universität innsbruck
Fakultät für Mathematik, Informatik und Physik

Martin Nocker

R-Paket für Kanalkodierung mit Faltungskodes

Bachelorarbeit

3. Juni 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: **3. Juni 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, 3. Juni 2016

Zusammenfassung

Diese Bachelorarbeit beinhaltet die Implementierung von Faltungskodes in R. Faltungskodes sind ein häufig verwendeter Typ der Kanalkodierung um Nachrichten über einen Kanal, der die Nachrichten durch Rauschen verfälscht, zu senden. Dabei wird zur Information Redundanz hinzugefügt, sodass der Empfänger trotz fehlerhafter Übertragung die Originalnachricht dekodieren kann. Die implementierten Funktionen stehen in einem R-Paket zur Verfügung, welches Studierenden beim Untersuchen von Faltungskodes zu Lehrzwecken dient.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Kanalkodierung stellt, aufgrund der stetigen Ausbreitung digitaler Kommunikationssysteme, ein wichtiges Teilgebiet der Kodierungstheorie dar. Informationen, die über einen Kanal zwischen Quelle und Empfänger übertragen werden, können aufgrund von Rauschen verändert werden. Die Kanalkodierung stellt Methoden zur Verfügung, um Fehler, die während der Übertragung über einen verrauschten Kanal auftreten, zu korrigieren. Verwendung findet die Kanalkodierung in der Mobil- und Satellitenkommunikation, da beispielsweise das erneute Senden von Satellitendaten, bei Auftreten von Rauschen, aufgrund der Laufzeiten unpraktisch wäre. Weiters wird die Kanalkodierung zur Speicherung von Daten, etwa auf Compact-Disks, verwendet. Rauschen kann durch thermische Störungen bzw. Kratzer oder Fingerabdrücke auf Compact-Disks hervorgerufen werden.

Eine Art der Kanalkodierung stellen Faltungskodes dar, auf welche sich diese Arbeit konzentriert. Die wichtigste Anwendung von Faltungskodes ist die Konstruktion von Turbo-Kodes, welche eine Erweiterung der Faltungskodes darstellen. Weiters existiert mit Blockkodes ein weiteres wichtiges Verfahren der Kanalkodierung.

Ziel dieser Arbeit ist die Implementierung von Faltungskodes mithilfe der Programmiersprache R. Das entwickelte R-Paket dient zukünftigen Studierenden zu Lehrzwecken und soll sie beim Studieren von Faltungskodes unterstützen. Studierenden soll dadurch, neben den theoretischen Grundlagen im Zuge einer Vorlesung, eine praktische Möglichkeit zur Anwendung von Faltungskodes geboten werden. Mithilfe dynamisch generierter Visualisierungen, sollen die Prinzipien von Faltungskodes vermittelt werden.

Die Arbeiten „R-Paket für Kanalkodierung mit Blockkodes“ [12] von Benedikt Wimmer und „R-Paket für Kanalkodierung mit Turbo-Kodes“ [13] von Daniel Witsch stellen Implementierungen der Blockkodes bzw. Turbo-Kodes zur Verfügung, die ebenfalls im Paket enthalten sind. Dadurch ergibt sich ein sowohl umfangreiches als auch kompaktes Paket der Kanalkodierung.

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. Der Inhalt dieser Kapitel orientiert sich an [3], sowie [4] und [6].

2.1 Kanalkodierung

Kanalkodierung kann als Zuordnung bzw. Abbildung von Quellzeichen zu Kanalzeichen angesehen werden. Quellzeichen sind Zeichen, die eine Informationsquelle emittiert. Kanalzeichen sind Zeichen, die über einen Kommunikationskanal übertragen 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, könnte die verfälschte Nachricht nicht wiederhergestellt werden. Daher fügt ein Kanalkodierer den Quellzeichen Redundanz hinzu, sodass empfängerseitig verfälschte Zeichen erkannt und korrigiert werden können.

Abbildung 2.1 zeigt einen Kommunikationskanal inklusive Kanalkodierung. Eine Nachricht $\mathbf{u} = (u_1, u_2, \ldots, u_k)$ wird in ein Kodewort $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, welches Redundanz enthält, kodiert. Vor der Übertragung über den Kanal

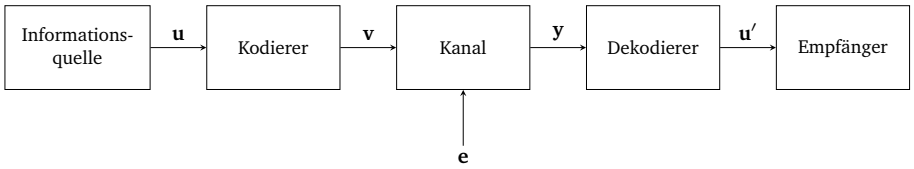


Abbildung 2.1 – Kommunikationskanal

werden die Bits folgendermaßen auf die Signalpegel +1 bzw. -1 abgebildet:

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

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

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

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

2.1.1 Koderate

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

$$R = \frac{k}{n} < 1 \tag{2.3}$$

Wobei k bzw. n den Längen des Quellworts bzw. Kodeworts entsprechen. Somit 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 Kodes mit niedriger Koderate länger, als bei Kodes mit höherer Koderate.

2.1.2 Hamming-Distanz

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

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

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

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

Das Hamming-Gewicht w eines binären Kodeworts $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ entspricht der Anzahl an Bits im Wort, für die gilt $a_i = 1$ mit $i \in \{1,2,\ldots,n\}$.

$$w(a) = |\{i \in \{1,2,\ldots,n\} \,|a_i = 1\}| = \sum_{i=1}^{n} a_i \tag{2.6}$$

2.2 Faltungskodierung

Faltungskodes sind blockfreie Kodes, d.h. Quellzeichen werden nicht in Blöcke fester Länge unterteilt, vielmehr wird ein Informationsstrom kodiert, sodass ein einziges Kodewort resultiert. Ein weiterer Unterschied zu Blockkodes 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, sodass mehrere Informationsbits gleichzeitig kodiert werden. 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 $\left(\mathbf{v} \in \{0,1\}^N\right)$ kodiert. Es ergibt sich somit eine Koderate von $R = \frac{1}{N}$. Ein Ausgang wird durch ein sogenanntes *Generatorpolynom* definiert. Ein Generatorpolynom stellt eine Linearkombination der M Elemente des Schieberegisters und dem Eingangssignal dar und wird durch ein XOR-Gatter abgebildet. Alle Generatorpolynome werden in einer *Generatormatrix*

$$G = (g_1, g_2, \ldots, g_N) \tag{2.7}$$

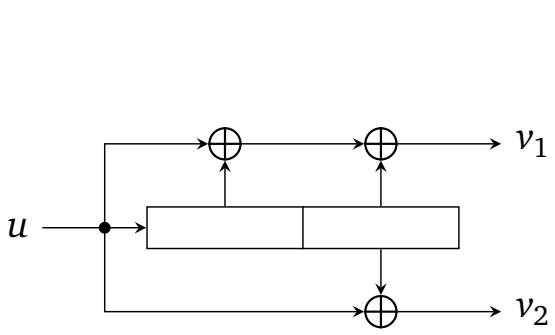
angegeben, wobei das Generatorpolynom g_i den Ausgang i definiert. Zur Definition eines Faltungskodierers im praktischen Teil der Arbeit müssen die Generatorpolynome angegeben werden.
Ein weiterer wichtiger Parameter von Faltungskodes ist die *Einflusslänge* (constraint length). Diese gibt an, wie oft sich ein Eingangsbit auf die Kodierung auswirkt. Die Einflusslänge wird durch die Länge des Schieberegisters bestimmt. Ein Eingangsbit beeinflusst $M + 1$ mal die Kodierung.

Die Verhaltensweise eines Faltungskodierers kann 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.

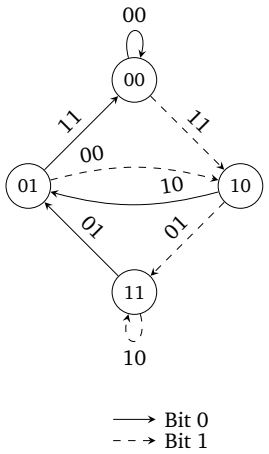
Beispiel 1. *Ein Faltungskodierer ist gegeben durch das Schaltbild bzw. Zustandsdiagramm in Abbildung 2.2. Der Faltungskodierer besitzt die Generatormatrix*

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

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 konzise zu oktalen Zahlen zusammengefasst. Bei der binären Schreibweise entspricht die Bitposition des Polynoms dem Element im Schaltbild, d.h. das Most Significant Bit (MSB) des Polynoms steht für das Eingangssignal, das Least Significant Bit (LSB) des Polynoms steht für den Inhalt des letzten



(a) Schaltbild



(b) Zustandsdiagramm

Abbildung 2.2 – Beispiel für Faltungskodierer

(am weitesten rechts liegenden) Elements des Schieberegisters. In die Linearkombination zur Definition des Ausgangssignals werden jene Elemente miteinbezogen, an deren Stelle im binären Polynom eine 1 steht. In [3] wird die Notation der binären Polynome über der Variable D („delay“) eingeführt. Das Eingangssignal und die Schieberegisterelemente entsprechen einer Potenz von D, wobei das Eingangssignal u der Potenz $D^0 = 1$ und das letzte Schieberegisterelement D^M entspricht. Das Generatorpolynom ergibt sich aus der Summe aller Potenzen deren Elemente Teil der Linearkombination sind. Darüber hinaus wird in Abbildung 2.2b das Zustandsdiagramm abgebildet. Die Knoten stellen die Zustände des Kodierer, 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 und 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

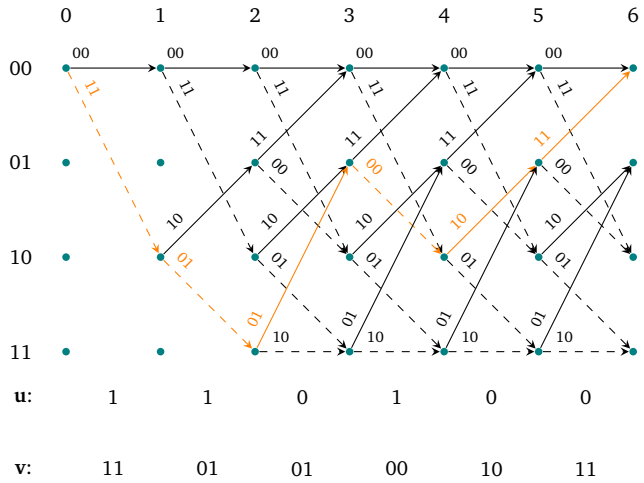


Abbildung 2.3 – Trellis für die Kodierung zu Beispiel 2

der Nachricht aus Beispiel 2. Das Trellis ist eine Erweiterung des Zustandsdiagramms um eine Zeitachse auf der Abszisse. Die Zustände sind auf der Ordinate 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 ergeben, bewertet.

Der Viterbi-Algorithmus verwendet das Trellis zur Dekodierung eines empfangenen Kodes. Dabei wird für den empfangenen Kode 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.

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 durchläuft das Trellis von links nach rechts, wobei die Metriken der Kanten berechnet werden. 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, die s und s' verbindet, und dem zum Zeitpunkt t empfangenen Teil des Kodes. Die Metrik eines Pfads im Trellis ist die Summe der Kantenmetriken des Pfads. Ein Pfad mit einer großen Metrik hat eine große Hamming-Distanz zum empfangenen Kode, 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 durchlaufen. 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 mit den 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 2.5 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 muss 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.

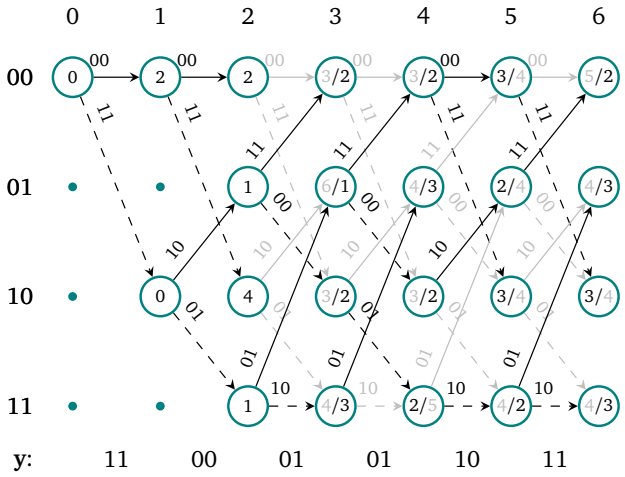


Abbildung 2.4 – Vollständiges Trellis der hard decision Dekodierung zu Beispiel 3

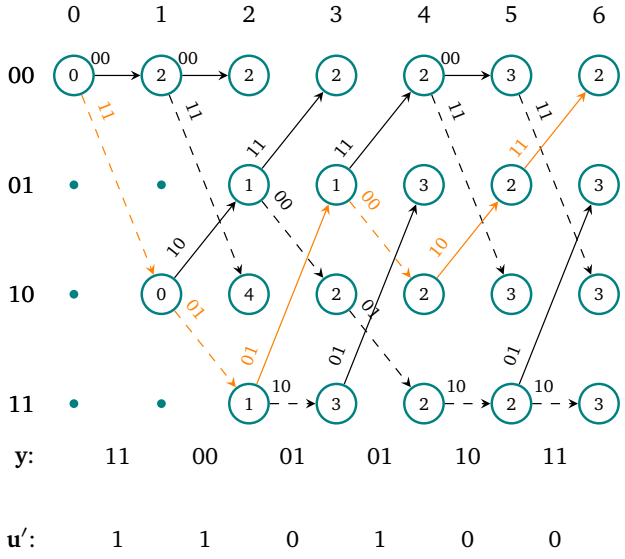


Abbildung 2.5 – Backtracking im Trellis der hard decision Dekodierung zu Beispiel 3

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 werden auf eine 0, negative auf eine 1 abgebildet. Der Betrag des Soft-Werts

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 [6, 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 2.6 zeigt die Dekodierung im Trellis mit den 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 2.7 werden die verworfenen Pfade nicht mehr angezeigt. Aus dem Backtracking resultiert die dekodierte Nachricht* $\mathbf{u}' = (110100)$*. Der Pfad der dekodierten Nachricht im Trellis 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 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 $\mathbf{u}'$, die möglichst $\mathbf{u}$ entsprechen sollte, berechnet werden. Dies ist bei einer fehlerfreien Übertragung, d.h. $\mathbf{v}' = \mathbf{v}$, sicherlich der Fall. Im Folgenden wird der Fall $\mathbf{v}' \neq \mathbf{v}$ untersucht: 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 Kode endliches Hamming-Gewicht hat. Katastrophale Kodierer sind zu vermeiden, da eine endliche Anzahl an Übertragungsfehler zu einer unendlichen Anzahl an Dekodierfehler führen kann. [3, S. 569]

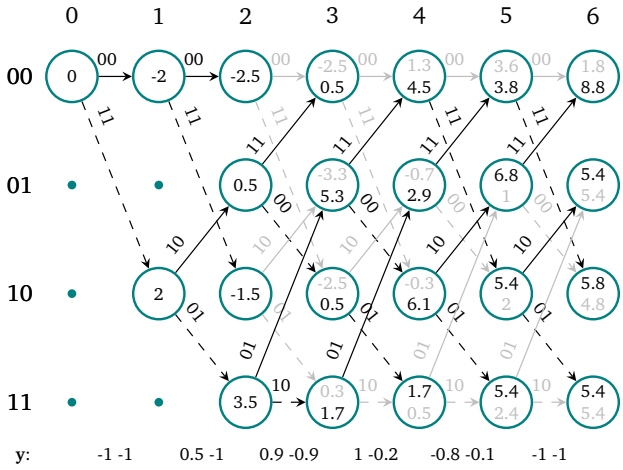


Abbildung 2.6 – Vollständiges Trellis der soft decision Dekodierung zu Beispiel 4

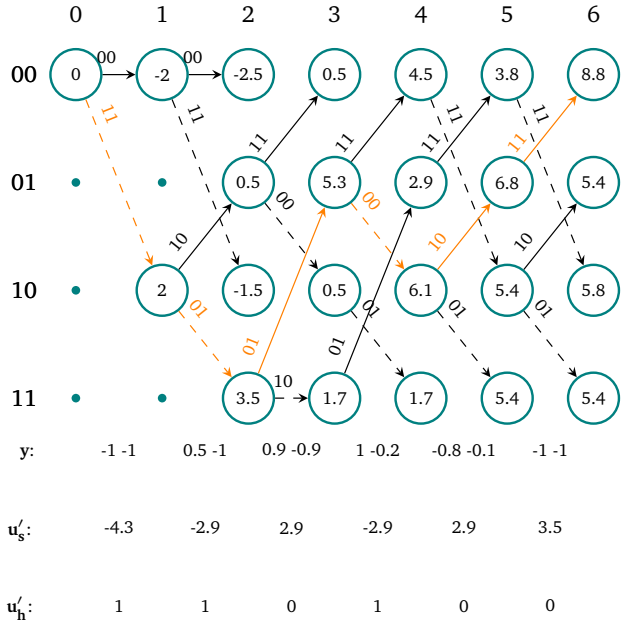
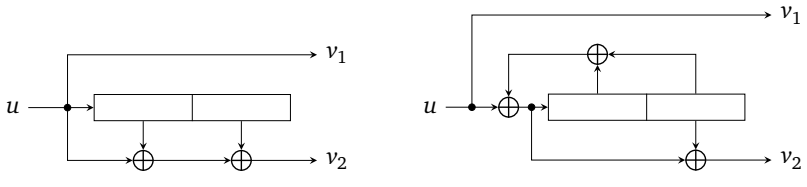


Abbildung 2.7 – Backtracking im Trellis der hard decision Dekodierung zu Beispiel 4

Zur Überprüfung, ob ein Kodierer katastrophal ist, hilft das Theorem von Massey-Sain [3, S. 570]. Sei ein Faltungskodierer mit einem Eingang und der Generatormatrix G als Polynome über D gegeben. Der Kodierer ist nicht katastrophal genau dann, wenn der größte gemeinsame Teiler der Generatorpolynome eine Potenz von D ist.



$G = (4_8, 7_8)$

(a) Systematischer Faltungskodierer

$G = \left(1, \frac{5_8}{7_8}\right)$

(b) Rekursiv Systematischer Faltungskodierer

Abbildung 2.8 – Verschiedene Faltungskodierertypen

2.2.4 Systematische Faltungskodierer

Bei systematischen Faltungskodierern entspricht ein Ausgang dem Eingangssignal. Die Quellinformation ist somit explizit im Kodewort enthalten. Abbildung 2.8a zeigt einen systematischen Faltungskodierer mit der Generatormatrix $G = (4_8, 7_8)$. Systematische Faltungskodierer sind nie katastrophal, sie sind jedoch weniger robust wie nichtsystematische Faltungskodierer. [6, S. 217]

2.2.5 Rekursiv systematische Faltungskodierer

Rekursiv systematische Faltungskodierer (RSC-Kodierer[1]) 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 Kodewort enthalten. RSC-Kodierer sind aufgrund ihrer Verwendung in Turbo-Kodes von großer Bedeutung.
Die Generatormatrix eines RSC-Kodierers mit einem Eingang wird folgendermaßen angegeben:

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

Zumeist befindet sich an erster Stelle eine 1, welche den systematischen Ausgang notiert. Das Polynom g_0 definiert die Rückkopplung des Kodierers. Die Polynome g_1 bis g_{N-1} stellen die Polynome der nichtsystematischen Ausgänge dar [4, S. 92 f.].
Zur Definition eines RSC-Kodierers im späteren Programm müssen die oktalen

[1] Recursive Systematic Convolutional Coder

Generatorpolynome der nichtsystematischen Ausgänge sowie der Rekursion angegeben werden. Das Polynom des systematischen Ausgangs muss nicht angegeben werden. Abbildung 2.8b zeigt einen rekursiv systematischen Faltungskodierer für die Generatormatrix $G = \left(1, \frac{5_8}{7_8}\right)$.

2.2.6 Terminierung

Die Terminierung bezeichnet das Zurückkehren des Kodierers, nach der vollständigen Kodierung einer Nachricht, 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 immer der Nullzustand ist und somit 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 $\mathbf{u} = (u_1, u_2, \ldots, u_k)$ wie folgt:

$$R_t = \frac{k}{M+k} R \tag{2.9}$$

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 2.9 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 und P spaltenweise durchlaufen wird. Das Gewicht der Matrix $w(P)$ entspricht der Anzahl nicht punktierter Kodebits je Periode, d.h. der Anzahl an Bits für die gilt $P_{ij} = 1$ mit $i \in \{1, 2, \ldots, N\}$ und $j \in \left\{1, 2, \ldots, \frac{p}{N}\right\}$. [6, S. 218]

Die Koderate des punktierten Kodes R_p berechnet sich wie folgt:

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

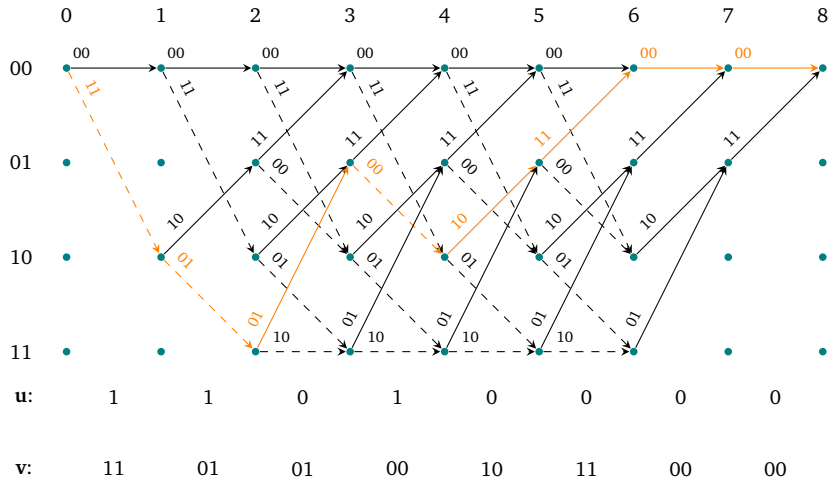


Abbildung 2.9 – Trellis für die terminierte Kodierung zu Beispiel 5

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.

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 Kodes 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, vor allem mithilfe des Pakets `Rcpp`, beschrieben. Schließlich wird in Kapitel 3.3 auf die Erstellung dynamischer Dokumente und Visualisierungen mittels `RMarkdown`, LaTeX und Ti*k*Z 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, große Datenmengen graphisch darzustellen. 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[2] oder GitHub sind über 8000 zusätzliche Pakete (Stand: Mai 2016) für die verschiedensten Anwendungsbereiche verfügbar. Diese Vielfalt an Paketen ist ein Grund für den Erfolg von R [9, S. 18].

[2]The Comprehensive R Archive Network: `https://cran.r-project.org/`

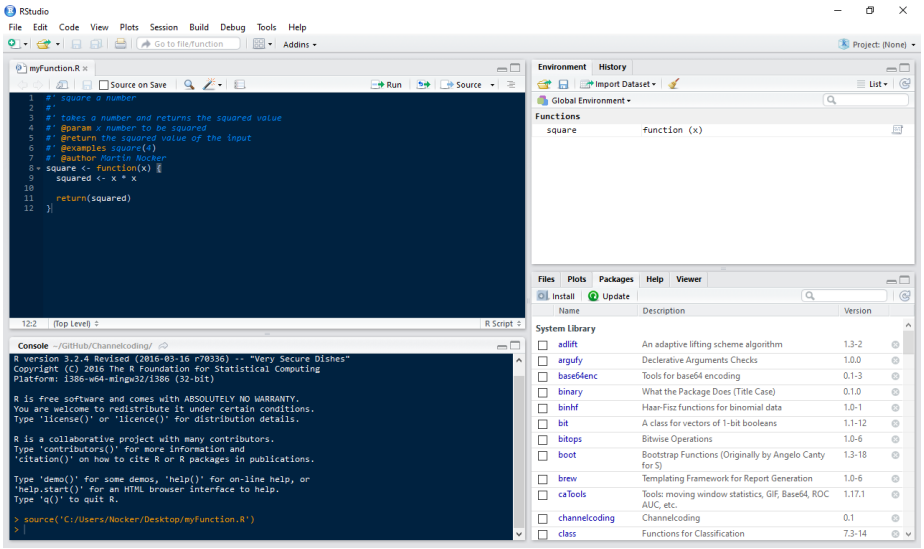


Abbildung 3.1 – RStudio Standardansicht

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 [9].

Ein wichtiges Paket, welches im Rahmen dieser Arbeit verwendet wurde, ist `roxygen2`. Mithilfe dieses Pakets wird, ähnlich zur 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. Diese 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. [11]

Ein weiteres hilfreiches Paket ist das `devtools`-Paket. Dieses Paket stellt Funktionen für die Erstellung (Build) von Paketen zur Verfügung und beschleunigt so den Build-Workflow für den Entwickler. [10]

RStudio ist eine freie 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

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 in diesen Sprachen 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 und deren Speicher muss vor dem Aufruf reserviert werden.

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, mithilfe des `return` Statements, die Möglichkeit eines Rückgabewerts.

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, wie normale R-Funktionsaufrufe aus 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. Die genaue Verwendung des `Rcpp`-Pakets ist in [8, S. 395 ff.] beschrieben.

3.3 RMarkdown, LaTeX, Ti*k*Z

Zur Erstellung von dynamischen Dokumenten wird das Paket `RMarkdown` verwendet. Durch die Kombination der Syntax von Markdown, R, und LaTeX ergibt sich ein flexibles und einfaches Werkzeug. Die unterstützen 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 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 `pandoc` weiterverarbeitet, das 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 einem einzigen `render`-Funktionsaufruf.

Für die Erzeugung dynamischer Grafiken wird das LaTeX-Sprachpaket Ti*k*Z verwendet. Mithilfe des Dokumenttyps Beamer in LaTeX lassen sich Präsentationen erstellen. Die Grafiken und Inhalte können dadurch dynamisch ein- oder ausgeblendet, sowie 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 verste-

Abbildung 3.2 – RMarkdown Überblick, Quelle: [1]

hen können, werden die Visualisierungen der Kodierung und Dekodierung sukzessive eingeblendet.

Kapitel 4

Implementierung

Dieses Kapitel gibt einen Einblick in den Workflow der R-Paket Erstellung und die Konzepte der Implementierung. Weiters werden wichtige Algorithmen im Pseudocode dargestellt und erläutert. Als Einstiegspunkt für die Implementierung stand eine C-Referenzimplementierung von Dusan Orlovic [5] zur Verfügung, die den Dekodier-Algorithmus für ein konkretes Beispiel beinhaltet. Es war also eine Anpassung des Algorithmus erforderlich um eine allgemeine Verwendbarkeit von Faltungskodes zu gewährleisten.

In Kapitel 4.1 wird der Workflow zur Entwicklung eigener R-Pakete beschrieben. In Kapitel 4.2 wird die implementierte Faltungskodierer-Datenstruktur erläutert. Anschließend wird der Kodierungsalgorithmus in Kapitel 4.3 erklärt, gefolgt vom Dekodierungsalgorithmus in Kapitel 4.4. Weiters wird die Umsetzung der Funktion zum Verrauschen einer Nachricht in Kapitel 4.5 präsentiert. Kapitel 4.6 beinhaltet die Implementierung der Punktierung. Schließlich geben die Kapitel 4.7 bzw. 4.8 einen Einblick in die Implementierung der Simulation bzw. Visualisierungen.

Bei den Implementierungen der Funktionen ab Kapitel 4.2 werden alle Parameter auf korrekt übergebene Werte überprüft. Auf eine Erklärung dieser Überprüfungen wird, außer bei Funktionen, die andernfalls unvollständig beschrieben wären, verzichtet. Darüber hinaus sind für den Großteil der Parameter Standardwerte hinterlegt. Diese Standardwerte kommen ersatzweise, für vom Benutzer nicht übergebene Parameter zum Einsatz.

4.1 R-Paket Entwicklung

Dieses Kapitel setzt die Installation der R-Pakete `roxygen2` und `devtools` (siehe Kapitel 3.1) voraus. Zur Installation eines Pakets kann im RStudio im *Packages*-Tab die *Install*-Funktion oder der Befehl in Zeile 1 des Listings 4.1 verwendet werden.
Bei der Verwendung von C/C++-Code werden ein entsprechender Compiler

```
1 install.package(<package-name>)
2 devtools::install_deps(pkg = <package-name>, dependencies
      = TRUE)
```

Listing 4.1 – Installation eines R-Pakets und dessen Abhängigkeiten

sowie, abhängig vom Betriebssystem, weitere Tools (bspw. RTools[3] unter Windows) benötigt. Ausführliche Informationen über die Paket-Entwicklung in R sind in [9] beschrieben.

RStudio ist prädestiniert für die Entwicklung eigener R-Pakete. Über *File* → *New Project* → *New Directory* → *R Package* kann ein neues Paket erstellt werden. Es wird ein Projekt-Ordner erstellt, der u.a. eine DESCRIPTION- und NAMESPACE-Datei, sowie einen *R*-Ordner enthält. Im *R*-Ordner sind alle R-Skripte zu finden. Bei der Verwendung von C/C++-Code ist dieser in einem *src*-Ordner abzulegen. Im *inst*-Ordner können beliebige Dateien abgelegt werden, die nach der Installation im Paket verfügbar sein sollen. [9, S. 28 ff.]
Die DESCRIPTION-Datei enthält allgemeine Informationen zum Paket, u.a. den Paketname, eine Beschreibung des Pakets, den Namen des Entwicklers bzw. der Entwickler, eine Versionsnummer, eine Lizenz und benötigte Pakete (Abhängigkeiten). Letztere werden bei der Installation eines Pakets von CRAN automatisch installiert. Bei einer Installation einer lokalen Archivdatei eines Pakets müssen die Abhängigkeiten manuell installiert werden. Zeile 2 des Listings 4.1 zeigt den Befehl dazu. [9, S. 67 ff.]
Die NAMESPACE-Datei verwaltet das Exportieren und Importieren von Funktionen in den Paketnamensraum. Die Datei wird bei der Verwendung von `roxygen2` automatisch erstellt. Bei der Verwendung von C++-Code und dem

[3] `https://cran.r-project.org/bin/windows/Rtools/` (besucht am 01.06.2016)

Rcpp-Paket muss der Code aus Listing 4.2 in ein R-Skript des Pakets geschrieben werden. [9, S. 144 ff.]
Um die devtools- und roxygen-Funktionen zu aktivieren, müssen die dafür

```
1 #' @useDynLib <my-package-name>
2 #' @importFrom Rcpp sourceCpp
```

Listing 4.2 – Notwendige roxygen-Kommentare bei der Verwendung von C++-Code

vorgesehenen Checkboxen in den Projekt-Optionen *(Build → Configure Build Tools...)* gesetzt werden. Abbildung 4.1 zeigt die *Build Tools* Einstellungen und die *Roxygen Options*, die durch das Klicken auf den *Configure*-Button geöffnet werden.
Nach der Implementierung der R-Skripte kann das endgültige R-Paket ge-

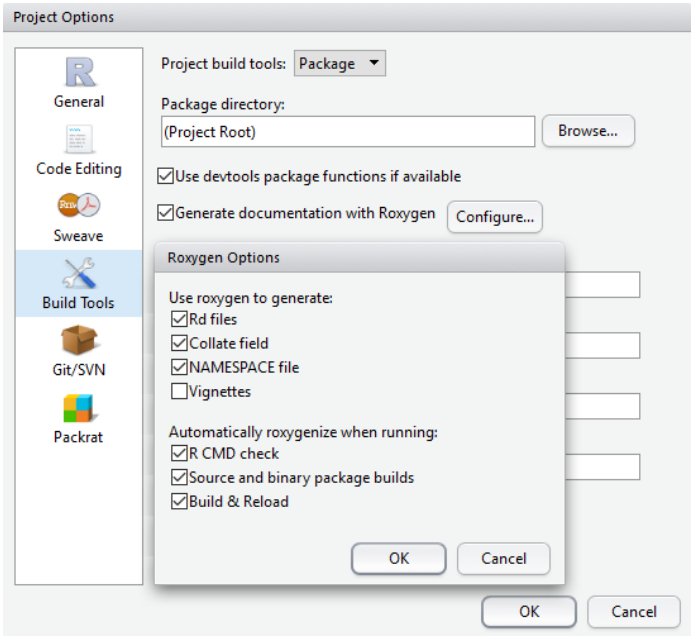


Abbildung 4.1 – Build- und Roxygen-Optionen des Projekts

neriert werden. Dazu muss auf den *Build & Reload*-Button im *Build*-Tab geklickt werden. Bei der Verwendung von C++-Code werden nun C++-Dateien kompiliert und R-Wrapper-Funktionen erstellt, die den Zugriff auf die C++-Funktionen erleichtern.

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

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

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

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

Am Ende des Build-Prozesses wird das Paket lokal installiert und ist bereit verwendet zu werden. Um das Paket auf anderen Rechnern installieren zu können muss im *Build*-Tab via *More* → *Build Binary Package* das Paket zu einem Archiv gepackt werden. Dieses Archiv ist jedoch nur für Rechner mit demselben Betriebssystem geeignet, d.h. das Paket muss für jedes Betriebssystem separat kompiliert und gepackt werden.

4.2 Faltungskodierer

Ein Faltungskodierer ist gegeben durch

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

Die Angabe von M, die Teil der Benutzereingabe zur Generierung eines Faltungskodierers ist, ist hier redundant und wurde durch [4] inspiriert.

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,
- 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 rekursiv systematischer Kodierer (RSC, siehe Kapitel 2.2.5),
- ein *Terminierungsvektor*, der für rekursiv 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 (z.B. für alle nichtrekursiven Kodierer) 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. Nicht benötigte Felder der Matrix beinhalten den Wert -1. Alle Matrizen beinhalten dezimale Werte. Um die Bit-Werte zu erhalten, wie z.B. die Bits der Ausgabematrix, die für die Kodierung notwendig sind, muss mit bitweisen Shift-Operationen gearbeitet werden. Ein Codeausschnitt der Implementierung zur Erzeugung von Faltungskodierer ist in Listing 4.3 zu sehen.

Der Terminierungsvektor ist für nichtrekursive 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 Rekursionpolynoms 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 euklidschen Algorithmus implementiert. Sowohl der euklidsche Algorithmus, als auch die dafür notwendige binäre Polynomdivision wird an eine C++-Funktion delegiert.

```
1 for (int state = 0; state < NUM_STATES; state++) {
2   for (int input = 0; input < 2; input++) {
3     int current_state = (input << M) | state;
4
5     int out = 0;
6     for (int i = 0; i < N; i++) {
7       int temp = sumDigits(current_state & generator[i], 2) % 2;
8       out = (out << 1) | temp;
9     }
10    output(state, input) = out;
11    nextState(state, input) = current_state >> 1;
12  }
13 }
14
15 for (int state = 0; state < NUM_STATES; state++) {
16   for (int input = 0; input < 2; input++) {
17     if (previousState(nextState(state, input), input) == -1) {
18       previousState(nextState(state, input), input) = state;
19     }
20     else {
21       previousState(nextState(state, input), 2) = state;
22     }
23   }
24 }
```

Listing 4.3 – Codeausschnitt der Implementierung der Erzeugung von Faltungskodierer

4.3 Kodierung

Bei Faltungskodes stellt die Kodierung den 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 wird das Terminierungsbit anhand des Terminierungsvektors ermittelt, 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 4.1 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· bit
17:    result = concat(result, signal)
18: end for
19: return result
```

Algorithmus 4.1 – Pseudocode der Faltungskodierung

4.4 Dekodierung

Die Dekodierung stellt bei Faltungskodes den wesentlich komplexeren Teil dar. Der Pseudocode des Viterbi-Algorithmus ist in Algorithmus 4.2 zu sehen. Der Algorithmus zeigt die hard decision Dekodierung, deren Implementierung im Folgenden diskutiert wird. Zunächst werden zu jedem Zeitpunkt die Metriken aller Zustände berechnet und in eine Matrix `metric` geschrieben. Die Metriken zum Zeitpunkt 0 werden initialisiert. Der Nullzustand, der dem Startzustand entspricht, wird mit dem Wert 0 initialisiert. Die restlichen Zustände zum Zeitpunkt 0 werden mit einer sehr hohen Metrik initialisiert, da es unmöglich ist, dass ein korrekter Pfad aus einem anderen Zustand als

dem Nullzustand startet. Dadurch werden Pfade, die in diesen Zuständen beginnen, durch ihre hohe Metrik immer verworfen.

Die verschachtelte Schleife in den Zeilen 3 - 14 implementiert den in Kapitel 2.2.2.1 beschriebenen Algorithmus. Die `survivor.bit`-Matrix speichert (für jeden Zustand, zu jedem Zeitpunkt) jenes der beiden Bits mit dem der Pfad mit der niedrigeren Metrik in den Zustand gelangt. Es wird also nicht das letzte Bit des verworfenen Pfads, sondern das (letzte) Bit des Pfads mit der niedrigeren Hamming-Distanz.

Im Anschluss werden die *survivor states* gesucht. Diese entsprechen den Zuständen, die bei der Kodierung im Trellis durchlaufen werden. Aus ihnen kann, zusammen mit der `survivor.bit`-Matrix, die dekodierte Nachricht berechnet werden. Die Suche entspricht dem Backtracking. Zunächst wird der survivor state am Ende des Trellis gesucht. Wurde die Nachricht bei der Kodierung terminiert entspricht der survivor state dem Nullzustand. Andernfalls muss der Zustand mit der niedrigsten Metrik gesucht werden. Dieser wird dann zum survivor state. Die darauf folgende Schleife führt das Backtracking bis zum Zeitpunkt 0 fort.
Im letzten Schritt wird die dekodierte Nachricht rekonstruiert.

1: NUM.STATES $= 2^M$
2: msg.length $= \frac{length(\text{code})}{N}$
3: **for** $t = 1$ **to** msg.length **do**
4: **for** $s = 0$ **to** NUM.STATES-1 **do**
5: $m_1 = \text{metric}[t-1][\text{prev.state}_1] + d_{H,\text{prev.state}_1 \to s}$
6: $m_2 = \text{metric}[t-1][\text{prev.state}_2] + d_{H,\text{prev.state}_2 \to s}$
7: $\text{metric}[t][s] = min(m_1, m_2)$
8: **if** $m_1 < m_2$ **then**
9: $\text{survivor.bit} = bit(\text{prev.state}_1 \to s)$
10: **else**
11: $\text{survivor.bit} = bit(\text{prev.state}_2 \to s)$
12: **end if**
13: **end for**
14: **end for**
15: **if** is.terminated **then**
16: survivor $= 0$
17: **else**
18: minimum $=$ metric[msg.length][0]
19: **for** $s = 1$ **to** NUM.STATES **do**
20: **if** metric[msg.length][s] $<$ minimum **then**
21: minimum $=$ metric[msg.length][s]
22: survivor $= s$
23: **end if**
24: **end for**
25: **end if**
26: survivor.state[msg.length] $=$survivor
27: **for** $t =$ msg.length $- 1$ **to** 0 **do**
28: $S =$ survivor.state$[t+1]$
29: survivor.state$[t]$ $=$ previous.matrix[S, survivor.bit$[t+1][S]$]
30: **end for**
31: **for** $t = 1$ **to** msg.length **do**
32: result $= concat($result, survivor.bit$[t]$[survivor.state$[t]]$)
33: **end for**
34: **return** result

Algorithmus 4.2 – Pseudocode der hard decision Dekodierung

4.5 Rauschen

Um 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[4]) addiert werden um dieses zu verfälschen. In [7] wird eine alternative Implementierung zur eingebauten AWGN-Funktion in Matlab vorgestellt. 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.

4.6 Punktierung

Bevor ein Kode punktiert werden kann (siehe Kapitel 2.2.7) benötigt es eine Punktierungsmatrix. Die Funktion zur Generierung einer Punktierungsmatrix bringt den vom Benutzer mitgegebenen Punktierungsvektor in die korrekte Matrixform.
Zunächst wird der Punktierungsvektor überprüft, da dessen Länge ein Vielfaches von N (Anzahl der Ausgänge des Faltungskodierers) sein muss. Nach der Transformation des Vektors zur Punktierungsmatrix wird geprüft, ob die Matrix eine Nullspalte enthält, da eine Nullspalte zur Folge hat, dass die 0-Werte vor der Dekodierung nicht eindeutig eingefügt werden können.

Für die Punktierung eines Kodevektors ist die Verwendung von Indexvektoren in R hervorragend geeignet. Die Punktierungsmatrix muss lediglich in einen logischen Vektor umgewandelt und als Indexvektor zwischen eckige Klammern hinter den Kodevektor geschrieben werden. Somit lässt sich die Punktierung, für Nachrichten, die mindestens gleich lang wie die Punktierungsmatrix sind, in nur einer R-Codezeile implementieren. Für kürzere Kodes muss einzig vor der Punktierung die Punktierungsmatrix auf die Kodelänge abgeschnitten werden.

[4] Additive White Gaussian Noise

Im Gegensatz zur Punktierung ist das Einfügen der 0-Werte mit mehr Aufwand verbunden. Vor allem die Berechnung der Anzahl an 0-Werten, die eingefügt werden müssen, ist für allgemeine Punktierungsmatrizen nicht trivial und wurde daher in eine C++-Funktion ausgelagert.
Die Anzahl an 0-Werten, die eingefügt werden müssen, berechnet sich für eine allgemeine Punktierungsmatrix $P \in \{0,1\}^{N \times \frac{p}{N}}$, die q 1en enthält, und ein punktiertes Kodewort $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ wie folgt:

$$zeros = \left\lfloor \frac{n}{q} \right\rfloor (p-q) + m \tag{4.1}$$

In Gleichung (4.1) entspricht m der Anzahl an 0en, die in den ersten k Spalten der Punktierungsmatrix stehen, wobei k dem kleinsten Wert entspricht, sodass die ersten k Spalten mindestens $(n \bmod q)$ 1en enthalten. Für $n \bmod q = 0$ ist $m = 0$.
Der erste Summand ergibt sich aus der Anzahl, wie oft der punktierte Kode vollständig die 1en der Punktierungsmatrix füllt, multipliziert mit der Anzahl an 0en $(p-q)$. Die Anzahl an zusätzlichen 0-Werten m, die eingefügt werden müssen, ergibt sich aus den restlichen Kodebits, die die 1en der Punktierungsmatrix nicht genau voll machen und dadurch vom ersten Summanden nicht berücksichtig werden.

Das Einfügen der 0-Werte ist mithilfe einer Schleife, welche die Punktierungsmatrix in jeder Iteration an der entsprechenden Stelle prüft und das depunktierte Kodewort aufbaut, zu implementieren.

Beispiel 7. *Gegeben seien die punktierten Kodes* $\mathbf{v_{p,1}} = (110010)$ *bzw.* $\mathbf{v_{p,2}} = (10011101001)$ *mit einer Länge von 6 bzw. 11 Bits, sowie die Punktierungsmatrix*

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

Nach Gleichung (4.1) *ergeben sich für* $\mathbf{v_{p,1}}$ $\left\lfloor \frac{6}{4} \right\rfloor (6-4) + 0 = 2$ *bzw. für* $\mathbf{v_{p,2}}$ $\left\lfloor \frac{11}{4} \right\rfloor (6-4) + 1 = 5$ *0-Werte, die einzufügen sind.* $\mathbf{v_{p,1}}$ *füllt die 1en der Punktierungsmatrix einmal vollständig, wodurch sich zwei zusätzliche 0-Werte ergeben. Die übrigen zwei Bits füllen die 1en der ersten Spalte, welche keine 0en enthält, daher kommen keine weiteren 0-Werte dazu.* $\mathbf{v_{p,2}}$ *füllt die die 1en der Punktierungsmatrix zweimal vollständig, wodurch sich vier zusätzliche 0-*

Werte ergeben. Die übrigen drei Bits füllen die 1en der ersten beiden Spalten, welche eine 0 enthalten, daher kommt ein weiterer 0-Wert hinzu. Die depunktierten Kodes lauten $\mathbf{v_1} = (11\underline{0}00\underline{0}10)$ *bzw.* $\mathbf{v_2} = (10\underline{0}01\underline{0}11\underline{0}01\underline{0}00\underline{0}1)$*. Die unterstrichenen 0-Bits stellen die eingefügten Werte dar.*

4.7 Simulation

Die Simulation führt ein Kodierungsverfahren für zufällige Nachrichten mehrmals für verschiedene Signal-Rausch-Verhältnisse durch. Dabei werden die Nachrichten kodiert, verfälscht und wieder dekodiert, sowie die resultierende Bitfehlerrate (BER[5]), also der Prozentsatz unterschiedlicher Bits zwischen originaler und dekodierter Nachricht, berechnet. Dieser Vorgang wird je Signal-Rausch-Verhältnis mehrmals durchgeführt. Die Bitfehlerrate ergibt sich aus dem Mittelwert aller Wiederholungen. Dadurch werden statistische Ausreißer vermieden. Bei der Dekodierung wird die soft decision Dekodierung verwendet.

Die Simulation kann über folgende Parameter gesteuert werden:

- Faltungskodierer, der für die Kodierung verwendet wird
- Nachrichtenlänge der zufällig erzeugten Nachrichten
- Sigal-Rausch-Verhältnisse, die getestet werden, durch Angabe von Minimum, Maximum und Schrittweite
- Anzahl der Wiederholungen je Signal-Rausch-Verhältnis
- Punktierungsmatrix (optional). Eine Punktierung erfolgt nur, falls eine Punktierungsmatrix mitgegeben wird.

Als Resultat wird ein Dataframe zurückgegeben, welches die Bitfehlerrate je Signal-Rausch-Verhältnis beinhaltet. Algorithmus 4.3 zeigt den Pseudocode der Simulation ohne Punktierung.

[5] bit error rate

1: total.errors $= 0$
2: dataframe $= \emptyset$
3: **for** snr $=$ snr.min **to** snr.max **step** snr.interval **do**
4: **for** $i = 1$ **to** iterations.per.snr **do**
5: msg $= \{0,1\}^{msg.length}$
6: encoded $=$ *encode*(msg)
7: noisy $=$ *addNoise*(encoded)
8: decoded $=$ *decode*(noisy)
9: total.errors $+=$ *sum*(msg $\oplus$ decoded)
10: **end for**
11: ber $= \frac{\text{total.errors}}{\text{msg.length} * \text{iterations.per.snr}}$
12: dataframe.*add*(snr, ber)
13: **end for**
14: **return** dataframe

Algorithmus 4.3 – Pseudocode der Simulation

Neben der Simulation von Faltungskodes wurde eine Kanalkodierungs-Simulation implementiert. Diese führt die Simulationen aller Verfahren der Kanalkodierung, die im R-Paket implementiert worden sind (Blockkodes, Faltungskodes und Turbo-Kodes), mit den gleichen Parametern aus und fasst die resultierenden Dataframes zusammen. Dadurch können die Verfahren mithilfe einer einzigen Funktion leicht miteinander verglichen werden.

4.8 Visualisierung

Die Visualisierung der Kodierung, Dekodierung und Simulation wurde mithilfe von `RMarkdown` (siehe Kapitel 3.3) implementiert. Die Generierung der Beamer-Dokumente wird mithilfe des Befehls in Listing 4.4 gestartet. Der erste Parameter entspricht der auszuführenden `RMarkdown`-Datei, inklusive Pfadangabe. Der `params`-Parameter kann verwendet werden, um der `RMarkdown`-Datei eine Liste von Parametern zu übergeben. Für die Visualisierung von Diagrammen innerhalb der Beamer-Dokumente ist die Installation des Pakets `ggplot2` notwendig. Informationen zum Paket können aus [2] entnommen werden. Listing 4.5 zeigt die Funktion zum Öffnen des erstellten PDF-Dokuments.

```
1 rmarkdown::render("path/to/file/rmarkdown.rmd", params)
```

Listing 4.4 – Erzeugen des Visualisierungs-Dokuments aus der RMD-Datei

```
1 rstudioapi::viewer("path/to/file/rmarkdown.pdf")
```

Listing 4.5 – Öffnen des Visualisierungs-Dokuments

```
1 title: "Faltungskodierung ohne Punktierung"
2 date: "`r format(Sys.time(), '%d %B, %Y')`"
3 output:
4   beamer_presentation:
5     keep_tex: true
6 params:
7   message: !r c(1,0,1,1,0)
8 header-includes:
9 - \usepackage{tikz}
10 - \usepackage{pgfplots}
11 - \usepackage{mathtools}
12 - \usetikzlibrary{arrows,positioning,calc}
```

Listing 4.6 – RMarkdown Header

Listing 4.6 zeigt einen Ausschnitt einer `RMarkdown`-Datei, in welcher der Header zu sehen ist. Zunächst werden Daten angegeben, die auf der späteren Titelfolie zu sehen sind, wie z.B. Titel und Datum. Im Anschluss werden die gewünschten Ausgabeformate definiert, wobei hier mit dem Beamer-Format nur eine Ausgabedatei erzeugt wird. Zusätzlich wird durch Zeile 5 die TEX-Datei, die von `pandoc` zur Erzeugung des endgültigen Dokuments verwendet wird, abgespeichert. Schließlich werden die Parameter samt Standardwerten definiert, sowie Header inkludiert, die für die Ausführung des LATEX-Codes notwendig sind. Unterhalb des Headers folgt der Markdown-, R- und LATEX-Code der weiteren Folien. [1]

Kapitel 5

R-Paket Schnittstelle

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
`ConvGenerateEncoder(N, M, generators)` Erzeugt einen Faltungskodierer für nichtrekursive Faltungskodes. **Argumente:** `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). **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 (`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 für jeden nichtsystematischen 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 eine 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 gegebenem Punktierungsvektor und Faltungskodierer eine Punktierungsmatrix.

Argumente:
`punctuation.vector` - Punktierungsvektor welcher 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` - Blockkode Parameter. Nicht zu verwenden. Standard: `FALSE`

Rückgabewert:
Verrauschtes Signal.

Funktion 5.9 – ApplyNoise

5.3.2 ChannelcodingSimulation

ChannelcodingSimulation

`ChannelcodingSimulation(msg.length, min.db, max.db, db.interval, iterations.per.db, turbo.decode.iterations, visualize)`
 	
Simulation von Block-, Faltungs- und Turbo-Kodes und Vergleich ihrer Bitfehlerraten bei unterschiedlichen SNR.
 	
Argumente:
`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
`turbo.decode.iterations` - Anzahl der Iterationen bei der Turbo-Dekodierung. Standard: 5
`visualize` - Wenn `TRUE` wird ein PDF-Bericht erstellt. Standard: `FALSE`
 	
Rückgabewert:
Dataframe, das alle Simulationsergebnisse der drei Kodierungsverfahren beinhaltet.

Funktion 5.10 – ChannelcodingSimulation

5.3.3 PlotSimulationData

PlotSimulationData
`PlotSimulationData(...)`

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

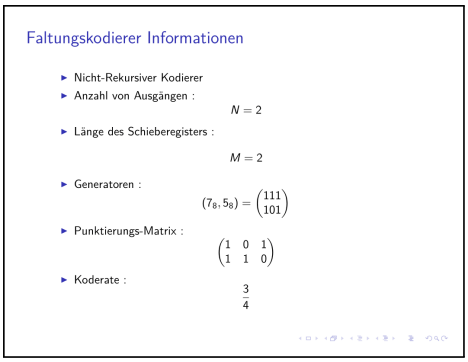
Wird der `visualize`-Parameter bei der Ausführung einer R-Paket-Funktion auf `TRUE` gesetzt, wird ein `RMarkdown`-Skript ausgeführt. Dieses generiert eine Präsentation vom LaTeX-Dokumenttyp *Beamer* mit Informationen und Visualisierungen zur ausgeführten Funktion.
Die erzeugten Präsentationen liegen innerhalb des Programmverzeichnisses von R im Installationsordner des Pakets. Dort gibt es den Ordner *pdf*, in dem sich die Dokumente befinden. Möchte man die Dokumente von RStudio aus öffnen, anstatt die Dokumente in der Verzeichnisstruktur suchen zu müssen, kann die `ConvOpenPDF`-Funktion (siehe Funktion 5.8) 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 LaTeX-Dateien, aus denen die Beamer Präsentationen generiert werden, liegen ebenfalls im selben Verzeichnis.
Da einerseits der Platz auf einer Folie begrenzt ist und andererseits Visualisierungen zu Lehrzwecken nur für kurze Bitsequenzen sinnvoll sind, gibt es Einschränkungen für die in einer Visualisierung verwendeten Nachrichten und Kodierer. Eine Visualisierung der Kodierung bzw. Dekodierung erfolgt für Nachrichten bzw. dekodierte Nachrichten inklusive Terminierungsbits bei einer Länge von bis zu 14 Bits. Bei einem Faltungskodierer mit einer Schieberegisterlänge von vier oder größer erfolgt keine Visualisierung der Dekodierung, da das Trellis zu groß ist. Bei der Kodierung wird nur das Zustandsdiagramm nicht abgebildet.
Bei den folgenden Beispielen der Visualisierungen wurde stets punktiert, da

ohne Punktierung die Informationen zur Punktierung in den Präsentationen fehlen und somit auch nicht erläutert werden können.

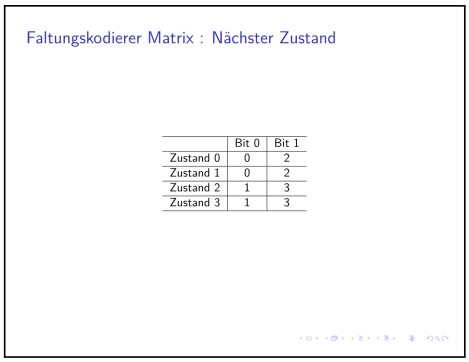
Der weitere Kapitelinhalt setzt sich folgendermaßen zusammen: Die Kapitel 6.1 bzw. 6.2 beinhalten Erläuterungen zur Visualisierung der Kodierung bzw. Dekodierung. Kapitel 6.3 beschreibt die Visualisierung der Simulation.

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 Ausgängen 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. Anschließend folgt die Visualisierung der Kodierung. Dabei wird



(a) Folie mit Kodierer-Kennwerten

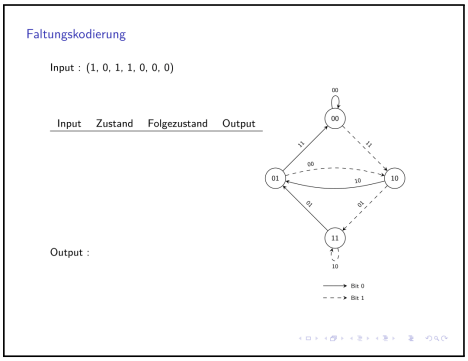


(b) Folie mit Zustandsübergangsmatrix

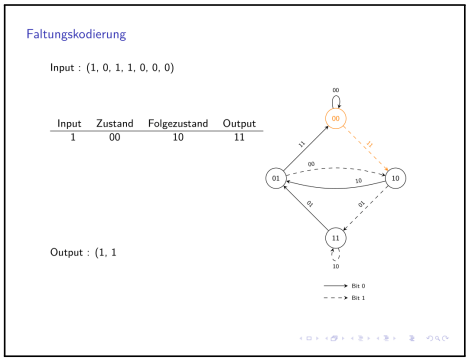
Abbildung 6.1 – Folien mit allgemeinen Informationen zum Faltungskodierer

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 erar-

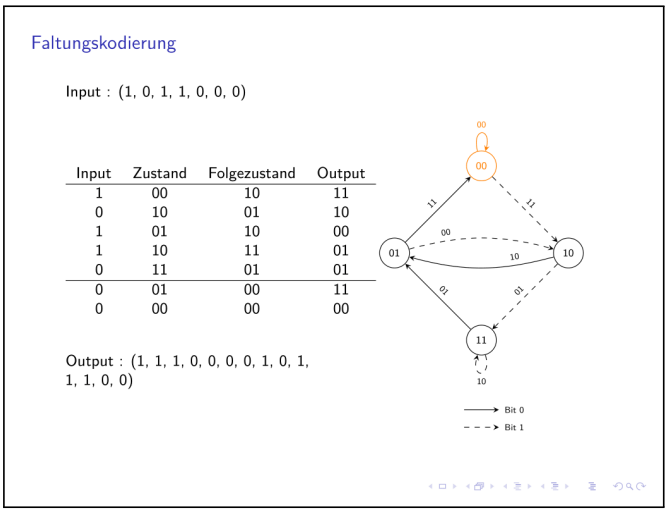
beitet. Durch diese Herangehensweise ist die Kodierung für den Benutzer einfach nachzuvollziehen. Abbildung 6.2b zeigt die Folie der Kodierung des ersten Bits. Das erste Bit des Inputs, der aktuelle Zustand, der Folgezustand sowie der resultierende Output werden in eine neue Zeile der Kodierungstabelle geschrieben. Der aktuelle Zustand sowie der entsprechende Übergang werden im Diagramm farblich hervorgehoben, um die Kodierung auch im Zustandsdiagramm verfolgen zu können. 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 in der Tabelle stellen die Terminierungsbits dar. Da die Kodierungsfunktion



(a)



(b)



(c)

Abbildung 6.2 – Folien der Kodierung

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.

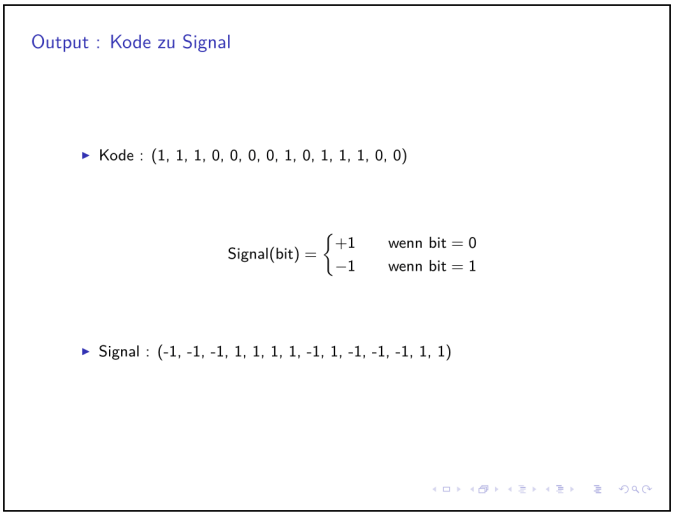


Abbildung 6.3 – Folie mit der Abbildung der Kodebits zu den Signalpegeln

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 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 anschließende tatsächlich punktierte Signal, bei dem die punktierten Werte fehlen, was dem Rückgabewert der Funktion entspricht.

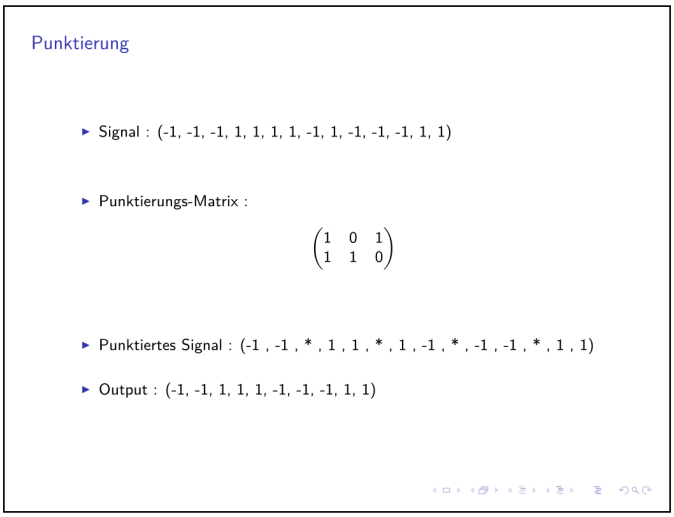


Abbildung 6.4 – Folie mit Punktierung

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 Depunktierung 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 Dekodie-

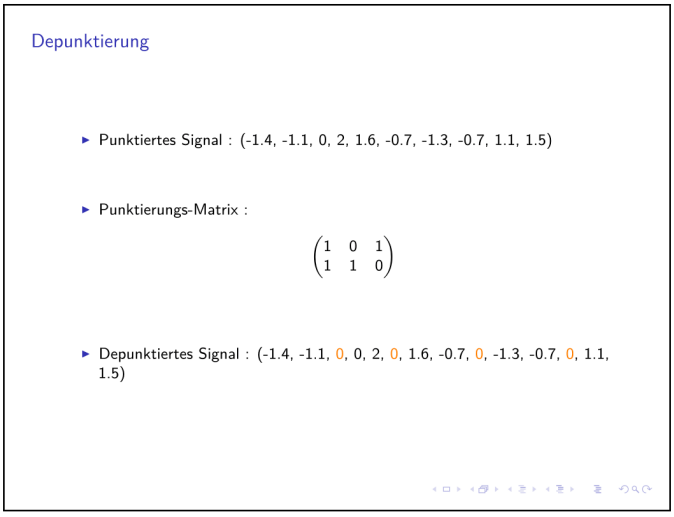


Abbildung 6.5 – Folie mit Depunktierung

rung das Kodewort als kontinuierliche Signalwerte, die möglicherweise durch Anwendung der `ApplyNoise`-Funktion verfälscht worden sind. Die soft decision Dekodierung verwendet zur Dekodierung zwar direkt diese Signalwerte, da aber sowohl die hard decision Dekodierung Bitwerte zur Dekodierung verwendet, als auch 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-

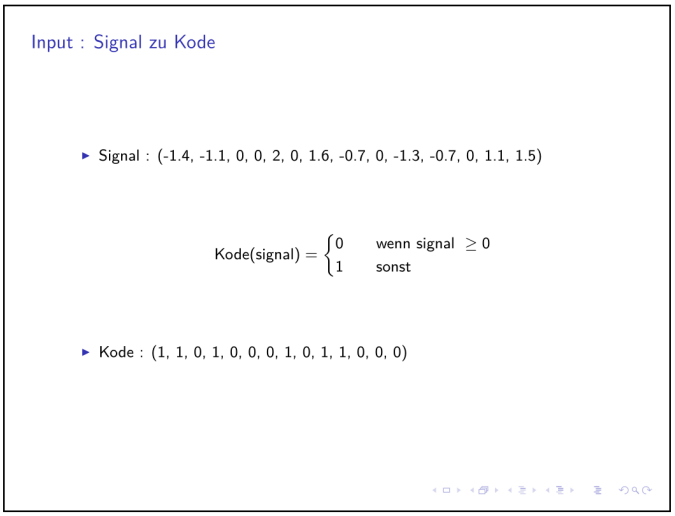


Abbildung 6.6 – Folie mit der Abbildung der Signalpegel zu den Kodebits

Algorithmus mithilfe des 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 6.7 und 6.8 zu sehen. Danach erfolgt Schritt für Schritt die Rekonstruktion der Nachricht mittels Backtracking. Der gewählte Pfad beim Backtracking wird farblich hervorgehoben. Die übrigen Pfade werden ausgegraut. Eine Folie während des Backtrackings wird in Abbildung 6.9 veranschaulicht. Am Ende befindet sich unter dem Trellis-Diagramm die farblich hervorgehobene dekodierte Nachricht, wie in Abbildung 6.10 dargestellt. Die

einzelnen Zwischenschritte vermitteln dem Benutzer wie der Algorithmus funktioniert und wie sich die dekodierte Nachricht ergibt.

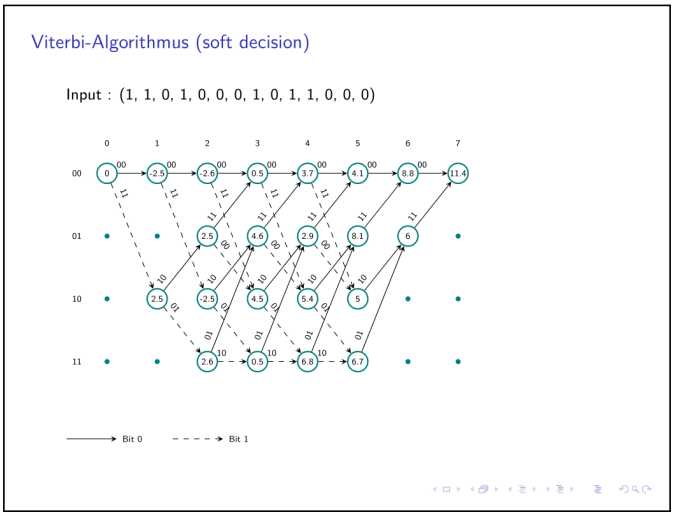


Abbildung 6.7 – Folie der soft decision Dekodierung und dem vollständigen Trellis

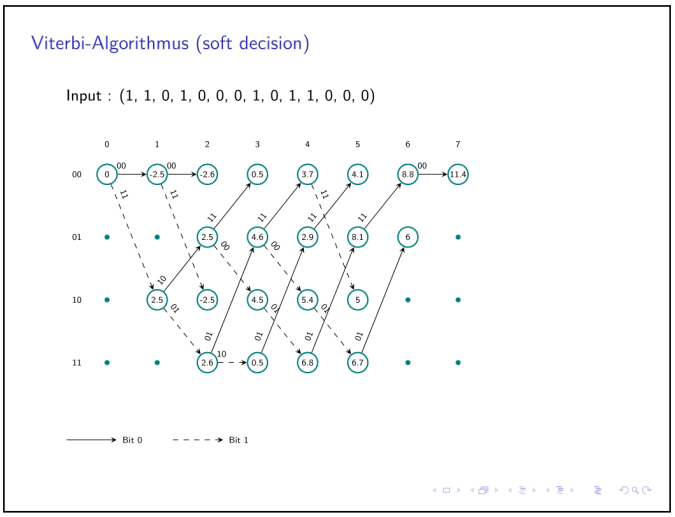


Abbildung 6.8 – Folie der soft decision Dekodierung und dem Trellis nach dem Entfernen einiger Pfade

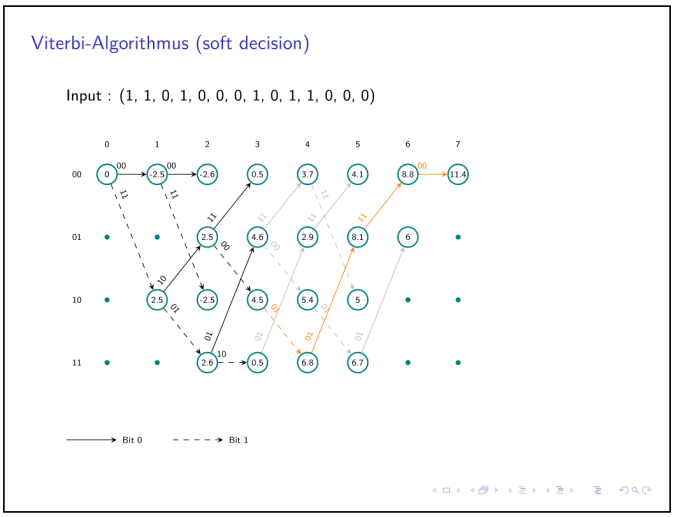


Abbildung 6.9 – Folie der soft decision Dekodierung im während dem Backtracking

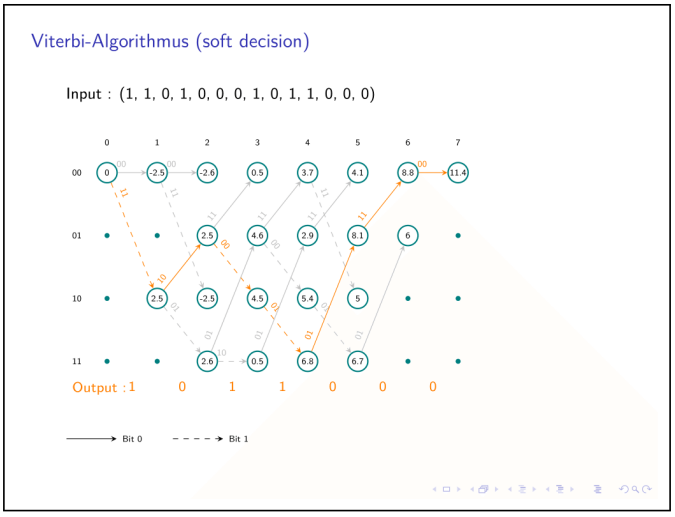


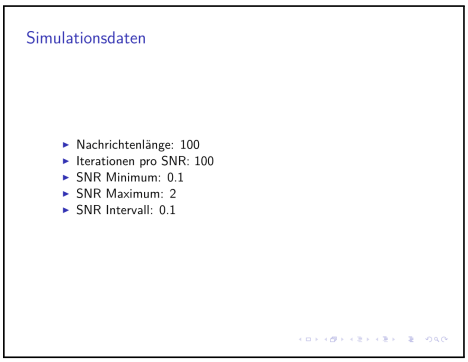
Abbildung 6.10 – Folie der soft decision Dekodierung mit der dekodierten Nachricht

6.3 Simulation

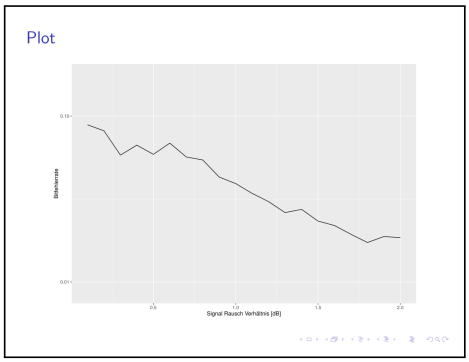
Auch die Simulationsfunktionen bieten durch den `visualize`-Parameter die Möglichkeit eine Visualisierung generieren zu lassen. Kapitel 6.3.1 erläutert die Präsentation der Faltungskode-Simulation. Die Folien der Simulation verschiedener Varianten der Kanalkodierung, um diese vergleichen zu können, sind in Kapitel 6.3.2 beschrieben.

6.3.1 Faltungskodierung

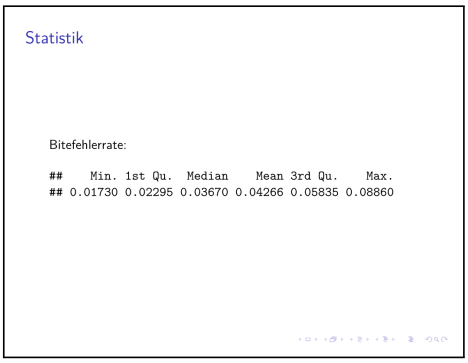
Die folgenden Folien sind das Resultat der Ausführung der `ConvSimulation`-Funktion. Auf den ersten Folien sind, wie schon bei der Kodierung und Dekodierung, Informationen zum verwendeten Faltungskodierer angegeben. Anschließend folgt eine Folie mit den Eckdaten der Simulation, wie in Abbildung 6.11a ersichtlich.
Die nächste Folie, dargestellt in Abbildung 6.11b, stellt ein Diagramm der Daten des erzeugten Dataframes dar. Die x-Achse entspricht dem Signal-Rausch-Verhältnis. Auf der y-Achse werden die gemessenen Bitfehlerraten aufgetragen. Es ist in diesem Beispiel sehr gut zu erkennen, dass die Fehleranzahl mit steigendem Signal-Rausch-Verhältnis abnimmt.
Abschließend werden statistische Kennzahlen der Bitfehlerraten wie Minimum, Maximum, Median etc. auf der letzten Folie, wie in Abbildung 6.11c zu sehen, aufgelistet.



(a) Folie mit Simulationseckdaten



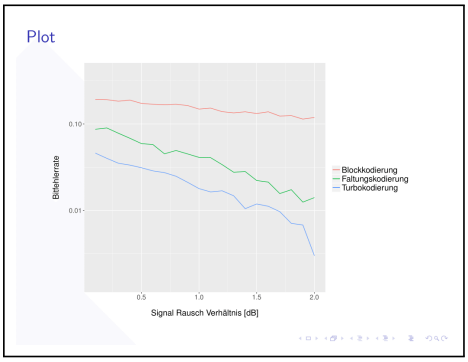
(b) Folie mit Diagramm der Simulationsergebnisse



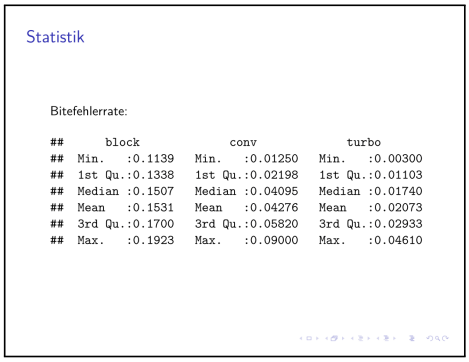
(c) Folie mit Statistik der Bitfehlerraten

Abbildung 6.11 – Folien der Faltungskodierungs-Simulation

6.3.2 Kanalkodierung



(a) Folie mit Diagramm der Ergebnisse



(b) Folie mit Simulationsstatistik

Abbildung 6.12 – Folien der Kanalkodierungs-Simulation

Für einen Vergleich der im R-Paket implementierten Verfahren der Kanalkodierung kann die `ChannelcodingSimulation`-Funktion ausgeführt werden, deren Simulationsergebnisse ebenfalls dargestellt werden können. Zu Beginn der Kanalkodierungs-Visualisierungen werden auf einer Folie die Simulationseckdaten aufgelistet, wie es schon bei der Visualisierung der Faltungskodierungs-Simulation der Fall war.
Abbildung 6.12a zeigt die Folie mit dem Diagramm, in dem die Ergebnisse der drei Simulationen dargestellt werden und miteinander verglichen werden können. Die Art der Kodierung einer Kurve kann anhand der Farbe zugeordnet werden. Wie auch schon in Kapitel 6.3.1 befindet sich auf der letzten Folie eine Statistik der Bitfehlerraten. Dabei werden die Werte der verschiedenen Kanalkodierungs-Varianten gegenübergestellt. Abbildung 6.12b zeigt ein Beispiel zur Statistik-Folie.

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 erfolgen Definitionen von Hilfsvariablen, die für die folgenden Beispiele verwendet werden. Variablen wie der Faltungskodierer oder die Punktierungsmatrix müssen jedoch nicht unbedingt bei der Kodierung oder Dekodierung mit übergeben 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, falls der Benutzer keine Werte bestimmt. 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 [4, 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).

```
input <- c(1,0,1,1,0)

nasa.encoder <- ConvGenerateEncoder(2, 6, c(171,133))

p.matrix <- ConvGetPunctuationMatrix(c(1,0,1,1,0,1),
    nasa.encoder)
p.matrix
     [,1] [,2] [,3]
[1,]    1    1    0
[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

```
encoded <- ConvEncode(input, nasa.encoder)
encoded
[1] -1 -1 -1  1  1  1 -1  1  1 -1  1 -1  1  1  1 -1 -1  1
    -1 -1  1  1

encoded.noisy <- ApplyNoise(encoded, SNR.db = 0.2)
round(encoded.noisy, 2)
[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

decoded.soft <- ConvDecodeSoft(encoded.noisy,
    nasa.encoder)
decoded.soft
$output.soft
[1] -6.67485  6.67485 -6.67485 -6.67485  6.67485

$output.hard
[1] 1 0 1 1 0

decoded.hard <- ConvDecodeHard(encoded.noisy,
    nasa.encoder)
decoded.hard
[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 zusammen mit dem Kodierer der Kodierungsfunktion `ConvEncode` 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 Kodeworts mit dem verrauschten Kodewort. 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 dekodierten Bits 0 bzw. 1 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 einsehbar. 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, dass nicht die gesamte Liste zu verwenden ist, sondern nur das originale oder punktierte Listenelement. 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

```
encoded.punctured <- ConvEncode(input, nasa.encoder,
    punctuation.matrix = p.matrix)
encoded.punctured
$original
[1] -1 -1 -1  1  1  1 -1  1  1 -1  1 -1  1  1  1 -1 -1  1
    -1 -1  1  1

$punctured
[1] -1 -1  1  1 -1  1 -1 -1  1  1 -1  1 -1  1  1

decoded.soft <-
    ConvDecodeSoft(encoded.punctured$punctured,
    nasa.encoder, punctuation.matrix = p.matrix)
decoded.soft
$output.soft
[1] -6  5 -5 -5  5

$output.hard
[1] 1 0 1 1 0

decoded.hard <-
    ConvDecodeHard(encoded.punctured$punctured,
    nasa.encoder, punctuation.matrix = p.matrix)
decoded.hard
[1] 1 0 1 1 0
```

Listing 7.3 – Kodierung und Dekodierung mit Punktierung

Kanalkodierung des R-Pakets (Blockkodes, 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 Bits. 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

```
> df1 <- ConvSimulation(nasa.encoder, msg.length = 2000,
    min.db = 0.01, max.db = 0.1, db.interval = 0.01,
    iterations.per.db = 100)

df1
      db       ber
1   0.01 0.151810
2   0.02 0.140220
3   0.03 0.150020
4   0.04 0.143695
5   0.05 0.142150
6   0.06 0.142900
7   0.07 0.138555
8   0.08 0.137155
9   0.09 0.141615
10  0.10 0.137050
```

Listing 7.4 – Simulation der Faltungskodierung

Signal-Rausch-Verhältnis gebildet. Der Rückgabewert der Simulation ist ein Dataframe, welches 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 Bits 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.

```
ChannelcodingSimulation(msg.length = 2000, min.db = 0.01,
    max.db = 0.1, db.interval = 0.01, iterations.per.db =
    100, turbo.decode.iterations = 3)

      db block.ber conv.ber turbo.ber
1   0.01  0.197045 0.088875  0.051680
2   0.02  0.197095 0.088670  0.051570
3   0.03  0.195555 0.090415  0.051935
4   0.04  0.196410 0.089065  0.052815
5   0.05  0.193475 0.089780  0.052700
6   0.06  0.194410 0.090370  0.050620
7   0.07  0.195250 0.089125  0.049625
8   0.08  0.194870 0.086845  0.050740
9   0.09  0.192465 0.089515  0.049495
10  0.10  0.193470 0.084220  0.048935
```

Listing 7.5 – Kanalkodierungs-Simulation

7.4.3 Vergleich von Simulationen

Im folgenden Beispiel wird die `PlotSimulationData`Funktion präsentiert. 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.

```
catastrophic.encoder <- ConvGenerateEncoder(2,2,c(6,5))
df2 <- ConvSimulation(catastrophic.encoder, msg.length =
    2000, min.db = 0.01, max.db = 0.1, db.interval =
    0.01, iterations.per.db = 100)

df2
      db       ber
1   0.01 0.492040
2   0.02 0.502815
3   0.03 0.490140
4   0.04 0.497805
5   0.05 0.499780
6   0.06 0.489430
7   0.07 0.496505
8   0.08 0.504305
9   0.09 0.493600
10  0.10 0.490645

PlotSimulationData(df1,df2)
```

Listing 7.6 – Vergleich mehrerer Simulationsdaten

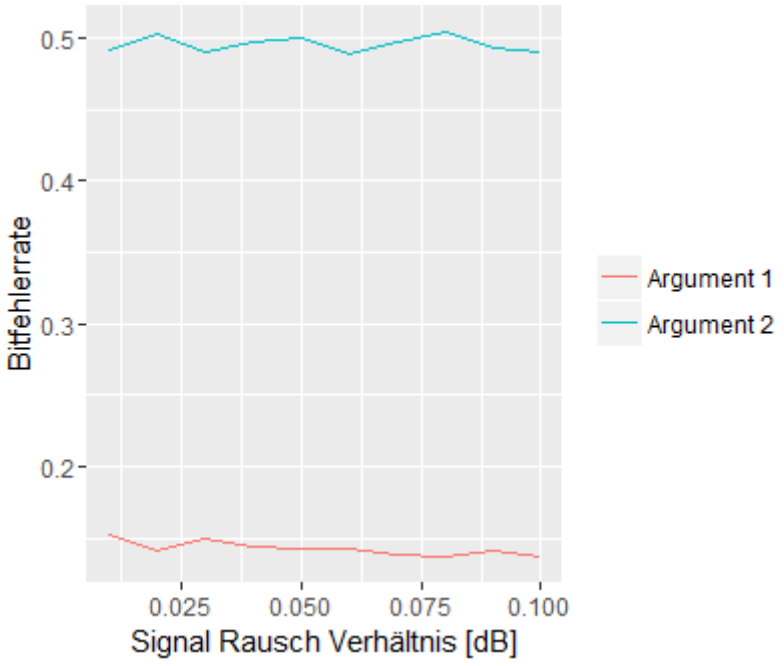


Abbildung 7.1 – Plot zweier Simulations-Dataframes

Kapitel 8

Zusammenfassung und Ausblick

Diese Arbeit beinhaltet die Implementierung von Faltungskodes, einem Verfahren zur Kanalkodierung, innerhalb eines R-Pakets. Außerdem stehen Funktionen für Blockkodes und Turbo-Kodes zur Verfügung, welche aus anderen Bachelorarbeiten stammen. Dadurch ergibt sich ein kompaktes und vielseitig verwendbares R-Paket für die Kanalkodierung. Das entwickelte R-Paket besitzt einen beachtlichen Funktionsumfang, u.a. stehen Funktionen zur Kodierung, Dekodierung und Simulation verschiedener Kanalkodierungen zur Verfügung.

In erster Linie wurde das Paket zu Lehrzwecken für zukünftige Studierende entwickelt. Diese sollen durch die Verwendung des Pakets die Prinzipien der Faltungskodierung bzw. Kanalkodierung verstehen können. Dabei helfen dynamisch generierte Visualisierungen, welche die Kodierung, Dekodierung und Simulation veranschaulichen.

Erweiterungspotenzial besteht vor allem im Bereich der Visualisierung:

- Schaltbild des Faltungskodierers
  Die Darstellung der Zustandsdiagramme der Kodierer ist nur bei einer Schieberegisterlänge von maximal drei möglich. Durch die Darstellung mittels des Kodierer-Schaltbilds könnten auch Kodierer mit längeren Schieberegistern dargestellt werden, ohne unübersichtlich zu werden.
- Berechnung der Metriken der Dekodierung
  Eine Darstellung der Herleitung bzw. Erklärung zur Berechnung der Metriken gäbe den Werten, aus der Perspektive des Benutzers, mehr Bedeutung.

- Berechnung der Soft-Werte der soft decision Dekodierung
  Eine Darstellung, wie sich die Soft-Werte der soft decision Dekodierung errechnen, würde dem Benutzer helfen die resultierenden Soft-Werte interpretieren zu können.

Der Funktionsumfang des Pakets könnte weiters durch die Implementierung des BCJR-Algorithmus erweitert werden. Mit dem BCJR-Algorithmus existiert eine etwas komplexere Alternative zum Viterbi-Algorithmus zur Dekodierung von Faltungskodes. Anstelle der Suche nach dem wahrscheinlichsten Pfad im Trellis wird ein Pfad gesucht, sodass die Fehlerwahrscheinlichkeit der einzelnen Bits minimal ist [6, S. 233 ff.].

Abbildungsverzeichnis

Listingverzeichnis

Algorithmenverzeichnis

Funktionsverzeichnis

Literatur

[1] J.J. Allaire u. a. *rmarkdown: Dynamic Documents for R*. 2016. URL: http://rmarkdown.rstudio.com/ (besucht am 01. 06. 2016).

[2] W. Hadley. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2010. ISBN: 9780387981406. URL: http://ggplot2.org (besucht am 01. 06. 2016).

[3] W. C. Huffman und V. Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2010. ISBN: 9780521131704. URL: http://www.cambridge.org/us/academic/subjects/engineering/communications-and-signal-processing/fundamentals-error-correcting-codes?format=PB (besucht am 01. 06. 2016).

[4] R. H. Morelos-Zaragoza. *The Art of Error Correcting Coding*. John Wiley & Sons, 2006. ISBN: 9780470035696. URL: http://www.the-art-of-ecc.com/ (besucht am 01. 06. 2016).

[5] D. Orlovic. *Turbo code example for MAP and SOVA decoding algorithm*. 2009. URL: https://sites.google.com/site/duleorlovic/solutions/turbo-code-example-for-map-and-sova-decoding-algorithm (besucht am 01. 06. 2016).

[6] D. Schönfeld, H. Klimant und R. Piotraschke. *Informations- und Kodierungstheorie*. Vieweg+Teubner Verlag, 2012. ISBN: 9783834882189. URL: http://www.springer.com/us/book/9783834806475 (besucht am 01. 06. 2016).

[7] M. Viswanathan. *How to generate AWGN noise in Matlab/Octave(without using in-built awgn function)*. 2015. URL: http://www.gaussianwaves.com/gaussianwaves/wp-content/uploads/2015/06/How_to_generate_AWGN_noise.pdf?02772a (besucht am 01. 06. 2016).

[8] H. Wickham. *Advanced R*. CRC Press, 2015. ISBN: 9781498759809. URL: http://adv-r.had.co.nz/ (besucht am 01. 06. 2016).

[9] H. Wickham. *R Packages*. O'Reilly Media, 2015. ISBN: 9781491910597. URL: http://r-pkgs.had.co.nz/ (besucht am 01. 06. 2016).

[10] H. Wickham und W. Chang. *devtools: Tools to Make Developing R Packages Easier*. 2016. URL: https://CRAN.R-project.org/package=devtools (besucht am 01. 06. 2016).

[11] H. Wickham, P. Danenberg und M. Eugster. *roxygen2: In-Source Documentation for R*. 2015. URL: https://CRAN.R-project.org/package=roxygen2 (besucht am 01. 06. 2016).

[12] B. Wimmer. „R-Paket für Kanalkodierung mit Blockkodes". Bachelorarbeit. Universität Innsbruck, 2016.

[13] D. Witsch. „R-Paket für Kanalkodierung mit Turbo-Kodes". Bachelorarbeit. Universität Innsbruck, 2016.