOD)`

Parameter:

- `N` - Dimension der Matrix `A`
- `A` - Input: `KLZ`-Matrix
- `B` - Output: Cholesky-Faktor `R` mit gleichem Besetzungsmuster wie `A`
- `L` - Leitvektor zu `A` und `B`
- `eps` - Fehlerschranke für Diagonalelemente, (`REAL*8`)
- `IPOD` - Fehleranzeiger: Ein Wert $i > 0$ zeigt an, daß bei Berechnung von r_{ii} ein Fehler auftrat ($r_{ii}^2 < \varepsilon$)

Funktion: Berechnung der einfachen unvollständigen Cholesky-Zerlegung für eine symmetrische (`KLZ-`) Matrix $A \approx RR^\top$ (nicht $R^\top R = LL^\top$!). Als Diagonalelemente werden die reziproken Werte r_{ii}^{-1} gespeichert.

4.19 `micklz` - Modifizierte unvollständige Cholesky-Zerlegung

Rufzeile: `call micklz (N,A,B,L,eps,IPOD,delta)`

Parameter:

- `N,A,B,L,eps,IPOD` - wie `ic0klz` (4.18)
- `delta` - skalare Größe zum Anheben der Diagonalelemente a_{ii} mit dem Faktor $(1 + \delta)$; es wird empfohlen, für Operatoren 2. Ordnung $\delta \approx h^{-2}$ zu verwenden ($\approx N^{-1}$ für 2D- und $N^{-\frac{2}{3}}$ für 3D-Probleme).

Funktion: Berechnung der modifizierten unvollständigen Cholesky-Zerlegung für eine symmetrische (`KLZ`) Matrix $A \approx RR^\top$. Als Diagonalelemente werden die reziproken Werte r_{ii}^{-1} gespeichert.

4.20 `rueklz / vorklz` - Lösen von Dreieckssystemen

Rufzeile: `call rueklz / vorklz (N,A,L,X,Y)`

Parameter:

- `N` - Dimension der Matrix `A`
- `A` - `KLZ`-Matrix
- `L` - Leitvektor zu `A`
- `X` - Lösungsvektor
- `Y` - rechte Seite

Funktion: Durch `rueklz` bzw. `vorklz` werden entsprechend die Gleichungssysteme

$$Rx = y \quad \text{bzw.} \quad R^\top x = y$$

durch Rückwärts- bzw. Vorwärtseinsetzen gelöst (z. B. für `MIC` oder `IC0`). Die obere Dreiecksmatrix R ist als `KLZ`-Matrix `A` gespeichert (mit reziproken Elementen auf der Diagonalen).

4.21 `mafklz` - Diagonale der MAF-Vorkonditionierung

Rufzeile: `call mafklz (N,A,L,D)`

Parameter:

`N` - Dimension der Matrix `A`
`A` - `KLZ`-Matrix
`L` - Leitvektor zu `A`
`D` - Output: Vektor zur MAF-Vorkonditionierung

Funktion: Für die MAF-Vorkonditonierung mit $C = (D + R^\top)D^{-1}(D + R)$ wird D so bestimmt, daß $Ae \approx Ce$, mit $e = (1, 1, \ldots, 1)^\top$.
(entspricht MIC(0)$*$ nach Gustafsson)

4.22 `vodklz / rudklz` - Lösen von Dreieckssystemen

Rufzeile: `call vodklz / rudklz (N,A,L,D,X,Y)`

Parameter:

`N` - Dimension der Matrix `A`
`A` - `KLZ`-Matrix
`L` - Leitvektor zu `A`
`D` - Vektor, der als Diagonalmatrix zu verwenden ist
`X` - Lösungsvektor
`Y` - rechte Seite

Funktion: Durch `vodklz` bzw. `rudklz` werden entsprechend die Gleichungssysteme

$$(D + R^\top)x = y \quad \text{bzw.} \quad (D + R)x = y$$

durch Vorwärts- bzw. Rückwärtseinsetzen gelöst (z. B. für MAF). Als Matrix R wird das obere Dreieck (ohne Diagonale) der `KLZ`-Matrix `A` verwendet.

5 Beispiel eines KLZ-Unterprogramms

Die Erfahrung zeigt, daß ein Beispiel oft mehr Klarheit schafft als mehrseitige Erklärungen. Betrachten wir deshalb eine Möglichkeit, die Addition zweier KLZ-Matrizen zu realisieren:

```
      SUBROUTINE MaddKLZ (N,L,A,B,alfa,C)
      DOUBLE PRECISION A(*),B(*),C(*),alfa,AwithJ,AA
      INTEGER  L(*),N,LU,LO,K,I,J,JfromA

      LU = 1                        ! Diagonale
      DO 50 K=1,N
        LO=L(K)                     ! Ende der Zeile
        IF (LU.GT.LO) GOTO 50
        DO 30 I=LU,LO
          AA = B(I)
          J  = JfromA(AA)           ! Spaltenindex merken
          AA = AA + alfa*C(I)
          A(I) = AwithJ(AA,J)       ! und wieder speichern
   30   CONTINUE
   50 LU = LO + 1
      RETURN
      END
```

Für den „Fortgeschrittenen“, der gern auf die Funktionsaufrufe `JfromA` und `AwithJ` verzichten möchte, hier die etwas komplizierter scheinende Realisierung als Inline-Version, die zu obiger aber völlig äquivalent ist:

```
      SUBROUTINE MaddKLZ(N,L,A,B,alfa,C)
      DOUBLE PRECISION   A(*),B(*),C(*),alfa,AA
      INTEGER  L(*),N,LU,LO,K,I,J
      INTEGER  JAA(2)
        include 'klz.inc'      ! Common-Variablen, die durch
                               ! klz_init definiert sind
      EQUIVALENCE (AA,JAA(1))  ! Ueberlagerung REAL/INTEGER

      LU = 1
      DO 50 K=1,N
        LO=L(K)
        IF (LU.GT.LO) GOTO 50
        DO 30 I=LU,LO
          AA = B(I)
          J  = IAND(JAA(I_OVL),MASK_1)      ! statt JfromA
          AA = AA + alfa*C(I)
          JAA(I_OVL)=IOR(J,IAND(JAA(I_OVL),MASK_0))
          A(I) = AA                         ! statt AwithJ
   30   CONTINUE
   50 LU = LO + 1
      RETURN
      END
```

6 Verfügbarkeit der Bibliothek

Die in Abschnitt 4 aufgelisteten Unterprogramme stehen als Bibliothek `libKLZ.a` zur Verfügung. Durch Verwendung der Initialisierungsroutine konnte auch die Portabilität der Routinen gewährleistet werden. Die Funktionsfähigkeit (ohne Quelltextänderung) wurde bisher auf allen gängigen Prozessortypen nachgewiesen. Dazu gehören PC's mit 80x86/87-Prozessor, Workstations verschiedener Fabrikate, Transputer, nCube-2s und i860-Systeme.

Eine geringe Modifikation der Initialisierungsroutine kann notwendig sein, wenn die Grundvoraussetzung verletzt ist, daß eine `DOUBLE PRECISION`-Variable den Speicherplatz von zwei `INTEGER`-Variablen umfaßt. So ist es beispielsweise beim 64-Bit-Prozessor von Kendall Square Research (KSR), wo eine `INTEGER`-Variable 8 Byte lang ist und die 16-Byte-Option für `DOUBLE PRECISION` wegen der vergleichsweise unzumutbaren Rechenzeit nicht verwendet werden sollte.