Compilation I

Gesamtsteifigkeitsmatrix = Summe der Elementsteifigkeitsmatrizen

$$\boldsymbol{K} = \sum_i \boldsymbol{K}_i^e$$

Die Compilation muß

* vollautomatisch erfolgen
* verschiedene Elementtypen mit unterschiedlichen Knotenanzahlen einbauen können
* unterschiedliche Anzahlen von Freiheitsgraden an den Knoten einer Struktur berücksichtigen
* die Gesamtsteifigkeitsmatrix möglichst speichersparend aufbauen

Compilation II

1. Datensatz: Allgemeine Angaben zur Struktur wie

- Dimension des Problems (d.h. 2D oder 3D)
- Anzahl der Knoten
- Anzahl der finiten Elemente
- Anzahl der Freiheitsgrade
- Anzahl Elastizitätsgesetze
- Steuerflags für Koordinatentransformationen usw.

2. Datensatz: Knotenkoordinaten wie

- Knotennummer
- Anzahl der Freiheitsgrade an diesem Knoten
- X-Koordinate
- Y-Koordinate
- Z-Koordinate

3. Datensatz: Koinzidenzliste, besteht aus zwei Zeilen und enthält

Zeile 1: Elementnummer und Elementtyp

Zeile 2: Die Knotennummern

Compilation III

Der an sich nötige vierte Datensatz (Materialgesetze) kann zunächst entfallen, da er für die folgenden Betrachtungen nicht wichtig ist.

Diesem „Strickmuster“ folgen eigentlich alle klassischen FEA-Programme – auch in der Reihenfolge. Der erste Datensatz liefert in der Hauptsache die Zähler für die Schleifen, die für den zweiten und dritten Datensatz ablaufen. Man kann den ersten Datensatz auch weglassen und statt dessen den folgenden Datensätzen Schlüsselworte wie sinngemäß KOOR und KOIN voranstellen, damit das Programm erkennt, was mit dieser Datenzeile zu tun ist. Aber weitaus schneller geht es beim Einlesen, wenn man im ersten Datensatz z.B. 1000 Knoten vorgibt und daraus dann folgend in der Sektion 2.Datensatz 1000-mal eine Zeile einliest. Das ist ganz FORTRAN-gemäß, aber auch Sprachen wie C profitieren von einem derartigen Aufbau, denn gerade das Dekodieren von Schlüsselworten (sprich Characterverarbeitung) dauert relativ lange.

Compilation IV

Das Abspeichern der Knotenkoordinaten ist einfach und geradlinig: Hier können wir drei Vektoren mit z.B. 1.000 Speicherplätzen definieren, also

```
x[1000],  y[1000],  z[1000]
```

und man könnte damit 1.000 Koordinatentripel abspeichern. Besser:

```
x[1001],  y[1001],  z[1001]
```

Oder dynamisch:

```
double *x, *y, *z;
x= (double *) calloc (1001, sizeof(double));
y= (double *) calloc (1001, sizeof(double));
z= (double *) calloc (1001, sizeof(double));
```

Compilation V

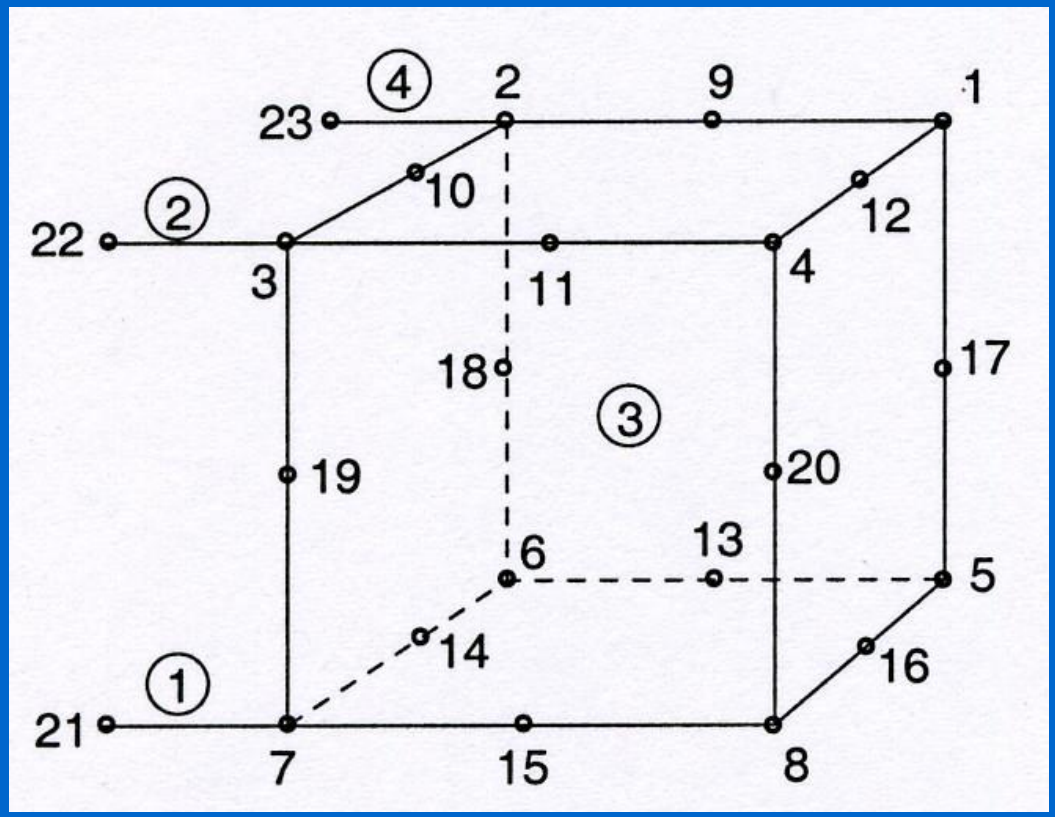
Zu jedem Knoten die Anzahl der Freiheitsgrade abspeichern:

```
ifrei[1001]
```

Beim Abspeichern der Koinzidenz wird es schon etwas schwieriger, denn zu jedem Finiten Element gehören einerseits immer mehrere Knoten und ferner weisen unterschiedliche Finite Elementetypen unterschiedliche Knotenanzahlen auf. Es ist daher geschickt, zwei Vektoren zu definieren:

Der erste Vektor `koi` enthält die Koinzidenzliste und der zweite Vektor `koffs` ist ein Pointer darauf, um die Adreßberechnung durchführen zu können.

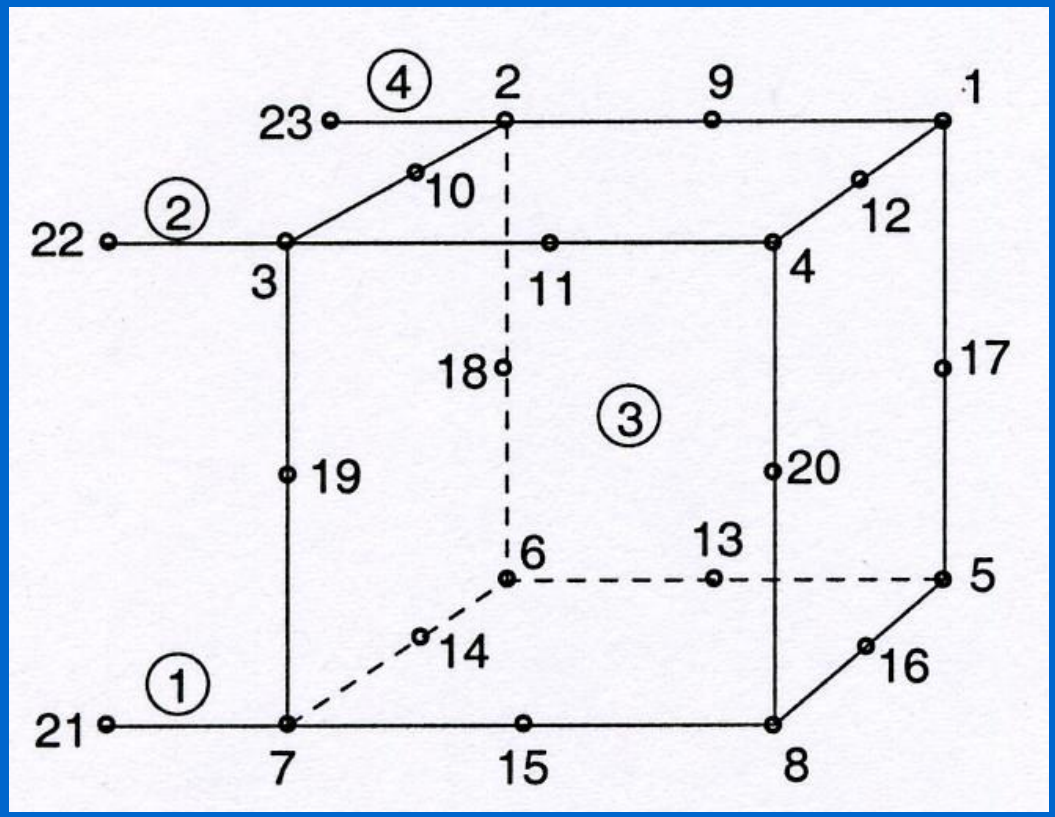
Compilation VI



Einmal angenommen, wir würden in unserer Struktur mit zwei Stäben beginnen, d.h. pro Element zwei Knoten. Dann enthält der Vektor koi als erstes Vektorelement eine Null, weil wir ja beim Index 1 beginnen wollen. Vektorelemente 2 und 3 enthalten die beiden Knotennummern des ersten Stabes, Vektorelemente 4 und 5 die beiden Knotennummern des zweiten Stabes und so fort. Bis dahin bräuchten wir noch keinen Offset-vektor.

Was wäre aber, wenn als drittes finites Element kein Stab folgen würde, sondern ein Hexaeder mit 20 Knoten – was ohne weiteres zulässig wäre? Hier setzt unser Offsetvektor `koffs` ein: Sein erstes Element ist 0, sein zweites Element immer 1,

Compilation VII



das dritte Element zeigt auf den Startpunkt des zweiten finiten Elements im Koinzidenzvektor `koi`, das vierte Vektorelement zeigt auf den Startpunkt des dritten finiten Elements im Koinzidenzvektor `koi` und so fort. Nehmen wir nun noch an, daß unser viertes finites Element wieder ein Stab sei. Dann wird:

Zugriff auf einen Knoten `j` des finiten Elements `i`:

```
koi[koffs[i] + j –1]
```

```
koi={0,21,7,22,3,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
15,16,17,18,19,20,   23,2...}
koffs = {0, 1, 3, 5, 25, ....}
```

UNIVERSITÄT BAYREUTH

Speicherverfahren I

Für FEA_Programme extrem wichtig:
Dynamische Speicheranforderung, weil Speicherbedarf selbst für einzelne Bauteile im Gbyte-Bereich!
Geht gut mit C, C++, FORTRAN > 90.

```
DYNAMIC START
  COMMON START
      MAXGS 1000000
      MAXKOI  20000
      MAXK     4000
      MAXE     2000
      MAXNFG  10000
  COMMON END
DYNAMIC END
```

Mit F77: Man hat den größten Teil seines FEA-Programms im Objectcode vorliegen. Lediglich eine kleine F77-Subroutine liegt im Quellcode vor. Hier trägt der Preprozessor die Speicherbedarfe für den aktuellen Berechnungsfall ein; anschließend wird diese kleine Subroutine automatisch und für den Benutzer unsichtbar übersetzt und mit dem Programmrest, der schon compiliert vorliegt, gelinkt. Sehr raffiniert. Mit diesem Trick arbeitet z.B. das Großprogramm *MARC*.

Speicherverfahren II

Bei einer mittelgroßen Raumstruktur mit, sagen wir 20.000 Knoten, wäre der theoretische Speicherbedarf (20.000 * 3 Freiheitsgrade * 20.000 * 3 Freiheitsgrade * 8 Bytes = 28,8 Gbyte.

Damit ist ersichtlich, daß die üblichen Matrizenmethoden, wie sie in den mathematischen Lehrbüchern zu finden sind, völlig ungeeignet für Finite Elemente Analysen sind. Man muß die Eigenheiten der FEA-Gleichungs-systeme kennen und ausnutzen, um Speicherplatz im großen Stil zu sparen. Es sind für FEA-Anwendungen im wesentlichen drei grundlegend unterschiedliche Verfahren bekannt geworden:

- Die Bandspeicherung
- Das Skyline-Verfahren oder als Abwandlung, das Jennings-Verfahren
- Speichern der Nicht-Null Elemente

Ganz grob: Von oben nach unten steigt die Programmkomplexität massiv an, aber genauso fällt der Speicherbedarf massiv!

UNIVERSITÄT BAYREUTH

Speicherverfahren III

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
				x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
					x	x	x	x	x	x	x	x	x	x	x	x	x	x
						x	x	x	x	x	x	x	x	x	x	x	x	x
							x	x	x	x	x	x	x	x	x	x	x	x
								x	x	x	x	x	x	x	x	x	x	x
									x	x	x	x	x	x	x	x	x	x
										x	x	x	x	x	x	x	x	x
											x	x	x	x	x	x	x	x
												x	x	x	x	x	x	x
													x	x	x	x	x	x
														x	x	x	x	x
															x	x	x	x
																x	x	x
																	x	x
																		x

1. Symmetrisch

x																	x		
	x	x	x		x									x					
		x									x	x							
			x	x		x	x	x			x							x	
				x	x								x		x			x	
					x		x	x		x	x								
						x	x		x				x	x	x				
							x				x								
								x	x			x							
									x	x		x							
										x						x			
											x		x	x					x
												x	x						
													x						
														x	x				
															x	x	x	x	x
																x			
																	x		x
																		x	
																			x

2. Dünn besetzt

Speicherverfahren IV

1									
	10								
		100							
			1000						
				10000					
					1				
						1000			
							0,1		
								$1 \cdot 10^5$	
									100

3. Schlecht konditioniert, daher immer mit double (C) bzw. DOUBLE PRECISION (FORTRAN) arbeiten. 8 Byte, auf grossen Maschinen auch Quad-Precision mit 16 Byte möglich

H							D												
	H	x	x		x			D											
		H							x										
			H	x		x	x	x		D									
				H	x						D								
					H		x	x		x	x	D							
						H	x		x				x						
							H				x			x					
								H	x			x			D				
									H	x		x				D			
										H						x	D		
											H		x	x				x	
												H	x						D
													H						
														H	x				
															H	x	x	x	x
																H			
																	H		x
																		H	
																			H

I. Bandspeicherung

Speicherverfahren V

$$a_{ij} = 0 \quad \text{für alle } i, j \text{ mit } |i - j| > m$$

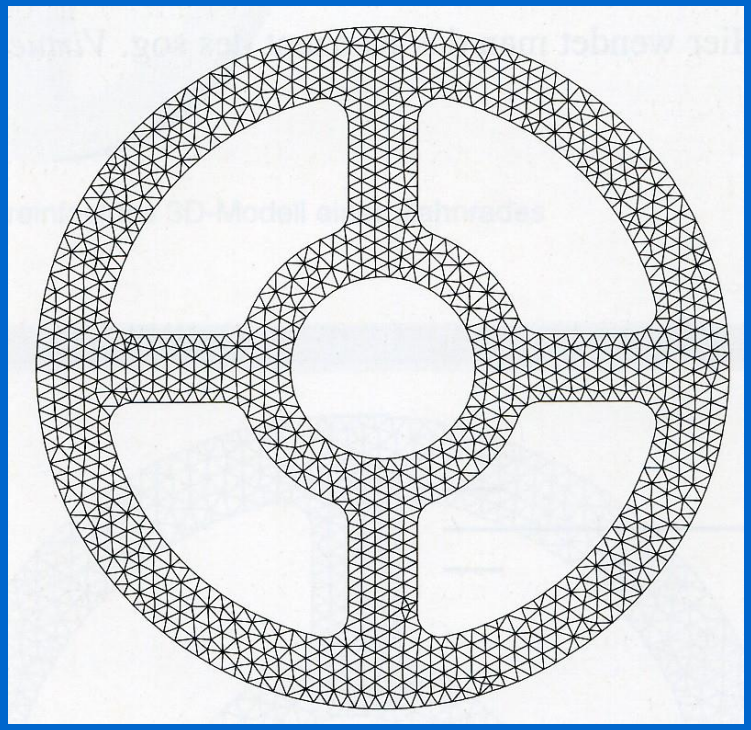
Damit ist die Bandbreite *m* einfach die Anzahl der Nebendiagonalen oberhalb (bei einer unteren Dreiecksmatrix ***L*** dann unterhalb) der Hauptdiagonalen, die Nicht-Null Elemente enthalten. So ist die Bandbreite in der unten gezeigten Matrix dann 7.

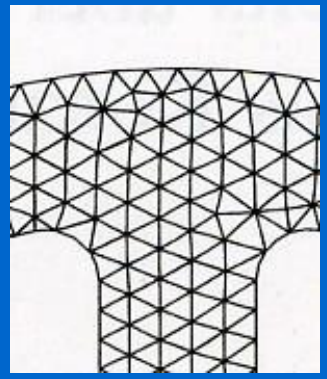
Das Ziel muß offensichtlich sein, die Bandbreite *m* möglichst klein zu halten. Sie wird bestimmt durch die größte Indexdifferenz der Knoten-nummern, die innerhalb der jeweiligen finiten Elemente auftauchen.

Beispiel:
Element Nr. 815: 31 – 34 – 67 – 55 – 89 – 63 – 40 – 48
Element Nr.4711: 10 – 19 – 123 – 56 – 99 – 1010 – 777 – 613
Bei Element Nr.815 wäre die max. Knotenzahldifferenz 89 – 31 = 58
Bei Element Nr.4711 ist die max. Knotenzahldifferenz 1010 – 10 = 1000

Speicherverfahren VI

x				D														x	x
	x	x	x		x													x	x
		x				D													
			x	x		x	x												
				x	x			D											
					x		x	x	D										
						x	x		x	D									
							x				x								
								x	x			x							
									x	x		x	D						
										x				D					
											x		x	x	D				
												x	x			D			
													x				D		
														x	x			D	
															x	x	x	x	x
																x			
																	x		x
																		x	
																			x

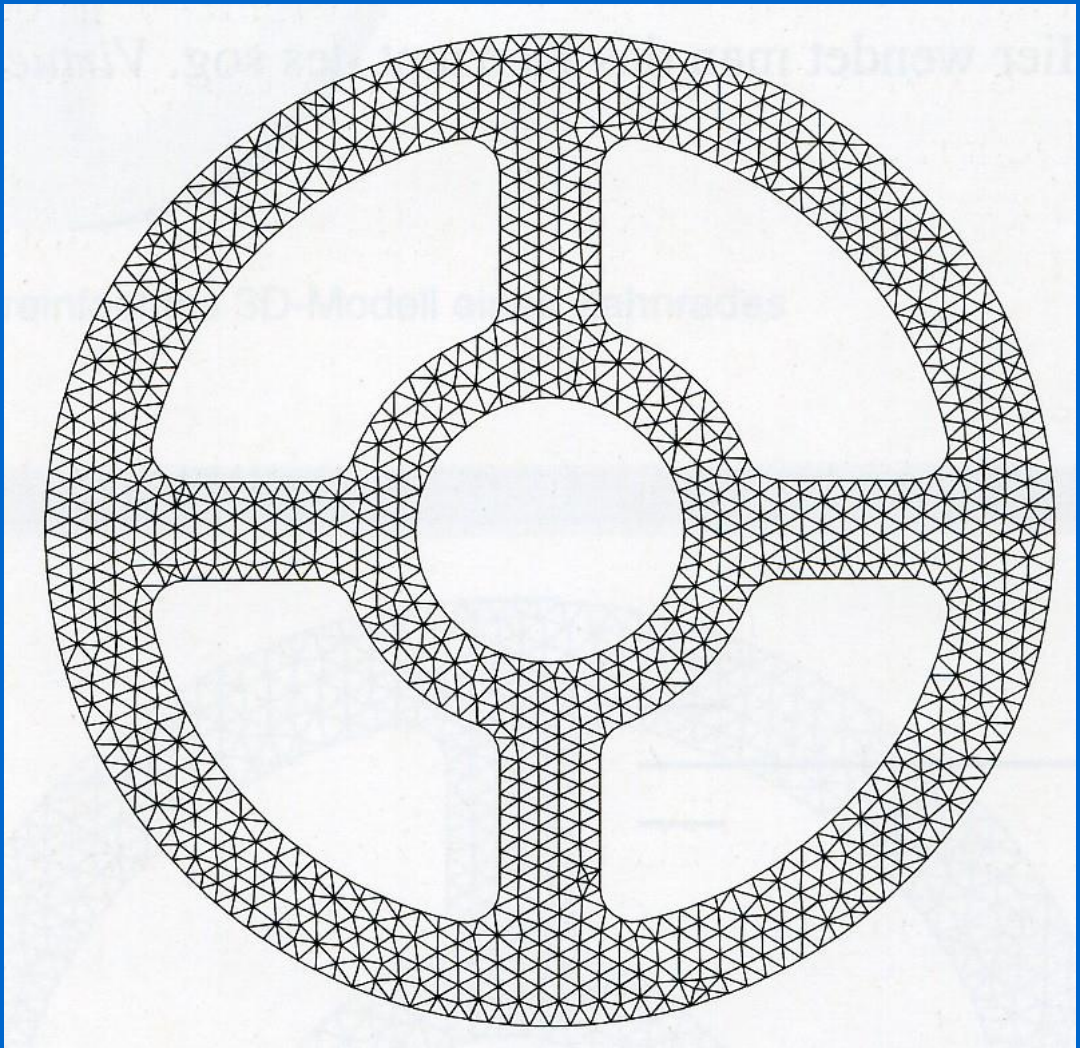




Ringförmige Strukturen: Immer hohe Knotenzahldifferenz!

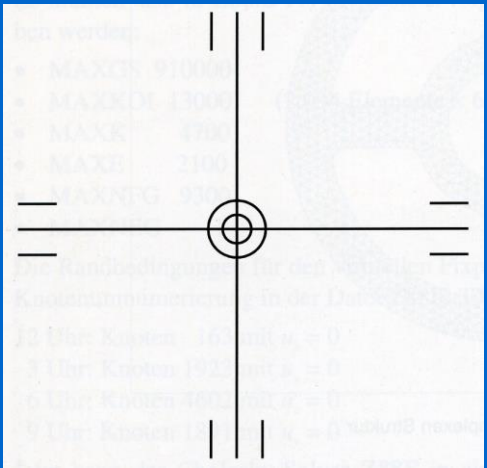
Was kann helfen? Z.B. der *Cuthill-McKee* Algorithmus oder das Verfahren von *Rosen*: Umnumerierung der Knoten. Z88H

Speicherverfahren VII



Beispiel: B141.COS (Pro/MECH)
Z88G
Z88F Testmode (270 MB)

Z88G
Z88H
Z88F Testmode (7 MB, -97%)



Virtueller Fixpunkt

Speicherverfahren VIII

S																	S		
	S	S	S		S												x		
		x	0		0						S						x		
			x	S	0	S	S	S			x						x		
				x	x	0	0	0			0						0		
					x	0	x	x		S	x					S	0		
						x	x	0	S	0	0		S	S		x	0		
							x	0	0	0	x		0	0		0	0		
								x	x	0	0	S	0	0		x	0		
									x	x	0	x	0	0		0	x		
										x	0	0	0	0		x	x		
											x	0	x	x		0	x		S
												x	x	0	S	0	0		x
													x	0	x	0	0		x
														x	x	x	0		0
															x	x	S	S	x
																x	0	0	0
																	x	0	x
																		x	0
																			x

	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	
i=1	1	X			X				
i=2		2	X	X	0				
i=3			4	0	0	X	X		
i=4				6	0	0	0		
i=5					9	X	0	X	
i=6						14	0	0	
i=7							18	0	
i=8								23	27

II. Skyline- Verfahren

ip = {0, 1, 2, 4, 6, 9, 14, 18, 23, 27} Pointer auf Hauptdiagonalelemente

gs = {0, a_{11}, a_{22}, a_{12}, a_{33}, a_{23}, a_{44}, a_{34}, a_{24}, a_{55}, ..., $a_{n,n+1}$} GSM

Speicherverfahren IX

	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	
i=1	1	X			X				
i=2		2	X	X	0				
i=3			4	0	0	X	X		
i=4				6	0	0	0		
i=5					9	X	0	X	
i=6						14	0	0	
i=7							18	0	
i=8								23	27

ip = {0, 1, 2, 4, 6, 9, 14, 18, 23, 27} Pointer auf Hauptdiagonalelemente

gs = {0, a_{11}, a_{22}, a_{12}, a_{33}, a_{23}, a_{44}, a_{34}, a_{24}, a_{55}, ..., $a_{n,n+1}$} GSM

Damit wird: gs_{ij} = gs[ip[j] + j – i]

<u>Beispiele</u>

gs_{15} = gs[9 + 5 – 1] = 13. Vektorelement

gs_{11} = gs[1 + 1 – 1] = 1. Vektorelement

gs_{58} = gs[23+ 8 – 5] = 26. Vektorelement

Speicherverfahren X

Im Programmablauf muß noch getestet werden, ob das angefragte Element überhaupt vorhanden ist:

	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	
i=1	1	X			X				
i=2		2	X	X	0				
i=3			4	0	0	X	X		
i=4				6	0	0	0		
i=5					9	X	0	X	
i=6						14	0	0	
i=7							18	0	
i=8								23	27

```
ianz = ip[j+1] – ip[j]
if(ianz >= j – i + 1)
```
dann Element vorhanden

<u>Beispiel</u>

gs_{58} vorhanden? `ianz = 27 – 23 = 4, 4 >= 8 – 5 + 1,` das stimmt.

Um die ggf. unterschiedlichen Freiheitsgrade am jeweiligen Knoten zu berücksichtigen, wird noch ein Vektor `ioffs` eingeführt:

```
ioffs[1]= 1;

for(i= 2; i <= nkp; i++)
  ioffs[i]= ioffs[i-1] + ifrei[i-1];
```

Speicherverfahren XI

J																			
	J																		
	J	x																	
	J		x																
J	x	0	0	x															
		J	x	0	x														
		J	0	0	x	x													
J	0	0	x	0	0	0	x												
J	x	0	0	0	x	x	x	x											
				J	0	0	0	0	x										
				J	x	x	x	0	x	x									
	J	x	x	x	0	0	0	0	x	x	x								
		J	0	x	x	0	0	x	0	0	0	x							
										J	x	0	x						
			J	x	x	0	0	0	0	0	0	x	0	x					
		J	0	0	0	0	x	x	x	0	0	x	x	x	x				
										J	x	0	0	x	x	x			
									J	0	0	0	x	x	0	0	x		
J	0	0	x	x	x	0	0	0	0	0	0	x	x	0	0	0	x	x	
				J	x	x	x	x	0	x	x	x	0	0	0	x	x	x	x

III: Jennings-Speicherung

1. Schritt:

```
/*-------------------------
* ip auf Startwerte setzen
*-------------------------*/
for(i= 1;i <= (nfg+1);i++)
  ip[i]= i;
```

`ip` ist Zeiger auf die Diagonalelemente

Speicherverfahren XII

2. Schritt:

```
for(k= 1;k <= ne;k++)
  {
/*-----------------------------------------------------------
*  Beispiel: Start  Balkenelement
*-----------------------------------------------------------*/
  else if(ityp[k]== 2)
    {
    mcomp[1]= ioffs[ koi[koffs[k]  ]] -1;
    mcomp[2]= ioffs[ koi[koffs[k]+1]] -1;

    mxknot= 2;
    mxfrei= 6;

    goto L7000;
    }
```

Speicherverfahren 13

```
/*-----------------------------------------------------------
* nun Aufbau der Jennings- Skyline
*-----------------------------------------------------------*/
  L7000:;
  for(j= 1;j <= mxknot;j++) {
    for(j2= 1;j2 <= mxfrei;j2++) {
      for(i= 1;i <= mxknot;i++)       {
        for(i2= 1;i2 <= mxfrei;i2++)      {
          mcompi= mcomp[i]+i2;
          mcompj= mcomp[j]+j2;
          if(mcompj > mcompi) continue;
          if(ip[mcompi] > mcompj)
            ip[mcompi]= mcompj;
          }
        }
      }
    }
  }
```

mxknot	: Anzahl Knoten an dem gerade betrachteten finiten Element
mxfrei	: Anzahl Knoten an dem gerade betrachteten finiten Element
mcompi	: der Index *i* der Matrix
mcompj	: der Index *j* der Matrix
ip	: Pointer auf die Hauptdiagonalelemente

Speicherverfahren 14

3.Schritt

```
/*****************************************************************
* Aufbau von ip
*****************************************************************/
ip[1]= 1;
for(i= 2;i <= nfg; i++)
  ip[i]= ip[i-1] + (i - ip[i]) +1;
```

Speicherverfahren 15

	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	j=9	j=10
i=1	100									
i=2	63	30								
i=3	55	0	600							
i=4	23	0	7	101						
i=5	55	6	0	0	153					
i=6	11	0	0	112	0	217				
i=7			444	0	32	77	262			
i=8			43	13	145	0	0	328		
i=9					99	53	776	0	371	
i=10					97	421	0	89	0	431

Beispiel:

ip = {0, 1, 3, 6, 10, 15, 21, 26, 32, 37, 43}

gs = {0, a_{11}, a_{12}, a_{22}, a_{31}, a_{32}, a_{33}, a_{41}, a_{42}, a_{43}, ..., $a_{n,n}$}

gs = {0,100, 63, 30, 55, 0,600, 23, 0, 7, ..., 431}

gs_{ij} = gs[ip[i] – (i – j)]

Beispiele:

$Gs_{7,4}$ = gs[26 – (7 - 4)] = 23. Vektorelement

$Gs_{4,4}$ = gs[10 – (4 – 4)] = 10. Vektorelement

$Gs_{9,7}$ = gs[37 – (9 – 7)] = 35. Vektorelement

Speicherverfahren 16

Speicherung der Nicht- Nullelemente

Das ist das programmtechnisch mit Abstand schwierigste Verfahren und eignet sich auch nur für den Einsatz mit Iterationssolvern. Dort ist es aber dann auch bestens aufgehoben.

Bei den vorherigen Verfahren ist uns schon aufgefallen, daß zwischen den letzten Skyline- bzw. Jennings-Elementen durchaus sehr viel Nullen auftreten können. Bei manchen Strukturen können ohne weiteres Nullanteil von 99 % auftreten. Das heißt, 99% des kostbaren Hauptspeichers werden regelrecht zugemüllt, und, was mindestens genauso schlimm ist, mit diesen Nullelementen werden die typischen Dreicks-Zerlegungsoperationen stur, wie Computer halt so sind, ausgeführt. Es werden also sehr viele Multiplikationen mit 0 ausgeführt.

Es wäre doch sehr schlau, wirklich nur die Elemente zu speichern, die echt nicht Null sind.

Speicherverfahren 17

Auf einem Papierzettel ist das sehr einfach. Das allerdings im Computer-speicher bei großen Strukturen abzubilden, ist alles andere als einfach. Das eigentlich Schwierige ist nicht unbedingt das Speichern dieser sog. Nicht-Nullelemente, sondern das schnelle Wiederauffinden.

Für einen Überblick empfehle ich *Saad* und den Klassiker von *Young.* Es gibt auch eine Reihe von fertigen FORTRAN- Routinen, z.B. *SPARSEPACK*. Diese bekannten und z.T. auch aus dem Internet downloadbaren Routinen haben aber meist einen enormen Schönheitsfehler: Sie unterstellen – da sie zum Lösen von partiellen Differentialgleichungen und dgl. entwickelt wurden – einen gewissen regelmäßigen Aufbau der Matrix in dem Sinne, daß diese Routinen z.B. dann nur für 20- Knoten Hexaeder und aufsteigend sortierte Koeffizienten gelten. Dahingehend haben wir selbst recht effiziente Versuche unternommen, /Seminararbeit Butscher + Ondrusch, Fak. I in Zusammenarbeit mit LS Rieg, 2000/.

Speicherverfahren 18

Richtig spannend wird es, wenn Sie willkürlich und unsystematisch numerierte Netze, vielleicht noch obendrein mit verschiedenen Elementtypen gemischt, vorliegen haben – also den Normalfall einer typischen Finite Elemente Analyse des Maschinenbau- Ingenieurs, des Bauingenieurs, des Fahrzeugtechnikers.
Um diese kleinen Unstimmigkeiten zu überwinden, habe ich zusammen mit Dr. Frank Koch ein universelles Verfahren zum Abspeichern von typischen und komplizierten FE-Netzen entwickelt, das im späteren Zugriff auf die Matrixelemente recht flott ist:

Wir gehen am besten in zwei Schritten vor: Es werden zunächst folgende Pointer für die untere Hälfte der Gesamt- Steifigkeitsmatrix GS aufgebaut:

- Pointervektor `IP` zeigt auf die Diagonalelemente `GS(i,i)`
- Pointervektor `IEZ` zeigt auf die Spaltenindices `GS(x,j)`

Dazu wird zunächst eine Structure `IJ` aufgebaut, die dann mit einem QSORT-Algorithmus sortiert wird.

Speicherverfahren 19

Beispiel:
GSM `GS`

GS(1,1)					
GS(2,1)	GS(2,2)				
	GS(3,2)	GS(3,3)			
GS(4,1)			GS(4,4)		
GS(5,1)		GS(5,3)		GS(5,5)	
	GS(6,2)		GS(6,4)		GS(6,6)

als Vektor
`GS`:

GS(1,1)	GS(2,1)	GS(2,2)	GS(3,2)	GS(3,3)	GS(4,1)	GS(4,4)
GS(5,1)	GS(5,3)	GS(5,5)	GS(6,2)	GS(6,4)	GS(6,6)	

Pointervektor `IEZ` zeigt auf die Spaltenindices `GS(x,j)`

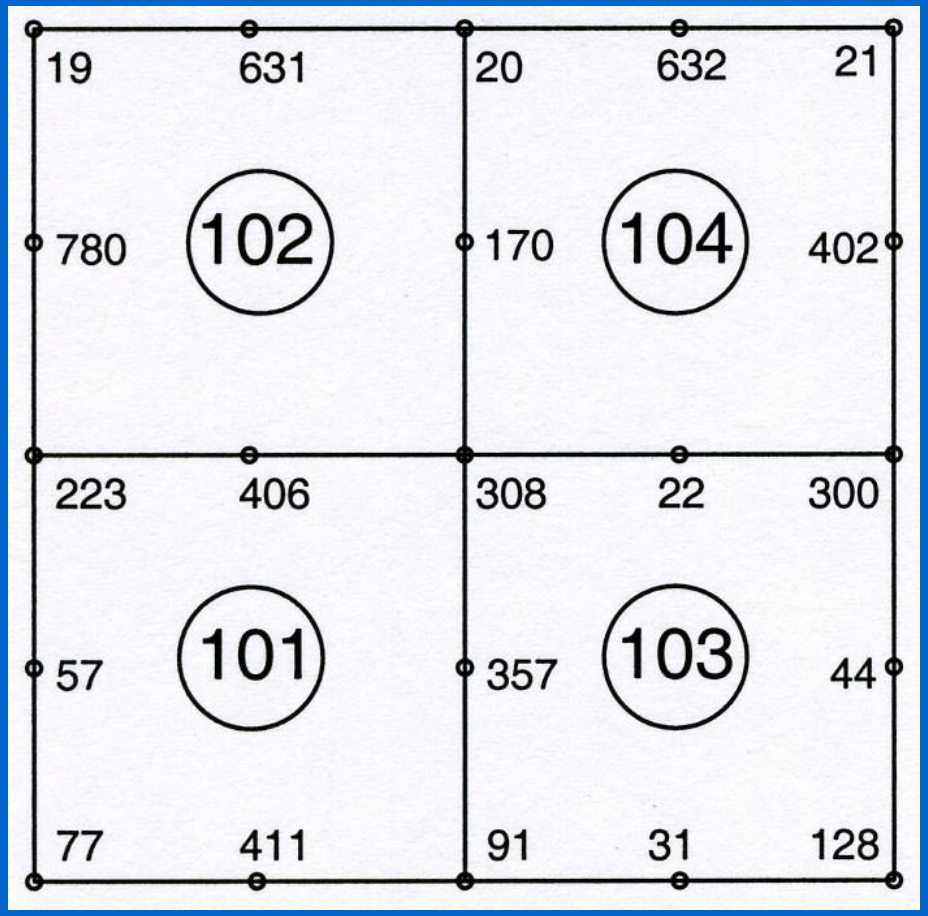
1	1	2	2	3	1	4	1	3	5	2	4	6

Pointervektor `IP` zeigt auf die Diagonal-elemente `GS(i,i)`

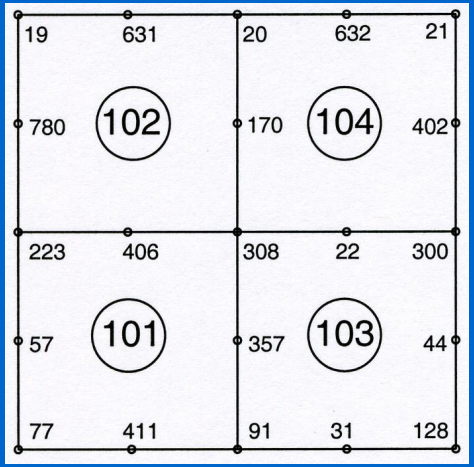
1	3	5	7	10	13

Speicherverfahren 20

Das sieht auf dem Papier ganz einfach und logisch aus. Aber der Weg dahin ist schwierig: Bedingt durch die Tatsache, daß bei realen, großen FE-Netzen aus dem Ingenieurbereich die Elemente, die Knoten und die Koinzidenzlisten *eben nicht* schön sauber aufsteigend angeordnet sind. Angenommen, wir hätten eine Scheibenberechnung mit quadratischen Serendipity-Scheiben mit je 8 Knoten (z.B. Typ 7 von Z88) auszuführen, und folgende vier Elemente mit ihren zugehörigen Knotennummern seien mitten aus einer größeren Struktur herausgegriffen:



Speicherverfahren 21



Knoten	Nr.1	Nr.2	Nr.3	Nr.4	Nr.5	Nr.5	Nr.7	Nr.8
...	...	...	...	...	...	...	...	...
Element 101	77	91	308	223	411	357	406	57
Element 102	223	308	20	19	406	170	631	780
Element 103	91	128	300	308	31	44	22	357
Element 104	308	300	21	20	22	402	632	170
...	...	...	...	...	...	...	...	...

Jetzt müßten wir das noch in je 2 Freiheitsgrade pro Knoten unterteilen, aber um zu zeigen, auf was es ankommt, reicht es, die Betrachtung nur mit den Knotennummern auszuführen: Es werden nach aufsteigenden Elementnummern sortiert die Knotennummern in den Vektor GS gegeben, der wieder bei 0 startet:

```
gs= {0, 77,  91, 308, 223, 411, 357, 406,  57,
     223, 308,  20,  19, 406, 170, 631, 780,
      91, 128, 300, 308,  31,  44,  22, 357,
     308, 300,  21,  20,  22, 402, 632, 170}
```

Speicherverfahren 22

Jetzt werden wir diesen Vektor sauber nach aufsteigenden Knotennummern sortieren mit einem *QSORT*-Algorithmus („Standard“-Function in eigentlichen C-Compilern):

```
gs= {0, 19,  20,  20,  21,  22,  22,  31,  44,
     57,  77,  91,  91, 128, 170, 170, 223,
    223, 300, 300, 308, 308, 308, 308, 357,
    357, 402, 406, 406, 411, 631, 632, 780}
```

Der Vektor ist zwar nunmehr schön sortiert, aber verschienene Nummern (= in der Realität die Indizes *i* und *j*) tauchen doppelt und die Nummer 308 sogar vierfach auf. Diese mehrfachen Nummern müssen noch ausgefiltert werden.

Speicherverfahren 23

	Speicher-anforderung	Empfindlichkeit auf Knotennumerierung	Gleichungs-löser	Programmier-aufwand
Band-speicherung	meist groß	hoch	alle	minimal
Skyline-Speicherung	besser als Band	mittel	alle	mittel
Jennings-Speicherung	Wie Skyline	mittel	alle	höher als Skyline
Nicht-Null-elemente	minimal	fast unempfindlich	Nur Iterations-Solver	sehr hoch

Zusammenfassung der Ergebnisse