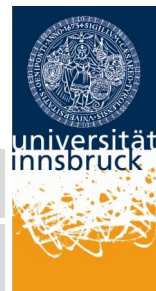


universität innsbruck

Fakultät für Mathematik, Informatik und Physik



Martin Nocker

# R-Paket für Kanalkodierung mit Faltungskodes

Bachelorarbeit

31. Mai 2016



Universität Innsbruck  
Institut für Informatik  
Technikerstr. 21a · 6020 Innsbruck · Österreich  
<http://informatik.uibk.ac.at/>

# **R-Paket für Kanalkodierung mit Faltungskodes**

Bachelorarbeit

vorgelegt von

**Martin Nocker**

geb. am 1. Mai 1993  
in Innsbruck

angefertigt am

**Institut für Informatik  
Leopold-Franzens-Universität Innsbruck**

Betreuer: **Univ.-Prof. Dr. Rainer Böhme  
Dr. Pascal Schöttle**

Abgabe der Arbeit: **31. Mai 2016**

## Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

## Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.

I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Martin Nocker)

Innsbruck, 31. Mai 2016

## **Zusammenfassung**

---

# Inhaltsverzeichnis

---

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Grundlagen</b>	<b>2</b>
2.1	Kanalkodierung . . . . .	2
2.1.1	Koderate . . . . .	3
2.1.2	Hamming-Distanz . . . . .	3
2.1.3	Hamming-Gewicht . . . . .	4
2.2	Faltungskodierung . . . . .	4
2.2.1	Kodiererdarstellung und Kodierung . . . . .	4
2.2.2	Dekodierung . . . . .	6
2.2.2.1	hard decision . . . . .	7
2.2.2.2	soft decision . . . . .	8
2.2.3	Katastrophale Faltungskodierer . . . . .	9
2.2.4	Systematische Faltungskodierer . . . . .	10
2.2.5	Rekursiv systematische Faltungskodierer . . . . .	11
2.2.6	Terminierung . . . . .	12
2.2.7	Punktierung . . . . .	12
<b>3</b>	<b>Verwendete Technologien</b>	<b>14</b>
3.1	R, RStudio, Pakete . . . . .	14
3.2	C++, Rcpp . . . . .	16
3.3	RMarkdown, L <sup>A</sup> T <sub>E</sub> X, TikZ . . . . .	17
<b>4</b>	<b>Implementierung</b>	<b>19</b>
4.1	Faltungskodierer . . . . .	19
4.2	Kodierung . . . . .	21
4.3	Dekodierung . . . . .	22
4.4	Rauschen . . . . .	23

---

4.5	Punktierung . . . . .	24
<b>5</b>	<b>R-Paket Schnittstelle</b>	<b>25</b>
5.1	Faltungskodierung . . . . .	25
5.1.1	ConvGenerateEncoder . . . . .	25
5.1.2	ConvGenerateRscEncoder . . . . .	26
5.1.3	ConvEncode . . . . .	27
5.1.4	ConvDecodeSoft . . . . .	28
5.1.5	ConvDecodeHard . . . . .	28
5.1.6	ConvSimulation . . . . .	29
5.2	Hilfsfunktionen . . . . .	30
5.2.1	ConvGetPunctuationMatrix . . . . .	30
5.2.2	ConvOpenPDF . . . . .	30
5.3	Kanalkodierung . . . . .	31
5.3.1	ApplyNoise . . . . .	31
5.3.2	ChannelcodingSimulation . . . . .	32
5.3.3	PlotSimulationData . . . . .	32
<b>6</b>	<b>Visualisierung</b>	<b>34</b>
6.1	Kodierung . . . . .	35
6.2	Dekodierung . . . . .	37
6.3	Simulation . . . . .	40
<b>7</b>	<b>Beispiele</b>	<b>41</b>
7.1	Erzeugen von Kodierer und Punktierungsmatrix . . . . .	41
7.2	Kodieren und Dekodieren ohne Punktierung . . . . .	42
7.3	Kodieren und Dekodieren mit Punktierung . . . . .	43
7.4	Simulation . . . . .	43
7.4.1	Faltungskodierung . . . . .	44
7.4.2	Kanalkodierung . . . . .	45
7.4.3	Vergleich von Simulationen . . . . .	46
<b>8</b>	<b>Fazit, Ausblick, Erweiterungen</b>	<b>48</b>

---

## Kapitel 1

# Einleitung

---

Kanalkodierung stellt einen wichtigen Teil der Nachrichtentechnik dar. Kanalkodierung stellt Methoden zur Verfügung, um Fehler, die während der Übertragung über einen verrauschten Kanal auftreten, zu korrigieren. Die Leistungsfähigkeit und Zuverlässigkeit vieler digitaler Systeme basiert auf der Verwendung von Kanalkodierung. Eine Art der Kanalkodierung stellen Faltungskodes dar, auf welche sich diese Arbeit konzentriert. Verwendung finden Faltungskodes in der Mobil- und Satellitenkommunikation aber vor allem bilden sie die Basis für Turbokodes, welche die Faltungskodes mittlerweile aufgrund ihrer noch höheren Leistungsfähigkeit abgelöst haben.

Ziel dieser Arbeit ist die Implementierung von Faltungskodes mithilfe der Programmiersprache R. Das entwickelte R-Paket dient zukünftigen Studenten zu Lernzwecken und soll sie beim Verstehen von Faltungskodes unterstützen.

---

## Kapitel 2

# Grundlagen

---

In diesem Kapitel werden die theoretischen Grundlagen eingeführt. In Kapitel 2.1 werden Prinzipien und Eigenschaften der Kanalkodierung beschrieben. Kapitel 2.2 geht auf Faltungskodes ein, eine Art der Kanalkodierung auf die sich diese Arbeit konzentriert.

### 2.1 Kanalkodierung

Kanalkodierung kann als Zuordnung bzw. Abbildung von Quellzeichen (Zeichen die eine Informationsquelle emittiert) zu Kanalzeichen (Zeichen die über den Kommunikationskanal übertragen werden) angesehen werden. Der Kanal enthält ein Rauschen, d.h. Informationen die von der Quelle emittiert werden, können verändert beim Empfänger ankommen. Daher wird der Kanal auch als *verrauschter* Kanal bezeichnet. Würde eine Information unkodiert über den verrauschten Kanal übertragen werden, kann die verfälschte Nachricht nicht wiederhergestellt werden. Daher fügt der Kanalkodierer den Quellzeichen Redundanz hinzu, sodass empfängerseitig verfälschte Zeichen erkannt und korrigiert werden können. Abbildung 2.1 zeigt einen Kommunikationskanal inklusive Kanalkodierung.

Eine Nachricht  $\mathbf{u} = (u_1, u_2, \dots, u_k)$  wird in ein Kodewort  $\mathbf{v} = (v_1, v_2, \dots, v_n)$ , das Redundanz enthält, kodiert. Vor der Übertragung über den Kanal werden die Bits auf die Signalpegel +1 bzw. -1 folgendermaßen abgebildet:

$$\text{Signal}(\text{Bit}) = \begin{cases} +1 & \text{wenn Bit} = 0 \\ -1 & \text{wenn Bit} = 1 \end{cases} \quad (2.1)$$



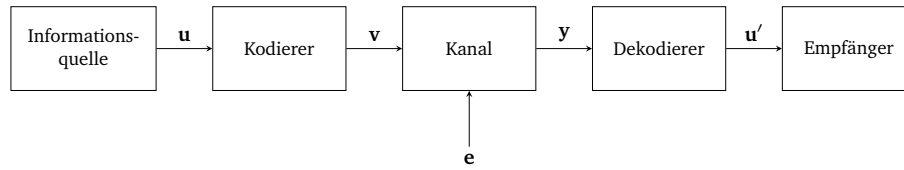


Abbildung 2.1 – Kommunikationskanal

Bei der Übertragung wird das Signal durch Rauschen, welches als Fehlervektor  $\mathbf{e} = (e_1, e_2, \dots, e_n)$  dargestellt wird, verfälscht. Der Empfangsvektor ergibt sich aus der Überlagerung von  $\mathbf{v}$  und  $\mathbf{e}$ . Der Empfangsvektor wird vor der Dekodierung von den Signalpegel zurück auf die Bit-Werte 0 bzw. 1 abgebildet.

$$\text{Kode}(\text{Signal}) = \begin{cases} 0 & \text{wenn Signal} \geq 0 \\ 1 & \text{sonst} \end{cases} \quad (2.2)$$

Das daraus resultierende Kodewort wird vom Dekodierer zur Schätzung  $\mathbf{u}'$  der originalen Nachricht dekodiert. Ziel der Kanalkodierung ist es die Wahrscheinlichkeit, dass  $\mathbf{u}' = \mathbf{u}$  ist, zu maximieren.

### 2.1.1 Koderate

Die Koderate  $R$  eines Codes beschreibt das Verhältnis der Länge zwischen Quellwort  $\mathbf{u} = (u_1, u_2, \dots, u_k)$  und Kodewort  $\mathbf{v} = (v_1, v_2, \dots, v_n)$ .

$$R = \frac{k}{n} < 1 \quad (2.3)$$

In anderen Worten beschreibt die Koderate das Verhältnis zwischen Information und Redundanz im übertragenen Kodewort. Bei hoher Redundanz ergibt sich eine niedrige Koderate. Die Übertragung ein und derselben Information bei gleicher Übertragungsgeschwindigkeit dauert bei Codes mit niedriger Koderate länger, als bei Codes mit höherer Koderate.

### 2.1.2 Hamming-Distanz

Die Hamming-Distanz  $d$  (auch  $d_H$ ) zweier Kodewörter  $\mathbf{a} = (a_1, a_2, \dots, a_n)$  und  $\mathbf{b} = (b_1, b_2, \dots, b_n)$  entspricht der Anzahl an Stellen in denen sich die beiden Kodewörter unterscheiden:

$$d(a, b) = |\{i \in \{1, 2, \dots, n\} \mid a_i \neq b_i\}|. \quad (2.4)$$

Für binäre Kodewörter ergibt sich die Hamming-Distanz aus der binären Addition der Kodewörter:

$$d(a, b) = \sum_{i=1}^n (a_i \oplus b_i). \quad (2.5)$$

### 2.1.3 Hamming-Gewicht

Das Hamming-Gewicht  $w$  eines binären Kodeworts  $\mathbf{a} = (a_1, a_2, \dots, a_n)$  entspricht der Anzahl an 1 im Wort:

$$w(a) = \sum_{i=1}^n a_i. \quad (2.6)$$

## 2.2 Faltungskodierung

Faltungskodes sind blockfreie Codes, d.h. Quellzeichen werden nicht in Blöcke fester Länge unterteilt, sondern es wird ein Informationsstrom kodiert, sodass ein einziges Kodewort resultiert. Ein weiterer Unterschied zu Blockcodes besteht darin, dass Kodebits nicht nur vom aktuellen Eingangsbit abhängen, sondern auch von vorherigen Eingangsbits. Die Redundanz wird bei Faltungskodes kontinuierlich in das Kodewort eingefügt. Im Allgemeinen können Faltungskodierer beliebig viele Eingänge haben. Trotz besserer Koderate bei Kodierern mit mehreren Eingängen sind nur Kodierer mit einem Eingang von praktischer Relevanz. Im Folgenden werden nur noch Faltungskodierer mit einem Eingang betrachtet.

### 2.2.1 Kodiererdarstellung und Kodierung

Faltungskodierer lassen sich einfach durch ein Schieberegister und mehrere logische XOR-Gatter darstellen. Bei einem Faltungskodierer mit  $N$  Ausgängen und einem Schieberegister der Länge  $M$  wird ein Eingangsbit  $u \in \{0, 1\}$  in eine Kodesequenz  $\mathbf{v}$  der Länge  $N$  ( $\mathbf{v} \in \{0, 1\}^N$ ) kodiert. Es ergibt sich somit eine Koderate von  $R = \frac{1}{N}$ . Ein Ausgang wird durch ein sogenanntes *Generatorpolynom* definiert. Dieses stellt eine Linearkombination der  $M$  Elemente des Schieberegisters und dem Eingangssignal dar und werden durch die XOR-Gatter abgebildet. Alle Generatorpolynome werden in einer *Generatormatrix*

$$G = (g_1, g_2, \dots, g_N) \quad (2.7)$$

angegeben, wobei das Generatorpolynom  $g_i$  Ausgang  $i$  definiert. Zur Definition eines Faltungskodierers im späteren Programm müssen die oktalen Generatorpolynome angegeben werden.

Ein weiterer wichtiger Parameter von Faltungskodes ist die *Einflusslänge* (constraint length) die angibt wie oft sich ein Eingangsbit auf die Kodierung auswirkt und wird durch die Länge des Schieberegisters bestimmt. Ein Eingangsbit beeinflusst  $M + 1$  mal die Kodierung.

Die Verhaltensweise eines Faltungskodierers kann weiters durch seinen *Zustandsgraphen* beschrieben werden. Ein Zustand entspricht einer bestimmten Bitbelegung des Schieberegisters. Für einen Faltungskodierer mit einem Schieberegister der Länge  $M$  ergeben sich  $2^M$  Zustände. Faltungskodierer starten, falls nicht explizit angegeben, im Nullzustand, d.h. die Elemente des Schieberegisters sind mit 0 initialisiert.

**Beispiel 1.** Ein Faltungskodierer ist gegeben durch das Schaltbild bzw. Zustandsdiagramm in Abbildung 2.2. Der Faltungskodierer besitzt die Generatormatrix

$$G = (7_8, 5_8) = \begin{pmatrix} 111 \\ 101 \end{pmatrix} = (1 + D + D^2, 1 + D^2)$$

Beispiel 1 zeigt die verschiedenen Notationen für die Generatormatrix. Die am häufigsten verwendete Schreibweise ist die Darstellung der Generatorpolynome in oktaler Form. Dabei werden die Polynome in binärer Schreibweise konzipiert zu oktalen Zahlen zusammengefasst. Bei der binären Schreibweise entspricht die Bitposition des Polynoms dem Element im Schaltbild, d.h. das MSB<sup>1</sup> des Polynoms steht für das Eingangssignal, das LSB<sup>2</sup> des Polynoms steht für den Inhalt des letzten (am weitesten rechts liegende) Element des Schieberegisters. In die Linearkombination zur Definition des Ausgangssignals werden jene Elemente miteinbezogen, an deren Stelle im binären Polynom eine 1 steht. [2] führt als Notation die binären Polynome über der Variable  $D$  („delay“) ein. Das Eingangssignal und die Schieberegisterelemente ent-

<sup>1</sup>Most Significant Bit

<sup>2</sup>Least Significant Bit

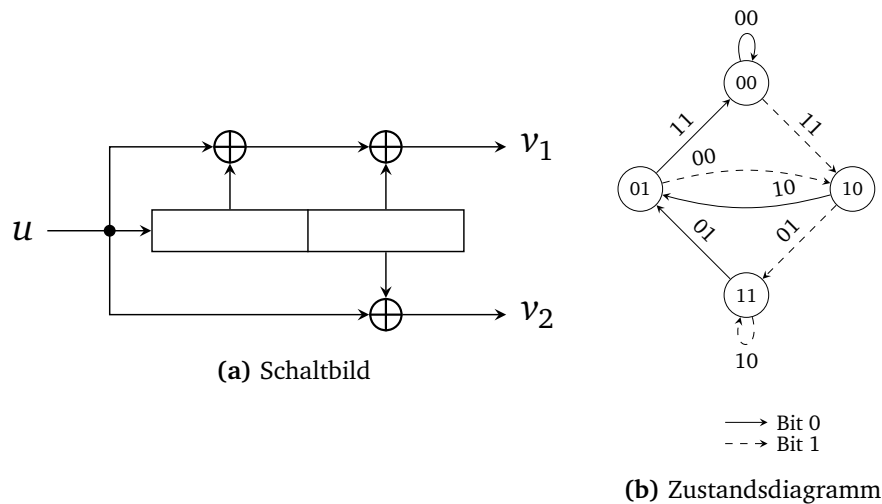


Abbildung 2.2 – Beispiel für Faltungskodierer

sprechen einer Potenz von  $D$ , wobei das Eingangssignal  $D^0$  und das letzte Schieberegisterelement  $D^M$  entspricht. Das Generatorpolynom entspricht der Summe aller Potenzen deren Elemente Teil der Linearkombination sind.

Weiters wird das Zustandsdiagramm abgebildet. Die Knoten stellen die Zustände des Kodierers, d.h. die Belegungen des Schieberegisters, dar. Die gerichteten Kanten entsprechen einem Übergang bei einem Eingangsbit, wobei eine durchgezogene Kante einer 0 als Eingangsbit entspricht, eine gestrichelte Kante einer 1. Die Kantenbewertungen entsprechen den Ausgangsbits.

**Beispiel 2.** Gegeben sei der Faltungskodierer aus Beispiel 1. Eine Nachricht  $\mathbf{u} = (110100)$  wird in den Kode  $\mathbf{v} = (11\ 01\ 01\ 00\ 10\ 11)$  kodiert.

### 2.2.2 Dekodierung

Zur Dekodierung von Faltungskodes wird der *Viterbi-Algorithmus* angewendet. Der Algorithmus verwendet zur Dekodierung einer Kodesequenz das *Trellis-Diagramm* (kurz: Trellis). Abbildung 2.3 zeigt das Trellis zur Kodierung der Nachricht aus Beispiel 2.

Das Trellis ist eine Erweiterung des Zustandsdiagramms um eine Zeitachse (x-Achse). Die Zustände sind auf der y-Achse aufgetragen. Das Diagramm startet wie die Kodierung im Nullzustand. Ein durchgezogener Pfeil entspricht einer 1 als Eingangsbit, ein gestrichelter Pfeil einer 0 als Eingangsbit. Die Pfeile sind wiederum mit den entsprechenden Ausgangsbits, die das Kodewort

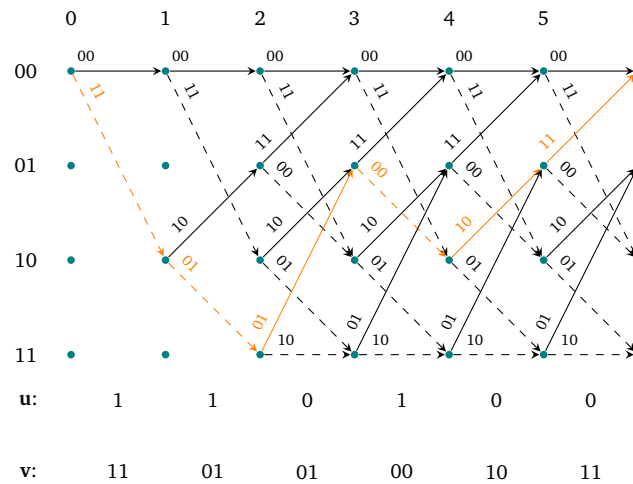


Abbildung 2.3 – Trellis für die Kodierung aus Beispiel 2.

ergeben, bewertet.

Der Viterbi-Algorithmus verwendet das Trellis zur Dekodierung eines empfangenen Codes. Dabei wird für den empfangenen Code im Trellis jener Pfad gesucht der eine bestimmte Metrik minimiert bzw. maximiert. Die Metrik hängt von der Art der Dekodierung ab. Es wird zwischen der *hard decision* Dekodierung und *soft decision* Dekodierung unterschieden werden.

### 2.2.2.1 hard decision

Die hard decision Dekodierung sucht den Pfad mit der geringsten Anzahl an Bitfehlern im Trellis. Der Algorithmus startet im Nullzustand und geht von links nach rechts durch das Trellis. Es werden die Metriken der Kanten berechnet. Für eine Kante die einen Zustand  $s$  zum Zeitpunkt  $t$  mit einem Zustand  $s'$  zum Zeitpunkt  $t + 1$  verbindet, ist die Metrik die Hamming-Distanz zwischen der Bewertung der Kante zwischen  $s$  und  $s'$  und dem zum Zeitpunkt  $t$  empfangenen Teil des Codes. Die Metrik eines Pfads im Trellis ist die Summe der Kantenmetriken des Pfads. Daher wird der Pfad mit der minimalen Metrik gesucht. Zu jedem Zeitpunkt wird in allen Zuständen die Pfadmetrik zu diesem Zustand berechnet. Treffen zwei Pfade aufeinander, wird der Pfad mit der größeren Hamming-Distanz verworfen. Am Ende erhält man durch *Backtracking* die dekodierte Nachricht. Beginnend bei der niedrigsten Metrik am Ende des Trellis wird der Pfad zum Nullzustand rückwärts durchgelaufen.

Als Ergebnis resultiert die Nachricht, dessen Kode die geringste Hamming-Distanz zum empfangenen Kode hat.

**Beispiel 3.** Gegeben sei der Faltungskodierer aus Beispiel 1 und ein empfangenes Kodewort  $\mathbf{y} = (11\ 00\ 01\ 01\ 10\ 11)$  welches durch Rauschen verfälscht wurde und dekodiert werden soll. Abbildung 2.4 zeigt die Dekodierung im Trellis. Abbildung 2.4a zeigt die Metriken aller Pfade. Die verworfenen Pfade sind grau dargestellt. Beispielsweise ergibt sich die Metrik zum Zeitpunkt 1 im Zustand 00 durch  $0 + d(00, 11) = 2$ , im Zustand 10 durch  $0 + d(11, 11) = 0$ . Der erste Summand entspricht der Hamming-Distanz des vorigen Zustands. In Abbildung 2.4b werden die verworfenen Pfade nicht mehr angezeigt. Durch Backtracking erhält man die dekodierte Nachricht  $\mathbf{u}' = (110100)$ . Der resultierende Pfad ist orange hervorgehoben. Man erkennt aus der Metrik am Ende der Trellis, dass der empfangene Kode zwei Bitfehler enthielt.

#### 2.2.2.2 soft decision

Vor der Übertragung eines Kodeworts werden die Kodebits 0 bzw. 1 nach Gleichung (2.1) auf die Signalzustände +1 bzw. -1 abgebildet. Bei der hard decision Dekodierung wird vor der Dekodierung das Signal nach Gleichung (2.2) wieder zu einem Bitvektor zurück umgewandelt. Die soft decision Dekodierung lässt diese Rücktransformation aus und verwendet die exakten Signalpegel. Dadurch erzielt die soft decision Dekodierung eine noch bessere Fehlerkorrektur. Die Metrik für eine Kante, die einen Zustand  $s$  zum Zeitpunkt  $t$  mit einem Zustand  $s'$  zum Zeitpunkt  $t + 1$  verbindet, entspricht dem Skalarprodukt der Signalzustände der Bewertung der Kante zwischen  $s$  und  $s'$  und dem zum Zeitpunkt  $t$  empfangenen Teil des Signals. Der Viterbi-Algorithmus funktioniert analog zur soft decision Dekodierung, jedoch wird der Pfad mit der maximalen Metrik gesucht. Treffen zwei Pfade aufeinander, wird der Pfad mit der kleineren Metrik verworfen. Das Backtracking beginnt hier bei der größten Metrik. Als Ergebnis resultiert die Nachricht, dessen Signal das größte Skalarprodukt mit dem empfangenen Signal hat.

Die Nachricht wird mittels *Soft-Werten* und *Hard-Werten* angegeben. Die Soft-Werte geben neben der dekodierten Nachricht die Zuverlässigkeitswerte für die Bits an, d.h. mit welcher Wahrscheinlichkeit das Bit mit dem tatsächlich gesendeten Bit der Quellnachricht übereinstimmt. Positive Soft-Werte

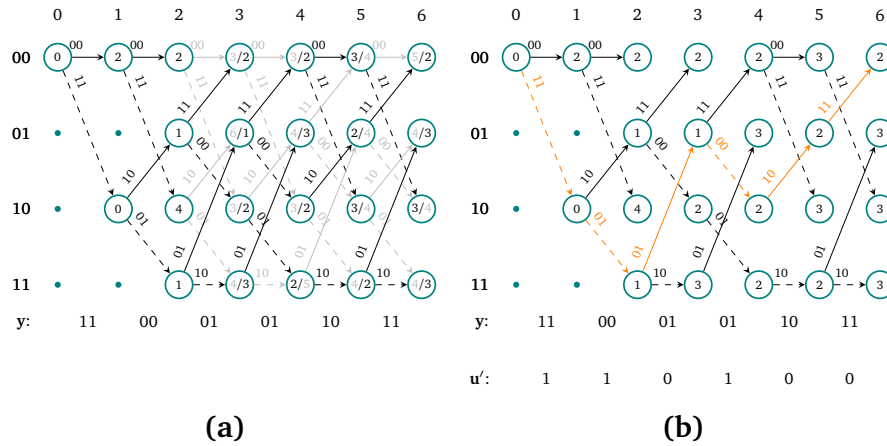


Abbildung 2.4 – Trellis der hard decision Dekodierung aus Beispiel 3.

werden auf eine 0, negative auf eine 1 abgebildet. Der Betrag des Soft-Wertes gibt die Zuverlässigkeit an, wobei gilt, je höher der Betrag, desto zuverlässiger das dekodierte Bit. Die Hard-Werte entsprechen der Abbildung der Soft-Werte auf die Bit-Werte 0 bzw. 1. Der Viterbi-Algorithmus mit Soft-Werten wird auch SOVA (Soft Output Viterbi Algorithm) genannt. Die Berechnung der Soft-Werte ist in [5, S. 228 ff.] beschrieben.

**Beispiel 4.** Gegeben sei der Faltungskodierer aus Beispiel 1 und ein empfangenes Signal  $\mathbf{y} = (-1 \ -1 \ 0.5 \ -1 \ 0.9 \ -0.9 \ 1 \ -0.2 \ -0.8 \ -0.1 \ -1 \ -1)$  welches durch Rauschen verfälscht wurde und dekodiert werden soll. Abbildung 2.5 zeigt die Dekodierung im Trellis. Abbildung 2.5a zeigt die Metriken aller Pfade. Die verworfenen Pfade sind grau dargestellt. Beispielsweise ergibt sich die Metrik zum Zeitpunkt 1 im Zustand 00 durch  $0 + (1 \cdot (-1) + 1 \cdot (-1)) = 0 - 2 = -2$ , im Zustand 10 durch  $0 + ((-1) \cdot (-1) + (-1) \cdot (-1)) = 0 + 2 = 2$ . Der erste Summand entspricht der Metrik des vorigen Zustands. Die Bits der Kantenbewertung im Trellis müssen für die Berechnung des Skalarprodukts auf die Signalzustände  $+1$  bzw.  $-1$  abgebildet werden. In Abbildung 2.5b werden die verworfenen Pfade nicht mehr angezeigt. Durch Backtracking erhält man die dekodierte Nachricht  $\mathbf{u}' = (110100)$ . Der resultierende Pfad ist orange hervorgehoben.

### 2.2.3 Katastrophale Faltungskodierer

Sei  $\mathbf{u}$  eine Nachricht, die mit den Generatorpolynomen in  $G$  zum Kode  $\mathbf{v}$  kodiert wird. Nach der Übertragung erhält der Dekodierer den Kode  $\mathbf{y}$ , der

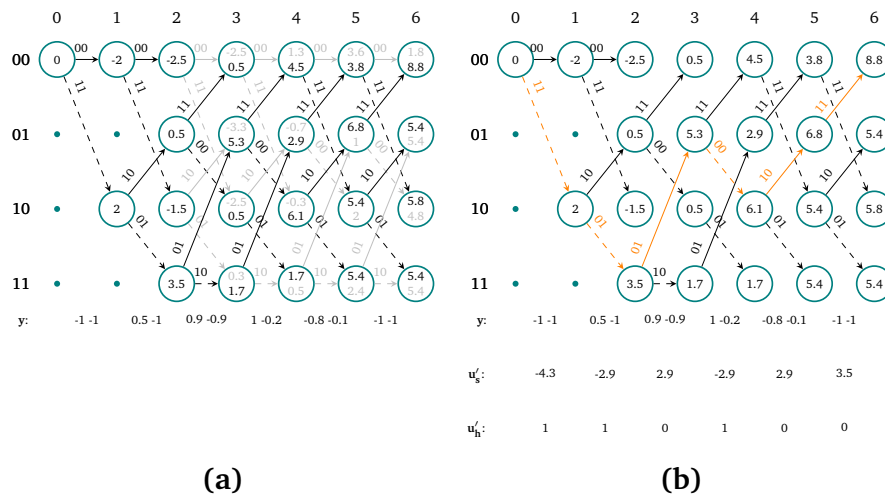


Abbildung 2.5 – Trellis der soft decision Dekodierung aus Beispiel 4.

aufgrund von Rauschen verändert sein könnte. Der Dekodierer findet ein Kodewort  $\mathbf{v}'$  welches  $\mathbf{v}$  am nächsten ist. Aus  $\mathbf{v}'$  kann die Schätzung  $\mathbf{u}'$  berechnet werden, die möglichst  $\mathbf{u}$  entspricht. Dies ist bei einer fehlerfreien Übertragung, d.h.  $\mathbf{v}' = \mathbf{v}$ , sicherlich der Fall. Zu untersuchen ist der Fall  $\mathbf{v}' \neq \mathbf{v}$ : Zu erwarten wäre, wenn sich  $\mathbf{v}'$  und  $\mathbf{v}$  in endlich vielen Stellen unterscheiden, dass sich auch  $\mathbf{u}'$  und  $\mathbf{u}$  in endlich vielen Stellen unterscheiden. Wenn sich  $\mathbf{u}'$  und  $\mathbf{u}$  in unendlich vielen Stellen unterscheiden wäre das „katastrophal“. Ein Faltungskodierer wird katastrophal genannt, wenn es eine Nachricht mit unendlichem Hamming-Gewicht gibt, sodass sein Code endliches Hamming-Gewicht hat. Katastrophale Kodierer sind zu vermeiden, da eine endliche Anzahl an Übertragungsfehler zu einer unendlichen Anzahl an Dekodierfehler führen kann. [2, S. 569]

Zur Überprüfung, ob ein Kodierer katastrophal ist, hilft das Theorem von Massey-Sain [2, S. 570]. Gegeben ein Faltungskodierer mit einem Eingang und der Generatormatrix  $G$  als Polynome über  $D$ , so ist der Faltungskodierer nicht katastrophal genau dann, wenn der größte gemeinsame Teiler der Generatorpolynome eine Potenz von  $D$  ist.

### 2.2.4 Systematische Faltungskodierer

Bei systematischen Faltungskodierern entspricht ein Ausgang dem Eingangssignal. Die Quellinformation ist somit explizit im Kodewort enthalten. Abbil-



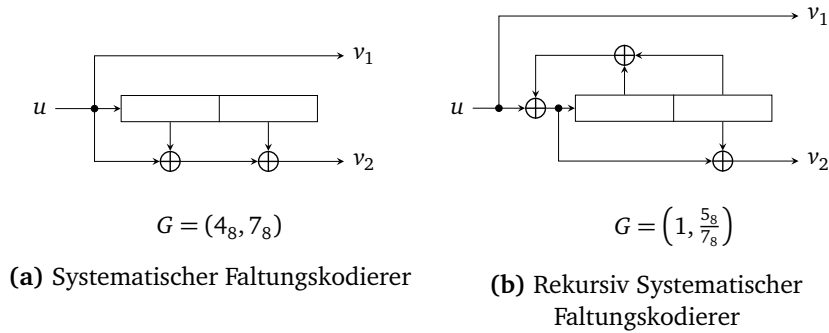


Abbildung 2.6 – Verschiedene Faltungskodierertypen

Abbildung 2.6a zeigt einen systematischen Faltungskodierer mit der Generatormatrix  $G = (4_8, 7_8)$ . Systematische Faltungskodierer sind nie katastrophal, sind jedoch weniger robust wie nichtsystematische Faltungskodierer. [5, S. 217]

### 2.2.5 Rekursiv systematische Faltungskodierer

Rekursiv systematische Faltungskodierer (RSC-Kodierer<sup>3</sup>) weisen sowohl einen systematischen Ausgang als auch eine Rückkopplung des Schieberegisters zum Eingang vor. Aus Letzterem ergibt sich eine unendliche Einflusslänge. Das Eingangssignal ist wie bei allen systematischen Kodierern explizit im Codewort enthalten. RSC-Kodierer sind aufgrund ihrer Verwendung in Turbo-Kodes von großer Bedeutung.

Die Generatormatrix eines RSC-Kodierers mit einem Eingang wird folgendermaßen angegeben:

$$G = \left( 1, \frac{g_1}{g_0}, \dots, \frac{g_{N-1}}{g_0} \right) \quad (2.8)$$

Zumeist befindet sich an erster Stelle eine 1 und notiert den systematischen Ausgang. Das Polynom  $g_0$  definiert die Rückkopplung des Kodierers. Die Polynome  $g_1$  bis  $g_{N-1}$  stellen die Polynome der nichtsystematischen Ausgänge dar [3]. Zur Definition eines RSC-Kodierers im späteren Programm müssen die oktalen Generatorpolynome der nichtsystematischen Ausgänge sowie der Rekursion angegeben werden. Das Polynom des systematischen Ausganges muss nicht angegeben werden. Abbildung 2.6b zeigt einen rekursiv systematischen Faltungskodierer für die Generatormatrix  $G = \left( 1, \frac{5_8}{7_8} \right)$ .

<sup>3</sup>Recursive Systematic Convolutional Coder

### 2.2.6 Terminierung

Die Terminierung bezeichnet nach der vollständigen Kodierung einer Nachricht das Zurückkehren des Kodierers in den Nullzustand. Dazu wird das Schieberegister mit  $M$  0-Bits befüllt. Die Terminierung wirkt sich positiv auf die Fehlerkorrekturfähigkeit bei der Dekodierung aus, da der Endzustand im Trellis bekannt ist. Jedoch geschieht dies auf Kosten der Koderate  $R$ , die durch die Terminierung sinkt. Die Koderate des terminierten Kodes  $R_t$  berechnet sich für eine nicht terminierte Nachricht  $\mathbf{u} = (u_1, u_2, \dots, u_k)$  wie folgt:

$$R_t = \frac{k}{M + k} R \quad (2.9)$$

Für lange Nachrichten ist  $R_t \approx R$  und kann daher vernachlässigt werden.

**Beispiel 5.** Gegeben sei die Nachricht  $\mathbf{u} = (110100)$  aus Beispiel 2. Die Kodierung der terminierten Nachricht ist in Abbildung 2.7 zu sehen und ergibt das Kodewort  $\mathbf{v} = (11\ 01\ 01\ 00\ 10\ 11\ 00\ 00)$ .

### 2.2.7 Punktierung

Zur Erhöhung der Koderate  $R$  eines Kodes gibt es die Möglichkeit der Punktierung. Dabei werden bestimmte Bits vor der Übertragung anhand der sogenannten *Punktierungsmatrix*  $P \in \{0, 1\}^{N \times \frac{p}{N}}$  gestrichen, wobei  $p$  als Periode der Punktierungsmatrix bezeichnet wird und  $P$  spaltenweise durchlaufen wird. Das Gewicht der Matrix  $w(P)$  entspricht der Anzahl nicht punktierter Kodebits je Periode, d.h. der Anzahl an 1 in  $P$ . [5, S. 218]

Die Koderate des punktierten Kodes  $R_p$  berechnet sich wie folgt:

$$R_p = \frac{p}{w(P)} R \quad (2.10)$$

Vor der Dekodierung erfolgt die Depunktierung, d.h. die punktierten Kodebits werden wieder eingefügt. Bei der soft decision Dekodierung wird der Signalwert 0, also genau zwischen den eigentlichen Signalwerten +1 und -1, an den zuvor punktierten Stellen eingefügt. Bei der hard decision Dekodierung wird das Bit 0 eingefügt.

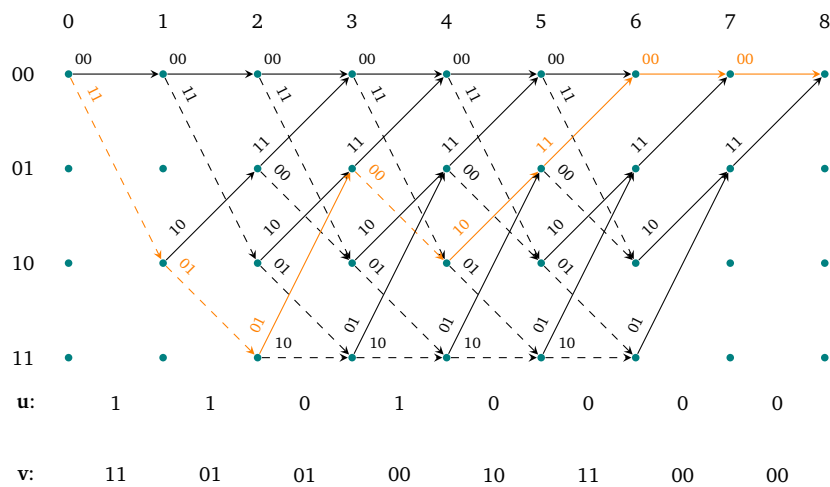


Abbildung 2.7 – Trellis für die terminierte Kodierung aus Beispiel 5.

**Beispiel 6.** Gegeben sei das Kodewort  $\mathbf{v} = (11\ 01\ 01\ 00\ 10\ 11)$  aus Beispiel 2 und die Punktierungsmatrix

$$P = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Nach der Punktierung ergibt sich das Kodewort  $\mathbf{v}_p = (11 * 10 * 00 * 01 *) = (11100001)$ . Die Koderate des punktierten Codes beträgt  $R_p = \frac{6}{4} \cdot \frac{1}{2} = \frac{3}{4}$ .

---

## Kapitel 3

# Verwendete Technologien

---

Dieses Kapitel setzt sich folgendermaßen zusammen: Kapitel 3.1 behandelt die Programmiersprache R und die verwendete Entwicklungsumgebung RStudio. In Kapitel 3.2 werden die Möglichkeiten der Einbindung von C/C++-Code, v.a. mithilfe des Pakets Rcpp, beschrieben. Schließlich wird in Kapitel 3.3 auf die Erstellung dynamischer Dokumente und Visualisierungen mittels RMarkdown,  $\text{\LaTeX}$  und TikZ eingegangen.

### 3.1 R, RStudio, Pakete

R ist eine, im Jahre 1992 entwickelte, schwach und dynamisch typisierte Programmiersprache, die vor allem in der Statistik für die Analyse von großen Datenmengen Anwendung findet. Ein weiteres Motiv für die Verwendung von R sind die vielseitigen Möglichkeiten, bei gleichzeitig einfacher Handhabung, graphischer Darstellungen großer Datenmengen. R-Code wird nicht kompiliert, sondern nur interpretiert und ist daher plattformübergreifend verwendbar. Datentypen müssen zur Übersetzungszeit nicht bekannt sein. Die Typüberprüfung findet zur Laufzeit statt. Diese Eigenschaft erschwert das Finden von Fehlern im Code erheblich.

Der Funktionsumfang der Sprache kann durch sogenannte Pakete erweitert werden. Bei der Installation von R sind die wichtigsten Pakete inkludiert. Über Repositories wie CRAN<sup>4</sup> oder GitHub sind über 8000 zusätzliche Pakete (Stand: Mai 2016) für die verschiedensten Anwendungsbereiche verfügbar.

---

<sup>4</sup>The Comprehensive R Archive Network: <https://cran.r-project.org/>

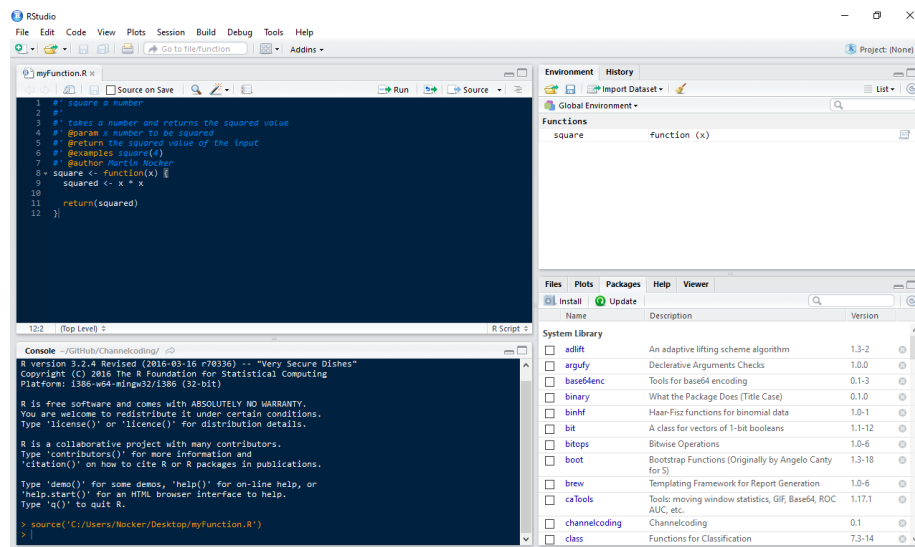


Abbildung 3.1 – RStudio Standardansicht

Diese Vielfalt an Paketen ist ein Grund für den Erfolg von R [7]. Pakete werden laufend aktualisiert und verbessert. Selbst entwickelte Pakete können via CRAN für andere Entwickler veröffentlicht werden, müssen jedoch strenge Auflagen zur Aufrechterhaltung der Konsistenz bei Inhalt, Form und Dokumentation der Pakete einhalten [4].

Ein wichtiges Paket welches im Rahmen dieser Arbeit verwendet wurde ist roxygen. Mithilfe dieses Pakets wird, ähnlich wie JavaDoc für Java, durch spezielle Kommentare und Annotations überhalb der Paketfunktionen automatisch die Paketdokumentation erstellt. Die roxygen-Kommentare der Paketfunktionen, die für wartbaren Code ohnehin unabdingbar sind, sind für den Entwickler erheblich angenehmer als die Paketdokumentation von Hand zu schreiben. Roxygen-Kommentare werden durch das Kommentarsymbol `#'` am Zeilenbeginn eingeleitet. Zu den wichtigsten Annotations gehören jene für die Beschreibung der Parameter (`@param`) und Rückgabewerte (`@return`) sowie Beispiele zur Ausführung der Funktion (`@examples`). Weiters wird über die `@export` Annotation geregelt welche Funktionen nach Auslieferung des Pakets von außen aufrufbar sind.

Ein weiteres hilfreiches Paket ist das devtools-Paket. Das Paket stellt Funktionen für die Erstellung (Build) von Paketen zur Verfügung und beschleunigt

so den Build-Workflow für den Entwickler.

RStudio ist eine freie und open-source Entwicklungsumgebung für R. RStudio verfügt über alle notwendigen Funktionalitäten für die Softwareentwicklung mit R und bietet darüber hinaus Funktionen für eine vereinfachte Entwicklung von R-Paketen an. Abbildung 3.1 zeigt die Version 0.99.893.

## 3.2 C++, Rcpp

Die Vorteile von R wie die einfache Analyse von Datenmengen kommen mit einem Nachteil: R ist keine schnelle Sprache. Typische Flaschenhälse sind Schleifen und rekursive Funktionen. Die Performance kann in solchen Fällen durch Auslagern von Funktionen und Algorithmen in C oder C++ erheblich verbessert werden, da der Code kompiliert und somit optimiert werden kann anstatt nur interpretiert zu werden.

R bietet drei Möglichkeiten C/C++-Code aufzurufen:

- .C-Schnittstelle
- .Call-Schnittstelle
- Rcpp-Paket

Die .C-Schnittstelle ist die einfachste Variante C-Code auszuführen, jedoch auch jene mit den größten Einschränkungen. Im C-Code sind keinerlei R-Datentypen oder R-Funktionen bekannt. Alle Argumente sowie der Rückgabewert müssen als Zeiger in der Parameterliste übergeben werden deren Speicher vor dem Aufruf reserviert werden muss.

Bei der .Call-Schnittstelle handelt es sich um eine Erweiterung der .C-Schnittstelle. Die Implementierung ist komplexer, dafür sind R-Datentypen verfügbar und es gibt die Möglichkeit eines Rückgabewerts mittels `return` Statement. [7]

Sowohl bei der .C-Schnittstelle als auch bei der .Call-Schnittstelle muss der C-Code vor dem Aufruf per Hand kompiliert und in der R Session geladen werden. Das Rcpp-Paket ermöglicht die Verwendung von C++-Code ohne diesen Aufwand. Im C++-Code stehen R-Datentypen wie Vektoren, Matrizen oder

Listen ohne komplizierte Syntax zur Verfügung. Die Funktionsaufrufe sehen, im Gegensatz zu den C-Schnittstellen, aus wie normale R-Funktionsaufrufe und machen dadurch den Code erheblich lesbarer. Weiters stehen Vektorfunktionen zur Verfügung, d.h. eine auf einen Vektor angewandte Funktion wird auf jedes Vektorelement ausgeführt und erspart somit beispielsweise eine Schleife. Bei der Entwicklung eines eigenen Pakets ist es bei der Verwendung des Rcpp-Pakets zusammen mit RStudio sehr einfach C++-Code zu integrieren. Durch all diese Vorteile ist das Rcpp-Paket die zu wählende Schnittstelle. Während der Paketerzeugung kompiliert RStudio automatisch alle C++-Dateien und erstellt automatisch Wrapper-Funktionen, die den Zugriff auf die Funktionen erleichtern.

Die C++-Datei muss mit folgenden Zeilen starten:

---

```
#include <Rcpp.h>
using namespace Rcpp;
```

---

Sowie jede Funktion, die in R verfügbar sein soll muss folgenden Präfix erhalten:

---

```
// [[Rcpp::export]]
```

---

Die genaue Verwendung des Rcpp-Pakets ist in [6] beschrieben.

### 3.3 RMarkdown, $\text{\LaTeX}$ , TikZ

Zur Erstellung von dynamischen Dokumenten wird das Paket RMarkdown verwendet. Durch die Kombination der Syntax von Markdown, R, und  $\text{\LaTeX}$  ergibt sich ein flexibles und einfaches Werkzeug. Die unterstützten Ausgabeformate beinhalten u.a. HTML, PDF, MS Word und Beamer (Präsentationen).

Abbildung 3.2 zeigt den Workflow für die Generierung eines dynamischen Dokuments mittels RMarkdown. Der Markdown-, R- und  $\text{\LaTeX}$ -Code wird zusammen mit dem gewünschten Ausgabeformat, wobei mehrere Angaben möglich sind, in die RMarkdown-Datei (Dateiendung .rmd) geschrieben. Die RMD-Datei wird dem knitr-Paket übergeben, welches den R-Code ausführt und eine neue Markdown-Datei (Dateiendung .md) erstellt, die den R-Code und dessen Ergebnisse beinhaltet. Die erzeugte Markdown-Datei wird von



Abbildung 3.2 – RMarkdown Überblick, Quelle: [1]

pandoc weiterverarbeitet, was für die Erstellung des endgültigen Dokuments im gewünschten Format zuständig ist. Bei der Verwendung von RStudio ist pandoc automatisch verfügbar. Den eben beschriebenen Ablauf kapselt das RMarkdown-Paket in einen einzigen `render`-Funktionsaufruf.

Für die Erzeugung dynamischer Grafiken wird das  $\text{\LaTeX}$ -Sprachpaket TikZ verwendet. Mithilfe des Dokumenttyps Beamer in  $\text{\LaTeX}$  lassen sich Präsentationen erstellen. Die Grafiken und Inhalte können dadurch dynamisch ein- oder ausgeblendet werden oder farblich hervorgehoben werden. Dies ist insofern wertvoll, da Informationen, die Schritt für Schritt vervollständigt werden, es dem Benutzer leichter machen den Ablauf nachzuvollziehen. Damit Benutzer des R-Pakets dieser Arbeit die Prinzipien von Faltungskodes besser verstehen können, werden die Visualisierungen der Kodierung und Dekodierung sukzessive eingeblendet.



---

## Kapitel 4

# Implementierung

---

Dieses Kapitel gibt einen Einblick in die Konzepte der Implementierung. Als Einstiegspunkt stand eine Referenzimplementierung<sup>5</sup> zur Verfügung, die den Dekodier-Algorithmus für Turbo-Kodes beinhaltet, jedoch für ein konkretes Beispiel. Dieser musste angepasst werden um für allgemeine Faltungskodes verwendbar zu sein.

Kapitel 4.1 beinhaltet den Entwurf der Faltungskodierer-Datenstruktur. [TODO: Fertigstellung]

### 4.1 Faltungskodierer

Ein Faltungskodierer ist gegeben durch

- $N$ : Anzahl an Ausgangsbits je Eingangsbit,
- $M$ : Länge des Schieberegisters,
- $G$ : Vektor von Generatorpolynomen.

Die Angabe von  $M$  ist hier redundant, jedoch Teil der Benutzereingabe zur Generierung eines Faltungskodierers, welche durch [3] inspiriert wurde.

Zur leichteren Implementierung der Kodierung und Dekodierung wird die Kodierer-Datenstruktur um folgende Elemente erweitert:

- eine *Zustandsübergangsmatrix*, die angibt, in welchen Zustand der Kodierer bei einem Eingangsbit wechselt,

---

<sup>5</sup>[http://vashe.org/turbo/turbo\\_example.c](http://vashe.org/turbo/turbo_example.c) (01.06.2016)

- eine *inverse Zustandsübergangsmatrix*, die angibt, aus welchem Zustand der Kodierer bei einem Eingangsbit kommt,
- eine *Ausgabematrix*, die angibt, welche Kodebits der Kodierer bei einem Eingangsbit in einem bestimmten Zustand ausgibt,
- ein Flag zur Markierung rekursiver systematischer Kodierer (RSC, siehe Kapitel 2.2.5),
- ein *Terminierungsvektor* die für rekursiver systematische Kodierer angibt, ob ein Eingangsbit 0 oder 1 in einem bestimmten Zustand für die Terminierung zu verwenden ist.

Die Implementierung der Matrizen wurde aus der Referenzimplementierung übernommen, musste jedoch erweitert werden, um für allgemeine Faltungskodes verwendbar zu sein. Für alle gilt, die Anzahl an Zeilen entspricht der Anzahl an Zuständen  $2^M$ . Der Zeilenindex entspricht dem Zustand. Die Zustandsübergangsmatrix sowie die Ausgabematrix besitzen jeweils zwei Spalten. Je eine Spalte steht für ein Eingangsbit (0 oder 1), wobei der Spaltenindex dem Eingangsbit entspricht. Die inverse Zustandsübergangsmatrix benötigt eine dritte Spalte. Für viele Kodierer (v.a. nicht-rekursive) tritt der Fall ein, dass nur durch *ein bestimmtes* Eingangsbit in einen bestimmten Zustand gewechselt werden kann. Sei ein Zustand bspw. nur durch das Eingangsbit 0 erreichbar, so bedeutet das, dass es für diesen Zustand mit dem Bit 0 *zwei* Vorgängerzustände gibt, für ein Eingangsbit 1 jedoch keinen Vorgänger. Diese zweite Möglichkeit wird in der dritten Spalte gespeichert.

Der Terminierungsvektor ist für nicht-rekursive Kodierer nicht notwendig, da ein Kode eines solchen Kodierers immer mit  $M$  0-Bits terminiert wird. Bei einem rekursiven Kodierer ist es nicht trivial zu sagen mit welchem Eingangsbit in einem bestimmten Zustand terminiert wird, um den Kodierer in den Nullzustand zu bringen. Dies hängt von der Definition des Rekursionspolynoms ab. Der Terminierungsvektor wird bei der Erzeugung rekursiver Kodierer berechnet.

Bei der Erzeugung von Faltungskodierern ist zu prüfen ob es sich um einen katastrophalen Kodierer handelt. RSC-Kodierer sind, wie in Kapitel 2.2.4 beschrieben, nicht zu prüfen. Zur Prüfung wird nach dem Theorem von

Massey-Sain (siehe Kapitel 2.2.3) der größte gemeinsame Teiler der Generatorpolynome berechnet. Die Berechnung des größten gemeinsamen Teilers wurde mithilfe des euklidischen Algorithmus implementiert. Sowohl der euklidische Algorithmus als auch die dafür notwendige binäre Polynomdivision wird an eine C++-Funktion delegiert.

## 4.2 Kodierung

Bei Faltungskodes stellt die Kodierung den bei Weitem einfacheren Teil dar. Es muss lediglich jedes Bit der zu kodierenden Nachricht zusammen mit dem aktuellen Zustand, der nach jedem Bit mithilfe der Zustandsübergangsmatrix aktualisiert wird, auf die Ausgabematrix angewendet werden. Die Terminierung funktioniert analog, einzig das zu kodierende Bit muss ermittelt werden. Für RSC-Kodierer muss im Terminierungsvektor nachgeschaut werden, andernfalls ist das Terminierungsbit immer 0. Abgeschlossen wird die Kodierung mit dem Abbilden der Kodebits 0 bzw. 1 auf die Signalwerte +1 bzw. -1 nach Gleichung (2.1). Algorithmus 1 zeigt den Kodierungsalgorithmus.

---

```
1: state = 0, code = result = " "  
2: for each bit in message do  
3:   output = output.matrix[state][bit]  
4:   code = concat(code, output)  
5:   state = state.transition.matrix[state][bit]  
6: end for  
7: if terminate code then  
8:   for  $i = 0$  to  $M - 1$  do  
9:     termination.bit = rsc-coder ? termination.vector[state] : 0  
10:    output = output.matrix[state][termination.bit]  
11:    code = concat(code, output)  
12:    state = state.transition.matrix[state][termination.bit]  
13:  end for  
14: end if  
15: for each bit in code do  
16:   signal =  $1 - 2\text{bit}$   
17:   result = concat(result, signal)  
18: end for  
19: return result
```

---

Algorithmus 1 – Faltungskodierung

## 4.3 Dekodierung

Die Dekodierung stellt den wesentlich komplexeren Teil der Faltungskodes dar.

---

```

1:  $NUM\_STATES = 2^M$ 
2: for  $t = 1$  to  $length(message)$  do
3:   for  $s = 0$  to  $NUM\_STATES - 1$  do
4:      $m_1 = metric[t-1][prev.state1] + \delta_1$ 
5:      $m_2 = metric[t-1][prev.state2] + \delta_2$ 
6:      $metric[t][s] = \min(m_1, m_2)$ 
7:      $survivor.bit = xyz(0, 1, \min(\min(m_1, m_2)))$ 
8:   end for
9: end for
10: return result

```

---

**Algorithmus 2** – Faltungsdekodierung

## 4.4 Rauschen

Um auch zeigen zu können, dass die Dekodierung auch tatsächlich für verrauschte Signale funktioniert, benötigt es eine Funktion, die die Übertragung einer Nachricht über einen verrauschten Kanal simuliert, d.h. das Signal mit Rauschen überlagert. Zum Signal soll ein additives weißes gaußsches Rauschen (AWGR oder AWGN<sup>6</sup>) addiert werden um dieses zu verfälschen. [apply noise quelle] stellt eine alternative Implementierung zur eingebauten AWGN-Funktion in Matlab vor. Die Implementierung wurde übernommen bzw. nach R übersetzt. Durch die Möglichkeit das Signal-Rausch-Verhältnis über einen Parameter zu steuern, können verschiedene Übertragungskanäle simuliert und Nachrichten somit verschieden stark verrauscht werden. Der Benutzer kann dadurch herausfinden, ab wann eine Nachricht zu viel Rauschen enthält, um sie korrekt dekodieren zu können. Weiters kann nach mehrfacher Ausführung auf Fehlermuster geschlossen werden, mit denen die Dekodierung gut bzw. schlecht umgehen kann. Durch Versuche mit anderen Kanalkodierungs-Methoden können Vergleiche mit diesen angestellt werden. Die genannten Punkte helfen dem Benutzer sein Verständnis für Faltungskodes noch besser zu stärken.

---

<sup>6</sup>additive white Gaussian noise

## 4.5 Punktierung

Punktierung leicht, jedoch Depunktierung vor der Dekodierung nicht trivial.

---

## Kapitel 5

# R-Paket Schnittstelle

---

In diesem Kapitel wird die Schnittstelle für den Benutzer erläutert. Kapitel 5.1 listet Funktionen zur Erzeugung von Faltungskodierern, der Kodierung, Dekodierung und Simulation von Faltungskodes. Kapitel 5.2 beinhaltet Hilfsfunktionen für Faltungskodes. Schließlich beschreibt Kapitel 5.3 weitere nützliche Funktionen der Kanalkodierung.

### 5.1 Faltungskodierung

#### 5.1.1 ConvGenerateEncoder

ConvGenerateEncoder
<pre>ConvGenerateEncoder(N, M, generators)</pre>
Erzeugt einen Faltungskodierer für nichtrekursive Faltungskodes.
<b>Argumente:</b> N - Anzahl an Ausgangssymbole je Eingangssymbol. M - Länge des Schieberegisters des Kodierers. generators - Vektor der N oktale Generatorpolynome enthält (ein Polynom je Ausgangssymbol, siehe Kapitel 2.2.1).
<b>Rückgabewert:</b>

Faltungskodierer, abgebildet als Liste mit folgenden Feldern:

- *N*: Anzahl an Ausgangssymbole je Eingangssymbol
- *M*: Länge des Schieberegisters des Kodierers
- *generators*: Generatorpolynomvektor
- *next.state*: Zustandsübergangsmatrix
- *prev.state*: inverse Zustandsübergangsmatrix
- *output*: Ausgabematrix
- *rsc*: RSC-Flag (FALSE)
- *termination*: Terminierungsvektor (`logical(0)`)

Funktion 5.1 – ConvGenerateEncoder

### 5.1.2 ConvGenerateRscEncoder

#### ConvGenerateRscEncoder

`ConvGenerateRscEncoder(N, M, generators)`

Erzeugt einen Faltungskodierer für rekursiv systematische Faltungskodes (RSC-Kodierer).

**Argumente:**

*N* - Anzahl an Ausgangssymbole je Eingangssymbol.

*M* - Länge des Schieberegisters des Kodierers.

*generators* - Vektor der oktale Generatorpolynome enthält (ein Polynom je nicht-systematischen Ausgang und ein Polynom für die Rekursion, siehe Kapitel 2.2.5).

**Rückgabewert:**



Faltungskodierer, abgebildet als Liste mit folgenden Feldern:

- *N*: Anzahl an Ausgangssymbole je Eingangssymbol
- *M*: Länge des Schieberegisters des Kodierers
- *generators*: Generatorpolynomvektor
- *next.state*: Zustandsübergangsmatrix
- *prev.state*: inverse Zustandsübergangsmatrix
- *output*: Ausgabematrix
- *rsc*: RSC-Flag (TRUE)
- *termination*: Terminierungsvektor

Funktion 5.2 – ConvGenerateRscEncoder

### 5.1.3 ConvEncode

#### ConvEncode

```
ConvEncode(message, conv.encoder, terminate,  
punctuation.matrix, visualize)
```

Erzeugt einen Faltungskode aus einer unkodierten Nachricht.

#### Argumente:

*message* - Nachricht die kodiert wird.

*conv.encoder* - Faltungskodierer der für die Kodierung verwendet wird.

*terminate* - Markiert ob der Kode terminiert werden soll. Standard: TRUE

*punctuation.matrix* - Wenn ungleich NULL wird die kodierte Nachricht mit der Punktierungsmatrix punktiert. Standard: NULL

*visualize* - Wenn TRUE wird ein PDF-Bericht der Kodierung erstellt. Standard: FALSE

#### Rückgabewert:

Die kodierte Nachricht mit den Signalwerten +1 und -1 welche die Bits 0 und 1 darstellen. Falls punktiert wurde Liste mit dem Originalkode (nicht punktiert) und dem punktiertem Kode.

Funktion 5.3 – ConvEncode

### 5.1.4 ConvDecodeSoft

#### ConvDecodeSoft

```
ConvDecodeSoft(code, conv.encoder, terminate,  
punctuation.matrix, visualize)
```

Dekodiert einen Faltungskode mittels soft decision Dekodierung.

**Argumente:**

code - Faltungskode der dekodiert wird.

conv.encoder - Faltungskodierer der für die Kodierung verwendet wurde.

terminate - Markiert ob der Kode terminiert ist. Standard: TRUE

punctuation.matrix - Wenn ungleich NULL wird der Kode vor der Dekodierung depunktiert. Standard: NULL

visualize - Wenn TRUE wird ein PDF-Bericht der Dekodierung erstellt. Standard: FALSE

**Rückgabewert:**

Liste die die Soft-Werte und Hard-Werte der dekodierten Nachricht enthält.

Funktion 5.4 – ConvDecodeSoft

### 5.1.5 ConvDecodeHard

#### ConvDecodeHard

```
ConvDecodeHard(code, conv.encoder, terminate,  
punctuation.matrix, visualize)
```

Dekodiert einen Faltungskode mittels hard decision Dekodierung.

**Argumente:**

code - Faltungskode der dekodiert wird.

conv.encoder - Faltungskodierer der für die Kodierung verwendet wurde.

terminate - Markiert ob der Kode terminiert ist. Standard: TRUE

`punctuation.matrix` - Wenn ungleich NULL wird der Kode vor der Dekodierung depunktiert. Standard: NULL

`visualize` - Wenn TRUE wird ein PDF-Bericht der Dekodierung erstellt. Standard: FALSE

**Rückgabewert:**

Vektor der dekodierten Nachricht.

**Funktion 5.5 – ConvDecodeHard**

### 5.1.6 ConvSimulation

#### ConvSimulation

```
ConvSimulation(conv.coder, msg.length, min.db, max.db,
db.interval, iterations.per.db, punctuation.matrix,
visualize)
```

Simulation einer Faltungskodierung und -dekodierung nach einer Übertragung über einen verrauschten Kanal mit verschiedenen Signal-Rausch-Verhältnissen (SNR).

**Argumente:**

`conv.coder` - Faltungskodierer der für die Simulation verwendet wird. Kann mittels `ConvGenerateEncoder` oder `ConvGenerateRscEncoder` erzeugt werden.

`msg.length` - Nachrichtenlänge der zufällig generierten Nachrichten. Standard: 100

`min.db` - Untergrenze der getesteten SNR. Standard: 0.1

`max.db` - Obergrenze der getesteten SNR. Standard: 2.0

`db.interval` - Schrittweite zwischen zwei getesteten SNR. Standard: 0.1

`iterations.per.db` - Anzahl der Iterationen (Kodieren und Dekodieren) je SNR. Standard: 100

`punctuation.matrix` - Wenn ungleich NULL wird die kodierte Nachricht punktiert. Kann mittels `ConvGetPunctuationMatrix` erzeugt werden. Standard: NULL

`visualize` - Markiert ob ein Simulationsbericht erzeugt wird. Standard: FALSE

**Rückgabewert:**

Dataframe das die Bitfehlerrate für die getesteten Signal-Rausch-Verhältnisse beinhaltet.

Funktion 5.6 – ConvSimulation

## 5.2 Hilfsfunktionen

### 5.2.1 ConvGetPunctuationMatrix

#### ConvGetPunctuationMatrix

`ConvGetPunctuationMatrix(punctuation.vector, conv.coder)`

Erzeugt aus dem gegebenen Punktierungsvektor und Faltungskodierer eine Punktierungsmatrix.

**Argumente:**

`punctuation.vector` - Vektor der die Punktierungsinformation enthält welche in eine Punktierungsmatrix transformiert wird.

`conv.coder` - Faltungskodierer der für die Matrixdimension verwendet wird.

**Rückgabewert:**

Punktierungsmatrix die für `ConvEncode`, `ConvDecodeSoft`, `ConvDecodeHard` und `ConvSimulation` verwendet werden kann.

Funktion 5.7 – ConvGetPunctuationMatrix

### 5.2.2 ConvOpenPDF

#### ConvOpenPDF

`ConvOpenPDF(encode, punctured, simulation)`

Öffnet die mit ConvEncode, ConvDecodeSoft, ConvDecodeHard und ConvSimulation erzeugten PDF-Berichte.

**Argumente:**

encode - Markiert ob Kodierungsbericht (TRUE) oder Dekodierungsbericht (FALSE) geöffnet wird. Standard: TRUE

punctured - Markiert ob Berichte mit Punktierung geöffnet werden. Standard: FALSE

simulation - Markiert ob Simulationsbericht geöffnet wird. Standard: FALSE

Funktion 5.8 – ConvOpenPDF

## 5.3 Kanalkodierung

### 5.3.1 ApplyNoise

**ApplyNoise**

ApplyNoise(msg, SNR.db, binary)

Verrauscht ein Signal basierend auf dem AWGN Modell (Additive White Gaussian Noise), dem Standardmodell für die Simulation eines Übertragungskanals.

**Argumente:**

msg - Die zu verrauschende Nachricht

SNR.db - Signal-Rausch-Verhältnis (signal noise ratio) des Übertragungskanals in dB. Standard: 3.0

binary - Blockcode Parameter. Nicht zu verwenden. Standard: FALSE

**Rückgabewert:**

Verrauschtes Signal.

Funktion 5.9 – ApplyNoise

### 5.3.2 ChannelcodingSimulation

ChannelcodingSimulation
<pre>ChannelcodingSimulation(msg.length, min.db, max.db, db.interval, iterations.per.db, turbo.decode.iterations, visualize)</pre>
<p>Simulation von Block-, Faltungs- und Turbo-Kodes und Vergleich ihrer Bitfehler-raten bei unterschiedlichen SNR.</p>
<p><b>Argumente:</b></p> <p>msg.length - Nachrichtenlänge der zufällig generierten Nachrichten. Standard: 100</p> <p>min.db - Untergrenze der getesteten SNR. Standard: 0.1</p> <p>max.db - Obergrenze der getesteten SNR. Standard: 2.0</p> <p>db.interval - Schrittweite zwischen zwei getesteten SNR. Standard: 0.1</p> <p>iterations.per.db - Anzahl der Iterationen (Kodieren und Dekodieren) je SNR. Standard: 100</p> <p>turbo.decode.iterations - Anzahl der Iterationen bei der Turbo-Dekodierung. Standard: 5</p> <p>visualize - Wenn TRUE wird ein PDF-Bericht erstellt. Standard: FALSE</p>
<p><b>Rückgabewert:</b></p> <p>Dataframe das alle Simulationsergebnisse der 3 Kodierungsverfahren beinhaltet.</p>

Funktion 5.10 – ChannelcodingSimulation

### 5.3.3 PlotSimulationData

PlotSimulationData
<pre>PlotSimulationData(...)</pre>

Stellt die mitgegebenen Dataframes bzw. die Bitfehlerraten für verschiedene Signal-Rausch-Verhältnisse in einem Diagramm dar. Dataframes können mittels `ConvSimulation`, `TurboSimulation` und `BlockSimulation` erzeugt werden.

**Argumente:**

. . . - Dataframes die mit den Simulationsfunktionen erzeugt wurden.

**Funktion 5.11** – `PlotSimulationData`

---

## Kapitel 6

# Visualisierung

---

Wird der `visualize` Parameter bei der Ausführung einer R-Paket Funktion auf `TRUE` gesetzt, wird ein RMarkdown-Skript ausgeführt. Dieses generiert eine Präsentation vom  $\text{\LaTeX}$ -Dokumenttyp Beamer mit Informationen und Visualisierungen zur ausgeführten Funktion.

Die erzeugten Präsentationen liegen innerhalb des Programmverzeichnisses von R im Installationsordner des Pakets. Dort gibt es den *pdf* Order in dem sich die Dokumente befinden. Möchte man die Dokumente vom RStudio aus öffnen, anstatt die Dokumente in der Verzeichnisstruktur suchen zu müssen, kann die `ConvOpenPDF`-Funktion (siehe Kapitel 5.2.2) verwendet werden. Möchte man die Dokumente zu einem späteren Zeitpunkt erneut einsehen wird empfohlen eine Kopie der Datei abzuspeichern, da bei einer erneuten Ausführung der gleichen Funktion inklusive Visualisierung das zuvor erzeugte Dokument im *pdf* Ordner überschrieben wird. Die vom RMarkdown-Skript erzeugten  $\text{\LaTeX}$ -Dateien aus denen die Beamer Präsentationen generiert werden liegen ebenfalls im selben Verzeichnis.

Bei den folgenden Beispielen der Visualisierungen wurde stets punktiert, da ohne Punktierung die Informationen in den Präsentationen dazu fehlen und somit auch nicht erläutert werden können.

Der weitere Kapitelinhalt setzt sich folgendermaßen zusammen: [TODO]



## Faltungskodierer Informationen

- ▶ Nicht-Rekursiver Kodierer

- ▶ Anzahl von Ausgängen :

$$N = 2$$

- ▶ Länge des Schieberegisters :

$$M = 2$$

- ▶ Generatoren :

$$(7_8, 5_8) = \begin{pmatrix} 111 \\ 101 \end{pmatrix}$$

- ▶ Punktierungs-Matrix :

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

- ▶ Koderate :

$$\frac{3}{4}$$

## Faltungskodierer Matrix : Nächster Zustand

	Bit 0	Bit 1
Zustand 0	0	2
Zustand 1	0	2
Zustand 2	1	3
Zustand 3	1	3

(a) Folie mit Kodierer-Kennwerten

(b) Folie mit Zustandsübergangsmatrix

**Abbildung 6.1** – Folien mit allgemeinen Informationen zum Faltungskodierer

## 6.1 Kodierung

Bei der Kodierung befinden sich auf den ersten Folien allgemeine Informationen zum verwendeten Faltungskodierer. Abbildung 6.1a zeigt die Folie mit den Kennwerten des Faltungskodierers: Anzahl an Ausgänge  $N$ , Länge des Schieberegisters  $M$ , Generatorpolynome, Punktierungsmatrix und Koderate. Auf den nächsten beiden Folien werden die Zustandsübergangsmatrix und die Ausgabematrix abgebildet. In Abbildung 6.1b ist die Folie der Zustandsübergangsmatrix zu sehen. Analog dazu wird die Ausgabematrix auf der nächsten Folie veranschaulicht. Diese wird aus Platzgründen hier nicht abgebildet.

Daraufhin folgt die Visualisierung der Kodierung. Dabei wird die zu kodierende Nachricht Bit für Bit auf einer eigenen Folie verarbeitet. Abbildung 6.2 zeigt die ersten beiden Schritte sowie den letzten Schritt der Kodierung. In Abbildung 6.2a ist die Folie vor der Kodierung des ersten Bits zu sehen. Zunächst ist die zu kodierende Nachricht (Input), das Zustandsübergangsdiagramm sowie eine noch nicht befüllte Kodierungstabelle zu sehen. Das Kodewort wird auf den folgenden Folien Schritt für Schritt erarbeitet. Durch diese Herangehensweise ist die Kodierung für den Benutzer einfach nachzuvollziehen. Abbildung 6.2b zeigt den nächsten Schritt der Kodierung. Das erste Bit des Inputs, der aktuelle Zustand, Folgezustand sowie der resultierende Output wird in eine neue Zeile der Kodierungstabelle geschrieben. Der aktuelle Zustand sowie der entsprechende Übergang werden im Diagramm farblich hervorgehoben, um die Kodierung auch im Zustandsdiagramm verfolgen zu können.

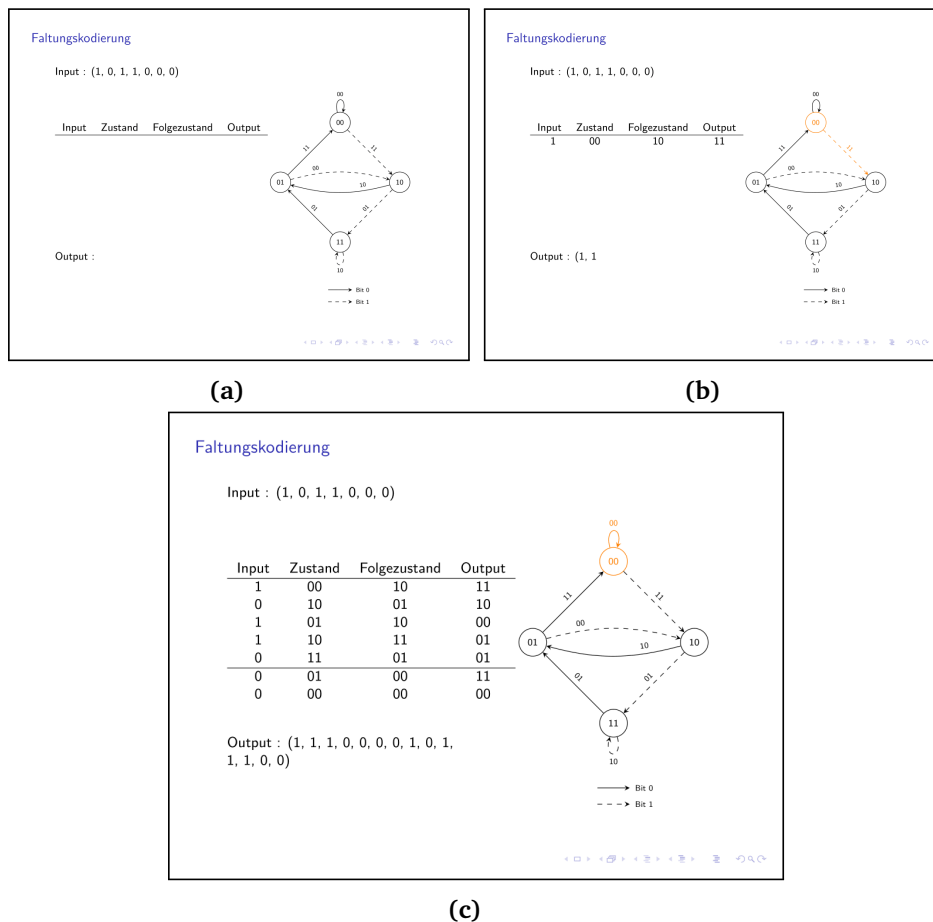


Abbildung 6.2 – Folien der Kodierung

Der Output wird auch unterhalb der Tabelle eingetragen und wächst mit jedem Schritt bis schlussendlich die gesamte Nachricht kodiert wurde. Die Visualisierung am Ende der Kodierung ist in Abbildung 6.2c zu sehen. Die Bits unterhalb der horizontalen Trennlinie stellen die Terminierungsbits dar.

Da die Kodierungsfunktion nicht die Bitwerte des Kodeworts zurückliefert sondern die Signalwerte (für eine Übertragung über einen Kanal) wird auf einer weiteren Folie, wie in Abbildung 6.3 zu sehen, die Abbildung der Kodebits zu den Signalpegeln (nach Gleichung 2.1) dargestellt.

Abbildung 6.4 zeigt die Folie der Punktierung. Auf dieser wird die Punktierung des Signals, d.h. das Entfernen von Signalwerten (definiert durch die Punktierungsmatrix) dargestellt. Dabei wird neben dem originalen Signal und

Output : Kode zu Signal

► Kode : (1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0)

$$\text{Signal}(\text{bit}) = \begin{cases} +1 & \text{wenn bit} = 0 \\ -1 & \text{wenn bit} = 1 \end{cases}$$

► Signal : (-1, -1, -1, 1, 1, 1, 1, -1, 1, -1, -1, -1, 1, 1)

Abbildung 6.3 – Folie mit der Bit zu Signal Abbildung

Punktierung

► Signal : (-1, -1, -1, 1, 1, 1, 1, -1, 1, -1, -1, -1, 1, 1)

► Punktierungs-Matrix :

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

► Punktiertes Signal : (-1, -1, \*, 1, 1, \*, 1, -1, \*, -1, -1, \*, 1, 1)

► Output : (-1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 1)

Abbildung 6.4 – Folie mit Punktierung

der Punktierungsmatrix das punktierte Signal dargestellt, wobei zunächst die punktierten Signalwerte, d.h. die entfernten Werte, durch Asterisk-Symbole (\*) ersetzt werden. Diese Darstellung dient als visueller Zwischenschritt für das danach folgende tatsächlich punktierte Signal, bei dem die punktierten Werte fehlen, was auch dem Rückgabewert der Funktion entspricht.

## 6.2 Dekodierung

Bei der Dekodierung befinden sich ebenfalls, wie bei der Kodierung, allgemeine Informationen des Faltungskodierers auf den ersten Folien.

Abbildung 6.5 zeigt die Folie der Depunktierung. Auf dieser wird die Depunk-

Depunktierung

- Punktiertes Signal : (-1.4, -1.1, 0, 2, 1.6, -0.7, -1.3, -0.7, 1.1, 1.5)
- Punktierungs-Matrix :
 
$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$
- Depunktiertes Signal : (-1.4, -1.1, 0, 0, 2, 0, 1.6, -0.7, 0, -1.3, -0.7, 0, 1.1, 1.5)

Abbildung 6.5 – Folie mit Depunktierung

tierung des Signals, d.h. das Einfügen des Signalwerts 0 (definiert durch die Punktierungsmatrix), dargestellt. Die eingefügten 0-Werte sind zur leichteren visuellen Erkennung farblich hervorgehoben.

Als Input erhält die Dekodierung das Kodewort als kontinuierliche Signal-

Input : Signal zu Kode

- Signal : (-1.4, -1.1, 0, 0, 2, 0, 1.6, -0.7, 0, -1.3, -0.7, 0, 1.1, 1.5)
- $$\text{Kode}(\text{signal}) = \begin{cases} 0 & \text{wenn signal} \geq 0 \\ 1 & \text{sonst} \end{cases}$$
- Kode : (1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0)

Abbildung 6.6 – Folie mit der Signal zu Bit Abbildung

werte, die möglicherweise durch Anwendung der ApplyNoise Funktion verfälscht worden sind. Die soft decision Dekodierung verwendet zur Deko-

dierung zwar direkt diese Signalwerte, da aber sowohl die hard decision Dekodierung Bitwerte zur Dekodierung verwendet und die Kanten des Trellis mit Bitwerten beschriftet sind, wird der Input, um konsistent zu bleiben, auf Bitwerte abgebildet. Die Folie der Abbildung der Signalwerte auf Bitwerte (nach Gleichung 2.2) ist in Abbildung 6.6 zu sehen.

Anschließend folgt die Visualisierung des Viterbi-Algorithmus mithilfe des

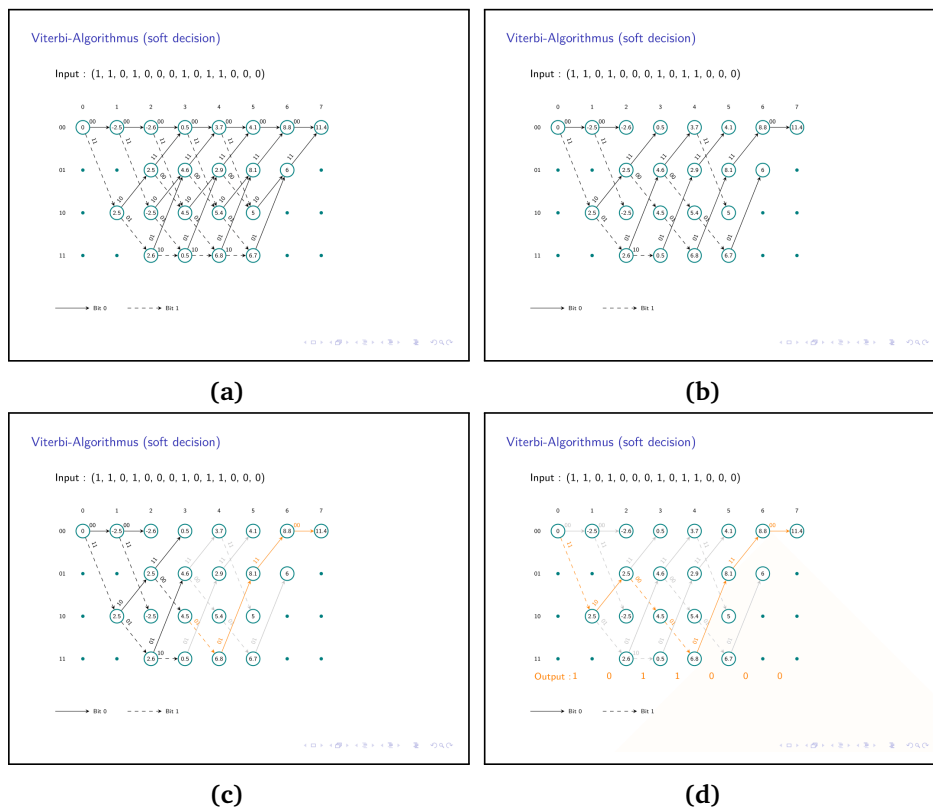


Abbildung 6.7 – Folien mit der soft decision Dekodierung

Trellis-Diagramms. In den farbigen Kreisen befinden sich die Metriken der Pfade die zum jeweiligen Zustand führen. Zunächst werden, zur besseren Übersicht bei großen Diagrammen, jene Pfade entfernt, für die es eine bessere Alternative gibt. D.h. es werden jene Pfade entfernt die bei der soft decision Dekodierung eine niedrigere Metrik bzw. bei der hard decision Dekodierung eine höhere Metrik besitzen. Dieser Schritt ist zwischen den Abbildungen 6.7a und 6.7b zu sehen. Danach erfolgt Schritt für Schritt mittels Backtracking die Rekonstruktion der Nachricht. Der gewählte Pfad beim Backtracking wird

farblich hervorgehoben. Die übrigen Pfade werden ausgegraut. Eine Folie während des Backtrackings wird in Abbildung 6.7c veranschaulicht. Am Ende befindet sich unter dem Trellis-Diagramm die farblich hervorgehobene dekodierte Nachricht, wie in Abbildung 6.7d zu sehen ist. Die einzelnen Zwischenschritte vermitteln dem Benutzer wie der Algorithmus funktioniert und wie sich die dekodierte Nachricht ergibt.

### 6.3 Simulation

Weiters können Berichte der Simulation generiert werden, die die resultierenden Daten u.a. in einem Diagramm darstellen.

---

## Kapitel 7

# Beispiele

---

In diesem Kapitel werden Beispiele zur Verwendung der R-Paket Funktionen der Faltungskodierung veranschaulicht und erklärt. Kapitel 7.1 führt Variablen ein, die über die weiteren Beispiele hinweg verwendet werden. Die Kapitel 7.2 bzw. 7.3 beinhalten die Kodierung und Dekodierung ohne bzw. mit Punktierung. Schließlich werden in Kapitel 7.4 Beispiele für die Ausführung von Simulationen vorgestellt.

### 7.1 Erzeugen von Kodierer und Punktierungsmatrix

Vor den Beispielen der Kodierung und Dekodierung werden Hilfsvariablen definiert, die für die folgenden Beispiele verwendet werden. Variablen wie der Faltungskodierer oder die Punktierungsmatrix müssen jedoch nicht unbedingt bei der Kodierung oder Dekodierung mitgegeben werden. Fehlt bei den Parametern der Kodierer wird der Standard-Faltungskodierer aus Beispiel 1 verwendet. Gibt man keine Punktierungsmatrix mit, so wird das Signal auch nicht punktiert. Die meisten weiteren Parameter (Terminierung, Visualisierung etc.) besitzen auch Standardwerte, die verwendet werden, sollten vom Benutzer keine Werte bestimmt werden. Listing 7.1 zeigt die Erzeugung der Variablen für die weiteren Beispiele. Zunächst wird eine Quellnachricht `input` erzeugt, gefolgt von der Definition des NASA-Standardkodierers [3, S. 90]. Zuletzt wird eine Punktierungsmatrix aus dem mitgegebenen Punktierungsvektor und dem verwendeten Kodierer generiert. Der Kodierer ist notwendig, da die Anzahl der Zeilen der Punktierungsmatrix gleich der Anzahl der Kodiererausgänge  $N$  ist (siehe Kapitel 2.2.7).

---

```

1 input <- c(1,0,1,1,0)
2
3 nasa.encoder <- ConvGenerateEncoder(2, 6, c(171,133))
4
5 p.matrix <- ConvGetPunctuationMatrix(c(1,0,1,1,0,1),
   nasa.encoder)
6 p.matrix
7      [,1] [,2] [,3]
8 [1,]    1    1    0
9 [2,]    0    1    1

```

---

Listing 7.1 – Erzeugung von Kodierer und Punktierungsmatrix

## 7.2 Kodieren und Dekodieren ohne Punktierung

Im folgenden Beispiel wird eine Nachricht kodiert, verfälscht und sowohl mithilfe der soft decision Dekodierung als auch mithilfe der hard decision Dekodierung dekodiert. Listing 7.2 beinhaltet den Code zur Kodierung und

---

```

1 encoded <- ConvEncode(input, nasa.encoder)
2 encoded
3 [1] -1 -1 -1 1 1 1 -1 1 1 -1 1 -1 1 1 1 -1 -1 1
   -1 -1 1 1
4
5 encoded.noisy <- ApplyNoise(encoded, SNR.db = 0.2)
6 round(encoded.noisy, 2)
7 [1] -0.83 -0.75 -0.36 1.31 1.05 0.16 0.37 -0.12 1.55
   -0.92 3.48 -1.31 0.22 2.49 1.07 -1.19 -1.02 2.00
   -1.40 -0.78 1.12 0.50
8
9 decoded.soft <- ConvDecodeSoft(encoded.noisy,
   nasa.encoder)
10 decoded.soft
11 $output.soft
12 [1] -6.67485 6.67485 -6.67485 -6.67485 6.67485
13
14 $output.hard
15 [1] 1 0 1 1 0
16
17 decoded.hard <- ConvDecodeHard(encoded.noisy,
   nasa.encoder)
18 decoded.hard
19 [1] 1 0 1 1 0

```

---

Listing 7.2 – Kodierung und Dekodierung ohne Punktierung

Dekodierung ohne Punktierung. Für die Kodierung der Nachricht wird diese



einfach der Kodierungsfunktion `ConvEncode` zusammen mit dem Kodierer übergeben. Als Resultat erhält man das Kodewort mit den Signalpegeln +1 bzw. -1. Anschließend wird dieses mit der `ApplyNoise`-Funktion und einem Signal-Rausch-Verhältnis von 0,2dB verrauscht und der `encoded.noisy` Variable zugewiesen. Die Ausgabe dient dem Vergleich des unverfälschten und des verrauschten Kodeworts. Die Werte werden aus Platzgründen auf zwei Nachkommastellen gerundet. Schließlich erfolgt die `soft decision` und die `hard decision` Dekodierung. Der Rückgabewert der `soft decision` Dekodierung ist eine Liste welche sowohl die Soft-Werte als auch die Hard-Werte beinhaltet. Die `hard decision` Dekodierung gibt nur einen Vektor mit den „hart“ dekodierten Bits zurück. Das Beispiel zeigt die Fähigkeit der Faltungskodierung, stark verrauschte Signale korrekt dekodieren zu können.

### 7.3 Kodieren und Dekodieren mit Punktierung

Das kodierte Signal kann vor der Übertragung zur Verbesserung der Koderate punktiert werden. Die Punktierung streicht anhand der Punktierungsmatrix Bits aus dem Kodewort, wie in Kapitel 2.2.7 beschrieben. Im Folgenden Beispiel wird eine Nachricht kodiert und punktiert bevor sie dekodiert wird. Die Kodierung und Dekodierung mit Punktierung ist in Listing 7.3 zu sehen. Die Punktierung erfolgt durch das Mitgeben der Punktierungsmatrix bei der Kodierung. Das Ergebnis der Kodierung ist eine Liste mit dem originalen (nicht punktierten) Signal und dem punktierten Signal. Bei der weiteren Verwendung der Variable ist zu beachten nicht die gesamte Liste zu verwenden sondern das Listenelement des punktierten Codes. Dies ist beim Aufruf der Dekodierung zu sehen. Die Dekodierung benötigt auch die Punktierungsmatrix für das Auffüllen der punktierten Stellen im Kodewort. Als Ergebnis erhält man wie bei der Dekodierung ohne Punktierung eine Liste der Soft-Werte und Hard-Werte bei der `soft decision` Dekodierung bzw. einen Vektor der dekodierten Nachricht bei der `hard decision` Dekodierung.

### 7.4 Simulation

Die folgenden Kapitel demonstrieren die verschiedenen Möglichkeiten Simulationen durchzuführen. Kapitel 7.4.1 beinhaltet ein Beispiel für eine Simulation der Faltungskodierung. Eine Simulation der drei Verfahren der

---

```

1 encoded.punctured <- ConvEncode(input, nasa.encoder,
  punctuation.matrix = p.matrix)
2 encoded.punctured
3 $original
4 [1] -1 -1 -1 1 1 1 -1 1 1 -1 1 -1 1 1 -1 -1 1
   -1 -1 1 1
5
6 $punctured
7 [1] -1 -1 1 1 -1 1 -1 -1 1 1 -1 1 -1 1 1
8
9 decoded.soft <-
  ConvDecodeSoft(encoded.punctured$punctured,
    nasa.encoder, punctuation.matrix = p.matrix)
10 decoded.soft
11 $output.soft
12 [1] -6 5 -5 -5 5
13
14 $output.hard
15 [1] 1 0 1 1 0
16
17 decoded.hard <-
  ConvDecodeHard(encoded.punctured$punctured,
    nasa.encoder, punctuation.matrix = p.matrix)
18 decoded.hard
19 [1] 1 0 1 1 0

```

---

Listing 7.3 – Kodierung und Dekodierung mit Punktierung

Kanalkodierung des R-Pakets (Blockcodes, Faltungskodes und Turbo-Kodes) wird in Kapitel 7.4.2 präsentiert. Kapitel 7.4.3 beinhaltet eine Beispielanwendung der Hilfsfunktion zur Darstellung verschiedener Simulationsergebnisse.

### 7.4.1 Faltungskodierung

Das folgende Beispiel zeigt die Ausführung einer Simulation der Faltungskodierung. Die Parameter der Funktion wie der verwendete Kodierer, die Nachrichtenlänge, die zu testenden Signal-Rausch-Verhältnisse, die Anzahl an Wiederholungen je Signal-Rausch-Verhältnis sowie die Punktierungsmatrix können bestimmt werden, sind jedoch auch mit Standardwerten hinterlegt. Listing 7.4 zeigt die Simulation mithilfe der `ConvSimulation`-Funktion. Auf eine Punktierung wird in diesem Beispiel verzichtet. Die Länge der getesteten Nachrichten beträgt 2000. Diese werden bei einem Signal-Rausch-Verhältnis zwischen 0,01dB und 0,1dB verrauscht und im Anschluss dekodiert. Es wird der Mittelwert der Bitfehlerraten über 100 Wiederholungen je Signal-Rausch-

---

```
1 > dfl <- ConvSimulation(nasa.encoder, msg.length = 2000,
  min.db = 0.01, max.db = 0.1, db.interval = 0.01,
  iterations.per.db = 100)
2
3 dfl
4      db      ber
5 1 0.01 0.151810
6 2 0.02 0.140220
7 3 0.03 0.150020
8 4 0.04 0.143695
9 5 0.05 0.142150
10 6 0.06 0.142900
11 7 0.07 0.138555
12 8 0.08 0.137155
13 9 0.09 0.141615
14 10 0.10 0.137050
```

---

**Listing 7.4** – Simulation der Faltungskodierung

Verhältnis gebildet. Der Rückgabewert der Simulation ist ein Dataframe das die Bitfehlerrate (ber) je Signal-Rausch-Verhältnis (db) beinhaltet. Für eine Visualisierung kann der Visualisierungs-Parameter der Funktion gesetzt werden (`visualize = TRUE`) um einen Simulationsbericht als PDF-Datei generieren zu lassen. Eine weitere Möglichkeit bietet die in Kapitel 7.4.3 präsentierte Hilfsfunktion.

### 7.4.2 Kanalkodierung

Für einen Vergleich aller drei Verfahren der Kanalkodierung, die dieses R-Paket umfasst, kann die Funktion `ChannelcodingSimulation` verwendet werden. Diese Funktion ist eine Wrapper-Funktion, die jede Simulationsfunktion der verschiedenen Kanalkodierungen mit den gleichen Parametern aufruft und die resultierenden Bitfehlerraten in ein Dataframe schreibt und zurückgibt. Das Beispiel in Listing 7.5 führt eine Simulation für eine Nachrichtenlänge von 2000 aus. Die getesteten Signal-Rausch-Verhältnisse liegen zwischen 0,01dB und 0,1dB. Je Signal-Rausch-Verhältnis werden 100 Wiederholungen zur Ermittlung der Bitfehlerrate durchgeführt. Die Dekodierung der Turbo-Kodes erfolgt mit 3 Iterationen. Für eine Visualisierung der Daten kann der Visualisierungs-Parameter der Funktion gesetzt werden (`visualize = TRUE`) um einen Simulationsbericht als PDF-Datei generieren zu lassen.

---

```

1 ChannelcodingSimulation(msg.length = 2000, min.db = 0.01,
  max.db = 0.1, db.interval = 0.01, iterations.per.db =
    100, turbo.decode.iterations = 3)
2
3      db block.ber conv.ber turbo.ber
4 1 0.01 0.197045 0.088875 0.051680
5 2 0.02 0.197095 0.088670 0.051570
6 3 0.03 0.195555 0.090415 0.051935
7 4 0.04 0.196410 0.089065 0.052815
8 5 0.05 0.193475 0.089780 0.052700
9 6 0.06 0.194410 0.090370 0.050620
10 7 0.07 0.195250 0.089125 0.049625
11 8 0.08 0.194870 0.086845 0.050740
12 9 0.09 0.192465 0.089515 0.049495
13 10 0.10 0.193470 0.084220 0.048935

```

---

Listing 7.5 – Kanalkodierungs-Simulation

### 7.4.3 Vergleich von Simulationen

Das folgende Beispiel zeigt die Ausführung der `PlotSimulationData`-Funktion. Mithilfe dieser Funktion lassen sich die mit den Simulationsfunktionen erzeugten Dataframes in einem Plot darstellen, was den Vergleich erheblich vereinfacht. In Listing 7.6 wird zunächst ein zweites Dataframe erstellt, welches mit dem in Listing 7.4 erstellten Dataframe visualisiert werden soll. Als Vergleich wird ein katastrophaler Faltungskodierer (siehe Kapitel 2.2.3) erzeugt, mit dem die Simulation zur Erzeugung von `df2` ausgeführt wird. Abbildung 7.1 zeigt den erzeugten Plot.

---

```
1 catastrophic.encoder <- ConvGenerateEncoder(2,2,c(6,5))
2 df2 <- ConvSimulation(catastrophic.encoder, msg.length =
    2000, min.db = 0.01, max.db = 0.1, db.interval =
    0.01, iterations.per.db = 100)
3
4 df2
5      db      ber
6 1 0.01 0.492040
7 2 0.02 0.502815
8 3 0.03 0.490140
9 4 0.04 0.497805
10 5 0.05 0.499780
11 6 0.06 0.489430
12 7 0.07 0.496505
13 8 0.08 0.504305
14 9 0.09 0.493600
15 10 0.10 0.490645
16
17 PlotSimulationData(df1,df2)
```

---

Listing 7.6 – Vergleich mehrerer Simulationsdaten

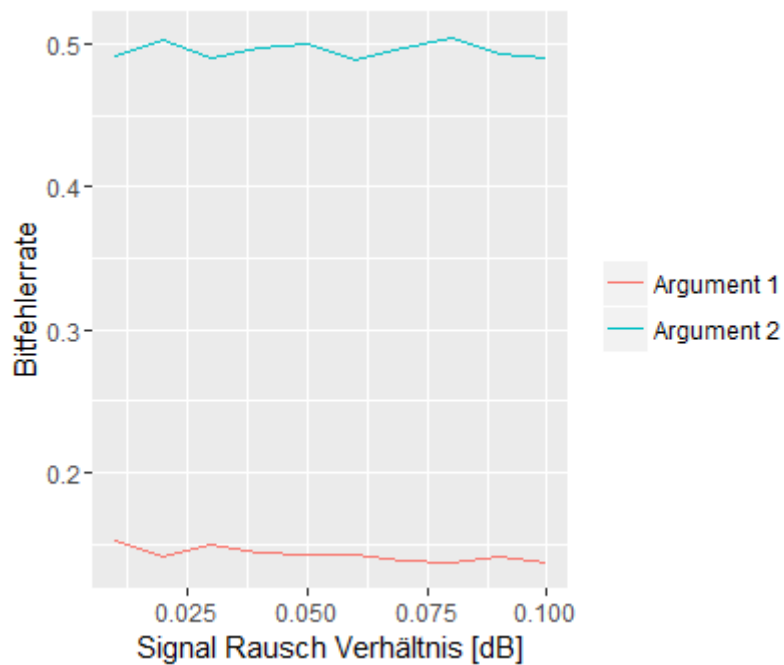


Abbildung 7.1 – Plot zweier Simulations-Dataframes

---

## **Kapitel 8**

## **Fazit, Ausblick, Erweiterungen**

---

---

## Abbildungsverzeichnis

---

2.1	Kommunikationskanal . . . . .	3
2.2	Beispiel für Faltungskodierer . . . . .	6
2.3	Trellis für die Kodierung aus Beispiel 2. . . . .	7
2.4	Trellis der hard decision Dekodierung aus Beispiel 3. . . . .	9
2.5	Trellis der soft decision Dekodierung aus Beispiel 4. . . . .	10
2.6	Verschiedene Faltungskodierertypen . . . . .	11
2.7	Trellis für die terminierte Kodierung aus Beispiel 5. . . . .	13
3.1	RStudio Standardansicht . . . . .	15
3.2	RMarkdown Überblick . . . . .	18
6.1	Folien mit allgemeinen Informationen zum Faltungskodierer	35
6.2	Folien der Kodierung . . . . .	36
6.3	Folie mit der Bit zu Signal Abbildung . . . . .	37
6.4	Folie mit Punktierung . . . . .	37
6.5	Folie mit Depunktierung . . . . .	38
6.6	Folie mit der Signal zu Bit Abbildung . . . . .	38
6.7	Folien mit der soft decision Dekodierung . . . . .	39
7.1	Plot zweier Simulations-Dataframes . . . . .	47

---

## Funktionsverzeichnis

---

5.1	ConvGenerateEncoder . . . . .	26
5.2	ConvGenerateRscEncoder . . . . .	27
5.3	ConvEncode . . . . .	27
5.4	ConvDecodeSoft . . . . .	28
5.5	ConvDecodeHard . . . . .	29
5.6	ConvSimulation . . . . .	30
5.7	ConvGetPunctuationMatrix . . . . .	30
5.8	ConvOpenPDF . . . . .	31
5.9	ApplyNoise . . . . .	31
5.10	ChannelcodingSimulation . . . . .	32
5.11	PlotSimulationData . . . . .	33



---

## Listingverzeichnis

---

7.1	Erzeugung von Kodierer und Punktierungsmatrix . . . . .	42
7.2	Kodierung und Dekodierung ohne Punktierung . . . . .	42
7.3	Kodierung und Dekodierung mit Punktierung . . . . .	44
7.4	Simulation der Faltungskodierung . . . . .	45
7.5	Kanalkodierungs-Simulation . . . . .	46
7.6	Vergleich mehrerer Simulationsdaten . . . . .	47

---

## Literatur

---

- [1] JJ Allaire u. a. *rmarkdown: Dynamic Documents for R*. R package version 0.9.5. 2016. URL: <http://rmarkdown.rstudio.com/>.
- [2] W Cary Huffman und Vera Pless. *Fundamentals of error-correcting codes*. Cambridge university press, 2010.
- [3] Robert H Morelos-Zaragoza. *The art of error correcting coding*. John Wiley & Sons, 2006.
- [4] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2016. URL: <https://www.R-project.org>.
- [5] D. Schönfeld, H. Klimant und R. Piotraschke. *Informations- und Kodierungstheorie*. Vieweg+Teubner Verlag, 2012. ISBN: 9783834882189. URL: <https://books.google.at/books?id=PDkpBAAAQBAJ>.
- [6] H. Wickham. *Advanced R*. CRC Press, 2015. URL: <https://books.google.at/books?id=FfsYCwAAQBAJ>.
- [7] H. Wickham. *R Packages*. O'Reilly Media, 2015. URL: <https://books.google.at/books?id=eq0xBwAAQBAJ>.