

Semesterarbeit WS 05/06

# Verlustfreie Kompression von G.711

Autoren:

**Giger Marco und Tanner Reto**

Betreuer:

**Prof. Dr. G. Schuster**

Modul Digitale Medien

Abteilung Elektrotechnik  
HSR Hochschule für Technik Rapperswil  
Rapperswil, 10. Februar 2006

# Inhaltsverzeichnis

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>EINLEITUNG .....</b>                                     | <b>4</b> |
| 1.1      | Abstract .....  | 4        |
| 1.2      | Zum Bericht .....   | 4        |
| 1.3      | Die Aufgabenstellung.....                                   | 5        |
| 1.4      | Zusammenfassung .....                                       | 6        |
| 1.5      | Zeitaufteilung (geplant).....                               | 7        |
| 1.6      | Zeitaufteilung (tatsächlich).....                           | 8        |
| <b>2</b> | <b>THEORIE .....</b>  | <b>9</b> |
| 2.1      | Datenkomprimierung .....                                    | 9        |
| 2.1.1    | Allgemein.....  | 9        |
| 2.1.2    | Einteilung in Verlustfrei & Verlustbehaftet .....           | 9        |
| 2.2      | Statistische Modelle .....                                  | 10       |
| 2.2.1    | Statisches Modell .....                                     | 10       |
| 2.2.2    | Dynamische Modelle .....                                    | 10       |
| 2.2.3    | Ordnungen der Statistischen Modelle .....                   | 10       |
| 2.2.4    | Grafische Darstellung der Wahrscheinlichkeitstabellen ..... | 11       |
| 2.3      | Entropiecodierung .....                                     | 12       |
| 2.3.1    | Entropie.....   | 12       |
| 2.4      | Lempel-Ziv-Welch .....                                      | 13       |
| 2.4.1    | Codierung.....  | 13       |
| 2.4.2    | Decodierung.....  | 13       |
| 2.5      | Huffman Code.....   | 14       |
| 2.5.1    | Funktion .....  | 14       |
| 2.5.2    | Varianten.....  | 14       |
| 2.5.3    | Der Huffman-Baum .....                                      | 15       |
| 2.6      | Shannon-Fano .....  | 17       |
| 2.7      | Laufängen Codierung (RLE).....                              | 17       |
| 2.8      | Arithmetische Kodierung.....                                | 17       |
| 2.8.1    | Grundprinzip .....  | 17       |
| 2.8.2    | Codierung.....  | 18       |
| 2.8.3    | Decodierung.....  | 18       |
| 2.8.4    | Beispiel für Codierung und Decodierung.....                 | 19       |
| 2.9      | G.711 .....   | 20       |
| 2.10     | Paketlänge und Übertragungszeit .....                       | 21       |

|            |   |           |
|------------|---|-----------|
| <b>3</b>   | <b>PRAKTISCHE ARBEITEN.....</b>                       | <b>22</b> |
| <b>3.1</b> | <b>Kompressor und Dekompressor.....</b>               | <b>22</b> |
| 3.1.1      | Modell von Kompressor und Dekompressor .....          | 23        |
| <b>3.2</b> | <b>LPC Codierung.....</b>                             | <b>25</b> |
| 3.2.1      | Variante1 „Koeffizienten selber suchen“ .....         | 26        |
| 3.2.2      | Variante2 „Koeffizienten aus Matlab“ .....            | 28        |
| 3.2.3      | Variante 3 „Einfacher aber besser“ .....              | 30        |
| <b>3.3</b> | <b>Struktur des Kompressors .....</b>                 | <b>34</b> |
| 3.3.1      | Grundidee.....  | 34        |
| 3.3.2      | Codecs.....   | 38        |
| 3.3.3      | Flussdiagramm des Encoders.....                       | 39        |
| 3.3.4      | Flussdiagramm des Decoders.....                       | 40        |
| 3.3.5      | PCM und G.711 .....                                   | 41        |
| 3.3.6      | Codierungsalgorithmen.....                            | 43        |
| 3.3.7      | Funktionsbeschreibungen.....                          | 45        |
| <b>4</b>   | <b>TESTRESULTATE .....</b>                            | <b>49</b> |
| <b>4.1</b> | <b>Kurzes Gespräch mit anschliessender Pause.....</b> | <b>49</b> |
| <b>4.2</b> | <b>Pausenloses Gespräch.....</b>                      | <b>49</b> |
| <b>4.3</b> | <b>Stille .....</b>                                   | <b>49</b> |
| <b>5</b>   | <b>FAZIT.....</b>                                     | <b>53</b> |
| <b>5.1</b> | <b>Erreichte Ziele.....</b>                           | <b>53</b> |
| <b>5.2</b> | <b>Nicht Erreichte Ziele .....</b>                    | <b>54</b> |
| <b>6</b>   | <b>AUSBLICK.....</b>                                  | <b>54</b> |
| <b>A</b>   | <b>ANHANG .....</b>                                   | <b>55</b> |
| <b>A.1</b> | <b>Inhalt der CD.....</b>                             | <b>55</b> |
|            | <b>LITERATUR .....</b>                                | <b>55</b> |

# 1 Einleitung

## 1.1 Abstract

In der Welt der Internettelefonie lässt sich im Moment ein sprunghaftes Wachstum feststellen. Durch die daraus entstehenden Datenmengen werden die Datenautobahnen, welche schon den gesamten Internet und E-Mailverkehr bewältigen müssen, zusätzlich belastet. Wie stark diese Belastung durch Audiogespräche in Zukunft noch steigen wird lässt sich im Moment noch nicht festhalten. Es zeichnet sich allerdings ab, dass das Datenaufkommen das durch Internettelefonie verursacht wird noch stark zunehmen wird. Ziel dieser Semesterarbeit ist es, die Datenrate eines solchen Internettelefonats durch verlustfreie Verfahren zu vermindern.

## 1.2 Zum Bericht

Dieser Bericht hält die Überlegungen, die Erkenntnisse und Erfahrungen, sowie die Ergebnisse und Resultate fest, die wir während unserer Semesterarbeit an der Hochschule für Technik Rapperswil (HSR) erarbeitet oder gemacht haben. Unsere Aufgabenstellung haben wir vom Labor für Digitale Medien ([www.medialab.ch](http://www.medialab.ch)) und unserem Dozenten Prof. Dr. Guido M. Schuster erhalten. Ebenfall von Prof. Dr. Guido M. Schuster wurden wir während der Arbeit an der HSR betreut, sowie durch Ratschläge und Hilfestellungen unterstützt. Zusätzlich wird die Bewertung unserer Arbeit durch ihn vorgenommen.

Die Anforderungen an diesen Bericht umfassen Inhalt, Sprache, Darstellung und eine klare Gliederung. Da der Inhalt natürlich sehr technisch orientiert ist, kann er nur schwer so verfasst werden, dass er für jedermann verständlich ist. Daher wurden die Texte so geschrieben, dass sie für einen Studenten der HSR am Ende des vierten Semesters klar verständlich sein sollten. Der Inhalt umfasst sowohl einen grundsätzlichen Teil der Theorie die für die Bearbeitung der Aufgabe benötigt wurde, als auch alle Experimente, Versuche und Probeläufe mit Verschiedenen Methoden. Zusätzlich alle Ergebnisse die wir herausgearbeitet haben und auch die planerischen Aspekte der gesamten Arbeit. Die Darstellung sollte ein ansprechendes Bild eines technischen Berichtes ergeben. Dieses haben wir versucht mit Hilfe von grafischen Darstellungen zu möglichst vielen Erklärungen etwas aufzulockern. Die Gliederung ergibt sich wie folgt: Im ersten Teil des Berichtes befinden sich Verzeichnisse, Zeiteinteilung, die Aufgabenstellung und eine Zusammenfassung der gesamten Arbeit. Im zweiten Teil haben wir versucht, einen kleinen Überblick über die theoretischen Hintergründe unserer Arbeit zusammenzustellen. Danach finden sich die Dokumentationen der von uns geschriebenen Funktionen, die Ergebnisse der Tests und Versuche die mit diesen erlangt wurden sowie eine Auswertung dieser Resultate. Im letzten Teil werden die Ergebnisse noch einmal kurz zusammengefasst, einige Anmerkungen dazu gemacht wie diese noch verbessert werden könnten, Literatur und Quellenangaben sowie genaues Inhaltsverzeichnis der beiliegenden CD.

## 1.3 Die Aufgabenstellung

**Thema:** Verlustfreie Kompression von G.711

**Studenten:** Giger Marco und Tanner Reto

**Betreuer:** Guido Schuster

### **Kurzbeschreibung**

Das Ziel dieser Studienarbeit ist es, G.711 (A-Law und  $\mu$ -Law, 10 und 20 ms Frames) in Echtzeit verlustfrei zu komprimieren. G.711 ist das Standardkompressionsverfahren, welches von fast jedem Voice over IP (VoIP) System unterstützt wird. Der Hauptvorteil von G.711 ist die gute Qualität. Da dies auch der ISDN Koder ist, tönt ein VoIP gleich wie ein ISDN Telefon. Der Hauptnachteil ist die hohe Datenrate von 64kBits/s ohne Headers und etwa 100kBits/s (pro Richtung) mit Headers. Mit einer verlustfreien Kompression (wobei die Bitrate jetzt auch variabel sein kann) bleibt die Qualität gleich und die Datenrate sollte im Durchschnitt signifikant kleiner werden.

### **Aufgabenstellung**

Einarbeitung in die Theorie

Evaluation/Selektierung eines geeigneten Kompressionsverfahren mit Matlab/C++

Entwicklung der Echtzeitsoftware in Visual C++

Austesten der Lösung

### **Erwartete Ergebnisse**

Dokumentation der Theorie, des Selektionsverfahren und der Software

Ein funktionsfähiges Echtzeitprogramm auf einem PC

Je ein Laborbuch

### **Arbeitsweise**

Sie führen ein persönliches Laborbuch, wo Sie aufschreiben wann Sie was für wie lange machen und was die Ergebnisse sind

Sie schicken mir vor jeder Sitzung eine Zusammenfassung welche dokumentiert, was Sie in der letzten Woche gemacht haben.

## 1.4 Zusammenfassung

Die Aufgabe bestand darin, einen verlustfreien Komprimieralgorithmus zu entwickeln, mit welchem die Audio-Daten die zwischen zwei IP-Telefonen verschickt werden, verlustfrei komprimiert und dekomprimiert werden können. Das Ziel ist es gewesen, einen Kompressionsfaktor von ca. 0.5 zu erreichen.

Die direkt von der Soundkarte kommenden PCM Daten durften dabei zuerst dem ITU-T G.711 Standard entsprechend, verlustbehaftet quantisiert werden. Das bedeutet, dass die Daten zuerst mittels diesem Verfahren quantisiert, und erst danach vom zu entwerfenden Komprimieralgorithmus komprimiert werden.

Die dazu erforderlichen Schritte umfassten das Einarbeiten in die grundlegenden Theorien der Datenkompression, das Finden und die Auswahl eines Verfahrens mit welchem die geforderten Ziele erreicht werden können, als auch die Programmierung und das Austesten des entworfenen Kompressors.

Eine grosse Herausforderung war, dass unser Verfahren schlussendlich ohne Verluste arbeiten musste und dies ist in der Welt der Audiokompression und dem Zeitalter des mp3's oder allgemein der MPEG-Komprimierung mehr als unüblich. Wir mussten also eine Möglichkeit finden, ohne all diese verlustbehafteten Verfahren eine ansehnliche Kompressionsrate zu erlangen.

Auf der Suche nach solchen Möglichkeiten stösst man relativ schnell auf die Theorie der Entropiecodierung und die dazu gehörenden bereits existierenden Verfahren mit den Namen Huffman, Shannon-Fano, Arithmetische Codierung und Lempel-Ziv-Welch mit welchem auch das allgemein bekannte ZIP Format arbeitet. Es stellte sich jedoch schnell heraus, dass z.B. ZIP für die Audiokompression ungeeignet ist, und sowohl Huffman als auch Shannon-Fano laut der Theorie nicht an die Leistung eines Arithmetischen Coders herankommen sollten. All diese Coder arbeiten nach dem Prinzip der Entropiecodierung und lag daher nahe, dass wir, um dem Coder die Arbeit zu erleichtern, oder dessen Leistung zu erhöhen, versuchen sollten die Entropie der Daten vor der Übergabe an den Coder zu optimieren.

Wir entschieden uns grundsätzlich, den Kompressor in der Form eines arithmetischen Coders zu bauen, da dieser laut Theorie die bestmögliche verlustfreie Kompression erzielen sollte, und mit verschiedenen anderen Methoden die anfallenden Daten für diesen Coder zu optimieren.

Die Optimierung der Daten, das heisst die Minimierung der Entropie in Datenblöcken, bedeutet grob gesagt, Ordnung in die Datenblöcke zu bringen oder anders gesagt, deren Informationsgehalt zu verkleinern. Auch dies musste natürlich geschehen ohne dass irgendwelche Verluste dabei entstanden. Über die Idee der Differenzbildung stiessen wir auf das LPC Verfahren, mit welchem die Daten mittels Voraussage von Zeichen und Übermittlung der Differenz der Voraussage und dem realen Wert eine kleinere Entropie erhalten sollten. Auf dem Weg zu einer für uns geeigneten Methode mussten wir verschiedene Varianten dieses Verfahrens entwerfen und testen. Mit diesen Tests waren einige Rückschläge verbunden, die von einer Verschlechterung der Entropie bis zur Einsicht reichten, dass sich eine viel versprechende Variante nicht verlustfrei wieder rückgängig machen liess, was natürlich für die Dekompression zwingend notwendig gewesen wäre. Zuletzt ist es uns aber gelungen eine Variante fertig zu stellen mit welcher es möglich ist, die Entropie von Daten, die Audioinformationen eines Gesprächs enthalten, merklich zu senken.

## 1.5 Zeitaufteilung (geplant)

| ID | Ausgabe/Jhrte Arbeit   | Start      | Ende       | Dauer | Okt 2005 |       | Nov 2005 |       |       | Dez 2005 |      |       |       | Jan 2006 |     |     |      | Feb 2006 |      |     |
|----|--|------------|------------|-------|----------|-------|----------|-------|-------|----------|------|-------|-------|----------|-----|-----|------|----------|------|-----|
|    |  |            |            |       | 10.27    | 10.30 | 11.03    | 11.13 | 11.20 | 11.27    | 12.4 | 12.11 | 12.18 | 12.25    | 1.1 | 1.8 | 1.15 | 1.22     | 1.29 | 2.5 |
| 1  | Führen eines Laborbuches -- MG/RT  | 19.10.2005 | 10.02.2006 | 83d   |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 2  | Einarbeiten in die Theoretischen Grundlagen zu den Verschiedenen Coder und Methoden -- MG/RT                                   | 26.10.2005 | 01.12.2005 | 27d   |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 3  | Auswahl der Verfahren die in Frage kommen -- MG/RT   | 01.12.2005 | 07.12.2005 | 5d    |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 4  | Erstellen eines Zwischenberichts -- MG/RT  | 12.12.2005 | 23.12.2005 | 10d   |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 5  | Entwerfen und ausprogrammieren einer verlustlosen LPC lösung zur Entropieoptimierung der Daten -- MG                           | 20.12.2005 | 30.01.2006 | 30d   |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 6  | Anpassen des Arithmetischen Coders an unsere Anwendung sowie mittels Mex-Funktion von Matlab her zugänglich machen -- RT       | 20.12.2005 | 30.01.2006 | 30d   |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 7  | Erstellen von Testdaten, Methoden zur Wandlung in unsere Form von G.711 sowie Aufteilung in 160 Byte Blöcke entwerfen -- MG/RT | 09.01.2006 | 12.01.2006 | 4d    |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 8  | Austesten der Methoden und optimieren der Funktionen -- MG/RT  | 16.01.2006 | 03.02.2006 | 15d   |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 9  | Integrieren des Kompressors in das Media-Lab-SIP-Telefon   | 06.02.2006 | 07.02.2006 | 2d    |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |
| 10 | Dokumentation der Arbeit -- MG/RT  | 25.01.2006 | 10.02.2006 | 13d   |          |       |          |       |       |          |      |       |       |          |     |     |      |          |      |     |

## 1.6 Zeitaufteilung (tatsächlich)

| ID | Ausgeführte Arbeit  | Start      | Ende       | Dauer | Gantt Chart |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
|----|---|------------|------------|-------|-------------|----------|----------|----------|----------|-------|------|-------|-------|-------|-----|-----|------|------|------|-----|
|    |   |            |            |       | Ok 2005     | Nov 2005 | Dez 2005 | Jan 2006 | Feb 2006 |       |      |       |       |       |     |     |      |      |      |     |
| 1  | Führen eines Laborbuches – MG/RT  | 19.10.2005 | 10.02.2006 | 83d   | 10.23       | 10.30    | 11.6     | 11.13    | 11.20    | 11.27 | 12.4 | 12.11 | 12.18 | 12.25 | 1.1 | 1.8 | 1.15 | 1.22 | 1.29 | 2.5 |
| 2  | Einarbeiten in die Theoretischen Grundlagen zu den Verschiedenen Coder und Methoden – MG/RT                                   | 26.10.2005 | 06.12.2005 | 30d   |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 3  | Auswahl der Verfahren die in Frage kommen – MG/RT   | 06.12.2005 | 12.12.2005 | 5d    |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 4  | Erstellen eines Zwischenberichts – MG/RT  | 12.12.2005 | 23.12.2005 | 10d   |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 5  | Entwerfen und ausprogrammieren einer verlustlosen LPC lösung zur Entropieoptimierung der Daten – MG                           | 20.12.2005 | 08.02.2006 | 37d   |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 6  | Anpassen des Arithmetischen Coders an unsere Anwendung sowie mittels Mex-Funktion von Matlab her zugänglich machen – RT       | 20.12.2005 | 09.02.2006 | 38d   |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 7  | Erstellen von Testdaten, Methoden zur Wandlung in unsere Form von G.711 sowie Aufteilung in 160 Byte Blöcke entwerfen – MG/RT | 09.01.2006 | 12.01.2006 | 4d    |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 8  | Austesten der Methoden und optimieren der Funktionen – MG/RT  | 16.01.2006 | 09.02.2006 | 19d   |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 9  | Anpassen des Huffman Coders an unsere Anwendung – RT  | 08.02.2006 | 10.02.2006 | 3d    |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |
| 10 | Dokumentation der Arbeit – MG/RT  | 25.01.2006 | 10.02.2006 | 13d   |             |          |          |          |          |       |      |       |       |       |     |     |      |      |      |     |

## 2 Theorie

In diesem Teil werden alle Verfahren und Methoden zur Datenkompression die im Zusammenhang mit dieser Arbeit stehen, kurz beschrieben. Diese Teile umfassen keine komplette theoretische Erfassung dieser Themen. Um einen tieferen Einblick zu bestimmten Themen zu erhalten, verweisen wir auf die im Literaturverzeichnis aufgeführten Bücher und Quellen.

### 2.1 Datenkomprimierung

#### 2.1.1 Allgemein

Als Datenkomprimierung oder auch Datenkompression bezeichnet man Verfahren, welche zur Reduktion des Speicherbedarfs von Daten eingesetzt werden. Bei solchen Verfahren wird die Datenmenge reduziert, indem der Informationsgehalt in den Daten optimiert wird. Das heisst, dass die Signifikanz (Bedeutsamkeit) der einzelnen Zeichen erhöht wird und dadurch die Entropie der Nachricht kleiner. Entropie ist kurz gesagt ein Mass für die Unordnung in einem System (der Begriff wird später noch ausführlicher erklärt).

#### 2.1.2 Einteilung in Verlustfrei & Verlustbehaftet

Bei der Komprimierung und Dekomprimierung von Daten wird grundsätzlich zwischen zwei Formen der Komprimierungsverfahren unterschieden: Da wären zum einen die verlustfreien Verfahren und zum anderen die verlustbehafteten Verfahren. Verlustfreie Verfahren werden überall dort eingesetzt, wo es darauf ankommt, dass die zu verarbeitenden Daten unverändert wiederhergestellt werden können. Das ist zum Beispiel der Fall, wenn Texte oder Messdaten übertragen werden sollen. Zu dieser Sorte Verfahren zählen unter anderem die Huffman-Kodierung, die Lauflängenkodierung und die Verfahren nach Lempel-Ziv. Die Gemeinsamkeit dieser Verfahren ist, dass sie Daten nur komprimieren können, solange eine gewisse Redundanz vorhanden ist. Die verlustfreien Kompressionsverfahren haben deshalb einen eingeschränkten Wirkungsgrad. Im Gegensatz dazu können die verlustbehafteten Verfahren Daten auch dann komprimieren, wenn keine redundanten Informationen vorliegen. Dadurch werden teilweise erheblich höhere Kompressionsraten erreicht. Diese Verfahren kommen in der Regel dann zum Einsatz, wenn Daten verarbeitet werden, die später mit den menschlichen Sinnen wahrgenommen werden sollen, also zum Beispiel bei Bildern oder Musik. Dabei muss man allerdings in Kauf nehmen, dass die komprimierten Informationen nicht mehr mit den Ausgangsdaten übereinstimmen, es werden Daten fehlen oder verändert sein. Es werden aber nicht einfach „willkürlich“ irgendwelche Informationen weggelassen. Vielmehr werden bei der Entwicklung der Algorithmen biologische und wahrnehmungspsychologische Modelle benutzt, mit deren Hilfe festgestellt werden kann, welche Informationen für menschliche Sinne wichtig sind und welche nicht. Da die Audiodaten, welche im Rahmen dieser Arbeit komprimiert werden bereits durch den Einsatz von G711 verlustbehaftet komprimiert wurden, kommen für die weitere Komprimierung nur noch Verfahren in Frage, welche verlustfrei arbeiten. Unter verlustfrei versteht man, das durch den Dekompressor wieder exakt die Daten erstellt werden, welche zuvor dem Kompressor übergeben worden sind. Also eine Komprimierung, wie sie auch bei Daten (Programmdateien, Textdateien etc.) eingesetzt werden muss, da dort keine Abweichungen von den Originaldaten zulässig sind.

## 2.2 Statistische Modelle

### 2.2.1 Statisches Modell

Bei einem statischen Modell wird eine statistische Tabelle erstellt, welche die Auftrittswahrscheinlichkeit aller Zeichen enthält. Diese Tabelle wird sowohl vom Kodierer wie auch vom Dekodierer verwendet und bleibt unverändert. Dadurch, dass zur Komprimierung in diesem Fall nur ein einziges Modell zur Verfügung steht, ist die Leistung des Verfahrens besonders bei Daten die sehr unterschiedlich sein können, sehr eingeschränkt.

### 2.2.2 Dynamische Modelle

Bei dynamischen Modellen werden die Wahrscheinlichkeiten laufend angepasst an die tatsächlichen Daten. Man unterscheidet generell zwischen „Vorwärts Dynamisch“ und „Rückwärts Dynamisch“. „Vorwärts Dynamisch“ bedeutet, die Wahrscheinlichkeit bezieht sich auf bereits kodierte Zeichen, d.h. nach dem kodieren eines Zeichens wird seine Wahrscheinlichkeit erhöht. „Rückwärts Dynamisch“ heisst, dass vor dem kodieren ausgezählt wird, wie oft jedes Zeichen vorkommt. Daraus lassen sich genaue Wahrscheinlichkeiten ermitteln. Die Anzahl der Zeichen werden dann während dem kodieren erniedrigt, sodass gegen Ende die Wahrscheinlichkeiten für die einzelnen Zeichen sehr exakt werden.

### 2.2.3 Ordnungen der Statistischen Modelle

Statistische Modelle werden in verschiedene Ordnungen eingeteilt. Die Ordnung eines Modells gibt an, wie viel der vorangegangenen Zeichen für die Bestimmung der folgenden Zeichen berücksichtigt werden. Ein Modell mit „Ordnung 0“ berücksichtigt demnach gar keine Vergangenheit für die Berechnung, sondern arbeitet mit einer festen Tabelle in der für jedes Zeichen angegeben ist wie gross seine Auftrittswahrscheinlichkeit ist. Ein Modell der „Ordnung 1“ berücksichtigt immer das vorangegangene Zeichen für die Bestimmung des folgenden, „Ordnung 2“ demnach die vorangegangenen zwei Zeichen usw. Je höher die Ordnung eines Modells ist, umso höher sollte die Genauigkeit werden mit der das nächste Zeichen bestimmt werden kann. Diese höhere Genauigkeit bringt allerdings auch Nachteile mit sich, man bezahlt die Genauigkeit mit stark wachsendem Speicherbedarf der ein solches Modell benötigt. In einem Zeichensatz mit 3 Zeichen zum Beispiel benötigt ein Modell der „Ordnung 0“ eine Tabelle, in der 3 Wahrscheinlichkeiten angegeben sind, mit der die Zeichen auftreten. Ein Modell der „Ordnung 1“ benötigt bereits drei Tabellen mit je drei Einträgen und ein Modell der „Ordnung 2“ schon drei Mal drei, also neun Tabellen, mit je drei Einträgen. Man kann sich vorstellen, dass in einem Zeichensatz mit zum Beispiel 256 Zeichen der Speicherbedarf der Tabellen eines Modells der „Ordnung 10“ nicht unerheblich ist. In der nachfolgenden Grafik werden die benötigten Tabellen noch mal aufgezeigt, jeder Kasten stellt eine Tabelle dar. Man sieht, dass bereits bei einem Modell der „Ordnung 2“ ein grosser Satz Tabellen benötigt wird, und diese würden in Wirklichkeit natürlich einiges mehr als zehn Einträge enthalten.



## 2.3 Entropiecodierung

Die Entropiecodierung ist eine Methode, mit der es möglich ist, Daten verlustfrei zu komprimieren. Dabei wird die vorhandene Redundanz in den zu komprimierenden Daten ausgenutzt. Zur Gruppe der Entropiecodierer gehören Lempel-Ziv-Welch, Shannon-Fano, Huffman und auch die arithmetische Codierung. Alle diese Verfahren beruhen darauf, dass sie die Daten, die von ihnen komprimiert werden, versuchen möglichst Entropieoptimal abzuspeichern. Dazu werden in allen genannten Verfahren verschiedene Varianten von Wahrscheinlichkeitstabellen eingesetzt, um damit Zeichen mit grosser Wahrscheinlichkeit mit möglichst kleinen Bitfolgen codieren zu können. Die Entropie von Daten liefert in nahezu allen Fällen eine rationale Zahl. Da es nur möglich ist Zeichen mit ganzen Anzahlen von Bits zu Codieren ist es nicht möglich eine absolut optimale Möglichkeit zu implementieren. Dies aus dem genannten Grund der ganzen Bits die benötigt werden, aber auch weil wie zum Beispiel beim arithmetischen Coder nicht mit unendlich genauen reellen Zahlen gerechnet werden kann und deshalb eine Einbusse bei der Genauigkeit hingenommen werden muss. Der ideale Entropiecodierer kann also auf einem realen Computer nicht umgesetzt werden.

### 2.3.1 Entropie

Der Begriff Entropie im Zusammenhang mit Informationstechnologie geht auf Claude E. Shannon zurück, der die Definition der Entropie 1948 in seiner Arbeit veröffentlichte. Die Entropie eines Datensatzes ist grundsätzlich eine Angabe dafür wie gross der Informationsgehalt innerhalb der Daten ist. In der Physik wird der Begriff Entropie auch als Mass für die Unordnung eines Systems eingesetzt. Je grösser die „Unordnung“ innerhalb eines Datensatzes ist, desto grösser ist auch sein Informationsgehalt und umgekehrt. Eine Datenfolge mit kleiner Entropie enthält auch Redundanzen oder statistische Regelmässigkeiten. Bei Datensätzen die einen sehr kleinen Informationsgehalt und damit eine kleine Entropie besitzen, werden also Entropiecodierer einen viel grösseren Kompressionsfaktor haben als bei solchen mit grosser Entropie.

Die Entropie hat die Pseudoeinheit „shannon“, die angibt, mit wie vielen Bit eine Information oder ein Datenblock minimal abgebildet werden kann. Dieser Wert ist auch der von einem Entropiecodierer maximal erreichbare Wert. Eine bessere Kompression kann damit nicht erreicht werden.

## 2.4 Lempel-Ziv-Welch

Das Lempel-Ziv-Welch Verfahren basiert grundsätzlich auf dem von zwei Israelischen Forschern, Jacob Ziv und Abraham Lempel, in den Jahren 1977 und 1978 vorgestellten Algorithmus. 1983 wurde von Terry Welch eine schnellere Variante des Lempel-Ziv Algorithmus entwickelt, welcher heute als LZW-Komprimierung bekannt ist. Das Verfahren arbeitet in der Standardimplementierung mit einem Wörterbuch, in welchem Zeichenfolgen abgespeichert werden, wie sie in den Daten vorkommen. Dies würde für einen Text zum Beispiel bedeuten, dass häufig wenn ein „q“ im Text vorkommt auch ein „u“ darauf folgt und daher als Muster im Wörterbuch abgelegt würde. Auf den Grundlagen von Lempel und Ziv entstand eine Vielzahl von Codierungsverfahren, die allgemein als LZ-Verfahren bezeichnet werden. In der Praxis durchgesetzt hat sich vor allem das Verfahren LZ77 das ein Bestandteil der bekannten Formate ZIP und GZIP sind. Auch das von CompuServe entwickelte GIF Format basiert auf dem LZW Algorithmus. Ebenfalls können die Formate JPEG und TIFF mit diesem Verfahren komprimiert werden.

### 2.4.1 Codierung

Bei der Codierung wird zunächst ein in der Implantation integriertes fixes Wörterbuch initialisiert, welches alle möglichen Einzelzeichen enthält. Dann wird mit der Kompression begonnen. Es wird also zunächst das erste Zeichen gelesen und übertragen. Danach wird das Zeichen hinzugenommen und die Folge von zwei Zeichen (erstes und zweites Zeichen) im Wörterbuch abgelegt, das Zeichen wird übertragen und das dritte wird eingelesen. Die Folge vom zweiten und dritten Zeichen wird ebenfalls im Wörterbuch abgelegt und so weiter. Wenn nun eine Folge die bereits im Wörterbuch eingetragen ist erneut auftaucht, wird nur noch die Nummer des Eintrages im Wörterbuch übertragen und zusätzlich an diese bereits vorhandene Folge noch das nächst folgende Zeichen angefügt und an einer anderen Stelle im Wörterbuch abgelegt. Bei jedem Auftreten einer Folge wird also ein weiteres Zeichen an die Folge angehängt und ebenfalls abgelegt. Damit werden die Folgen immer länger und die Kompressionsrate bei einem erneuten Auftreten einer solchen Folge immer besser. Treten solche Zeichenfolgen nicht mehr auf werden sie mit der Zeit von anderen überschrieben.

### 2.4.2 Decodierung

Die Decodierung funktioniert grundsätzlich gleich wie die Codierung. Es wird ebenfalls mit demselben vorinitialisierten Wörterbuch wie bei der Codierung begonnen. Das erste Zeichen entspricht dem Zeichen wie es auch bei der Codierung war, denn es wurde direkt übertragen. Auch das zweite Zeichen wurde nicht verändert, trifft unverändert beim Decodierer ein. Wie schon der Codierer legt nun der Decodierer diese Folge von zwei Zeichen an einer vorgegebenen Stelle in seinem Wörterbuch ab. Er liest dann das dritte Zeichen ein und legt auch die Folge vom zweiten und dritten Zeichen in seinem Wörterbuch ab usw. Auf diese Weise erstellt er für sich genau dasselbe Wörterbuch wie es schon der Codierer erstellt hat und kann, wenn vom Codierer anstelle eines Zeichens ein Wörterbucheintrag übertragen wird, die Zeichenfolge einfach aus seinem Wörterbuch auslesen. Natürlich muss das Wörterbuch dann auch mit diesen Zeichen weitergeführt werden.

## 2.5 Huffman Code

### 2.5.1 Funktion

Der Huffman Code ist nach dem Mathematiker David A. Huffman benannt, von welchem er 1952 entwickelt wurde. Huffman arbeitet nach dem Prinzip der Entropiecodierung. Der Code macht sich die Tatsache zu nutzen, dass zum Beispiel in einem Text, verschiedene Zeichen unterschiedlich oft vorkommen. So wird zum Beispiel der Buchstabe „e“ sehr viel häufiger verwendet als der Buchstabe „q“. In einer unkomprimierten Datei wird nun jeder Buchstabe nach dem ASCII-Code abgespeichert und beansprucht so je acht Bit. Ziel der Huffman Codierung ist es nun einen Code ähnlich dem ASCII-Code zu erstellen, bei welchem Zeichen oder Buchstaben die häufiger vorkommen, mit weniger Bit codiert werden als solche die nur selten vorkommen. So soll durch die Häufigkeit der Zeichen die mit wenigen Bits beschrieben sind, die Anzahl Bits die man im Durchschnitt pro Zeichen benötigt, kleiner gemacht werden. Der Trick besteht also darin, eine Tabelle zu erstellen, welche die Auftrittswahrscheinlichkeit jedes Zeichens erfasst und so einen optimalen Code erstellen kann. Mit solchen Tabellen arbeiten verschiedene Coder, wobei jeder Coder seine eigene Variante einer solchen Tabelle erstellt. Der Huffman-Coder liefert aber, in Gegensatz zu Shannon-Fano zum Beispiel, beweisbar immer eine optimale Lösung mit einer ganzen Anzahl Bits.

### 2.5.2 Varianten

- Statisch:* Die Tabelle ist unveränderlich. Sie ist bereits im Kompressor vorhanden und wurde, aus für den Einsatz des Kompressors statistisch optimalen Daten ermittelt. Vorteil dabei ist, das die Zeit die für das Erstellen der Tabelle benötigt wird, eingespart werden kann und der Kompressor so sehr schnell arbeitet. Nachteil ist jedoch bei sich ändernden Daten, dass der Code nicht genau mit den Tatsächlichen Wahrscheinlichkeiten übereinstimmt.
- Dynamisch:* Die Tabelle wird nach den in den zu komprimierenden Daten tatsächlichen Wahrscheinlichkeiten erstellt. Es wird also in einem ersten Schritt eine Code-Tabelle erstellt, nach welcher im zweiten Schritt die Daten komprimiert werden. Vorteil dieser Variante ist, das die Tabelle immer genau zu den Daten passt die komprimiert werden. Nachteile sind, dass mehr Zeit für die Kompression in Anspruch genommen wird und dass die erstellte Tabelle immer für die Dekompression mitgespeichert werden muss.
- Adaptiv:* Dies ist eine Mischung der beiden ersten Varianten. Es existiert eine Tabelle die von Beginn an benutzt wird, welche aber laufend an die zu bearbeitenden Daten angepasst wird. Ein Vorteil dabei ist es, dass nicht die ganze Tabelle mitgespeichert werden muss wie bei der dynamischen Variante.

### 2.5.3 Der Huffman-Baum

Die Wahrscheinlichkeitstabelle im Huffman Code wird auch Huffman-Baum genannt. Dieser Name hat sie von ihrem Aufbau her erhalten. In einem Huffman-Baum stehen die Zeichen die die grösste Auftrittswahrscheinlichkeit haben ganz oben, und je kleiner die Wahrscheinlichkeit eines Zeichens ist, desto weiter unten im Baum befindet es sich. Hier ein Beispiel eines solchen Baumes.

In diesem Beispiel eines Huffman-Baumes kommen nur die Zeichen a,b,r,k,d vor. Der Baum wird bei einem grösseren Alphabet entsprechend grösser. Die Häufigkeiten der verschiedenen Zeichen wie sie im entsprechenden Datensatz vorkommen lässt sich jeweils im Kreis der sich beim Zeichen befindet ablesen. So hat zum Beispiel das Zeichen „a“ die Häufigkeit 5. Im Kreis des Nebenastes steht die Häufigkeit 6. Dies ist die Häufigkeit aller sich in den Unterästen dieses Astes befindlichen Zeichen.

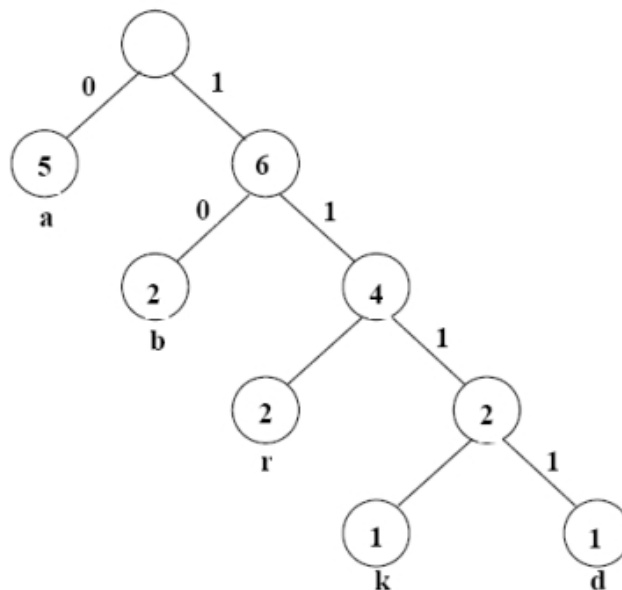


Abbildung 2

Der Baum funktioniert nun wie folgt:

Bei der Kompression erhält nach dem Erstellen des Baumes jedes vorkommende Zeichen seinen binären Wert den es innerhalb des Baumes besitzt. So wird also

einem „a“ der binäre Wert 0 zugeteilt, dem Zeichen „k“ der Wert 1110.

Bei der Dekompression werden die Bits nacheinander eingelesen und immer dem entsprechenden Ast im Baum gefolgt bis ein Zeichen als Ergebnis vorliegt. Nachdem ein Zeichen gefunden wurde wird mit dem folgenden Bit wieder ganz oben im Baum begonnen. Der Vorgang wird wiederholt bis alle Zeichen decodiert sind.

Der Baum wird nach folgenden Regeln erstellt:

1. Für jedes Zeichen wird ein Baum gesetzt, bei welchem am Beginn des Stammes die Häufigkeit seines Auftretens, bzw. seine Wahrscheinlichkeit angegeben ist.
2. Die zwei Bäume mit der kleinsten Wahrscheinlichkeit werden zu einem Baum zusammengefasst, welcher selber von nun an behandelt wird wie jeder andere Baum auch.
3. Schritt zwei wird solange wiederholt bis nur noch ein Baum existiert.

Der daraus entstandene Baum funktioniert wieder gleich wie der schon oben erklärte. Jede Abbiegung nach links entspricht im binären Code einer 0, jede Abbiegung nach rechts einer 1.

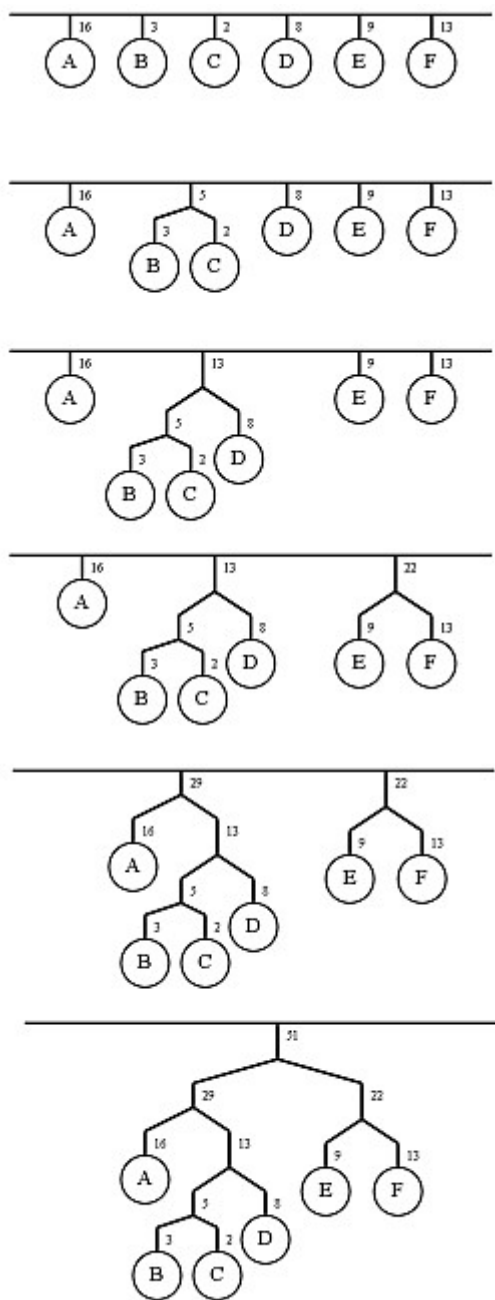


Abbildung 3

## 2.6 Shannon-Fano

Der Shannon-Fano Code ist nach Claude E Shannon und Robert M. Fano benannt und wurde 1960 entwickelt. Shannon-Fano arbeitet sehr ähnlich wie der Huffman-Code ebenfalls nach dem Prinzip der Entropiecodierung. Ähnlich wie schon bei der Huffman-Codierung wird eine wirksame Wahrscheinlichkeitstabelle benötigt, um eine Komprimierung zu erzielen. Für die Erstellung dieser Tabelle werden die Zeichen als Erstes nach ihrer Häufigkeit sortiert. Entlang der daraus entstandenen Reihenfolge werden die Zeichen dann so in zwei Gruppen eingeteilt, dass die Summe der Häufigkeiten der beiden Gruppen möglichst gleich ist. Diese Gruppen entsprechen in der Darstellung des Baumes der linken und der rechten Seite des Baumes. Für die daraus entstandenen Gruppen wird dann erneut dasselbe Verfahren wiederholt, bis die letzten Gruppen, also die äussersten Äste des Baumes aus nur noch einem Zeichen bestehen. Die Methode nach Huffman arbeitet grundsätzlich besser als die Lösung von Shannon-Fano da Shannon-Fano nicht immer in der Lage ist einen optimalen Baum zu erstellen. Shannon-Fano wird also die Leistung des Huffman Codes nicht übertreffen können.

## 2.7 Lauflängen Codierung (RLE)

Die Lauflängen Codierung auch RLE genannt ist ein Verfahren das grössere Folgen von gleichen Zeichen in einem Datensatz erkennt und diese günstig komprimiert. Dies erfolgt durch Auszählung der Folgen und Angabe von deren Länge. Damit dieses Verfahren arbeiten kann muss eine Spezielle art von Daten gegeben sein, die zum Beispiel oft bei Bildern die im BMP Format gespeichert sind vorliegt. Darin finden sich oft lange Bildpunktefolgen welche dieselbe Farbe haben. RLE komprimiert als Beispiel die Zeichenfolge „RRRRRRRRRAAAAAA-AAIII“ zu „8R9A3I“. Bei einem normalen Text wird diese Methode aber kaum gute Kompressionsraten erreichen, da längere Zeichenfolgen eigentlich nie vorkommen. Ebenso wenig ist sie geeignet um Audiosprachdaten zu komprimieren.

## 2.8 Arithmetische Kodierung

### 2.8.1 Grundprinzip

Bei der arithmetischen Codierung wird zunächst ein Intervall festgelegt, welcher in Coder und Decoder übereinstimmen muss. Dieser ist normalerweise in der Implementierung fest verankert und wird nicht verändert. Üblich ist zum Beispiel ein Intervall zwischen 0 und 1 also  $[0;1]$ . Innerhalb dieses Intervalls wird dann die ganze Komprimierung gemacht, wobei reelle Zahlen mit zum Teil sehr vielen Kommastellen entstehen. Um diese Methode nach Theorie optimal umsetzen zu können müsste mit unendlich genauen Realzahlen gerechnet werden, was mit einem Computer natürlich nicht möglich ist, da die Genauigkeit durch die Anzahl Bits eingeschränkt wird mit der ein Datentyp dargestellt wird. Bei der Implementation eines solchen Coders muss also gerundet werden, und dies zulasten der Optimalität des Coders. Die Implementation eines optimalen arithmetischen Coders ist auf einem Computer nicht realisierbar.

## 2.8.2 Codierung

Der Coder beginnt entweder damit, dass er, wie auch schon Huffman und Shannon-Fano, jedem vorkommenden Zeichen eine Auftrittswahrscheinlichkeit zuweist, oder aber er beginnt mit einer fixen ihm vorgegebenen Starttabelle und passt diese im Laufe der Codierung an. Als nächstes wird der Grundintervall, in unserem Fall  $[0;1]$  in Abschnitte aufgeteilt, welche in ihrer Grösse der Wahrscheinlichkeit der Zeichen entsprechen. Arbeitet der Coder mit der Methode in der er die Wahrscheinlichkeitstabelle laufend anpasst, wird diese Aufteilung von Zeichen zu Zeichen anders aussehen. Nun kann das erste Zeichen codiert werden. Dafür wird das Grundintervall durch den Teilintervall des zu codierenden Zeichens ersetzt und dieses neue Intervall erneut in diese den Wahrscheinlichkeiten entsprechenden Abschnitte aufgeteilt. Dieses Vorgehen wird nun wiederholt, bis alle Zeichen codiert sind. Zuletzt bleibt nach dem Codieren des letzten Zeichens ein Intervall stehen, aus welchem nun ein beliebiger Wert heraus genommen werden kann der dem Decoder übermittelt wird. An dieser Stelle spielt beim arithmetischen Codierer auch noch der Faktor Glück eine Rolle. Wenn dieser Intervall zum Beispiel zufällig in der Form  $[0.199999994746 ; 0.2000000001123]$  ausfällt, kann daraus die Zahl „0.2“ gewählt werden, welche natürlich mit viel weniger Bits übertragen werden kann als wenn viele Nachkomastellen übermittelt werden müssen. Diese Fälle treten natürlich relativ selten auf.

Dem Decoder wird also diese aus dem Intervall gewählte Zahl übermittelt. Zusätzlich dazu ist es notwendig das der Decoder weiss, wie viele Zeichen er decodieren muss, da sein Algorithmus sonst nicht weiss wann er mit dem Decodieren fertig ist. Diese Information erhält der Decoder entweder mittels einer mitgeschickten Zahl die die Anzahl Zeichen festlegt, oder es wird ein bestimmtes Zeichen definiert, welches nur dann auftritt wenn das Ende des Datenblocks erreicht ist. Damit der Decoder richtig arbeiten kann muss er sein Intervall genau gleich aufteilen können wie es der Coder gemacht hat. Dazu muss er, wenn der Coder seine Tabelle durch Auszählen erstellt hat, die Tabelle vom Coder ebenfalls erhalten. Arbeitet der Coder mit der veränderlichen Tabelle, kennt der Decoder die Grundtabelle selber und diese Daten müssen nicht übertragen werden. Nachteil ist jedoch, dass die Codierung am Anfang nicht optimal arbeitet, da die Wahrscheinlichkeitswerte nicht mit den effektiven Wahrscheinlichkeiten im Datenblock übereinstimmen.

## 2.8.3 Decodierung

Der Decoder beginnt mit demselben Intervall wie der Coder. Er macht die Aufteilung in Abschnitte nach der Tabelle die er vom Coder übermittelt bekommen hat, oder, bei einer sich ändernden Tabelle, mit der fixen Starttabelle. Nun schaut er, in welchem Abschnitt des Intervalls die Zahl liegt die der Coder ihm übertragen hat, in unserem Fall die „0.2“. Das Zeichen zu dem der entsprechende Abschnitt gehört ist nun das erste decodierte Zeichen. Der Abschnitt in dem die Zahl lag wird als neues Grundintervall gesetzt und in Abschnitte aufgeteilt. Diese Abschnitte können im Fall das die Wahrscheinlichkeiten angepasst, werden bereits anders aussehen als noch beim Zeichen zuvor, denn natürlich muss in diesem Fall auch der Decoder jetzt seine Starttabelle anpassen indem er die Wahrscheinlichkeit des Zeichens das er grade decodiert hat, erhöht. Nur so ist sichergestellt das die Decodierung fehlerfrei abläuft.

Die Anzahl Zeichen die noch decodiert werden müssen wird um eins erniedrigt und dann wird wieder geschaut welches Zeichen in seinem Abschnitt die Zahl 0.2 hat. So wird Zeichen um Zeichen decodiert, bis die Anzahl zu decodierenden Zeichen null ist.

## 2.8.4 Beispiel für Codierung und Decodierung

Der Ablauf bei der Codierung der Zeichenfolge „HSR“ ist in Abbildung 4 gezeichnet. Es wird also als erstes der Teilintervall des Zeichens H gewählt, dieser dann erneut in Abschnitte aufgeteilt, dann der Teilintervall von S gewählt, wieder aufgeteilt und aus diesem Intervall der Teilintervall von R gewählt. Aus diesem Teilintervall also aus dem Intervall  $[0.65 ; 0.5]$  kann jetzt eine beliebige Zahl gewählt werden die dem Decoder mitgeteilt werden muss. Bei diesem Beispiel arbeiten Coder und Decoder mit einer fixen Wahrscheinlichkeitstabelle, die dem Zeichen „S“ eine Wahrscheinlichkeit von 50%, dem Zeichen „H“ 30% und dem Zeichen „R“ 20% zuweist. Dem Decoder teilen wir nun zum Beispiel die Zahl „ $x = 0,628$ “ und der Wert „ $y = 3$ “ als Anzahl Zeichen die zu decodieren sind mit.

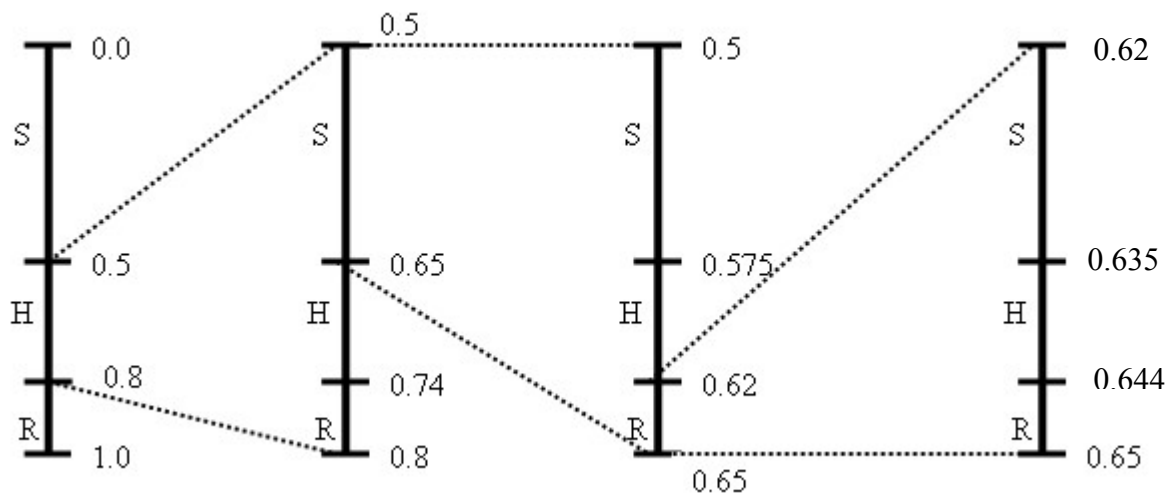


Abbildung 4

Bei der Decodierung wird der Ablauf fast exakt wiederholt, es wird nach jedem neuen Einteilen des Teilintervalls geschaut, in welchem Abschnitt die Zahl „ $x = 0.628$ “ die vom Coder übertragen wurde liegt und das entsprechende Zeichen ausgegeben. Daraus entsteht wieder die Zeichenfolge „HSR“.

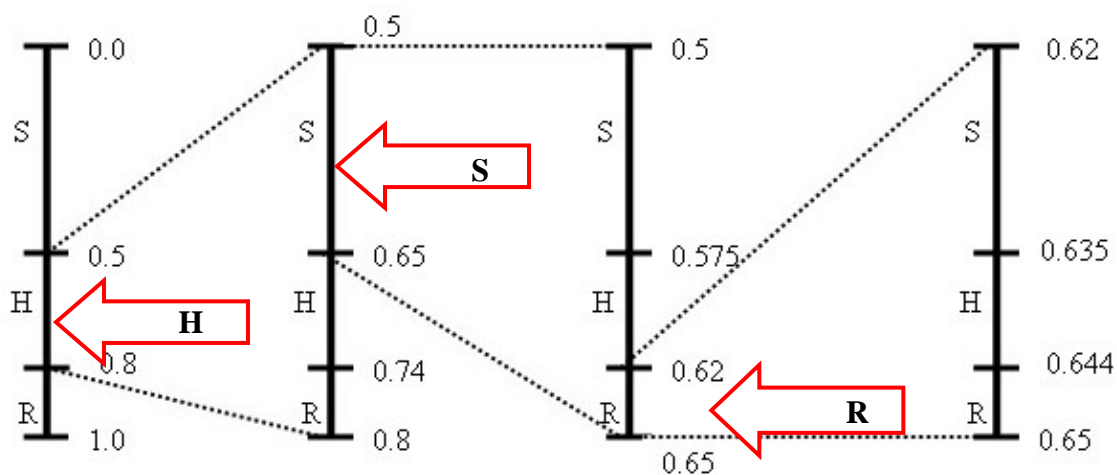


Abbildung 5

## 2.9 G.711

Der ITU-T Standard G.711 wird verwendet, um die Datenmengen eines digitalisierten Audiosignals zu reduzieren. Dabei werden die Daten die von einem Mikrofon über die Soundkarte als PCM 16-Bit Werte abgetastet werden auf 8-Bit Werte reduziert. Diese Reduktion wird mittels einer nicht linearen Quantisierung erreicht. Nicht linear ist sie, um den Fehler, also die Veränderung des Audiosignals, für das menschliche Ohr möglichst nicht hörbar zu machen. Das heisst, dass die Quantisierungsschritte für laute Signale viel grösser sind als für leise, da auch der Mensch Veränderungen in leisen Signalen viel besser wahrnehmen kann als in lauten. Es werden insgesamt 15 Stufen unterschieden, bei denen der Quantisierungsfehler unterschiedlich gross ist und in allen Stufen zusammen gibt es 256 verschiedene Werte. Die Quantisierung wird nach der Kurve im folgenden Bild (Abbildung 6) vorgenommen. Es gibt zwei Varianten von G.711, zum einen die in Europa verwendete A-Law genannte, und zum anderen die in Japan und Nordamerika eingesetzte Variante  $\mu$ -Law. Die beiden Varianten gleichen sich stark sind jedoch nicht kompatibel. A-Law Daten müssen daher durch entsprechende Konverter in  $\mu$ -Law gewandelt werden und umgekehrt.

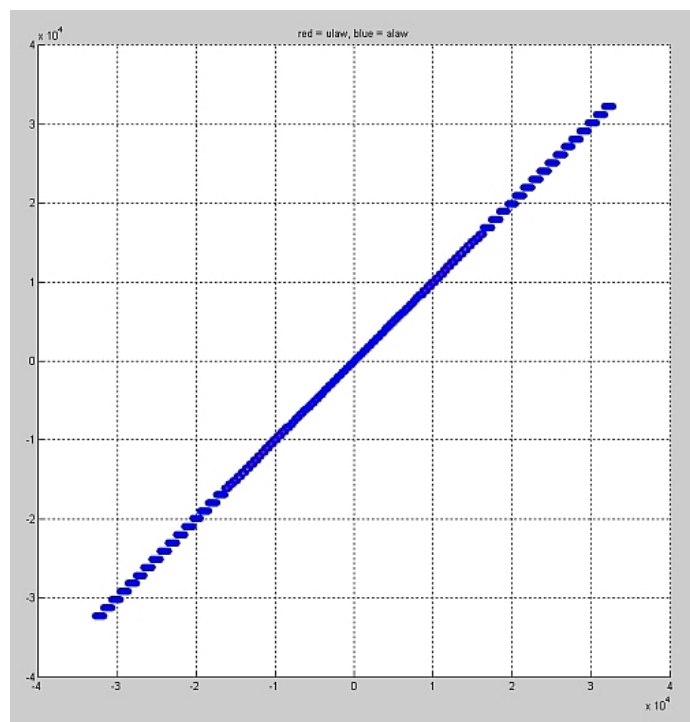


Abbildung 6

Bei einer näheren Betrachtung der Kurve zeigen sich die kleinen Unterschiede zwischen den beiden Varianten (Abbildung 7). Rot dargestellt ist  $\mu$ -Law und blau A-Law. Gut erkennen kann man auch, ungefähr in der Bildmitte, einen Übergang von einer Stufe zur anderen. Man kann sehen dass ab dem Wert 16000 die Balken länger werden, das quantisierte Signal also über einen grösseren Bereich denselben Wert behält und der Quantisierungsfehler daher grösser wird umso lauter das Eingangssignal ist.

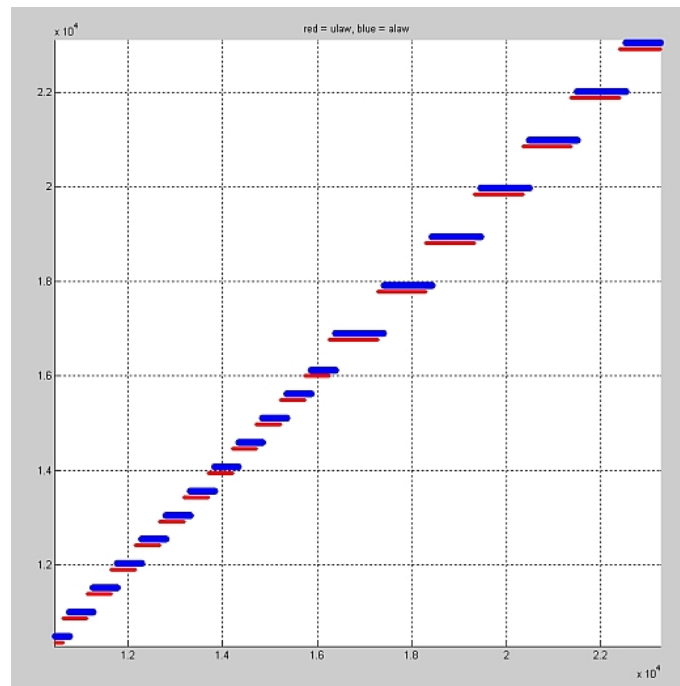


Abbildung 7

Für unser Verfahren wurde grundsätzlich die  $\mu$ -Law Variante benutzt, welche noch für diese Anwendung angepasst wird. Um besser mit den Werten arbeiten zu können, und um einfachere vorzeichenlose 8-Bit Werte zu erhalten ist die  $\mu$ -Law Werte so verändert worden das nur noch Werte zwischen 0 und 255 vorkommen, was den 256 Schritten von G.711 entspricht. Zusätzlich sind sie so geordnet worden, dass die Audiokurve grundsätzlich durch eine Verschiebung um -127 wieder dargestellt werden kann. Die Qualität der Signale entspricht aber exakt der von G.711  $\mu$ -Law und somit der eines ISDN Telefons. Die vorgenommenen Veränderungen können absolut verlustfrei wieder rückgängig gemacht werden.

## 2.10 Paketlänge und Übertragungszeit

Die Paketlänge sagt aus, wie lange die Abschnitte sind, in welche das Gespräch zerschnitten wird. So werden die Daten immer während einer Paketlänge aufgezeichnet, als Datenblock gespeichert und übertragen oder in unserem Fall dem Kompressor übergeben und dann übertragen. Als Übertragungszeit gilt die Zeit, die vom Moment an in dem etwas gesprochen wird vergeht, bis das gesprochene beim Gesprächspartner ankommt. Bei einer Paketlänge von 20ms vergehen also die ersten 20ms ohne dass das gesprochene überhaupt auf den Weg geschickt wird. Da der Datenblock der so aus dem in 20ms aufgezeichneten Gesprächsteil besteht noch komprimiert muss, vergeht während der Kompression weitere Zeit die schlussendlich der Übertragungszeit zugerechnet werden muss. Danach benötigen die Übertragung selber und die Dekompression beim Empfänger noch weitere Zeit die wieder bei der Übertragungszeit aufaddiert wird. Bereits eine Übertragungszeit von 150ms wird von empfindlichen Benutzern als störend empfunden und ab 300ms wird sie auch für unempfindliche Benutzer lästig. Es ist also nötig, dass Kompressor und Dekompressor relativ schnell arbeitet, damit auch wenn die Übertragung selber etwas mehr Zeit benötigt die Übertragungszeit nicht über die tolerierbare Zeit hinaus ansteigt.

## 3 Praktische Arbeiten

### 3.1 Kompressor und Dekompressor

Die folgende Grafik (Abbildung 8) zeigt ein Modell unseres Kompressors und Dekompressors, das den grundsätzlichen Aufbau darstellt. Wie zu sehen ist, verwenden wir mehrere Modelle zur Kompression, zwischen denen während der Laufzeit umgeschaltet wird. Grundsätzlich setzen wir die Modelle vor dem Coder dazu ein, die Daten in eine optimierte Form zu bringen die es den beiden Coder, arithmetischer Coder und Huffman Coder, ermöglicht eine möglichst grosse Kompressionsrate zu erreichen. Die daraus entstehenden Daten werden dann von den beiden Codern komprimiert. Der in der Grafik dargestellte Auswerter empfängt alle Varianten der komprimierten Daten und vergleicht ihre Grösse. Er entscheidet aufgrund der Grösse der Pakete welches übertragen wird. Damit soll eine optimale Kompression erreicht werden, sowohl für die Stille die in einem Telefongespräch einen entscheidenden Anteil einnimmt, als auch für die Blöcke die Sprache enthalten. Der Dekompressor kann anhand eines von jedem Modell den Blöcken angefügten Bytes entscheiden mit welchem Modell er den erhaltenen Block dekomprimieren muss. Beim Dekompressor laufen somit nicht mehrere Modelle parallel wie dies im Kompressor der Fall ist, sondern nur noch dasjenige das effektiv benötigt wird. Diese Methode bei der ein Byte als Information für den Dekompressor übertragen werden muss, spart bei der Dekompression Zeit und garantiert einwandfreies Funktionieren auch wenn einzelne Blöcke verloren gegangen sind.

### 3.1.1 Modell von Kompressor und Dekompressor

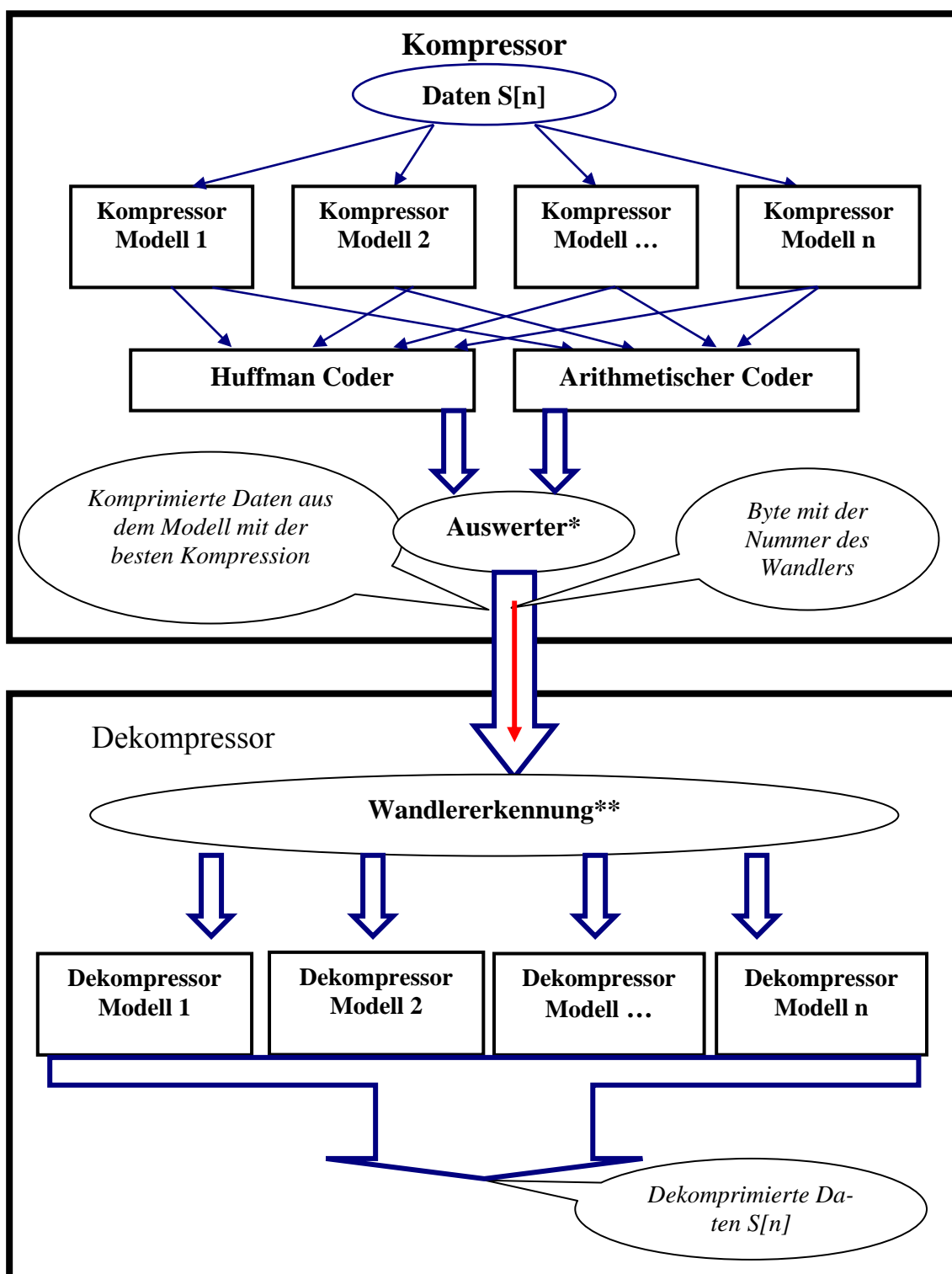


Abbildung 8

- \*Auswerter: Der Auswerter vergleicht die Grössen der von den verschiedenen Modellen gelieferten Pakete. Er überträgt nur das kleinste Paket. Die Pakete enthalten ein Byte in dem angegeben ist, welcher Kompressor es komprimiert hat. Dies ist nötig für die Dekompression.
- \*\*Wandlererkennung: Am Eingang des Dekompressors wird das Byte das angibt von welchem Kompressor das Packet komprimiert wurde ausgewertet und das Packet anschliessend mit dem entsprechenden Dekompressor dekomprimiert.

## 3.2 LPC Codierung

LPC Systeme sind seit geraumer Zeit im Einsatz. So ist zum Beispiel LPC-10 (bedeutet das mit einem Filter mit 10 Parametern gearbeitet wird) seit 1984 standardisiert und erlaubt die Übertragung von Sprache mit 2400Bit/s bei einer Datenrate des Eingangssignals von 64kBit/s. LPC ist ein verlustbehaftetes Verfahren und kommt somit in seiner normalen Form für unsere Anwendung nicht in Frage. Zunächst trotzdem einige Erklärungen zur Funktion von LPC. Das LPC (Linear Predictive Coding) Verfahren, zu deutsch lineare prädiktive Kodierung, geht von einem vereinfachten Modell der menschlichen Stimmerzeugung aus. Dabei wird die Stimmerzeugung in zwei Bereiche aufgeteilt, zum einen von den Stimmbändern erzeugte Töne, diese werden durch Signalgeneratoren ersetzt, und zum anderen vom Artikulationsstrakt erzeugte Geräusche, welche durch ein System von linearen Filtern nachgebildet werden. Bei der Wiedergabe werden dann stimmhafte Laute durch Impulsgeneratoren und stimmlose Laute durch Rauschgeneratoren nachgebildet. Diese Unterscheidung macht für unser Verfahren keinen Sinn. Was wir uns zu Nutze machen möchten ist das Verfahren für die Voraussage von Werten. Mit deren Hilfe ist es möglich, die Entropie der Daten zu verbessern und sie dadurch in eine optimale Form, also eine Form mit möglichst kleiner Entropie, für den arithmetischen Coder zu bringen. Je weiter die Entropie damit gesenkt werden kann umso bessere Kompressionswerte sollten erreicht werden können. Laut Theorie sollte es möglich sein, dass die Kompression sehr nahe an die Werte der Entropie herankommt.

Beim LPC-Verfahren war es wichtig darauf zu achten, dass die Methode absolut verlustfrei bleibt. Bei der Berechnung der Voraussagen entstanden aber reelle Zahlen mit beliebig vielen Kommastellen, was auch bei der Bildung der Differenz zwischen Voraussage und Effektivem Wert zu einer reellen Zahl führte. Dieser Wert konnte aber vom Kompressor nicht verarbeitet werden, da er nur mit vorzeichenlosen Ganzzahlen zwischen 0 und 255 arbeitet (unsigned char). Es musste also ein Weg gefunden werden die Fehler die bei der Rundung der Werte entstehen können auszumerzen, sodass das System verlustfrei arbeitet. Dieses Problem wurde so gelöst, dass direkt nach dem Erstellen der Voraussage die Rundung des vorhergesagten Wertes vorgenommen wird. Damit wird aus der Voraussage eine ganze Zahl (integer Wert) mit welchem wir auch problemlos die Differenz in ganzen Zahlen berechnen können.

Ein weiteres Problem das sich ergab waren Sprünge, also Differenzen, die negative Wert bei der Differenzbildung ergaben. Da wir mit dem Datentyp „unsigned Char“ arbeiten konnten negative Werte nicht übergeben werden. Diese Schwierigkeit wird umgangen, indem wir die negativen Sprünge einfach in positive umrechnen, was für uns passende positive Werte ergibt und gleichzeitig genau dasselbe Ergebnis ergibt. Dabei wird ausgenutzt, das bei einem Sprung von zum Beispiel 4 auf 1, also einem Sprung um -3 auch der längere Weg von 256 +(-3) also um +253 zur gleichen Stelle gelangt, wie in Abbildung 9 dargestellt.

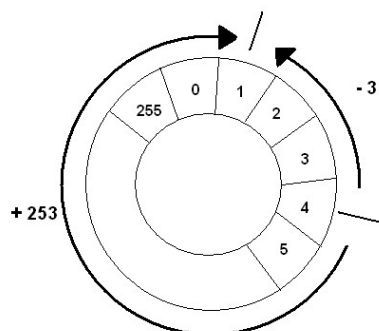


Abbildung 9

### 3.2.1 Variante1 „Koeffizienten selber suchen“

Für eine LPC Variante mit 3 Koeffizienten haben wir versucht, die Koeffizienten zu ermitteln indem wir mit Hilfe einer Schleife alle möglichen Varianten eingesetzt und den entstehenden Block komprimiert haben. Daraus hätten wir die Koeffizienten finden wollen mit denen die grösste Kompression möglich wäre. Es hat sich gezeigt, dass es keine Koeffizienten gibt, welche eine wesentliche Verbesserung der Kompression im Vergleich zur Kompression ohne LPC gibt. Der dunkelblaue Balken in Abbildung 11 stellt die Grösse der Blöcke nach der Kompression dar, wobei immer 6 Blöcke mit denselben Werten gerechnet wurden und die grösse Addiert ist. Die Darstellung ist nicht Optimal, die Grafik wurde aber nicht noch einmal erstellt, da die Berechnung ca.18 Stunden in Anspruch nimmt. Der kleinste hier erreichte Wert ist 1011 was eine durchschnittliche grösse pro Block von  $(1011/6)$  168.5Byte ergibt. Das bedeutet dass unser Kompressor die Blöcke durchschnittlich um 8.5Byte grösser gemacht hat als sie es zuvor waren. Der Kompressor ist also in dieser Variante nicht brauchbar. Der hellblaue Balken stellt die Koeffizienten dar die sich natürlich laufend in einem bestimmten Intervall geändert haben und so nur einen breiten Balken hinterlassen haben.

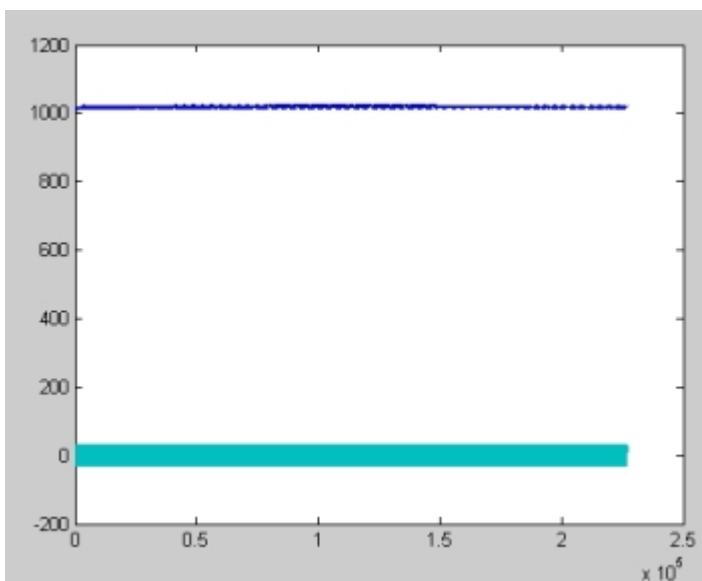


Abbildung 11

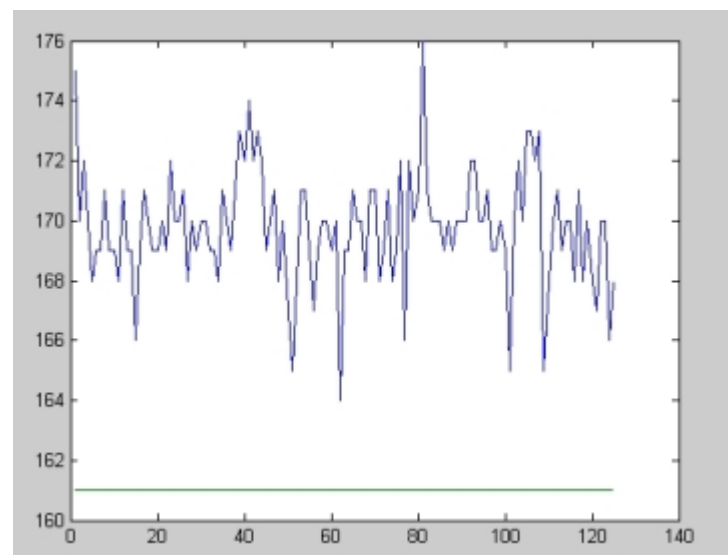
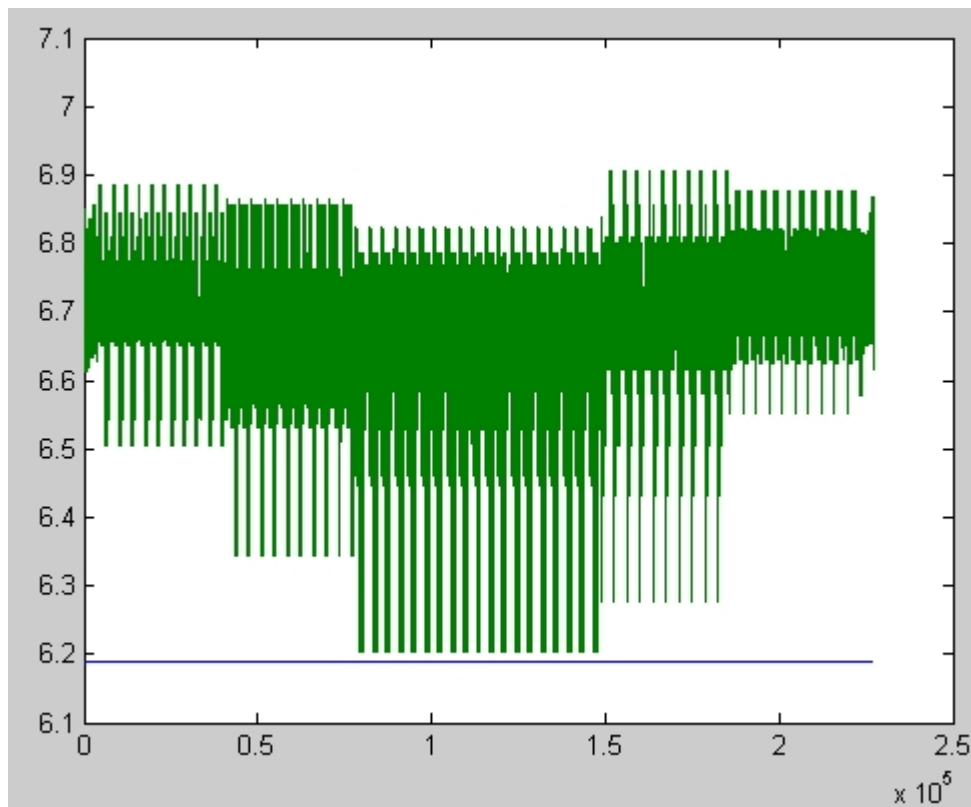


Abbildung 10

Besser sichtbar wird das Problem das die Datenblöcke vergrössert anstatt verkleinert werden in Abbildung 10. Die untere Linie zeigt hier die Blockgrösse die dem Kompressor übergeben wird, sie beträgt konstant 161Byte. Die obere blaue Linie zeigt die Grösse der Blöcke, die vom Kompressor zurückgegeben wurden. Auch hier wieder einiges grösser als zuvor.

Wir haben dann mit Tests festgestellt, dass dieser Effekt sowohl bei Blöcken auftritt die durch den LPC Wandler gelaufen sind als auch bei solchen die im Originalzustand sind. Also mussten wir die Funktion von LPC anders testen können, da uns die Blockgrösse keine verlässlichen Daten brachte. Also wurde der gleiche Versuch, diesmal jedoch wurde nicht die grösse der Blöcke sondern die Entropie der Blöcke vor und nach der Veränderung durch LPC ausgewertet, was in Abbildung 12 zu sehen ist.



**Abbildung 12**

Die untere blaue Linie stellt die Entropie des Blockes an dem der Test durchgeführt wurde dar, welcher natürlich konstant blieb, da alle Koeffizienten am selben Block getestet wurden. Die obere grüne Linie zeigt die Entropie nach der Behandlung des Blockes mit LPC. Wie klar zu sehen ist bringt diese Methode sogar eine Verschlechterung der Entropie und ist daher kontraproduktiv.

Die Funktionen die hier verwendet wurden sind im Verzeichnis LPC1 auf der beiliegenden CD zu finden. Die Mex-Funktion des Arithmetischen Coders, „compress\_mex.c“ muss in Matlab zusammen mit den Files „bitio.c“ und „arith1e.c“ kompiliert werden, also wie folgt: „mex compress\_mex.c bitio.c arith1e.c“. Die Matlab Funktionen (M-Files) mit welchen die Tests durchgeführt wurden befinden sich ebenfalls in diesem Verzeichnis. Eingesetzt haben wir lediglich die Variante mit drei Koeffizienten, da die Variante mit 10 Koeffizienten unseren kleinen Hochrechnungen zu Folge über drei Tage benötigt hätte. Bei der dritten Funktion können Koeffizienten mit denen die Mex-Funktion arbeitet direkt von Hand festgelegt werden. Diese stammten in der Regel von Matlab wo man mit Hilfe des Befehls „lpc( Daten, Anzahl Koeffizienten)“ Koeffizienten direkt für eine Variable mit entsprechenden Testdaten bestimmen kann.

### 3.2.2 Variante2 „Koeffizienten aus Matlab“

Da das Finden der 3 Koeffizienten im ersten Versuch bereits 18 Stunden in Anspruch genommen hat war es aussichtslos, eine Schleife mit noch mehr Koeffizienten laufen zu lassen. Wir brauchten daher eine andere, schnellere Methode um Koeffizienten zu finden. Nahe liegend wäre natürlich gewesen eine Methode zu entwickeln, welche nicht einfach alle Varianten durchprobiert, sondern sich schrittweise durch Entscheidung ob besser oder schlechter der bestmöglichen Variante annähert. Dazu sind wir aber nicht mehr gekommen. Wir konnten zunächst mit Matlab Koeffizienten ermitteln, welche aufgrund eines Datensatzes berechnet wurden, welchen man Matlab übergab. Da mit diesen Koeffizienten auch bei den Daten für die sie eigentlich bestimmt wurden keine wesentliche Verkleinerung der Entropie erzielt wurde, haben wir darauf verzichtet einen relativ komplizierten Algorithmus zum Finden von Koeffizienten auszuprogrammieren. Versucht haben wir dieses Verfahren mit 4, 6, 8 und 10 von Matlab errechneten Koeffizienten, was bedeutet, dass die Werte aus den 4, 6, 8 oder 10 vorangegangenen Werten berechnet werden. Die nachfolgende Grafik zeigt die Entropie von 125 Blöcken eines aufgezeichneten Sprachsignals vor und nach der Behandlung mit LPC. Die obere Linie zeigt die Entropie der Eingangsblöcke und die untere jene der Blöcke die mit LPC daraus erstellt wurden. Bei den Werten die in der Grafik dargestellt sind wurde eine Variante mit sechs Koeffizienten benutzt, welche je auf vier Nachkommastellen genau waren. Hier zeigt sich eine leichte Verbesserung, das heisst Verkleinerung der Entropiewerte.

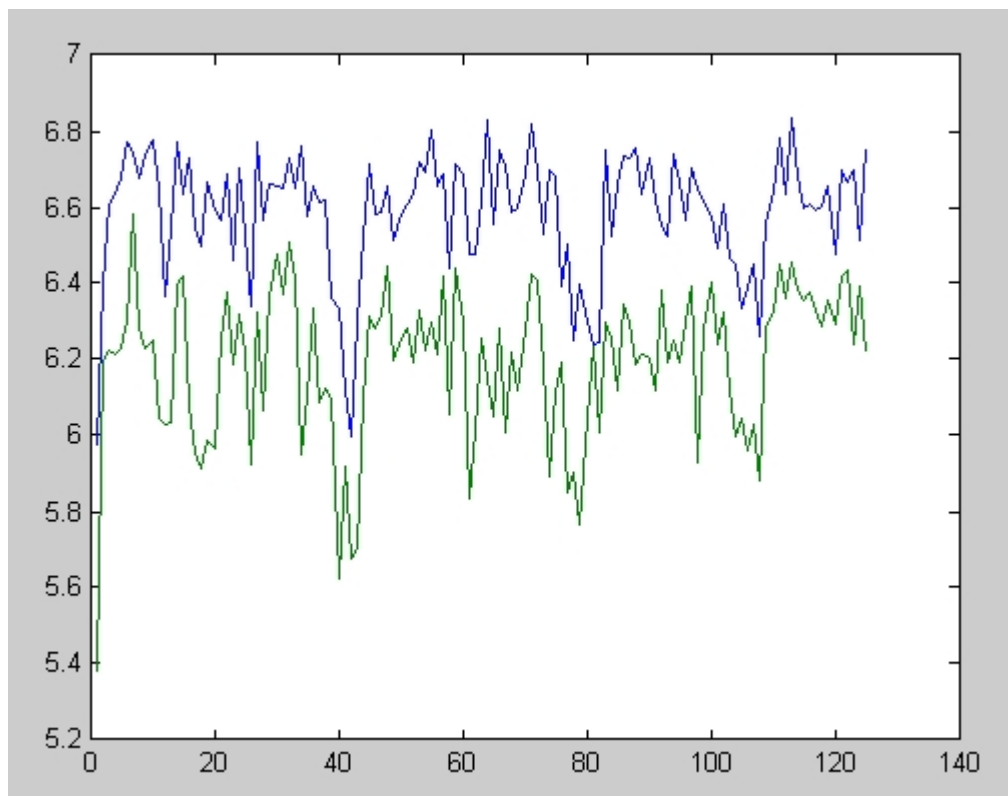


Abbildung 13

Bei einem Rauschen, welches in einem stillen Raum aufgezeichnet wurde, was Daten wie sie entstehen wenn eine Person am Telefon schweigt fingieren sollen, zeigen sich folgende Ergebnisse (Abbildung 14). Es wurden wie bei der Grafik oben 125 Blöcke verglichen mit dem gleichen Verfahren und den gleichen sechs Koeffizienten wie zuvor. Wie hier zu sehen ist, ist die Entropie dieses Rauschens interessanter sogar kleiner als diejenige des Sprachsignals. Dies ist darauf zurückzuführen, dass das Signal des Rauschens wesentlich leiser ist und dadurch die Datenwerte in einem viel engeren Bereich liegen.

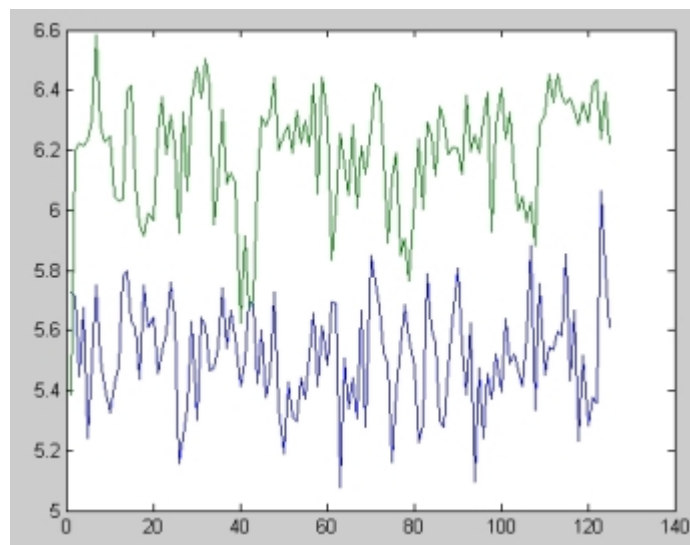


Abbildung 14

Zu finden sind die Funktionen die wir für diese Tests verwendet haben auf der CD im Verzeichnis LPC2. Wir haben diese LPC Variante wie bereits erwähnt, mit 4,6,8 und 10 Koeffizienten ausprogrammiert. Die zugehörige Umkehrversion zum entsprechenden Wandler ist jeweils mit der Namensverlängerung „back“ gekennzeichnet. So gehören z.B. die Funktionen „LPC4.cpp“ und die Umkehrfunktion „LPC4 back.cpp“ zusammen. Bei diesen Funktionen wurden jeweils die ersten 4,6,8 oder 10 (je nachdem wie viele Koeffizienten eingesetzt wurden) Zeichen direkt übermittelt, da diese für die Voraussage auch in der Umkehrfunktion benötigt wurden. Weiter befindet sich in diesem Verzeichnis auch noch die Funktion „newLPC10.cpp“ mit der zugehörigen Umkehrfunktion „newLPC10back.cpp“, bei welcher keine Zeichen direkt übertragen werden. Die fehlenden Zeichen für die Voraussage bei der Rückwandlung wurden durch feste Werte ersetzt, aus welchen jeweils die Voraussage der ersten Zeichen berechnet wurde. Dadurch fielen die direkt zu übermittelnden Zeichen weg, welche sich jeweils stark von den anderen Zeichen unterschieden haben, und so die Entropie stark beeinträchtigten. Leider brachte aber auch diese Änderung keine entscheidende Verbesserung.

Zusätzlich sind noch zwei Funktionen die für den Einsatz in Matlab als Mex-Funktion angepasst wurden in diesem Verzeichnis. Sie besitzen jeweils den Zusatz „mex\_“ am Anfang im Namen. Danach entspricht ihr Name den Namen ihrer ursprünglichen Funktion.

### 3.2.3 Variante 3 „Einfacher aber besser“

Leider erst sehr spät haben wir erkannt, dass wir mit einem sehr viel einfacheren Verfahren die Entropie einiges besser beeinflussen können. Auch dieses Verfahren basiert auf dem LPC Methode. Durch Bilden einer einfachen gewichteten Differenz zwischen vorangegangenen und momentanem Wert (also  $Differenz = Koeffizient * S[n] - S[n+1]$ ) kann die Entropie um einiges mehr gesenkt werden als mit den zuvor versuchten Verfahren. Die benötigten Koeffizienten konnten relativ schnell, wieder mit Hilfe einer Schleife gefunden werden. Versuche mit verschiedenen aufgezeichneten Daten haben gezeigt dass diese Koeffizienten im Bereich zwischen 0.8 und 1.1 liegen. Nachfolgend einige Darstellungen die dies gezeigt haben.

Beim erstellen der Kurve in Abbildung 15, wurde der Koeffizient von 0.01 bis 1.6 gefahren. Er wurde in 0.01 grossen Schritten erhöht, dabei wurde bei jedem Schritt 1 Block (immer der gleiche) behandelt und die Entropie ermittelt und aufgezeichnet. Wir sehen hier ein Minimum bei ca. 0.9. Bei der Kurve in Abbildung 16 wurde genau dasselbe gemacht, ausser das hier der Koeffizient von 0.6 auf 1.2 in 0.001 grossen Schritten erhöht wurde. Man kann sehen das eine höhere Auflösung beim Koeffizienten keine wesentlich Steigerung mehr bringt. Da hier bei 0.6 begonnen wurde, muss zum Wert auf der X-Achse noch 600 dazuaddiert werden damit man den Wert des Koeffizienten ablesen kann. Auch hier wieder sieht man ein Minimum bei ca. 0.9 (was auf der X-Achse dem Wert 300 entspricht).

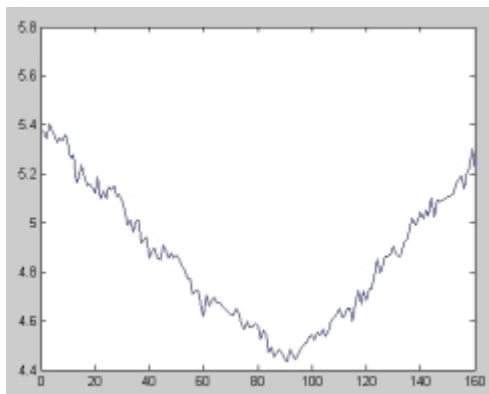


Abbildung 15

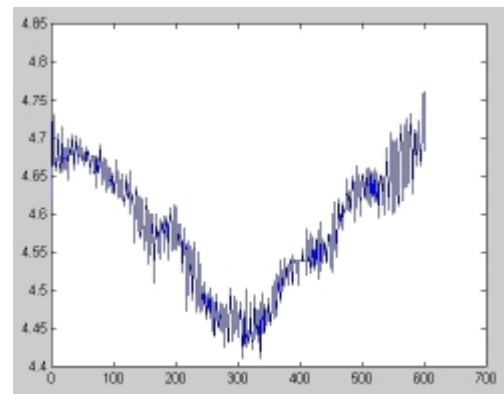


Abbildung 16

Was für einen einzelnen Block funktioniert muss natürlich nicht für alle Blöcke gut sein. Daher haben wir in den nachfolgenden Grafiken noch Durchschnittswerte aus mehreren Blöcken dargestellt. Zuerst Abbildung 17, bei welcher der Durchschnitt der Entropie über 125 Blöcke eines aufgezeichneten Sprachsignals dargestellt ist. Der Koeffizient wurde hier von 0.6 bis 1.3 in 0.001 grossen Schritten erhöht und jeweils mit jedem Koeffizient alle 125 Blöcke behandelt. Hier sehen wir dass sich das Minimum im Vergleich zu oben leicht von 0.9 in Richtung 1.0 verschoben hat. In Abbildung 18 wurde der Koeffizient in 0.01 grossen Schritten von 0 auf 1.5 erhöht. Mit jedem Wert wurden 400 Blöcke eines aufgezeichneten Telefongesprächs, in welchem auch Stillen vorkommen, behandelt. Auch hier liegt das Minimum zwischen 0.9 und 1.0 ist aber dank den stillen Abschnitten noch etwas tiefer als beim reinen Sprachsignal.

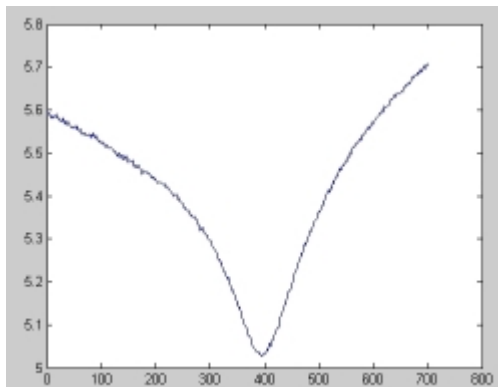


Abbildung 17

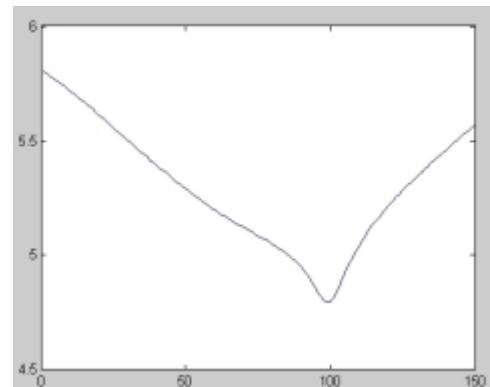


Abbildung 18

Um sicher zu gehen dass diese Resultate nicht zufällig nur für die von uns gesprochene Aufzeichnung zutreffen wurden noch zwei Kurven erstellt, welche Durchschnittswerte über je 9000 Blöcke von zwei von Nachrichtensendern aufgezeichneten Sprachsignalen zeigen. Der Koeffizient wurde hierfür von in Schritten von 0.001 von 0.6 auf 1.3 erhöht. Auch hier ist das Minimum wieder klar zu erkennen und liegt bei ca. 1.0. Es lässt sich also daraus schliessen, dass sich mit dieser Methode mit einem Koeffizienten zwischen 0.9 und 1.1 für nahezu alle Datenblöcke eine mehr oder weniger deutliche Senkung der Entropie herbeiführen lässt und die Datenblöcke so für den arithmetischen Coder optimiert werden können.

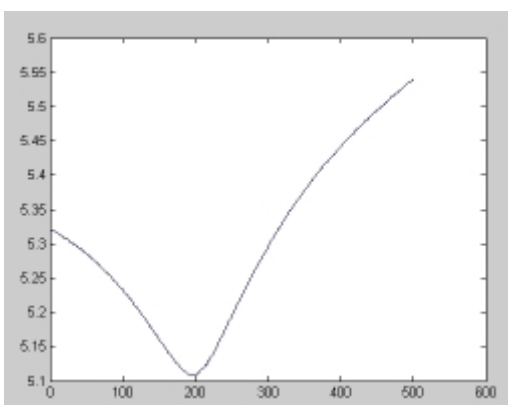


Abbildung 20

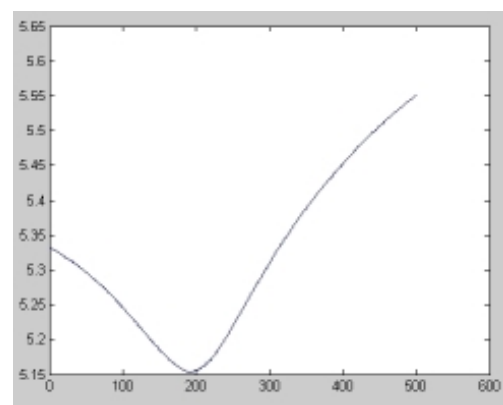


Abbildung 19

Um die Differenz der Entropiewerte darstellen zu können haben wir zwei Grafiken aufgezeichnet, in welchen der Entropiewert von jedem Block vor der Änderung durch LPC und danach verglichen wird (Abbildung 22). Es wird hier konstant mit einem Koeffizienten von 0.95 gerechnet. Es sollte daher durch Einsatz von mehreren Varianten mit verschiedenen Koeffizienten im Kompressor möglich sein, das Ergebnis noch zu optimieren. Es lässt sich aber erkennen, dass auch bei einem konstanten Koeffizienten bei allen Blöcken eine Verbesserung erreicht wurde. Die Grafik links zeigt den Vergleich von 125 Blöcken eines Sprachsignals, die Grafik rechts den eines Rauschens. Interessant ist auch hier wieder, dass die Entropie des Rauschsignals schon vor LPC kleiner ist als diejenige des Sprachsignals, was aber wieder auf die Lautstärke des Signals zurückzuführen ist. Der Vergleich mit Abbildung 13 und Abbildung 14 zeigt, dass die Entropie mit dieser Methode einiges grösser ist als mit der vorherigen Methode.

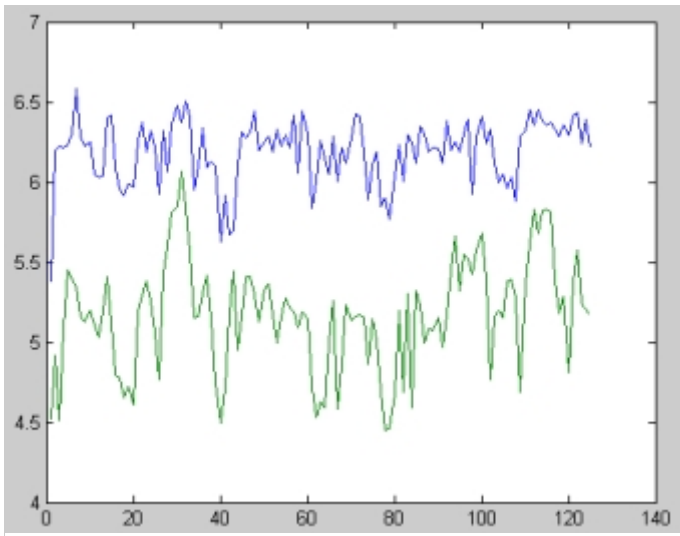


Abbildung 22

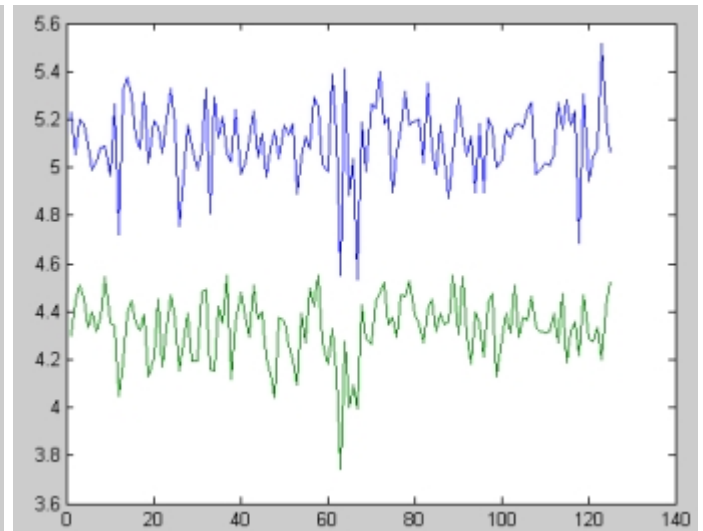


Abbildung 21

Weiter haben wir versucht ob es möglich ist bessere Resultate zu erzielen wenn man anstelle von nur einem mit zwei Koeffizienten arbeitet. Wie man in Abbildung 23 erkennen kann sind die Schwankungen in der Entropie, sie wird von der blauen Linie dargestellt welche nur über einen kleinen Bereich variiert, relativ klein. Sie werden erst stärker, wenn der erste Koeffizient, er wird von der grünen Treppe von unten nach oben dargestellt, grösser als 1 ist. Die Tiefpunkte der Entropie sind immer an den Stellen, an denen der zweite Koeffizient, er wird von der roten stark auf und ab springenden Linie dargestellt, kleiner als Null ist. Deshalb haben wir in Abbildung 24 noch einmal den gleichen Vorgang, allerdings mit anderen Bereichen für die Koeffizienten. Hier zeigt sich, dass die Entropiewerte nie unter fünf fallen und daher ist das Verfahren weniger effizient als das Verfahren mit einem Koeffizienten und wurde deshalb weggelassen.

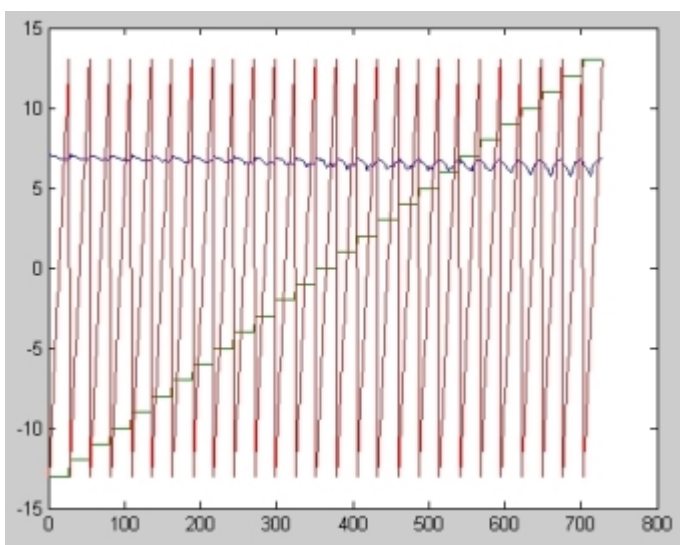


Abbildung 23

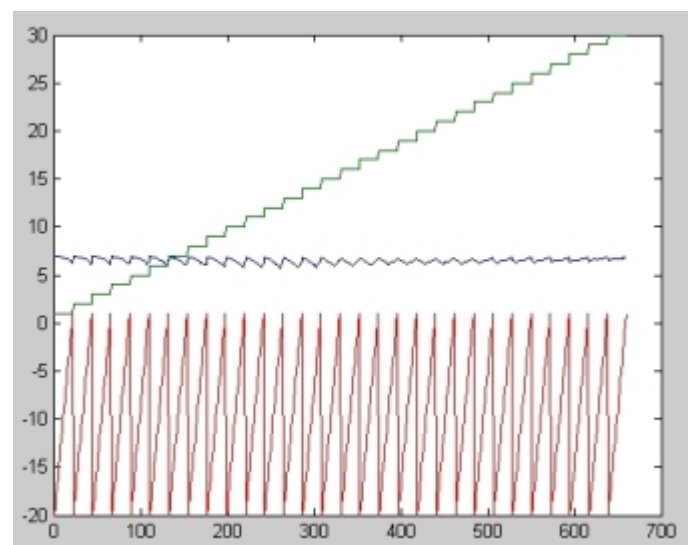


Abbildung 24

Alle Files die bei dieser Variante verwendet wurden, sind im Verzeichnis LPC3 zu finden. Im Kompressor eingesetzt wird die Variante die im Filenamen die Erweiterung 2 hat. Bei der ersten Variante, welche mit der Rechnung „ $Differenz = S[n] - Koeffizient * S[n+1]$ “ arbeitet hat sich herausgestellt, dass sie sich bei einem Sprung welcher über die 0-Grenze der Char-Variable hinausgeht nicht wieder rückgängig machen lässt, daher wurde sie aufgegeben. Die benutzte Variante in dem der erste Wert gewichtet wird anstelle des zweiten erzielt aber beinahe exakt dieselben Ergebnisse, dies einfach jeweils mit etwas anderen Koeffizienten. Die enthaltenen M-Files dienen dazu die oben dargestellten Grafiken zu erstellen, und damit dazu die optimalen Koeffizienten zu finden.

### 3.3 Struktur des Kompressors

Die ganze Struktur des aus vielen Komponenten bestehenden Kompressors soll im folgenden Teil genauer beschrieben werden. Auch sind Erläuterungen aller im Kompressor verwendeten Funktionen sowie Methoden zu finden, und auch die Funktionsnamen im C-Code sind angegeben.

#### 3.3.1 Grundidee

Die Effizienz eines Coders kann stark verbessert werden wenn die Daten vor einem arithmetischen oder einem Huffman-Coder bearbeitet werden. Eine Möglichkeit ist die einfache Differenzbildung oder die Voraussage mittels LPC. Um die verschiedenen Zusammensetzungen von Algorithmen zur Optimierung der Daten und Codierungsverfahren zu testen werden sinnvoll erscheinende Kombinationen erstellt und anschliessend mittels Testdaten ausgewertet.

In Tabelle 1 sind die getesteten Kombinationen aufgelistet. Die verwendeten Abkürzungen sind in Abschnitt XYZ erläutert.

Um aussagekräftige Resultate zu erhalten dienen als Testdaten vorher aufgezeichnete Gespräche. Alle Kombinationen werden von jedem 160 Byte grossen Datenblock einmal durchlaufen. Da so relativ grosse Datenmengen ausgewertet werden müssen wird nach jedem Durchlauf gespeichert welches Codec den kürzesten Code erzeugt hat. Dies geschieht indem am Ende jedes Codes ein Byte mit der Codec-Nr. angefügt wird.

Der Kürzeste Datenblock wird gespeichert und die restlichen verworfen. In einem Matlab-Array, in dem jede Position einer Codec-Nr. entspricht wird darauf der betreffende Wert um eins erhöht. Falls mit keinem Codec die Datenmenge reduziert werden kann werden die 160 Byte unkomprimiert übergeben. In diesem Fall wird auch kein Byte am Ende angefügt. Das unkomprimierte Frame wird alleine an seiner Grösse erkannt.

Zur Kontrolle ob die wieder decodierten G.711-Werte auch Bit für Bit den ursprünglichen G.711-Werten entsprechen, wird die Variable „fehler“ bei jedem abweichenden Byte um eins erhöht.

Die dekomprimierten Datenblöcke werden am Ende wieder zusammengefügt und werden abgespielt.

## Matlab-Code

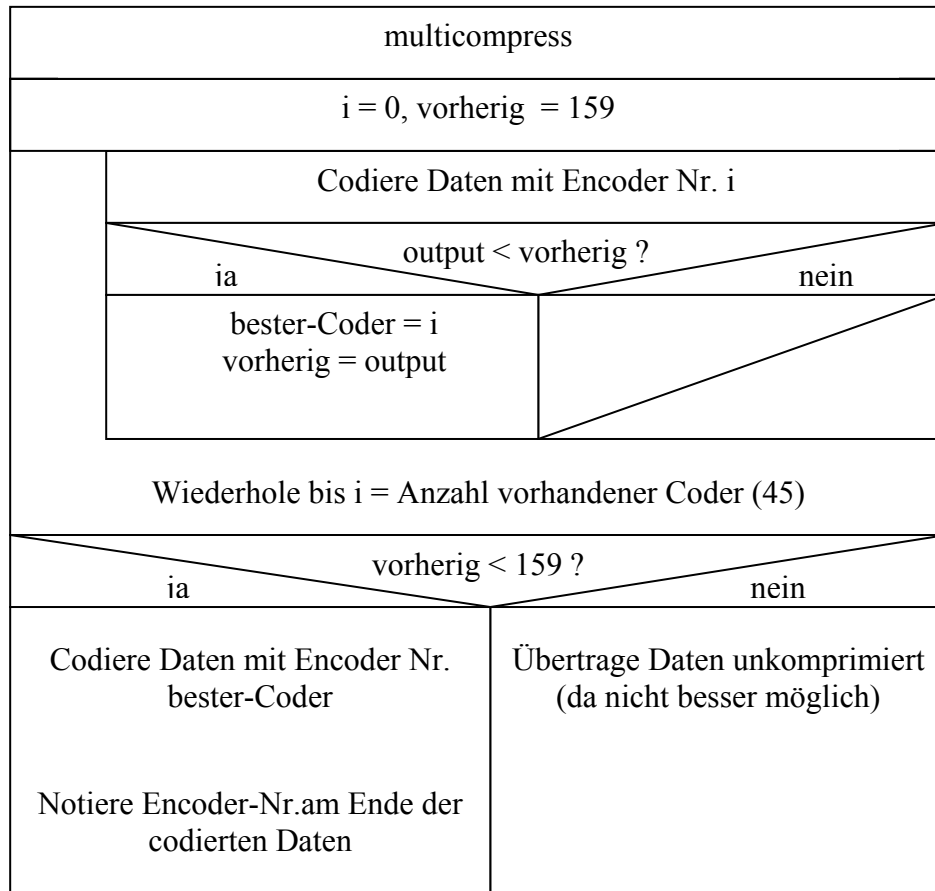
```
“Sizeofl.m”

y = wavrecord(10*8000,8000,'int16');
groesse = size(x);
unkomprimierte_groesse = groesse(1)*groesse(2)
fehler =0;
codecs=double(zeros(1,46));
komprimierte_groesse = double(0);
for n = 1:groesse(1)
    x1 = int16(x(n, 1:160));           % Ein Frame aus den gesamten Daten nehmen
    x2 = pcm_ulaw_mex(x1);           % 16bit-PCM in ulaw wandeln
    x3 = ulaw_ord_mex(x2);          % ulaw in geordnete Werte wandeln
    z = multicompress_mex(x3);      % Frame komprimieren
    if(length(z) ~= 160)
        codecs(z(length(z))+1)=codecs(z(length(z))+1)+1;
    else
        codecs(1)=codecs(1)+1;
    end
    x3new = multiexpand_mex(z);      % Frame dekomprimieren
    x2new = ord_ulaw_mex(x3new);     % geordnete Werte in ulaw wandeln
    x1new = ulaw_pcm_mex(x2new);    % wandelt ulaw wieder in 16bit PCM
    ergebnis(1, (n*160-159):(n*160))=x3new; % 8Bit Frames zusammensetzen
    ergebnis_pcm(1, (n*160-159):(n*160))=x1new; % 16Bit Frames zusammensetzen
    fehler = fehler + (160 -sum(x2==x2new)); % Fehler zusammenzählen!
    komprimierte_groesse = komprimierte_groesse + uint32(length(z));
end
fehler
komprimierte_groesse
komprimierung = double(komprimierte_groesse) / double(unkomprimierte_groesse)
stem(codecs)
soundsc(double(ergebnis_pcm),8000)
```

Die Funktionen die innerhalb dieses M-Files zur Kompression aufgerufen werden:

C-Funktionsname: multicompress

Matlab-Interface: multicompress\_mex.c



**Abbildung 25: Ablaufdiagramm der Funktion „multicompress“**

Dekompression:

C-Funktionsname: multiexpand

Matlab-Interface: multiexpand\_mex.c

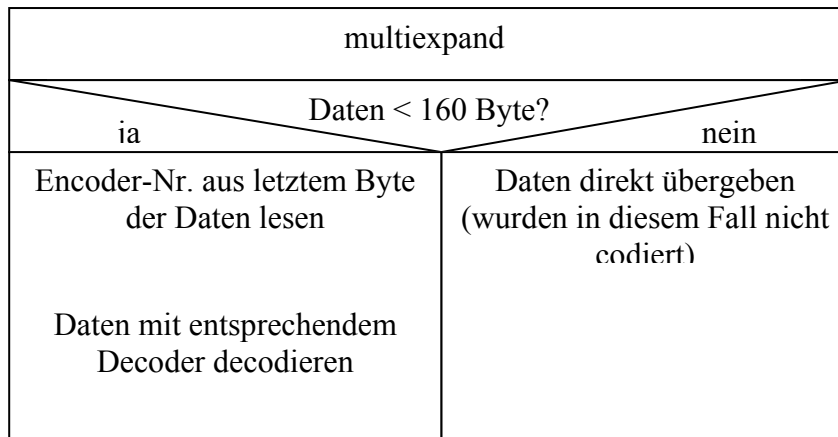


Abbildung 26: Ablaufdiagramm des Decoders

### 3.3.2 Codecs

Die Tabelle zeigt eine Auflistung aller Codiermethoden die im Kompressor implementiert sind inklusive der zu jeder Methode gehörenden Codec-Nummer.

**Tabelle 1**

| Codec-Nr. | Beschreibung                       |
|-----------|------------------------------------|
| 0         | direkt                             |
| 1         | arith0                             |
| 2         | arith1e                            |
| 3         | absolute-> arith0                  |
| 4         | absolute-> arith1e                 |
| 5         | differenz -> arith0                |
| 6         | differenz -> arith1e               |
| 7         | four2eight -> arith0               |
| 8         | four2eight -> arith1e              |
| 9         | two2eight -> arith0                |
| 10        | two2eight -> arith1e               |
| 11        | eight2seven -> arith0              |
| 12        | eight2seven -> arith1e             |
| 13        | eight2seven                        |
| 14        | differenz -> eight2seven           |
| 15        | eight2six                          |
| 16        | differenz -> eight2six             |
| 17        | differenz -> four2eight -> arith0  |
| 18        | differenz -> four2eight -> arith1e |
| 19        | differenz -> two2eight -> arith0   |
| 20        | differenz -> two2eight -> arith1e  |
| 21        | eight2five                         |
| 22        | differenz -> eight2five            |
| 23        | ahuff                              |
| 24        | differenz -> ahuff                 |
| 25        | LPC_0.90 -> ahuff                  |
| 26        | LPC_0.95 -> ahuff                  |
| 27        | LPC_1.00 -> ahuff                  |
| 28        | LPC_1.05 -> ahuff                  |
| 29        | LPC_1.10 -> ahuff                  |
| 30        | LPC_0.90 -> arith0                 |
| 31        | LPC_0.90 -> arith1e                |
| 32        | LPC_0.95 -> arith0                 |
| 33        | LPC_0.95 -> arith1e                |
| 34        | LPC_1.00 -> arith0                 |
| 35        | LPC_1.00 -> arith1e                |
| 36        | LPC_1.05 -> arith0                 |
| 37        | LPC_1.05 -> arith1e                |
| 38        | LPC_1.10 -> arith0                 |
| 39        | LPC_1.10 -> arith1e                |
| 40        | LPC_0.90 -> eight2six              |
| 41        | LPC_0.95 -> eight2six              |
| 42        | LPC_1.00 -> eight2six              |
| 43        | LPC_1.05 -> eight2six              |
| 44        | LPC_1.10 -> eight2six              |
| 45        | differenz -> eight2six -> ahuff    |

### 3.3.3 Flussdiagramm des Encoders

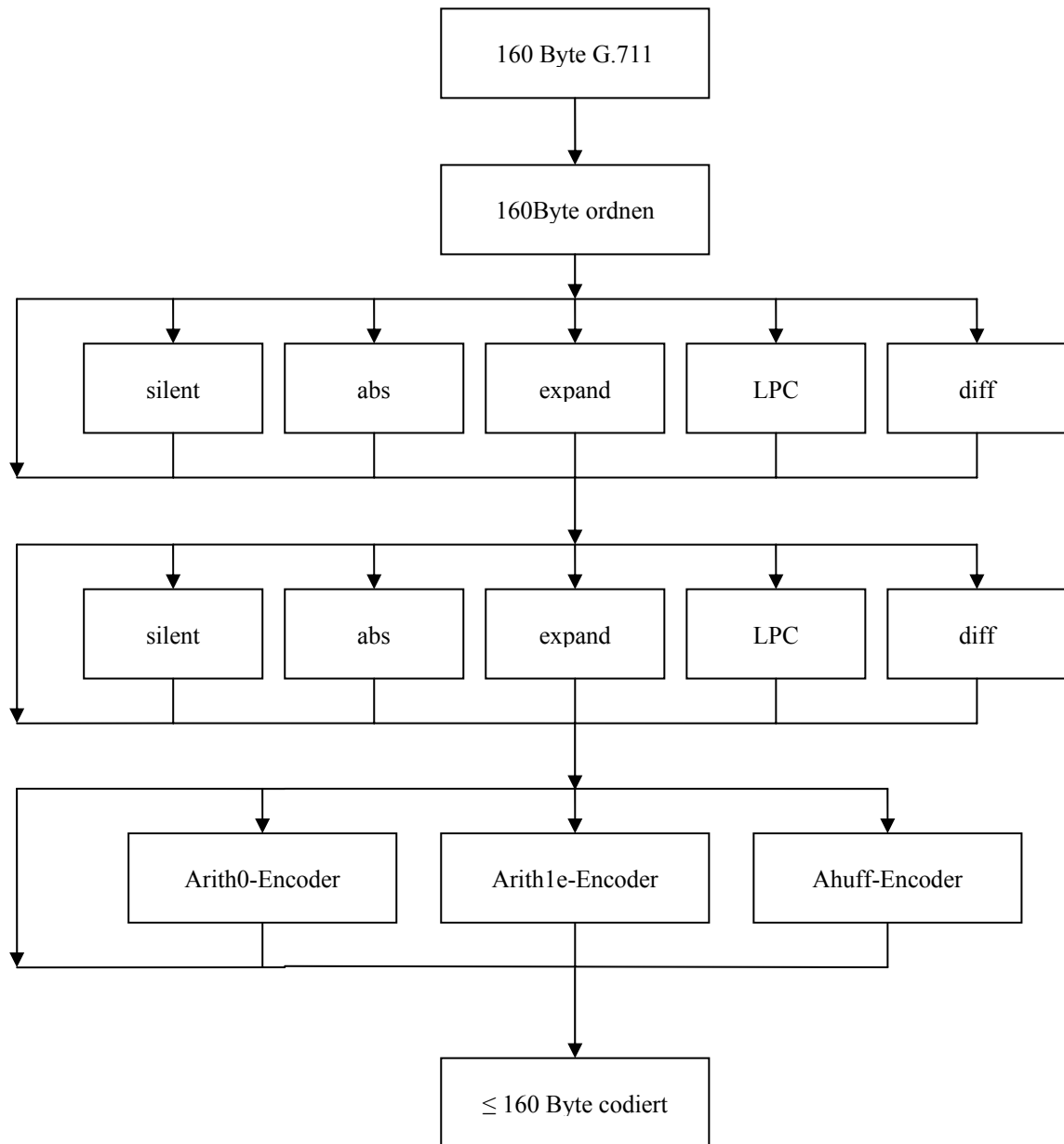


Abbildung 27: Verschiedene Möglichkeiten einen Encoder zusammenzustellen

### 3.3.4 Flussdiagramm des Decoders

Decoder

C-Funktionsname: multiexpand

Matlab-Interface: multiexpand\_mex.c

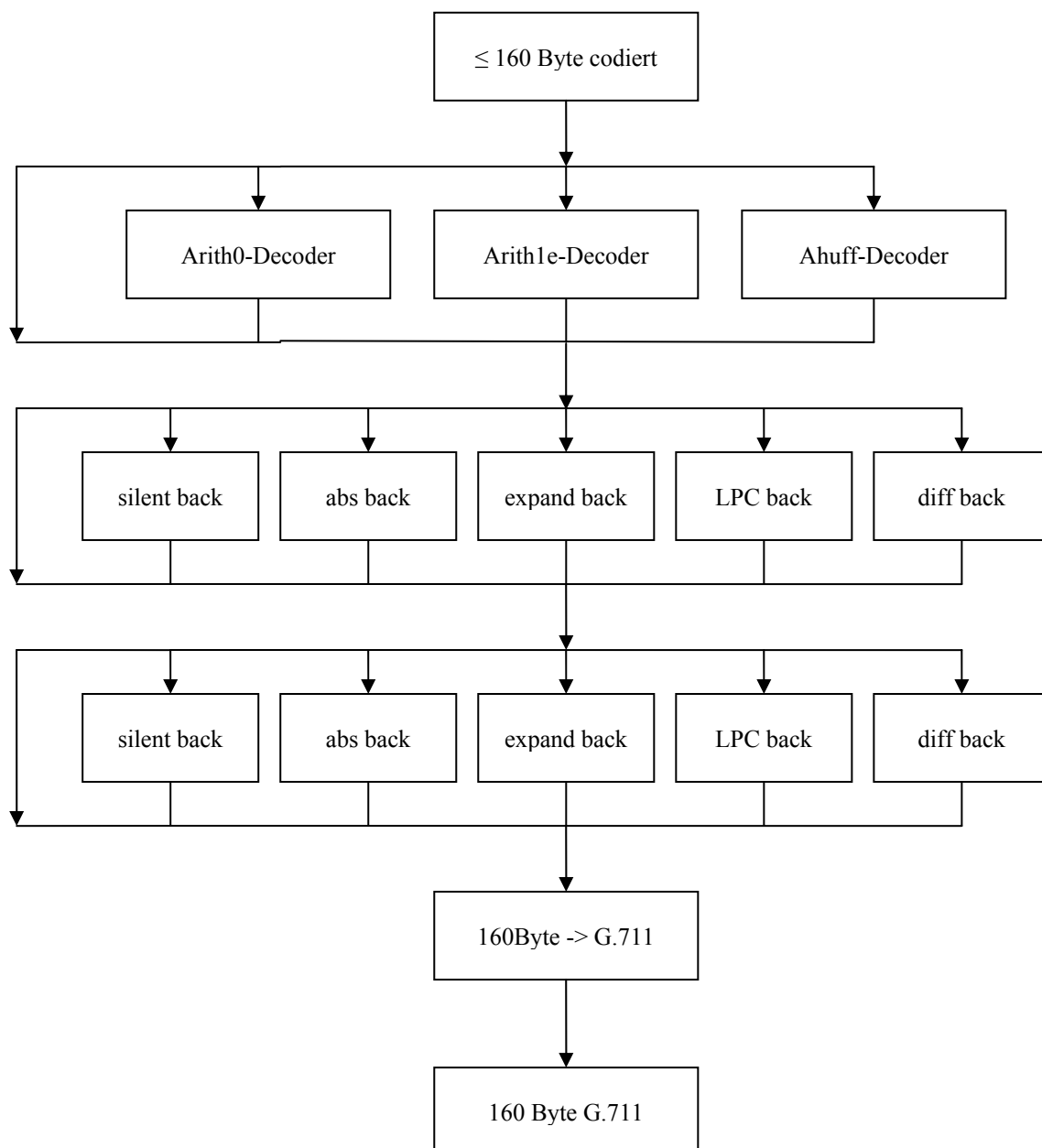


Abbildung 28: Verschiedene Möglichkeiten den zum Encoder passenden Decoder zusammenzustellen

### 3.3.5 PCM und G.711

Unsere Datenblöcke werden bevor sie vom eigentlichen Kompressor komprimiert werden auf G.711 Qualität, was der Qualität des ISDN Telefons entspricht, komprimiert. Wir passen die Daten von G.711 zusätzlich noch an, indem wir sie ordnen. Unsere G.711 Datenblöcke sind also nicht mehr mit dem ITU-T Standard kompatibel, werden aber im Dekompressor wieder in solche zurückgewandelt.

#### 3.3.5.1 PCM → G.711

*C-Funktionsname: linear\_ulaw*

*Matlab-Interface: pcm\_ulaw\_mex.c*

Abbildung 29 zeigt 20ms des ursprünglichen Tones als 16Bit „signed integer“-Werte wie sie von einer Soundkarte kommen. Mittels der Funktion „linear2ulaw“ aus der Datei g711.c wandelt diese Funktion 160 „short“-Werte welche den selben Wertebereich wie 2Byte-„integer“-Werte aufweisen in 8Bit-„unsigned char“-Werte um. Die dabei entstandenen Werte entsprechen den G.711-ulaw-Codeworten, die in Abbildung 30 zu sehen sind.

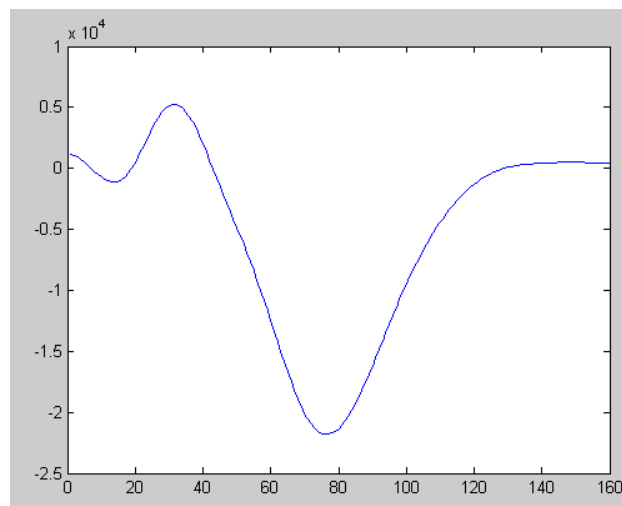


Abbildung 29 : 20ms Ton (16Bit signed integer)

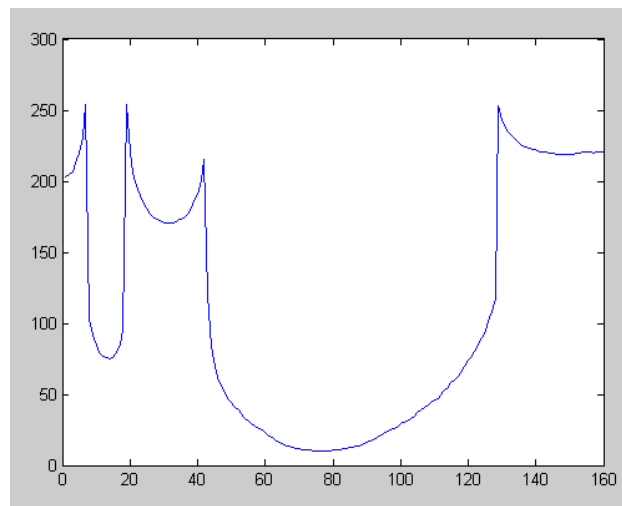


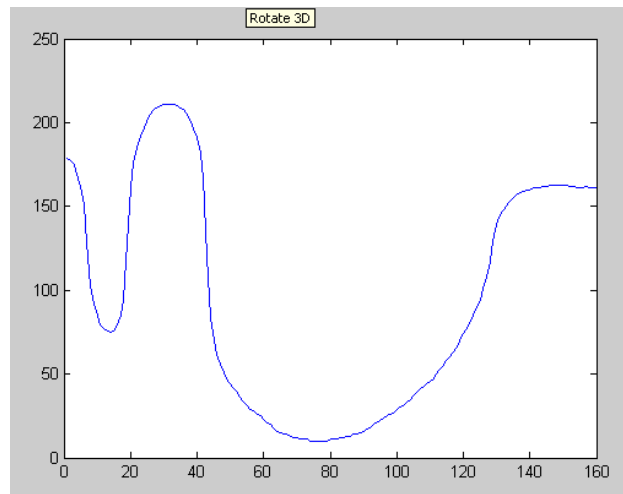
Abbildung 30: 20ms G.711-ulaw-Code

### 3.3.5.2 G.711 → G.711 geordnet

*C-Funktionsname:* `ulaw_ord`

*Matlab-Interface:* `ulaw_ord_mex.c`

Diese Funktion ordnet den 160 ulaw-Codeworten Werte zu, die der ursprünglichen Form des Signals entsprechen. Die Verzerrung welche Abbildung 31 im Vergleich zu Abbildung 29 zeigt, entsteht durch die G.711-Kompression.



**Abbildung 31: 20ms geordnete ulaw-Werte**

160Byte -> G.711:

*C-Funktionsname:* `ord_ulaw`

*Matlab-Interface:* `ord_ulaw_mex.c`

Hiermit werden die geordneten 8Bit-Werte wieder in G.711-ulaw-Codes umgerechnet.

*C-Funktionsname:* `ulaw_pcm`

*Matlab-Interface:* `ulaw_pcm_mex.c`

Mittels der Funktion „`ulaw2linear`“ aus der Datei `g711.c` werden den ulaw-Codes entsprechende 16Bit-PCM-Werte zugeordnet.

### 3.3.6 Codierungsalgorithmen

Als Vorlage für den statischen arithmetischen Coder, den adaptiven arithmetischen Coder erster Ordnung und den adaptiven Huffman Coder dienen die Quellcodes aus „The Data Compression Book“.

#### 3.3.6.1 Statischer arithmetischer Coder Arith0

Diese Funktionen sind in der Datei arith.c zu finden. Dazu gehört die weitere C-Datei bitio.c. Arith0 enthält den Code für den statischen arithmetischen Encoder und Decoder. „Statisch“ ist in diesem Zusammenhang so zu verstehen dass für den gesamten Datenblock die selbe Häufigkeitstabelle zur Codierung benutzt wird. Die Tabelle ändert sich erst mit einem neuen Datenblock.

Encoder:

*C-Funktionsname: compress\_string\_0*

Diese Funktion zählt als erstes die Häufigkeit jedes vorkommenden Zeichens und speichert diese Daten am Anfang des Ausgangsdatenblockes in komprimierter Form. Danach beginnt die eigentliche Codierung der Daten.

Decoder:

*C-Funktionsname: expand\_string\_0*

Als erstes werden die Häufigkeiten der einzelnen Zeichen aus den Eingangsdaten ausgelesen. Nun ist es möglich die Daten zu decodieren.

#### 3.3.6.2 Adaptiver arithmetischer Coder Arith1e

Diese Funktionen sind in der Datei arith1e.c zu finden. Dazu gehört die weitere C-Datei bitio.c. Arith1e enthält den Code für den adaptiven arithmetischen Encoder und Decoder.

Encoder:

*C-Funktionsname: compress\_string*

Der adaptive arithmetische Encoder passt sein Wahrscheinlichkeitstabelle zur Codierung der Daten mit jedem zusätzlich codierten Byte an. Als erstes werden zur Initialisierung des Decoders zwei Byte in den Ausgangsdatenblock geschrieben. Es ist nicht notwendig eine Häufigkeitstabelle an den Anfang der Ausgangsdaten zu schreiben.

Decoder:

*C-Funktionsname: expand\_string*

Die zwei ersten Bytes in den Eingangsdaten dienen zur Initialisierung von drei Variablen die der Decoder benötigt um mit der Decodierung zu beginnen.

### 3.3.6.3 Adaptiver Huffman Coder Ahuff

Diese Funktionen sind in der Datei ahuff.c zu finden. Dazu gehört die weitere C-Datei bitio.c. Ahuff enthält den Code für den adaptiven Huffman-Encoder und -Decoder.

Encoder:

*C-Funktionsname: compress\_string\_h*

Der adaptive Huffman-Encoder passt sein Wahrscheinlichkeitsbaum zur Codierung der Daten mit jedem zusätzlich codierten Byte an. Auch hier ist es nicht notwendig die Baumstruktur mit den Daten mitzugeben.

Decoder:

*C-Funktionsname: expand\_string\_h*

Die Funktion expand\_string\_h decodiert die Daten die mit compress\_string\_h codiert wurden. Byte für Byte passt auch der Decoder sein Wahrscheinlichkeitsbaum an um die weiteren Daten zu decodieren.

### 3.3.7 Funktionsbeschreibungen

Im folgenden Teil sind alle Funktionen beschrieben, aus welchen die 45 verschiedenen Methoden des Kompressors zusammengesetzt werden.

#### 3.3.7.1 Methoden bei denen Vorzeichenlose Werte bearbeitet werden

*C-Funktionsname: absolute*

Diese Funktion trennt die Vorzeichen der Daten von den Werten. Hierzu wird von den „unsigned char“-Werten 128 subtrahiert und bei den negativen Werten die daraus entstehen, das Vorzeichen weggelassen. Die Werte sind nun noch maximal 7bit gross, sind aber trotzdem noch in 8 Bit grossen „unsigned char“-Werten verpackt.

|           |                        |             |             |            |             |
|-----------|------------------------|-------------|-------------|------------|-------------|
| Byte<br>0 | Absolute Werte (7 Bit) | Byte<br>159 | Byte<br>160 | Vorzeichen | Byte<br>179 |
|-----------|------------------------|-------------|-------------|------------|-------------|

**Abbildung 32**

Wie Abbildung xy zeigt, werden die 160 Bit Vorzeichen in 20 Byte am Ende des Blockes angehängt und können so getrennt bearbeitet werden.

*C-Funktionsname: absback*

In dieser Funktion wird die Arbeit von von „absolute“ umgekehrt. D.h. die 160 7 Bit grossen Werte werden wieder mit Vorzeichen versehen und in „unsigned char“-Werte gepackt.

#### 3.3.7.2 Methoden bei denen mit Differenzwerten gearbeitet wird

*C-Funktionsname: differenz*

Statt den Werten selbst werden die Differenzen zum vorherigen Byte gespeichert. Das erste Byte wird eins zu eins übernommen. In vielen Fällen kann so der Wertebereich eines 20ms langen Frames auf 7, 6 oder selten sogar 5 Bit reduziert werden.

*C-Funktionsname: diffBack*

Sie dient der Umkehrung der Funktion „differenz“. Das erste Byte wird direkt übernommen und zu den restlichen 159 Byte wird jeweils die Differenz addiert so dass wieder die ursprünglichen Daten bereit stehen.

### 3.3.7.3 Methoden bei denen die Bytes in Teile zerlegt werden

*C-Funktionsname: four2eight*

Aus einem Byte werden jeweils die ersten und die letzten vier Bit separiert und in „unsigned char“-Werten gespeichert. Die Bytezahl eines Frames verdoppelt sich so.

|           |             |             |             |             |             |
|-----------|-------------|-------------|-------------|-------------|-------------|
| Byte<br>0 | 160 x 4 Bit | Byte<br>159 | Byte<br>160 | 160 x 4 Bit | Byte<br>319 |
|-----------|-------------|-------------|-------------|-------------|-------------|

Abbildung 33

*C-Funktionsname: eight2four*

Dies ist die Umkehrfunktion von „four2eight“. Damit werden aus 320 4Bit-Werten wieder 160 8Bit-Werte geformt.

*C-Funktionsname: two2eight*

Aus einem Byte werden jeweils vier mal zwei Bit separiert und in „unsigned char“-Werten gespeichert. Die Bytezahl eines Frames vervierfacht sich so.

|           |                |             |             |                |             |             |                |             |             |                |             |
|-----------|----------------|-------------|-------------|----------------|-------------|-------------|----------------|-------------|-------------|----------------|-------------|
| Byte<br>0 | 160 x<br>2 Bit | Byte<br>159 | Byte<br>160 | 160 x<br>2 Bit | Byte<br>319 | Byte<br>320 | 160 x<br>2 Bit | Byte<br>479 | Byte<br>480 | 160 x<br>2 Bit | Byte<br>639 |
|-----------|----------------|-------------|-------------|----------------|-------------|-------------|----------------|-------------|-------------|----------------|-------------|

Abbildung 34

*C-Funktionsname: eight2two*

Dies ist die Umkehrfunktion von „two2eight“. Damit werden aus 640 2Bit-Werten wieder 160 8Bit-Werte geformt.

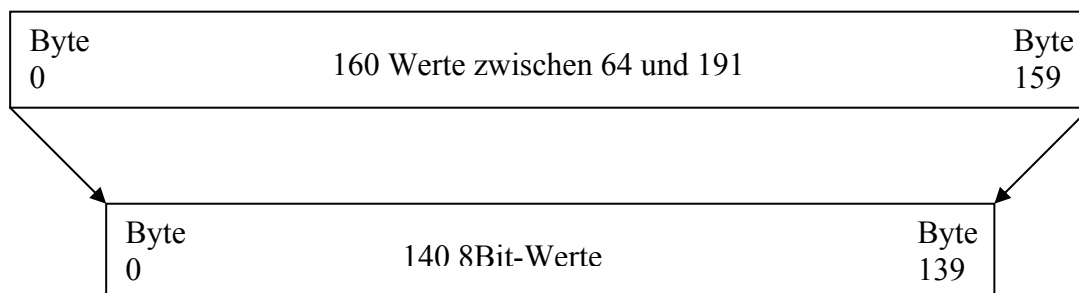
### 3.3.7.4 Methoden bei denen kleine Lautstärken ausgenutzt werden

Diese Funktionen dienen dazu leise Frames oder Frames die zuvor mittels der diff-Funktion bearbeitet wurden in Blöcke mit weniger Bytes zu komprimieren. Dazu wird zuerst überprüft ob alle Werte im entsprechenden Frame mit einer geringeren Anzahl Bit dargestellt werden kann.

*C-Funktionsname: test7*

Testet ob alle Werte zwischen 64 und 191 liegen. Falls dies der Fall ist kann das gesamte Frame mittels „eight2seven“ in 140 Byte abgespeichert werden.

*C-Funktionsname: eight2seven*



**Abbildung 35**

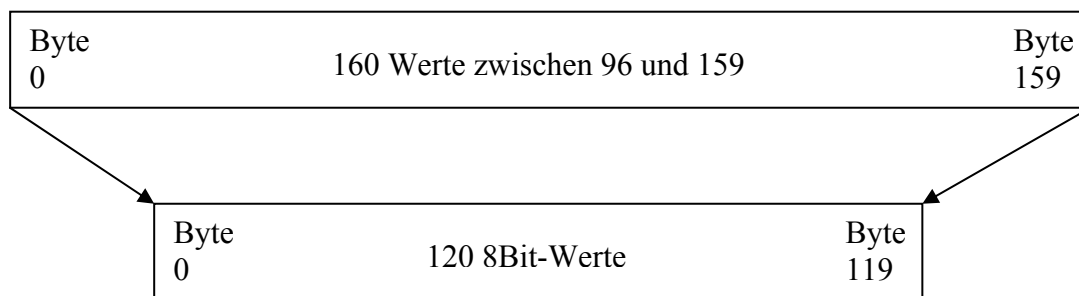
*C-Funktionsname: test6*

Testet ob alle Werte zwischen 96 und 159 liegen. Falls dies der Fall ist kann das gesamte Frame mittels „eight2six“ in 120 Byte abgespeichert werden.

*C-Funktionsname: six2eight*

Damit werden die 120 „unsigned char“-Werte wieder in die ursprünglichen 160 Byte aufgeteilt.

*C-Funktionsname: eight2six*

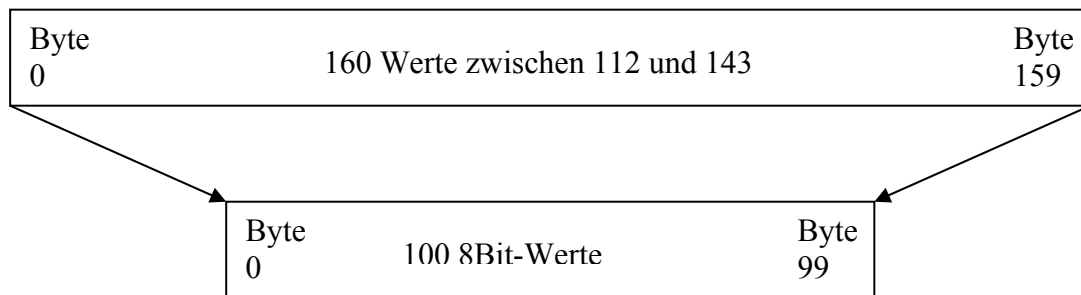


**Abbildung 36**

*C-Funktionsname: test5*

Testet ob alle Werte zwischen 112 und 143 liegen. Falls dies der Fall ist kann das gesamte Frame mittels „eight2five“ in 100 Byte abgespeichert werden.

*C-Funktionsname: eight2five*



**Abbildung 37**

*C-Funktionsname: five2eight*

Damit werden die 100 „unsigned char“-Werte wieder in die ursprünglichen 160 Byte aufgeteilt.

## 4 Testresultate

Die folgenden Abbildungen stellen die Resultate von drei verschiedenen 10s langen Testaufnahmen dar. Den Amplitudengängen in Abbildung 38, Abbildung 40 und Abbildung 42 folgen jeweils die Darstellungen der Arrays mit den verwendeten Codecs in Abbildung 39, Abbildung 41 und Abbildung 43. Dabei ist zu beachten dass die Werte auf der X-Achse jeweils um den Wert 1 höher sind als die entsprechende Codec-Nr. Welche Codec-Nr. welcher Kombination von Coder und Datenvorbereitung entspricht ist in Tabelle 1 aufgelistet.

### 4.1 Kurzes Gespräch mit anschliessender Pause

Abbildung 38 zeigt ca. 4s Gespräch mit anschliessender Pause. Die gesamte Aufnahmedauer beträgt exakt 10s. Die Datenmenge konnte auf 42.57% der ursprünglichen Grösse komprimiert werden.

In Abbildung 39 kann man erkennen dass hauptsächlich Codec Nr. 24 zum Erfolg führte, also die Kombination aus einer einfachen Differenzbildung mit anschliessender adaptiven Huffman-Codierung.

### 4.2 Pausenloses Gespräch

In Abbildung 40 sind 10s Gespräch ohne Unterbruch dargestellt. Hier konnten die Daten weniger stark komprimiert werden. Die gesamte Datenmenge beträgt noch 61.70% der unkomprimierten ulaw-Daten.

Wie Abbildung 41 zeigt, wurde auch hier hauptsächlich wieder Codec Nr. 24 verwendet. Allerdings leisten auch die Kombinationen aus Differenz und arith0-Coder (Codec Nr. 5) und Differenz und arith1e-Coder (Codec Nr. 6) einen wesentlichen Beitrag.

### 4.3 Stille

Der dritte Test komprimiert 10s Stille. Das leise Rauschen das dabei aufgenommen wurde zeigt Abbildung 42. Die Datenmenge konnte in diesem Fall auf 23.16% reduziert werden.

Die Codecs welche die beste Komprimierung ermöglichten waren vor allem Nr. 1, Nr. 2 und Nr. 5. Dies sind der statische arithmetische Coder, der adaptive arithmetische Coder und der statische arithmetische Coder mit Differenzbildung.

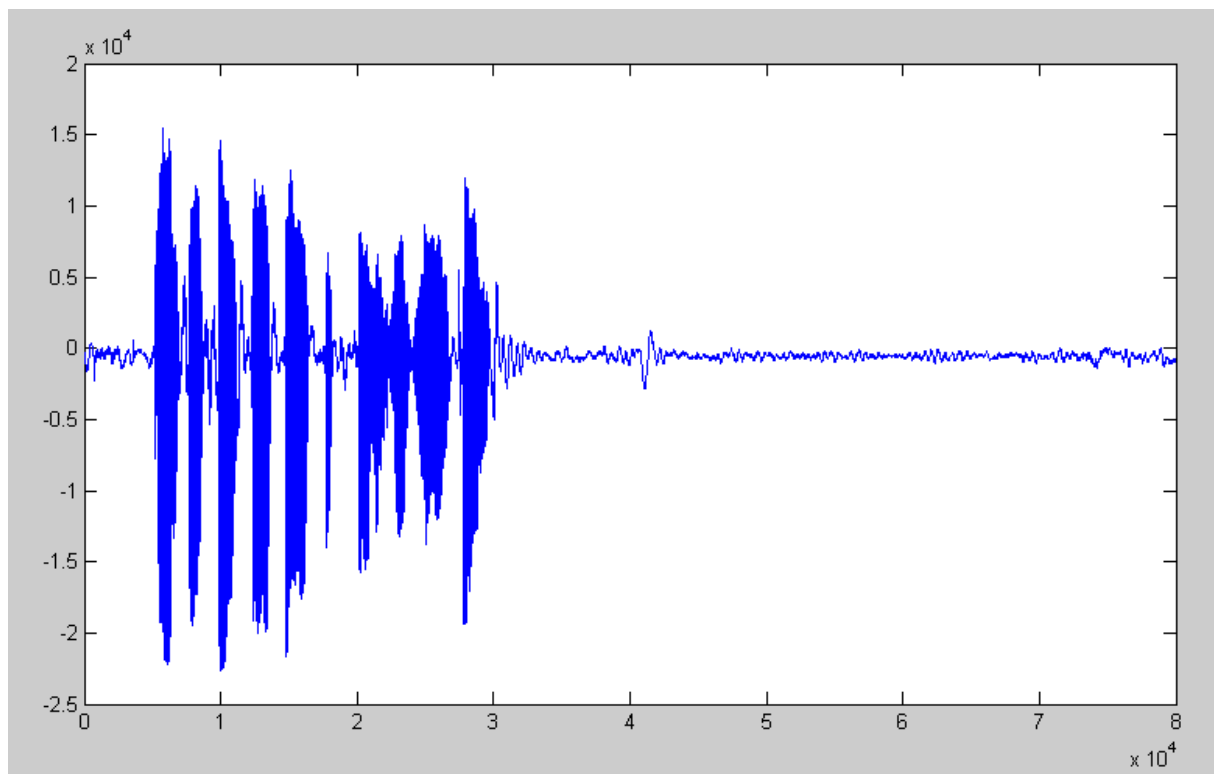


Abbildung 38: Testdaten1 (Gespräch mit anschließender Stille)

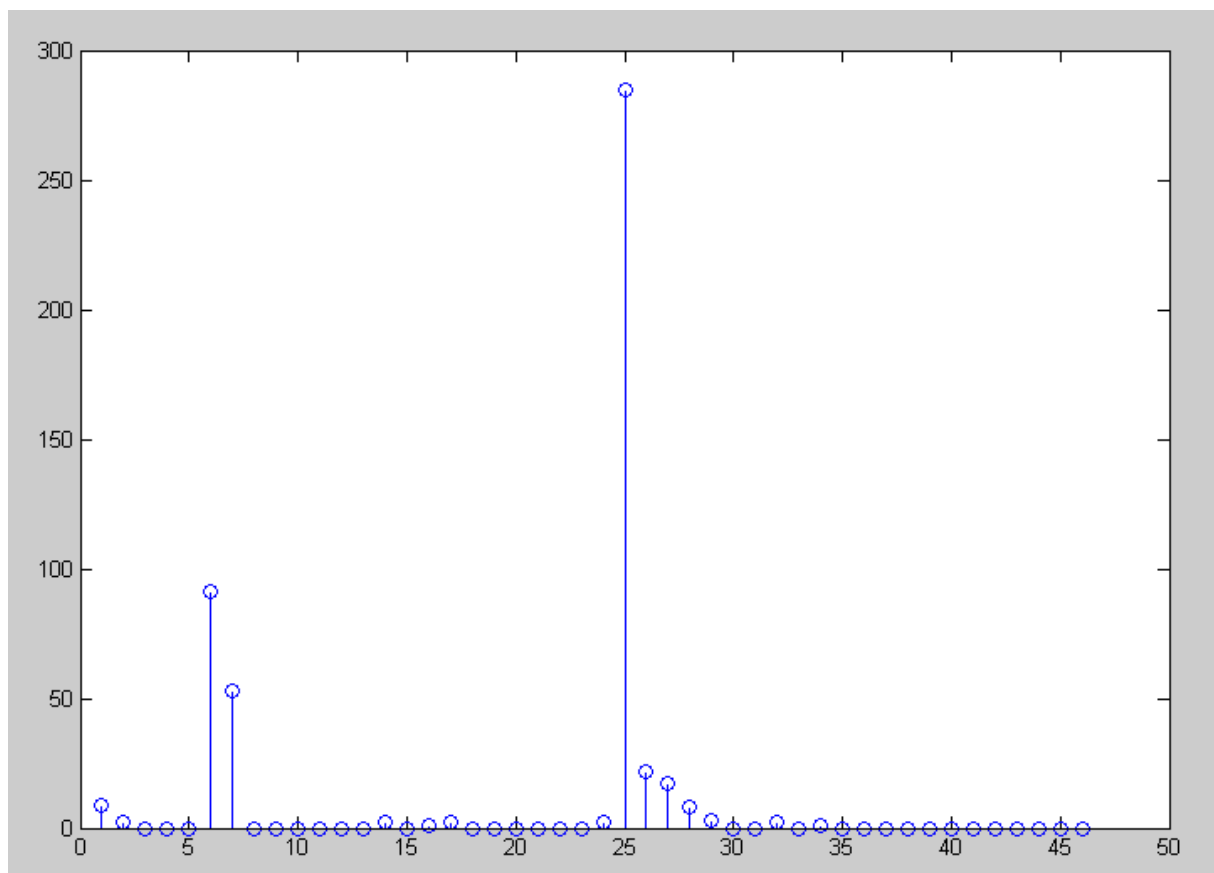


Abbildung 39: verwendete Codex bei den Testdaten1

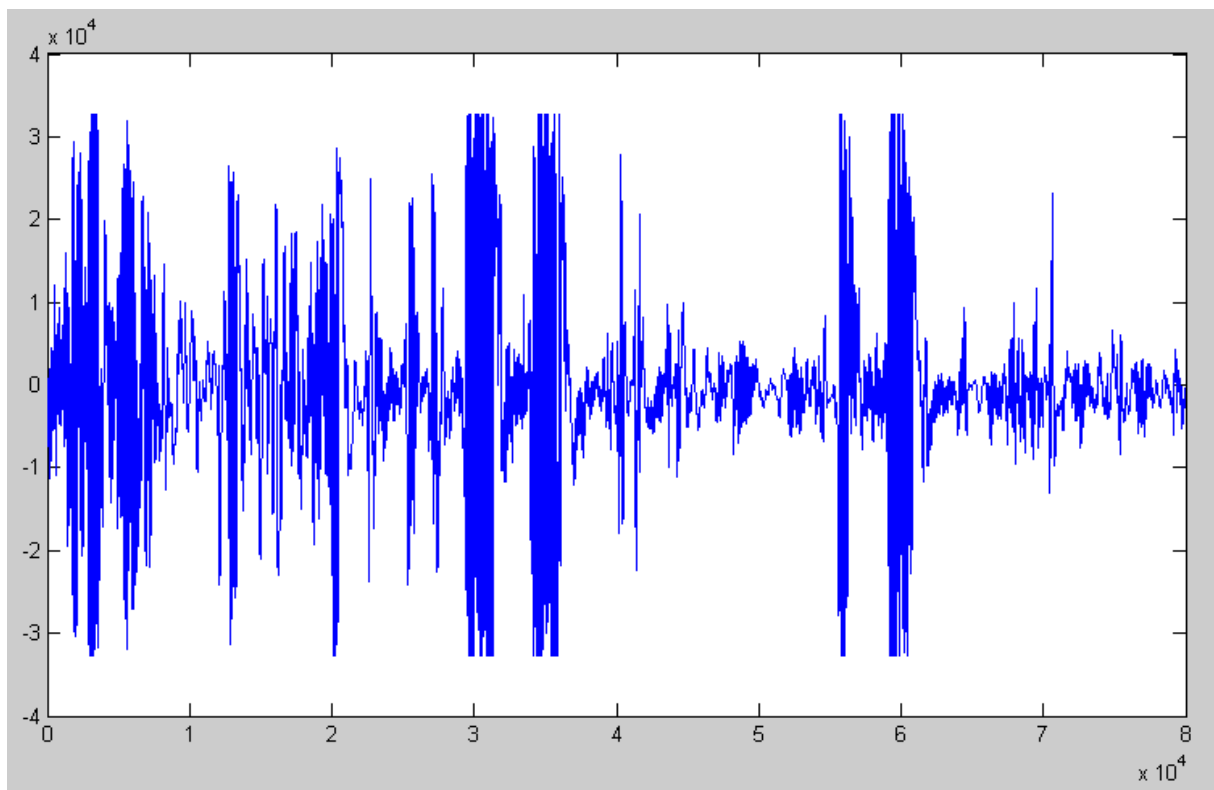


Abbildung 40: Testdaten2 (ununterbrochenes Gespräch)

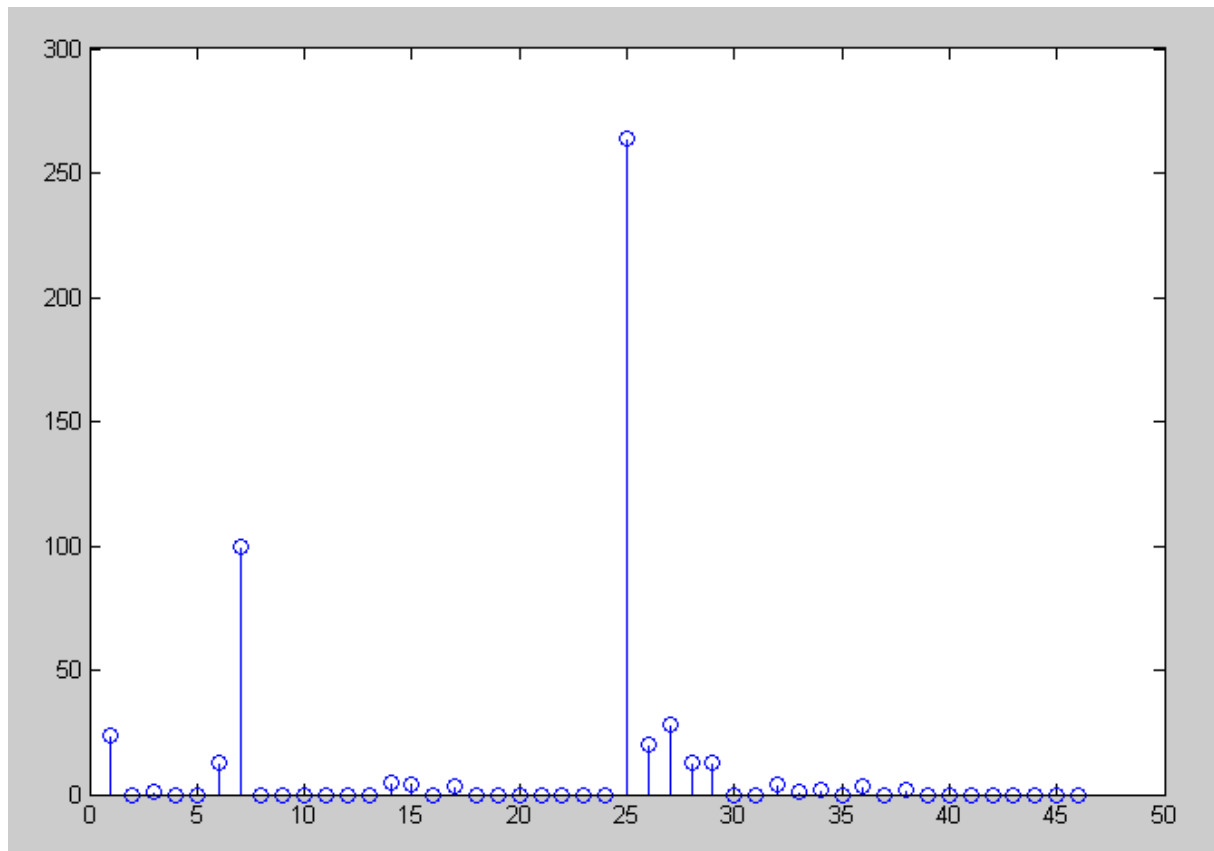


Abbildung 41: verwendete Codecs bei den Testdaten2

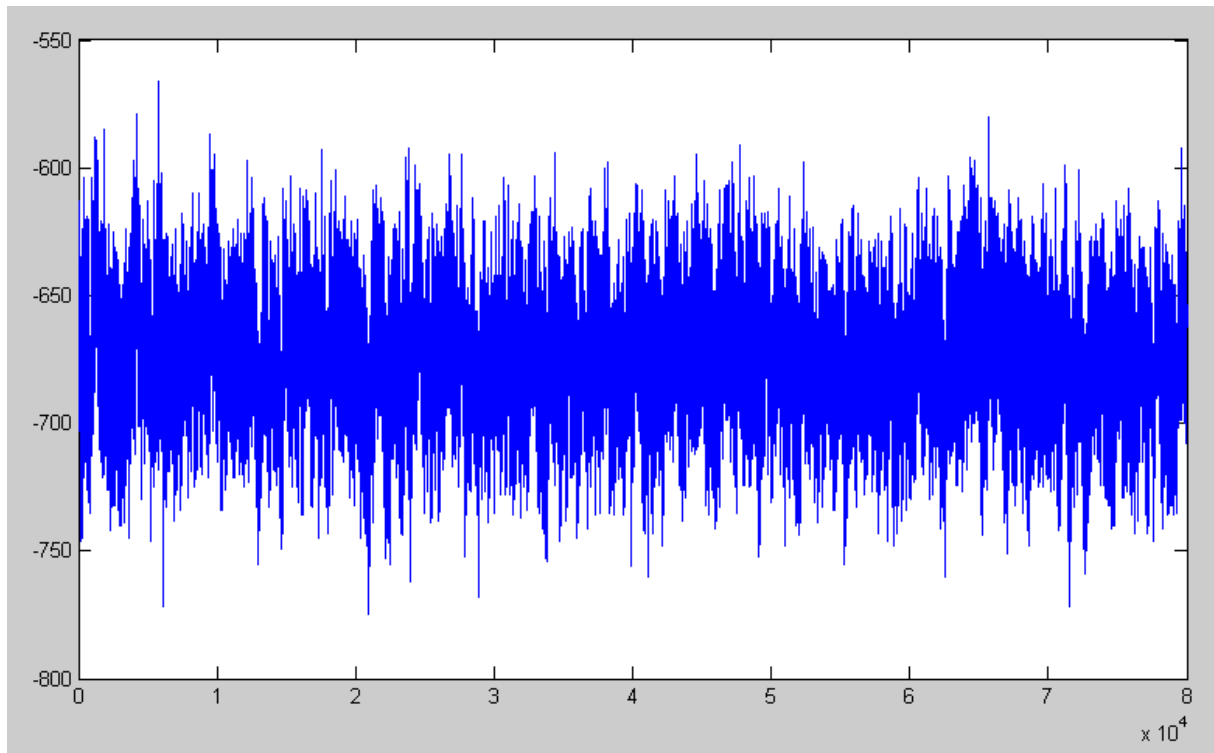


Abbildung 42: Testdaten3 (Stille)

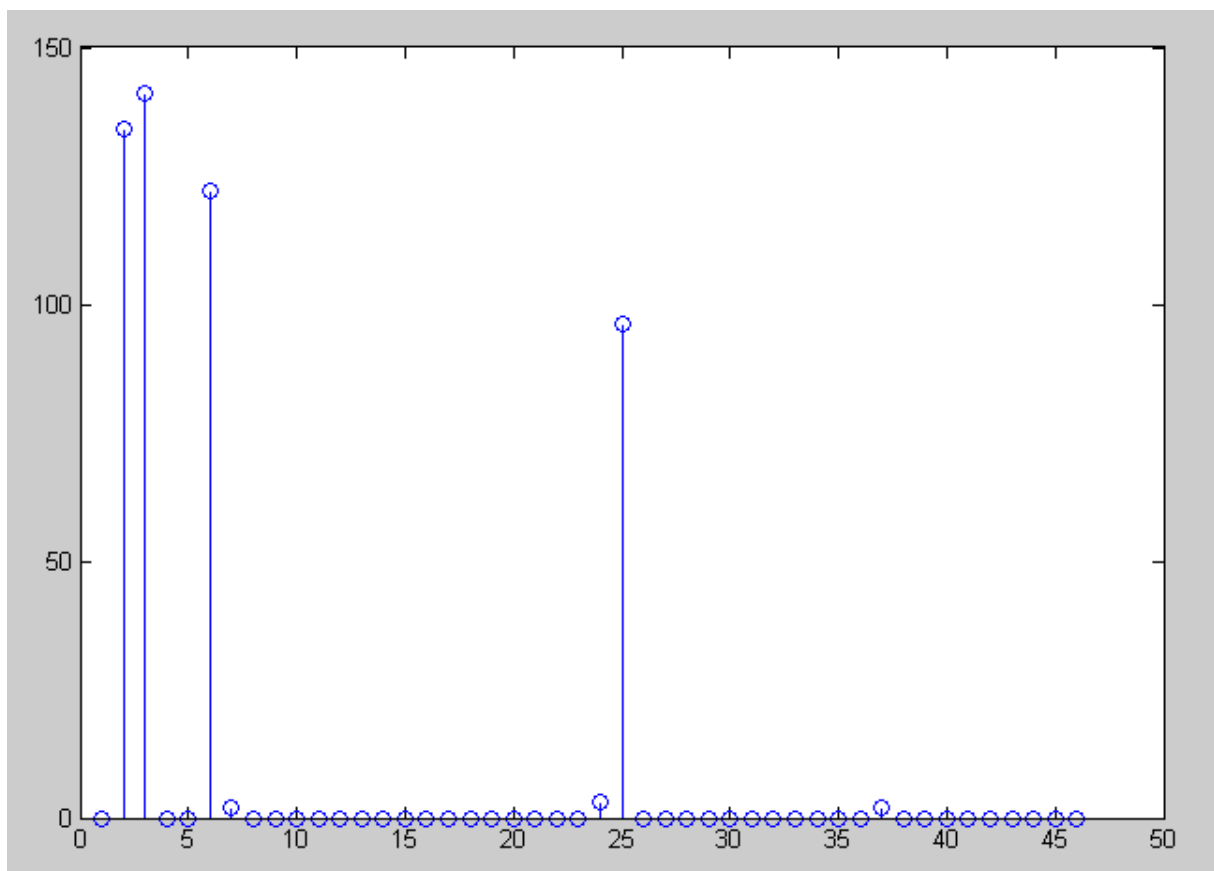


Abbildung 43: verwendete Codecs bei den Testdaten3

## 5 Fazit

Im Verlauf einer solchen Arbeit macht man verschiedene Erfahrungen. Manche sind positiv manche weniger. Wichtig ist es auf jeden Fall das man aus jeder dieser Erfahrungen etwas lernen kann und so hat auch die negativste Erfahrung ihren positiven Aspekt. Wir mussten während unserer Arbeit einige Male ziemlich niederschmetternde Ergebnisse hinnehmen. So war es bei den ersten Versuchen, bei denen wir beide mit grossen Erwartungen, unserem nagelneuen Kompressor, die ersten Blöcke zur Kompression übergaben sehr frustrierend zu sehen, dass uns der Kompressor gleich alle Blöcke wesentlich grösser zurückgibt als sie vor der Kompression waren. So hätten wir das beide nicht erwartet. Also startet man ein wenig geschockt von neuem damit sich zu überlegen auf welche weise man denn bessere Ergebnisse erzielen kann. Was wir aus diesen Ergebnissen sicher haben mitnehmen können war, dass unser Kompressor sicher die Möglichkeit haben muss, Blöcke die er sogar grösser macht als sie es zuvor waren direkt in Originalform zu übermitteln. Damit würden wir verhindern können, dass der Kompressionsfaktor grösser als 1 wird.

### 5.1 Erreichte Ziele

Die Ziele unserer Arbeit waren, ein Verfahren zu evaluieren und zu implementieren, mit dem es möglich ist, die Audiodaten eines Telefongesprächs verlustfrei zu komprimieren. Zusätzlich sollte das Verfahren ins Media-Lab-SIP-Telefon integriert werden.

Es ist uns schlussendlich gelungen, einen Kompressor mit zugehörigem Dekompressor fertig zu stellen, welcher eine Kompression um einen Faktor von bis zu 60% erreicht. Dazu war es nötig, insgesamt 45 verschiedene Arten der Codierung zu implementieren, von welchen der Kompressor immer das Verfahren verwendet, welches für den momentanen Block das beste Resultat erzielt. Entgegen allen Voraussagen wird der arithmetische Coder nicht als häufigste Methode eingesetzt. Dieser hätte der Theorie nach eigentlich die besten Ergebnisse erzielen sollen, was er aber offensichtlich nicht macht. Der arithmetische Coder arbeitet in zwei Ausführungen in unserem Kompressor. Zum einen arbeitet er mit „Ordnung 0“ zum anderen mit „Ordnung 1“. Beide Ausführungen erreichen nicht die von ihnen erwarteten Leistungen. Dies ist mit grosser Wahrscheinlichkeit eine Folge davon, dass unsere Datenblöcke mit 160 Byte pro Block sehr klein sind. Bei der Komprimierung von grösseren Datenblöcken steigt die Effizienz dieses Coders. Der Coder mit „Ordnung 0“ arbeitet mit einer fixen Wahrscheinlichkeitstabelle, die er vor Beginn der Codierung erstellt. Diese Tabelle muss er dem Decoder übermitteln. Unsere Vermutung ist es nun, dass diese Tabelle bei so kleinen Datenblöcken die erreichte Kompression wieder zunichte macht. Der Coder mit „Ordnung 1“ beginnt mit einer vorgegebenen Wahrscheinlichkeitstabelle die den Daten erst zu Laufzeit angepasst wird. Das führt dazu dass er am Anfang eines Blockes nicht sehr effektiv arbeitet und erst mit der Zeit seine Leistung verbessern kann. Hier ist unsere Vermutung, dass die Datenblöcke so kurz sind, dass der Coder es nicht schafft seine Effektivität so zu steigern, dass gute Kompression möglich ist. Huffman Coder, welcher eigentlich eine einfachere Art eines Entropiecodierers darstellt, erzielt wesentlich bessere Kompressionsraten. Durch die Behandlung mit dem LPC-Verfahren lässt sich die Entropie in unseren Datenblöcken merklich senken. Es hat sich jedoch herausgestellt, dass die einfachste Variante von LPC überhaupt, nämlich die einfache Differenzbildung zwischen den Zeichen eines Blockes die besten Resultate erzielt. Die Datenrate des SIP-Telefons sollte mit integriertem Kompressor um ca. einen Drittel gesenkt werden können.

Fazit: „Manchmal ist weniger mehr“.

## 5.2 Nicht Erreichte Ziele

Es ist uns leider nicht mehr gelungen, unseren Kompressor und Dekompressor in das Media-Lab-SIP-Telefon zu integrieren. Da wir besonders mit dem arithmetischen Coder sehr viel Zeit verloren haben ist es uns zum Schluss aus Zeitgründen nicht mehr möglich gewesen den Kompressor zu integrieren, deshalb haben wir diesen Teil in Absprache mit Herrn Schuster weggelassen.

## 6 Ausblick

Es sollte ohne grössere Schwierigkeiten möglich sein den vorgelegten Kompressor in das SIP-Telefon zu integrieren. Mit einigen weiteren Tests könnten genauere Statistiken darüber erstellt werden, welche Codierungsarten effizient arbeiten und somit am häufigsten eingesetzt werden. Mit den daraus gewonnenen Erkenntnissen könnte der Kompressor noch um einige Codierungsarten erleichtert, und seine Geschwindigkeit so noch gesteigert werden.

Die Datenreduktion mit unserem Kompressor beträgt nachdem der Protokollheader für die Übertragung angefügt ist noch ca. einen drittel, also eine Reduktion von rund 100kbit/s auf 66kBit/s entspricht. Dies ermöglicht die Übertragung von vier Telefongesprächen über eine Leitung über die zuvor nur drei Gespräche übertragen werden konnten, und dies bei exakt gleicher Qualität. Besonders bei ausgelasteten Leitungen wäre der Einsatz des Kompressors also nicht uninteressant, zumal dadurch das Ersetzen der Leitung durch eine Leistungsstärkere zumindest herausgeschoben werden könnte.

## **A Anhang**

- ITU-T Recommendation G.711

### **A.1 Inhalt der CD**

- Aufgabenstellung
- Quellcode des Endgültigen Kompressors
- Quellcode aller Entworfenen Funktionen
- Quellcode Matlab
- Bericht
- ITU-T Recommendation G.711.pdf

## **Literatur**

- (1) Kalid Sayood ; Data Compression  
ISBN: 1558605584
- (2) Mark Nelson, Jean-Loup Gailly; The Data Compression Book  
ISBN: 1558514341
- (3) Anatol Badach; Voice over IP  
ISBN: 3446403043