Hash Based Data Structures for Tetrahedral Meshes

Christoph Bosshard

Supervisor: Prof. Dr. R. Hiptmair (SAM, D-MATH)

8. Mai 2005

Inhaltsverzeichnis

Zusammenfassung

Field. Tetrahedral meshes, mesh refinement, data structures, hash tables, software devel-opment

Problem . Finite element schemes for boundary value problems on three-dimensional computational domains may place degrees of freedom (d.o.f.) on vertices, edges, faces, and/or cells of a tetrahedral mesh. However, it would be wasteful to store edges and faces of a mesh, if these are not needed. Thus, it becomes desirable to have a data structure that allows the easy allocation of edges and faces whenever required.

Hash based data structure . The idea is to store only the vertices of the mesh along with the incidence information of vertices and cells. Edges and faces are index by the ordered tupel of their vertices, which is also used as hash key. In the beginning the two hash tables representing edges and vertices are empty. If d.o.f. on edges and vertices are introduced, these hash tables are filled.

Implementation . The implementation should be done in C++ extensively using the Standard Template Library and its extensions providing hash tables.

Issues . During the term project different hash strategies (hash functions) should be im-plemented and their performances should be compared. A detailed profiling of the code should be carried out. The meshes should be imported from a mesh generator (NetGen developed by J.Schöberl at Uni Linz)

Teil I
Der Bericht zur Semesterarbeit

1 Grundlagen

Die Idee des Hashens ist, aus dem Schlüssel eines Datensatzes die dazugehörige Adresse zu berechnen. Dies erlaubt den Zugriff auf Elemente in O(1) unabhängig von der Anzahl der Elemente. Typischerweise bildet die Hash-Funktion die Schlüssel auf einen Index $\in \mathbb{N}$ ab.
Wenn zu den Schlüsseln von zwei verschiedenen Elementen die Hashfunktion den gleichen Wert annimmt nennt man dies eine Kollision. Eine gute hash-Funktion hat folgende Eigenschaften

- Werte können mit geringem Aufwand berechnet werden.
- Kollisionen sollten minimiert werden

Im Details gibt es viele verschiedene Varianten von Hash-Tabellen. Dabei kann man nach der Art wie Kollisionen gelöst werden, zwei Gruppen unterscheiden:

- chaining
- open adressing

open adressing habe ich für diese Arbeit nicht verwendet, und werde nicht weiter darauf eingehen. Beim chaining , welches für diese Arbeit verwendet wurde, werden alle Elemente, die an den selben Ort hashen in eine verlinkte Liste gespeichert.

2 Implementation der Datenstrukturen

2.1 Zellen und Knoten

$\longrightarrow$ Siehe auch die von Doxygen erzeugte Dokumentation

Ein Gitter Besteht aus Zellen. In meinem Fall sind die Zellen Tetraeder. Ein Tetraeder besteht aus vier Eckpunkten, die Gitterknoten. Ein Eckpunkt wird durch seine drei kartesischen Koordinaten beschrieben.

```
struct node{
//int ID; //globale Nummer des Punktes
// x- Koordinate
float x; //x-Koordinate
// y-Koordinate
float y; //y-Koordinate
// z- Koordinate
float z; // z-Koordinate
};
```

Abbildung 1: struct für die Knoten

```
struct Cell{
// Index Tetraederecke in Knotenliste Nodes
int a;
// Index  Tetraederecke in Knotenliste Nodes
int b;
// Index  Tetraederecke in Knotenliste Nodes
int c;
// Index  Tetraederecke in Knotenliste Nodes
int d;
//a,b,c,d sind Indizes der vier Tetraederpunkte
};
```

Abbildung 2: struct für eine Zelle

Für die Eckpunkte habe ich das struct node verwendet. Alle Eckpunkte speichere ich in der Hash-Map Nodes. Jedem Punkt ist eine Nummer zugeordnet (die globale Nummer). Diese Nummer verwende ich als Schlüssel für die Hash-Map.
Für die Zellen verwende ich das Strukt Cell. Eine Zelle ist durch die Nummern seiner Eckpunkte bestimmt. Die Zellen speichere ich in der Liste cells.

2.2 Knoten-, Flächen- und Kantenfreiheitsgrade

Freiheitsgrade können sich auf Ecken, Kanten und Flächen befinden. Dafür habe ich die drei Hash-Maps V_Dofs, E_Dofs, F_Dofs verwendet. Näheres zu ihrer Funktionsweise im nächsten Abschnitt.

```
list<Cell> cells;
```

Abbildung 3: Die Liste mit den Zellen des Gitters

```
hash_map<const int, node, V_hash_fcn, V_eqstr> Nodes;
```

Abbildung 4: Hashtable mit den Knoten

```
// Hash - Tabelle mit den Knotenfreiheitsgraden
hash_map<const int, float,V_hash_fcn, V_eqstr> V_Dofs;
// Hash - Tabelle mit den Kantenfreiheitsgraden
hash_map<const Edges, float,E_hash_fcn, E_eqstr> E_Dofs;
// Hash - Tabelle mit den Flächenfreiheitsgraden
hash_map<const Faces, float,F_hash_fcn, F_eqstr> F_Dofs;
```

3 Die Hash-Map von SGI

Für die Implementierung benutzte ich die Klasse hash_map aus der sgi-Erweiterung der Standartbibliothek. Kollisionen werden dabei mittels chaining gelöst. Leider habe ich keine Dokumentation gefunden, in der die Implementierung dieser hash_map von sgi genauer beschrieben wird und ich habe mir leider nicht die Mühe genommen, all die Details mit Hilfe der source davon herauszufinden.Die Klasse hash_map von sgi hat folgende Form:

hash_map<Key, Data, HashFcn, EqualKey, Alloc>

Im folgenden werde ich die Parameter beschreiben

3.1 Key -Der Schlüssel

Bei der Knotenhashtabelle und der Eckenfreiheitsgradehashtabelle habe ich den globalen Knotenindex als Schlüssel gewählt .
Bei der Eckenfreiheitsgradehashtabelle habe ich die der Grösse nach geordnete Indizes der Eckpunkte der Ecken genommen (3,5) und (5,3) beschreiben die gleiche Kannte, → Schlüssel (3,5).
Für die Flächenfreiheitsgrade habe ich analog zu den Eckenfreiheitsgraden das geordnete Tripel der Eckpunkte genommen (2,4,7) = (4,7,2) = (7,2,4) = (2,7,4) = (4,2,7) = (7,4,2) hat den Schlüssel (2,4,7).

3.2 Data

Die zu den oberen Hashtabellen gehörenden Daten sind:

- Knoten des Gitters.
- Werte der Knotenfreiheitsgrade
- Werte der Kantenfreiheitsgrade
- Werte der Flächenfreiheitsgrade

3.3 HashFcn

Eigentlich ist das ein Funktionsobjekt. Näheres zu den vom mir verwendeten Hashfunktionen in section 8

Der Operator () dieses Funktionsobjektes darf nur ein Argument annehmen. Da aber für viele interessante Hashfunktionen aber die Anzahl Knoten des Gitters, sowie noch weitere Werte benötigt werden musste ich diese mit Hilfe von globalen Hilfsvariablen übergeben.

3.4 EqualKey

Diese Funktion (eigentlich ist es als Funktionsobjekt implementiert) bestimmt, wann zwei Schlüssel als gleich angesehen werden. Ich habe dies so gewählt, dass zwei Schlüssel gleich sind, wenn alle paarweise Indizes übereinstimmen

3.5 alloc und Speicherplatzverwaltung

Ich habe hier den default Allokator stehen gelassen. In der Dokumentation der sgi -STL steht dazu:

> The details of the allocator interface are still subject to change, and we do not guarantee that specific member functions will remain in future versions. You should think of an allocator as a black box. That is, you may select a container's memory allocation strategy by instantiating the container template with a particular allocator [2], but you should not make any assumptions about how the container actually uses the allocator.

und

> alloc The default allocator. It is thread-safe, and usually has the best performance characteristics.

[sgi] Der Allokator bestimmt NICHT, wann die Hashtabelle wie gross ist.

Speicherplatzverwaltung der Hashtabelle Die Hash-Tabelle besteht aus sogenannten Buckets. In einem Bucket befinden sich alle Elemente die zur selben Adresse gehasht wurden.
Die Anzahl Buckets wird mit Hilfe des folgenden structs bestimmt:

```
static const unsigned long __stl_prime_list[__stl_num_primes] =
{
  53ul,         97ul,         193ul,       389ul,       769ul,
  1543ul,       3079ul,       6151ul,      12289ul,     24593ul,
  49157ul,      98317ul,      196613ul,    393241ul,    786433ul,
  1572869ul,    3145739ul,    6291469ul,   12582917ul,  25165843ul,
  50331653ul,   100663319ul,  201326611ul, 402653189ul, 805306457ul,
  1610612741ul, 3221225473ul, 4294967291ul
};
```

Wenn also z.B. das 193te Element eingefügt wird, (194 in erster Zeile, 3. Spalte) wird die Anzahl buckets auf 389 (erste Zeile, 4. Spalte), also auf die nächsthöhere Zahl im Strukt oben erhöht.
Interessanterweise hatte ich immer eine Mindestbucketzahl von 193 auch wenn noch gar keine Elemente eingefügt worden waren.

4 Zeitmessung

Zur Messung der cpu Zeit habe ich die Funktion clock aus time.h von der Standartbibliothek genommen. Dazu ein Ausschnitt aus der Dokumentation von libc, der GNU C-Bibliothek [cli].

```
Processor And CPU Time
======================

If you're trying to optimize your program or measure its efficiency,
it's very useful to know how much processor time it uses.  For that,
calendar time and elapsed times are useless because a process may spend
time waiting for I/O or for other processes to use the CPU.  However,
you can get the information with the functions in this section.

   CPU time (*note Time Basics::) is represented by the data type
'clock_t', which is a number of "clock ticks".  It gives the total
amount of time a process has actively used a CPU since some arbitrary
event.  On the GNU system, that event is the creation of the process.
While arbitrary in general, the event is always the same event for any
particular process, so you can always measure how much time on the CPU
a particular computation takes by examinining the process' CPU time
before and after the computation.

   In the GNU system, 'clock_t' is equivalent to 'long int' and
'CLOCKS_PER_SEC' is an integer value.  But in other systems, both
'clock_t' and the macro 'CLOCKS_PER_SEC' can be either integer or
floating-point types.  Casting CPU time values to 'double', as in the
example above, makes sure that operations such as arithmetic and
printing work properly and consistently no matter what the underlying
representation is.

   Note that the clock can wrap around.  On a 32bit system with
'CLOCKS_PER_SEC' set to one million this function will return the same
value approximately every 72 minutes.

   For additional functions to examine a process' use of processor time,
and to control it, *Note Resource Usage And Limitation::.

CPU Time Inquiry
----------------

To get a process' CPU time, you can use the 'clock' function.  This
facility is declared in the header file 'time.h'.

   In typical usage, you call the 'clock' function at the beginning and
end of the interval you want to time, subtract the values, and then
divide by 'CLOCKS_PER_SEC' (the number of clock ticks per second) to
get processor time, like this:

     #include <time.h>
```

```
clock_t start, end;
double cpu_time_used;

start = clock();
... /* Do the work. */
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Teil II
Messung

5 Allgemeines

Die Gitter sind aus Beispielegeometrien, die dem Programm Netgen [net] hinzugefügt waren. Ich habe immer weiter verfeinert, bis dass die Anzahl Elemente grösser als 100'000 war. Beim Versuch noch weiter zu verfeinern stiess ich auf Probleme: auf den Windows-Computern der ETHZ .
Dabei habe ich folgende Meldung erhalten:
indows-Virtual-Memory Minimum too low. Your system is low on virtual memory. Windows is increasing the size of your virtual memory paging file. During this proces, memory requests for some applications may be denied. For more information see help.
Ich hätte allerdings durchaus noch Möglichkeiten gehabt, um zu versuchen grössere Gitter herzustellen:

- Die Windows -Version von Netgen habe ich als binary erhalten. Ich könnte den Netgen mal auf Unix selber compilieren und dann die Gitter via einer Programmbibliothek erzeugen.
- Netgen hat noch verschiedene Optionen für die Gittererzeugeung.
- Es gibt auch noch andere Programme, wie z.B. Tetgen [tet]

6 Der verwendete Computer

Ich habe alle Messungen auf dem slab47.ethz.ch durchgeführt. Hier ist seine cpuinfo-Date (proc/cpuinfo)

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 3
model name      : Intel(R) Pentium(R) 4 CPU 2.80GHz
stepping        : 4
cpu MHz         : 2793.133
cache size      : 1024 KB
fdiv_bug        : no
```

```
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
```

7 Die einzelnen Gitter

7.1 boxcyl.vol

- Dimension: 3
- Points 74793
- Elements 403456
- Surface Els 29440
- Segments 1536

7.2 cone.vol

- Dimension: 3
- Points 32497
- Elements 172032
- Surface Els 14848
- Segments 464

7.3 cube.vol

- Dimension: 3
- Points 68705
- Elements 393216
- Surface Els 12288
- Segments 768

7.4 fichera.vol

- Dimension: 3
- Points 19057
- Elements 102400
- Surface Els 7680
- Segments 736

7.5 lshape3d.vol

- Dimension: 3
- Points 138401
- Elements 786432
- Surface Els 28672
- Segments 1408

7.6 shaft.vol

- Dimension: 3
- Points 34833
- Elements 165248
- Surface Els 27968
- Segments 2840

7.7 sphere.vol

- Dimension: 3
- Points 89793
- Elements 491520
- Surface Els 30720
- Segments 32

7.8 trafo.vol

- Dimension: 3
- Points 54265
- Elements 317504
- Surface Els 26784
- Segments 2776

7.9 twocubes.vol

- Dimension: 3
- Points 54265
- Elements 317504
- Surface Els 26784
- Segments 2776

```
struct Edges{
// Punkt einer Kante mit kleinerem Index
// in Knotenliste
int a; // zwei Punkte bestimmen eine Kante
// Punkt einer Kante mit grösserem Index
// in Knotenliste
int b;
};
```

Abbildung 5: Kantenfreiheitsgrad

```
struct Faces{
// Punkt mit kleinstem Index der Fläche
int a; //3 Punkte bestimmen eine Tetraederfläche
// Punkt mit mittlern Index der Fläche
int b;
// Punkt mit grösstem Index der Fläche
int c;
};
```

7.10 twocyl.vol

- Dimension: 3
- Points 55953
- Elements 294400
- Surface Els 26752
- Segments 1152

8 Die Hash-Funktion

8.1 Allgemeines

Falls die Hashfunktionen Werte grösser als die Anzahl Buckets annimmt, wird automatisch modulo die Anzahl buckets gerechnet

8.2 Die Hashfunktionen, welche ich ausprobiert habe

8.2.1 Allgemeine Bemerkungen

Für die Knotenfreiheitsgrade habe ich die Identität verwendet (das ist trivial). Für die Kantenfreiheitsgrade habe ich das struct Edges verwendet.

```
struct E_hash_fcn {
///Hashfunktion
int operator()(const Edges s ) const {
int a = 0;
a = s.a * s.b * s.c;
//Beispiel für eine Hashfunktion
return a;
  }

};
```

8.2.2 Hashfunktionen für die Kantenfreiheitsgrade

Seien s.a, s.b die zwei Komponenten einer Kante, der ein Kantenfreiheitsgrad in der Kantenfreiheitsgrad-Hashtabelle zugeordnet werden soll. Der Operator des Hash-Funktionsobjekts soll also eine Abbildung

$$(s.a + s.b) \mapsto [0, b]$$

ausführen, wobei b die Anzahl Buckets der Kanten-Hashtabelle ist. Bei der folgenden Beschreibung werde ich die beiden Komponenten der Kanten immer $s.a$ und $s.b$ nennen (Es gilt $s.a < s.b$).
Weiter sei
n_v: Anzahl der Knotenfreiheitsgrade
n_e: Anzahl der Kantenfreiheitsgrade.

Auswahl der Hashfunktionen Die Auswahl meiner Hashfunktionen ist eher zufällig. Im Paper [mau] und in [lum] habe ich Ideen für Hashfunktionen gefunden.
Leider wird von der SGI hash_map das b vorgegeben (vgl. Speicherplatzverwaltung in Abschnitt 3.5). Das hat zur Folge, dass einige geeignete Hashfunktionen nicht optimal funktionieren können. Eine solche Methode ist zum Beispiel einfach soviele Bits auf eine festgelegte Weise aus $s.a$ und $s.b$ herauszunehmen und damit die Adresse der Hashfunktion zu bilden. Diese Methode ergibt aber Adressen mit Länge einer Potenz von 2 (Die Adresslängen in der SGI-Hash_map scheinen so gewählt worden zu sein, dass sie nicht in der Nähe einer Potenz von 2 sind!).
Es folgen nun die Beschreibung der Hashfunktionen, die ich verwendet habe.
Ich könnte wetten, dass ich noch irgendeine gute Art von Hashfunktionen nicht gesehen habe und optimiert sind meine auf keinen Fall.

A Multiplikation aller Komponenten

$$(s.a, s.b) \mapsto s.a * s.b; \tag{1}$$

B Addition aller Komponenten

$$(s.a, s.b) \mapsto s.a + s.b \tag{2}$$

Diese Hashfunktion hat einen Nachteil. Bei den Gittern die ich verwendet habe gibt es etwa 6.5 mal mehr Kantenfreiheitsgrade als Gitterknoten. Der höchste Wert den diese Hashfunktion aber annehmen kann ist $n_v + n_v = 2n_v \approx 2 * (n_e/6.5) \approx 1/3n_e$ Die Hashfunktion nützt also nur einen Teil der ganzen Hashtabelle aus.

C Eine Art Divisionsmethode

$$(s.a, s.b) \mapsto s.a * n_v + s.b; \tag{3}$$

Eine Art von Hashfunktion ist es eine positive ganzzahlige Adresse modulo eine bestimmte ganze Zahl zu teilen. Da bei der SGI hash_map die Anzahl buckets vorgegeben ist siehe 3.5 geschieht das bei allen meinen Hashfunktionen. Mit der Hashfunktion 3 wollte ich nun die Kanten der Form (s.a, s.b) auf die natürlichen Zahlen abbilden und dabei sicher keine Kollisionen zu erzeugen, aber auch keine unnötigen Lücken.

D: Multiplikationsmethode aus [cor]

$$(s.a, s.b) \mapsto \lfloor 2^{24} * ((s.a * n_v + s.b)\frac{\sqrt{5}-1}{2} \; mod \; 1)\rfloor \tag{4}$$

Wobei *mod* modulo bedeutet und $\lfloor \ldots \rfloor$ die nächst untere ganze Zahl. Diese Funktion hat den grossen Nachteil, dass sie zu teuer zum auswerten ist (Falls die Zugriffszeiten auf die Hashtabelle gross sind, hätte sie ihre Berechtigung).

E: Methode mit XOR

$$(s.a, s.b) \mapsto s.a \wedge (s.b * 314); \tag{5}$$

Das $\wedge$ -Zeichen steht für die bitweise XOR Verknüpfung. Meine Idee war folgende: seien u und v zwei „zufällige Zahlen“. Da sie zufällig sind, sollten auch an jeder Stelle die Bitwerte 0 und 1 je mit Wahrscheinlichkeiten 0.5 auftreten. Für die bitweise XOR-Verknüpfung gelten folgende Gleichungen:

$$0 \wedge 0 = 0$$
$$0 \wedge 1 = 1$$
$$1 \wedge 0 = 1$$
$$1 \wedge 1 = 0$$

Damit sollte dann auch $u+v$ eine "ziemlich zufällige" Zahl sein. Ausserdem hat die XOR-Verknüpfung den Vorteil, dass sie schnell auszuwerten ist. Mit diesen Hintergedanken habe ich dann einfach mal irgendeine solche Funktion ausprobiert und sie funktionierte ausgezeichnet.
Das müsste ich allerdings noch etwas genauer untersuchen.

F:Methode fast ohne Addition und Multiplikation In [mau] habe ich folgendes gefunden:

> Folding and its generalizations. A very fast method of obtaining a k-bit hash code from an n-bit key, for $k < n$, is by picking out several k-bit fields from the n-bit key and adding them up, or, alternatively, taking the excIusive OR.

[...] Dann habe ich etwas ähnliches für mein Beispiel konstruiert.

$$(s.a, s.b) \mapsto ((s.a) << 7) \wedge (s.b); \tag{6}$$

Bei dieser Methode gibt es wegen der Implementation der SGI hash_map doch noch eine Modulo-Rechnung durch die Anzahl vorhandener Buckets obwohl das eigentlich nicht nötig wäre.
$<<$ und $>>$ sind die shift-Operatoren.
$<< n$ bedeutet, dass alle Bits (der 32 Bit integer Zahl) um n Stellen nach links verschoben werden und von rechts mit Nullen aufgefüllt. Die n bits, die am weitesten links waren verschwinden.
$>> n$ bedeutet, dass alle Bits (der 32 Bit integer Zahl) um n Stellen nach rechts verschoben werden und von links mit 0 aufgefüllt. Die n Bits, die ganz rechts waren verschwinden.
Beispiel:
$37 >> 3 = 4$, Denn 32 in Bitschreibweise $32 \equiv 11001$
$100101 >> 3 = 100$ in Bits $= 4$ im Dezimalsystem

G: modifizierte Version von A

$$(s.a, s.b) \mapsto s.a * 100 + s.b \tag{7}$$

H: wie in F mit plus statt XOR

$$(s.a, s.b) \mapsto ((s.a) << 7) + (s.b); \tag{8}$$

8.2.3 Hashfunktionen für die Flächenfreiheitsgrade

Hier habe ich ähnliche Funktionen wie bei den Kantenflächenfreiheitsgraden gewählt, obwohl das natürlich nicht so sein muss!

A: Multiplikation aller Komponenten

$$(s.a, s.b, s.c) \mapsto s.a + s.b + s.c; \tag{9}$$

B: Addition aller Komponenten

$$(s.a, s.b, s.c) \mapsto s.a * s.b * s.c \tag{10}$$

Hier gilt analoges wie das , was ich für die Kantenhashfunktion B gesagt habe.

C: Overflow-Methode ...

$$(s.a, s.b, s.c) \mapsto s.a * n_v * n_v + s.b * n_v + s.c; \tag{11}$$

Die Ideen für diese Hashfunktion sind die gleichen, wie bei der Hashfunktion C für Kantenfreiheitsgrade. Sie funktioniert ziemlich gut.
Was ich erst viel später gemerkt habe: **Der erste Term wird so gross dass es einen Overflow gibt**. Das Programm ist damit ziemlich unberechenbar, liefert aber dennoch gute Hashfunktionen.Angesichts der guten Ergebnisse wäre es vielleicht interessant, mal zu schauen was der Computer eigentlich macht (ich habe gesehen, dass etwa die Hälfte der Werte negativ werden ...) und damit eine weitere Hashfunktion (ohne Overflow!) zu konstruieren.

D: Multiplikationsmethode aus [cor]

$$(s.a, s.b) \mapsto \lfloor 2^{24} * ((s.a * n_v * n_v + s.b * n_v + s.c)\frac{\sqrt{5}-1}{2} \; mod \; 1)\rfloor \tag{12}$$

Wobei *mod* modulo bedeutet und $\lfloor \ldots \rfloor$ die nächst untere ganze Zahl. Das ist die Overflow-Methode aus C kombiniert mit der Multiplikationsmethode aus [cor]. Die Zeiten sind erwartungsgemäss höher als bei den anderen Methoden.

E: Methode mit XOR

$$(s.a, s.b, s.c) \mapsto s.a * 17 \wedge s.b \wedge s.c * 31;$$

Alles was ich für die Hashfunktion E für Kantenfreiheitsgrade geschrieben habe gilt auch hier.

F:Methode fast ohne Addition und Multiplikation

$$(s.a, s.b, s.c) \mapsto (s.a << 7) \wedge ((s.b << 3) \wedge s.c)$$

Alles was ich für die Hashfunktion E für Kantenfreiheitsgrade geschrieben habe gilt auch hier.

G:modifizierte Version von A

$$(s.a, s.b, s.c) \mapsto s.a * 100 + 10 * s.b + s.c$$

H:wie in F mit plus statt XOR

$$(s.a, s.b, s.c) \mapsto (s.a << 7) + ((s.b << 3) + s.c)$$

9 Ergebnisse

9.1 Was ich gemessen habe

9.1.1 Zeitmessung und Anzahl Vergleiche

Zur Zeitmessung habe ich folgende Aktionen verwendet:

Kantenfreiheitsgrade Ich durchlaufe die Zellliste und extrahiere von jeder Zellen alle Kanten. Für allen Kanten probiere ich eine 1 einzufügen. Falls das nicht klappt, weil ich den entsprechenden Freiheitsgrad schon beim durchlaufen einer früheren Zelle eingefügt habe, addiere ich eins zum Freiheitsgrad dazu.
Für diese Aktion habe ich die benötigte Zeit, sowie die Anzahl Aufrufe der Gleichheitsfunktion der Kantenhashtabelle gemessen.

Flächenfreiheitsgrade Gleich, wie bei den Kantenfreiheitsgraden, nur dass jetzt anstelle der Kanten Flächen sind, und anstelle der Kantenfreiheitsgrad-Hashtabelle die Flächenfreiheitsgrad- Hashtabelle.

Mittlere Kettenlänge Seien n_0 Buckets mit null Elementen gefüllt, n_1 Buckets mit einem Element gefüllt, n_2 Buckets mit zwei Elementen , n_3 mit drei Elementen, ... dann ist die mittlere Kettenlänge definiert als

$$\text{Mittlere Kettenlänge} = \frac{n_1 * 1 + n_2 * 2 + n_3 * 3 + \ldots}{n_1 + n_2 + n_3 + \ldots}$$

Die mittlere Kettenlänge ist ein Mass für die Qualität einer Hashfunktion, dass nicht zu stark von der Gittergrösse abzuhängen scheint.

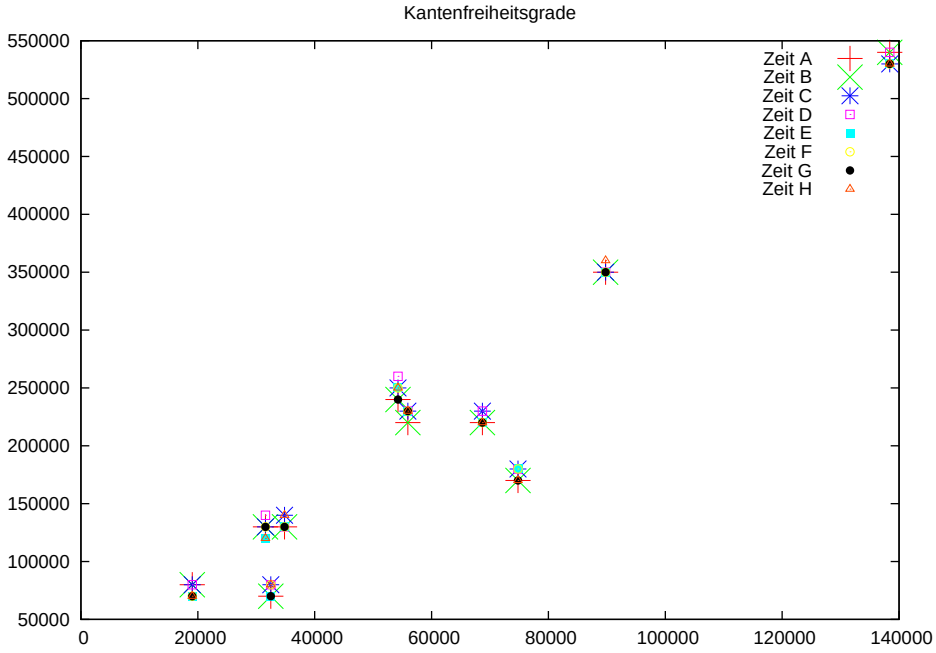
Der Ladefaktor Kantenfreiheitsgrade: Sei n_e die Anzahl Elemente in der Hashtabelle und e_bucket_count die Anzahl Buckets. Dann ist der Ladefaktor definiert als

$$\text{Ladefaktor} = \frac{n_e}{e\_bucket\_count}$$

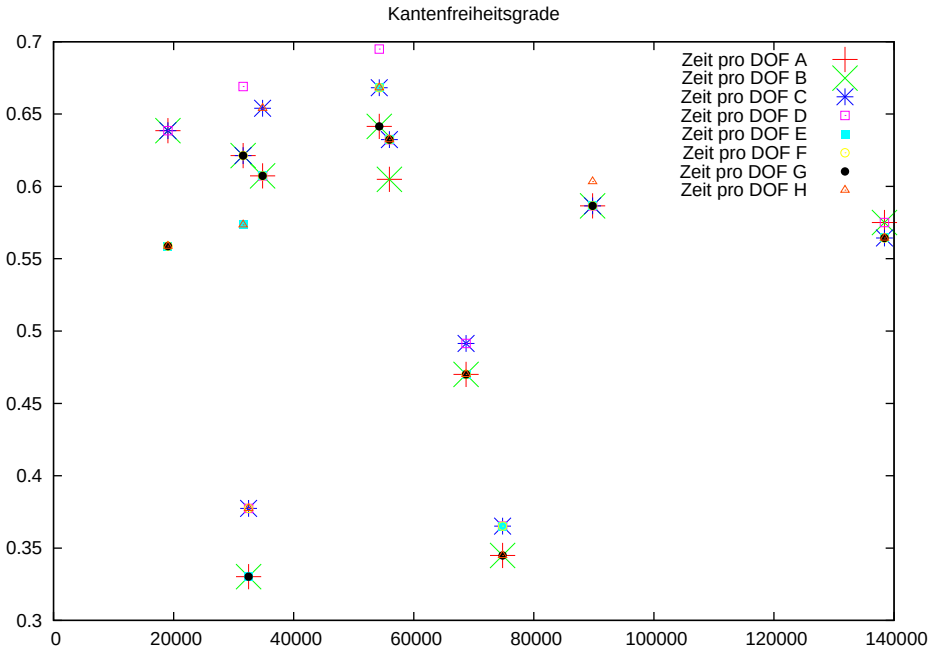
Je grösser der Ladefaktor ist, desto mehr Kollisionen gibt es.

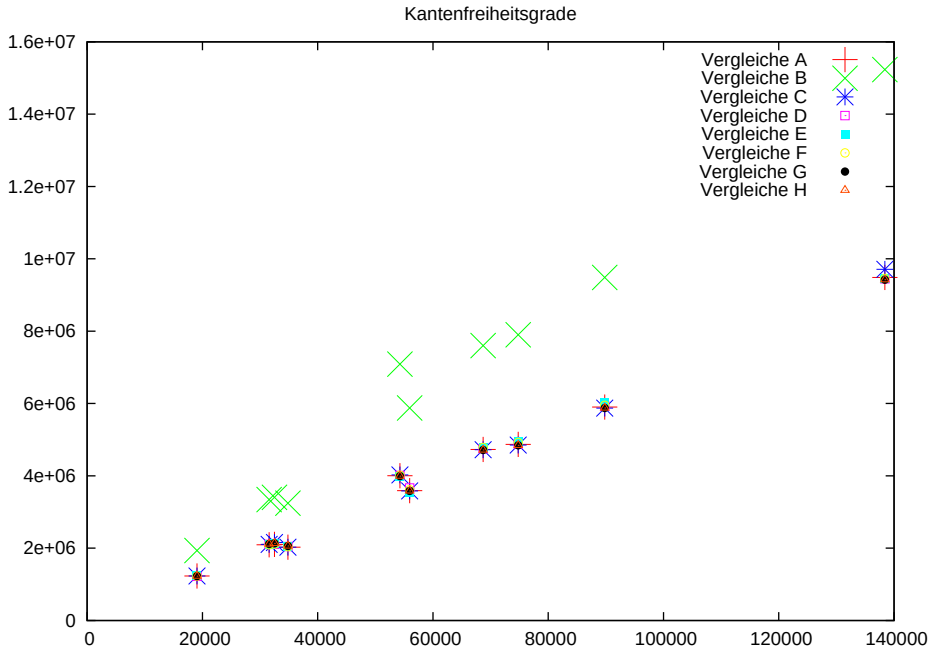
```
1   list<Cell>::iterator iterB;
2
3   for (iterB = cells.begin(); iterB != cells.end(); iterB++) {
4
5   q = iterB -> a;
6   r = iterB -> b;
7   s = iterB -> c;
 8  t = iterB -> d;
 9
10  edgeFillIn.a = min(q,r);
11  edgeFillIn.b = max(q,r);
12
13   typedef pair <Edges, float> E_Pair;
14
15  if (!E_Dofs.insert ( E_Pair ( edgeFillIn, 1 ) ).second)
16  {
17
18  E_Dofs[edgeFillIn]  +=1;
19  }
20
21  edgeFillIn.a = min(q,s);
22  edgeFillIn.b = max(q,s);
23  if (!E_Dofs.insert ( E_Pair ( edgeFillIn, 1 ) ).second)
24  {
25  E_Dofs[edgeFillIn]  +=1;
26  }
27
28  edgeFillIn.a = min(q,t);
29  edgeFillIn.b = max(q,t);
30  if (!E_Dofs.insert ( E_Pair ( edgeFillIn, 1 ) ).second)
31  {
32  E_Dofs[edgeFillIn]  +=1;
33  }
34
35  edgeFillIn.a = min(r,t);
36  edgeFillIn.b = max(r,t);
37  if (!E_Dofs.insert ( E_Pair ( edgeFillIn, 1 ) ).second)
38  {
39  E_Dofs[edgeFillIn]  +=1;
40  }
41
42
43  edgeFillIn.a = min(r,s);
44  edgeFillIn.b = max(r,s);
45  if (!E_Dofs.insert ( E_Pair ( edgeFillIn, 1 ) ).second)
46  {
47  E_Dofs[edgeFillIn]  +=1;
48  }
49                              17
50  edgeFillIn.a = min(s,t);
51  edgeFillIn.b = max(s,t);
52  if (!E_Dofs.insert ( E_Pair ( edgeFillIn, 1 ) ).second)
53  {
54  E_Dofs[edgeFillIn]  +=1;
55  }
56
57  }
58
```

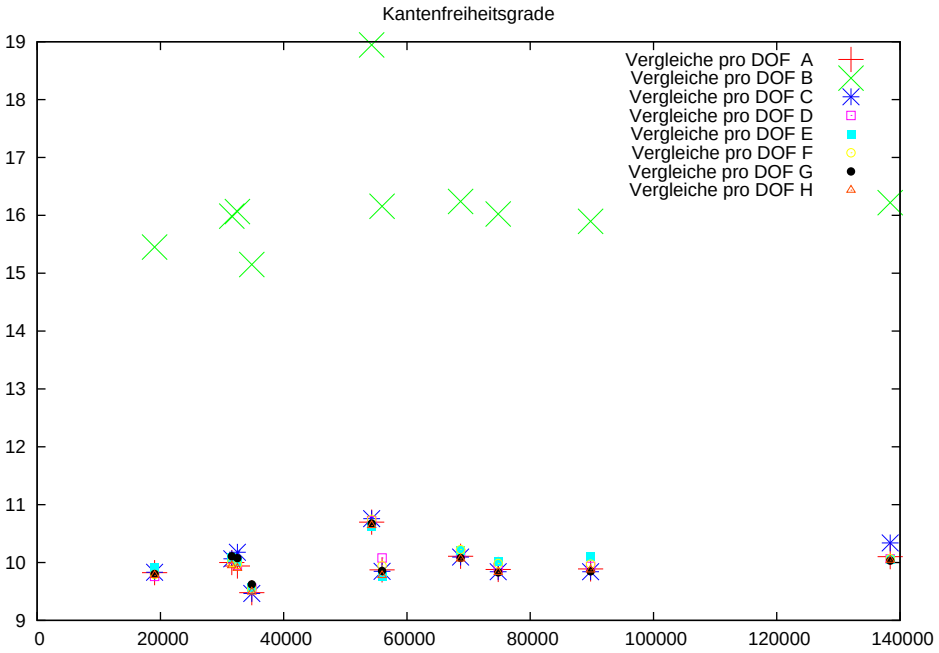
Abbildung 6: Durchgeführte Aktion für die Kanten

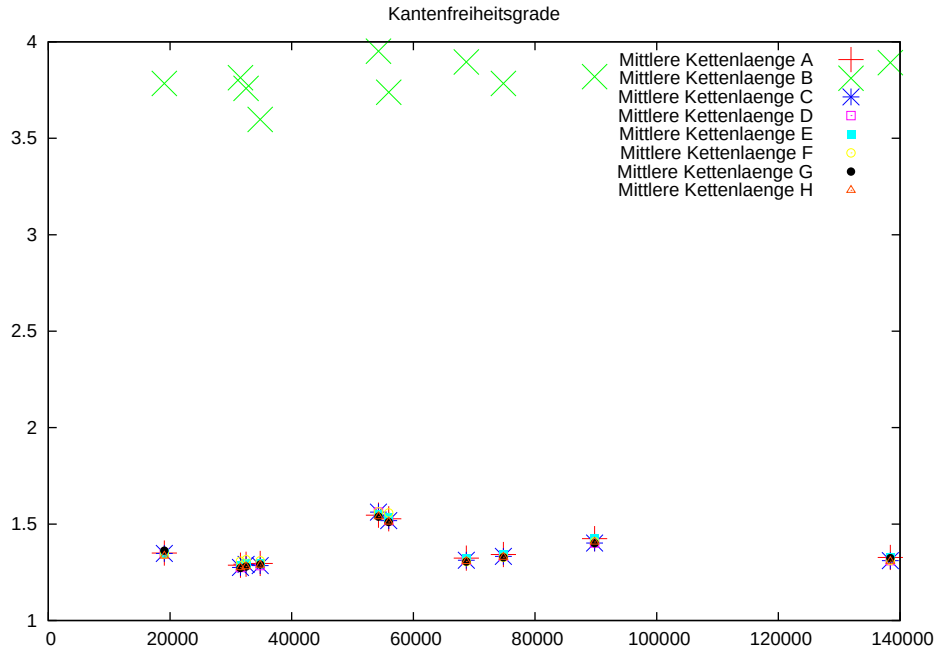


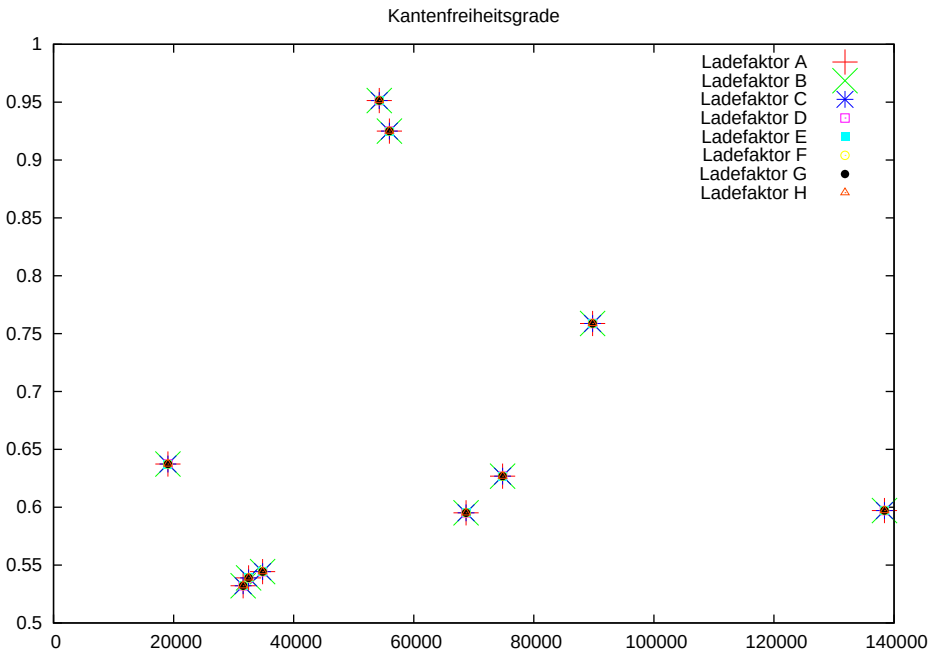
9.2 Ergebnisse für Kantenfreiheitsgrade

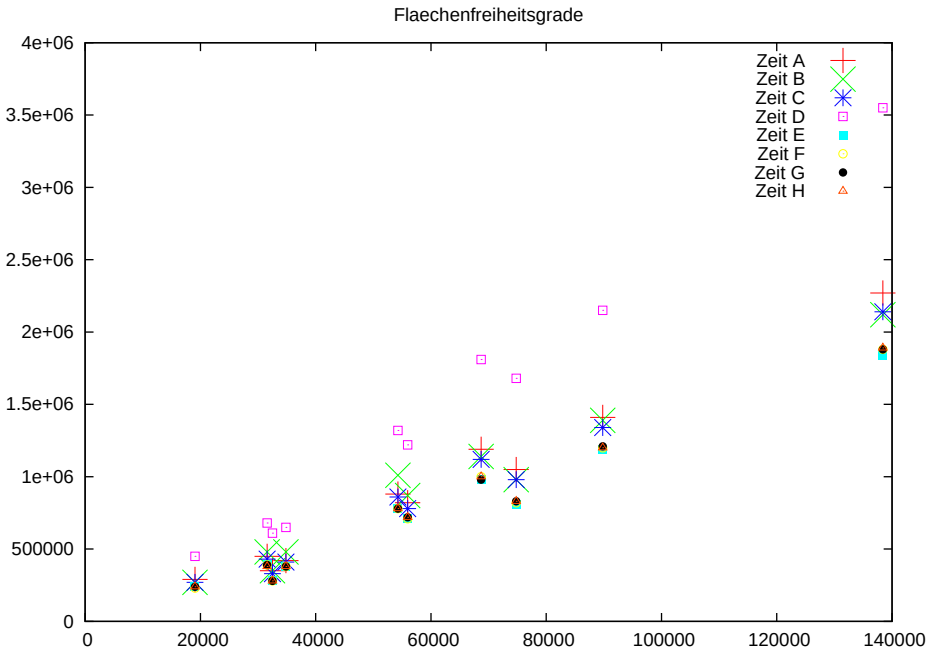

Kantenfreiheitsgrade
Zeit pro DOF A
Zeit pro DOF B
Zeit pro DOF C
Zeit pro DOF D
Zeit pro DOF E
Zeit pro DOF F
Zeit pro DOF G
Zeit pro DOF H
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0
20000
40000
60000
80000
100000
120000
140000


Kantenfreiheitsgrade
Vergleiche A
Vergleiche B
Vergleiche C
Vergleiche D
Vergleiche E
Vergleiche F
Vergleiche G
Vergleiche H
1.6e+07
1.4e+07
1.2e+07
1e+07
8e+06
6e+06
4e+06
2e+06
0
0
20000
40000
60000
80000
100000
120000
140000

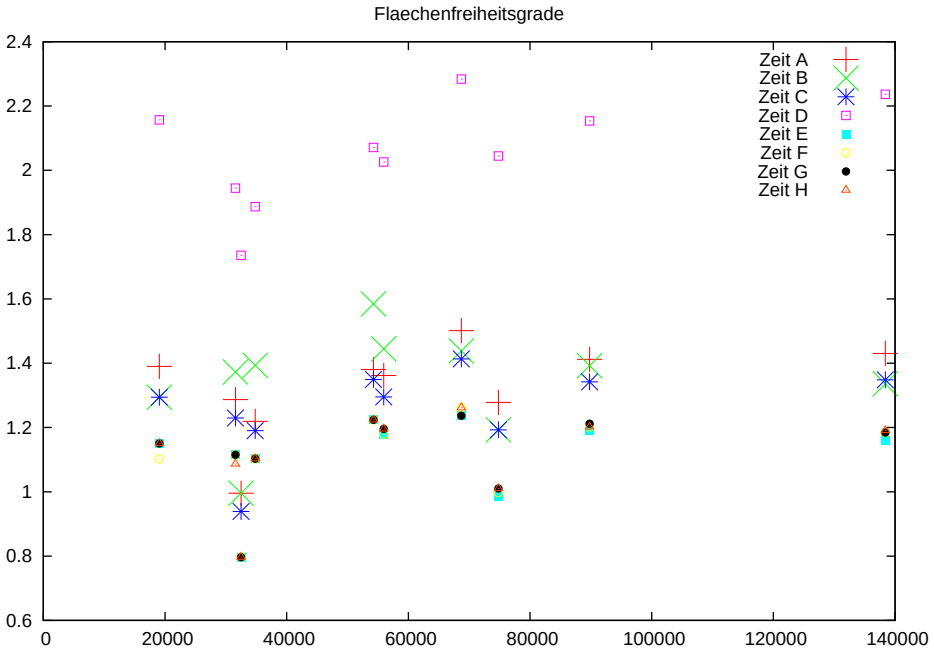

Kantenfreiheitsgrade
Vergleiche pro DOF A
Vergleiche pro DOF B
Vergleiche pro DOF C
Vergleiche pro DOF D
Vergleiche pro DOF E
Vergleiche pro DOF F
Vergleiche pro DOF G
Vergleiche pro DOF H
19
18
17
16
15
14
13
12
11
10
9
0
20000
40000
60000
80000
100000
120000
140000

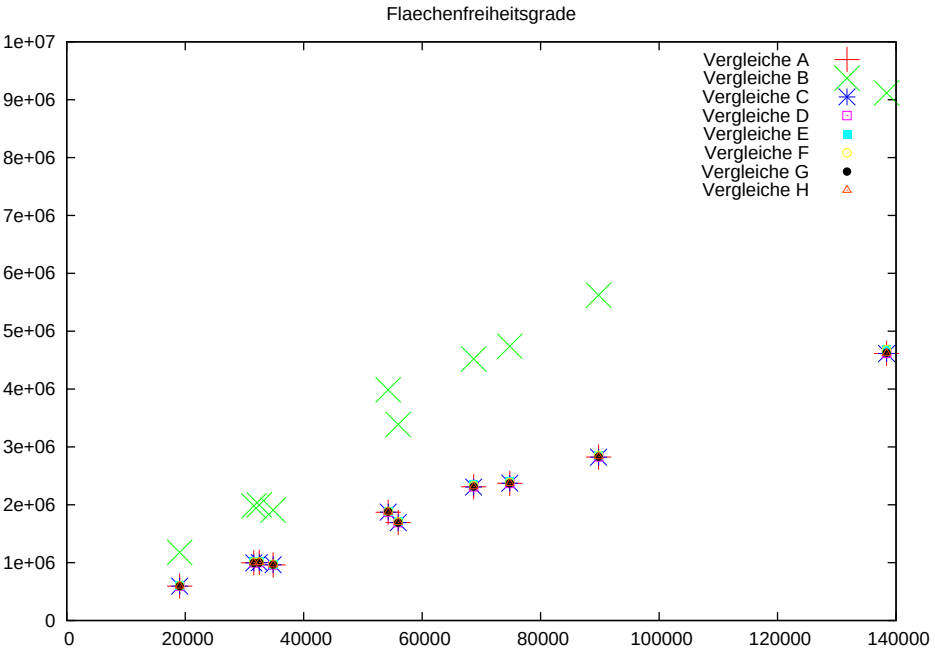

Kantenfreiheitsgrade
Mittlere Kettenlaenge A
Mittlere Kettenlaenge B
Mittlere Kettenlaenge C
Mittlere Kettenlaenge D
Mittlere Kettenlaenge E
Mittlere Kettenlaenge F
Mittlere Kettenlaenge G
Mittlere Kettenlaenge H
4
3.5
3
2.5
2
1.5
1
0
20000
40000
60000
80000
100000
120000
140000

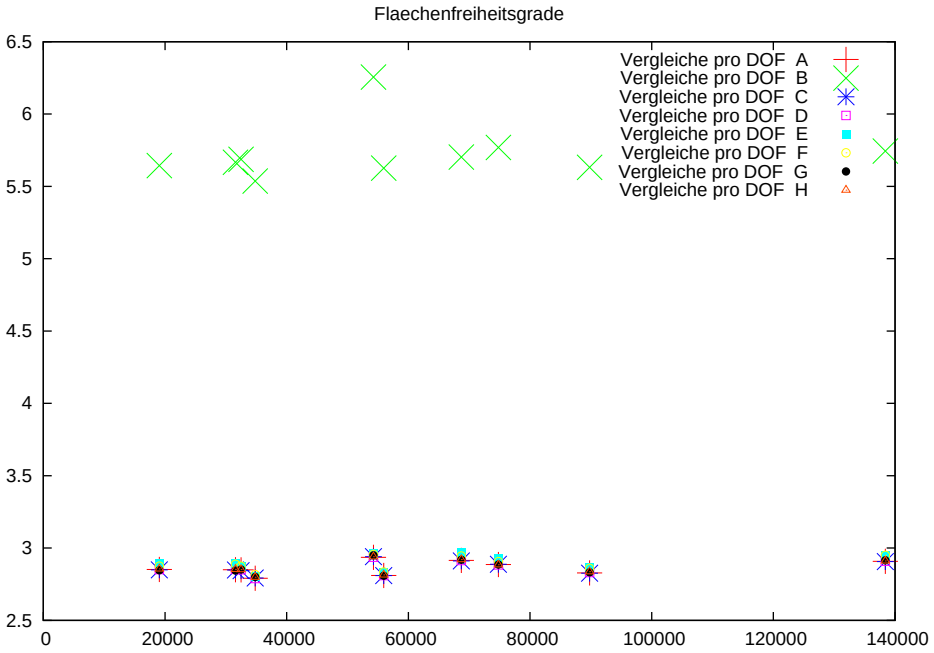

Kantenfreiheitsgrade
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0
20000
40000
60000
80000
100000
120000
140000
Ladefaktor A
Ladefaktor B
Ladefaktor C
Ladefaktor D
Ladefaktor E
Ladefaktor F
Ladefaktor G
Ladefaktor H

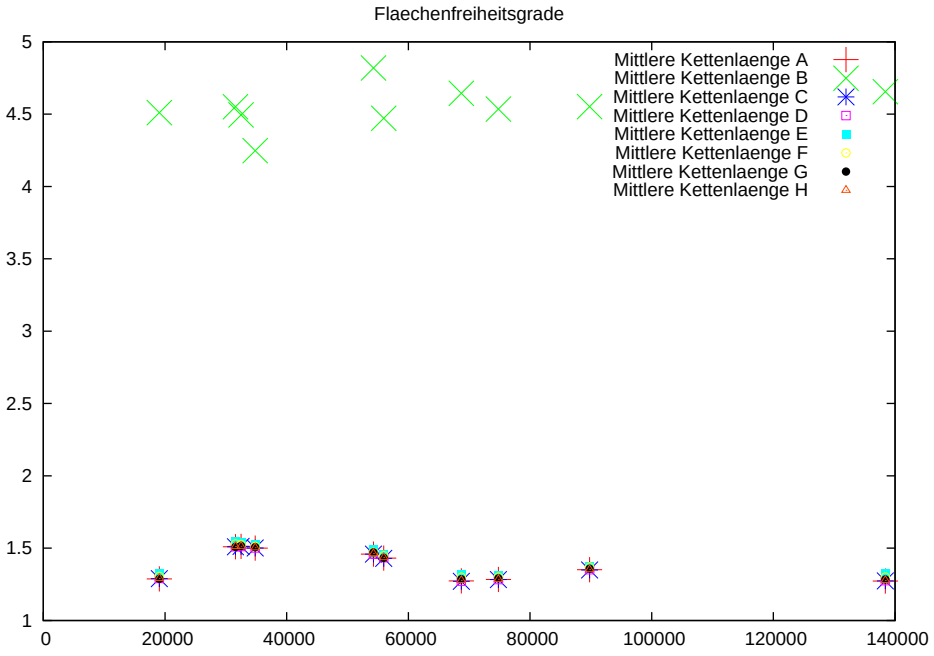


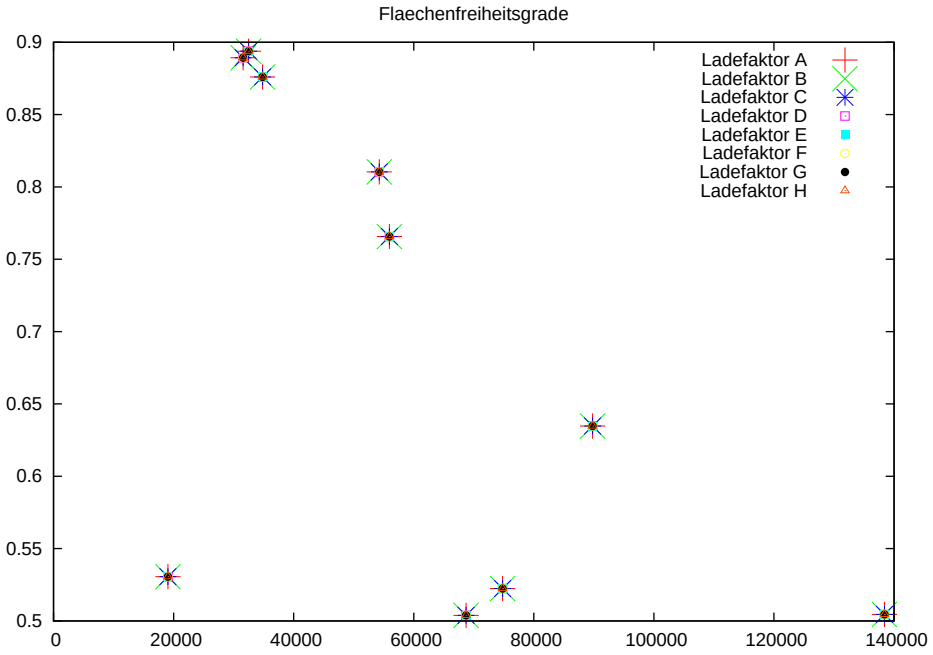
9.3 Ergebnisse für Flächenfreiheitsgrade


Flaechenfreiheitsgrade
Zeit A
Zeit B
Zeit C
Zeit D
Zeit E
Zeit F
Zeit G
Zeit H
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0
20000
40000
60000
80000
100000
120000
140000


Flaechenfreiheitsgrade
Vergleiche A
Vergleiche B
Vergleiche C
Vergleiche D
Vergleiche E
Vergleiche F
Vergleiche G
Vergleiche H
1e+07
9e+06
8e+06
7e+06
6e+06
5e+06
4e+06
3e+06
2e+06
1e+06
0
0
20000
40000
60000
80000
100000
120000
140000


Flaechenfreiheitsgrade
Vergleiche pro DOF A
Vergleiche pro DOF B
Vergleiche pro DOF C
Vergleiche pro DOF D
Vergleiche pro DOF E
Vergleiche pro DOF F
Vergleiche pro DOF G
Vergleiche pro DOF H
6.5
6
5.5
5
4.5
4
3.5
3
2.5
0
20000
40000
60000
80000
100000
120000
140000


Flaechenfreiheitsgrade
Mittlere Kettenlaenge A
Mittlere Kettenlaenge B
Mittlere Kettenlaenge C
Mittlere Kettenlaenge D
Mittlere Kettenlaenge E
Mittlere Kettenlaenge F
Mittlere Kettenlaenge G
Mittlere Kettenlaenge H
5
4.5
4
3.5
3
2.5
2
1.5
1
0
20000
40000
60000
80000
100000
120000
140000


Flaechenfreiheitsgrade
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0
20000
40000
60000
80000
100000
120000
140000
Ladefaktor A
Ladefaktor B
Ladefaktor C
Ladefaktor D
Ladefaktor E
Ladefaktor F
Ladefaktor G
Ladefaktor H

9.4 Interpretation

9.4.1 Kantenfreiheitsgrade

Die Ergebnisse sind nicht sehr eindeutig und ich kann hier nicht allzuviel sagen. Für eine bessere Interpretation müsste man wohl noch viel mehr Messungen machen.
Trotzdem habe ich mal im Zeitdiagramm gezählt, wieviel mal eine Funktion zuunterst erscheint. Falls die Punkte praktisch übereinanderliegen habe ich alle Funktionen gezählt. Dabei kam folgendes heraus:
G : 8 mal Zeit "minimal"
A,B,E : 7 mal Zeit "minimal"
F,H : 7 mal Zeit "minimal"
C,D : 2 mal Zeit "minimal"
D : 1 mal Zeit "minimal"
Eigentlich sollte die Zeit für die Messung linear mit der Grösse des Gitters zunehmen. Das ist aber beim 7. und 8. Gitter nicht der Fall. Ein Grund dafür ist sicher, dass nach dem 5. und 6. Gitter die Hashtabelle vergrössert wird.
Weiter fällt auf, dass die Methode B (Addition aller Komponenten) gar nicht so schlechte Resultate liefert. Das könnte daran hängen, dass Additionen billiger auszuwerten sind als Multiplikationen.

9.4.2 Flächenfreiheitsgrade

Ich habe gleich wie bei den Kantenfreiheitsgraden gezählt. Herausgekommen ist dabei das folgende:
E: 8 mal Zeit "minimal"
F: 5 mal Zeit "minimal"
G: 4 mal Zeit "minimal"
H: 3 mal Zeit "minimal"
A: 1 mal Zeit "minimal"
Auch hier müsste ich noch mehr Messungen machen für eine sinnvolle Interpretation. E scheint die beste zu sein. Ich vermute auch, dass man je nach Grösse der Gitter unterschiedliche Hashfunktionen wählen sollte.

Teil III
Anhang

A Kompatibilität des Programms

elems_in_bucket Diese Funktion ist zwar definiert in der sgi -HashMap, kommt aber in der offiziellen Version nicht vor. Sie hat mir aber keinerlei Probleme gemacht

B TODO

Was ich noch alles tun könnte und sollte

- Freiheitsgrade die sich im Innern der Tetraeder befinden habe ich in dieser Arbeit nicht betrachtet. Natürlich könnte man auch für diese eine Hashtable verwenden und verschiedene Hashfunktionen ausprobieren.
- Allgemein Programm-Code verbessern
- weitere Messungen auch von Gittern in anderen Grössen und von anderen Gittergeneratoren
- Parameter von Hashfunktionen (automatisch) optimieren.
- Das Programm ist nicht "gesichert". Wenn z.B. ein Gitter Fehlerhaft ist, stürzt sofort das ganze Programm ab.

Literatur

[sgi] http://www.sgi.com/tech/stl/

[cor] Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford
Introduction to algorithms. 2nd ed
Cambridge, MA: MIT Press

[lum] V.Y. Lum, P.S.T Yuen, M. Dodd
CACM 14 (1971),
228 - 239

[net] http://www.hpfem.jku.at/netgen/

[tet] http://tetgen.berlios.de/

[cli] http://www.gnu.org/software/libc/manual/

[mau] Hash Table Methods
W.D. Maurer, T.G. LEWIS
Computing Surveys, Vol. 7, No.1, March 1975
(Findet man in der Informatikbibliothek oder im Internet)